

A Framework to Extend Distributed Mutual Exclusion Algorithms to Support Multiple Requests Per-Node

Enrique Guadalupe¹, Federico Meza², Jorge Pérez², and José Piquer³

¹ Depto. de Ciencia de la Computación, Pontificia Universidad Católica de Chile,
Santiago - CHILE

`eguadalu@puc.cl`

² Depto. de Ingeniería de Sistemas, Universidad de Talca, Curicó - CHILE
{`fmeza, jperez`}@utalca.cl

³ Depto. de Ciencias de la Computación, Universidad de Chile, Santiago - CHILE
`jpiquer@dcc.uchile.cl`

Abstract. The problem of defining distributed mutual exclusion algorithms to support multiple requests per-node has been solved by extending currently existing distributed mutual exclusion algorithms. In this paper, we present a generic framework to solve this problem as an extension of a distributed mutual exclusion algorithm and a local mutual exclusion module. These two components are integrated through an extension *key* and the definition of clear communication interfaces for the *key*. The interaction between the distributed mutual exclusion algorithm and the mutual exclusion module is transparent, therefore these two components are not co-depending. The objective is to preserve correctness properties of the original algorithm in such a way that the extended algorithm is also correct. Our framework provides guidelines to build future extension works in a correct and clean manner.

Key words: Mutual exclusion, critical section, algorithms, distributed mutual exclusion, multithreading.

1 Introduction

The critical section problem occurs when several processes try to concurrently use shared resources [1]. A solution to the problem must guarantee mutual exclusion through time for several concurrent processes trying to access their critical sections. Within a monoprocessor or a shared memory multiprocessor the problem is solved using some synchronization hardware primitive to implement basic synchronization mechanisms such as semaphores, locks or monitors.

In a distributed system, the lack of a global state difficults the implementation of a solution for the critical section problem. The solution is obtained with a distributed mutual exclusion algorithm. In the literature there are plenty of these

algorithms and they are classified as centralized and distributed algorithms [1]. In centralized algorithms there is a coordinator node that manages and grants the access to the critical section. In distributed algorithms, all nodes are symmetric and do the same work. A distributed algorithm can be based either on *token* or permissions [2].

Almost all distributed mutual exclusion algorithms do not support multiple requests per-node. This constraint affects the implementation of distributed exclusion algorithms particularly in distributed systems with multithreading support. On these systems, processes executing at each node may have multiple concurrent threads executing. Thereby, the study of distributed mutual exclusion algorithms with multithreading support is important to solve the critical section problem for the several threads executing on different nodes of the distributed system.

To the best of our knowledge, there are only a few algorithms with these properties. The DSM-PEPE system implements a decentralized distributed mutual exclusion algorithm with multithreading support [3,4], which can be considered an extension of Raymond's distributed mutual exclusion algorithm [5]. Mueller proposes a similar algorithm [6], but it is also an extension of another distributed mutual exclusion algorithm, in that case, proposed by Naimi *et al.* [7]. Perez [8] presents an extension scheme to several *token*-based distributed mutual exclusion algorithms in order to be able to support multithreading.

Thus all related works for distributed systems with multi-threading support that we know, have been presented as extensions of normal distributed mutual exclusion algorithms which can only serve one request per node at a time.

In this paper we present a generic framework for the construction of distributed exclusion algorithms with multiple requests per-node support. Each algorithm is built from a distributed mutual exclusion algorithm and a local mutual exclusion module. These two components are integrated through an extension *key* and the definition of clear communication interfaces for the *key*. The framework defines guidelines to extend any distributed mutual exclusion algorithm. The main idea is to maintain transparency in the interaction between the components, so the involved algorithms are not aware of the existence of the other one. The purpose is to preserve correctness properties of the original distributed mutual exclusion algorithm, implying that the extended algorithm is also correct.

From now on, we will refer to the concept of distributed mutual exclusion as *dmutex* and to local mutual exclusion among processes within a single node as *mutex*.

This paper is organized as follows: in section 2, we describe the framework and the interaction between its components, in section 3 a generic implementation of the framework is proposed. In Section 4 we present the correctness proof outline. Finally, in section 5, we present our conclusions, current and future work.

2 The Framework

Our framework allows the extension of existing *dmutex* algorithms to support multiple request per-node. Particularly, the framework is designed for distributed systems with multithreading support.

To the best of our knowledge, all works that implement *dmutex* algorithms with multithreading support, have been done through the extension of existing *dmutex* algorithms. Generally, an extension work consists in joining a *dmutex* algorithm, to insure mutual exclusion between nodes, and *mutex* mechanisms, such as a service policy and synchronization methods, that serve local threads when the distributed right to access the critical section has been granted to a node. These extensions have a particular criterium to guide the service policy. In the process of joining the components, they are modified and merged.

Also, there are suggested extensions for *dmutex* algorithms based on a simple criterium. Ricart and Agrawala [9] and Raymond [5] state that the criterium to extend their *dmutex* algorithms is to serve one local request per node at a time, using the *dmutex* algorithm and serializing the service of local requests from the same node. Their work lacks of a study about the effects in the performance induced by this approach. We could think of some other criteria to serve these local requests, decreasing the amount of total messages. We could use for example, a number n of local requests to serve per node at a time, with $n > 1$.

The purpose of our framework is to give guidelines to build the joining process in a transparent way. This should not modify neither the properties nor the structure of the original *dmutex* algorithm during the extension process. Thereby, the framework maintains a high transparency level in the union of the *dmutex* algorithm and the *mutex* mechanisms. This transparency also lets us use several criteria or strategies to guide the service of the local requests.

We present the *mutex* module concept, as the encapsulation of the *mutex* mechanisms. Then a *mutex* module is mainly composed by a service policy and synchronization methods. The service policy establishes the order in which the threads are served and the synchronization methods provide local mutual exclusion. The transparency that our framework looks for is between the *dmutex* algorithm and the *mutex* module.

