A Scalable Real-Time Synchronization Protocol for Distributed Systems

Injong Rhee *  Graham R. Martin
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, U.K

Abstract
A distributed protocol is proposed for the synchronization of real-time tasks that have variable resource requirements. The protocol is simple to implement and is intended for large-scale distributed or parallel systems in which processes communicate by message passing. Critical sections, even when nested, may be executed on any processor. Thus, given an adequate number of processors, the execution of critical sections can be completely distributed. More significantly, since the protocol enables the distributed allocation of critical sections, the benefits of various allocations can be analyzed and the system optimized to provide minimal blocking. This has important application in global optimization techniques for allocating large numbers of hard real-time tasks in multiprocessor systems.

1 Introduction
In concurrent systems, processes often share system resources, such as data files, semaphores and I/O devices. These resources usually require a safety condition that no two processes use the same resource at the same time. Real-time tasks are no exception. Any realistic real-time scheduling technique should incorporate a scheme to synchronize simultaneous attempts to access resources by concurrent tasks, while avoiding a deadlock. A schedulability test for hard real-time tasks should also take into account the blocking time of a task caused by such synchronization, which is informally the total time period that the task is blocked before finishing critical sections where it uses resources. The goal of real-time synchronization is often to minimize the worst-case blocking time of each task, especially that of a high priority task. The problem is further complicated by the underlying CPU scheduling algorithm being priority-driven and preemptive. Synchronizing tasks with different priorities may introduce priority inversion, where higher priority tasks wait for a lower priority task to finish using resources [7], possibly resulting in an unbounded blocking time.

This paper investigates the real-time synchronization problem in a priority-driven preemptive scheduling environment, especially in distributed or parallel systems where tasks are statically bound to a processor and communicate through messages.

*contact author: email: rhee@dcs.warwick.ac.uk, Phone: 44-41-203-523290, Fax: 44-41-203-522714

The advent of large-scale multiprocessor systems provides the potential to schedule a large number of hard real-time tasks that may not be schedulable in a small-scale system. To harness this potential, the underlying real-time synchronization protocol must provide a scalable and flexible solution to minimize the worst case blocking time. However, existing techniques such as in [5, 4, 8] do not scale very well. The multiprocessor priority ceiling protocol (MPCP) of Rajkumar et al (and also the protocol of Sun et al [8]) forces all the (nested) sections of tasks in the transitive closure of the conflict relation1 to be executed in a single synchronization processor. In a large system which may contain hundreds or thousands of tasks, these centralized synchronization processors may introduce performance bottlenecks. Though the protocol later developed by Rajkumar [4] for shared memory multiprocessor systems allows critical sections to be executed on any processor, it assumes that critical sections are not nested.

In this paper, we develop a simple protocol that distributes the execution of critical sections over a number of synchronization processors. The protocol imposes no restriction on the allocation of critical sections2, and also allows critical sections to be nested. As an adequate number of processors are available, the execution of critical sections can be completely distributed and the blocking time reduced accordingly.

To facilitate comparison, let us assume that tasks access one (nested) critical section. We define the following terms. Two tasks p and q are said to be directly in conflict if they require the same resource, and to be indirectly in conflict if there exists a sequence of tasks p1, p2, ..., pk, such that p and p1; p1 and p2; and pk and q are directly in conflict respectively. If tasks i and j are directly in conflict, and i has a higher priority than j, then we denote this by j <i.

The following example illustrates the performance advantage of our protocol over the MPCP. Suppose that there are tasks i, j and k, such that j <i k <i i, and k has a higher priority than j, but is not directly in conflict with j.

1Two (nested) critical sections conflict with each other if they require the same resource.

2We say that a critical section is allocated to a synchronization processor if it is executed in that processor.
In the MPCR, all the tasks that are directly or indirectly in conflict execute their critical sections in a single synchronization processor, so, i, j, and k are forced to execute their critical sections in the same synchronization processor (say P). By priority ceiling\(^3\), k can be blocked by j although j and k are not directly in conflict. Figure 1 shows the situation where k is blocked by j because j holds a resource that i requires (i.e., the priority of k is lower than the priority ceiling of the resource). In the uniprocessor environment, this blocking prevents a chain of blocking where i is blocked by both j and k at the same time, and thus has to wait for both j and k to complete their critical sections.

However, in the multiprocessor environment, we can reduce the blocking time of k by allocating the critical section of k to a different processor Q (see Figure 2). Since k and j can now run in parallel, j doesn’t have to block k in order to reduce the blocking time of i. Note that the blocking time of i is still bounded to the maximum of the critical section times of j and k.

Not every allocation of critical sections can result in reduction of the blocking time. As an allocation becomes more distributed, a task can be blocked by tasks in other processors, which creates more opportunity for lower priority tasks in the same processor to interfere with the task. Suppose that a task j′ in P requires a resource with the priority ceiling higher than i’s priority, but is not directly in conflict with i. Suppose that i in P is blocked by k in Q when attempting to acquire a resource, already taken by k. In the meantime, j′ in P may enter its critical section. When i finally becomes unblocked and ready to enter its critical section, it may again have to wait for j′ to finish because j′ has a higher priority ceiling than i’s priority (see Figure 3).

