Resource access control for dynamic priority distributed real-time systems

Article *in* Real-Time Systems · October 2006

DOI: 10.1007/s11241-006-8642-5 · Source: DBLP

Chen Zhang
Bryant University

Resource Access Control for Dynamic Priority Distributed Real-Time Systems

Chen Zhang
Dept. of Computer Information Systems
Bryant College
Smithfield, RI 02917-1284

David Cordes
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487-0290

Abstract

Many of today's complex computer applications are being modeled and constructed using the principles inherent to real-time distributed object systems. In response to this demand, the Object Management Group's (OMG) Real-Time Special Interest Group (RT SIG) has worked to extend the Common Object Request Broker Architecture (CORBA) standard to include real-time specifications. This group's most recent efforts focus on the requirements of dynamic distributed real-time systems. One open problem in this area is resource access synchronization for tasks employing dynamic priority scheduling.

This paper presents two resource synchronization protocols that meet the requirements of dynamic distributed real-time systems as specified by Dynamic Scheduling Real-Time CORBA (DSRT CORBA). The proposed protocols can be applied to both Earliest Deadline First (EDF) and Least Laxity First (LLF) dynamic scheduling algorithms, allow distributed nested critical sections, and avoid unnecessary runtime overhead. These protocols are based on (i) distributed resource preclaiming that allocates resources in the message-based distributed system for deadlock prevention, (ii) distributed priority inheritance that bounds local and remote priority inversion, and (iii) distributed preemption ceilings that delimit the priority inversion time further.

1. Introduction

Real-time middleware has experienced significant growth in recent years. This work has led to the establishment of numerous standards for RT middleware. Building on the industry's most widely accepted middleware standard, OMG formed the Real-Time Special Interest Group (RT SIG) with the goal of extending the CORBA (Common Object Request Broker Architecture) standard with real-time specifications. The original RT CORBA 1.0 specifications (Chapter 24 in [1]) were restricted to *static* real-time systems, that is, systems where the workload is known *a priori*. The most recent effort of OMG's RT SIG attempts to meet the requirements of *dynamic* distributed real-time systems. This effort is captured in the Dynamic Scheduling Real-Time CORBA (DSRT CORBA) specifications [2]. In a dynamic distributed real-time system, both the workload on the system and the time constraints realized by activities are subject to change. In contrast with the offline scheduler used by static real-time systems, *online schedulers* are required for dynamic distributed systems to provide the needed flexibility and scalability. Such an online scheduler comes with a price, however, introducing overhead into the system due to the online scheduler's computation and dispatching.

Although the DSRT CORBA specifications [2] discuss resolving any shared resource dependency between tasks via mutexes, existing research has not focused on shared resource synchronization in a dynamic priority, distributed real-time system, such as one specified by the DSRT CORBA. Furthermore, the application of different dynamic scheduling algorithms such as Earliest Deadline First (EDF) and Least Laxity First (LLF) has also not been addressed. This paper addresses these issues, providing effective resource synchronization protocols for such situations. It then applies them to the dynamic scheduling policies, and offers an analytic proof of the properties of the synchronization protocols in the DSRT CORBA system.

2. Background and Related Work

Work related to this paper includes work in both real-time scheduling theory and resource synchronization protocols in real-time systems. Before presenting these protocols, we first provide a brief background in both of these areas.

2.1 Real-time scheduling theory

A number of fixed and dynamic priority driven scheduling algorithms for real-time systems have been developed over the past three decades [6]. These algorithms have been applied to real-time systems including Real-Time Operating Systems (RTOS), Real-Time Database Management System (RTDBMS), and most recently to distributed object-oriented middleware environments such as the OMG RT CORBA. Since [8], a milestone in the field of hard real-time scheduling, fixed priority scheduling algorithms and dynamic priority scheduling algorithms have been studied separately.

Fixed priority scheduling:

For periodic tasks, the relative deadline of a task and the period of the tasks are static time constraints (note that the absolute deadline of a given task is equal to the invocation time plus the relative deadline). If the system workload is also fixed, then the scheduler can assign a set of static priorities to the task set using either the relative deadlines or the periods. No runtime modifications to priorities are necessary.

The original work in [8] established the Rate-Monotonic (RM) scheduling algorithm. This work was later extended to the Deadline-Monotonic (DM) scheduling algorithm [7]. In both algorithms, the static priorities assigned to tasks are inversely proportional to the task periods and the task relative deadlines respectively.

