Transparent distributed data management in large scale distributed systems

Soumeya Leila Hernane, Jens Gustedt

Transparent distributed data management in large scale distributed systems*

Soumeya Leila Hernane[a†] Jens Gustedt[b,a]
[a]*ICube, University of Strasbourg, France*
[b]*INRIA, France*

Abstract

In this chapter, we deal with sharing resources transparency in large distributed systems. By using the *Data Handover* (DHO), together with a *peer-to-peer* system we provide an easy-to-use architecture to claim resources in *dynamic* environments: data resources are distributed over a set of *peers* that may appear and disappear. By means of **DHO** functions, users request the mapping of data into local memory (for reading or writing) without prior knowledge neither of the location of that data nor of the underlying structure nor of the mobility of *peers*. This abstraction level is ensured by three managers that interact within our three level architecture.

Two algorithms, *Exclusive Locks with Mobile Processes* (ELMP) and *Read-Write Locks with Mobile Processes* (RW-LMP), are introduced on the lowest level of the architecture. They ensure data access consistency despite the dynamicity of the environment. Both algorithms satisfy *Safety* and *Liveness* properties. Experimental studies show good performance as well as the stability of our approach.

1 Introduction and overview

Mobile computing, fault tolerance, high availability, remote information access are some examples of research fields that have emerged and led to ubiquitous and pervasive computing. In pervasive computing, the connectivity of devices is always ensured and provided by the underlying technologies including internet, middlewares, operating systems, and interfaces Satyanarayanan (2001). Devices should completely be hidden from users. Mostly, they are embedded in the upper software environment.

We are witnessing great needs to request large scale distributed resources (CPU, storage, data...) while providing responses in a reasonable time-frame. These additional needs increase the complexity of the design and confront us with new challenges. The emphasis of this research is on *data resources*; we aim to provide ubiquitous accesses to such resources for resource-intensive applications.

Consistency, *availability* and *fault tolerance* issues are often raised in distributed settings. It is difficult to ensure for shared resources in dynamic large scale environments, especially when resources are not hosted at fixed locations. Trade-offs must be made to ensure at least two properties in highly scalable systems.

†corresponding author.

The famous CAP principle of Fox and Brewer (1999) states that any shared-data system can have at most two of three desirable properties:

Consistency (C) At any time, every node sees the same information

High availability (A) All requests receive a response

Tolerance to network partitions (P) No two subsets of nodes may evolve separately.

Based on the talk of Brewer (2000) at PODC, Gilbert and Lynch (2002) proved this principle and presented the CAP theorem.

Usually, data consistency in distributed systems provides guarantees between the set of items comprising the system. Here, items can be a set of replicas of data on one site and users that use the data on the other. Consistency constraints include for example ordering requirements or global visibility of data existing as multiple copies. Xhafa et al. (2015) proposed a suitable replication system for XML files with fast consistency for *peers* joining last. For databases, usually the ACID properties (Atomicity, Consistency, Isolation, Durability) are considered. They prioritize consistency and partition-tolerance at the cost of potentially lower availability.

Several consistency models for parallel and distributed machines have been discussed in the literature. We mention a few of them: *strict* consistency assumes a shared clock between the processes and ensures that a value that is read is always the value that was written by the most recent write operation. For the *sequential* consistency model, all processes see the same order of all access operations, while for *Fifo* (also known as *PRAM*) consistency, writes are seen by all other processes in the same order in which they were issued.

The *release* consistency model relaxes sequencing requirements and only imposes synchronizations between processes by attributing *acquire* and *release* properties to read and write operations that are issued for the same specific data. Operations on different data then may follow different causal relations and other sequencing. As a consequence, such a system as a whole does not need to implement a common event ordering for all read and write events. It only has to guarantee that the causal relationship between these events is respected transitively for each node and data.

All these models provide *implicit* consistency, that is they do not require additional operations other than reads or writes. They are more suited for small data on shared memory systems. Some variant of these are nowadays implemented as *atomic instructions* for word-sized data on all commonly used CPUs and corresponding operations and consistency models have been added to major programming languages such as C and C++. In contrast to that, these models do not scale well for large data or for large distributed environments.

In the present chapter we will explore a different type of consistency, namely *explicit consistency*, where read and write operations to shared data must be *requested* beforehand, can only be effected if such a request has been *acquired* and must be *released* at the end to the other hosts. Usual lock based strategies such as mutexes or condition variables are well known simple examples for such a consistency model, which in fact combine request and acquire into a single operation, a *lock*.

The resulting consistency model that we will discuss is in fact an *explicit acquire-release* consistency model with *high availability*, but here availability is only ensured if the data had been requested beforehand. We are targeting applications with needs of intensive computing on remote data. We offer two access modes: exclusive, for read-write, or shared, for reading only.

The system as a whole provides no partition tolerance. However, it enables voluntary arrivals and departures of *peers*. For the convenience of users such changes should be *transparent* and *easy-to-use*. Data's access between concurrent *peers* is fair, regulated by a first-come first-serve policy.

We are looking for a *service* and for a whole architecture which ensures simplicity of data access for users at application level, while the *peers* who host the resources are dynamically assigned. We do that by designing an API and an underlying programming model that allows applications to manage computations while transferring large amount of data simultaneously. The target public are users that should just be familiar with the C programming language and with some commonly used tools for parallel or distributed computing. Their only role should be to handle computational tasks by inserting the set of proposed functions for claiming resources in existing applications.

1.1 Main Objectives

More specifically, our objectives are on two levels: system design and user environment, that is the application level. Our challenge is to reach the transparency and the simplicity needed for users at application level, while hiding all complex features of the underlying structure. Regarding the system design level, the aims are summarized as follows:

1. Provide a set of *peers* hosting data resources and an API that can be used in applications written in C or in C++ by inserting function calls.
2. Ensure scalability of the system such that the population of *peers* may vary.
3. Ensure *data consistency* and *data availability* despite the volatility of *peers*.
4. Allow the overlapping of tasks: simultaneously, the system has to deal with data requests while *peers* are joining and exiting.

The user environment that mediates between users and the proposed architecture should satisfy the following properties:

- *Simplicity* and *transparency* of access: users should neither care about the current locations of data resources nor about details of the underlying *peer* structure. The name of the data must be sufficient to retrieve it. The library has to be easy to implement on top of existing applications.
- *Interoperability*: the API has to be as close as possible to known standards and to existing operating systems.
- *Independence* between computation and data transfer: following a data request, an application can continue computations during some time while the data acquisition is processed in the background. In other words, the API should provide *non-blocking* functions.

1.2 Contributions

In order to meet requirements described above, we propose a complete architecture consisting of an API and a grid service for shared resources in large distributed systems. In summary:

- We develop a library interface called Data Handover (DHO) Gustedt (2006a) with a set of functions which can be included in existing applications. It overcomes the short comings of message passing interfaces (see *e.g* mpi-2) and shared memory paradigms. **DHO**, see Gustedt (2006a), combines the simplicity of control of MPI with the random access of memory. It introduces an abstraction level between memory and data through objects we call *handles*. An experimental study has already been given for the *client-server* paradigm, see Hernane et al. (2011).

- We propose a grid service that is modeled by a three-level architecture. It guarantees responses to all data requests of users on the higher level (the third level), through **DHO** routines. With two processes (the Resource manager and the Lock manager) that interact with a data *handle*, the grid service transparently achieves the desired properties related to data acquisition.

- We propose the Exclusive Locks with Mobile Processes (ELMP) algorithm which is an extension of the distributed mutual exclusion algorithm of Naimi and Tréhel (1988). **ELMP** is then used by the *lock manager* on the lower level of the grid service such that we can condition the resource acquisition to the entrance of the **critical section**. **ELMP** ensures consistency for exclusive accesses. Then we extend the capabilities of **ELMP** to allow shared and exclusive locks, resulting in the *Read-Write Locks with Mobile Processes* algorithm **(RW-LMP)**[1].

  Both algorithms exhibit a $O(\log n)$ complexity in terms of messages per request. Proofs of *Safety* and *Liveness* properties are also provided. The potential flooding caused by too many new arriving *peers* can be addressed by known strategies such as Jagadish et al. (2005, 2006); Galperin and Rivest (1993); Andersson (1999).

- We integrate these tools into a three-level *peer-to-peer* architecture. We then introduce the notion of **critical resource** and a *state* concept for each process (*manager*).

- We present performance analysis of our architecture by a variety of benchmarks and experiments that were carried out on a real grid platform.

Through the proposal architecture, this chapter presents an appropriate methodology of sharing remote critical resources between users who are unaware about their localization. We describe all phases through which a given request of a resource passes, from its insertion until its release. However, to ease the description we start from the lowest to the highest level. This is why we first emphasize on distributed mutual exclusion algorithms in Section 2.

After giving the basic definitions of mutual exclusion concept, we describe the Naimi and Tréhel algorithm. Then we outline the problems which may arise in the presence of concurrent requests and the reasons that led us to extend the original algorithm to ELMP we present in Section 3.

Our system is designed to offer two modes of data access. Depending on the nature of the application, the data may be accessed concurrently for reading or exclusively for writing: several *peers* may simultaneously read data without modifying it, but access to modify the data must be restricted to one *peer* at a time.

This is why, in Section 4 we extend the capabilities of **ELMP** to allow both exclusive and shared accesses. We present then the Read-Write Locks with Mobile Processes (RW-LMP) algorithm. Previously, we briefly had introduced this approach in Hernane et al. (2012). Both algorithms exhibit a $O(\log n)$ complexity in terms of messages. Proofs of *Safety* and *Liveness* properties are also provided.

[1] We have briefly introduced this approach in Hernane et al. (2012)

The potential imbalance of the underlying structure is addressed by both algorithms through well known strategies. We suggest choices in Section 3.4 and explain their possible integration in our algorithms.