In order to build an extension, we need a joining mechanism that establishes the communications and determines the interactions between the *dmutex* algorithm and the *mutex* module. We introduce the concept of *key* as the joining mechanism, a third component that must be defined to build an extension. This *key* is central to the joining process. However, its main task is to establish the strategy or criterium to guide the service of the local requests.

On the other hand, if we want to maintain the components separated but working together, we need to build corresponding interfaces. Therefore, we need an

interface for the communication between the *dmutex* algorithm and the *key* and an interface for the communication between the *mutex* module and the *key*.

The framework establishes that, in order to extend a *dmutex* algorithm, it is necessary to define the following: (1) a *dmutex* algorithm - *key* interface, (2) a *mutex* module - *key* interface and (3) an extension *key*. The interfaces defined can be different for each *dmutex* algorithm and for each *mutex* module. Besides, different definitions of *keys* can be used. This scheme can be seen in the Figure 1.

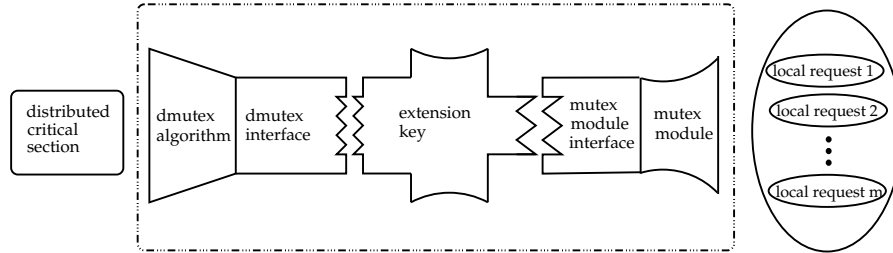


Fig. 1. Extension Scheme

Transparency in the extension process is obtained by using correct interfaces. It is important that all the interfaces be clean and transparent. In this way, high obliviousness is produced between the *dmutex* algorithm and the *mutex* module. Each one works independently and is not aware about the existence of the other and their interaction.

Clean and well designed interfaces allow the use of the same interface or its use as a pattern over different *dmutex* algorithms. The same principle can be applied to several *mutex* modules. Likewise, they allow certain independency of the *key* with respect to the other components.

The *key* must use the interfaces to build the interaction between the components, but it also can obtain information to develop its strategy to guide the local requests' service. This possibility allows the strategy of the *key* to depend or not on the interfaces. Hence, there is a trade-off between generality and efficiency.

The correctness of the resulting extension depends on the correctness of the *dmutex* algorithm and the correctness of the *mutex* module. Then, a correct *dmutex* algorithm and a correct *mutex* module must be used. In section 4 we will prove the correctness of the extension process having these hypotheses and some other important considerations.

In this paper we use the following concepts:

- **Node/Local Request:** Request to access the critical section. A node request is generated by the node and handled by the *dmutex* algorithm in order to get *dmutex* for the node. A local request is

- generated locally, inside the node, and handled by the *mutex* module in order to get *mutex* within the node.
- **CS**: Used to refer to the critical section.

2.1 The *Dmutex* Interface

The definition of an interface between the *dmutex* algorithm and the *key* –that we will call *dmutex* interface– involves the establishment of basic algorithm properties and a set of events that characterizes it. The properties of the algorithm define functionality and the events allow the definition of *key* operations.

The main task of the *dmutex* interface is to receive and serve the node requests generated by a node. The node generates a node request through the *key*, which will receive and process the reply to this request. Therefore, the interface basically offers methods for the *key* to execute entry and exit protocols for the *dmutex* algorithm. Also, the interface offers event notifying service, for events like the arrival of the distributed right to access the CS at the node (*dmutex* granted).

The execution of these methods and events follows the order of a normal CS protocol. The node request is issued by the *key* by calling the method to execute the entry protocol on behalf of the node. When the *dmutex* algorithm grants the distributed right to access the CS to the node, it notifies the *key* with an event. When the job of the *key* is done, it relinquishes the distributed right by calling the method that corresponds to the exit protocol.

Differences in the implementation of the *dmutex* algorithms are not relevant to the extension process, because the communication with the *dmutex* algorithm is accomplished through a well defined interface. For example, for decentralized *token*-based *dmutex* algorithms, the event of receiving the *token* represents the reception of the distributed right to access the CS. However, for decentralized permission-based *dmutex* algorithms, the event of receiving all the permissions required is to obtain the distributed right to access the CS. In both cases, the *key* only needs to know when the distributed right to access the CS has been granted to the node.

Furthermore, the interface with the *dmutex* algorithm must offer relevant data concerning algorithm management that could be useful to the *key*. For example, permission-based *dmutex* algorithms frequently use *timestamps* for event serialization. The *dmutex* algorithm should provide these *timestamps* to the *key* for its use. However, a *key* that builds its strategy to guide the service based on this event loses generality.

A generic *dmutex* interface provides two methods exported by the *dmutex* algorithm and one event to which the *key* subscribes to and whose occurrence is notified by the algorithm. These elements are:

- `DMUTEX_REQUEST()`: Method to request *dmutex* (entry protocol).

- `DMUTEX_RELEASE()`: Method to release *dmutex* (exit protocol).
- `ON_DMUTEX_GRANTED`: Event raised when the distributed mutual exclusion is granted to the node.

2.2 The *Mutex* Module Interface

A *mutex* module is an encapsulation of *mutex* mechanisms. The *mutex* mechanisms are commonly composed by a service policy and synchronization mechanisms. The service policy is basically a service queuing order for the multiple requests to enter the CS generated by a node locally. The most simple and known policy is FIFO. On the other hand, the synchronization mechanisms ensures the mutual exclusion among local requests from the same node. They consists of synchronization methods like `Acquire()` and `Release()`.

The porpouse of using a *mutex*-module concept is to focus on transparency in the extension process. This allows us to use different service policies, using the same interface for the synchronization methods from the synchronization mechanisms.

The definition of an interface between the *mutex* module and the *key* –that we will call *mutex*-module interface– involves the establishment of a protocol between the *key* and the local-request service policy of the *mutex* module. This protocol allows the *key* to guide the service of local requests, although the local requests are actually served by the *mutex* module.