Clearly, there are tradeoffs among various allocations of critical sections. The principle advantage of our protocol is that by enabling the tradeoffs of many different allocations to be evaluated, it can be used to minimize the blocking time. This is unlike the MPCR where the allocation of critical sections is limited by conflict relation of critical sections. In fact, the upper bound on the blocking time in our protocol subsumes that in the MPCR with only a marginal difference. That is, when given an appropriate allocation and task set, the protocol provides a smaller upper bound on the blocking time than the MPCR. Furthermore, when given the same allocation as the MPCR, the protocol provides approximately the same upper bound on the blocking time. The difference is at most that between the largest and the smallest critical section times in the system. A global optimization technique, such as the simulated annealing based technique [9], can be used with our protocol to find a particular allocation of tasks and critical sections in a multiprocessor that satisfies the time constraints of the system.

As an arbitrary allocation of nested critical sections may give rise to a deadlock, a distributed resource preclaiming technique is adopted to prevent it. Resource preclaiming is a resource allocation technique that requires each task, before entering a (nested) critical section, to acquire all the resources needed within that critical section. In the protocol, the control of resource allocation is also completely distributed, so that for each resource in the system, there is one resource manager which deals with the requests pertaining only to that resource. Each resource manager strictly uses

\(^3\)We assume that readers are familiar with the priority ceiling protocol in [7, 5]
local information, and communicates only with the tasks. Resource allocation techniques based on resource preemption have been extensively studied for asynchronous distributed systems (see [6] for review).

The remainder of the paper is organized as follows. Section 2 defines the system model and the problem formally, and Section 3 describes the protocol. Detailed analysis of our protocol and comparison with other techniques can be found in Sections 4 and 5.

2 Model and Problem Definition

Informally speaking, we model each outermost critical section in a task as a separate subtask, called the user. Some techniques to decompose a task into several users are discussed in [8]. It is assumed that a user is becomes active when the task that the user is from tries to execute the corresponding critical section. Since we are only interested in specifying the synchronization protocol, we focus on modeling users (not tasks). We use the timed I/O automaton [1] to describe the system model.

In the system being modeled, there are finite and fixed sets of resources \( R = \{ r_1, r_2, \ldots, r_n \} \), and users that need a subset of \( R \) at various times, for their execution. There is a set of resource managers, each of which is assigned to one resource in \( R \). Let \( \mathcal{R}_i \) be the resource requirement of user \( i \). If for any users \( i \) and \( j \), \( \mathcal{R}_i \cap \mathcal{R}_j \neq \emptyset \), then we say that \( i \) conflicts with \( j \) and vice versa.

There is a set of processors \( \mathcal{P} \) \(^4\), and each processor is an automaton that consists of a finite number of users, a finite number of resource managers and a scheduler. We assume that a user \( i \) (and a resource manager and a scheduler) is statically bounded to a processor \( P_i \) (i.e. not changed during computation), and users, resource managers and the scheduler in a processor are not modeled as separate automata. They are simply part of the processor automaton. We require that it is possible to partition the actions of a processor automaton into the users, the resource managers and the scheduler in the processor. Users, resource managers and the scheduler in the same processor communicate through shared variables while those in different processors communicate by passing messages. We assume that messages are delivered in the order they are sent (i.e. FIFO) and the message delay is within \([0, \alpha]\).

The states of each user \( i \) include thinking, runnable, running and blocked, and are denoted by a variable \( \text{state}_i \). Only the scheduler in \( P_i \) may update \( \text{state}_i \). The code (i.e. a partition of a processor automaton) of a user is executed only when \( \text{state}_i = \text{running} \). The transition diagram of \( \text{state}_i \) is shown in Figure 4. We say that user \( i \) is preempted if \( \text{state}_i \) changes from running to runnable; arrives if \( \text{state}_i \) changes from thinking to running or runnable or blocked; and finally, leaves if \( \text{state}_i \) changes from running to thinking.

The input actions of each processor \( P \) consist of \{Arrival\} for all users \( i \) in \( P \), where \( \text{Arrival} \) deals with the arrival of a user \( i \) in processor \( P \). The output actions of each processor \( P \) consist of \{Leaving\} for all users \( i \) in \( P \), where \( \text{Leaving} \) deals with the leaving of a user \( i \) in processor \( P \).

The states of a user \( i \) are partitioned into four regions. The user enters the thinking region by executing \( \text{Arrival}_i \), where the user is vying for its required resources. Then, once acquiring all the required resources, the user enters the critical region. It remains running in the region for some bounded time period while using the resources. When finished with the resources, it enters the remainder region, where it relinquishes the resources. Then the user enters the remaining region by executing \( \text{Leaving}_i \).

While in the critical region, user \( i \) does not release any resource. This abstracts resource preemption. The total time that the user is running in the period between its last arrival and its subsequent leaving is at most \( \text{CS}_i \), which is the critical section time of user \( i \).