We then give a modeling of the entire three-level architecture (with **DHO** API) in Section 5 which leads to a *peer-to-peer* system. As a main ingredient, it adds a concept of *states* for all processes comprising the system.

Section 6 gives a performance analysis of our proposed **DHO** *Peer-to-peer* (p2p) system. We discuss a variety of benchmarks and present some experiments carried out on a real grid platform before concluding in section 8.

2 Mutual Exclusion

When concurrent processes share a file or a data resource, it is often necessary to ensure exclusiveness of access to it at a given time. In concurrent programming, a **critical section** is part of a multi-process program that cannot be executed simultaneously by more than one process. Typically, a **critical section** protects a shared data resource that should be updated by exactly one process. In other words, mutual exclusion algorithms are designed to protect one or several sections of the code that are critical. Usually, they are identified by the following requirements:

- At a given time t, at most one process may be inside the **critical section**.
- A process that requests access to a **critical section** should succeed within a finite time.
- For most of mutual exclusion algorithms, the fairness between processes is ensured. They assume that they have the same opportunity to succeed. They do not imply any order of priority between processes that want simultaneously accessing the **critical section**. Many API implement it that way.

Concurrent programming control was first introduced by Dijkstra (1965). This led to the emergence of the discipline of concurrent and distributed algorithms that implement mutual exclusion. They fit into two types of architectures. In shared memory environments, data control is ensured by synchronization mechanisms between processes or threads. In distributed environments, processes communicate by asynchronous message passing. Provided that no process stays forever inside the **critical section**, a mutual exclusion algorithm must ensure the two following properties:

Liveness: A process requesting access to a **critical section** will eventually obtain it.

Safety: At any given time, at most one process is inside the **critical section**.

In other words, liveness prevents starvation of processes, while safety guarantees the integrity of concurrent processes.

2.1 Distributed mutual exclusion algorithms

Several distributed algorithms have been proposed over the years. Depending on the technique that is used, these algorithms have been classified as permission-based (Lamport (1978); Maekawa (1985); Ricart and Agrawala (1981)) and token-based algorithms (Naimi and Tréhel (1988); Raymond (1989)).

For the first, the process only enters a **critical section** after having received the permission from all the other processes. For the second, the entrance into a **critical section** is conditioned by the possession of

a token which is passed between processes. We focus on this class of algorithms for the sake of the message complexity; the distributed algorithm of Naimi and Tréhel (1988) based on path reversal is *the* benchmark for mutual exclusion in this class. It exhibits a $O(\log n)$ complexity in terms of the number of messages per request. We will merely refer to it in the version of Naimi, Tréhel, and Arnold (1996), that additionally provides the proofs for the properties that we will use.

Section 2.2 gives a full description of the algorithm. Many extensions of this algorithm have already been proposed in the literature. We mention a few of them. A fault tolerant token based mutual exclusion algorithm using a dynamic tree was presented by Sopena et al. (2005). Wagner and Mueller (2000) have proposed token based read-write locks for distributed mutual exclusion. Quinson and Vernier (2009) provide a byte range asynchronous locking of the Naimi-Trehel algorithm based on sub-queues when partial locks are requested. Courtois et al. (1971) extend the algorithm to *Readers/Writers* in distributed systems. However, they assume two distinct classes of processes (a reader class and a writer class) where the process cannot switch from one class to another. Lejeune et al. (2013) have presented a new priority based mutual exclusion algorithm for situations where high priority processes are allowed to overtake low priority processes that are waiting for the resource. That approach is complementary to ours, where we need a strict FIFO policy for lock acquisition.

2.2 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.

The basics of the algorithm are summarized following Naimi, Tréhel, and Arnold (1996):

1. There is a logical dynamic tree structure. The *root* of the tree is always the process that requested the token the latest. 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 holds (in insertion order) the requests that have not yet been satisfied. 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 releases its lock.

3. As soon as a process ρ wants to reclaim the lock, 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*, 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 does not detain the lock, it releases the token to ρ. If it still holds the lock or waits for the token to obtain the lock, it points its **Next** to ρ.

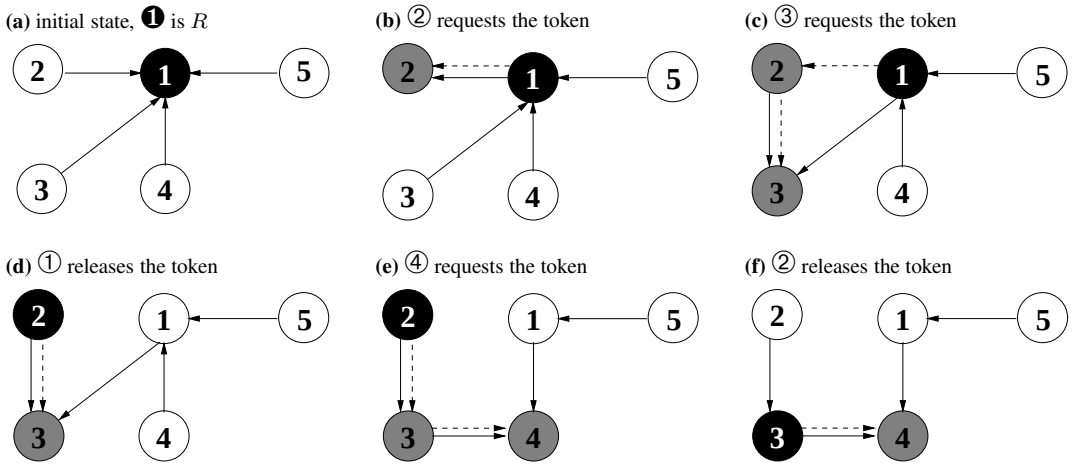
Each process maintains local variables that it updates while the algorithm evolves:

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 lock.

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.

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



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* the proof of which was given in Naimi-Trehel-Arnold Naimi et al. (1996)

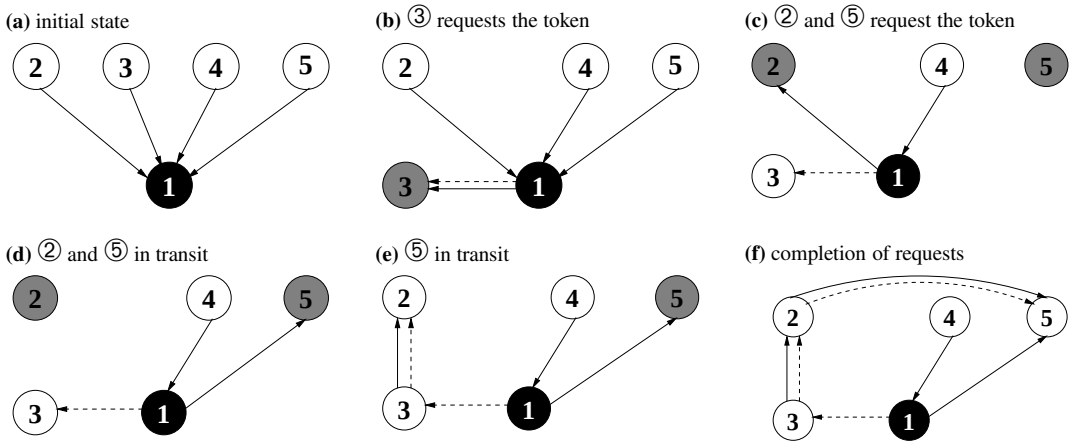
An example of the execution of the algorithm is shown in Figure.1. Gray circles denote processes with requests, while the black circle is one that holds the token. Initially, process ❶ holds the token, Figure.1a. It is the **Parent** of the remaining processes and the *root*, R of the **Parent** tree. Process ② asks the token from its **Parent**, Figure.1b. Thus, ❶ points towards ② and updates its **Next** variable to the same process. Afterwards, ③ requests the token, Figure.1c. ❶ then forwards the request to its new **Parent**, process ② which updates in turn its variables, **Parent** and **Next** to ③. In Figure.1d, ① releases the lock, while ❷ obtains it and then ④ in turn, requests the **critical section**, Figure.1e. 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, Figure.1f.

2.3 Simultaneous requests

To be able to handle simultaneous requests by the same process we made an extension to the original structure.

During the course of the original algorithm, a given **Parent** can be queried simultaneously by different processes, see Figure.2. This example is taken from Naimi and Tréhel (1988) where it is presented in the context of node failures.

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



Initially (Figure.2a), 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, Figure.2b. 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, Figure.2c. 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, Figure.2d. In Figure.2e, request of process ② is achieved and that of ⑤ ends in Figure.2f.

We notice that processes set their **Parent** variable to *null* as soon as they forward the request. Thus, as long as they haven't arrived at the new *root* they disconnect from the tree.

Within a system of n processes, n-1 processes could request the token concurrently. This will generate n disjoint components, the **Parent** relation then is not a tree but only a forest.

To avoid that, we amend the original algorithm by requiring that processes be never cut from the **Parent** tree. This will help us to keep all links of the data structures alive if a given process leaves the system. We assume that a process does not accept a new request before the previous one is completed.

In the following sections, we provide two extensions of the algorithm. For the first, we allow processes to enter or to leave the system. If one of them wants to enter the group, it just has to choose a relative **Parent** and to join the tree and to be connected to the system. The difficulty in node dynamicity lies in the departure of processes, onto which we will thus focus. The second extension provides two ways to enter a **critical section**; either in exclusive mode or shared.

The term of **critical section** that refers to an abstract concept will be used recurrently in our extended algorithms. However, it will refer later to a concrete realization with remote **critical resources**.

3 Algorithm for Exclusive Locks with Mobile Processes (ELMP)

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 requests the token. The connectivity information is only maintained implicitly 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 does not meet our needs for large-scale dynamic systems.

