Priority Inheritance and Ceilings for Distributed Mutual Exclusion

Frank Mueller
Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de phone: (+49) (30) 2093-3011 fax:-3010

Abstract

The contributions of this paper are threefold. First, a solution to the problem of prioritized mutual exclusion in a distributed system is proved correct. This protocol is based on fewer requirements than prioritized extensions of other protocols and outperforms other protocols with an average complexity of $\Theta(\log n)$ and a worst-case complexity of $O(n)$ messages for n nodes. Second, the concept of relative fairness *is introduced, which quantifies the relation between parallel events in terms of their ordering in the absence of synchronized clocks. This concept is applied to the protocol in order to determine a requirement to guarantee a certain order between events when message delays are bounded. Third, the protocol is extended to prevent priority inversion by incorporating the priority inheritance and the priority ceiling protocols. The extensions are shown to integrate well with the original protocol. They impose the same message overhead as mentioned before for each dynamically raised priority due to resource contention while early priority boosting, such as for the priority ceiling emulation protocol, requires no additional overhead at all.*

1. Introduction

Common resources in a distributed environment may require that they are used in mutual exclusion. This problem is similar to mutual exclusion in a shared-memory environment. However, while shared memory architectures generally provide atomic instructions (*e.g.,* test-and-set) that can be exploited to provide mutual exclusion, such provisions do not exist in a distributed environment. Furthermore, commonly known mutual exclusion algorithms for shared-memory environments that do not rely on hardware support still require access to shared variables.

In a distributed environment, mutual exclusion has to be provided via a series of messages passed between nodes that share a certain resource. Several algorithms to solve mutual exclusion for distributed systems have been developed [5]. They can be distinguished by their approaches as token-based and non-token-based. The former ones may be based on broadcast protocols or they may use logical structures with point-to-point communication. The use of such protocols in real-time systems requires additional support for priorities and should address the requirements imposed by real-time schedulability theory.

Real-time scheduling provides a theory for uniprocessor allocation of tasks and provides guarantees to meet deadlines in the presence of resources [13, 1, 22, 2, 3, 23]. Even for multiprocessor environments with shared memory certain guarantees can be provided if tasks with shared resources are scheduled on the same processor. For common resources in multi-processors, on the other hand, real-time scheduling approaches often utilize heuristic methods since the general problem of distributed scheduling is NP-complete [21]. Heuristic algorithms have also been proposed for distributed systems to select nodes for tasks when they become ready and consider the local availability of shared resources [20]. Another approach assumes that tasks are independent but subtasks may be scheduled on different processors while the communication between subtasks can be modeled as a task by itself [24]. Rajkumar *et al.* describe a method to schedule tasks statically assigned to processors in a shared-memory system but they allow proxy execution of critical sections to handle requests for remote resources [19]. These approaches limit the general model of shared resources while we make no assumptions about the location of resources or task interdependencies. We focus on a protocol to acquire resources in a distributed environment, bound the message overhead and provide support to prevent priority inversion so that this protocol can be used in arbitrary distributed scheduling methods for real-time systems. Thus, this work is applicable but not limited to rate-monotonic priority assignments [13] for static or dynamic priority preemptive scheduling where the highest priority task ready to run is being executed on each processor.

We first describe a protocol to provide mutual exclusion in a distributed environment that supports priority queuing [16]. The protocol uses a token-passing approach with point-to-point communication along a logical structure. Each node keeps a local queue and records the time of requests locally. These queues form a virtual global queue ordered by priority and FIFO within each priority level with regard to the property of relative fairness introduced in this paper. We neither use a global logical clock nor timestamps to avoid the associated message-passing overhead.

The main purpose of our protocol is to provide an efficient solution to the problem of mutual exclusion in distributed systems with known average and worst-case behavior and to support strict priority scheduling for real-time systems at the same time. Thus, the worst-case behavior of the algorithm can be determined for known bounds on message delays. We are also aiming at enhancing distributed programming environments by priority support in a POSIX-like manner [25], *e.g.*, within our DSM-Threads environment [14, 15]. The resulting algorithm works asynchronously in out-of-order message systems and supports nested critical sections. After the description of the protocol, its message overhead is analyzed and its correctness is shown. Furthermore, the protocol is extended to support priority ceiling and inheritance in a distributed environment.

2. The Model

First, we assume a fully connected network (complete graph). Network topologies that are not fully connected can still use our protocol but will have to pay additional overhead when messages from A to B have to be relayed via intermediate nodes. Second, we assume reliable message passing (no loss, duplication, or modification of messages) but we do allow out-of-order message delivery with respect to a pair of nodes, *i.e.*, if two messages are sent from node A to node B, then they may arrive at B in a different order that they were sent from A. Our assumption is that local operations are several orders of a magnitude faster than message delivery since this is the case in today's networks and the gap between processor speed and network speed still seems to widen. Thus, our main concern is to reduce the amount of messages at the expense of local data structures and local operations. For the property of *relative fairness* for priority requests (introduced in Definition 1), we also assume bound message delays, *i.e.*, deterministic latency and throughput characteristics of the communication medium.

3. Token-based Mutual Exclusion

In token-based algorithms for mutual exclusion, a single token is passed between nodes within the system for a lock object [18]. Possession of the token represents the right to enter the critical region of the lock object. Requests that cannot be served right away are registered in a distributed linked list originating at the token owner. Once a token becomes available, the owner passes it on to the *next* requester within the distributed list. In addition, nodes form a logical tree pointing via probable *owner* links towards the root. Initially, the root is the token owner. When requests are issued, they are guided by a chain of probable owners to the current root. Each node on the propagation path sets its probable owner to the requester, *i.e.*, the tree is modified dynamically.

In Figure 1 the root T holds the token for mutual exclusion.

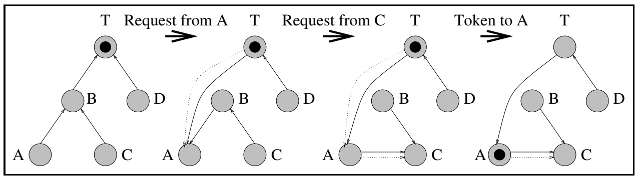

Figure 1. Unprioritized Example

The request by A is sent to B following the probable owner (solid arcs). Node B forwards the request along the chain of probable owners to T and sets its probable owner to A. When the request arrives at the root T, the probable owner and a *next* pointer (dotted arc) are set to the requester A. The next request from C is sent to B. B forwards the request to A following the probable owners and sets its probable owner to C. Node A sets its *next* pointer and probable owner to C. When the token holder T returns from its critical section, the token is sent to the target node that *next* points to. Hence, T passes the token to A and deletes the *next* pointer.

