A Comparison of Fast and Low Overhead Distributed Priority Locks*

Theodore Johnson and Richard Newman-Wolfe
Dept. of CIS, University of Florida
Gainesville, Fl 32611-2024
ted@cis.ufl.edu, nemo@cis.ufl.edu

Abstract

Distributed synchronization is necessary to coordinate the diverse activities of a distributed system. Priority synchronization is needed for real time systems, or to improve the performance of critical tasks. Practical synchronization techniques require fast response and low overhead. In this paper, we present three priority synchronization algorithms that send $O(\log n)$ messages per critical section request, and use $O(\log n)$ bits of storage per processor. Two of the algorithms are based on Li and Hudak's path compression techniques, and the third algorithm uses Raymond's fixed-tree structure. Since each of the algorithms have the same theoretical complexity, we make a performance comparison to determine which of the algorithms is best under different loads and different request priority distributions. We find that when the request priority distribution is stationary, the path-compression algorithm that uses a singly-linked list is best overall, but the the fixed-tree algorithm requires fewer messages when the number of processors is small and the load is high (100% or greater). When the request priority distribution is non-stationary, the fixed-tree algorithm is requires the fewest messages when the load is 100% or greater. The double-link algorithm is better when the load is low (less than 100%), or if minimizing execution time overhead is more important than minimizing message overhead.

1 Introduction

Distributed synchronization is an important activity that is required to coordinate access to shared resources in a distributed system. A set of n processors synchronize their access to a shared resource by requesting an exclusive privilege to access the resource. The privilege is often represented as a *token*. Real time systems, or systems that have critical tasks that must execute quickly for good performance, need prioritized synchronization. In priority synchronization, every request for the token has a priority attached. When the token holder releases the token, it should be given to the processor with the highest priority request.

In this paper, we present three distributed priority synchronization algorithms, each requiring $O(\log n)$ bits of storage per processor (the $O(\log n)$ bits are required to store the names of $O(1)$ processors), and $O(\log n)$ messages per critical section entry. The low space and message passing overhead make them scalable and practical for implementation. The processors synchronize by sending and interpreting messages according to a synchronization protocol. We assume that every message that is sent is eventually received. The third protocol also requires that messages are

*We acknowledge the support of USRA grant #5555-19 and NSF grant DMS-9223088

delivered in the order sent. Since all three of the protocols have similar theoretical performance, we implemented a simulation of the algorithms and made a performance study.

Considerable attention has been paid to the problem of distributed synchronization. Lamport [11] proposes a timestamp-based distributed synchronization algorithm. A processor broadcasts its request for the token to all of the other processors, which reply with a permission. A processor implicitly receives the token when it receives permissions from all other processors. Ricart and Agrawala [18] and Carvalho and Roucairol [2] improve on Lamport's algorithm by reducing the message passing overhead. However, all of these algorithms require $O(n)$ messages per request.

Thomas [19] introduces the idea of *quorum consensus* for distributed synchronization. When a processor requests the token, it sends a vote request to all of the other processors in the system. A processor will vote for the critical section entry of at most one processor at a time. When a processor receives a majority of the votes, it implicitly receives the token. The number of votes that are required to obtain the token can be reduced by observing that the only requirement for mutual exclusion is that any pair of processors require a vote from the same processor. Maekawa [13] presents an algorithm that requires $O(\sqrt{n})$ messages per request and $O(\sqrt{n}\log n)$ space per processor. Kumar [10] presents the hierarchical quorum consensus protocol, which requires $O(n^{.63})$ votes for consensus, but is more fault tolerant than Maekawa's algorithm.

Li and Hudak [12] present a distributed synchronization algorithm to enforce coherence in a distributed shared virtual memory (DSVM) system. In DSVM, a page of memory in a processor is treated as a cached version of a globally shared memory page. Typical cache coherence algorithms require a home site for the shared page, which tracks the positions of the copies of the page. The 'distributed dynamic' algorithm of Li and Hudak removes the need for a fixed reference point that will locate a shared page. Instead, every processor associates a pointer with each globally shared page. This pointer is a guess about the current location of the page. When the system is quiescent, the pointers form a tree that is rooted at the current page owner. Trehel and Naimi [21, 20] present two algorithms for distributed mutual exclusion that are similar to the 'distributed dynamic' algorithm of Li and Hudak. Chang, Singhal, and Liu [4] present an improvement to the Trehel and Naimi [20] that reduces the average number of messages required per critical section entry.

When a processor faults on a non-resident page, it sends a request to the pointed-to processor. Eventually, the page is returned and the faulting processor unblocks. The request for the page follows the chain of pointers until it reaches a processor that owns the page (or will own the page shortly). If a processor owns a page and receives a request for it, the processor services the request by returning the page and setting its pointer to the new page owner. If a processor is requesting a page and receives a request for the page, the request is blocked until the processor receives and uses the page. If a processor receives a request for a page, and neither owns nor is requesting the page, the processor forwards the request and changes its pointer to the requestor (who will soon own the page).

Though a request for a page might make $n-1$ hops to find the owner, the path compression that occurs while the hops are being made guarantees that a sequence of K requests for the page requires only $O(n + K\log n)$ messages. However, the blocking that the algorithm requires incurs a $O(n)$ space overhead, to store the identities of the blocked requests.

Raymond [17] has proposed a simple synchronization algorithm that can be configured to require $O(\log n)$ storage per processor and $O(\log n)$ messages per critical section request. The algorithm organizes the participating processors in a fixed tree. The execution of the algorithm is similar to that of the Li and Hudak algorithm. Neilsen and Mizuno [16] present an improved version of Raymond's algorithm that requires fewer messages because it passes tokens directly between

processors instead of through the tree. Woo and Newman-Wolfe [22] use a fixed tree based on a Huffman code. Because the tree is fixed, however, it does not adapt to the pattern of requests in the system. Often, only a small population of the processors make requests for the token. Processors that do not request the token should not be required to take part in the synchronization. Li and Hudak show that their path compression algorithm requires $O(n + K \log q)$ messages if only q of the n processors use the page.

Some work has been done to develop prioritized critical section algorithms. Goscinski [7] has proposed a fully distributed priority synchronization algorithm. However, this algorithm requires $O(n \log n)$ storage per processor and $O(n)$ messages per critical section request. Recent work on multiprocessor priority synchronization algorithms has focused on contention-free algorithms, with algorithms proposed by Markatos and LeBlanc [14], Craig [5], and Johnson and Harathi [8].

Two of our priority synchronization algorithm use the path compression technique of Li and Hudak to achieve low message passing overhead. To avoid the $O(n)$ storage cost of blocking, the algorithm uses *distributed lists* to block processor externally instead of internally. The third algorithm uses a fixed tree, as in Raymond's approach.

Some work has been done on distributed lists, primarily in the context of directory-based cache coherence algorithms. For example, the Scalable Coherent Interface (SCI) [9] uses a distributed queue to chain together all of the processors that are requesting access to a memory block. The algorithm is greatly simplified because the pointer to the head of the list is stored in a standard place (the home memory block). Translated to a distributed algorithm, the SCI algorithm requires a manager at a fixed site to remember the head of the list. In a path compression algorithm, no such fixed-site manager is available. We note that Li and Hudak found that their path compression algorithm had far superior performance to algorithms that required fixed site managers. A shared memory synchronization algorithm that is quite similar in nature to the SCI algorithm is the contention-free lock of Mellor-Crummey and Scott [15]. The recent contention-free priority locks are based on the Mellor-Crummey and Scott lock. Recent distributed synchronization algorithms by Naimi and Trehel [21], Neilsen and Mizuno [16], and by Chang, Singhal and Liu [4] make use of simple distributed lists to improve the performance of their algorithms and avoid $O(n)$ storage.

The contribution of this paper is to present three new practical distributed priority synchronization algorithms that require only $O(\log n)$ storage per processor and $O(\log n)$ message per synchronization request, where n is the number of processors. Two of the algorithms make a novel use of distributed lists to transfer the burden of remembering which processors are blocked to the blocked processors themselves. We provide a performance analysis to show which of the algorithms have the best performance. We find that the algorithm performance depends on the request priority distribution and on the metric used to measure the algorithm.

2 Three Distributed Priority Locks

The algorithms execute as a distributed protocol. When a processor decides to ask for the token, it sends its request to the processor indicated by a forwarding pointer. Eventually, the processor will receive the token and enter the critical section. A processor will receive unsolicited messages, and will treat them as events to be handled. Event handling will in general cause a change to the local state variables and often cause messages to be sent. A picture of the processor architecture is shown in Figure 1. The unprocessed events are stored in a pending event queue. Since we want the algorithms to require only $O(\log n)$ bits of storage, they must in general process all events immediately.

To simplify the presentation, we assume that all priorities are unique. Non-unique priorities

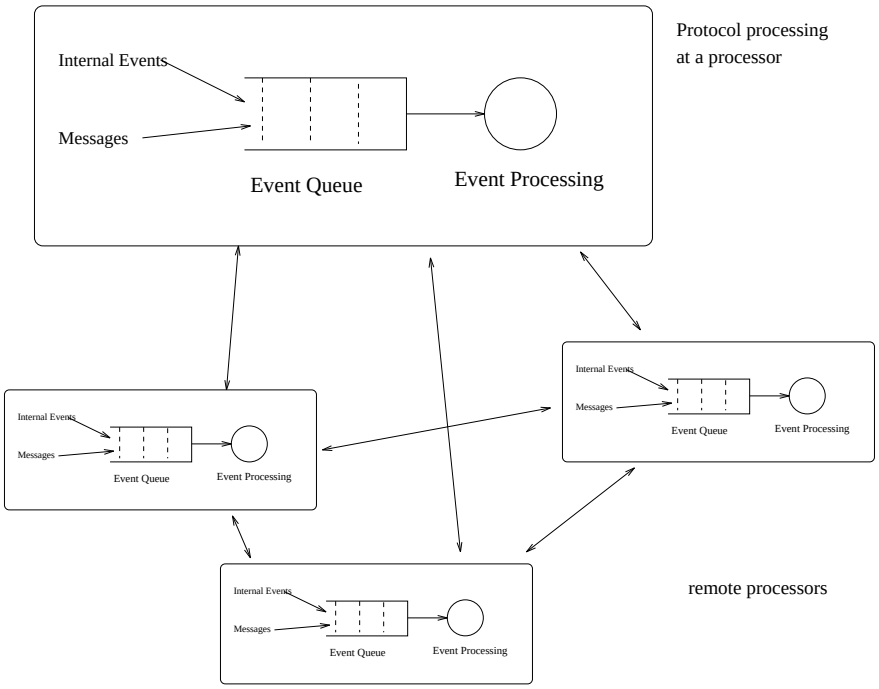

Figure 1: Architecture of the protocol.

can be made unique by attaching a timestamp and processor ID. In addition, the algorithms can be modified to handle non-unique priorities directly.

2.1 Single-link Protocol

The first algorithm keeps blocked processors in a singly-linked list. Our starting point is the path compression algorithm of Li and Hudak. Translated to distributed synchronization, their algorithm works as follows. Every processor has a pointer, `currentdir` that initially points in the direction of the current token holder. When processor A decides that it needs the token, it sends a request to the processor indicated by `currentdir`. If a processor that neither holds nor is requesting the token receives the request from A, it forwards the request to the processor indicated by its version of `currentdir` and then sets `currentdir` to A. If a processor that is using or is requesting the token receives A's request, it stores the request and takes no further action. This processor is illustrated in Figure 2. When a processor releases the token, it checks to see if there are any blocked requests. If so, the processor sends the token to one of the requestors, sets its version of `currentdir` to the new requestor, and unblocks any remaining blocked requests. If there are no blocked requests, the processor holds the token until a request arrives, and sends the token to the requestor (with a corresponding update to `currentdir`).