Consider the problem of scheduling a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of periodic tasks on a single processor. Each task τ_i can be represented by the tuple $\{e_i, D_i, T_i\}$, where e_i, D_i and T_i respectively represent its worst-case computation time, relative deadline and period. Let p_i represent the priority assigned to task τ_i. For RM scheduling, $p_i \propto \frac{1}{T_i}$ and for DM scheduling, $p_i \propto \frac{1}{D_i}$. The scheduler chooses the most eligible pending task and dispatches it for CPU execution. The scheduler assumes that:

1. A Highest Priority First (HPF) policy is utilized.
2. The tasks are preemptive. That is, the processing of any task can be interrupted by a higher priority task if the two tasks are not dependent on each other.
3. The CPU is non-idling in the presence of a pending task. That is, if a task is blocked or completed, the next most eligible pending task is executed.

The original OMG RT CORBA 1.0 specifications [1] specified a static offline *Scheduling Service*. This *Scheduling Service* is designed to work in a "closed" CORBA system where fixed priorities are utilized with a static set of tasks. Fixed priority scheduling algorithms, such as RM and DM, can be implemented using this *Scheduling Service*.

Dynamic Priority Scheduling

Dynamic priority scheduling assumes the same execution policies of HPF, preemptive and non-idling. The difference with a dynamic algorithm is the priority assignment method. Dynamic priority scheduling algorithms assign priorities to tasks according to some dynamic optimal criteria. With this constraint, an online scheduler is required for reassigning the most eligible task according to the updated priorities. The online scheduler is activated by the system at specific scheduling points. These scheduling points include:

- Creation of a task
- Termination (abortion or completion) of a task
- Creation of a shared resource
- Blocking on a request for an exclusive shared resource
- Unblocking as a result of the release of an exclusive shared resource
- Other special events such as CORBA method invocation and response.

Whenever a scheduling point occurs in the system, the most eligible pending task will be reselected and dispatched to the underneath operating system threads. The two most popular dynamic priority scheduling algorithms utilized today are Earliest Deadline First (EDF) and Least Laxity First (LLF).

The Earliest Deadline First (EDF) dynamic scheduling algorithm was originally presented in [8]. In this algorithm, the processor always executes the task with the earliest absolute deadline. EDF scheduling algorithm has been proven to be optimal [8] in the sense that no other dynamic, or fixed, priority driven scheduling algorithm can lead to a valid schedule that cannot be obtained by EDF.

The Least Laxity First (LLF) scheduling algorithm introduced by [5] will, at any scheduling point, choose the task with the smallest laxity. Laxity is defined as the task's absolute deadline minus the current time minus estimated remaining execution time. LLF has been proven to be optimal [9] in the same sense as EDF.

When comparing fixed and dynamic priority scheduling algorithms, it should be noted that both have benefits and drawbacks. The advantages of fixed priority scheduling algorithms include easier implementation and lower run-time overhead. However, the system task set for fixed priority scheduling cannot be modified at runtime. In addition, when task periods are not harmonic (a harmonic task set is one with a base frequency of which the frequencies of all the tasks are multiples), the maximum schedulable percentage of the CPU is $n(2^{1/n}-1)$. For very large *n*, this schedulable bound is slightly larger than 69% [8].

Dynamic scheduling algorithms schedule periodic tasks and aperiodic tasks in a similar manner. A task set operating in a system that utilizes dynamic scheduling algorithms can be updated without restarting the whole system. In contrast to fixed priority scheduling algorithms, the schedulable bound for EDF and LLF is always 100%. However, computations associated with the online scheduler introduce overhead, at the price of the flexibility of dynamic priority scheduling.

2.2 Resource Synchronization Protocols

In a real-time system, exclusive access to a resource is typically achieved by semaphore-like operations. The usage of semaphores in real-time system must be carefully monitored by the proper resource synchronization protocol. Besides ensuring mutually exclusive execution within critical sections and preventing deadlocks, resource synchronization protocols in a real-time system must strictly bind the priority inversion. Priority inversion occurs when a higher priority task must wait for a lower priority task to release a needed resource, such as the CPU or other exclusive shared resources, including both logical (e.g. semaphore) and physical (e.g. printer or communication channel) resources. A higher priority task is said to be “blocked” when priority inversion occurs. Unbounded priority inversion causes unpredictable timing behavior in real time systems.