3.1. Priority Queuing Support and Local Queues

In a prioritized environment, requests should be ordered first by priority and then (within each priority level) FIFO. Assume that a request by node N is issued at priority $\Im(N)$. The basic idea for priority support is to accumulate priority information on intermediate nodes during request forwarding. Thus, if the request by N passes along a node I, the probable owner for requests with priority $\leq \Im(N)$ is set to N. (Higher priorities denote more important requests.) We utilize an *owners* set of a node to aggregate the last request seen for each priority level (see Figure 2). We also observe that a high priority request would have to be inserted in the middle of the distributed *next* list, which causes race conditions when multiple requests are in transit. Our protocol replaces the *next* list by a *queue* of locally stored requests. Each request seen by a node is stored in this local *queue* with its request time (reqt)

```
owners=(start, top);      // set of (node, prio), sorted by prio
queue=ϕ;                  // set of (node, prio, rect, uset, reqt)
                          // in local Q, sorted 1) prio 2) uset 3) rect
requests=ϕ;               // set of (node, prio, t) of local requests
kill_list=ϕ;              // set of (node, node, prio, reqt) to delete
req_done=ϕ;               // set of (node, reqt) for received kill lists
critical_section=ϕ;       // (node, prio, t) set in critical section
token_available=(self==start);      // token on self unused?
cond = COND_INIT;         // signal if token locally available
kill_cond = COND_INIT;    // signal if new kill list available
IF token_available THEN owners = ϕ;
```

Figure 2. Node's Local State and Initialization

and the entries are ordered first by decreasing priority, second by decreasing usage times (uset) and third by increasing receive time (rect). When the token arrives at a requesting node, the requests are gradually removed from the *queue* of other nodes. For this purpose, a *kill_list* is employed that contains footprints of locally queued requests that have been served.

The local state of a node and its initialization are summarized in Figure 2 for a lock object. Additional local state is needed to register local *requests*, store requests whose kill list has been received, log activity inside the *critical_section* and determine through the flag *token_available* if the current node (self) both holds the token and nobody is engaged within a critical section. Finally, condition variables *cond* to wait for a token and *kill_cond* to wait for a kill list are required [25]. For the start node, *owners* is cleared since this node posses the token. These conditions are part of a monitor to arbitrate access for concurrent requests on a node.

The example in Figure 3 depicts a sequence of token requests of A, B, C and D with priority 1, 2, 1 and 3, respectively. (Disregard local queues of nodes for now.) The token resides in node T while the requests arrive. For the sake of simplicity, we assume that the initial logical structure is a tree whose edges (directly or indirectly) point to the token owner for all priorities (labeled as top τ). Each request results in new forwarding information for certain priority levels depicted by labeled edges. For example, request $(A, 1)$ results in edges $A \rightarrow A$ labeled ≤ 1, which indicates that future requests received by A with priority ≤ 1 are handled by A. Other edges include $A \rightarrow B(> 1)$, $B \rightarrow T(> 1)$ and $B \rightarrow A(\leq 1)$.

The initial request for mutual exclusion is issued by a *lock* operation. Recall that all operations are part of a monitor controlling concurrent accesses of local tasks on a node. Figure 4 depicts this *lock* operation but omits the lock object parameter to facilitate

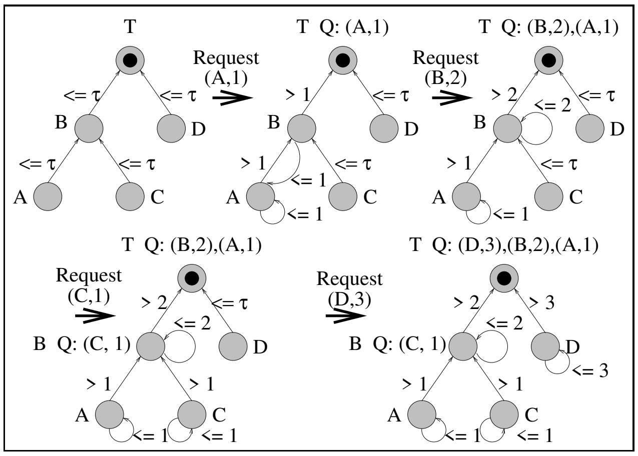

Figure 3. Example of Token Requests

```
PROC lock (prio) IS
    now = read_clock();
    requests += (self, prio, now);
    (R, prioR) = probable_owner(owners, prio);
    owners -= all (X, prioX) st. prioX ≤ prio;
    owners += (self, prio);
    IF ¬ token_available THEN
        kill = (self, self, prio, now);
        queue += (self, prio, now, 0, now);
        send token request (self,prio,now,kill,self) to R;
        WHILE ¬ token_available DO
            wait(cond);
        req_done -= (self, now);
        queue = tail(queue);
    ENDIF;
    token_available = FALSE;
    critical_section = head(requests);
    requests = tail(requests);
    owners = φ;
END lock;
```

Figure 4. Acquiring a Lock

the presentation. The request is locally logged, then the probable owner is determined as the last node whose request was seen with greater or equal priority. The probable owners are updated so that edges with less or equal priority than the current request are replaced by the new request. If the token is not present locally, the request is added to the local queue and a message is sent to the probable owner. This message contains the node, priority, request time, a kill list for the locally queued request and the local node as the first node of the propagation path. After the request is sent, the caller waits for availability of the token. Arrival of the token is realized via *receive_token* discussed later. When the token has become available, the current request can be removed from *queue* and *req_done* since pending information concerning the kill list has been processed. The caller then starts using the token, enters the critical section and removes the request from the list of outstanding local *requests*. The probable *owners* are cleared since any request can always be queued at the token location. Finally, the critical section can be entered.

The *unlock* operation exits the critical section and is depicted in Figure 5. The critical section is cleared. If other requests are queued, forward the token to the highest priority requester. Otherwise, make the token locally available.

When a node R issues a request at priority $\Im(R)$, this request propagates along the chain of probable owners until it reaches a node I with a pending request at priority $\geq \Im(R)$. It is then locally queued at I. The basic idea is that node I has to be served before our request. Thus, R's request may be stored locally until the token arrives. Once the token arrives, R's request should be served after I's if no other requests have arrived since then. This is also depicted in Figure 3, where local queues are associated with nodes T and B. Node B stores C's request at priority one since B has an outstanding request at priority two that will be served before.