We assume that the priorities of users are set up by some priority assignment function \( \text{Pr} \) that maps each user in the system to a unique integer, called the priority. \( \text{Pr} \) satisfies the following condition: for any two users \( i \) and \( j \) in the same processor or that conflict with each other, then \( \text{Pr}(i) \neq \text{Pr}(j) \), and for any users \( i, j, k \), if \( \text{Pr}(i) < \text{Pr}(j) \) and \( \text{Pr}(j) < \text{Pr}(k) \), then \( \text{Pr}(i) < \text{Pr}(k) \). If \( \text{Pr}(i) < \text{Pr}(j) \) and \( i \) conflicts with \( j \), then we denote this by \( i <_c j \). We define the priority ceiling of a resource \( r \in \mathcal{R} \) to be \( \text{max}\{\text{Pr}(i) : \text{for all users, s.t. } r \in \mathcal{R}_i\} \), and the priority ceiling of a user \( i \) to be the maximum of the priority ceilings of all the resources \( r \in \mathcal{R}_i \), denoted by \( \text{ceil}(i) \).

A multiprocessor real-time synchronization protocol consists of the automata of all the processors in \( \mathcal{P} \), whose execution satisfies the following conditions:

- **(Exclusion)** When a user \( i \) is in the critical region, there is no conflicting user of \( i \) in the critical region.
- **(No-deadlock)** (modified from [3]) (1) If in some state, some user is in the critical region, no user is in the critical region, and subsequently no user arrives, then subsequently some user enters the critical region, and (2) if in some state, some user is in the exit region, and subsequently no user arrives, then subsequently some user enters the remainder region.

---

\(^4\)This processor is called the synchronization processor in [5].
The blocking time of a user $i$, denoted by $B_i$, in a multiprocessor real-time synchronization protocol is the maximum time period between when $i$ arrives and when it leaves (the critical region) subsequently.

3 The Protocol

3.1 Informal Description

For convenience of presentation, we assume that resource managers are bounded to different processors from users. It is straightforward to compose resource managers with users and the scheduler when they are bounded to the same processor.

Basically, in the protocol, each resource manager $rm_i$ maintains a priority queue $q_i$ that contains (the IDs of) users currently requesting resource $k$. The scheduler implements priority-driven preemptive scheduling, and interacts with resource managers for the resource needs of users in the processor of the scheduler. The scheduler ensures that the task running in the processor is the one with the highest priority among all those runnable.

When a user $i$ arrives and enters the trying region with resource requirement $R_i$, the scheduler in $P_i$ blocks $i$, and then requests all the resources in $R_i$ on behalf of $i$ by sending a request $i$ message to each $rm_r$, $r \in R_i$. Receiving the message, $rm_r$ adds $i$ in $q_r$ in order of $i$'s priority. $rm_r$ grants resource $r$ to the user at the head of $q_r$ by sending a grant message to the scheduler of the user. When the scheduler receives a grant $i$ message from all $rm_r$, $r \in R_i$, it unblocks $i$ and lets $i$ enter the critical region. When entering the exit region, $i$ releases its resources by prompting the scheduler to send a release $i$ message to each $rm_r$. Receiving a release $i$ message, $rm_r$ removes $i$ from $q_r$ and grants $r$ to the next user at the head of $q_r$.

This simple scheme, however, may lead to a deadlock because it is possible that some lower priority user $j$ is granted the resource that $i$ is requesting, while $j$ is waiting for the resource that $i$ is granted. To prevent the deadlock, we let the lower priority user give up its resource. This is done as follows. When $rm_r$ receives a request $i$, if $r$ is already granted to a lower priority user $j$, it sends a preempt $r$ to the scheduler of $j$. If $j$ is not in the critical region, the scheduler returns $r$ immediately by sending a return $r$ message to $rm_r$. If $i$ is in the critical region, by the assumption that users are running in the critical region for a finite time, it will eventually leave the region and send release $r$ messages. Receiving either a return $r$ or release $r$ message, $rm_r$ grants $r$ to the user in $q_r$ with the highest priority (i.e., $i$). This way, a user with the highest priority among its conflicting users in the trying region always enters the critical region. This solves the deadlock problem.

However, since users still have to wait for lower priority users in the critical region, this may lead to another problem, called the unbounded priority inversion problem [7], where a user $i$ may be blocked indefinitely because of a lower priority conflicting user $j$ in the critical region. This can happen when $j$ is preempted by another user $k$ in $P_j$, such that $k$ has a higher priority than $j$, but a lower priority than $i$. Thus, $i$ has to be blocked for the critical section times of $j$ and $k$. A chain of blocking may form as another user of a lower priority than $i$, but of a higher priority than $k$ blocks $k$, and so on.

In order to bound priority inversion, we adopt priority ceiling. We require that a user $i$ enters the critical region only if there is no user in $P_i$ in the critical region with a priority ceiling higher than the priority of $i$. As the condition ensures that any user $k$ whose priority is lower than $i$ does not preempt $j$ running in the critical region, it prevents a chain of blocking among users in the same processor. Thus, a user in a processor is blocked by no more than one lower priority user in the same processor at any time that the user is in the trying region.

3.2 Formal Description

Figure 5 shows the trying and exit regions of user $i$, and part of the code of the scheduler in $P_i$ concerned with interacting with $i$. Figure 6 shows the code of resource manager $rm_i$.