The protocol is represented through an interface that offers methods and notification of events that: *i*) signal when a new local request is generated, *ii*) let the *mutex* module to serve the next local request, and *iii*) signal when a local request has been served. The event in *i*) allows the *key* to be aware of new local requests being handled by the *mutex* module. The method in *ii*) allows the *key* to signal the *mutex* module to continue and serve the next local request according to the *mutex*-module service policy. Finally, the event in *iii*) singals when the *mutex* module finishes the service of a local request.

Furthermore, the *mutex*-module interface must provide data about the local requests active in the node, that could be useful to the *key*. Like in the case of the *dmutex* interface, a *key* with more data available could be more specialized to a particular extension.

A generic *mutex*-module interface provides one method exported by the *mutex* module and two events for the *key* to subscribe. The occurrence of each of these event's is notified by the module. They are:

- `SERVE_LOCAL_REQUEST()`: Method to command the service of a local request service.
- `ON_NEW_LOCAL_REQUEST`: Event that notifies when a new local request occurs.
- `ON_LOCAL_REQUEST_COMPLETED`: Event that notifies the completion of the current local request service.

2.3 Components Interaction

The work done by the *key* summarizes the interaction of the components. There are six steps that conform these interaction dynamics in which the *key* does the main part. This can be seen in Figure 2. We describe these steps using references to the generic *dmutex* interface and the *mutex*-module interface proposed:

1. A local request to enter the CS is generated and handled by the *mutex* module. The user program issues a new request to the CS through some synchronization mechanism or method of which the *mutex* module is aware.
2. The *mutex* module notifies the *key* that there is a local request to the CS. The *mutex* module interface provides to the *key* a subscription service to the `ON_NEW_LOCAL_REQUEST` event. When this event occurs the *key* is notified.
3. The *key* notifies the *dmutex* algorithm that there is a node request. When the *key* knows that there is at least one local request pending, it issues a node request. This involves the *key* executing the entry protocol of the *dmutex* algorithm by calling the method `DMUTEX_REQUEST()` provided by the *dmutex* interface. If a node request was previously issued and it has not been served yet by the *dmutex* algorithm, then the *key* just waits for the `ON_DMUTEX_GRANTED` event.
4. The *dmutex* algorithm notifies the *key* when the distributed right to access the CS has been granted to the node. Previously, the *dmutex* algorithm must communicate with other nodes in order to ensure the distributed right to enter the CS among nodes. Once the right is granted to the node, the `ON_DMUTEX_GRANTED` event is raised to notify the *key* through the interface.
5. The *key* executes the cycle for service of local requests. Only when the distributed right to access the CS is granted to the node, the *key* signals the *mutex* module to serve the next local request. The *key* signals the *mutex* module by calling the `SERVE_LOCAL_REQUEST()` method of the *mutex* module interface. Once the service of a single local request is completed, the *mutex* module signals the `ON_LOCAL_REQUEST_COMPLETED` event to the *key*. Then, the *key* can signal the *mutex* module again, starting the next iteration of the cycle for service of local requests. The *key* decides when to stop serving consecutive local requests according to its strategy of service. When this happens, the cycle ends.
6. The *key* notifies the *dmutex* algorithm that it must release the distributed right to access the CS. When the cycle for service of local requests ends the *key* calls the method `DMUTEX_RELEASE()` that invokes the exit protocol of the *dmutex* algorithm, releasing the distributed right to access the CS on behalf of the node. After this, if the *key* knows that there is still at least one local request pending, the *key* will issue another node request.

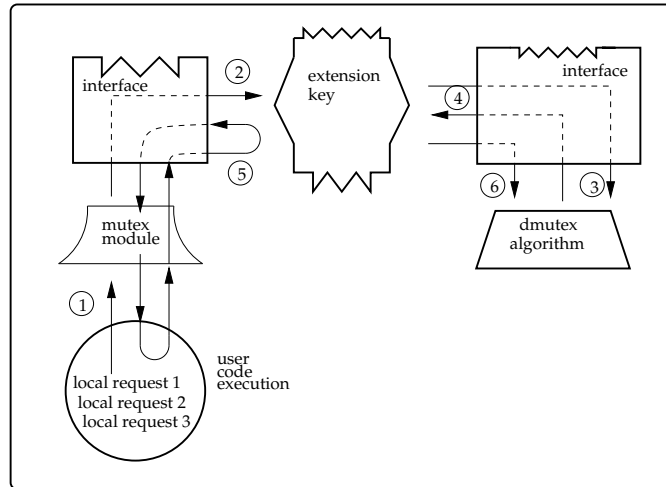


Fig. 2. Interaction Steps: 1) A local request to enter the CS is generated and handled by the *mutex* module 2) The *mutex* module notifies the key that there is a local request to the CS 3) The key notifies the *dmutex* algorithm that there is a node request 4) The *dmutex* algorithm notifies the key that distributed right to access the CS has been granted to the node 5) The key executes the cycle for service of local requests 6) The key notifies the *dmutex* algorithm to release the distributed right to access the CS

2.4 The Key

The definition of the *key* involves building an interaction mechanism between the *dmutex* algorithm and the *mutex* module. The *key* achieves this interaction using the events and methods offered by the interfaces. In addition to establish this interaction, the *key* settles the strategy to guide the service of the local requests when the *dmutex* is granted to the node. This is the main part of the definition of an extension *key*. This strategy involves that the *key* decides in which moment it will stop guiding the *mutex* module to serve local requests because the *key* can signal the *mutex* module several times during the cycle for service of local requests.

We present three simple generic criteria to define when to stop signaling the service of local requests:

- **To serve until N local requests:** If we have $N=1$, we will serve just one local request per-node, forcing the node to make a new node request if there are more than one local request pending. It is important to mention that this is the most common proposal made by other authors as a valid method for the extension of *dmutex* algorithms to support multithreading.