Figure 6 depicts the operations invoked upon receiving a re-

```
PROC unlock IS
    (self, prio, t) = critical_section;
    critical_section = φ;
    IF queue ≠ φ THEN
        forward_token();
    ELSE
        token_available = TRUE;
END unlock;
```

Figure 5. Releasing a Lock

```
PROC receive_token_request(R, prioR, reqt, kill, visited_set) IS
    (S, prioS) = probable_owner(owners, prioR);
    queue += (R, prioR, now, 0, reqt);
    kill += (self, R, prioR, reqt);
    IF owners == φ OR S == self THEN
        IF head(queue)==(R,prioR,rect,uset,reqt) THEN
            IF token_available THEN // case 1
                kill_list += kill;
                forward_token();
            ELIF critical_section==wait_for_kill THEN // case 2
                kill_list += kill;
                signal(kill_cond);
            ELSE // case 3
                kill_list += kill;
        ELSE // case 3
            kill_list += kill;
    ELIF S ∈ visited_set THEN // case 4
        kill -= (R, R, prioR, reqt);
        send kill list (kill, reqt) to R;
        owners -= (X, prioR);
        owners += (R, prioR);
    ELSE // case 5
        visited_set += self;
        send token req(R,prioR,reqt,kill,visited_set) to S;
        owners -= (X, prioR);
        owners += (R, prioR);
END receive_token_request;
```

Figure 6. Receiving a Request

quest. Determine the probable owner in case the request needs to be forwarded. Then, log the request in the local queue and, due to the additional queue entry, add a corresponding element to the request's kill list. Several cases have to be distinguished:

1. If the node holds an available token that has not been requested, then forward the token to the current requester. Merge the local kill list and the kill list of R's request.
2. While waiting for the kill list, a higher priority request from R caught up with the token location. Merge the local kill list and the kill list of R's request. By signaling the kill condition, the token will then be forwarded to R within *receive_token* (see below).
3. If the probable owner is the current node, then the current request has been queued locally and it suffices to merge the local kill list and the requester's kill list.
4. If the probable owner had been seen on the propagation path, then avoid cycles in requests by removing the initial request of the requester from the kill list and send the kill list to the requester. Replace the probable owner at the requesting priority with the requester.
5. Otherwise, log the current node in the propagation path to later avoid cycles. Then forward the request to the probable owner. Replace the probable owner at the requesting priority with the requester.

A subtable problem with a set of distributed queues is the handling of FIFO queuing at the same priority level. Consider T sending the token to I. Node T has an entry for node I at the head of its local queue and possibly more entries at the same or at a lower priority. When I receives the token, the queue of T is piggybacked. Node I then has to merge its local queue and T's queue. However, if two entries have the same priority in different queues, which entry should be served first?

In a distributed system, FIFO policies can be enforced by a global logical clock via timestamps [12, 8]. However, timestamps result in additional message-passing overhead for communication with processes that never request a certain lock and overhead for acknowledgments. Instead, we utilize *local time* facilities to enforce FIFO ordering. Notice that local time only has a meaning on the current node. However, when the token is transmitted from T to I and T's queue is piggybacked, then the first request in T's queue is a request by node I. If all requests by I log the local initiation time, we are able to measure the time between request initiation $t_I(req)$ and token reception on the local node (now). This interval is called the request latency $t_I(latency)$ (see Figure 7). The latency itself is composed of the request transit time $t_I(req\_transit)$, the token transit time $t_I(token\_transit)$ and the token usage time $t_I(usage)$.

$$t_I(latency) = \\ t_I(req\_transit) + t_I(token\_transit) + t_I(usage)$$

Index I may be replaced by other indices for other nodes. Notice that the queue piggybacked with a token message contains local requests *and* the accumulated token usage time. The token usage time can be calculated as the time spent in critical sections on

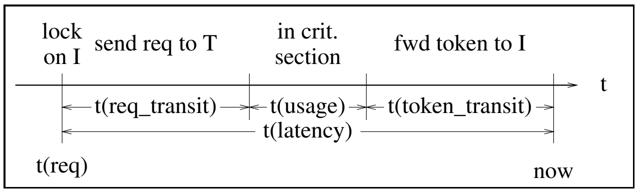

Figure 7. Events/Times for Token Requests

nodes holding the token since request reception. Thus, latency and usage can be measured but the transit times may only be inferred indirectly due to a lack of global clocks.

We can now determine the approximate usage times for nodes X within the local queue due to the following observation. The local queue contains requests from the current node and possibly other nodes with the associated reception time $t_X(receipt)$ of each request by the local node. Thus, the usage time for local requests from X is determined relative to the request by I on the head of the piggybacked queue:

$$t_X(new\_usage) = t_X(usage) + \\ (now - t_X(receipt)) * (t_I(usage)/t_I(latency))$$

The usage time on the current node for request X is estimated as the portion of time since receiving request X relative to I's latency. Consequently, X's local usage time is smaller than I's accumulated usage time, which is consistent since X was received after I had been issued. Once the usage times have been determined for local requests, the local queue can be merged with the piggybacked queue from T, which already contains accumulated usage times. Requests are ordered by descending priority and, within each priority level, by descending usage time. Thus, the usage time provides a means of aging that guarantees requests to be eventually served. This avoids starvation of requests, discussed in more detail in section 5.3. The reception time of all requests is set to the current local time after merging. Notice that the usage time provides a good heuristic for FIFO ordering as long as message latency is constant within the network. The example below even shows that the heuristic is independent of number of hops for requests by nodes X and I, even when the two requests have a different number of hops.

Figure 8 depicts a temporal sequence of exchanged messages for the sequence of requests of Figure 3. We will focus on the

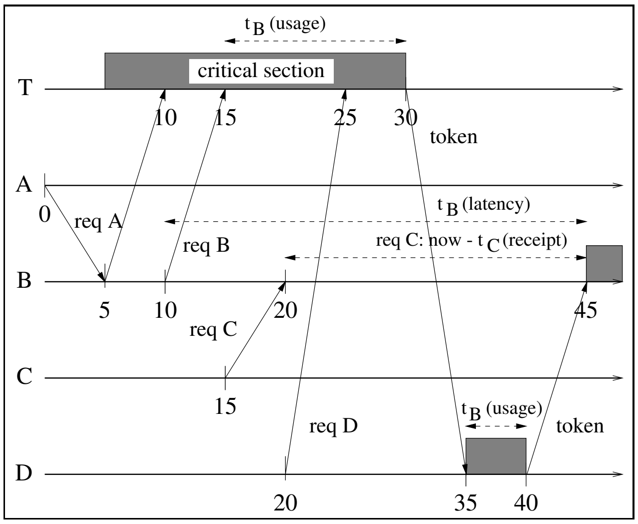

Figure 8. Example of Times for Token Usage

arbitration of the requests from A and C, both issued at the same priority level. After requests from A and B are forwarded to T, request C stops propagating at B. The request from D arrives at T before T leaves the critical section. Upon availability of the critical section, the token is sent to D. Notice that the usage time of A's request at this point is 20 (now minus receive time of the request). On D, the token is used another 5 units in a critical section before the token is passed to B. On B, the usage time of C's request is calculated according to the above formula:

$$t_C(new\_usage) = t_C(usage) + \\ (now - T_C(receipt)) * (t_B(usage)/t_B(latency)) \\ = 0 + (45 - 20) * (20/35) = 14.3$$