The automata in Figures 5 and 6 are composed with a network automaton Net, for communication. Net is an automaton whose state contains a FIFO queue $channel_{i,j}$ for all tasks and schedulers $i$ and $j$, holding the messages sent from $i$ to $j$ but not yet received by $j$. $channel_{i,j}$'s input and output actions are $Send_i(m, j)$ and $Receive_i(m, j)$, respectively, for every message $m$. When $Send_i(m, j)$ occurs, $m$ is enqueued in $channel_{i,j}$. $Receive_i(m, j)$ can occur when $m$ is at the head of $channel_{i,j}$ and results in $m$ being dequeued. Thus, there is an input action $Receive_i(m, j)$ of a task $i$ (or a scheduler) composed with an output action $Send_i(m, j)$ of Net, and an output action $Send_i(m, j)$ of $i$ composed with an input action $Receive_i(m, j)$ of Net. This definition of Net is borrowed from [10, 1].

The following variables are used in user $i$, its scheduler and resource manager $rm_i$.

- $region_i$: a variable that indicates the current region that user $i$ is in or about to be in.
- $critical_i$: a boolean variable that indicates whether user $i$ is in the critical region.
- $grant-set_i$: a set that contains the IDs of resource managers from which the scheduler has received a grant message and hasn't yet received a preempt message for user $i$.
- $msg$s$_j$: a FIFO queue of messages for $j$, but not yet sent. Initially empty.
- $grant_id$: the ID of the user for which $rm_i$ sent a grant$_i$,$grant_id$, but hasn't received a release$_i$,$grant_id$ or sent a preempt$_i$,$grant_id$ since it sent the grant message. Initially 0. We assume that no user has ID 0.
- preempt_id$: the ID of the user for which $rm_i$ sent a preempt message, but hasn't received a return or release message since it sent the preempt message. Initially 0.
The following procedures are used, but not specified in the code. Note that procedure `max_priority_task()`, which is described in the following, is called to enforce priority ceiling.

- `send(m, j)`: enqueues message `m` in FIFO order in `msgs[j]`.
- `dequeue(m, msgs[j])`: removes the head of `msgs[j]` and returns it as `m`.
- `priority_enqueue(i, qj)`: adds `i` to a priority queue `qj` by `priority(i)`.
- `scheduler(i)`: returns the ID of the scheduler of user `i`.
- `max_priority_task()`: returns the ID of the user `j` with the highest priority among those runnable or running that satisfies the following: if `(region_i = critical)` and `(criticali = false)`, then there is no user `k` in `I_j` s.t. `ceil(k) > Pr(j)` and `criticali = true`.

### 3.3 Correctness Proofs

Correctness proofs are organized into two theorems for the exclusion and no-deadlock conditions respectively. For convenience of presentation, we say that a user `i` receives a message `grant0`, `preempt`, if the scheduler of user `i` receives `grant0`, `preempt`, and also that a resource manager receives `request`, `return`, or `release` from user `i` if the message is sent by the scheduler of the user. We use `send` in the similar way.

**Theorem 3.1** When a user `i` is in the critical region, there is no conflicting user `j` in the critical region.

**Proof** Since `i` enters the critical region only if `region_i` is `critical` (Line 3), we need to check only when it becomes `critical`. According to the code, `region_i` becomes `critical` only when `grant_set` is equal to `R_i` (Line 4). `grant_set` adds a member only when it receives a `grant0`, message for any `k` in `R_i` (Line 42). `grant_set` loses a member `k` only when (1) `release` is sent when `i` enters the exit region (Line 32), or (2) when `return` is sent when `i` is not in the critical region (Line 55). According to the code of `rm`, `grant` is sent only when (1) `grant_id = 0` (Line 76), or (2)

[Figure 5: The trying and exit regions of user `i` and the code of the scheduler]
it receives a release, for some j s.t., i and j conflict with each other (Lines 92, 99), or (3) it receives return; (Line 106). Since grantjad becomes 0 only when rmj receives a release; (Line 88), or initially when no user is in the critical region, we can conclude that if k ∈ grant-set;, then k ∉ grant-set;j. The theorem follows.

Theorem 3.2 (No-deadlock) (1) If in some state, some user is in the trying region, no user is in the critical region, and subsequently no user enters the critical region, and (2) if in some state, some user is in the exit region, and subsequently no user arrives, then subsequently some user leaves.

Proof: (Sketch) The second part of the theorem can be shown easily from the code. We only show the first part.

Let t be the time that some user is in the trying region, no users are in the critical region, and subsequently no user arrives. At any time, if some user is in the trying region, some user must have the highest priority among its conflicting users in the trying region. Let j be such user at time t. Then, j must be at the head of all qk, k ∈ Rj at time t, or grantjad = j, or preemptjad = j.

Let t’(≤ t) be the last time that Receive1 (requesti;) is executed in a rmj, t ∈ Rj. At time t’, either (1) some user i of a lower priority than j is granted the resource and is in the trying or critical region, or (2) some user(s) i of a higher priority than j is either in qj or has currently been granted the resource.

Case 1: i < j. By the protocol, i eventually receives a preemptjad; message because i < c j. If i is not in the critical region, it sends a returni; message, which triggers rmj to send a grant message to j or some user with a higher priority than j. If i is in the critical region, it will leave the region in a finite time before t, and send a release; to rmj, prompting rmj to send a grantjad; to j or some user with a higher priority than j. Because j is the highest priority user among all its conflicting users at t, users with a higher priority than j must have entered and left the critical region (i.e. sent a release message to rmj) by time t.