In a non-prioritized lock, it is permissible to block a request at a processor that is requesting the token because the blocker should get the token first anyway. In priority synchronization, every request has a priority attached. The blocked request might have a higher priority than the blocker. So, a processor must be able to find the set of all current requestors to register its request.

Thus, all requesting processors must know about each other. Since we are allowed only $O(\log n)$ storage, the processors can only use pointers to form lists. In the first algorithm we make the

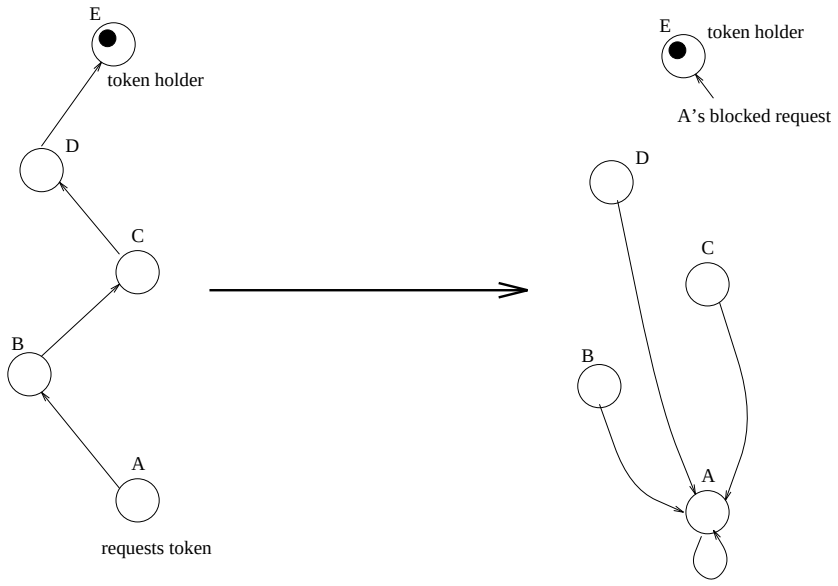

Figure 2: Execution of Li and Hudak's path compression synchronization.

requesting processors form a *waiting ring*. All waiting processors point to the next lower priority processor, except for the lowest priority processor which points to the highest priority processor. In addition to knowing the identity of your successor, a task in the waiting ring also knows the priority of its successor's request. The token holder must be able to release the token into the waiting ring, so it points to a processor in this ring. The processors that are not requesting the token might make a request, so they also lie on paths that point to the ring. The structure of the synchronization is shown in Figure 3.

The processors handle two classes of events: those related to requesting the token, and those related to releasing the token. Although these activities have some subtle interactions, we initially describe them separately.

2.1.1 Requesting the Token

Each processor p that is not holding or requesting the token stores a guess about the identity of the token holder in the local variable `currentdir`. Note that each processor has its own version of `currentdir`.

When a processor decides that it needs to use the token, it sends a Token-Request to the processor indicated by `currentdir` (if the processor already has the token, the processor can use it directly). When a processor that neither holds nor is requesting the token receives a Token-Request message, it forwards the request to the processor indicated by `currentdir`. After receiving the message, the processor knows that the requestor will soon have the token, so the processor changes `currentdir` to point to the requesting processor. This will continue until the Token-Request message reaches the token holder or a processor that is in the waiting ring.

When a Token-Request message reaches a processor, P, in the waiting ring, the correct position

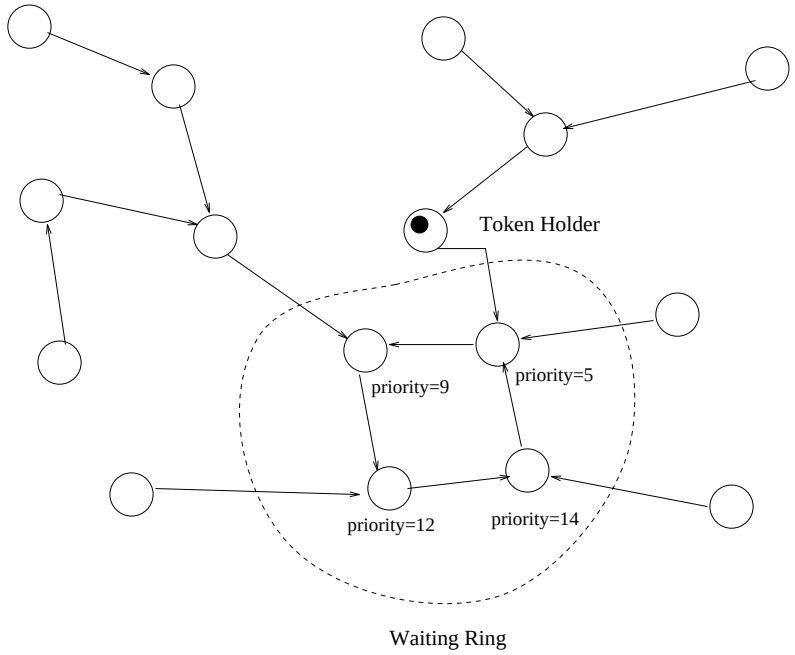

Figure 3: Structure of the distributed priority synchronization.

in the ring must be found for the requestor. If the requestor's priority is between that of P and P's successor, or if P is the lowest priority processor in the ring and the requestor's priority is lower than P's or greater than P's successor, then the requestor should follow P. P changes its ring pointer (we can re-use `currentdir`) to point to the requestor, and sends the requestor a Request-Done message, indicating the requestor's successor in the ring. When the requestor receives the Request-Done message, it sets its value of `currentdir` to its successor in the waiting ring. Otherwise, P sends the Token-Request message to its successor in the ring.

If a Token-Request message arrives at the token holder and the token holder is not using the token, the token holder replies to the requestor with the token. If the token holder is using the token, then there might or might not be a waiting ring. If the waiting ring exists, the message is forwarded into the waiting ring. Otherwise, the token holder replies with a Request-Done message indicating that the requestor is the only processor in the waiting ring.

2.1.2 Releasing the Token

If there are no processors waiting to acquire the token, the token holder sets an internal flag to indicate that token is available. Otherwise, the token holder releases the token into the waiting ring. When a processor receives the token, it cannot immediately use the token, since the processor does not know if it is the highest priority processor. Instead, the processor passes the token to its successor. The lowest priority processor knows that it is the lowest priority processor (because its successor has a higher priority) and therefore that its successor is the highest priority processor. Thus, the lowest priority processor marks the token before passing it to its successor. When a processor receives a marked token, it accepts the token and enters the critical section.

After new token holder accepts the token, the waiting ring structure must be repaired. The new

token holder sends the address of its successor to its predecessor (the processor that forwarded the token), which updates its `currentdir` pointer. Exceptions occur if the new token holder was alone in the waiting ring, or if the previous token holder released the token directly to the new token holder.

At this point, we can wonder if it might be better to reverse the direction of the pointers in the waiting ring. If we require a processor in the waiting ring to point to the processor with the next higher priority request, it is the highest priority processor that knows its status. However, the ring still needs to be repaired, and the token holder still needs to update the `currentdir` pointer in its predecessor. In general, finding the predecessor would require a circuit of the waiting ring, incurring a high message cost.

2.1.3 Implementation Details

The essential operation of the algorithm is as described in the previous section. However, there are a number of details that must be addressed to ensure the correct execution of the algorithm. The complications occur primarily because of two concerns: out-of-order messages and the $O(\log n)$ storage requirement.

Out-of-Order Messages Out-of-order messages occur when a message arrives that was not expected. Usually, the problem is due to non-causal message delivery. That is, processor A sends message m_1 to processor B, then sends message m_2 to processor C. Processor C receives m_2 and sends m_3 to B. At B, message m_3 arrives before message m_1. For example, a requesting processor can be given the token before being told that it is part of the waiting ring. This problem usually occurs when a new processor is admitted to the waiting ring. If A has successor B in the waiting ring, then admits C into the ring as its new successor, the first message from C to B might arrive before the last message from A to B. In our example, the last message from A to B tells B that it is in the waiting ring, and the first message from C to B is the token. Considerable work has been done on implementing causal communications [1], but this work requires that all messages are broadcast, which in general requires $O(n)$ messages.

The messages that need to be processed in-order involve the waiting ring maintenance. Out-of-order reception can be detected (i.e., you are given the token before entering the waiting ring), and the processing of the too-early message can be blocked until the appropriate predecessor message arrives. There can be only O(1) messages that can arrive too early, so delaying their processing does not impose too large a space requirement. Because the protocol must handle non-causal messages, it also correctly handles messages from a processor that are delivered out of order.

No Blocking There are many occasions when a processor, A, cannot correctly interpret a message that has arrived (as discussed above). If the unexpected message involves A's state, then the message processing can be safely blocked because there can be only O(1) such messages. However, the message might be a request from a different processor, B. Since many processors might send their request to A, processor A must be able to handle these requests immediately.

In the algorithm by Li and Hudak, a processor blocks requests from other processors when it is using or is requesting the token. In our algorithm, the token holder does not block requests, instead it tells the requestors to form a waiting ring. Once a requesting processor joins the waiting ring, it helps the new requests to also join the waiting ring. However, between the time that a processor, A, requests the token and the time it joins the waiting ring, it cannot handle requests from other processors. The problem is that during this time, A's forwarding pointer `currentdir`

does not have a significant meaning. Handling the requests of others will cause cycles among the non-ring processors.

Since processor A cannot handle foreign requests and cannot block these requests internally, processor A will block them externally. During the time that processor A is requesting the token but has not yet been told that it is in the waiting ring, processor A will respond to token requests by linking them into a blocking list. The blocking list is managed as a LIFO. When a request arrives, Processor A responds with a Block message, with the address of the previous head of the blocking list (or a null pointer if the list is empty). When processor A joins the waiting ring, it sends an Unblock message to the head of the blocking list. The Unblock message is relayed down the chain of blocked processors. After unblocking, the processors resubmit their requests to A.

Miscellaneous When a processor accepts the token, it sends a message out to repair the ring. The ring must be repaired before the token can be released back into the ring. Therefore, when the ring is repaired an acknowledgement is sent to the token holder. The token holder blocks its release of the token until the acknowledgement that the ring is repaired has been received.

When a processor is in the waiting ring, it does not block new processors from entering the ring. In particular, the lowest priority processor will admit new processors into the ring during the time between sending a marked token to the highest processor and receiving a request to repair the ring. Only the processor that points to the token holder can repair the ring, so the request is passed along in the waiting ring until it reaches a processor that points to the token holder. This processor repairs the ring and sends an acknowledgement to the token holder. This process is illustrated in Figure 4.