Note that, our final aim is to make accessible critical resources that may be hosted temporary by *peers*, to users that handle that resources within different sections in the code (see section 5). For the time being, we tackle the issues of the dynamicity of processes and of the extensibility of the environment. The **ELMP** algorithm addresses following issues:

1. It maintains the connectivity such that any process is able to leave the group within a reasonable time-frame; reasonable here basically is the time that is needed to forward information to the other processes.

2. It allows new processes to join the system whenever possible.

3. It keeps control of the shape of the tree in order to meet the balancing requirement, such that all operations within the system have a complexity of $O(\log n)$.

For this purpose, we reinforce processes with additional information. First, unlike the original algorithm, initially we assume that processes are arranged in a balanced tree-structure such that all links point towards the direction of the *root* that holds the token. Then, each process σ handles additional variables that we will introduce gradually in the following.

3.1 The data structure

With the aim of maintaining the connectivity of the parental structure as well as of the linked list, we add the following variables to the internal structure of each process σ, see Section 2.2.

Predecessor: σ 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.

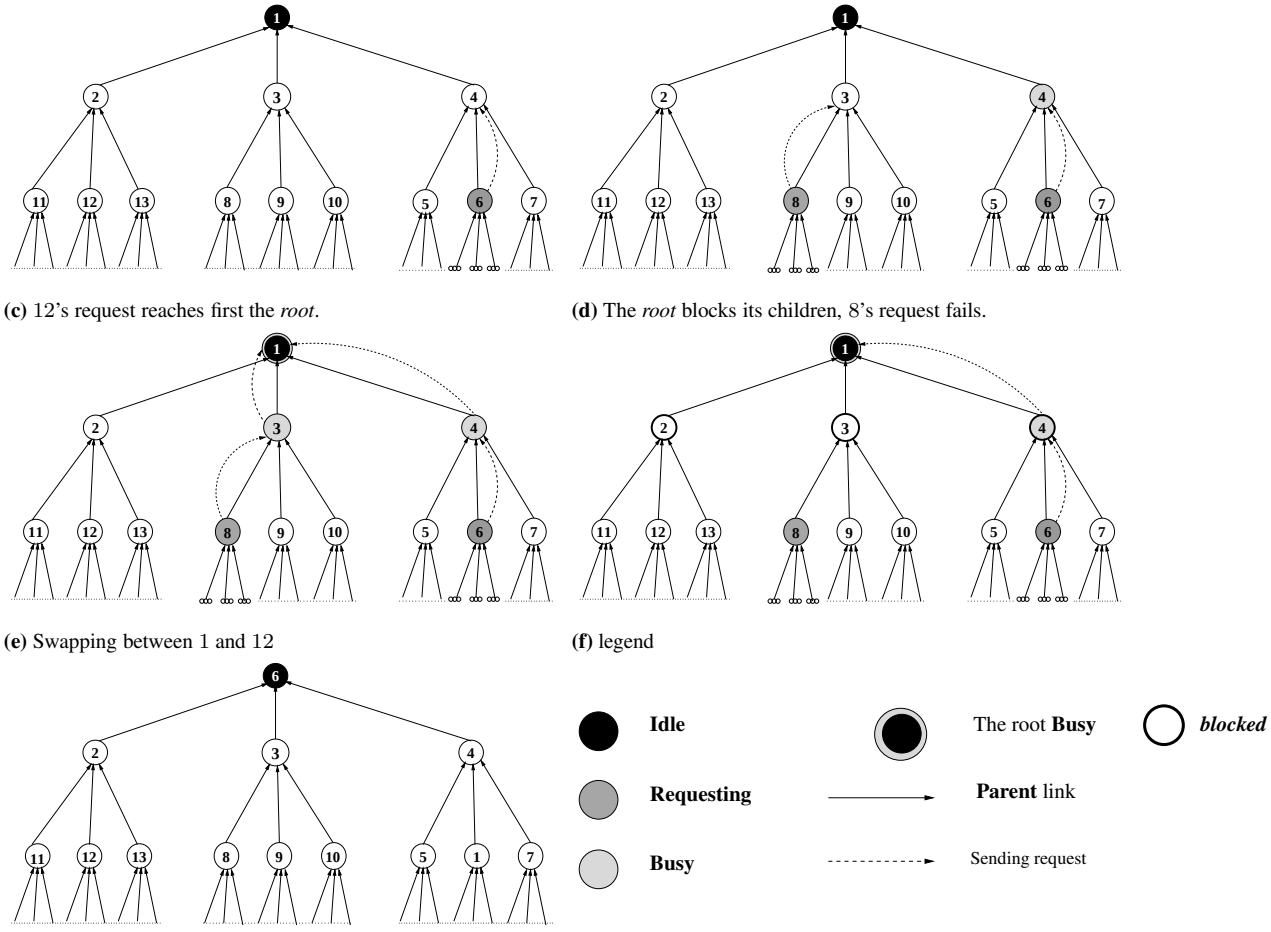
children: The list of processes that are children of σ. Henceforth, σ knows its **Parent** and its **children**.

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

ID We introduce a new variable **ID** that holds a number that will be used as a tie breaker during departure, see Section 3.2. 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* such a value can easily be maintained by that *root* and propagated along if the *root* changes.

The captions of this figure should use the `cerc` macro.

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



3.2 Atomic operations

Our aim is to avoid the overlapping of requests, so any process σ will not handle several requests at once. Before accepting to receive a new request, σ must have completed all previous processing.

The **State** variable we introduce below, manages the atomicity of that approach. It may have different values that indicate the specific task the process is currently completing. Figure.3 exemplifies our approach and emphasizes on the state concept.

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

Requesting: σ has started an insertion into the queue for requesting the token but that request is not yet completed (such as process 12 in Figures .3a and .3b). It is not ready to receive any request as long as it is in this state. Dark gray circles denote the **Requesting** processes. **blocked** list will include all corresponding children (bold circles, see section .3f). 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 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* (Figures .3b and .3c).
   Once the *root* is *Idle*, it in turn includes its children in the **blocked** list (processes 2, 3 and 4 (Figure.3d)).
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 (Figure.3d).
3. σ is the new *root*, process 12 in Figure.3e. It sends acknowledgments to processes $p_2, \ldots, p_{n-1}$ on the path. It changes its **Parent** to `empty` and its **Predecessor** to the old *root*. From here on, σ is queued in the end of the **Predecessor**-**Next** linked list.

Lemma 1. *The* **Next** *and* **Predecessor** *variables form a doubly linked list.*

Proof. It is easy to see that 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. □

Busy: ρ handles an insertion request for another process σ (light gray circles such as processes 4, 3 and the current *root* 1 in Figure.3). It is not ready to forward other insertion requests yet (such as process 3 that rejects 8's request), 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 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 whole 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' cannot be undertaken before σ, the new *root* returns to the *Idle* state. □

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 is disconnecting. In fact, ρ will promote its children to the **Blocked** state, such that they hold back any requests that might be pending in their subtrees, *e.g* processes 12 and 1 in the example (Figure.3). Among these blocked neighbors, σ will choose 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.
- σ is part of an unbalanced branch of the tree. Thus, it will be blocked for a further positioning in the tree. In Section 3.7, 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 cannot break away from its **Parent** nor from its children suddenly. It should rather 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 does not mean, that it cannot 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.

Lemma 4 (departure)**.** *A process σ that want 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 σ 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).

[2] If it has neither **Parent** nor children, the system consists only of 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 than 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 5), 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 from 4 to the 8) of the Exit atomic operation 3.2. □

3.3 Connecting to the system

There is no specific state for processes attempting a new connection. The connecting process is relatively straightforward. However, certain steps should be performed beforehand.

In fact, the placement of σ is closely linked to the adopted balancing strategy (Section 3.4) of the shape of the tree. If σ wants to join, first, it has to know a given process η in the tree. If η is not *Idle* or, if it has less than m authorized children, it may accept σ otherwise, it forwards σ's request to its **Parent** or to one of its children. If this is the case, σ's request is studied again. The process starts over again, probably down the tree, until σ finds an appropriate **Parent**.

From this point, the new connection is gradually announced along the **Parent** branch to the *root*. Once informed of the new connection, the *root* assigns an ID to σ by the same path. At the end of the insertion process, σ becomes *Idle*.

3.4 Balancing strategies

The competitive system in which evolve the algorithms should be as extensible as possible. However, we aim to control the shape of the tree in order to ensure a logarithmic complexity.

Many approaches have been proposed in the literature in order to achieve efficient maintenance for the tree, mainly if they are binary, with the aim to find a balance criteria that ensures a logarithmic height of the tree. We outline two approaches:

The first is to restrict the shape of the tree that should always be of order m. Jagadish et al. (2005, 2006) proposed a balanced tree structure to overlay on a *peer-to-peer* network. It is based on a binary balanced tree (BATON) and generalized to m-order trees (BATON*). They support joining

and departures of nodes and take no more than $O(\log n)$ steps. To ensure a balanced growth of the tree, new nodes are assigned to previously empty leaf positions. For departures, the authors propose replacement of *non-leaf* nodes by *leaf-nodes*.

Interesting for us in this schema is the additional links between siblings and adjacent nodes. They allow to jump in the tree, 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, see Galperin and Rivest (1993).

Andersson (1999) uses the concept of general balanced trees. So, as long as the height of the tree does not not exceed $\alpha.log|T|$ for some constant $\alpha > 1$ where T is the size of the tree, 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(σ)* $> \alpha \cdot \log |$*weight(σ)*$|$, is observed. Thus, a partial rebuild of the sub-tree starting from the scapegoat node is made. Many partial rebuilding techniques can be found in the literature as *e.g* in Galperin and Rivest (1993).

Whatever the policy, we should add the following variables to the previous data structure (section 3.1):

***height(σ)*:** The height of the sub-tree 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 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.