To address this potential problem, Sha and Rajkumar [12] adopted the priority inheritance protocol and developed the Priority Ceiling Protocol (PCP). The original PCP protocol applies to a single processor fixed priority real-time system, such as the Rate Monotonic (RM) or Deadline Monotonic (DM) scheduling system. The priority ceiling mechanism was later adopted into other resource synchronization protocols that apply to more general real-time systems. Below we give a brief review of the Priority Ceiling family of resource synchronization protocols, including:

- Priority Ceiling Protocol (PCP)
- Dynamic Priority Ceiling Protocol (DPCP)
- Stack Based Protocol (SBP)
- Distributed Priority Ceiling Protocol (DistPCP)
- Distributed Resource Preclaiming (DRP)

Priority Ceiling Protocol (PCP):

The two basic concepts in PCP are priority inheritance and priority ceiling [12]. The basic idea of priority inheritance is that when a task τ blocks one or more higher priority tasks, it ignores its original priority assignment and executes its critical section at the highest priority level of all the tasks it blocks. After executing its critical section, τ returns to its original priority level.

The *priority ceiling* of a semaphore is defined as the priority of the highest priority task that may lock this semaphore. Let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by tasks other than task τ. Before task τ enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data. If the priority of task τ is not higher than the priority ceiling of semaphore S^*, task τ will be blocked. In this case, if the task τ^* which holds the lock on S^* has lower priority than τ, τ^* will inherit priority of τ, $p(\tau)$, and keep it until τ^* releases S^*. The idea behind PCP is integrating priority inheritance to bound priority inversion and total ordering of resource access according to the priorities of possible resource accesses to prevent deadlock.

PCP is designed for uniprocessor systems, prevents deadlocks, and ensures that the blocking time that each instance of a task can experience is no more than the duration of one critical section of a lower-priority task. Therefore the blocking times of tasks are bounded and moreover, can be counted into the schedulability of the RM and DM scheduling easily. A sufficient schedulable condition of a system of n periodic tasks under the RM scheduling algorithm and PCP resource synchronization protocol is:

$$\forall i, 1 \le i \le n, \frac{E_1}{T_1} + \ldots + \frac{E_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1)\,. \tag{1}$$

Where E_i, T_i and B_i are the execution time, period and worst-case blocking time of task τ_i.

Dynamic Priority Ceiling Protocol (DPCP):

Chen and Lin [4] extended PCP, constructing a Dynamic Priority Ceiling protocol (DPCP), for EDF dynamic priority scheduling algorithms. The authors use the absolute deadline of a task directly as the task's priority. A smaller value represents a higher priority. When one instance of the periodic task completes, the task priority is updated to the absolute deadline of the next instance, which is greater in value and represents a lower priority.

The algorithm definition of DPCP is identical to the original PCP definition except for the dynamic priority ceiling. The priority ceiling $c_t(S)$ is defined as the priority of the highest priority task that may access semaphore S at time t. So when any task instance completes, the priority ceiling value needs to be recalculated.

DPCP prevents deadlocks. Under DPCP, the blocking time that each instance of a task can experience is no more than the duration of one critical section of a lower-priority task. This property is identical to the original PCP. A sufficient schedulable condition of a system of n periodic tasks under the EDF scheduling algorithm and DPCP resource synchronization protocol is:

$$\frac{E_1 + B_1}{T_1} + \ldots + \frac{E_i + B_n}{T_n} \le 1. \tag{2}$$

Where E_i, T_i and B_i are the execution time, period and worst-case blocking time of task τ_i.

Stack Based Protocol (SBP):

The Stack Based Protocol (SBP) given by Baker [3] is applicable to both fixed priority and dynamic priority scheduling. SBP allocates task runtime stack space in association with real-time scheduling. Tasks under fixed or dynamic priority scheduling are allocated the stack space according to the following rules:

1. Every task using the stack requires an initial allocation of at least one cell of stack space before it can start execution, and cannot relinquish that space until it completes execution. This means that the entire execution of each task is a "critical section" with respect to the stack.
2. A stack storage request can be granted to a task if and only if the task is not yet holding any stack space or it is holding the top of the stack.
3. Only the task at the top of the stack may execute, since an executing task may need to increase its stack size at any time.