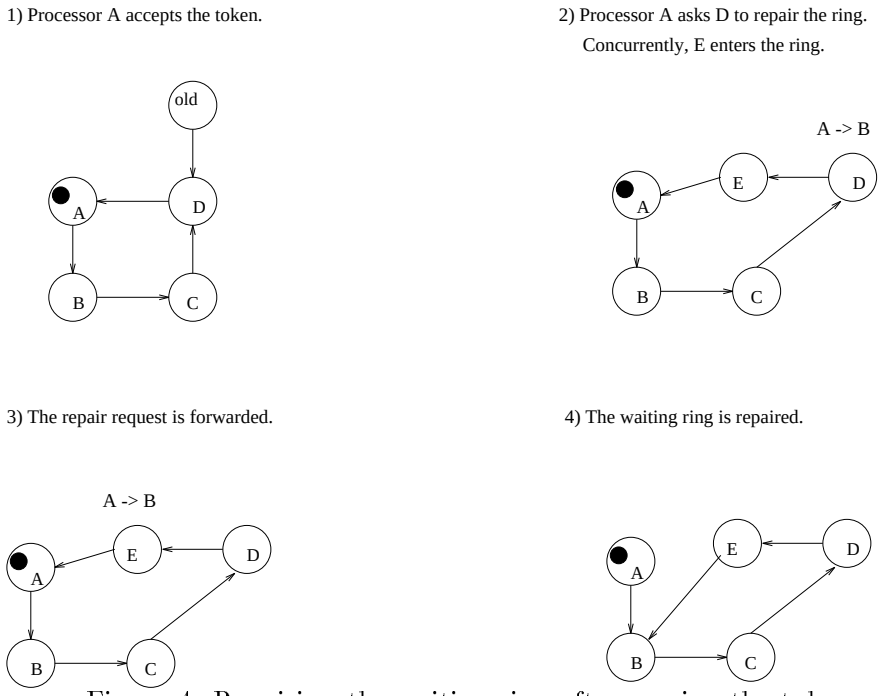

Figure 4: Repairing the waiting ring after passing the token.

2.1.4 The Algorithm

In this section we present the code for the algorithm that we have described in the previous two sections.

The protocol uses the following variables:

Boolean	tokenhldr	True iff. the processor holds the token
Boolean	incs	True iff. the processor is using the token
Boolean	in_ring	True iff. the processor knows it is in the waiting ring.
Boolean	isrequesting	True when the processor has made a token request but hasn't joined the waiting ring.
Boolean	changed_acked	True if the waiting ring has been repaired.
Boolean	areblocking	True if the processor is blocking token requests.
Boolean	areblocked	True if you are in a list of blocked requests.
Integer	currentdir	Direction of the successor processor.
Integer	blocked_dir	Head of a list of blocked requestors.
Integer	unblockdir	Next blocked requestor.
Integer	id	Name of the processor (never changes).
Real	priority	The priority of the processor's request.
Real	linkpri	Priority of the successor's request.

The protocol is specified by specifying how events are handled. We assume that each event is handled atomically at the processor where it occurs, except where a specific condition for waiting is specified. A processor makes an event occur at a remote processor by sending a message. The parameters of the send procedure are:

send(*processor id, event; parameters for the event*).

A message is sent to cause the specified event to occur on the remote processor, and the event is passed the parameters.

The protocol driver takes events off of a queue and calls the appropriate routine to handle them. The events may be due the the receipt of messages, or may be caused internally.

```
DPQ_handler()
    while(1)
        get an event from the event queue
        call the appropriate routine to process the event
```

Initially, the `currentdir` pointers form a tree with the token holder at the root. The token holder has the variable `tokenhldr` set TRUE, and the variable is set to FALSE at all other processes. The variables `incs`, `in_ring`, and `isrequesting` are FALSE. The constant `id` contains the processor name. All other variables are set before they are used.

When a processor needs to use the token, it generates a REQUEST_TOKEN event. We note here a particular use of the `currentdir` variable. If the processor holds the token and `currentdir` points to the processor, then there is no waiting ring and hence no blocked processors. If `currentdir` points to a remote processor, that processor is part of the waiting ring.

```
REQUEST_TOKEN(request_priority)
    priority=request_priority
```

```
if(tokenhldr) // If you hold the token, use it.
    incs=TRUE
    change_acked=TRUE // The ring doesn't need repair.
    currentdir=id // Indicate that the waiting ring is empty.
    in_ring=FALSE
else
    send(currentdir,RECEIVE_RQST; id,priority)
    isrequesting=TRUE
```

A processor enters the waiting ring when it receives a REQUEST_DONE message. Later, the processor will receive the token. Since the processor can handle token requests now, it will wake up any blocked requests.

```
REQUEST_DONE(successor,successor_priority)
    in_ring=TRUE
    currentdir=successor
    link_pri=successor_priority
    isrequesting=FALSE

    if(areblocking) // Wake up blocked requests, if any.
        send(blocked_dir,UNBLOCK; id)
        areblocking=FALSE
```

Most of the work of the protocol is in the RECEIVE_RQST event, which handles remote requests to obtain the token. The handling of the event depends on the state of the processor (holding the token, using the token, in the waiting ring, requesting but not in the waiting ring, not requesting). Note that the protocol as written requires a processor to enter the waiting ring before receiving the token, so two messages must be sent if the critical section is idle. This can be optimized by using a single special message.

```
RECEIVE_RQST(requestor,requestor_pri)
    if(tokenhldr and not incs) // Send the token.
        currentdir=requestor
        send(requestor,REQUEST_DONE; requestor,requestpri)
        send(requestor,TOKEN; id,FALSE,TRUE)
        tokenhldr[id]=FALSE

    else if(tokenhldr and incs)
        if(currentdir==id) // No waiting ring, so create one.
            linkpri=request_pri
            currentdir=requestor
            send(requestor,REQUEST_DONE; requestor,requestpri)
        else // Waiting ring exists, so forward the request there.
            send(currentdir,RECEIVE_RQST; requestor,request_pri)

    else if(in_ring)
```

```
    if(priority<link_pri and priority>=requestpri) // new lowest priority processor.
        send(requestor,REQUEST_DONE; currentdir,link_pri)
        currentdir=requestor
        link_pri=request_pri
    else if((request_pri<=priority and request_pri>link_pri) or
                (priority<link_pri and request_pri>link_pri)) // found position in the ring.
        send(requestor,REQUEST_DONE; currentdir,link_pri)
        currentdir=requestor
        link_pri=request_pri
    else // Not the right position, so forward the request.
        send(currentdir,RECEIVE_RQST; requestor,request_pri)

else // Not token holder, not in waiting ring.
    if(!isrequesting) // not requesting, so forward the request.
        send(currentdir,RECEIVE_RQST; requestor,request_pri)
        currentdir=requestor
    else // Can't forward the request, so block it.
        if(areblocking)
            send(requestor,BLOCK; blocked_dir,FALSE,id)
        else // First blocked processor.
            send(requestor,BLOCK; NULL,TRUE,id)
            areblocking=TRUE
        blockeddir=requestor
```

When the token holder releases the token, it sends the token to the waiting ring, if it exists. If no waiting ring exists, then `currentdir` will be pointing to the token holder.

```
RELEASE_TOKEN()
    wait until changed_acked is TRUE

    change_acked=FALSE
    incs=FALSE
    if(currentdir!=id) // There is a blocked processor.
        send(currentdir,TOKEN; id,FALSE,FALSE)
        tokenhldr=FALSE
```

This event unblocks the release of the token.

```
CHANGE_ACK()
    change_acked=TRUE
```

The lowest priority processor tags the token. When a processor receives the token, it accepts the token only if the token is tagged. As an optimization, the processor can accept the token if it is alone in the waiting ring. After accepting the token, the token holder points to the processor that sent the token, which is likely to be the lowest priority processor in the waiting ring. Finally, the waiting ring is repaired.

```
TOKEN(sender,tag,direct)
    wait until in_ring is TRUE

    if(tag or currentdir==id)
        tokenhldr=TRUE
        incs=TRUE
        if(currentdir==id) // If you are alone, don't need to repair the ring.
            change_acked=TRUE
        else
            if(!direct) // If the token holder didn't send the token, the
                                sender is probably the lowest priority processor.
                currentdir=sender
            send(currentdir,CHANGE_LINK; id,currentdir,link_pri)
        in_ring=FALSE
    else
        if(priority<link_pri) // lowest, so tag the token.
            send(currentdir,TOKEN,id,TRUE,FALSE)
        else
            send(currentdir,TOKEN,id,FALSE,FALSE)
```

This event repairs the ring after a new processor accepts the token.

```
CHANGE_LINK(tokhldr,successor, succ_pri)
    wait until in_ring is TRUE

    if(currentdir==tokhldr)
        currentdir=successor
        link_pri=succ_pri
        send(tokhldr,CHANGE_ACK)
    else
        send(currentdir,CHANGE_LINK; tokhldr,successor,succ_pri)
```

A requesting processor blocks if its request is forwarded to another requesting processor that has not yet joined the waiting ring. The blocked processors form a LIFO list, which removes the need for $O(n)$ storage per processor.

```
BLOCK(next_blocked,islast,blocker)
    unblock_dir=next_blocked
    lastblocked=islast
    currentdir=blocker
    areblocked=TRUE
```

When a requesting processor joins the waiting ring, it unblocks the head of the blocking list, which unblocks its successor and so on.

```
UNBLOCK(unblocker)
    wait until areblocked is TRUE
```

if(not lastblocked) // **Relay the unblocking.**
    send(unblockdir,UNBLOCK; unblocker)
currentdir[id]=unblocker
send(currentdir,RECEIVE_RQST; id,priority) // **Resubmit your request.**
isrequesting=TRUE
areblocked=FALSE

2.1.5 Correctness

In this section, we give some intuitive arguments for the single-link algorithm's correctness. We loosely refer to events as occurring at a point in 'time'. While global time does not exist in an asynchronous distributed system, we can view the events in the system as being totally ordered using Lamport timestamps [11], and view a point in time as being a consistent cut [3].

We note that all processors that are not requesting the token lie on a path that leads to a processor that either holds or is requesting the token. This property can be seen by induction. We assume the property holds initially (and this is required for correctness). The property can change if a processor modifies its `currentdir` pointer, or if the processor it points to changes its state. A processor that is not requesting the token will change its `currentdir` pointer if it relays a request. But then, it points to the requesting processor. A non-requesting processor can also change its pointer if it sends the token to another processor, but `currentdir` is set to the new token holder. A processor can change its state from not requesting to requesting, but the property still holds. Finally, a processor can change its state from holding the token to not-requesting. But, after changing state the processor points to the new token holder or to a requesting processor.

The token is not lost because it is only released to processors in the waiting ring (and a processor must enter the waiting ring before accepting the token). The token is released to the highest priority processor in the waiting ring at the time that lowest priority processor in the ring handles the token.

2.2 Double-link Algorithm

In the single-link algorithm, if a request hits the waiting ring at a processor with a lower priority, the request must traverse most of the ring until reaching its proper position. Thus, we can hope to improve on the single-link algorithm by having blocked processors point to both the next higher and the next lower priority processor. Thus, the waiting processors form a doubly linked list. Further, since a request can move directly towards its proper place, there is no need to maintain a waiting ring, so instead the processors form a waiting chain. This is illustrated in Figure 5.

The execution of the double-link protocol is similar in many ways to the single-link protocol, and is best described by its differences from the single-link protocol. The first difference is that when a processor is admitted to the waiting chain, it is told its successor and its predecessor. The processor that admits the newcomer knows who its new neighbor is, but the other neighbor of the newcomer doesn't. This is illustrated in Figure 6.

Thus, the second difference between the double-link and the single-link algorithm is that now only one of the links is guaranteed to be consistent, and the other can only be used for navigation. Which link is consistent depends on the policy for admitting a processor to the waiting chain. If the requestor is admitted if its priority is between that of the current processor and its successor (the next lower priority processor), then forward link is always correct, but the backwards link can be inconsistent. If the requestor can be admitted between current processor and its predecessor, then the backwards link is always consistent, but not the forwards link.