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 proposed algorithms (see Sections 3 and 4).

3.5 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 the BATON structure, see 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 (see **Exiting** and **Blocked** states in section 3.2).

   (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, see 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 can easily be done since *height* and *weight* variables give appropriate information (for σ that is on the top) of the sub-tree. 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 sub-tree is stated as balanced.

3.6 Balancing following a token request

The requesting processing we have presented in Section 3.2 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 tree remains unchanged.

3.7 Balancing following departure

The Exit strategy presented in Section 3.2 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 sub-trees on the top compute again their *height* and *weight* variables and seek again a possible scapegoat process.

3.8 The proof of the ELMP algorithm

We have shown in previous sections how to deal with concurrent requests, where processes are initially arranged in a balanced tree of $order$ m.

Both balancing approaches cited in Section 3.4 provide flexibility and elasticity in the structure, such that the tree can be enlarged by new processes or reduced by the current ones. The following lemma strengthens Lemma 2 when processes trigger the movements other than requests of token.

Lemma 6. *No other request of a process σ' 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. 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 3.2, Lemma 3 and Section 3.7.

For (2), new insertions never compromise the order of requests at the *root*, whatever the number of processes. Indeed, a process that wants to insert itself to the system first checks the availability of processes on its path to the *root* (Section 3.5). Otherwise, σ backtracks. Thus, there is no request in progress which cannot be achieved.

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

Theorem 1 (Liveness)**.** *A process σ that claims a **critical section** obtains it within a finite time.*

Proof. We know that the waiting time for the completion of a request is finite, and that the **Parent** tree as well as the doubly linked list are never disconnected (Lemma 5). Also, **ELMP** guarantees that requests are treated in the same chronological order as their reception at the active *root*. This even holds in case of departures or insertions (Lemma 6). Thus, a **critical section** can be accessed after all previous requests have been handled. Since the handling of each of these is finite, the overall access time is finite. □

Theorem 2 (Safety)**.** *At any given time there is exactly one token in the system.*

In other words, the system guarantees that at most one process is inside a **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*. As the **ELMP** algorithm evolves, the token is passed from one process to another across the linked list **Next**-**Predecessor**. Whence 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 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. □

The **ELMP** algorithm presented in this section implements exclusive sharing of the token between processes distributed over a tree. It features the voluntary departure of these processes. It also enables connecting to new arriving processes. This offers the degree of elasticity that we need for the whole system we propose, see Section 5.

4 Read Write Locks for Mobile Processes (RW-LMP) algorithm

As we have already mentioned, we aim to provide two ways to access the **critical section**, exclusively for writing by a single process or concurrently for reading by many. Therefore we provide an extension to the

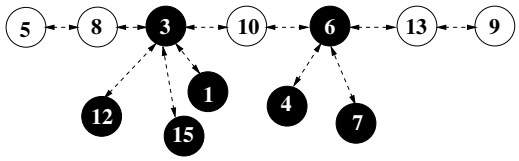


Figure .4: A doubly linked list with two *read managers*

ELMP algorithm such that several processes may share access to a **critical section** without compromising the consistency of the data.

The **RW-LMP** algorithm we propose inherits all properties of **ELMP**'s principal operations that are involved in the **Parent** tree and the balancing strategies. It differs however, in handling the entry to the **critical section**, namely by amending the FIFO data structure. In **ELMP**, the token is simply forwarded from one process σ to its **Next**, and σ just short-cuts between its **Predecessor** to its **Next** in case of departure. To be able to handle (read/shared – write/exclusive) locks we introduce a manager of readers in the **Next**-**Predecessor** structure. First, **Next** and **Predecessor** store the type (r or w) of the next or previous request in addition to the link itself. Then, we also maintain a pair (Read manager, *reader number*), referring to the first process that requests a shared token after an exclusive one, called a *read manager* and a counter of the number of ongoing read accesses that follow this *read manager* in the FIFO. Both are supposed to be properly initialized.

4.1 Handling requests in the linked-list

Several cases may occur when the request for a new process σ is appended to the FIFO:

If σ is a reader and its **Predecessor** is a writer, σ becomes *read manager*. It sets its own *read manager* variable to itself and starts counting the next possible readers. All readers that are inserted after σ in the FIFO will point towards σ which becomes their **Predecessor**, thereby forming a group that ends at the next **Next** with exclusive request, if any. Only σ links to this **Next** to whom it will pass the token (processes 3 and 6 in Figure .4) once the whole group of read accesses is terminated.

If both σ and its **Predecessor** are readers, σ now is the last reader of the group. It obtains the name of the *read manager* from its **Predecessor**. Then, σ informs the *read manager* that it joins the group and the later increments the counter.

If σ's request type is w, σ just links to its **Predecessor** by updating its variables and waits for the token. Likewise, **Next** is updated as soon as a new request is inserted after that of σ.

In Figure.4 white circles denote exclusive requests. ❸ and ❻ handle groups of readers with 4 and 3 processes, respectively.

4.2 Entering the critical section

As soon as a process receives the token from its **Predecessor**, it enters the **critical section** and accomplishes some operations depending on its position in the linked list. In case of an exclusive write access, the process is just removed from the FIFO. Otherwise, it is a *read manager*: it invites the members of its group to enter

the **critical section**. On entry, each of these processes of the reader group sends a message to the *read manager* which in turn increments the *reader number*.

4.3 Leaving the critical section

If the access was exclusive, the process just releases the **critical section** and passes the token to its **Next**. Otherwise, it sends a message to its *read manager*, which decrements the *reader number*. The *read manager* only releases the token to its **Next** (which is a writer) once the *reader number* is zero.

Invariant 2. *The* read manager *does not release the token to the first process requesting an exclusive token following the reader group, until the reader number is zero.*

Lemma 7. *Once the* read manager *enters a **critical section**, the reader group follows.*

Lemma 8. *The **Parent** tree and the doubly linked list are never disconnected.*

Proof. As long as there are no disconnections from the system (**blocked** state and Lemma 2), the statement is obvious.

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 from 4 to 8 of the Exit atomic operation 3.2). Likewise, σ links its **Predecessor** to its **Next** in the doubly linked list. before exiting □

Lemma 9. *A process with an exclusive request never shares a **critical section** with another process.*

Proof. Let σ be a process with an exclusive request. Let σ be followed in the **Predecessor**-**Next**-list of processes by processes $\{p_1..p_{j-1}, p_j, p_{j+1}..p_n\}$ that together form a reader group such that p_1 is the *read manager*. Assume that the reader group is followed by a process p_{n+1} with a pending write request.

First, assume that σ holds an exclusive token. σ never shares the **critical section** with p_1 since it always forwards the token to its **Next** after releasing.

Then, according to Invariant 2, the *read manager* keeps the token while the *reader number* > 0. Therefore, p_1 never forwards the token to its **Next**, p_{n+1}, while at least one reader remains in the **critical section**. □

4.4 Leaving the system

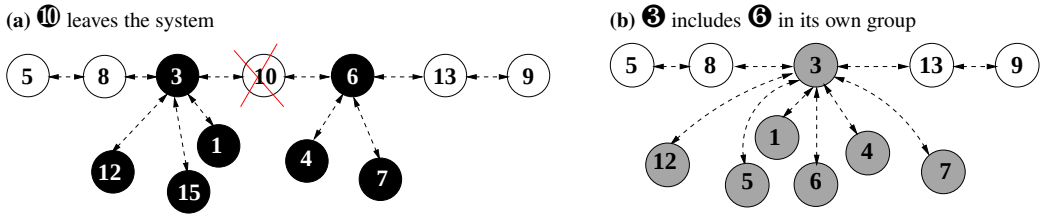
Readers that want to leave the system that are not *read managers* of their group may do so by simply notifying the *read manager*.

Now, another process σ that wants to leave the system while it has already obtained the token (either an exclusive or a shared one), has to ensure that its **Next** is not **Exiting**, itself. Once it has ensured that, it blocks its **Next** until its own **critical section** (and eventually the one of all readers in the group) is finished. It then forwards the token and unblocks **Next**.

If σ does not hold the token while it is leaving, different cases may arise depending on the request type of σ and that of its **Next** and its **Predecessor**. If σ is itself a writer or single reader between two other writers or a writer between a writer and a reader group it links its **Next** to its **Predecessor** and leaves.

If σ is a writer between two reader groups (❿ in Figure .5a), the two groups have to merge (Figure .5b). The second *read manager* (process ❻) will no longer be a read manager. However, it forwards the information about its group to the first *read manager*.

Figure .5: Union of two readers groups following the departure of a writer.



The remaining case is that of a *read manager* that has other readers in its group. σ waits as long as other readers in the same group are **Exiting**. Then, it chooses a member of its group, elects it as new *read manager*, notifies all members about that fact, and updates the FIFO to that new situation.

4.5 The proof of the RW-LMP algorithm

In this section we prove *Safety* and *Liveness* properties.

Theorem 3 (Liveness)**.** *A process σ that claims a **critical section** obtains it within a finite time.*

Proof. We know that the waiting time for the completion of a request is finite, and the **Parent** tree as well as the doubly linked list are never disconnected (Lemma 8). Based on Lemma 4 and Invariant 1, **RW-LMP** guarantees that requests are treated in the same chronological order as their reception at the current *root*, even in cases of departures and mixed read-write requests. Thus, we conclude that a **critical section** is obtained within a finite time. □

Theorem 4 (Safety)**.** *At any time the set of processes inside a **critical section** is either empty, consists of one process that has an exclusive request, or consists only of processes with read request.*

Proof. A process with an exclusive token never shares a **critical section** with a shared one and vice versa. This results directly from Lemma 9. □

At this point of the theoretical study, the **critical section** remains an abstract entity that is accessed consistently by one or many processes for a finite amount of time. *Consistency* and *availability* are guaranteed. In the following, we are going to link the **critical section** notion of the *ELMP* and *RW-LMP* algorithms with concepts of data and resources.