The latency of B is the current time (45) minus the request time (10). The estimated usage time for C of 14.3 is close to the actual value of 15 as depicted in the Figure. Since A's usage time (25) exceeds C's usage time (14.3), A will use the token before C. This is consistent with the FIFO requirement for requests at the same priority since A's request was issued before C's request.

Figure 9 depicts the actions for receiving the token. Upon receiving the token with a global queue of requesters and a global kill list of served requests, entries are deleted from the local queue and the global kill list if they are present in the latter. Should there be an entry in the queue for a request originating at the current node, then a corresponding entry is added to *req_done* since the kill list was received. Next, global and local kill lists are merged. The merger of the global and local queues involves sorting by priority, usage time and then request time. However, the usage time of local queue entries is approximated according to the above formula. The receive time of all (merged) entries is set to the current time to allow calculation of the local usage time once the token is passed on. The list of probable owners is cleared since any requests arriving at the token can be queued locally and will follow the token. If the next request to be served was issued by the current node but the kill list is still pending (discussed later in the context of Figure 11b), then wait for transmission of the kill list. If the next request to be served still remains the one from the current node, then make the token available and wake up the caller of *lock*. Otherwise, forward the token to the highest priority requester.

The token may be used to grant access in mutual exclusion on the local node. During token usage, other requests may be received, which are then logged with a zero usage time and the cur-

```
PROC receive_token(global_queue, global_kill_list) IS
    FORALL (self, X, prioX, reqt) ∈ global_kill_list DO
        queue -= (X, prioX, rect, uset, reqt);
        global_kill_list -= (Y, X, prioX, reqt);
        IF X == self THEN
            req_done += (X, reqt);
    OD;
    kill_list += global_kill_list;
    (self,prio,t_rec,t_usage,t_req) = head(global_queue);
    t_latency = now - t_req;
    FORALL (X, prioX, rect, uset, reqt) ∈ queue DO
        uset += (now - rect) * (t_usage / t_latency);
    queue += global_queue;
    FORALL (X, prioX, rect, uset, reqt) ∈ queue DO
        rect = now;
    owners = ϕ;
    (R, prioR, rect, uset, reqt) = head(queue);
    WHILE R == self ∧ (self, reqt) ∉ req_done DO
        critical_section = wait_for_kill;
        wait(kill_cond);
        cirtical_section = ϕ;
        (R, prioR, rect, uset, reqt) = head(queue);
    OD;
    IF R == self THEN
        token_available = TRUE;
        signal(cond);
    ELSE
        forward_token();
 END receive_token;
```

Figure 9. Receiving the Token

rent local time as a reception time. These requests are merged into the existing queue using the same precedence rules as before (descending priority and descending usage time within each priority) and, in addition, with descending reception time (within the same priority level and the same usage times).

3.2. Token Forwarding

Once the token is no longer being used on the local node, it may be forwarded to the first requester within the queue. The queue is also piggybacked with the forwarded token but usage times are recalculated before the forwarding to include the local usage time:

$$t_X(new\_usage) = t_X(usage) + now - t_X(receipt)$$

The example of Figure 3 is continued in Figure 10. The token is forwarded to nodes D and B. The previous token holder retains only one link to the next token holder for requests at any priority level. Notice that the figure does not depict usage times and their adjustment upon token forwarding.

When a request propagates along a path within the logical tree, a request from node R propagates along intermediate nodes I before reaching the token holder T, as depicted in Figure 11a. Token forwarding may, however, result in a race condition that can be characterized as follows. A request from a node R may pass intermediate node I, where another lower priority request had already been issued. If the token is sent from T while request R is in transit and has already passed I, a race between the request and the token location exists. For example, in Figure 10 consider a request $(A, 4)$ being issued and passed *via* B and T. Meanwhile, T has already forwarded the token to D and D has forwarded it to B. Hence, the new request would trail behind to D and B, the latter being a node the request had passed once before (forming a cycle).

Returning to the abstract characterization of the race condition, the communication overhead may no longer be bound by the number of nodes in the system since the request from R may pass repeatedly by I. We avoid this situation by adding request R to the local queue of all intermediate nodes I. Should the token be passed to I before request R reaches T (but after passing by I), node I forwards the token to the higher priority requester R before using it (see Figure 11b). Request R logs all intermediate nodes on the way and stops propagating when an edge along the probable owner path leads to an already logged node. Thus, request R stops propagating at T since T's probable owner is I (T sent the token to I) and the request had already passed I. In addition, the kill list of entries added to local queues on intermediate nodes is passed to the token requester R. R ensures that the request for mutual exclusion is only granted after this kill list has been received.

The added request on intermediate nodes will be processed when the token reaches the intermediate node the next time. There are two sub cases. Either the token passes by intermediate node X before reaching R, for example node I in Figure 11b. Then, the request will be merged with other pending requests and it is locally recorded that the request has been served.

In the other case, the token passes by intermediate node X after reaching R. The kill list indicates the entries to be removed from local queues of all intermediates nodes. This list is received before or while R holds the token (indicated by the dashed edge in Figure 11b). When the token reaches a node in the kill list, entries of the local queue are removed if the kill list indicates this.

Submission of a kill list is also necessary as a result of a request by R that stops propagating when a higher or equally high priority request is pending at an intermediate node I (on the path to the token) whose probable owner is I. The request from R is queued locally on all intermediate nodes on the way to I including node I. In addition, the kill list sent to R includes all intermediate nodes up to I for this request. The token may now either be received by I, where it is immediately forwarded to R. Or another node on the way to I receives this token before I since I's request may have been issued very recently. In this case, the token is also forwarded to R by this other node, possibly even before reaching I. The kill list guarantees that all entries for R's request are eventually deleted from local queues of intermediate nodes, regardless of the location of the node between R and I.