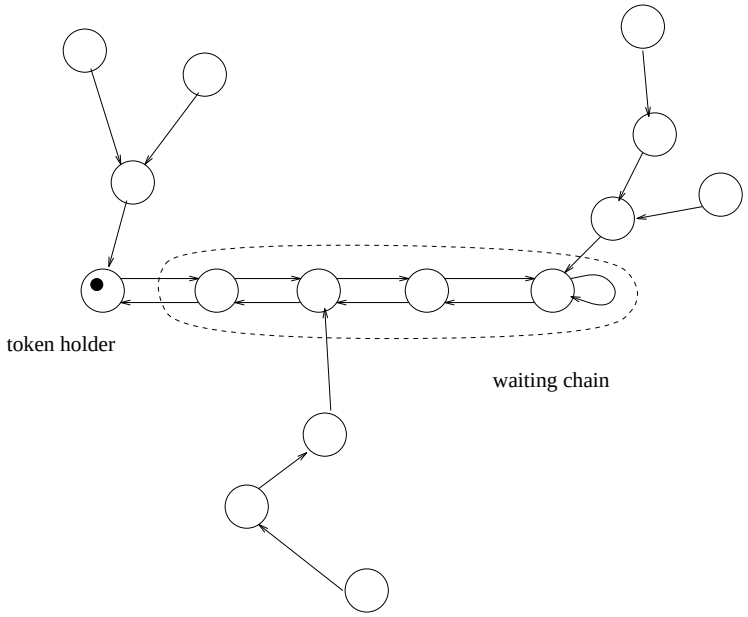

Figure 5: Structure of the waiting chain.

If we choose to make the forwards link consistent, then the token holder can pass the token on along its forward pointer. When a processor receives the token, it must have been the highest priority waiter, so it can enter the critical section immediately. However, before releasing the token, the token holder must wait until it is certain that there are no inconsistent backwards pointers that point to it. Otherwise, a request might be forwarded out of the waiting chain and cause cycles among non-chain requestors.

For example, processor A might release the token while processor C still has a backwards link to A. Suppose that a request from B is subsequently routed through A, and then to C. If B's request is subsequently routed through C's backwards link, B's request will again be routed through A. Because A's request had previously visited B, B points to A, so A's request is sent back to A, where it is blocked permanently.

If we make the backwards link always consistent, then all links of processors in the waiting chain point to the token holder, or to other processors in the waiting chain (with one exception which we discuss shortly). For this reason, we choose to keep the backwards link consistent. As a result, when a processor in the waiting chain receives the token, it must pass the token backwards if its backwards link does not point to the original token sender.

When the token holder releases its token into the waiting chain, the link that points back to it suddenly points to a processor that is outside of the waiting chain. If a request is sent along this inconsistent link, the system can experience blocking as described above. We can solve this problem by observing that if the highest priority waiting processor knows that it is the highest priority waiting processor, it will never send a request along its backwards link. Instead, when the highest priority waiting processor receives a request from a higher priority processor, it inserts the higher priority processor before it in the waiting chain (and tells the new processor that it is the highest priority waiting processor). So, to ensure correctness we need to ensure that the highest

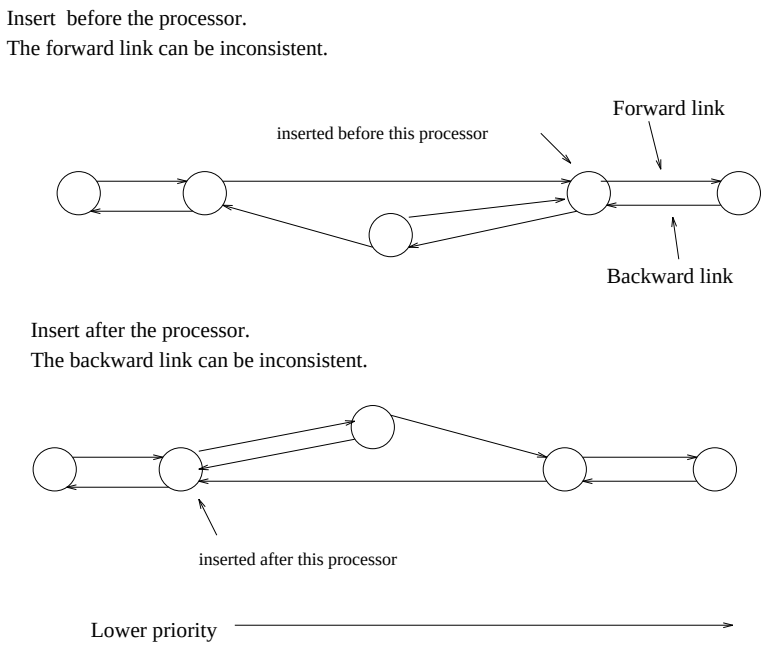

Figure 6: Inconsistency after admitting a processor to the waiting ring.

priority waiting processor knows that it is the highest priority waiting processor when the token holder releases the token. When a waiting processor receives the token, it sends a message to inform its predecessor that it is the highest priority waiting processor. The token holder cannot release the token until it receives an acknowledgement.

Since we do not assume causal message delivery, a processor may receive link change messages out of order. Fortunately, it is easy to determine the proper order of the link change messages, since a processor should only change its forwards link to point to a higher priority processor. An additional problem with the change link messages is that a link change request might arrive after the processor has already acquired and released the token, and has re-entered the waiting chain. To solve this problem, each processor appends an entry count to its name, and only responds to a link change request if the entry count on the request matches its current entry count.

2.2.1 The Double-Link Protocol

In this section we present the details of the double-link protocol. We omit some procedures that are identical to the ones in the single-link protocol.

The routine for requesting the token is essentially the same as the one for the single-link protocol. The routine for accepting notification that the request is finished is somewhat different, because the predecessor as well as the successor is passed. Also, notification of being the highest priority waiting processor is passed to the processor.

The protocol uses the following variables:

Boolean tokenhldr	True iff. the processor holds the token
Boolean incs	True iff. the processor is using the token
Boolean in_chain	True iff. the processor knows it is in the waiting chain.

Boolean	isrequesting	True when the processor has made a token request but hasn't joined the waiting ring.
Boolean	changed_acked	True if the highest priority waiting task knows that it is the highest.
Boolean	areblocking	True if the processor is blocking token requests.
Boolean	areblocked	True if you are in a list of blocked requests.
Integer	currentdir	Direction of the successor processor.
Integer	backdir	Direction of the predecessor processor.
Integer	blocked_dir	Head of a list of blocked requestors.
Integer	unblockdir	Next blocked requestor.
Integer	id	Name of the processor (never changes).
Real	priority	The priority of the processor's request.
Real	linkpri	Priority of the successor's request.
Real	backpri	Priority of the predecessor's request.

The routine to request the token is the same as in the single-link protocol. The Request-Done routine is similar to that of the single-link protocol, except that the predecessor processor is recorded, as well as the whether or not the requestor is the highest priority waiting processor.

```
REQUEST_DONE(successor,successor_priority,predecessor,predecessor_priority, youarefirst)
    currentdir=successor
    linkpri=successor_priority
    backdir=predecessor
    backpri=predecessor_priority
    isfirst=youarefirst

    in_chain=TRUE
    isrequesting=FALSE
    if(areblocking)
        send(blockeddir,UNBLOCK,id)
        areblocking=FALSE
```

As in the single-link protocol, most of the work is done in the Receive-Rqst routine. If the processor holds the token but is not using it, it replies with a Request-Done message, and then the token. Since in this case there is no predecessor, the Request-Done message specifies a Null predecessor. If the processor is using the token and the waiting chain is empty, the requestor becomes the first processor in the waiting chain, otherwise the request is passed into the waiting chain. If the requestor is in the waiting chain and it receives the request, there are three possibilities for admitting the requestor into the chain. First, the processor might be the highest priority waiting processor, and the requestor has a higher priority. In this case the privilege of being the highest priority waiting processor is passed along. Since it is possible that there is no predecessor, this condition is checked before issuing the Change-Link message. Second, the processor might be the lowest priority waiting processor (which is true if processor's forward link points to itself) and the request has a lower priority. In this case a new lowest priority waiting processor has been found. The requestor is added to the chain after the processor. This exception to the 'insert-before' rule does not cause a problem because the forwards and backwards pointers are consistent after the

insert. Third, the requestor might have a priority greater than the processor's but less than the predecessor's. In this case the request is granted and the predecessor is sent a Change-Link message. The remainder of the protocol is the same as in the single-link protocol.

```
RECEIVE_RQST(requestor,requestor_pri)
    if(tokenhldr and not incs)
        currentdir=requestor
        send(requestor,REQUEST_DONE; requestor,requestpri,NULL,0,TRUE)
        send(requestor,TOKEN)
        tokenhldr=FALSE

    else if(tokenhldr and incs)
        if(currentdir==id)
            linkpri=requestpri
            currentdir=requestor
            send(requestor,REQUEST_DONE; requestor,requestpri,id,priority,TRUE)
        else
            send(currentdir,RECEIVE_RQST; requestor,requestpri)

    else if(in_chain)
        if(requestpri>priority and isfirst) // Found a new highest priority
                        waiting processor.
            send(requestor,REQUEST_DONE; id,priority,backdir,backpri,TRUE)
            if(backdir!=NULL) // If there is a predecessor.
                send(backdir,CHANGE_LINK,requestor,requestpri)
            backdir=requestor
            backpri=requestpri
            isfirst=FALSE
        else if(requestpri>priority and backpri>requestpri) // Found the place to
                        insert the requestor.
            send(requestor,REQUEST_DONE; id,priority,backdir,backpri,FALSE)
            send(backdir,CHANGE_LINK; requestor,requestpri)
            backdir=requestor
            backpri=requestpri
        else if(priority>requestpri and currentdir==id)) // Found a new lowest priority
                        waiting processor.
            send(requestor,REQUEST_DONE,requestor,requestpri,id,priority,FALSE)
            currentdir=requestor
            linkpri=requestpri
        else if(requestpri>priority)
            send(backdir,RECEIVE_RQST; requestor,requestpri)
        else
            send(currentdir,RECEIVE_RQST,requestor,requestpri)
    else

        if(not isrequesting)
            send(currentdir,RECEIVE_RQST; requestor,requestpri)
            currentdir[id]=requestor
```

```
    else
        if(areblocking)
            send(requestor,BLOCK; blockeddir,FALSE,id)
        else
            send(requestor,BLOCK,NULL,TRUE,id)
            areblocking=TRUE
        currentdir=requestor
        blockeddir=requestor
```

The procedures for handling requests blocked on requesting processors is the same as in the single-link protocol. The Change-Link procedure restores the consistency of the forward links. Since messages can be received out of causal order, the Change-Link request should be executed only if it will make the forward pointer refer to a higher priority processor. The Change-Link request should also be checked to make certain that it did not originate from a previous occasion when the processor requested the token.

```
CHANGE_LINK(successor, succ_pri)
    if(succ_pri>linkpri)
        currentdir=successor
        linkpri=succ_pri
```

As in the single-link algorithm, the processor must first be admitted to the waiting chain before accepting the token. The processor can accept the token if it knows that it is the highest priority processor, otherwise it passes the token to its predecessor. If the processor accepts the token, and it has a successor, it must tell the successor that it is now the highest priority waiting processor. The Are-First routine is handled like the Change-Link routine in the single-link protocol, as both repair the blocking structure. The Are-First message is passed to the predecessor until it reaches a processor whose predecessor is the token holder. This processor sets its `arefirst` variable, and sends a First-Ack message to the token holder. The token holder sets the firstacked variable when it processes the First-Ack message.

```
TOKEN()
    wait until in_chain is TRUE

    if(isfirst)
        tokenhldr=TRUE
        incs=TRUE
        in_list=FALSE
        isfirst=FALSE
        if(currentdir !=id)
            send(currentdir,AREFIRST; id)
            firstacked=FALSE
        else
            firstacked=TRUE
    else
        send(backdir,TOKEN)
```

The Release-Token routine must block until the processor knows that the highest priority waiting processor knows that it is the highest priority waiter.

```
RELEASE_TOKEN()
    wait until firstacked is TRUE

    incs=FALSE
    if(currentdir!=id)
        send(currentdir,TOKEN)
        tokenhldr=FALSE
```

2.3 Fixed-Tree Algorithm

The single-link and the double-link algorithm maintain a dynamic tree, with the token holder as the root. We can take a different approach, and maintain the processors in a fixed tree. Each processor maintains a single pointer, `currentdir`, which indicates the location of the token. If the token is in a subtree rooted at the processor, `currentdir` points to the child whose subtree contains the token. If the token is not in the subtree rooted at the processor, `currentdir` points to the processor's parent. This structure is illustrated in Figure 7. Raymond [17] presents a simple distributed non-prioritized synchronization algorithm with guaranteed $O(\log n)$ performance by using this technique. Raymond's algorithm can be fairly easily modified to permit prioritized synchronization.