Another idea is to serve the current number of local requests when the distributed mutual exclusion is granted. This can be called L_{qCS} , where qCS

is the current local requests queue when the distributed mutual exclusion is granted and L_{qCs} is its length. Then, $N=L_{qCs}$.

The *key* with $N=0$ local requests fails to fulfill the progress property, producing starvation.

The criteria with $N > 0$ needs to check repeatedly if there still are local requests. If there are not, although N is not achieved, the *key* must stop serving local requests to avoid starvation.

- **To serve until N local requests are left:** The *keys* that are based on this criterium are practical, but they are not free of starvation problems. For instance, the *key* that serves local requests until $N=0$ are left, benefits the node possessing the distributed mutual exclusion, and it would decrease the number of messages, improving locality. Nevertheless, this is obtained with the risk of an eventual starvation problem.
- **To serve local requests within a Δt time frame:** Local requests are served meanwhile the time is not over and there are pending local requests. This guarantees maximum latencies per-node.

These *keys* are relatively simple and easy to implement, but they do not take advantage of the particular *dmutex* algorithm properties.

A *key* is able to extend different *dmutex* algorithms if the strategy follows a generic criterium to define when to stop local requests. Following a generic criterium only needs to use the generic interface proposed for the *dmutex* algorithm and the *mutex* module. But if we want to take advantage of the *dmutex* algorithm properties, we must use some particular interface. This is also the case for the *mutex* module interface. If the strategy to guide the service depends on these properties, it will probably depend on a particular interface, losing generality on the *dmutex* algorithms that it could extend.

Every extension *key* must consider a fundamental property: When the number of local requests issued by a node is limited to one, the behaviour of the extended *dmutex* algorithm should be similar to the original *dmutex* algorithm.

3 Generic Implementation

In order to prove some properties of correctness of the framework we need a general implementation that maintains the main purpose of this work, maintaining independence of any particular environment. Next, we present this generic implementation for each framework component: *mutex* module interface, *dmutex* interface and *key*.

3.1 The *Mutex* Module Level

The pseudocode below summarizes the usage of the methods offered by the *mutex* module to enter a distributed CS in a transparent way. It is important to mention that the order in which these methods are called must be followed as shown:

```
User_process:
...
MMX_Acquire();
/* {Critical Section} */
MMX_Release();
...
```

Then, the implementation of these methods looks like:

```
MMX_Acquire(){
    send_event(ON_NEW_LOCAL_REQUEST);
    Acquire();
    wait_for_event(ON_CS_PERMITTED);
}

MMX_Release(){
    Release();
    send_event(ON_LOCAL_REQUEST_COMPLETED);
}

SERVE_LOCAL_REQUEST(){
    send_event(ON_CS_PERMITTED);
}
```

At the application level, the user code executes within a process or thread, issuing requests through available synchronization methods. Conventionally, the synchronization methods available are `Acquire()` – blocking method– and `Release()`, which offer simple local mutual exclusion to access the CS. Our *mutex* module provides these methods to be used only for *mutex*. However, we must extend these methods for them to fit in the framework and provide *mutex* within *dmutex*.

The atomic methods `MMX_Acquire()` and `MMX_Release()` – MMX stands for *mutex* module extension – protect the CS and they are provided by the *mutex* module to be used in the user code to fit in the framework extension. These methods make a call to `Acquire()` and `Release()`, that are conventional synchronization methods of the *mutex* module that hide the local service policy applied. `MMX_Acquire()` triggers the `ON_NEW_LOCAL_REQUEST` event and blocks on conditional wait for an `ON_CS_PERMITTED` event. We introduce now this event and this implies that a process can continue and execute the CS. `MMX_Release()` triggers the `ON_LOCAL_REQUEST_COMPLETED` event.

When the *key* calls the method `SERVE_LOCAL_REQUEST()`, the `ON_CS_PERMITTED` event is triggered, allowing the process that is waiting for this event to resume its execution. This is the method that allows the *key* to signal the *mutex* module to serve the next local request.

Additionally, we must state that *the CS in the user code has a finite execution time*. This assumption will be critical for any proof of liveness property of the framework.

3.2 The *Dmutex* Algorithm Level

The *dmutex* algorithm works when a `DMUTEX_REQUEST()` method is called by the *key*. Likewise when the *key* executes a `DMUTEX_RELEASE()` method, it invokes the *dmutex* algorithm. `DMUTEX_REQUEST()` is a non-blocking method, because the *key* needs to be aware of the local `ON_NEW_LOCAL_REQUEST` events at any time.

The usage of these methods is shown in the pseudocode below. Note that this code illustrates how the *dmutex* interface would be used in the user code, independently of the framework. Thus, we must use conditional wait for the event `ON_DMUTEX_GRANTED`:

```
User_process:
...
DMUTEX_REQUEST();
wait_for_event(ON_DMUTEX_GRANTED);
/* {Critical Section} */
DMUTEX_RELEASE();
...
```

3.3 The *Key* Level

We use monitor syntax – and assume monitor semantics – to show the implementation of the *key*. An alternative but equivalent synchronization technique could be used instead. This way we avoid inconsistencies in the state of the *key*, produced by multiple concurrent requests within a node. Also, we can prevent a second consecutive execution of `DMUTEX_REQUEST()` if a `DMUTEX_RELEASE()` is not executed before.

We assume that a single *key* manager process works at every node. The procedures used by the *key* at node *i* are encapsulated in a monitor running at that node.