SBP handles dynamic task priority resource synchronization using static preemption levels and preemption ceilings. For EDF dynamic priority scheduling, the preemption level $\pi(\tau)$ of a task τ is inversely proportional to the relative deadline of the task. The preemption ceiling of a resource *R*, when $v_{\boldsymbol{R}}$ units is available, is:

$$\lceil R \rceil_{v_R} = \max(\{0\} \cup \{\pi(\tau) \mid v_R < \mu_R(\tau)\}) \tag{3}$$

Where $\mu_R(\tau)$ is the maximum units of *R* that task τ may request. SBP defines the current preemption ceiling of the system, $\overline{\pi}$, to be the maximum of the preemption level of the current task and current ceilings of all the resources. That is, if task τ_c is currently executing, $\overline{\pi} = \max(\{0, \pi(\tau_c)\} \cup \{\lceil R_i \rceil \mid i = 1, \ldots, n\})$. SBP requires that a task execution request be blocked from starting execution (i.e. from receiving its initial stack allocation) until $\overline{\pi} < \pi(\tau)$. Once a task τ has started execution, all subsequent resources requested by τ are granted immediately, i.e., without blocking.

SBP is designed for uniprocessor systems and ensures deadlock prevention. In SBP, no task can be blocked after it starts. If the oldest highest-priority task is blocked, it will become unblocked no later than the first instant that the currently executing job is not holding any nonpreemptable resources. This means that no job will be blocked for longer than the duration of one outmost critical section of a lower priority task. In an EDF dynamic priority scheduling system, if we assume the task set $\{\tau_1, \ldots, \tau_n\}$ is sorted in decreasing preemption level order. Then a sufficient schedulable condition of SBP is:

$$\underset{k=1,\ldots,n}{\forall k} \left(\sum_{i=1}^{k} \frac{E_i}{D_i} \right) + \frac{B_k}{D_k} \leq 1 \tag{4}$$

Where E_i, D_i and B_i are the execution time, relative deadline and worst-case blocking time of task τ_i.

Distributed Priority Ceiling Protocol (DistPCP):

In a distributed system, Rajkumar [10] considered the case where tasks executing locally need to access resources on remote processors and developed the Distributed Priority Ceiling Protocol (DistPCP). In DistPCP, the critical section (CS) that executes on remote processors is called a Global Critical Section (GCS). A processor hosts GCSs is called a synchronization processor. A task τ_i in its GCS is guarded by a global semaphore with the priority of p_i^{GCS}. This priority of GCS is higher than all local tasks and therefore will preempt all local tasks on the synchronization processor. After all GCSs have been assigned proper priority, synchronization processors in the distributed system can apply PCP locally, treating GCSs no differently than local CSs.

DistPCP applies only to fixed priority scheduling and assumes no nested GCSs. It ensures deadlock prevention and bounded priority inversion for both local CS and GCS. In DistPCP, a set of n periodic tasks can be scheduled using the RM scheduling algorithm if the following condition is satisfied for each processor of the distributed system:

$$\forall i, 1 \le i \le n, \frac{E_1}{T_1} + \ldots + \frac{E_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1) \qquad (5)$$

Where E_i, T_i and B_i are the execution time, period and worst-case blocking time of task τ_i.

In DistPCP, the worst case blocking time B_i is the sum of the four items shown in the table below. That is, the blocking time B_i of task τ_i is: $B_i = LB_i + GB_i + RW_i + SSB_i$