The operations for forwarding the token are depicted in Figure 12. First, the token usage times for queued requests are added to the already accumulated usage times. Then, the token is sent along with the queued requests and the kill list to the highest priority

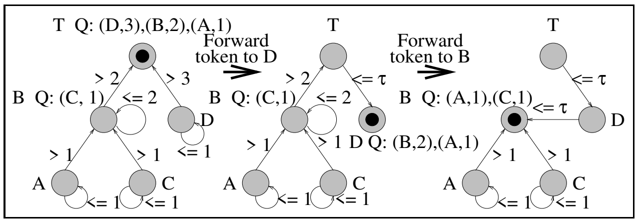

Figure 10. Example of Token Forwarding

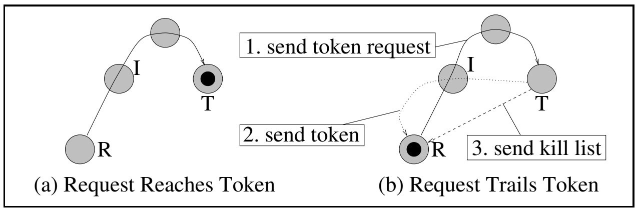

Figure 11. Model of Token Forwarding

```
PROC forward_token() IS
    FORALL (X, prioX, rect, uset, reqt) ∈ queue DO
        uset += now - rect;
    send token(queue, kill_list) to node(head(queue));
    kill_list = owners = φ;
    FOREACH (X, prioX, rect, uset, reqt) ∈ queue DO
        owners -= (Y, prioX);
        owners += (X, prioX);
    UNTIL X == self OR end of queue OD;
    queue = φ;
    token_available = FALSE;
END forward_token;

PROC receive_kill_list(kill, reqt) IS
    kill_list += kill;
    req_done += (self, reqt);
    signal(kill_cond);
END receive_kill_list;
```

Figure 12. Forwarding the Token

requester. The local kill list is cleared and the probable owners are reduced to the last entry in the *queue* of each priority level before the local queue is cleared as well. The token is marked as not available anymore. Figure 12 also depicts the actions for receiving the kill list. The received list is merged with the local kill list. The request by the current node with the original request time is added to *req_done*. Finally, the local receiver of the token is signaled to continue in *receive_token*.

4. Message Overhead

First, we analyze the number of messages necessary to register a single request for mutual exclusion. In an environment with n nodes, the algorithm requires an average of $\Theta(log\, n)$ messages for requests at one priority level since the algorithm uses a logical tree similar to the algorithm by Naimi *et al.* [18]. In fact, the prioritized algorithm reduces to an improved version of Naimi's algorithm when only one priority level is utilized. In the worst case, the links between nodes form a chain such that $n-1$ messages are required for the propagation of the request. This should be obvious from the last section, where we ensured that token requests cannot propagate in cycles, in other words, they cannot pass a node twice. There are at most two more messages required, one to send the token from an intermediate node to the requester and one to send the kill list (see Figure 11b). In the example, the token message from T to I is excluded from the message overhead of R's request since this message was caused by a different request, namely I's.

Even for multiple simultaneous requests, the worst case would occur when $n-1$ requests are issued at different priority levels in a chain of probable owners rooted in the token owner and increasing in priority with the distance from the root. If we assume the lowest priority request reaches the root first (after one hop) and all other requests reach the root in k hops (where k is the initial distance from the root) while the token is being transfered from the root to the lowest priority requester, the total number of messages would be $\frac{n(n-1)}{2}$ messages. Two additional message would be needed for each request to forward the token and send the kill list starting with the highest priority request at the head of the local queue of the root's neighbor. Thus, a quadratic number of messages would be issued before the token can be used by the highest priority requester. However, the highest priority request by itself still resulted in $n+1$ messages. All other messages resulted from different requests and were issued simultaneously for this worst-case scenario. Thus, we conclude that the message overhead may at worst be linear while the number of simultaneous messages may at worst be quadratic before a token can be used. The latter observation may impact the utilization of a shared communication medium for all nodes, such as Ethernet.

Our algorithm requires more overhead in terms of local bookkeeping than the algorithm by Naimi *et al.* In addition, we have to piggyback data structures with requests and token sends. Nonetheless, if the amount of messages is the main cause of overhead, as we assumed, our algorithm is the most efficient implementation of distributed mutual exclusion with priorities that we know of.

5. Correctness

We will give an informal proof of the correctness of the algorithm by first showing that mutual exclusion is satisfied, then that deadlocks cannot occur and finally that requests cannot starve in the absence of higher priority requests. These proofs take advantage of the observation that a request can be identified uniquely by its source node together with its request time since this information never changes after issuing the request. (We assume non-synchronized local clocks that return values which are strictly monotonically increasing.)

5.1. Mutual Exclusion

In the initial state, only one node, the start node, has the token. The token may be passed during an *unlock*, a *receive_token*, or a *receive_token_request*. In all cases, the token is passed only to one node and the current node loses it. A process may only enter a critical section when the node holds the token. Thus, mutual exclusion is guaranteed between nodes. Within a node, local tasks (threads) may compete for a critical section but the monitor semantics ensure than only one task can enter the critical section since a task may proceed only past the *ENDIF* in *lock* if *token_available* is true and will then set the flag to false before entering the critical section (and exiting the monitor).

5.2. Absence of Deadlock

In the following, we restrict ourselves to token requests from nodes other than the token holder. We have to ensure that if the token is available, a request will eventually be known to the token holder. Once the token holder knows of at least one request, the token is forwarded to the first request in the local queue by invoking *forward_token*. Finally, circular dependencies due to token forwarding must not occur. The property that a request becomes known to the token holder is shown by the following three lemmas.

Lemma 1 *For a requesting source node R (other than the token holder) with priority $\Im(R)$, there is a chain of owners leading either to the token holder or to another requester with priority $\geq \Im(R)$.*

Proof: Initially, all nodes without the token point to the token holder T via their owner sets. Assume that the lemma holds at some point. If R issues a request at $\Im(R)$ and $R \neq T$, then the request will propagate either to T or to an intermediate node I with outstanding request at priority $\geq \Im(R)$. Node R and all nodes N between R and T (or I) forward the request to a next node in the chain. In addition, they (and nodes I or T) add the entry $(R, \Im(R))$ to its owners. Thus, all nodes along the traversed path form a chain leading to node R at priority $\Im(R)$. A node X that was not traversed may either point (directly or indirectly) into the middle of this new chain, in which case its path also leads to R for $\Im(R)$. Or, alternatively, node X never leads to this new chain. Then, the old chain is unmodified and still points to T. If T had already received R's request (or another request from node S) and sent the token to R (or S), the chain now extends further from T to R (or S), the new token holder. On the other hand, if T still holds the token (and R's request is in transit), then the chain still ends in T, the token holder.