The values *T* and *F* are respectively *true* and *false*.

```
monitor KEY_KEEPER:
    int LOCAL_REQ_COUNT = 0;
    bool DMUTEX_REQ_COMPLETED = T;

    cond LOCAL_REQ_COMPLETED;

    procedure ON_NEW_LOCAL_REQUEST(){
        LOCAL_REQ_COUNT = LOCAL_REQ_COUNT+1;
        if (DMUTEX_REQ_COMPLETED ≠ F)
        {
            DMUTEX_REQ_COMPLETED = F;
            DMUTEX_REQUEST();
        }
    }

    procedure ON_LOCAL_REQUEST_COMPLETED(){
        LOCAL_REQ_COUNT = LOCAL_REQ_COUNT-1;
        signal(LOCAL_REQ_COMPLETED);
    }

    procedure ON_DMUTEX_GRANTED(){
        while (KEY_CONDITION = T)
        {
            SERVE_LOCAL_REQUEST();
            wait(LOCAL_REQ_COMPLETED);
        }
        DMUTEX_RELEASE();
        DMUTEX_REQ_COMPLETED = T;
        if (LOCAL_REQ_COUNT ≠ 0)
        {
            DMUTEX_REQ_COMPLETED = F;
            DMUTEX_REQUEST();
        }
    }
}
```

Description of variables and conditions used:

- `LOCAL_REQ_COUNT`: Integer counter, initially zero. Keeps the accumulated number of non-completed local requests.
- `DMUTEX_REQ_COMPLETED`: Boolean condition, initially *T*. Becomes *F* when a `DMUTEX_REQUEST()` is executed. Remains *F* until a `DMUTEX_RELEASE()` is executed. This condition makes that just one single `DMUTEX_REQUEST()` be executed until `DMUTEX_RELEASE()` is executed.
- `KEY_CONDITION`: Boolean condition that corresponds to a dynamical or static predicate. Represents the strategy of the *key* to guide the service of local requests. For instance, a simple predicate could be:

$$\text{KEY_CONDITION} \leftrightarrow \text{LOCAL_REQ_COUNT} \neq 0$$

- `LOCAL_REQ_COMPLETED`: Condition variable of the monitor, used to wait for the local request being served to be completed.

The first procedure is called when an `ON_NEW_LOCAL_REQUEST` event occurs, notified by the *mutex* module interface. This procedure executes a `DMUTEX_REQUEST()` if there is no previous one pending.

The second procedure is called when an `ON_LOCAL_REQUEST_COMPLETED` event occurs, notified by the *mutex* module interface. This procedure executes a `signal()` operation to wake up the process blocked at the `ON_DMUTEX_GRANTED()` procedure which was waiting for the condition variable `LOCAL_REQ_COMPLETED` in its `while()` cycle.

The third procedure is called when an `ON_DMUTEX_GRANTED` event occurs and is notified by the *dmutex* interface. This procedure implements two tasks: execute the cycle for service of local requests, and to execute a `DMUTEX_RELEASE()` on behalf of the node. Before this procedure ends, it executes a `DMUTEX_REQUEST()` if there still are local requests pending.

4 Correctness

We addressed two main correctness properties in our work: *i*) at any time, *at most one* single local request is being served system-wide, and *ii*) a local request will always be *served in finite time*.

We must consider certain assumptions and we introduce some variables to be used in the proof.

There are *n* nodes in the system. At any node with id *i*, there are at most *m* active local requests at any time. We make this restriction without losing generality, because a local request queue is always bounded and finite. This is the scheme used in multithreading, when each node starts with a fixed number of possible threads.

Also, we must consider and note important assumptions about the order of possible executions of the procedures used by the *key* and the events. The possible order of execution of procedures used by the *key* is defined by the events of the generic interfaces. This was described in the interaction of the components of the framework (section 2.3). Also, this order is shown in figures 3, 4 and 5.

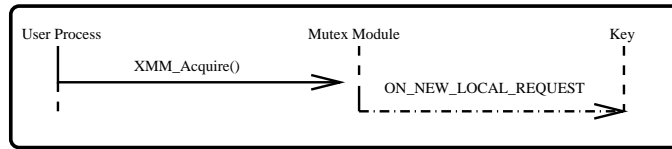


Fig. 3. Framework Entry Protocol

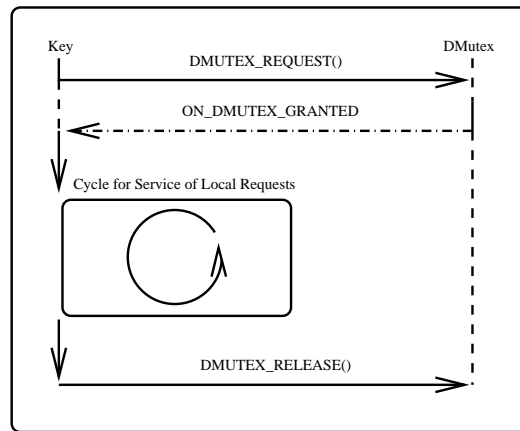


Fig. 4. Framework Entry - Exit Protocol at the *Dmutex* level

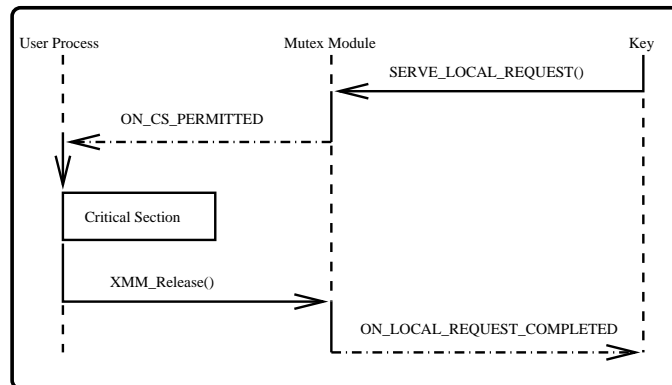


Fig. 5. Framework Cycle for Service of Local Requests

We consider an integer variable $dmutex_here$ at each node valued 0 or 1. It is initialized as 0 and is only modified when a node receives the distributed right to enter the CS from the $dmutex$ algorithm. We will refer to this variable at node i as $dmutex_here_i$.

The following code shows the use of this variable in the context of the example of section 3.2:

```
User_process:
...
DMUTEX_REQUEST();
wait_for_event(ON_DMUTEX_GRANTED);
dmutex_here = 1;
/* Critical Section */
dmutex_here = 0;
DMUTEX_RELEASE();
...
```

We also consider at each node an integer array $lmutex_here[1:m]$ valued 0 or 1. The variable $lmutex_here[j]$ is initially zero for all possible local request j and it is only modified when the local request j enters the CS. We will refer to the variable at node i for local request j as $lmutex_here_i[j]$.