5 Multi-level architecture and data abstraction

The locking mechanism that locks data will inevitably generate an overhead. This is induced by physical characteristics of the runtime environment (available memory, CPU time, resource size).

The memory space allocated for locks and the time that is required for lock acquisition define the lock overhead. For example, fine granularity of data locks with small sizes will increase the relative lock overhead and worsen the performance. Contrariwise, locking resources with large sizes result in costs for acquiring and releasing locks that are negligible compared to the cost of the rest of the computation.

Although the lock time itself might be negligible, transferring large shared resources is not, and delays that are related to the available bandwidth are unavoidable. Our aim is to hide the acquire and release delays between two *peers* in that unavoidable time of data transfer. A full experimental analysis of the proposed architecture will include variation in data size and lock times, Section 6.

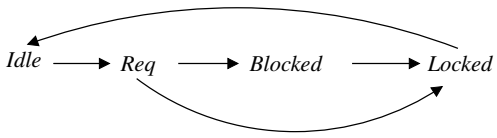
In the following, we assume a set of *peers* distributed over a reliable network which communicate by exchanging messages and such that each *peer* has some autonomy.

The multi-level architecture we present provides and manages concurrent access to remote resources. It ensures consistency and availability of critical resources distributed over *peers* that may appear and disappear. We strongly rely on the cooperation of the different processes that are implied. To achieve these objectives, we use an API called **DHO**. It is implemented with a *peer-to-peer* architecture that includes our mutual exclusion algorithms **ELMP** and **RW-LMP** underneath.

Data Handover, **DHO**, initially described by Gustedt (2006b), is an application API that combines global addressing, read-write locking, mapping and data forwarding. It has the following goals:

- Make remote data available locally for shared reading or for exclusive writing.
- Implement a strict and predictable FIFO policy for the access.
- Allow *peers* that compose the system to join and to leave while the system as a whole continues to handle requests.

Figure .6: DHO life cycle at the *peer* level



5.1 The basic model of the DHO API

With a set of **DHO** functions, applications evolve on the top of a multi-level architecture that is hidden to the user.

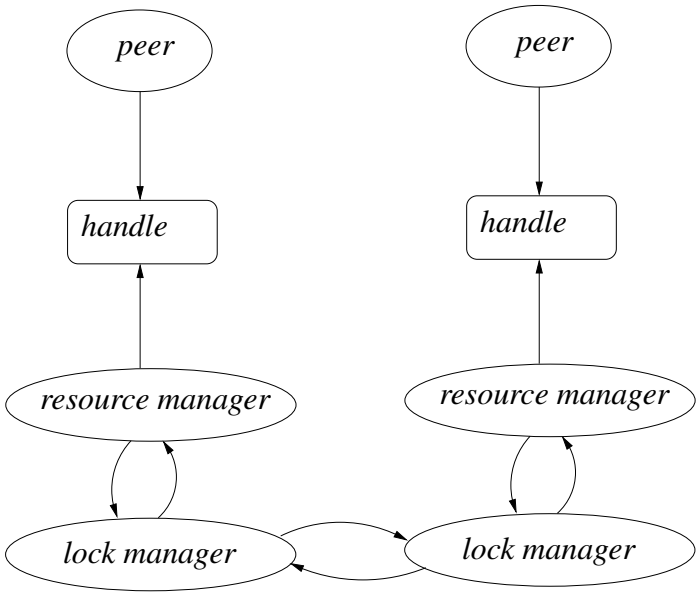
An application process (*peer*) attempts to gain access to a specific data **critical resource** without knowing if that resource is already present locally or on a remote machine. The first operation has always to be the `dho_create` function. It has to wait for a reply from the *peer* that holds the resource.

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

This function initiates a so-called *handle* for data encapsulation. The argument `h`, a *handle*, encapsulates all the necessary information about the remote access to **critical resource**, where the `name` function links the application to that resource. All remaining functions then only use that *handle* to specify the resource. As an example, the function `dho_ew_request(DHO_t* h)` requests the future exclusive-write access to a **critical resource** that is already linked with the *handle* `h`.

Applications using **DHO** routines need at least one *handle* per resource, but may even use several *handles* for the same. Thereby, a process may announce that it will need to access the same resource several times, eventually interleaved with the access by other processes.

Figure .7: Hierarchical order of processes



The `dho_destroy` function unlinks the corresponding data resource from the user application.

Our **DHO** implementation uses two asynchronous processes per data resource and *peer*, the *resource manager* and the *lock manager*. Once a **DHO** request is inserted by the user, the *resource manager*, a local process, takes control of that request and forwards it to the *lock manager*. It is also responsible to map the data in the local address space of the requester, and, to transfer an updated copy of the data to the next *peer* after release.

Figure .6 shows the different states of knowledge about an acquisition that a *peer* has, while Figure .10 gives more details (at a lower level) of the life cycle of a request by the states of the *resource manager*. We will detail these states later (Section 5.2).

Note that the states that we have assigned to the second asynchronous process *lock manager* (see Section 3.2) refer to the completion of inserting a request and not to the acquisition of the data.

To acquire the **critical resource**, the *resource manager* forwards the *locking* request to the *lock manager* which negotiates the *locking* remotely with others *lock managers* through message passing.

As a whole, the *lock managers* of all *peers* ensure the overall consistency of the data by using **ELMP** and **RW-LMP** (Sections 3 and 4) as a locking protocol. The *lock manager* plays the role of σ in these algorithms.

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 mutual exclusion algorithms 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 **critical resource** triggers events at the *resource manager* and crosses various states from request insertion until resource release.

5.1.1 DHO cooperation model

Here, we present the design of a **DHO** architecture to model cooperation between *peers*.

Figure.7 illustrates the cooperation between processes in the same *peer*, the *resource manager* and the *lock manager*, as well as the relationship between *lock managers* of different *peers*. Figure.8 shows more details and outlines the path of requests through different processes inside and outside the same *peer*.

Now, we describe how two different requests for the same resource by two neighboring *peers* pass through our three-level structure (write on the left and read on the right of Figure .8).

- Initially, the application just needs to link to the claimed data named `A` by calling `dho_create` (level ①). The application claims the **critical resource** and the system guarantees its combined *locking* and *mapping* within a finite time.
- Each *peer* runs two managers (*resource manager* and *lock manager*) that interact for acquisition and negotiation phases with each other ④ and with external managers ②
- A negotiation phase consists of requesting, acquiring and forwarding the resource. During such a phase the *resource manager* and the *lock manager* keep information about their current activities by means of assigned states, that are saved in the *handle* by the *peer* and the *resource manager* (Figure .7).
- Inside the same *peer*, the *resource manager* and the *lock manager* cooperate for *locking*/*mapping* the resource locally. The *resource manager* acts as an intermediate owner and is responsible for the *mapping* of the resource into local memory. It forwards the request to the lowest level, namely to the *lock manager* in the same *peer*, ④. In the mean time, the application may continue doing unrelated computations.
- The *lock manager* remotely negotiates the *locking* with other *lock managers* in the network, ⑤, whereas the *resource manager* is involved in transferring the data to its **Next**.
- Some *peers* may further take the additional role of *read manager* if they are the first of a number of successive readers, see Section 4.2.

Once the **critical resource** is linked to the *handle*, the request process may start. Level ① of Figure .8 shows a code prototype of a simple call for a data resource, named A.

The phases of a request that passes through the multi-level architecture are described by the following sections.

5.1.2 The path of a DHO request

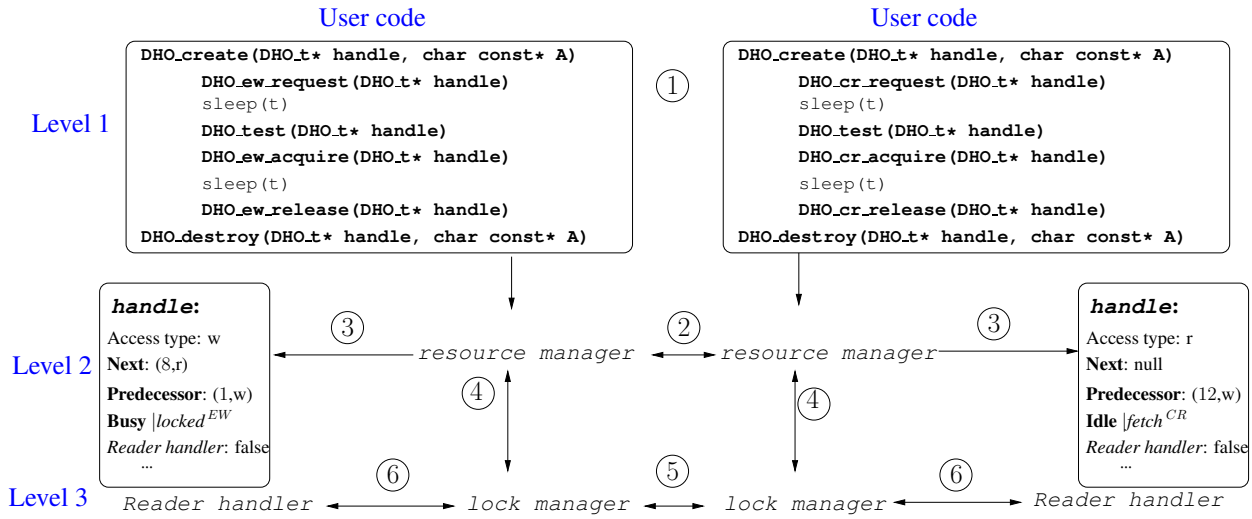
1. The application issues *non-blocking* requests (`dho_ew_request` or `dho_cr_request`) for future acquisition of the data resource. The *peer*'s state becomes *Req* (Figure .6). In the level below, the corresponding *resource manager* switches to the req^{ew} or to the req^{cr} state[3] and forwards the request to the *lock manager* of the same *peer*. The latter becomes **Requesting** . The *lock manager* goes back into the **Idle** state upon completing the request, see section 3.2.