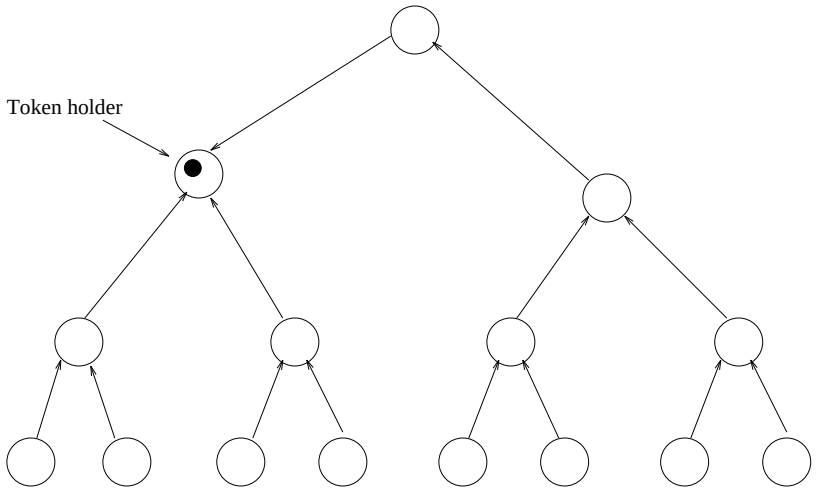

Figure 7: Structure of the fixed-tree algorithm.

In addition to keeping `currentdir`, each processor keeps a priority queue of the requests that it has received from its neighbors. When the processor receives a request from a neighbor (or makes a request itself), the request is put in the priority queue, replacing any previous requests from that

neighbor. If the request is the highest priority request in the list, the request is forwarded to the processor indicated by `currentdir`. This execution is illustrated in Figure 8.

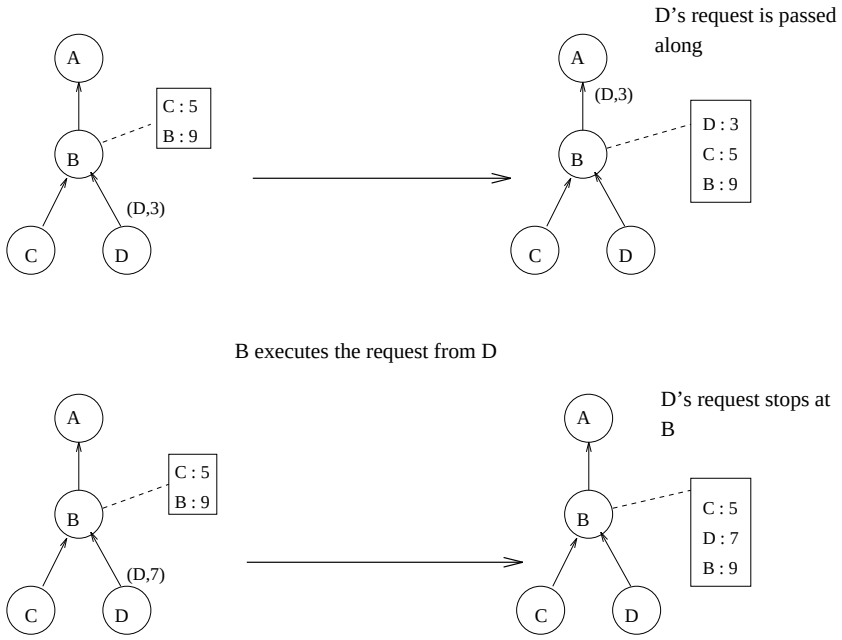

Figure 8: Processing a request in the fixed-tree algorithm.

When the processor receives the token, it removes the highest priority request from its priority queue and sends the token to the appropriate processor (possibly to itself), and modifies `currentdir` accordingly. If the queue is non-empty after forwarding the token, the processor sends a request to `currentdir` with the priority of the highest priority request in the priority queue.

2.3.1 The Algorithm

The protocol uses the following variables:

Boolean tokenhldr	True iff. the processor holds the token
Boolean incs	True iff. the processor is using the token
Integer currentdir	Direction of the successor processor.
Integer id	Name of the processor (never changes).

In addition, the protocol uses the following routines to manage the priority queue of pending requests. There must be room for as many entries in the priority queue as there are neighbors of the processor.

Boolean pushreq(processor, priority)
: Put the request from the processor with the attached priority into the priority queue. If there is another request from the same processor already in the queue, replace it.

	Return TRUE iff. the request is the highest priority one.
Integer popreq()	Remove and return the direction of the highest priority request.
Boolean qmt()	Return TRUE iff. the priority queue is empty.
Integer toppri()	Return the priority of the highest priority request in the priority queue.

In the Request-Token routine, if processor is the token holder it just uses the token. Otherwise the processor puts its request on the priority queue. If the request is the highest priority one, it passes the request in the direction of the token.

```
REQUEST_TOKEN(request_priority)
    priority=request_priority
    if(tokenhldr) // If you hold the token, use it.
        incs=TRUE
    else
        if(pushreq(id,priority)) // send the request on if its the highest priority.
            send(currentdir,RECEIVE_RQST; id,priority)
```

If the processor is holding but not using the token when it receives a request, return the token to the requestor. If the processor is using the token, it puts the request in its priority queue. Otherwise, the processor puts the request in its priority queue, and passes the request along if the request has a higher priority than previous requests. The processor will ignore requests from the direction of the token holder. Such requests can occur if the processor recently sent the token in the direction of `currentdir`.

```
RECEIVE_RQST(requestdir,requestpri)
    if(tokenhldr and not incs) // Pass the token.
        currentdir=requestdir
        send(requestdir,TOKEN; NULL,0)
        tokenhldr=FALSE
    else if(tokenhldr and incs) // Just remember the request.
        pushreq(requestdir,requestpri)
    else // Remember the request, pass along if highest.
        if(requestor != currentdir)
            if(pushreq(requestdir,requestpri))
                send(currentdir,RECEIVE_RQST; id,requestpri)
```

When the processor releases the token, it checks the priority queue to see if there are any pending requests. If so, the processor sends the token in the direction of the highest priority request. If there is another pending request, the processor must ask for the token back on behalf of the requestor. The protocol is written to make the additional request when it passes the token (a small optimization).

```
RELEASE_TOKEN()
    incs=FALSE
```

```
    if(not qmt())
If there is a waiting processor.
        currentdir=popreq()
        tokenhldr=FALSE
        if(qmt())
Is there a request to piggyback?
            send(currentdir,TOKEN,NULL,0)
        else
            send(currentdir,TOKEN,id,toppri())
```

When the processor receives the token, it consults the priority queue to determine the direction that the token should be sent. The processor also puts the piggybacked request (if any) in the priority queue. If the processor is the one who should receive the token, it enters the priority queue. Otherwise, it passes along the token possibly piggybacking another request.

```
TOKEN(requestdir,requestpri)
    currentdir=popreq()
    if(requestdir !=NULL)
test for the piggybacked request.
        pushreq(requestdir,requestpri)
    if(currentdir!=id)
        if(qmt())
            send(currentdir,TOKEN,NULL,0)
        else
            send(currentdir,TOKEN,id,toppri())
    else // accept the token if your request is the highest priority.
        tokenhldr=TRUE
        incs=TRUE
        isrequesting=FALSE
```

3 Theoretical Analysis

In the path-compression algorithms, there are two components to the number of messages required for a processor to enter the waiting structure: the number of hops for the request to reach the ring, and the number of hops around the waiting structure. Previous analyses show that the number of hops to find the waiting processes is $O(\log n)$ [6]. The number of hops that a request makes around the waiting structure is difficult to determine, but we can estimate that the number of hops is proportional to the number of waiting processors. Let C be the time to execute the critical section, and let R be the time between requests. If $n < R/C$, then the waiting structure will be small. If $n > R/C$, then on average there will be $n - R/C$ blocked processors. However, the utilization of the lock should be less than 100% in the long term, or the system suffers from a serialization bottleneck.

In the fixed-tree algorithm, a request must travel to the token, and then the token must travel to the requestor. Both distances will be proportional to the diameter of the tree, which is $O(\log n)$. However, some of the requests do not need to be forwarded the entire distance.

If the lock utilization is less than 100%, then an order-of-magnitude analysis tell us little about the relative performance of the algorithms. If the lock utilization is 100%, then the set of waiting processes is likely to be long. It would seem that the fixed-tree algorithm has the performance advantage, because of its guaranteed $O(\log n)$ performance. However, in the path compression algorithms a request might on average need to make significantly less than $n - R/C$ hops to enter the waiting structure. So, again the relative performance of the algorithms is not clear.

4 Performance Analysis

Since a theoretical analysis of the three distributed priority lock algorithms does not clearly show that one algorithm is better than another, we make a simulation study of the algorithms. The simulator modeled a set of processors that communicate through message passing. All delays are exponentially distributed. The parameters to the simulator are the number of processors, the message transit delay (mean value is 1 tick), the message processing delay (1 tick), the time between releasing the token and requesting it again (the inter-access time, varied), and the time that a token is held once acquired (the release delay, 10 ticks). The fixed-tree algorithm uses a nearly-complete binary tree (requiring about the same number of bytes per processor as the single-link algorithm).

We ran the simulator for varying numbers of processors and varying loads, where we define the load to be the product of the number of processors and the release delay divided by the inter access time (nC/R). For each run, we executed the simulation for 100,000 critical section entries. We collected a variety of statistics, but principally the amount of time to finish the simulation (which captures the time overhead of running the protocol), and the number of messages sent.