Local Blocking Time (*LB*$_i$) Duration of n_i^G+1 local critical sections that can block τ_i given by $LB_i = B(\tau_i) \times (n_i^G + 1)$. Here *B*($\tau_i$), the local blocking time of τ_i when τ_i suspends itself for GCS execution, is the duration of the longest critical section of a lower priority job that may block τ_i.	**Remote Waiting Time (*RW*$_i$)** Duration of global critical sections for higher priority jobs bound to the synchronization processors that would also be accessed by τ_i. $RW_i = \sum_{P_j \in GP_i} \sum_{\tau_m \in RH_i} \left(\left\lceil \frac{T_i}{T_m} \right\rceil \times GCS_m(P_j) \right)$. Here GP_i is the set of synchronization processors accessed by τ_i. RH_i is the remote higher priority task set of τ_i. T_i is the period of τ_i. T_m is the period of the remote higher priority task τ_m. $GCS_m(P_j)$ denotes the sum of durations of all the GCS of τ_m that executes on synchronization processor P_j.
Global Blocking Time (*GB*$_i$) A maximum duration of n_i^G global critical sections of lower priority jobs on remote synchronization processors accessed by τ_i. $GB_i = \sum_{j=1}^{n_i^G} B(GCS_i)$	**Synchronization Server Blocking Time (*SSB*$_i$)** Duration of GCSs of lower priority tasks bound to common processor *P* with τ_i. $SSB_i = \sum_{\tau_j \in RL_i} \left\lceil \frac{T_i}{T_j} \right\rceil \times GCS_j(P)$ Where RL_i is the remote lower priority of task τ_i and $GCS_j(P)$ denotes the sum of durations of all the GCSs of τ_j that execute on *P*.

Table 1. Blocking Time of Distributed Priority Inheritance Protocol

Distributed Resource Preclaiming (DRP):

Rhee and Martin [11] combined the concept of distributed resource preclaiming (DRP) with the PCP protocol. The new protocol prevents deadlock while eliminating the limitation in DistPCP regarding nested GCSs. DRP is a pessimistic resource allocation technique that requires each task, before entering a critical section, to acquire all the resources needed within that critical section. In DRP, the control of resource allocation is completely distributed and each resource has a Resource Manager. In order to bound priority inversion, DRP adopts the convention of PCP as condition for entering the critical region. Although not mentioned in the paper, DRP satisfies the schedulable condition of the fixed priority scheduling theory and therefore follows the same schedulability condition given in (5). The blocking time is further bounded by the expression of

$$B_i \le \sum_{j\in HP_i} \left\lceil \frac{B_i}{T_j} \right\rceil \left(B_j + H_j + \hat{L}_i + 2d\right) + L_i \tag{6}$$

Where d is the maximum message delay. In this computation, HP_i is the set of conflicting users with a higher priority than user i, T_j is the period of task τ_j, and H_j is the time that user j is critical. The user with the highest priority in its conflicting user set is said to be *eminent*. L_j is the maximum time period from a time t when j becomes eminent to when j enters the critical region if j keeps eminent since time t. $\hat{L}_j$ is the maximum time period that j is blocked by lower priority users in P_j that do not conflict with j but have a priority ceiling higher than priority of j, p_j. The bounds of H_j, L_j and $\hat{L}_j$ are given below.

$H_j \le CS_j + \sum_{k\in LHP(C_j)} \left\lceil \frac{H_j}{T_k} \right\rceil CS_k$, where *LHP(C_j)* is the set of all local CSs in processor P_j whose priorities are higher than priority ceiling of P_j.
If define $S(j,H_j) = CS_j + \sum_{k\in LHP(C_j)} \left\lceil \frac{H_j}{T_k} \right\rceil CS_k$, the bound of L_j is given by $L_j \le \max_{k\in LP_j} \{S(k,L_j)\} + \max_{l\in LP_j \wedge p(j)<C(l)} \{S(l,L_j)\} + 4d$, where LP_j is the set of all conflicting CSs of τ_j that have lower priority than τ_j.
The bound of $\hat{L}_j$ is given by $\hat{L}_j \le \max_{l\in LP_j \wedge p(j)<C(l)} \{S(l,L_j)\}$.

Table 2. Blocking Time of Distributed Preclaiming Protocol

Although DRP was originally designed for fixed priority scheduling algorithms, extending DRP to include the dynamic priority ceilings and dynamic priority ceilings update enables the synchronization of dynamic priority scheduling algorithms such as EDF and LLF. However, dynamic priority ceilings update causes high priority ceiling updates overhead. This overhead is even more significant given the network message based distributed system environment in which the dynamic priority ceilings update must involve multiple steps of network message multicasts. Detailed calculation of this overhead is provided in 4.3.

3. System Model Framework

Before addressing the specifics of the protocols that will be developed, we first need to establish the basic operational framework for the system in which our protocols apply. This framework includes a definition of the task model utilized, the resource manager model employed, and the communication framework.