Case 2: j < c i. Similarly, because j is the highest priority user among all its conflicting users by time t, users with a higher priority than j must have entered and left the critical region (i.e. sent a release message to rmj).

In either case, rmj will receive either a release or return message while j is the highest priority user among its conflicting users in the trying region. Let t’ be the time that Receive1 (requesti;) is executed in a rmj, t ∈ Rj. At time t’, either (1) some user i of a lower priority than j is granted the resource and is in the trying or critical region, or (2) some user(s) i of a higher priority than j is either in qj or has currently been granted the resource.

4 Performance Analysis

In this section, we bound Bk, the blocking time of a user i.

In the analysis, we make the following assumptions:

1. Each user i arrives at every time period Ti (if it leaves before its next period).

2. The maximum message delay d includes the time period between when a scheduler (a re-
source manager) \( p \) executes an output action \( \text{Send}_p (m, i) \), and a resource manager \( r m_j \) (or a scheduler) executes \( \text{Receive}_c (m) \). That is, message delay includes the time for handling messages.

We say that a user \( i \) is \textit{eminent} if \( i \) has the highest priority among its conflicting users in the trying region, and that a user \( i \) \textit{blocks} a user \( j \) if one of the following conditions is satisfied:

1. \( i \) is in the critical region, \( j \) is in the trying region, \( j \) is eminent, and \( i \) conflicts with \( j \).
2. \( i \) is in the critical region, \( j \) is in the trying region, \( j \) is eminent, and no conflicting user of \( j \) is in the critical region, \( P_i = P_j \), \( Pr(i) < Pr(j) \), and \( Pr(j) < \text{ceil}(i) \).
3. \( i \) blocks a user \( k \), s.t. there is a sequence of users \( j < c, j < ... < c \), \( k \) that are in the trying region.

A priority chain of a user \( u_j \) is defined to be a sequence of users \( u_1, u_2, \ldots, u_l \) such that \( u_{l+1} < c < u_l \), \( 1 \leq k < i \). We say that a user \( i \) is \textit{local} if its resource requirement doesn’t contain a resource that is required by the users in the other processors. We say that a user \( i \) is \textit{completely local} if all the users in every priority chain of \( i \) are local. The following terms are also used in describing \( B_i \).

- \( \pi_i \) is the set of all the users in every priority chain of user \( i \).
- \( \eta_i \) is the set of users with a higher priority than some user \( j \) in \( \pi_i \cup \{i\} \) that are allocated to \( P_j \), but are not in \( \pi_i \cup \{i\} \). More precisely, \( \eta_i = \{ k | k j \in \pi_i \cup \{i\} \wedge P_k = P_j \wedge Pr(j) > Pr(k) \wedge P_i = \pi_i \cup \{i\} \} \).
- \( \lambda_i \) is the set of users \( l \) with a lower priority than \( i \), that satisfy one of the followings: (1) \( l < c \); (2) \( Pr(i) < \text{ceil}(l) \); (3) \( Pr(i) < Pr(l) \) and \( P_i \neq P_j \).
- \( L_i \) is the total time that users in \( \lambda_i \) are running while blocking \( i \) when \( i \) is eminent. In other words, \( L_i \) is the portion of the blocking time of \( i \) directly attributable to the users in \( \lambda_i \).

The following lemma (proof omitted) show that if \( i \) is not blocked forever and there is no preemption, then \( i \) enters the critical region almost immediately.

**Lemma 4.1** When \( i \) is in the trying region, if \( i \) is not blocked, and there is no user in \( \eta_i \) in the critical region, then \( i \) enters the critical region in \( 2d \).

The following is true by the definition of blocking.

**Lemma 4.2** If \( k \) blocks \( i \), then \( k \) is in \( \pi_i \) or \( \lambda_i \) or \( \cup_{j \in \pi_i} \lambda_j \).

Let us call the union of \( \pi_i \), \( \lambda_i \) and \( \cup_{j \in \pi_i} \lambda_j \) the \textit{blocking set} of \( i \).

The following theorem bounds \( B_i \). The proof is informal to ease presentation and understanding.

**Theorem 4.3**

\[
B_i \leq \sum_{j \in \pi_i} [B_i/T_j] (CS_j + L_j + 6d) + \sum_{j \in \pi_i} [B_i/T_h] CS_h + CS_i + L_i + 4d
\]

**Proof:** Lemma 4.1 implies that the blocking time of \( i \) is the sum of (1) the portion of the blocking time attributed to blocking caused by users in the blocking set of \( i \), (2) the portion attributed to the preemption of users in the blocking set of \( i \), and (3) some message delays.

First, we consider the blocking time attributed to blocking. By Lemma 4.1, only those users in the blocking set of \( i \) can block \( i \). The total time that a user \( j \) is in the trying region is at most \( [B_i/T_j] \) times the critical section time of \( j \) \( (CS_j) \). The total time that a user \( l \) is running while blocking \( j \) and \( i \) when \( j \) is eminent at most \( [B_i/T_j] L_j \). Thus, the blocking time attributed to blocking is at most \( \sum_{j \in \pi_i} [B_i/T_j] (CS_j + L_j) \). Note that the blocking of \( j \) by lower priority users \( l \) while \( j \) is not eminent is also considered here because there must be some user \( k \) in order for \( l \) to block \( j \) such that \( k \) is in a priority chain of \( j \) (and thus of \( i \)) and is blocked by \( l \) and also eminent.