The performance of the algorithms depends on the distribution of the request priorities. If the requests are prioritized based on the importance of a job (i.e., critical tasks have high priorities), or if the requests are due to real-time requests that use static priority assignments, the request priority distribution will be stationary. If requests are prioritized based on the deadlines of the requestors, then the request priorities will tend to decrease over time. In our study, we simulated both types of request priority distributions. In the *stationary priority* experiments, the priority of a request is an integer chosen uniformly randomly between 1 and 10,000. In the *non-stationary priority* experiments, the priority of a request is a value chosen uniformly between 1 and twice the inter-request time minus the simulation time when the request is generated.

4.1 Stationary Priorities

In our first set of experiments, we plot the number of messages sent per critical section entry against the number of participating processors. Every processor issues requests at the same rate, and the load (nC/R) is varied between 50% and 200%. The results of these experiments are plotted in Figures 9 through 12. The single-link and double-link algorithm both require about the same number of messages per critical section, and far fewer messages than the fixed-tree algorithm when the load is less than 100%. However, the fixed-tree algorithm requires fewer messages when the load is high (greater or equal to 100%) and there are only a few (less than 20) processors. In addition, the single-link algorithm requires significantly fewer messages than the double-link algorithm when the load is high.

The single-link algorithm requires fewer messages than the fixed tree algorithm for two reasons. First, the fixed-tree algorithm must send the token over a long path when the token is released. The single-link algorithm usually requires only one or two token hops. Second, path compression generally does a good job of compressing the tree, while in the fixed tree algorithm most processors

are on the periphery of the tree. When the load is high (100% or more), the fixed-tree algorithm requires the least number of messages when the system is small. This occurs because the path-compression algorithms require a couple of overhead messages to repair the structure when the token is released, while the fixed-tree algorithm doesn't. Also, in the fixed-tree algorithm some requests get subsumed by higher priority requests, while in the path compression algorithms the requests need to traverse large waiting lists.

The single-link algorithm has surprisingly good performance when the load is high. This occurs because the lowest priority requesting processor requires a very long time to obtain the lock. When a processor releases the token, it sets its `currentdir` pointer to the lowest priority waiting processor. When it requests the token again, it is pointing to a processor that is in the waiting ring and which points to the highest priority waiting processor. In the double-link algorithm, a processor points to the highest priority waiter after it releases the token, and so is likely to be far away from the waiting chain when it requests the token next.

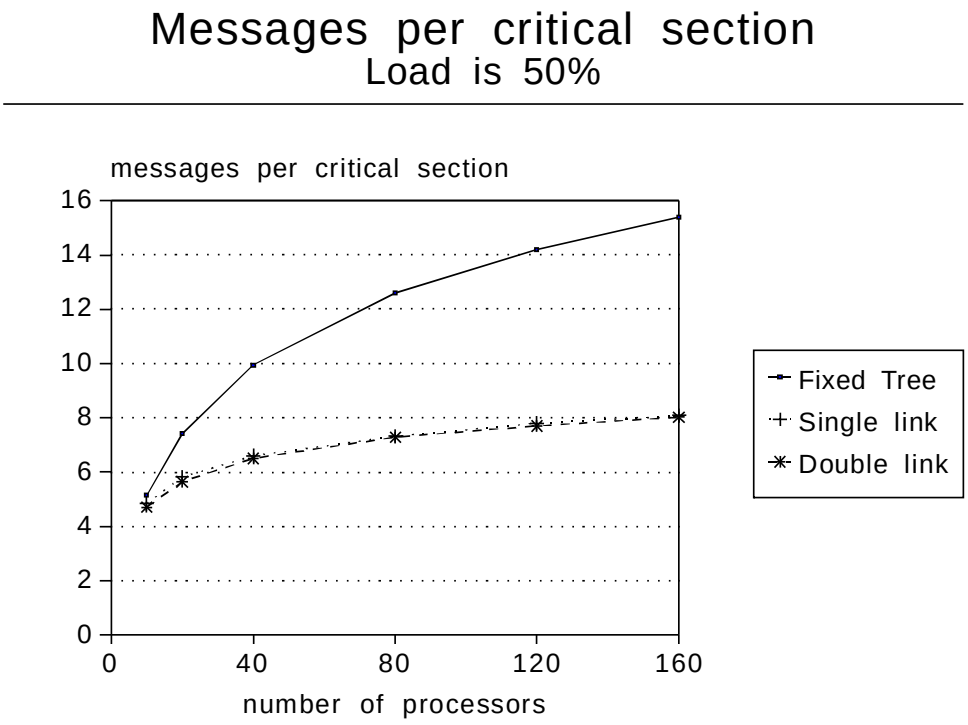

Figure 9: Number of messages per critical section entry, 50% load, uniform requests and stationary priorities.

One possible advantage of using a path-compression algorithm over the fixed-tree approach is that the path-compression algorithms react to the changes in the access patterns of the requestors. Typically, only a small subset of the processors will makes requests for the critical section, and this set changes dynamically over time. We revised the simulations so that about one tenth of the processors make requests at any given time. We re-ran the experiments with a 50% and a 100% load, and plot the results in Figures 13 and 14. The results confirm the hypothesis that path-compression algorithms react better to hot-spots in the request pattern than do fixed-tree algorithms, in addition to having better performance when the requests are uniformly generated among the processors. However, the effect is not large.

We also investigated the time overhead of running the distributed priority lock algorithms. The

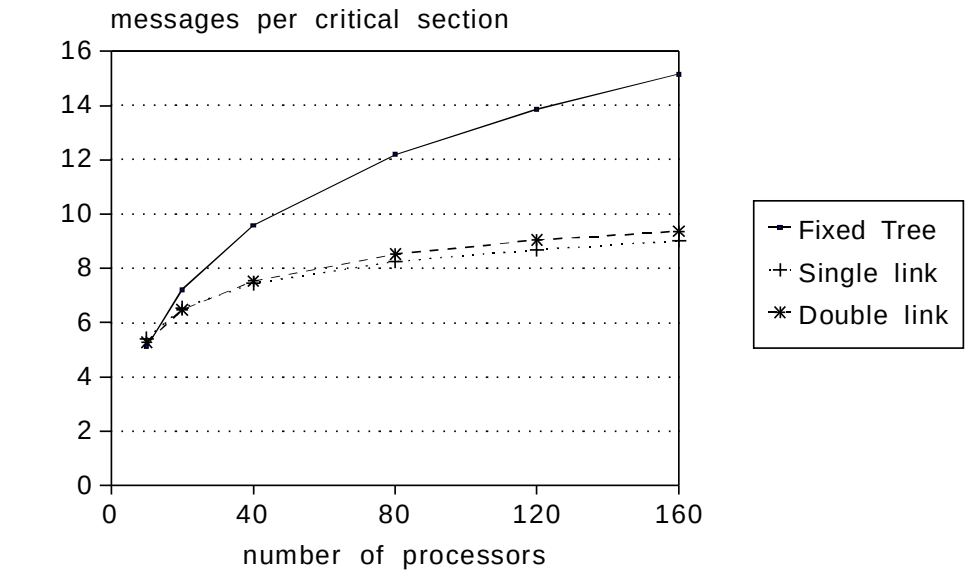

Figure 10: Number of messages per critical section entry, 75% load, uniform requests and stationary priorities.

rate at which a critical section can be accessed by different processors depends on the time to execute the critical section, and the time to pass the CS token from one processor to another. In Figure 15, we plot the amount of time that the token is busy (either in use by a processor or in transit due to a request) when the load is 50%. In Figure 16, we plot average time per critical section entry when the load is 100%. Under both low and high loads, the fixed-tree algorithm imposes a significantly larger time overhead than the path-compression algorithms do. The fixed-tree algorithm will saturate in its ability to serve the lock well before the path-compression algorithms do, and while in saturation will serve critical section requests at a lower rate. The path compression algorithms send the token almost directly to the next processor to enter, while the fixed-tree algorithm requires the token to follow the return path which it traveled over.

4.2 Non-Stationary Priorities

We ran a set of experiments to test the performance of the distributed priority locks when the request priority distribution is non-stationary (i.e, similar to that encountered with earliest deadline scheduling). In our first set of experiments, requests are uniformly generated among all of the processors, and we varied the load and the number pf processors. For non-stationary priorities, the load has a large effect on the number of messages sent. For this reason, we plot the number of messages required per critical section entry against the request load with 40, 80, and 160 processors in Figures 17 through 19.

When the load on the critical section is low (less than 100%), the path-compression algorithms require significantly fewer messages than the fixed tree algorithm, with the double-link algorithm having a slight edge over the single-link algorithm. When the load is 100%, the fixed tree algorithm

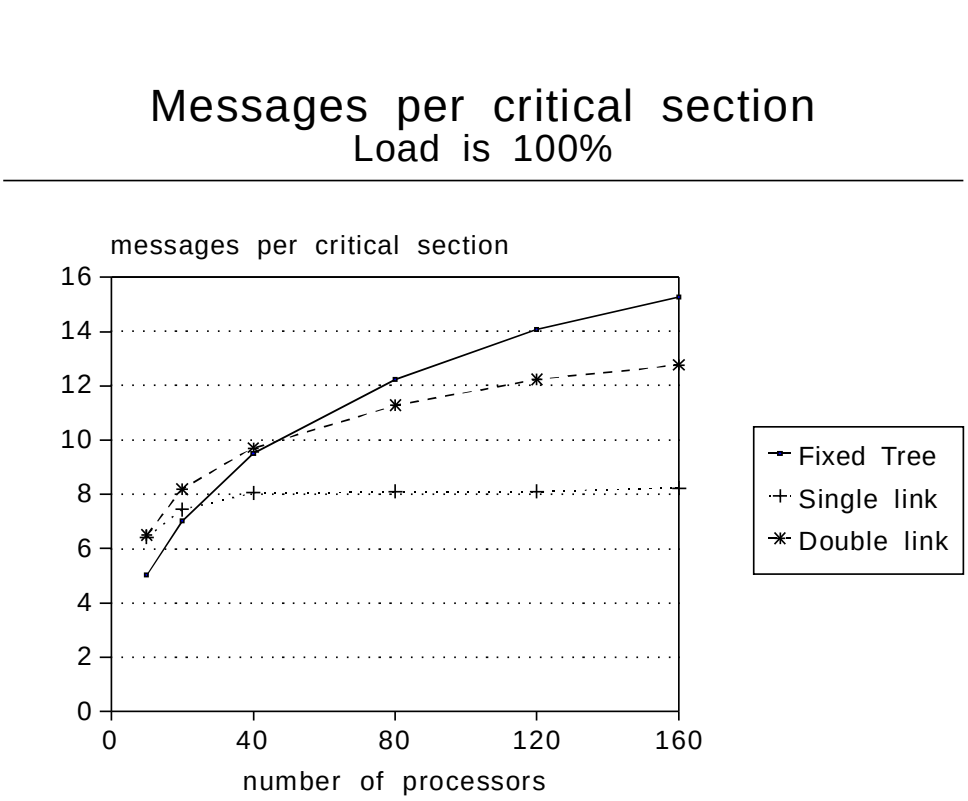

Figure 11: Number of messages per critical section entry, 100% load, uniform requests and stationary priorities.

requires fewer messages than the path compression algorithms. As the load increases beyond 100%, the path-compression algorithms require correspondingly more messages, while the fixed-tree algorithm actually requires fewer messages. When the load is less than 100% the number of messages per critical section entry increases logarithmically with the number of processors. When the load is 100% or greater, the number of messages that the path-compression algorithms require increases linearly with the number of processors, while the fixed-tree algorithm, has only a logarithmic increase.