Lemma 2 *For request R with priority $\Im(R)$, if a chain of owners from R to I does not lead to the token owner T, there is at least one node N on this chain (R and I inclusive) that has a request in transit toward T or N is in T's local queue of requests.*

Proof: This follows immediately from the last proof and the fact that a request is put into the queue in *receive_token_request*.

Lemma 2 implies that we restrict our discussion to requests at the same priority level from here on.

Lemma 3 *If request R is in transit to the token holder T, it will be queued with the current or a future token holder or an intermediate node N that the request traversed.*

Proof: If there is a path from R to T and T remains the token holder, then R's request reaches T eventually and is added to the local queue. If the token is passed on from T, the path will extend from T to the new token holder; R's requests will keep propagating until it reaches the new token holder or until the path leads to a node I that request R has already traversed. In the latter case, the request stops propagating before I (due to the *visited_set*), I is the token holder and R's request is present on I's local queue. If there is no path to T, there is a path to node N who has a request in transit to T according to the last lemma and R's request is queued locally in N. When N receives the token, R's request will be recognized.

Thus, we showed that a request will eventually be recognized by the token holder, even if the token holder changes while the request propagates. In addition, circular forwarding cannot occur since token forwarding can only take place if a higher priority request traverses an intermediate node that requested a lower priority request before. Since we assume a finite number of priority levels, there can only be a finite number of higher priority requests being served via token forwarding.

5.3. Absence of Starvation

Starvation in a protocol with strict priority ordering can be reduced to starvation within one priority level. This is due to the fact that higher priority requests *intentionally* bypass lower priority requests.

We have already showed that a request from R at a certain priority is queued in some local queue at node N. We will also show that the token is eventually sent to N at the same priority level. Once the token is received on N, the local queue is merged with the token queue by calculating the usage time relative to the request by N that caused the token to be forwarded to N. Thus, the initial calculation of the usage time of a request depends on the reception time of this request on the node where it is merged with the token queue. In other words, while request R is in transit to node N, other requests may still be registered before R's request. But once request R has reached N and is included in the local queue, future requests and requests in transit at the same priority are guaranteed to be served after R's request, even in the absence of global time facilities.

For a finite number of tasks (threads) to issue requests there can only be a finite number of requests ahead of R's request (once it is queued at node N). The requests ahead of R may be within any local queues and the token queue so that their number cannot increase once R's request is in N's local queue. The token will be forwarded to the nodes with requests ahead of R before it is finally forwarded to R. Thus, R will be served in a finite time assuming that critical sections take a finite amount of time.

5.4. Strict Priority Scheduling

A strict priority protocol requires that higher priority requests are served before lower ones, even if issued later. In a distributed environment, two events on different nodes do not have a temporal relationship in the absence of global time facilities. However, we established that the worst-case communication overhead for propagating a request is bound by $n-1$ messages. Thus, we can formulate the property of *relative fairness* with respect to the message overhead.

Theorem 1 *If two requests are issued Δ time units apart and the communication overhead for handling a request is at most k messages, bound by $\omega(k)$ time units, then the requests will be served in the same order as issued if $\Delta > \omega(k)$.*

Proof: If the first request is issued at $t(0)$ and takes k messages, it would be handled at $t(1)$ with $t(1) \leq t(0) + \omega(k)$. The second request arrives at $t(2)$ with $t(2) \geq t(0) + \Delta$. Since $\Delta > \omega(k) \Rightarrow t(0) + \Delta > t(0) + \omega(k) \Rightarrow t(2) > t(1)$.

Definition 1 *An algorithm guarantees* **relative fairness** *if any two requests issued Δ time units apart are served in the same order as issued.*

The above algorithm is relatively fair if $\Delta > \omega(n-1)$ since $n-1$ is the worst-case message overhead for propagating a request. Notice that "handling" a request means registering the request in this case since once a request stops propagating (being in transit), future requests (including those in transit) will be served later, as discussed in the context of starvation. This property can be used during off-line schedulability analysis to ensure FIFO processing of requests when required. The message size is bound by the transmitted queue and the kill list. The former is bound by the number of tasks (threads) in the entire system, the later is bound by the number of nodes times the number of tasks. For a communication medium with bound message delay (deterministic latency and throughput characteristics), the overhead can be calculated based on these bounds. Thus, two requests are guaranteed to be served according to strict priority ordering if they are issued at least Δ time units apart.

6. Preventing Priority Inversion

Two basic methods for preventing priority inversion can be incorporated into the protocol for distributed mutual exclusion. First, early priority boosting shall be discussed in the context of the priority ceiling emulation protocol (PCEP). Second, the distributed protocol is enhanced by dynamic priority changes due to resource contention to support the priority inheritance protocol (PIP). For both cases, changes to the distributed protocol are identified. In the following, we use the term *base priority* to refer to the priority of a task that does not hold any resources. The *effective priority* is the current priority of a task considered for scheduling as a result of the protocols to prevent priority inversion.

6.1. Priority Ceiling Emulation Protocol (PCEP)

Priority ceiling protocols in general require that a priority value, the *ceiling*, be associated with a resource and the corresponding lock. This ceiling is defined as the maximum priority of tasks contending for the resource. The original priority ceiling protocol (PCP) determines the ceiling off-line for static priority scheduling [22]. The dynamic priority ceiling protocol (DPCP) determines the ceiling on-line at the time of resource contention for dynamic priority scheduling [7]. Both PCP and DPCP only grant a resource to a requester if its priority is greater than the ceilings of all resources in use by other tasks. Should a task block due to this rule, its priority is inherited by the task that holds the requested resource. While both protocols avoid chained blocking (such as induced by the transitivity property of PIP), they still require the following properties:

DPCP only: The state of all resources must be inspected to determine the maximum ceiling value of held resources.

PCP and DPCP: A task's priority may be changed during execution of a critical section.

The first property cannot be realized efficiently in the absence of shared memory. In a distributed system, a number of messages had to be exchanged to collect information about global state while any other activities potentially modifying this state had to be suspended. Thus, DPCP is not suitable for distributed environments. The second property may result in multiple priority changes while holding a resource. This property can be realized by extending the protocol for distributed mutual exclusion for re-issuing requests, which will be discussed in the context of the priority inheritance protocol. Thus, PCP is suitable for distributed environments and can be realized by extensions of the distributed protocol.