   Thereafter, the *lock manager* expects the token from its **Predecessor**. In the mean time, it is still listening to requests from its children. It may switched to the **Busy** or **Blocked** states.

   The application process itself also may continue some computations regardless whether the resource has already been acquired or not.

[3] These symbols define the type of request: **ew** refers to write exclusive request, while **cr** denotes a concurrent shared read request

Figure .8: Handling requests between two neighboring *peers*.



Now two cases may occur:

(a) At the application level, the *peer* calls the `dho_test` function to know if the *locking* has already been granted. The corresponding *resource manager* asks the *lock manager* if it has already got the token. In that case, the *resource manager* assigns the *grant* $^{\text{ew|cr}}$ state to the *handle*. In the sequel, we will denote the time to achieve this state by $T_{\text{WaitGrant}}$.

(b) Otherwise, the *peer* calls the `dho_ew_acquire`|`dho_cr_acquire` function. It is then put into the *Blocked* state until the data is mapped into its address space. This is done by the *lock manager* that informs its corresponding *resource manager*, ④, which realizes the *mapping*, ②. Likewise, the *resource manager* updates the *handle* to the *blocked* $^{\text{ew|cr}}$ value. The time that the *peer* waits until that is denoted by T_{Wblocked}.

2. After the **Predecessor** has released the resource, it forwards the token to the *lock manager* (which is its **Next**). Once the token is acquired, the *lock manager* immediately informs the *resource manager* that updates the *handle*'s state. It will then enter the *grant* $^{\text{ew|cr}}$ state.

3. At that point, the *resource manager* fetches the data (within a time T_{fetch}) from its **Predecessor** and becomes *fetch* $^{\text{ew|cr}}$. It then *maps* the resource into the address space of the *handle*.

4. Once the *mapping* is done, the *resource manager* and the *peer* become the states *locked* $^{\text{ew|cr}}$ and *Locked*, respectively.

5. *Unlock* is an intermediate state assigned to the *resource manager* during which the *peer* has already released the resource (through the call of the `dho_ew_release`|`dho_cr_release` function), 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.

With this approach, the two managers simultaneously may accomplish different tasks. For example, the combined state **Busy**|*locked* $^{\text{ew}}$ (level 2 of Figure .8) of the *peer* on the left shows that the same *peer* is

Figure .9: *peers* involved in *Data Handover* functions

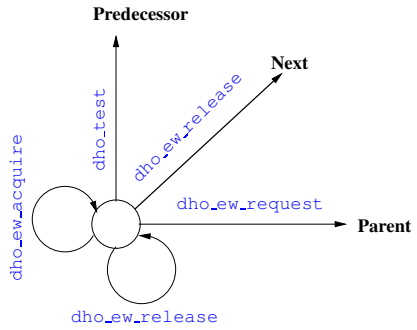
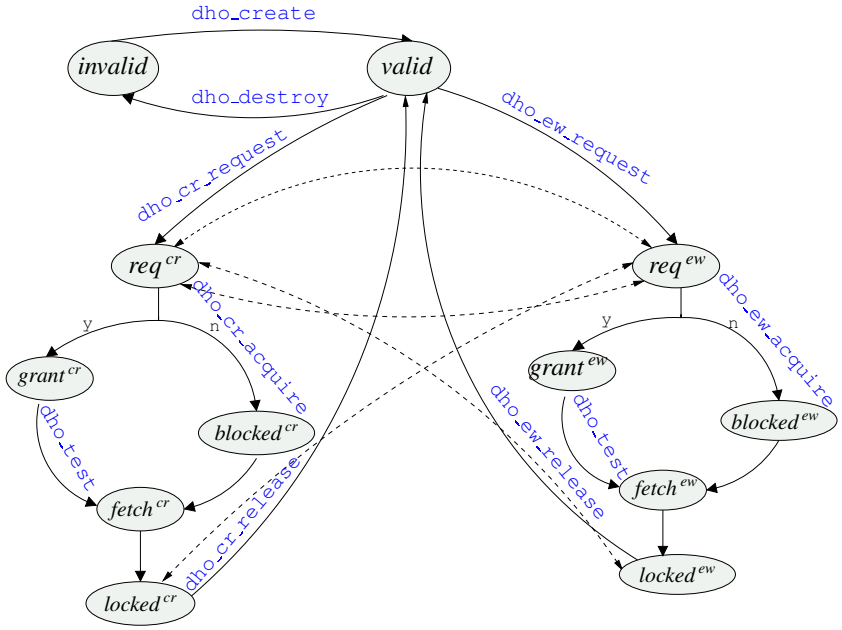


Figure .10: Modeling: life cycle of requests-states of the *resource manager*



handling another request while the application code is inside a **critical section**. The second state *locked*^{ew} is related to the *resource manager*, see Section 5.2.

On the right side, the combined state **Idle**|*fetch*^{cr} means that the corresponding *peer* is currently forwarding the resource to another *peer*. The **Idle** state assigned to the *lock manager* of that *peer* denotes that it has finished sending request.

The *read manager* interacts locally with the *lock manager*⑥. Figure .9 shows that the *peer* deals with three separate *peers* when it claims the resource, achieves the *locking*/*mapping* or grants access.

5.2 Modeling of DHO life cycle

For a given *peer*, a resource request triggers events at the *resource manager* and for the *lock manager*; it crosses various states from request insertion until resource release, forming a typical **DHO** life cycle. Figure .10 shows the different states the *resource manager* can take during the **DHO** cycle. The scheme is composed of two nearly symmetric sub-cycles, the shared cycle on the left and the exclusive one on the

Table .1: List of combined states. For readability, the *Idle* state of the *lock manager* is not represented. *p*: *peer*, *rm*: *resource manager*, *lm*: *lock manager*

p	**Idle**				*Req*						*Blocked*		*Locked*	
rm	*valid*		*Unlock*		*req* ew		*grant* ew		*fetch* ew		*blocked* ew		*locked* ew	
lm	**blocked**	**Busy**	**blocked**	**Busy**	**Requesting**	**Busy**	**blocked**	**Busy**	**blocked**	**Busy**	**blocked**	**Busy**	**blocked**	**Busy**

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

Note that the **DHO** cycle may or may not go through a *blocked* phase. *E.g*, in

$$\{req^{\text{cr}} \rightarrow grant^{\text{cr}} \rightarrow fetch^{\text{cr}} \rightarrow locked^{\text{cr}} \rightarrow valid\}$$

the shared access to the **critical resource** is acquired immediately just after the call of the `dho_cr_acquire` function. Whereas, the following is exclusive with a *blocked* phase.

$$\{req^{\text{ew}} \rightarrow blocked^{\text{ew}} \rightarrow fetch^{\text{ew}} \rightarrow lock^{\text{ew}} \rightarrow valid\}$$

DHO `request` and `test` functions are *non-blocking*. Thereby, the **DHO** API allows an application to continue execution regardless of the state of inserted requests. The application process will block eventually once it calls `acquire`.

5.3 Observed delays

In addition to the theoretical discussion, we aim to assess realistic delays that may be caused by the interaction between *peers*. The user asks for the control of a data resource and waits for a response. Aside delays that are caused by the system, there are also some that care controlled by the user: the **DHO** cycle comprises two application dependent delays, namely T_{locked}, the application time spent inside the **critical section**, and T_{Wblocked}, the time that the application spends waiting before switching to the *blocked* state, *i.e*, the call to one of the *acquire* functions.

We will vary them in our experiments to see the dependency of two other delays: T_{Wait}, the waiting time of request, *i.e*, the time the application is blocked inside a call to an `acquire` function, and T_{DHO}, the entire cycle time of a given request.

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

$$T_{\text{DHO}} = T_{\text{WaitGrant}} \mid T_{\text{Wblocked}} + T_{\text{grant}} \mid T_{\text{blocked}} + T_{\text{fetch}} + T_{\text{locked}} \quad (.2)$$

5.4 Connection and disconnection at application level

We have shown that disconnecting from the system is subject to certain rules that are governed by the *lock manager*. However, the initiating event is explicitly triggered on the application level. It just needs to issue function calls to login or log-off the resource.

The `dho_destroy` function unlinks the **critical resource** from the *handle*. It represents the voluntary departure of the hosting *peer* and can be issued regardless of the currently assigned state. All functions that follow after `dho_destroy` are ignored since the *handle* is *invalid*.

Once the *resource manager* receives a departure request from the corresponding *peer*, it informs the corresponding *lock manager*. The *lock manager* carries out the departure as described for our extended algorithms (Section 3.2). 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 map the data. At the end of the disconnection process the *resource manager* destroys the *handle* by assigning an *invalid* state. Finally, the *peer* enters the *Exit* state for the resource. All these stages may include the tree balancing (3.4)

5.5 Deviation from the normal DHO cycle

In addition to the above, **DHO** foresees all possible combinations of calls to its interfaces and acts accordingly: as a general policy, an application may choose 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.

Listing 1 shows an execution that deviates from the preferred order of execution. The effect of the `dho_ew_request`, *e.g*, is that the *resource manager* is returned to the state of *valid* and that all priorities and a read-lock that eventually already had been acquired are lost. After that, the request is appended to the FIFO and has to wait for its term to regain the front position of the FIFO.

```
1  char const* A;              // the name of the data resource
2  dho_t *h;                   // the handle
3  double T_WBlocked;          // application dependent delays
4  double T_Lock;
5  dho_create(&h, &A);         // creating the handle, linking to A
6     dho_cr_request(h);       // requesting a shared access
7     sleep(T_WBlocked);       // sleeping or doing other computations ...
8     dho_ew_request(h);       // aborting the previous request and
9                              // inserting a new exclusive request
10    request dho_test(h);     // testing if the access has being granted ...
11    dho_cr_release(h);       // ignored ...
12    dho_cr_acquire(&h);      // ignored ...
13    sleep(T_Lock);
14    dho_cr_release(h);       // ignored
15 dho_destroy(&h, &A);        // destroying the handle
```

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

The red dotted lines in Figure .10 illustrate such behavior.