The path compression algorithms require many messages when the load is high because the waiting structures become large, and new requests no longer have a good starting point. The guaranteed $O(logn)$ performance of the fixed-tree algorithm ensures that it never suffers from a high message passing overhead. However, the fixed-tree algorithm requires fewer messages than the path compression algorithms *only if the long-term average load is 100% or greater.* Since the simulators present a stochastic workload to the algorithms, the simulator often presents short periods of high demand to the algorithms when the load is less than 100%. In spite of the increased number of messages required by the path compression algorithms during these periods, the path compression algorithms still require fewer messages per critical section entry over the long term. The fixed-tree algorithm can be counted upon not to impose an excessive message passing overhead when the lock is a severe serialization bottleneck, but in this case the system has many problems beyond message passing overhead.

We also collected statistics about the time overhead imposed by the distributed priority lock algorithms under non-stationary priorities. The execution time overhead characteristics of the algorithms are essentially the same as those depicted in Figures 15 and 16 (to save space we do not repeat these figures).

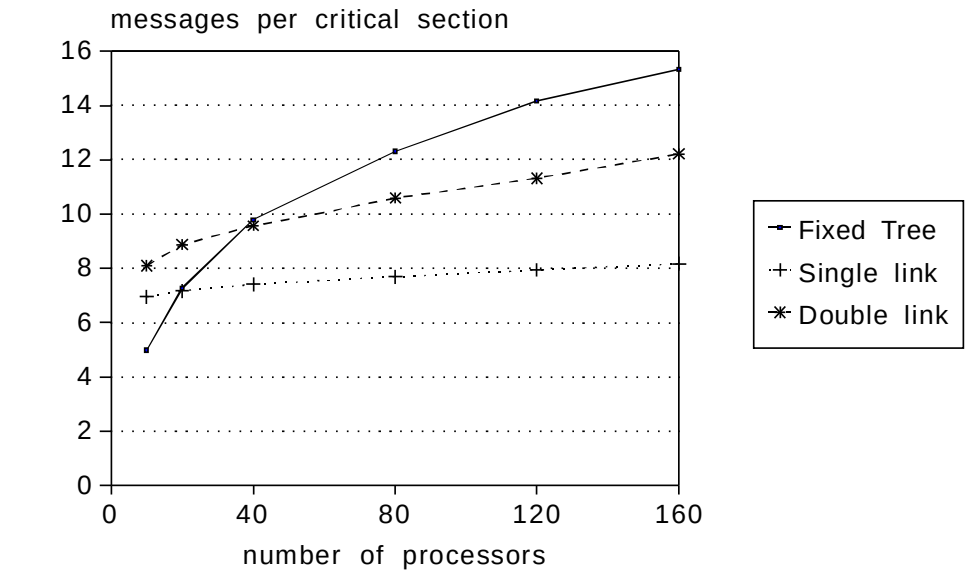

Figure 12: Number of messages per critical section entry, 200% load, uniform requests and stationary priorities.

We tested the effect of hot spots on the message passing overhead of the algorithms. Our results are similar to those encountered encountered in the stationary-priority experiments. When a small set of hot processors generate the requests, the path compression algorithms adapt to the changing request patterns better than the fixed tree algorithm does. however, the effect is not large. An example performance chart with 160 processors is shown in Figure 20.

5 Conclusion

We have presented three algorithms for prioritized distributed synchronization. Two of the algorithms use the path compression technique of Li and Hudak for fast access and low message passing overhead. The third algorithm uses the fixed-tree approach of Raymond. Each of these algorithms have a low message passing and space overhead. The $O(\log n)$ message passing overhead per request and the $O(\log n)$ bits of storage overhead per processor make the algorithms scalable.

To evaluate the algorithms, we made a simulation study. We examined the performance of the algorithms, in terms of message passing and time overhead, under two types of request priority distributions. The request priorities could be stationary (i.e., the priority is the importance of the task), or non-stationary (i.e., the priority id the deadline of the task). We found that when the request priority distribution is stationary, the single-link algorithm is best overall, but that the fixed-tree algorithm requires fewer messages when the load is high and the system is small. When the request priority distribution is non-stationary, the fixed-tree algorithm requires significantly fewer messages than the path compression algorithms when the long-term load on the critical section is 100% or greater. The double-link algorithm has better performance when the load is less than 100%, or when minimizing execution time overhead is more important than minimizing

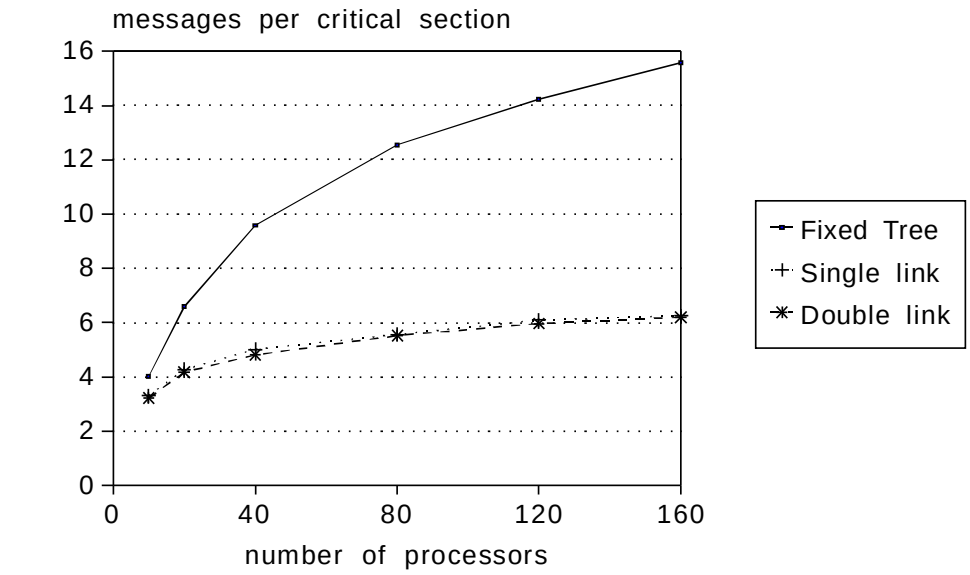

Figure 13: Number of messages per critical section entry, 50% load, hot spots in the requests and stationary priorities.

message passing overhead.

References

[1] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, 1991.

[2] O.S.F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Comm. of the ACM*, 26(2):146–147, 1983.

[3] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] Y.I. Chang, M. Singhal, and M.T. Liu. An improved $o(\log(n)$ mutual exclusion algorithm for distribtued systems. In *Int'l Conf. on Parallel Processing*, pages III295–302, 1990.

[5] T.S. Craig. Queuing spin lock alternatives to support timing predictability. Technical report, University of Washington, 1993.

[6] D. Ginat, D.D. Sleator, and R. Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 31:3–5, 1989.

[7] A. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *The Journal of Parallel and Distributed Computing*, 9:77–82, 1990.

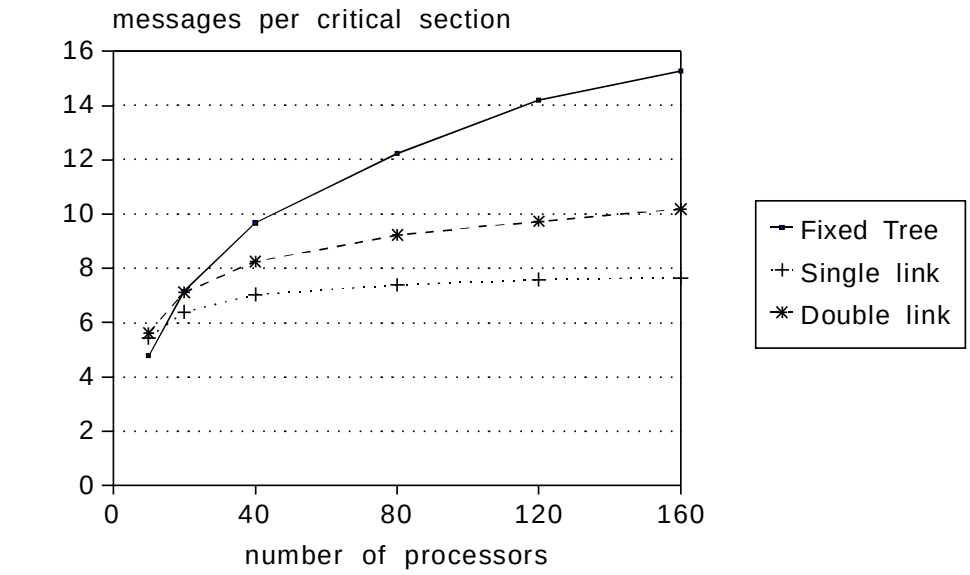

Figure 14: Number of messages per critical section entry, 100% load, hot spots in the requests and stationary priorities.

[8] K. Harathi and T. Johnson. A priority synchronization algorithm for multiprocessors. Technical Report tr93.005, UF, 1993. available at ftp.cis.ufl.edu:cis/tech-reports.

[9] D.V. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi. Scalable coherent interface. *Computer*, 23(6):74–77, 1990.

[10] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. on Computers*, 40(9):994–1004, 1991.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[13] M. Maekawa. A sqrt(n) algorithm for mutual exlcusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.

[14] E.P. Markatos and T.J. LeBlanc. Multiprocessor synchronization primitives with priorities. Technical report, University of Rochester, 1991.

[15] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[16] M.L. Neilsen and M. Mizuno. A dag-based neilsen for distributed mutual exclusion. In *International Conference on Distributed Computer Systems*, pages 354–360, 1991.

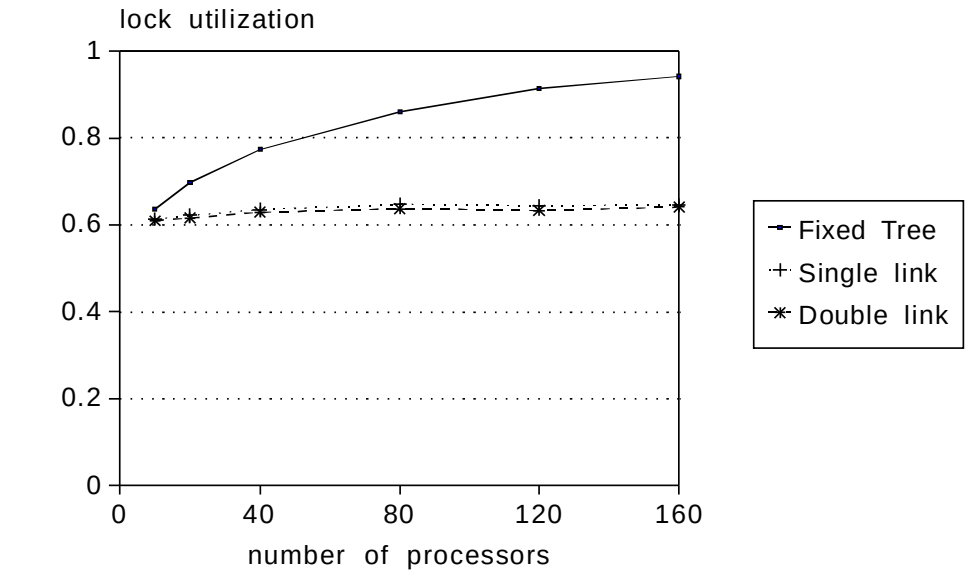

Figure 15: Lock utilization under a 50% load.

[17] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Systems*, 7(1):61–77, 1989.