The following code shows the usage of these variables for a local request j , using the implementation presented on section 3.1:

```
MMX_Acquire(){
    send_event(ON_NEW_LOCAL_REQUEST);
    Acquire();
    wait_for_event(ON_CS_PERMITTED);
    lmutex_here[j]=1;
}
MMX_Release(){
    lmutex_here[j]=0;
    Release();
    send_event(ON_LOCAL_REQUEST_COMPLETED);
}
```

4.1 Proof Outline: *At most one*

Our proof outline is based on the preservation of the *At most one* property in the $dmutex$ algorithm and in the $mutex$ module implemented under section 3.

- **Preservation of the property in the $Dmutex$ Algorithm:** In order to obtain a correct extension we must assure that the $dmutex$ algorithm used is correct. The proof is based on the following two hypothesis that we need to be correct:
 - **H1:** The original $dmutex$ algorithm to be extended is correct.

In addition to the correctness of the $dmutex$ algorithm, we also need the CS to be finite. In this framework, the CS for the $dmutex$ algorithm is represented by the procedure `ON_DMUTEX_GRANTED()`, which has a `KEY_CONDITION` in its inner `while()` cycle. Therefore, for this procedure to be finite, we need `KEY_CONDITION` become false eventually.

- **H2**: KEY_CONDITION becomes false eventually.

In fact, this constraint allow us to prevent starvation.

Then, using the variables introduced previously, implementation at the *dmutex*-algorithm level must always preserves the following invariant:

DMUTEX:

$$\{\forall i, k : 1 \leq i, k \leq n : dmutex_here_i = 1 \rightarrow (k \neq i \rightarrow dmutex_here_k = 0)\}$$

This invariant inherently includes the special case when none of the nodes have the distributed right to enter the CS (all *dmutex_here_i* = 0).

Then, we must prove the following lemma:

Lemma 1. : *The generic implementation of the framework preserves the DMUTEX invariant.*

Proof: ON_NEW_LOCAL_REQUEST() and ON_LOCAL_REQUEST_COMPLETED() procedures do not modify the *dmutex_here* variable.

On the other hand, the ON_DMUTEX_GRANTED() procedure is triggered only by an ON_DMUTEX_GRANTED event. By hypothesis **H1** this event will only occur at a single node in the system. The following code shows the usage of *dmutex_here* variable in this procedure:

Node *i*:

```

procedure ON_DMUTEX_GRANTED() {
  dmutex_herei = 1;
  while (KEY_CONDITION = T)
  {
    SERVE_LOCAL_REQUEST();
    wait(LOCAL_REQ_COMPLETED);
  }
  dmutex_herei = 0;
  DMUTEX_RELEASE();
  DMUTEX_REQ_COMPLETED = T;
  if (LOCAL_REQ_COUNT ≠ 0)
  {
    DMUTEX_REQ_COMPLETED = F;
    DMUTEX_REQUEST();
  }
}

```

Initially *dmutex_here* is 0 at every node. When ON_DMUTEX_GRANTED() executes at node *i*, *dmutex_here_i* is the only variable that changes its value to 1. Due to hypothesis **H2**, the KEY_CONDITION becomes eventually **F**, then the **while()** cycle ends and the variable is changed to 0. Then, before the DMUTEX_RELEASE() method is executed, *dmutex_here* variables at all nodes are 0 and they remain unchanged until another ON_DMUTEX_GRANTED() procedure is triggered in the system.

- **Preservation of the property in the *Mutex* Module:** We also need to assure that the property *At most one* is preserved by the *mutex* module, that is to say, we need that the *mutex* module to be correct. We introduce the following hypothesis:

- **H3:** The *mutex* module used is correct.

Therefore, implementation at the *mutex*-module level must always preserve the following invariant:

LMUTEX:

$$\{\forall i, j, l: 1 \leq i \leq n, 1 \leq j, l \leq m : \quad lmutex_here_i[j] = 1 \rightarrow (l \neq j \rightarrow lmutex_here_i[l] = 0)\}$$

Then, we must prove the following lemma:

Lemma 2. : *The generic implementation of the framework preserves the LMUTEX invariant.*

Proof: The proof is based in two conjectures.

C1: *There is only one local request j that changes $lmutex_here[j]$ to 1 at node i :* Initially all values in the array $lmutex_here[1:m]$ are 0 at node i . Then, the value of $lmutex_here[j]$ is changed to 1, only when the local request j has passed by `wait_for_event(ON_CS_PERMITTED)` within `MMX_Acquire()`.

However, before blocking at `wait_for_event(ON_CS_PERMITTED)`, local request j triggered an event `ON_NEW_LOCAL_REQUEST` and executed `Acquire()` which is another blocking method. Due to hypothesis **H3** we can assume that only one local request passed by the `Acquire()` command and blocks in `wait_for_event(ON_CS_PERMITTED)`.

Upon reception of the `ON_NEW_LOCAL_REQUEST` event the procedure with the same name executes. This procedure calls the `DMUTEX_REQUEST()` method, which triggers an `ON_DMUTEX_GRANTED` event to occur at the node, executing the `ON_DMUTEX_GRANTED` procedure.

When `ON_DMUTEX_GRANTED()` is executed, method `SERVE_LOCAL_REQUEST()` is called, which triggers an `ON_CS_PERMITTED` event, unblocking the only request that was waiting for this event.

C2: *The service cycle at node i do not execute again `SERVE_LOCAL_REQUEST()` in a new cycle until local request j , currently being served, changes its variable $lmutex_here[j]$ from 1 to 0:* When variable $lmutex_here[j]$ of local request j is 1, it means that this local request is executing its CS. After that, it executes `MMX_Release()` changing the value to 0 and triggering an `ON_LOCAL_REQUEST_COMPLETED` event. Then, the *key* procedure with the same name is executed.