Now we consider the blocking time attributed to the preemption of users in the blocking set. The preemption can happen among users in the blocking set, but the effect of those preemptions is already considered previously in the form of blocking because it was assumed that any user \( j \) in \( \pi_i \) blocks \( i \) whenever \( i \) and \( j \) are in the trying region at the same time. Thus, we need to consider the preemption only by those users in \( \pi_i \) and \( \lambda_i \) and \( \cup_{j \in \pi_i} \lambda_j \) that have a higher priority than \( j \), but are not in \( \pi_i \) (i.e. those in \( \eta_i \)). Note that the users that have a lower priority than \( j \) cannot preempt \( j \) or users in \( \lambda_i \) because of the priority ceiling rule in \texttt{max_prio_user()}(). Therefore, the blocking time attributed to preemption is at most \( \sum_{j \in \pi_i} [B_i/T_h] CS_h \).

We now consider the message delays caused by blocking. Note that no message delay is incurred by preemption. \( i \) will start waiting for \( j \) either (1) from when \( \text{request}_i \) arrives at \( r m_j \) \( k \in R_i \cap R_j \), in which case \( r m_j \) has not sent a \texttt{grant} message to \( i \), or (2) from when \( i \) receives a \texttt{preempt} message. Thus, the message delay caused by blocking, if \( j \) is not blocked by any lower conflicting users, is at most \( 4d \times (a) \) \( d \) time for a \texttt{return} message (in response to the \texttt{preempt} message) to be received; (b) \( d \) time for a \texttt{grant} message to be received by \( j \); (c) \( d \) time for a \texttt{release} message to be received; and (d) \( d \) time for a \texttt{grant} message to be received by \( i \).
The message delays that are incurred because a lower priority user $l$ blocks $j$ is at most $2d$: (1) $d$ time for a release message to be received, and (2) $d$ time for a grant message if received by $j$.

The following claims are used in bounding $L_j$.

Claim 4.4 During a period that a user $j$ is eminent, if $j$ is blocked by any (lower priority) conflicting user $l$, then (1) $l$ must have entered the critical region before or at time $t + 2d$ where $t$ is the time that $j$ became eminent; and (2) there is at most one such $l$ in each processor that blocks $j$ after time $t + 2d$.

Proof: (1) A resource manager grants the resource in the order of the priorities of the users from which it has received request messages. When a user $j$ enters the trying region at time $t$, its scheduler sends request messages to all $rm_j$, $k \in B_j$, which are received by time $t + d$. Thus, any lower priority users whose request message is received after time $t + d$ won’t be granted the resource until $rm_j$ receives a release message from $j$ (which happens after $j$ leaves the critical region). Since no grant message will be sent to $l$ after time $t + d$ while $j$ is in the trying and critical region, if $l$ blocks $j$, it must have entered the critical region before or at time $t + 2d$ (because at the very moment request$_l$; is received, $rm_j$ might have sent a grant$_l$ message, which will be received by $t+2d$).

(2) Since $l$ conflicts with $j$, the priority ceiling of $l$ must be at least the priority of $j$. Thus, no user in $P_j$ with a lower priority than $j$ can enter the critical region while $l$ blocks $j$ because of the priority ceiling rule in max_priority_user(). Since after $t + d$, no lower priority user that conflicts with $j$ can be granted the resource that $j$ requires until $j$ leaves the critical region, the lemma follows.

Claim 4.5 During a period that region$_j$ is critical, $j$ can be blocked only by at most one lower priority user in $P_j$, and after the user leaves the critical region, it doesn’t block $j$ again during the period.

Proof: If region$_j$ = critical, $j$ is granted all of its resources and thus, no lower priority conflicting user enters the critical region to block $j$. Thus, the only users that can block $j$ are those in $P_j$ with a priority ceiling higher than the priority of $j$. By the priority ceiling rule, no user with a lower priority than $j$ can enter the critical region after region$_j$ becomes critical. So any user that blocks $j$ now must have entered the critical region before region$_j$ becomes critical. Let $l$ be the user. Since $l$ has a priority ceiling higher than $j$, no user with a lower priority than $j$ can preempt $l$. So, there is no user in $P_j$ that is in the critical region and has a lower priority than $j$, and therefore, while $l$ blocks $j$, no other user with a lower priority than $j$ can block $j$.

The following terms are used in bounding $L_j$:

- $j$ is the set of all the users in $P_j$ that have a lower priority than $j$ and a higher priority ceiling than $j$’s priority.

Note that $\lambda_j = \delta_j \cup \psi_j$.

Lemma 4.6 $L_j \leq \max_{\forall i \in \psi_j} \{CS_i\} + \max_{\forall i \in \lambda_j} \{CS_i\}$

Proof: By Claim 4.5, after $t + 2d$, there is at most one lower priority conflicting user $k$ in each processor that blocks $j$. Thus, after running for $CS_k$, $k$ leaves the critical region (note that $L_j$ includes only the running time of $k$, but not its preempted time) and $k$ doesn’t block $j$ again until $j$ leaves the critical region. Then region$_j$ becomes critical. Then, by Claim 4.4, it can be blocked by at most one user $l$ in $P_j$. $l$ leaves the critical region after running for $CS_l$. Then, $l$ won’t block $j$ again, and $j$ enters the critical region.