[18] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. of the ACM*, 24(1):9–17, 1981.

[19] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[20] M. Trehel and M. Naimi. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. In *IEEE Phoenix Conference on Computers and Communications*, pages 36–39, 1987.

[21] M. Trehel and M. Naimi. An improvement of the log(n) distributed algorithm for mutual exclusion. In *Proc. IEEE Intl. Conf. on Distributed Computer Systems*, pages 371–375, 1987.

[22] T.K. Woo and R. Newman-Wolfe. Huffman trees as a basis for a dynamic mutual exclusion algorithm for distributed systems. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems*, pages 126–133, 1992.

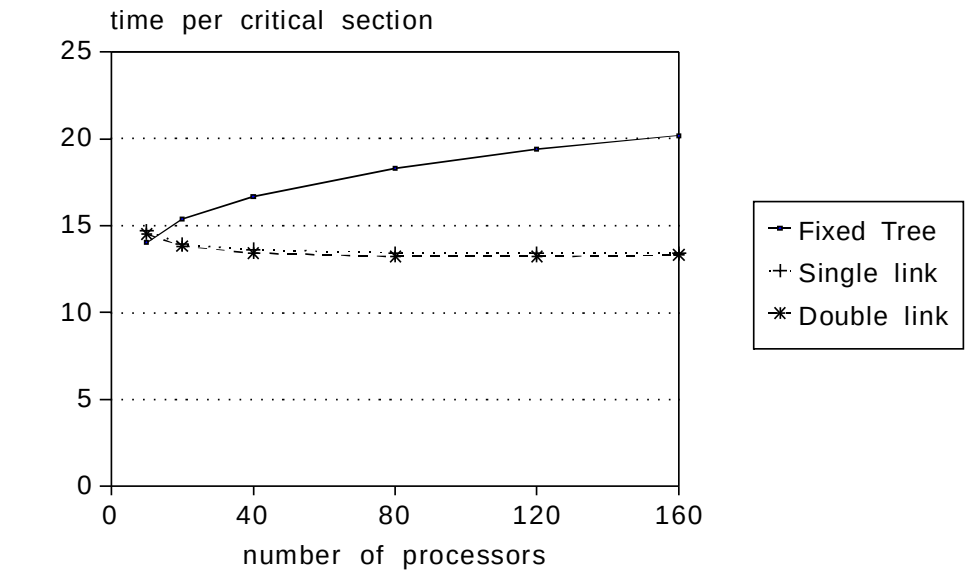

Figure 16: Time per critical section entry under a 100% load.

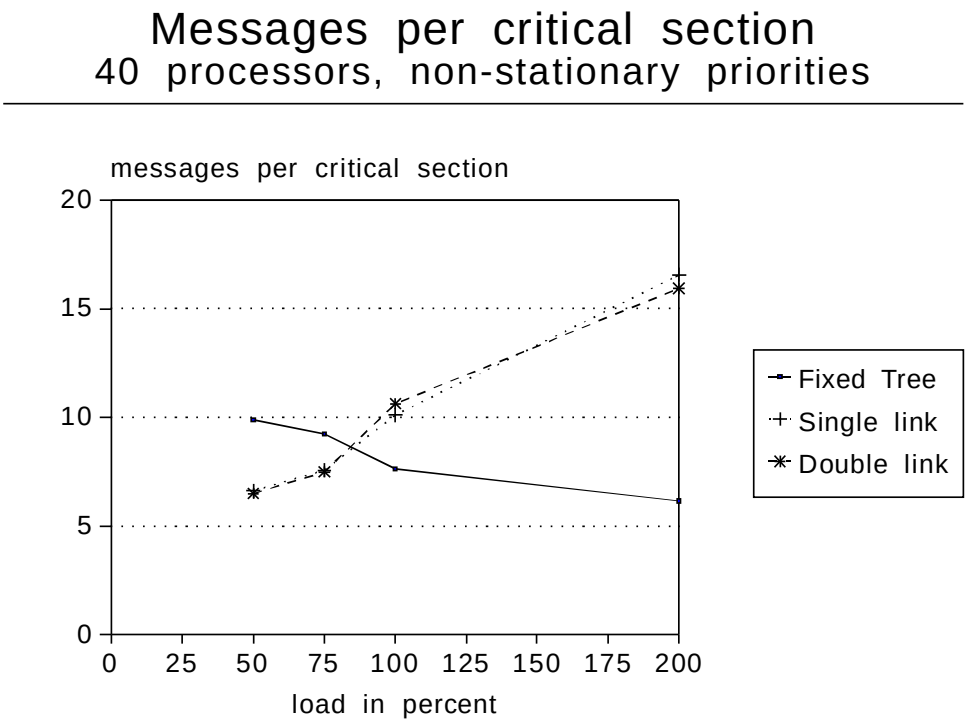

Figure 17: Number of messages per critical section entry, 40 processors, uniform requests and non-stationary priorities.

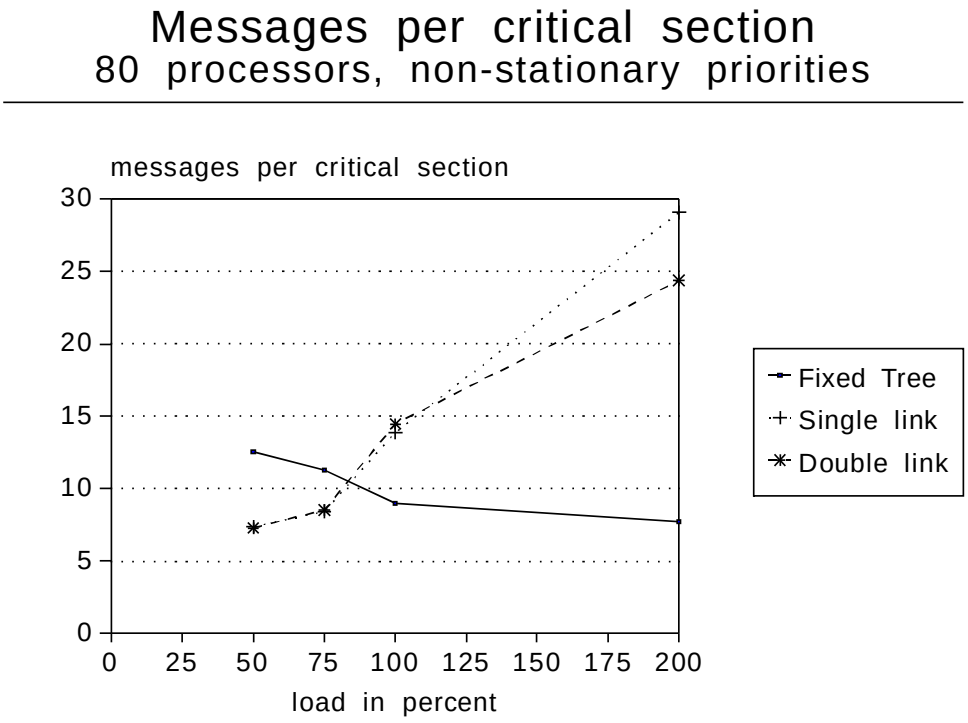

Figure 18: Number of messages per critical section entry, 80 processors, uniform requests and non-stationary priorities.

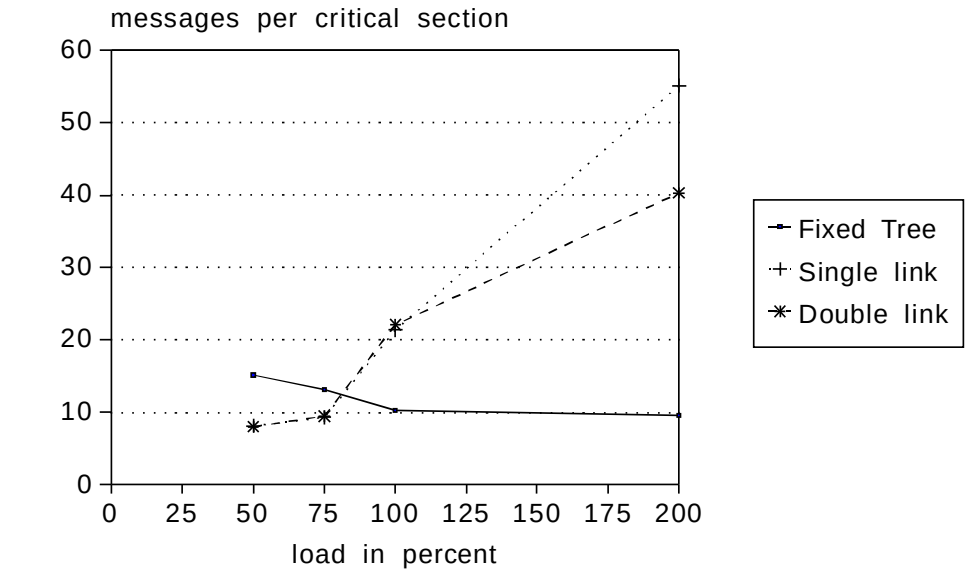

Figure 19: Number of messages per critical section entry, 160 processors, uniform requests and non-stationary priorities.

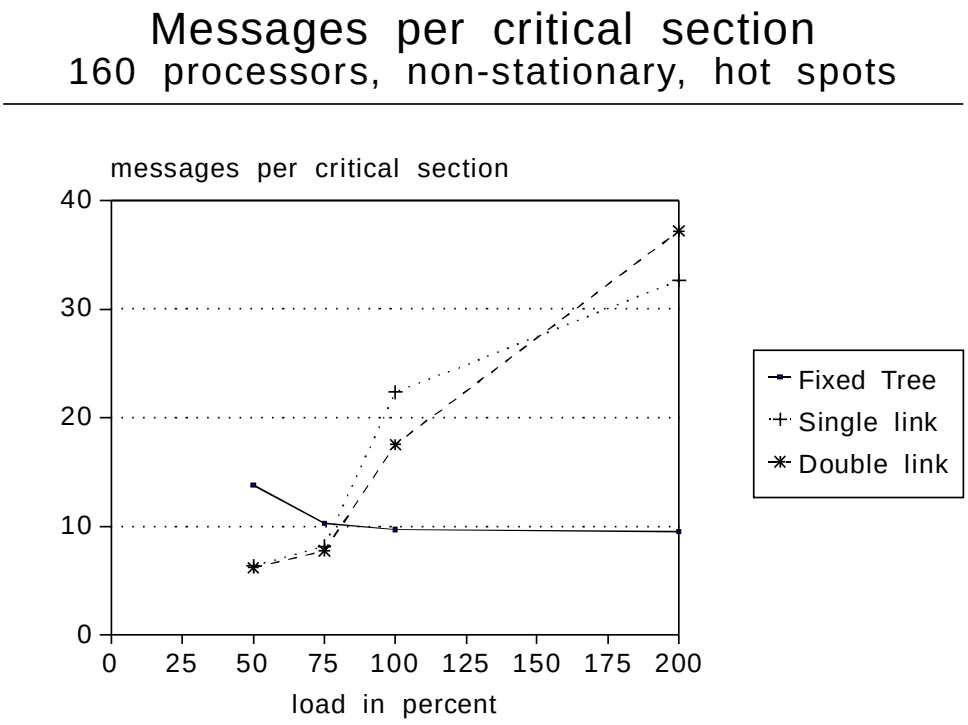

Figure 20: Number of messages per critical section entry, 160 processors, hot spot requests and non-stationary priorities.