This procedure signals the condition `LOCAL_REQ_COMPLETED` which unblocks the cycle for service of local requests at `ON_DMUTEX_GRANTED()`. Therefore, `SERVE_LOCAL_REQUEST()` executes when all `lmutex_here[j]` at node i are 0.

Having shown that the implementation at the *dmutex*-algorithm level and at the *mutex*-module level preserves the property *at most one*, we now prove that the implementation at the framework level preserves also the property *At most one*.

The Framework and the property *At most one* From the code of the `ON_DMUTEX_GRANTED()` procedure we can make the following remark:

Remark 1 *The cycle for service of local requests executes \leftrightarrow `dmutex_here` = 1*

Now we can prove the following lemmas:

Lemma 3. *Generic implementation of the framework preserves the **dm1lm1** invariant:*

$$\mathbf{dm1lm1} : \{ \forall i, l : 1 \leq i \leq n, 1 \leq l \leq m : \mathit{dmutex_here}_i = 0 \rightarrow \mathit{lmutex_here}_i[l] = 0 \}$$

Proof: By remark 1, if `dmutex_here` = 0 then `ON_DMUTEX_GRANTED()` is not executing or it is not executing the cycle for service of local requests. Thereby, all values of the array `lmutex_here[1:m]` at node i are 0.

Lemma 4. *Generic implementation of the framework preserves the **lm2dm2** invariant.*

$$\mathbf{lm2dm2} : \{ \forall i, j : 1 \leq i \leq n, 1 \leq j \leq m : \mathit{lmutex_here}_i[j] = 1 \rightarrow \mathit{dmutex_here}_i = 1 \}$$

Proof: If `lmutex_here_i[j]` = 1 then cycle for service of local requests is executing. By remark 1, this occurs because `dmutex_here_i` = 1.

Now we prove the following theorem:

Theorem 1 *Generic implementation of the framework preserves the **GDMUTEX** invariant.*

GDMUTEX:

$$\{ \forall i, j, k, l : 1 \leq i, k \leq n, 1 \leq j, l \leq m : \mathit{lmutex_here}_i[j] = 1 \rightarrow (k = i \wedge l \neq j \rightarrow \mathit{lmutex_here}_k[l] = 0) \wedge (k \neq i \rightarrow \mathit{lmutex_here}_k[l] = 0) \}$$

Proof: By lemma 2, we know that the generic implementation of the framework preserves the **LMUTEX** invariant, which can be re-written as:

$$\{ \forall i, j, k, l : 1 \leq i \leq n, 1 \leq j, l \leq m : \mathit{lmutex_here}_i[j] = 1 \rightarrow (k = i \wedge l \neq j \rightarrow \mathit{lmutex_here}_k[l] = 0) \}$$

By lemmas 4 and 1, then the framework preserves the *lm2dm1* invariant:

lm2dm1:

$$\{\forall i, k : 1 \leq i, k \leq n, 1 \leq j \leq m : lmutex_here_i[j] = 1 \rightarrow (k \neq i \rightarrow dmutex_here_k = 0)\}$$

By invariant *lm2dm1* and lemma 3 (changing *i* for *k*), the framework preserves the *lm2lm1* invariant:

lm2lm1:

$$\{\forall i, j, k : 1 \leq i, k \leq n, 1 \leq j \leq m : lmutex_here_i[j] = 1 \rightarrow (k \neq i \rightarrow lmutex_here_k[l] = 0)\}$$

Therefore, by preserving the invariants *LMUTEX* and *lm2lm1*, the framework preserves the *GDMUTEX* invariant \square .

4.2 Proof Outline: *Service in finite time*

We now will prove that a local request is always *served in finite time*.

We first have to state a constraint over the *KEY_CONDITION* predicate and the *ON_DMUTEX_GRANTED()* procedure. This constraint allows us to guarantee that the *KEY_CONDITION* and its strategy to guide the service of local requests will never interfere with the *served in finite time* property. Then, the next hypothesis is need to be correct:

- **H4**: *KEY_CONDITION* is not always false when it is checked in the first iteration of the cycle for service of local requests.

This Hypothesis means that if *KEY_CONDITION* is false at the moment when the *ON_DMUTEX_GRANTED()* procedure starts, the cycle for service of local requests will not execute. This can happen only for a finite number of *ON_DMUTEX_GRANTED()* executions before *KEY_CONDITION* becomes true when the *ON_DMUTEX_GRANTED()* procedure starts. This Hypothesis is based on the fact of building a correct *KEY_CONDITION*. This predicate must be true in order to allow the service of the local requests and to prevent starvation. For an instance, a *KEY_CONDITION* that is always true when *ON_DMUTEX_GRANTED()* starts is enough to verify the Hypothesis.

Also, in order to prove this property we assume that the *dmutex* algorithm is correct (**H1**) and that the *KEY_CONDITION* becomes false eventually (**H2**). We prove the following lemma:

Lemma 5. *ON_DMUTEX_GRANTED() always finishes.*

Proof: When *KEY_CONDITION* = **F**, the cycle ends and the procedure finishes. Hypothesis **H2** states that *KEY_CONDITION* becomes false eventually, thereby the lemma follows.

Besides it is necessary to prove that, after any execution of the `DMUTEX_REQUEST()` method, the procedure `ON_DMUTEX_GRANTED()` starts in finite time. Then, the following lemma is proven:

Lemma 6. *The procedure `ON_DMUTEX_GRANTED()` will start within a finite time after a `DMUTEX_REQUEST()` method is called at the same node.*

Proof: Hypothesis **H1** implies that the distributed right to enter the CS is granted in finite time and Lemma 5 imply that any other execution of the `ON_DMUTEX_GRANTED()` procedure is finite. Therefore the lemma follows.

We also need the service policy of the *mutex* module to be fair. Then, from Hypothesis **H3** we can state the following hypothesis:

- **H5:** The service policy of the *mutex* module is fair.