The bound in Theorem 4.3 is conservative because it assumes that all the users in $\pi_i$ can block $i$ whenever they are in the trying region with $i$. However, $i$ is blocked by a user $j$ in $\pi_i$ if all the users in $i < \ldots < j$ are in the trying region at the same time. More detailed analysis may be required to tighten the bound.

Here, we consider one case that the bound can be tightened, namely when $i$ is completely local. Note that since $i$ is completely local, all the users in $\pi_i$ are also completely local, and by allocating all the managers of resources required by all the users in $\pi_i$ to the same processor $P_i$, we avoid message delays incurred in acquiring resources. The following claim is used to bound $B_i$ when $i$ is completely local.

Claim 4.7 If $i$ is completely local, and $i$ is blocked by a lower priority user $l$, (1) then $l$ must have entered the critical region before $i$ entered the trying region, and (2) there is at most one such $l$ before $i$ subsequently leaves the critical region.

Proof: (1) By way of contradiction, assume that the claim is false. Let $l'$ be the first lower priority user that enters the critical region after $i$ entered the trying region. Then $l'$ must be blocked by some user $j'$ in $P_i$ because $Pr(l') < Pr(i) - \epsilon$ if $i$ were not blocked in the trying region (i.e. granted all the resources in $R_i$), then $i$ would enter the critical region. Then, it should be true that $Pr(i) < ceil(l')$. However, since $Pr(l') < Pr(i)$ and by the priority ceiling rule, $l'$ cannot enter the critical region. This is a contradiction.

(2) Since $l$ blocks $i$, the priority ceiling of $l$ is as high as the priority of $i$. While $l$ is in the critical region, no user with a lower priority than $i$ can preempt or block $l$ (and block $i$) by the priority ceiling rule.

Theorem 4.8 If $i$ is completely local, then

$$B_i \leq \sum_{\forall j \in \pi_i} \{B_i/T_j\} \cdot CS_j + \max_{\forall i \in \pi_i} \{CS_i\}$$
Proof: Claim 4.7 implies that \( i \) is blocked by no more than one lower priority user \( j \) before it leaves the critical region and also that it is blocked by the user at most once. The total time period that the lower priority user \( j \) is running in the critical region is at most \( L = \max_j \{ L_{CSj} \} \).

If a user \( j \) in \( \pi_i \) is blocked by a lower priority user \( l' \), \( i \) can also be blocked by the user. If \( l' \) has a higher priority than \( i \), then \( j \) should be in \( \pi_i \) or \( \eta_j \). If \( l' \) has a lower priority than \( i \), then \( l' \) should be in \( \lambda_i \), of which case is already considered in \( L \). Thus, we only need to consider the blocking by users in \( \pi_i \) or \( \eta_j \). The time period that \( i \) is blocked by a user \( j \) in \( \pi_i \cup \eta_j \) is at most \([B_i/T_j]CS_j\).

As we assume that all users are periodic, our analysis accounts only for the case where each task accesses one (nested) critical section during its period. However, it is straightforward to extend the analysis to the case where each task can enter the critical sections a number of times. This can be done by analyzing the blocking time of each user independently by setting its period to the period of the task that the user is from (i.e. an outermost critical section), as it is done in this section. The blocking time of a task is (1) the sum of the blocking times of all the users of the task and (2) the time period of blocks of the task that occur in the “host” processor of the task where the non-critical section part of its code is executed. The blocking of the task that occur in the “host” processor are described in [5].

5 Performance Comparison

In this section, we show that the upper bound on the blocking time in our protocol subsumes that in the MPCP with only a marginal difference. This is done by showing that (1) when given the same allocation of critical sections as the MPCP, our protocol provides approximately the same upper bound on the blocking time, and (2) there is at least one allocation with which our protocol gives a smaller upper bound. The comparison with Sun et al’s protocol is similar.

We first show that our protocol provides a similar upper bound on the blocking time when given the same allocation of critical sections as the MPCP.

Let \( B_{CSi} \) be the worst case blocking time of a user \( i \) in the MPCP.

\[
B_{CSi} \leq \sum_{j \in \eta_i} \left[ \frac{T_j}{T_i} \right] (CS_j + L_j) + L_i + CS_i
\]  

(1)

where \( \eta_i \) is the set of users that have a higher priority than \( i \) and are allocated to the same (synchronization) processor (note that it contains the set of users that have a higher priority than \( i \) and are directly or indirectly in conflict with \( i \)), and \( L_{CSi} \) is the time period that \( i \) is blocked by lower priority users. \( L_{CSi} \) is bounded by the maximum duration of one critical section of lower priority users that require a resource either that is required by \( i \), or whose priority ceiling is higher than the priority of \( i \).

Note that in the MPCP, all users are completely local. Thus, assuming that \( B_i \leq T_i \), we can rewrite Theorem 4.3 as follows.

\[
B_i \leq \sum_{j \in \pi_i \cup \eta_i} \left[ \frac{T_i}{T_j} \right] (CS_j + L_i + CS_i)
\]  

(2)

Note that \( \pi_i \cup \eta_i = \eta_i^G \).