A third variant, the priority ceiling emulation protocol (PCEP), offers an even easier solution. This protocol combines the general idea of the original PCP [22] with early priority boosting to the ceiling level upon granting resource requests as proposed in the stack resource protocol [2]. It also requires that ceilings be computed statically (as for PCP). When granting a resource, PCEP boosts the priority of a task to the maximum of its effective priority and the ceiling of the resource. When releasing a resource, PCEP lowers the effective priority to the maximum of the base priority of the task and the ceilings of the remaining resources held by the task. The protocol avoids chained blocking, priority changes while holding a resource and may reduce the number of context switches. On the other hand, it may experience "false blocking" in situations where the previous protocols would not have blocked. The most significant advantage of PCEP is its simplicity in terms of implementation so that PCEP has been adopted by industry standards such as POSIX and Ada95 [25, 11]. PCEP can be used with minimal changes to the previously introduced protocol of distributed mutual exclusion. The distributed protocol supports priorities at the time of issuing requests. The PCEP raises the priority of a task holding a resource. The distributed protocol is extended as follows (see bold-face extensions in Figure 13):

1. Boost the requester's priority upon acquiring the resource in *lock* or *receive_token* as described before.
2. Lower its priority upon releasing the resource in *unlock* as described before.

These modifications do not require any additional message overhead. The resulting protocol provides the most efficient solution to the priority inversion problem in a distributed environment with static priorities to our knowledge.

6.2. Priority Inheritance Protocol (PIP)

The PIP protocol requires that the priority of a task holding a resource be boosted to some value upon detection of contention for this resource [22]. The protocol also requires transitive boosting when task A requests resource R held by task B and task B has issued a request for resource S held by task C. This situation is commonly known as chained blocking. In this case, C will inherit A's priority (if greater than C's effective priority) since task C causes A to be blocked indirectly though B. Transitive priority changes may cause a chain of priority boosts along the contention dependencies of resources.

```
PROC lock IS
    ...
    IF ¬ token_available THEN
        ...
    ELSE // instead of ENDIF
        prio = max{base_prio, ceiling_prio(new lock)};
    ...
END lock;

PROC receive_token(global_queue, global_kill_list) IS
    ...
    IF R == self THEN
        token_available = TRUE;
        prio = max{base_prio, ceiling_prio(new lock)};
        signal(cond);
    ...
END receive_token;

PROC unlock IS
    ...
    IF prio ≠ base_prio THEN
        prio=max{base_prio,ceiling_prio(other locks)};
END unlock;
```

Figure 13. Extensions for Ceiling Emulation

In a distributed environment, contention-based priority boosting requires that requests for resources are re-issued at the inherited priority. This serves the following purposes:

1. The priority of the previous request is modified within all nodes that handled the original request.
2. The priority of a task holding the resource is boosted (if necessitated by higher priorities).
3. Pending requests of a task holding the resource are re-issued (if necessitated by higher priorities).

A request can be re-issued in the existing distributed protocol at a different priority simply by re-using its original request time. The re-issued request is required to follow the propagation path along the same nodes as the original requests allowing intermediate nodes to modify their notion of this request at the re-issued priority. This requires the following extensions to the distributed protocol shown in bold face in Figure 14, where L is the lock object for the current resource:

receive_token_request: Additional operations are required prior to the regular protocol:

- If the request was re-issued, then update the local state for the current resource and re-issue the request to the same node unless the token is present (check upon activity within the critical section).
- If a local task is executing the critical section, then boost the priority of local task to the maximum of its effective priority and the requester's priority. In addition, if the local task blocked on another resource then update the local state for the current and the other resource, re-issue the request for the other resource and send it to the same node.

unlock: If the task's priority was boosted, lower it to the maximum of the base priority and of all requesting priorities for

```
PROC receive_token_request(R, prioR, reqt, kill, visited_set) IS
    IF request re-issued THEN
        update(L, R, prioR, rect, uset, reqt);
        IF critical_section == φ THEN
            re-send token request(self, prioR, 0, 0, reqt);
    IF critical_section == (self, prio) THEN
        T = task waiting for L;
        prio=max{eff. prio, prioR} of T waiting for L;
        IF T awaits L at prioL ∧ prioL < prioR THEN
            update(L, self, prioR, rectL, usetL, reqtL);
            re-send token req(self,prioR,0,0,reqtL)for L;
    IF ¬ request re-issued THEN
        ... // all previous operations of ELSE part
END receive_token_request;

PROC unlock IS
  ...
  IF prio ≠ base_prio THEN
    prio=max{base_prio,requesting_prios(held locks)};
END unlock;

PROC update(L, R, prioR, rect, uset, reqt) IS
    WITH L DO
        in queue raise (R, prioX to prioR, ...);
        in kill raise (self, R, prioX to prioR, reqt);
        update owners: delete ≤ prioR, add (R, prioR);
END update_resource_state;
```

Figure 14. Extensions for Inheritance

resources still held by the task (by inspecting the local queue of requests for each resource).

Updating the local state includes raising the priority to the new level for requests already present in the queue and the kill list. Probable owners are updated accordingly. These operations are performed wrt. to lock object L protecting a certain resource.

The modifications for PIP require a message overhead of one additional request for each re-issued request, *i.e.*, the overhead is bounded by $O(n)$ messages. For chained blocking, each request in the chain is re-issued. Let c be the length of the chain. Then, the message overhead is bounded by $O(c * n)$. Notice that c is not a constant and one upper bound is given by $min(t, r)$ where t and r denote the number of tasks and resources, respectively. A tighter upper bound of c can typically be determined by analyzing resource dependencies. Let $RDG(R, E)$ be the resource dependency graph where resources R are vertices and edges $E : S \rightarrow T$ denote that resources S and T are nested such that S must be held before T is acquired. The RDG represents all possible concurrent resource dependencies. The graph does not need to be connected, *e.g.*, when certain resource usage is not nested. For an acyclic RDG, let h be the maximum height of all trees in the graph. Then, the maximum chain length c is bounded by $h - 1$. Notice that if the RDG was not acyclic, deadlocks could occur. Furthermore, the RDG could be relaxed by allowing multiple instances of resources to represent dependencies that are disjoint in time, *i.e.*, dependencies that cannot occur concurrently (thereby avoiding cycles in the RDG that actually cannot occur in practice).