Now we can state the next lemma:

Lemma 7. *If the cycle for service of local requests executes continuously, then all local requests are served.*

Proof: This lemma follows directly from Hypothesis **H5**. In a mono-node system where only the *mutex* module is needed, all requests are served in finite time if the service policy is fair.

This last lemma is not enough to complete the proof, due to Hypothesis **H2** which implies that the cycle for service of local requests always ends. However, we can state the following complementary lemma:

Lemma 8. *While there is at least one local request at the node, then the cycle for service of local requests will continue executing except for finite periods of time.*

Proof: The cycle for service of local requests executes within the execution of the `ON_DMUTEX_GRANTED()` procedure. On the other hand, if there is at least one local request pending at the node, then `DMUTEX_REQUEST()` was executed at some time. By Lemma 6, after `DMUTEX_REQUEST()` execution, `ON_DMUTEX_GRANTED()` will start in a finite time and so the cycle for service of local requests, due to Hypothesis **H4**. By Hypothesis **H2** the service cycle ends but, if there is at least one local request pending before `ON_DMUTEX_GRANTED()` finishes its execution, this procedure will call `DMUTEX_REQUEST()`. Therefore `ON_DMUTEX_GRANTED()` executes again and a new cycle will start within a finite time.

Now we can prove the following theorem:

Theorem 2 *All local requests are served in finite time.*

Proof: A local request that wants to enter to the CS calls `MMX_Acquire()` method, which triggers the proper event that makes the `ON_NEW_LOCAL_REQUEST()` procedure to be executed.

In this last procedure, if the variable `DMUTEX_REQ_COMPLETED = T`, then the `DMUTEX_REQUEST()` method is executed. By Lemma 6, `ON_DMUTEX_GRANTED()` starts within a finite time.

If `DMUTEX_REQ_COMPLETED = F`, then `ON_DMUTEX_GRANTED()` is executing and it is blocked at the `wait()` instruction, or `DMUTEX_REQUEST()` was previously executed at the node. In the latter, by Lemma 6, `ON_DMUTEX_GRANTED()` starts within a finite time.

The cycle for service of local requests executes within the `ON_DMUTEX_GRANTED()` procedure execution. Because of this and the possibilities for the execution of `ON_DMUTEX_GRANTED()`, the cycle is blocked at the `wait()` instruction, or will begin to execute within a finite time before `MMX_Acquire()` is executed.

The actual local request could be served during the current service cycle. If not, by Lemma 8, the cycle will continue executing, except for finite periods of time. By Lemma 7, if the cycle executes continuously, then the local request will be served within finite time \square .

5 Conclusions and Future Work

We have presented a framework that allows to extend distributed mutual exclusion algorithms in order to support multiple requests per-node. The framework applies specially to distributed systems with multithreading support.

The extended algorithms that are pretended to be obtained through these guidelines are constructed by defining extension *keys* and interfaces for these *keys* with the distributed mutual exclusion algorithm and with the local mutual exclusion module. The extensions allow to generate a transparent integration between the components and to preserve and/or inherit correctness properties from the extended distributed mutual exclusion algorithm. A fine definition for the *key* and the interfaces allows the use of a same *key* for different distributed mutual exclusion algorithms. We have also presented simple proposals for the *keys* and the interfaces.

The correctness is based upon a generic implementation of the framework and we show that it preserves two main properties. These properties summarize classic definitions of safeness and progress properties. The correctness is applied to every extension that could be done under the generic interface and complies with every single constraint and hypothesis. However, more extensions could be proven if every new event, method for the *dmutex* interface and procedure for the *key* also complies with all the constraints and validates all hypotheses established in the correctness section.

The *mutex* module does not need to be implemented as the proposed component in this framework. All that is needed from this component is its policy of service. All the synchronization needed is inside the cycle for service of local requests. Instead of generating an event for which only one local request is waiting, the event can be sent to the only local request that the service policy considers to be the next. We implemented the *mutex* module in the original proposed form, because of the transparency and to facilitate the correctness proof.

Current work involves the search for *keys* to capture existing current extension works. In particular, our efforts are addressed to Mueller's [6] extension and Meza's [10] extension. In future work, we will also focus on justice properties and efficiency comparisons by number of exchanged messages and average waiting times of local requests, working with the current extensions and the generic *keys* proposed. For some *keys* we will try to find some relation with the criterium of the *key* or a scheme for *keys* and the efficiency.

References

1. Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
2. Michel Raynal. A Simple Taxonomy for Distributed Mutual Exclusion Algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2):47–50, 1991.
3. Federico Meza, Alvaro E. Campos, and Cristian Ruz. On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers. In *International Conference on Computational Science and Its Applications, ICCSA 2003*, number 2667 in Lecture Notes in Computer Science, pages 967–976, Montreal, Canada, May 2003. Springer-Verlag.
4. Alvaro E. Campos and Federico Meza. DSM-PEPE: Un Sistema de Memoria Compartida Distribuida para Multicomputadores de Bajo Costo. In *Anales del VIII Congreso Argentino de Ciencias de la Computación, CACIC 2002*, pages 205–216, Buenos Aires, Argentina, October 2002. Article-C163.
5. Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.
6. F. Mueller. Decentralized Synchronization for Multithreaded DSMS. *Proc. of the 2nd. Workshop on Software Distributed Shared Memory (WSDSM 2000)*, May, 2000.
7. Mohamed Naimi, Michel Trehel, and André Arnold. A log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, 1996.
8. Jorge Pérez. Extending Distributed Mutual Exclusion Algorithms to Support Multithreading. Master's thesis, Departamento de Ciencia de la Computación, P. Universidad Católica de Chile, July 2004.
9. Glenn Ricart and Ashok K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Commun. ACM*, 24(1):9–17, 1981.
10. Federico Meza, Jorge Pérez, and Yadrán Eterovic. Implementing Distributed Mutual Exclusion on Multithreaded Environments: The Alien-Threads Approach. In *Fifth IEEE International Symposium and School on Advance Distributed Systems ISSADS 2005*, Guadalajara, Jalisco, México, January 2005.