Also, all **DHO** functions that follow the call of `dho_destroy` will be ignored since the **critical resource** is not linked. The `dho_create` function must be called before any other **DHO** function can take effect, it makes the *handle valid*, again. The *peer* is connected to a given **Parent** according to the policy of our algorithms and according to the adopted balanced strategy (Section 3.5). From that point onward, the *resource manager* will be able to handle requests submitted by the user, whilst the *lock manager* will be ready to deal with those coming from **children** in the **Parent** tree.

6 Experimental results

This section encompasses some results for different combinations of requests. We used the *Grid Reality and Simulation environment* (GRAS), see Quinson (2006). GRAS is a socket based API provided by the SimGrid toolkit, see Casanova et al. (2008) and that allows to implement distributed programs. With GRAS, we can either simulate executions or deploy them on real platforms without even modifying or recompiling the code. We just have to re-link the program with the corresponding version of the support library. Under simulation mode, we exploited a description of a realistic platform, which is a subset of Grid'5000.[4] SimGrid provides XML tags for the definition of homogeneous clusters. Here is the description format of selected nodes belonging two clusters used in our framework:

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
   <cluster id="suno"
      prefix="suno-" radical="0-25" suffix=".sophia.grid5000.fr"
      power="1Gf" bw="125MBps"  lat="50us"
      bb_bw="2.25GBps" bb_lat="500us"/>
   <cluster id="griffon"
      prefix="griffon-" radical="0-25" suffix=".nancy.grid5000.fr"
      power="1Gf" bw="125MBps"  lat="50us"
      bb_bw="2.25GBps" bb_lat="500us"/>
</platform>
```

The above XML file reflects real physical features of the set of nodes that are interconnected through private links. Here, this selects 50 nodes as a whole, 25 in each cluster. We use such a platform description to launch benchmarks in simulation mode of GRAS before carrying them out directly on 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* spends waiting until the call of `dho_ew_acquire`, while T_{Locked} is the *locking* time, that is the time the application spends inside the **critical section**.

T_{Idle} denotes the delay between two calls. It represents computational period of the application before inserting a new *Data Handover*. In addition to 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 request call and the return from fetching into the *locked* state.

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 5.3 above for the different times:

$$T_{\text{Wait}} = T_{\text{WaitGrant}} \mid T_{\text{Wblocked}} + T_{\text{grant}} \mid T_{\text{blocked}} + T_{\text{fetch}} \quad (.3)$$

$$T_{\text{Blocking}} = T_{\text{blocked}} + T_{\text{fetch}} \quad (.4)$$

$$T_{\text{Dho}} = T_{\text{Idle}} + T_{\text{Wait}} + T_{\text{Locked}} \quad (.5)$$

We assume that T_{Idle} is zero, so the *peers* insert a new request as soon as the previous cycle is achieved.

[4] Grid'5000 is a large-scale and versatile testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing including Cloud, HPC and Big Data.

peers carry out 100 cycles. Results refer to average values.

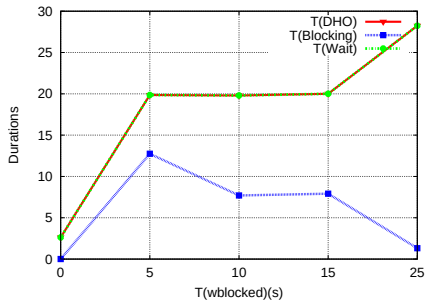
6.1 DHO cycle evaluation with asynchronous locks

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

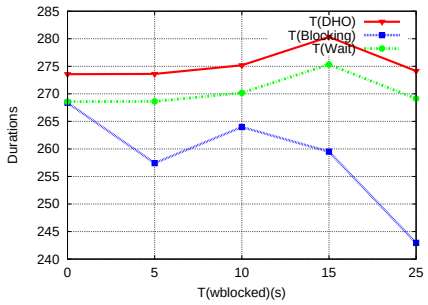
Here, after an application dependent time T_{Wblocked}, the `dho_test` function returns to the state of the *handle*. If *grant*$^{\text{ew}}$ then `dho_ew_acquire` just acts as an intermediate phase for the *fetch*$^{\text{ew}}$ state before then switching to that of *locked*$^{\text{ew}}$ (Listing 2).

Figure .11: Average duration of $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ and $\overline{T_{\text{Blocked}}}$ by varying T_{Wblocked}.

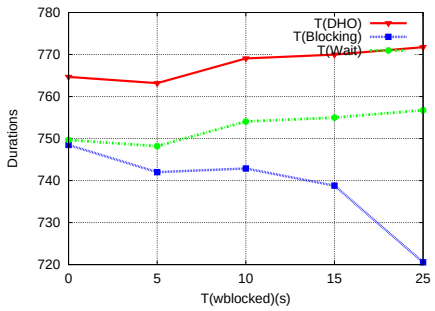
(a) $0s$ in **critical section**



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



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




```
1  char const* A;
2  int i;
3  dho_t h;
4  double DELAY, T_WBlocked, T_Lock;
5  ...
6  dho_create(&h, A);      // link to the critical resource named by "A"
7                          // and create the handle
8  do {
9     dho_ew_request(&h); // request the data for writing
10    sleep(T_WBlocked);   // wait a while or do some computations
```