While $h - 1$ provides a bound for all resources, schedulability analysis typically includes the calculation of blocking times on a per-task basis. The resource dependencies corresponding to a certain request of a task can be approximated even tighter when the corresponding instance of the resource in the RDG is located. Let d by the distance of this resource node to its root in the subtree. Then, d bounds the maximum number of re-issues for this particular resource requests and $(d + 1) * (n + 1)$ messages can be considered for the calculation of the blocking time. This includes an initial request and d re-issued requests, each of which take (up to) $n - 1$ messages to propagate the request and 2 messages to deliver the token and the kill list, respectively. Bounding c by d typically yields much tighter static estimate for the message overhead than the previously mentioned alternatives. The resulting protocol provides a solution to the priority inversion problem in a distributed environment with dynamic priorities.

7. Related Work

A number of algorithms exist to solve the problem of mutual exclusion in a distributed environment. Goscinski [10] proposed a prioritized algorithm based on broadcast requests using a token-passing approach. Chang [4] developed extensions to various algorithms for priority handling that use broadcast messages or fixed logical structures with token passing. Chang, Singhal and Liu [6] use a dynamic tree similar to Naimi *et al.* [17, 18]. In fact, the only difference between the algorithms seems to be that the root of the tree is piggybacked in the former approach while the latter (and older one) does not use piggybacking. Due to the similarity, we simply referred to the older algorithm in this paper. Other mutual exclusion algorithms (without token passing) employ global logical clocks and timestamps [12]. These algorithms can be readily extended to transmit priorities together with timestamps. However, all of the above algorithms, except Raymond's and Naimi's, have an average message complexity larger than $\Theta(log\, n)$ for a request. Finally, Raymond's algorithm uses a fixed logical structure while we use a dynamic one to allow the addition and deletion of nodes. Furthermore, Raymond needs an average of $\Theta(log\, n)$ messages to send the token to a requester, where our algorithm only requires either one or two messages (at the same priority level). The modified version of Raymond's algorithm by Fu *et al.* [9] is, in its essence, similar to our local queues but with just one entry. Finally, all of the previous algorithms use synchronous message passing and assume a single request per node at a time while our algorithm works asynchronously with multiple requests per node to provide more concurrency within a multi-threaded environment. None of the algorithms have been studied wrt. the priority inversion problem.

A number of approaches exist to address real-time scheduling for multi-processors and distributed systems. An optimal algorithm for non-preemptive scheduling with precedence constraints and exclusion relations is quite powerful [26]. Xu's work requires that exclusion constraints be specified for segments (section of a task's code) corresponding to critical sections. The constructed off-line schedule then ensures that a segment is executed at a time when no other segments specified in the exclusion set may execute at the same time. This constraint is more restrictive than our approach since it requires a fixed off-line ordering of critical sections (segments) while we support preemption and tolerate arbitrary ordering between critical sections, which results in a more flexible schedule. Most of the other approaches make restrictive assumptions, *e.g.*, tasks with shared resources are required to be scheduled on the same processor. Rajkumar *et al.* describe a method to schedule tasks statically assigned to processors in a shared-memory system but they allow proxy execution of critical sections to handle requests for remote resources [19]. For remote resources in multi-processors, real-time scheduling approaches often utilize heuristic methods since the general problem of distributed scheduling is NP-complete [21]. Heuristic algorithms have also been proposed for distributed systems to select nodes for tasks when they become ready and consider the local availability of shared resources [20]. Another approach assumes that tasks are independent but subtasks may be scheduled on different processors while the communication between subtasks can be modeled as a task as well [24]. These approaches differ from the more general model in this paper. We make similar assumptions as Rajkumar *et al.* in the sense that we require static or dynamic priority preemptive scheduling and assume that the highest priority task ready to run is being executed on each processor. Our model works for rate-monotonic priority assignments [13] but is not necessarily limited to just this model. We make no assumptions about the location of resources or task interdependencies. We focus on a protocol to acquire resources in a distributed environment, bound the message overhead and provide support to prevent priority inversion so that this protocol can be used in arbitrary distributed scheduling methods for real-time systems.

8. Conclusion

This paper makes the following three contributions. First, a solution to the problem of prioritized mutual exclusion in a distributed system is proved correct. This protocol is based on fewer requirements than prioritized extensions of other algorithms since it uses dynamic structures to guided prioritized requests, supports asynchronous out-of-order message handling and does not require synchronized clocks. With in average complexity of $\Theta(log\ n)$ and a worst-case complexity of $O(n)$ messages for n nodes, the protocol has the same message complexity as its best known non-prioritized counterparts while previous prioritized protocols have a higher message complexity. Second, the concept of *relative fairness* is introduced, which quantifies the relation between parallel events in terms of their ordering in the absence of synchronized clocks. This concept is applied to the protocol in order to determine a requirement to guarantee a certain order between events when message delays are bounded. Third, the protocol is extended to prevent priority inversion by incorporating the priority inheritance and the priority ceiling protocols. The extensions are shown to integrate well with the original protocol and impose the same message overhead as mentioned before for each dynamically raised priority due to resource contention while early priority boosting, such as for the priority ceiling emulation protocol, requires no additional overhead at all.

References

[1] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *J. of Real-Time Systems*, 8:173–198, 1995.

[2] T. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, Mar. 1991.

[3] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[4] Y. Chang. Design of mutual exclusion algorithms for real-time distributed systems. *J. Information Science and Engineering*, 10(4):527–548, Dec. 1994.

[5] Y. Chang. A simulation study on distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 33(2):107–121, Mar. 1996.

[6] Y. Chang, M. Singhal, and M. Liu. An improved O(log(n)) mutual exclusion algorithm for distributed processing. In *Int. Conference on Parallel Processing*, volume 3, pages 295–302, 1990.

[7] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[8] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, Aug. 1991.

[9] S. Fu, N. Tzeng, and Z. Li. Empirical evaluation of distributed mutual exclusion algorithms. In *International Parallel Processing Symposium*, pages 255–259, 1997.

[10] A. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *J. Parallel Distrib. Comput.*, 9(1):77–82, May 1990.

[11] ISO/IEC. *ISO/IEC 8652L 1995 (E), Information Technology – Programming Languages – Ada*, Feb. 1995.

[12] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.

[13] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[14] F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, Apr. 1997.

[15] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

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

[17] M. Naimi and M. Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. In *Int. Conference on Distributed Computing Systems*, 1987.

[18] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, Apr. 1996.

[19] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, Dec. 1988.

[20] K. Ramamritham and J. A. Stankovic. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, Aug. 1989.

[21] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, Apr. 1990.

[22] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, Sept. 1990.

[23] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer, 1998.

[24] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *Int. Conference on Distributed Computing Systems*, pages 38–45, May 1996.

[25] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[26] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 19(2):139–154, Feb. 1993.