The two bounds in Eq. (1) and (2) differ only in the blocking times caused by lower priority users, which are \( L_i \) and \( L_{CSi} \). \( L_i - L_{CSi} \) can be as large as the difference between the largest critical section time and the smallest critical section time of the lower priority users. This is because our protocol uses resource preclaiming, so each user acquires all the required resources before it enters the critical region although it may not use some of them until the end of the critical region. However, a similar situation may occur in the MPCP because of priority ceiling. Suppose that the outermost critical section in a nested critical section requires a resource that has the highest priority ceiling in the system. Then as long as a task \( i \) is in the outermost critical section, no matter how low the priority of \( i \) is, no other tasks can use even the resources that \( i \) acquires at the end of its critical section. As stated in [5], critical sections are usually very small. Thus, we believe that the difference is negligible.

As many different allocations of users to processors are considered, our protocol provides a possibility that the blocking time of a user (or task) can be less than that in the MPCP. In the following, we show one simple case when our protocol can give a better performance than the MPCP.

Given that one user is allocated to one processor, \( B_i \) can be bounded as below, ignoring the message delays. Note that \( \eta_i, \forall j \in \pi_i \), is empty in this case.

\[
B_i \leq \sum_{j \in \pi_i} \left[ \frac{T_i}{T_j} \right] (CS_j + L_j) + L_i + CS_i
\]  

(3)

To simplify the comparison, let us assume that the critical section times of all users are all same (\( CS_i = CS_j = CS \)). Then,

\[
B_i \leq \sum_{j \in \pi_i \cup \eta_i} \left[ \frac{T_i}{T_j} \right] 2CS
\]  

(4)

The upper bound on \( B_i \) is always less than that on \( B_{CSi} \) if the following condition is true:

\[
\sum_{j \in \pi_i \cup \eta_i} 2\left[ \frac{T_i}{T_j} \right] < \sum_{j \in \eta_i^G} \left[ \frac{T_i}{T_j} \right]
\]  

(5)

It should be noted that \( |\pi_i| \leq |\eta_i^G| \), and the difference between \( |\pi_i| \) and \( |\eta_i^G| \) can be as large as the total number of users in the system. Considering the worst case, suppose that there are a set of users \( u_1, u_2, u_3, \ldots, u_n \) (assume \( n \) is even), and \( u_i \) is directly in conflict with \( u_{i+1} \). Furthermore, suppose that if \( i \) is
odd, $u_i$ has a priority $[i/2]$, otherwise $u_i$ has a priority $(n/2 + [i/2])$. We can establish the following relation: for an even integer $i$, $u_i < u_{i-1}$ and $u_i < u_{i+1}$. There is no relation among $u_1, u_3, u_5, \ldots$, or among $u_2, u_4, u_6, \ldots$. Recall that $y_i$ contains all the users that are directly or indirectly in conflict with $i$ and have a higher priority than $i$. Thus, $|y_i|$ is at most 2 while $|y_i^a|$ is equal to $(|y_i| - 1) - 1$ if $i$ is odd, and equal to $(n/2 + |y_i|)$ if $i$ is even. $|y_i| - |y_i^a|$ is as large as the total number of users in the system.

Typical cases in which our protocol can provide a better bound than the MPCP constitute a tree structured directed conflict graph where a node represents a user and an directed edge from a node $i$ to a node $j$ represents $i < j$. Figure 7 shows one example, which consists of three subtrees $(i, j$, and $k)$. By allocating users in each subtree to different processors, the execution of users in one subtree doesn’t interfere with that of users in the other subtrees. However, in the MPCP, all the users in the entire tree have to be allocated to the same synchronization processor, which increases the interference among users and thus, the blocking time.

6 Conclusion

The focus of this paper is a distributed protocol that provides scalable synchronization in distributed or parallel systems where tasks communicate through messages. The protocol allows flexible allocation of critical sections to a number of synchronization processors, resulting in possible reduction of the blocking time. This is unlike the protocol of Rajkumar et al that forces all the critical sections that are indirectly or directly in conflict to be executed in the same synchronization processor. This is important because the advent of large-scale multiprocessor systems provides the potential to schedule hard real-time tasks that may not be schedulable in a small-scale system.

We showed that our result subsumes that of [5]. When given an appropriate allocation of critical sections and task sets, the protocol gives a smaller upper bound on the blocking time than the protocol of Rajkumar et al. Furthermore, given the same allocation as that of Rajkumar et al, the protocol provides approximately the same bound. It is not the point of this paper that our protocol always gives better bounds, but that it provides the choice to minimize the blocking time by evaluating the tradeoffs of various allocations of critical sections. Because of its flexibility and scalability, the protocol has application in global optimization allocation techniques such as the simulated annealing based technique [9] where many different allocations are evaluated to optimize the blocking time of each task in the system.

Acknowledgement: We would like to thank Jennifer L. Welch of Texas A&M University for assistance with simplifying the system model, and Mathai Joseph of the University of Warwick and Yang-Min Soo of Seoul N. University for their helpful comments on the earlier version of the paper.

References