```
11      dho_test(&h);          // check if the lock has been granted
12      dho_cr_acquire(&h);    // block and map the data in local memory
13      ...
14      sleep(T_Lock);         // keep the lock a while, for some modifications
15      dho_cr_release(&h);    // release the lock
16  } while (i<100)
17  ...
18  dho_destroy(&h);           // destroy the handle
```

Listing 2: Benchmark of exclusive locks

This series of benchmark is conducted with 50 *peers*. The data resource size is fixed to $50MiB$. Figure.11 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 $T_{\text{Wblocked}} = 0$, (Figure.11a), *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}, (Figures .11b and .11c), 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 Figures .11b and .11c, we can conclude that if $5s$ is taken for T_{Wblocked}, a good overlapping is provided for the application, specially between computation and data transferring.

6.2 Shared and exclusive requests

Now we aim to measure the **DHO** cycle with exclusive and shared requests, so in this scenario, applications claim the resource for reading and for writing in a different order.

First, we fix $T_{\text{Wblocked}} = 0$ such that the handle switches to the *blocking* state as soon as requests are issued. In total, the *peers* perform 200 cycles. We vary T_{locked} in both cycles, such that for each value that is used in the first cycle, four other values are provided in the second.

Figures .12a (write locks) and .12b (read locks) show similar results when T_{locked} is varying. Both times are growing, a bit less for reads. This is as expected, because here many requests can be simultaneously inside the same critical section.

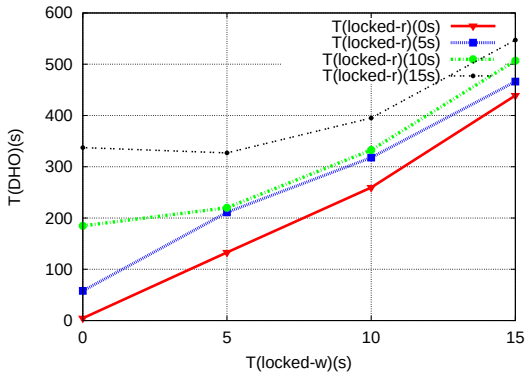
Finally, we measure the behavior of the **DHO** approach with asynchronous locks. Therefore, we impose a certain delay before the call of the `acquire` function. We set T_{locked} to $10s$ and we vary the T_{Wblocked} value.

Given the different values of T_{DHO}, we observe a slight growth in the cycle duration, see Figure .13. The values are largest when the life cycle of shared request is blocked, and thus T_{Wblocked} adds up to the time. For example, with the same value of $10s$ for T_{Wblocked}, T_{DHO} is approximately $423s$ in write cycle, while it is $457s$ in the shared mode.

In order to explain these results, recall that the *read manager* keeps the token while at least one reader in the group remains in the critical section. So, a large group of readers delays the next writer for a time that corresponds to T_{Wblocked} from the first to the last reader. The *read manager* keeps the token more time than a simple *peer* with an exclusive access. Moreover, we observe a slight decrease of T_{DHO} with a delay of $5s$

Figure .12: Average durations for $\overline{T_{\text{DHO}}}$. For each value of T_{locked} in the first cycle cycle, several other values are taken for the second cycle. The resource size is 50 `MiB`.

(a) Variation of T_{locked} in write cycles



(b) Variation of T_{locked} in read cycles

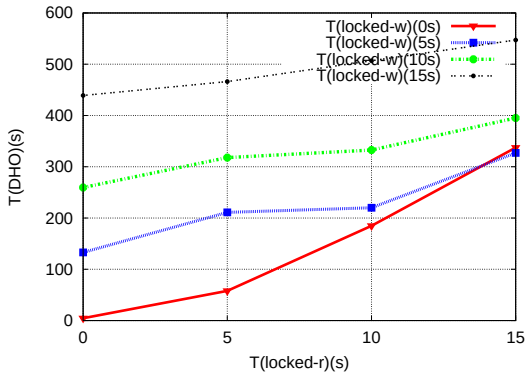
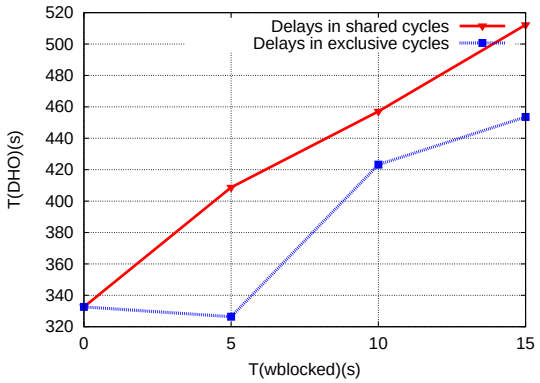


Figure .13: Impact of the T_{Wblocked} delay, in both cycles. T_{locked} is set to $10s$



(T_{Wblocked}) before the call `dho_ew_acquire` function. Thus, a slight delay at application level provides a good overlap between computation and resource control.

6.3 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 **ELMP** algorithm.

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

- *peers* in the first subset perform a complete cycle.
- In the second one, the *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*.

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

7 Discussion

The set of the **DHO** routines and the multi-level architecture provided in this chapter targets distributed parallel applications that need remote data resources for their computations. We have seen that through that API the user takes control of remote data, simply by inserting some **DHO** functions in existing code. Developers who are familiar with the MPI API will not have many problems to use **DHO** routines: **DHO** functions take only one or two arguments and the user has not to worry about technical details to access remote data resources.

Furthermore, access to data resources is requested by non-blocking functions. A set of *managers* handles asynchronous operations to allow the applications to continue doing computations after such a request has been registered. All negotiations for the data resource are transparently handled in the background during such a *non-blocking* phase, and the application can continue independently of the request. The user may also check through the `test` function whether the access to the data has been granted in such a *non-blocking* section of the code. All of this allows computations to overlap with the internal processing of request, which provides good performance for the total execution time.

Once the application process decides that it definitively needs the data resource it may acquire it by issuing a blocking function call. On return from that, the process has been granted the requested lock (exclusive-write or concurrent-read) and the data is mapped into its address space. There it is accessible through a `void*` or `void const*` pointer as long as the lock is held.

From a more technical point of view, our distributed system involves various processes that operate concurrently such that bottlenecks are avoided. The involved processes are mainly those from the same *peer* and from its "neighbors", that is other processes with which the *peer* interacted previously. For example, the mapping phase involves *resource managers* related to the *peer* and its **Predecessor**, that is the process that held the data resource previously and that is now transferring it to the *peer*. In the mean time, associated *lock managers* continue to deal with the arrival of new requests.

If the data resource is large, the bottleneck of a **DHO** system is necessarily the *fetch*$^{\text{ew}}$ operation that consists in transferring the data from one *peer* to the other. Obviously, such a phase is constrained by the overall bandwidth that is available to the application. The asynchronous operation of **DHO** warrants that this can remain the only constraint: latency constraints can be circumvented by overlapping data resource requests with computation.

8 Conclusion

In this chapter, we have presented a shared data system with two desirable properties from CAP: consistency and high availability. It also takes the need for transparency into account.

The proposed approach eases the development of resource-intensive applications that evolve in large scale environments. The proposed design model is centered around a locking mechanism and data encapsulation. This is achieved by restricting the access to data through a *handle*, and by forcing data consistency across critical sections.

We proposed mutual exclusion algorithms based on a hierarchical tree structure. They are used to guarantee the consistency of accesses. They have been proven theoretically and experimentally to be scalable and flexible. *Safety* and *Liveness* properties of both algorithms have been demonstrated. We also studied two methods for keeping the tree structure balanced. This is necessary to keep the overall message complexity low, after a set of conversions of the tree structure have been triggered.

In a series of experiments we measured waiting times and the life time of requests. The experiments have shown that our system guarantees good performance.

We think that in the future a number of open issues should be explored. For example, it will be interesting to extend the capabilities of the system. According to Brewer (2012), consistency and availability should not necessarily be sacrificed when new partitions exist in the network. We may thus consider to tolerate some partitions with a certain degree of consistency and/or of availability.

It will also be interesting to extend **ELMP** to manage exclusive and inclusive accesses to *byte ranges* as *e.g* for POSIX file locks. We will investigate the locking of ranges of the resource by different handles.

List of Acronyms

ELMP Exclusive Locks with Mobile Processes

RW-LMP Read-Write Locks with Mobile Processes

DHO Data Handover

p2p *Peer-to-peer*

Glossary

DHO

The Data Handover proposed interface is based on the abstract concept of the data and the local memory. By inserting the DHO function in the application code, the user claims a remote resource and then when available, maps that resource in local memory.

ELMP

The Exclusive Locks with Mobile Processes algorithm extends the capabilities of the Naimi-Trehel algorithm with the scalability property

GRAS

The Grid Reality and Simulation API is a Socket based library that provides a complete API to implement distributed applications on top of heterogeneous plateforms, either in simulation mode or in realistic environment.

Lock manager

The lock manager negotiates remotely the lock with other managers according ELMP and RW-LMP algorithms. It interacts locally with the resource manager.

Read manager

It handles the entry and the exit from the critical section by successive read processes. It is present in the RW-LMP algorithm solely.

Resource manager

It forwards user's requests to the lock manager, achieves the mapping of the resource in local memory as soon as the lock is granted, and allows uploading the resource to a possible Next.

RW-LMP

The Read Write Locks with Mobile Processes algorithm. This algorithm makes a second extension to the ELMP algorithm, by allowing both read and write requests to share the queuing (**Predecessor**, **Next**) list.

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.

E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.37.

E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: 10.1145/343477.343502. URL http://doi.acm.org/10.1145/343477.343502.

H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experiments. In *10th IEEE International Conference on Computer Modeling and Simulation - EUROSIM / UKSIM 2008*, Cambrige, UK, 2008. IEEE. URL http://hal.inria.fr/inria-00260697/en/.

P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, Oct. 1971. ISSN 0001-0782. doi: 10.1145/362759.362813. URL http://doi.acm.org/10.1145/362759.362813.

E. W. Dijkstra. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569, 1965. URL http://doi.acm.org/10.1145/365559.365617.

A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *IN HOTOS-VII*. Society Press, 1999.

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.

S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

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, 2006a. URL http://drops.dagstuhl.de/opus/volltexte/2006/297.

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, 2006b. 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.

J. Lejeune, L. Arantes, J. Sopena, and P. Sens. A prioritized distributed mutual exclusion algorithm balancing priority inversions and response time. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 290–299. IEEE Computer Society, 2013. ISBN 978-0-7695-5117-3. doi: 10.1109/ICPP.2013.38. URL http://dx.doi.org/10.1109/ICPP.2013.38.

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

mpi-2. Mpi-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

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.

M. Quinson. GRAS: a Research and Development framework for Grid services. Rapport de recherche RR-5789, INRIA, 2006. URL http://hal.inria.fr/inria-00070232.

M. Quinson and F. Vernier. Byte-range asynchronous locking in distributed settings. In *17th Euromicro International Conference on Parallel, Distributed and network-based Processing - PDP 2009*, Weimar, Germany, 2009. URL http://hal.inria.fr/inria-00338189/en/.

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.

M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Commun.*, 8(4):10–17, 2001. doi: 10.1109/98.943998. URL http://dx.doi.org/10.1109/98.943998.

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.

C. Wagner and F. Mueller. Token-based read/write-locks for distributed mutual exclusion. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 1185–1195, London, UK, 2000. Springer-Verlag. ISBN 3-540-67956-1.

F. Xhafa, A.-D. Potlog, E. Spaho, F. Pop, V. Cristea, and L. Barolli. Evaluation of intra-group optimistic data replication in p2p groupware systems. *Concurr. Comput. : Pract. Exper.*, 27(4):870–881, Mar. 2015. ISSN 1532-0626. doi: 10.1002/cpe.2836. URL http://dx.doi.org/10.1002/cpe.2836.

Index

T_{DHO}, 26, 29, 30
T_{Wait}, 26, 29
T_{Wblocked}, 29, 30
T_{locked}, 26, 29, 30
T_{DHO}, 30, 31
T_{Wblocked}, 30
fetch$^{\text{cr}}$, 24, 25
grant$^{\text{cr}}$, 25
locked$^{\text{cr}}$, 25
req$^{\text{cr}}$, 25
blocked$^{\text{ew}}$, 24
fetch$^{\text{ew}}$, 24, 26, 29
grant$^{\text{ew}}$, 24, 26
locked$^{\text{ew}}$, 24, 26, 29
req$^{\text{ew}}$, 26

Application dependent delays, 26
Atomic operations, 9, 15, 19

Balancing strategies, 14–16
Blocked, 9, 12–14
Busy, 11, 12, 24

Critical resource, 21, 22, 26
Critical section, 17, 19, 20, 26, 28

Data encapsulation, 3, 21, 33
DHO, 3, 5, 21, 22, 25–29, 32
DHO life cycle, 22, 25, 27, 30

ELMP, 8, 10, 14, 16–18, 21, 33
Exiting, 12, 13, 16, 19, 20, 26

Handle, 4, 21, 23, 24, 26, 27, 29, 31, 33

Idle, 11–14, 16, 24
Invalid, 12, 26, 27, 31

Linked list, 9, 11, 12, 14, 17–20
Liveness, 4, 7, 20, 33
Lock manager, 22–24, 26, 27, 31

Mapping, 23–25
Mobility, 3, 8, 9, 14, 15, 21
Multi-level architecture, 21, 23, 32
Mutual exclusion, 4–6, 21

non-blocking functions, 3, 26, 29, 32

Read manager, 23, 30
Requesting, 11, 24
Resource manager, 22, 27
Root of the tree, 13, 17, 20
RW-LMP, 4, 17, 20–22

Safety, 4, 7, 20, 33
State concept, 11, 16

TWblocked, 26

Unlock, 24

Valid, 24, 27