3.1. Task model

We assume that the system consists of a set $\{\tau_i\}_t$ of periodic tasks at time *t*. Each task τ_i has a property tuple $\{T_i, E_i, D_i\}$ that represents the period, worst-case execution time and relative deadline of task τ_i. By default, we imply the current set of tasks when mentioning $\{\tau_i\}$. Although we assume that tasks are periodic, the workload of the system is dynamic. Specifically, existing tasks can be terminated and new tasks can be loaded into the system during run time. In other words, the system is a scalable, real-time system in which runtime workload changes can be incorporated without forcing a system restart. This assumption is convenient: a system operating under this assumption is mutable between a pure static real-time system model in which the workload and all task time-constraints are known *a priori* and a pure dynamic real-time system model in which no *a priori* knowledge of the system is available.

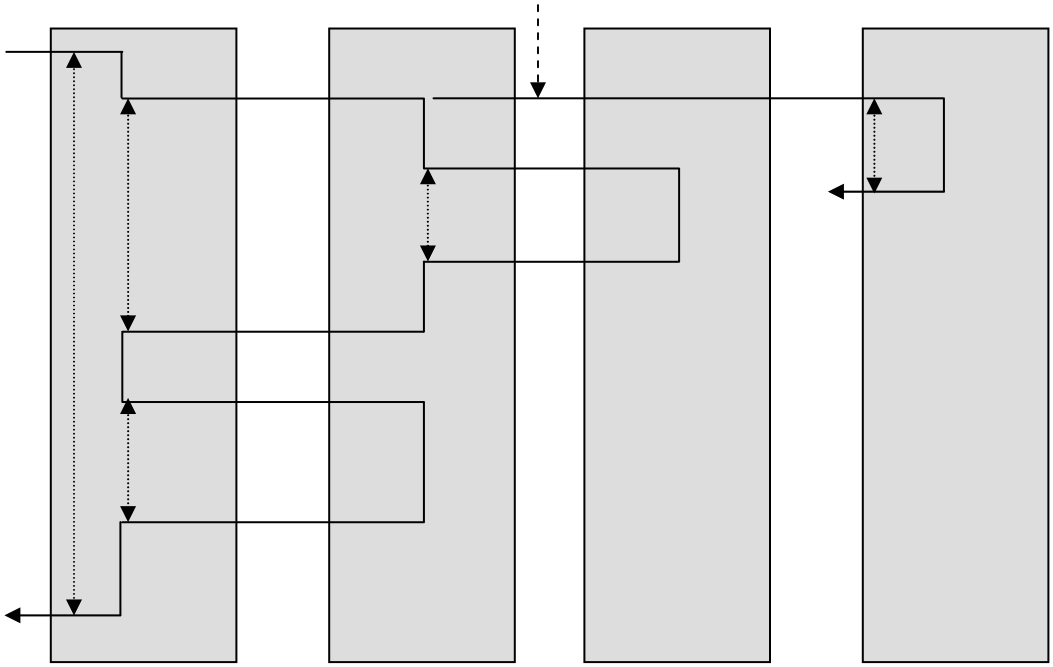


Figure 1. Segmentation of Distributable Threads

We use the Distributable Thread (DT) model defined in DSRT CORBA to represent a real-time task instance. In this model, a periodic task consists of an infinite set (from task start time to task termination time) of periodically released DTs. As highlighted in Figure N, a one-way CORBA invocation generates a separate DT. Each DT consists of a series of schedulable entities called *scheduling segments* (which we will simply refer to as *segments*). DSRT CORBA allows users to define segments by invocations of begin_scheduling_segment and end_scheduling_segment, we require a stricter, automated segmentation rule for all segments with exclusive resource requests.

For automated segmentation, we define the outermost segment to begin with the invocation of a DT and to end at the termination of the DT. A separate segment is defined upon the invocation of a method in another object, ending with the associated return (and including any nested segments). Figure N contains a diagram that shows the segmentation of the DTs.

A segment with an exclusive resource request is called a Critical Section (CS) segment. All CSs must be properly nested. For any CS, let $CS_{i,j}$ denote the *j*th CS of task τ_i. That is, given any pair of critical sections $CS_{i,j}$ and $CS_{i,k}$, then either $CS_{i,j} \subset CS_{i,k}$, $CS_{i,j} \supset CS_{i,k}$ or $CS_{i,j} \cap CS_{i,k} = \Phi$.

3.2. Resource Manager

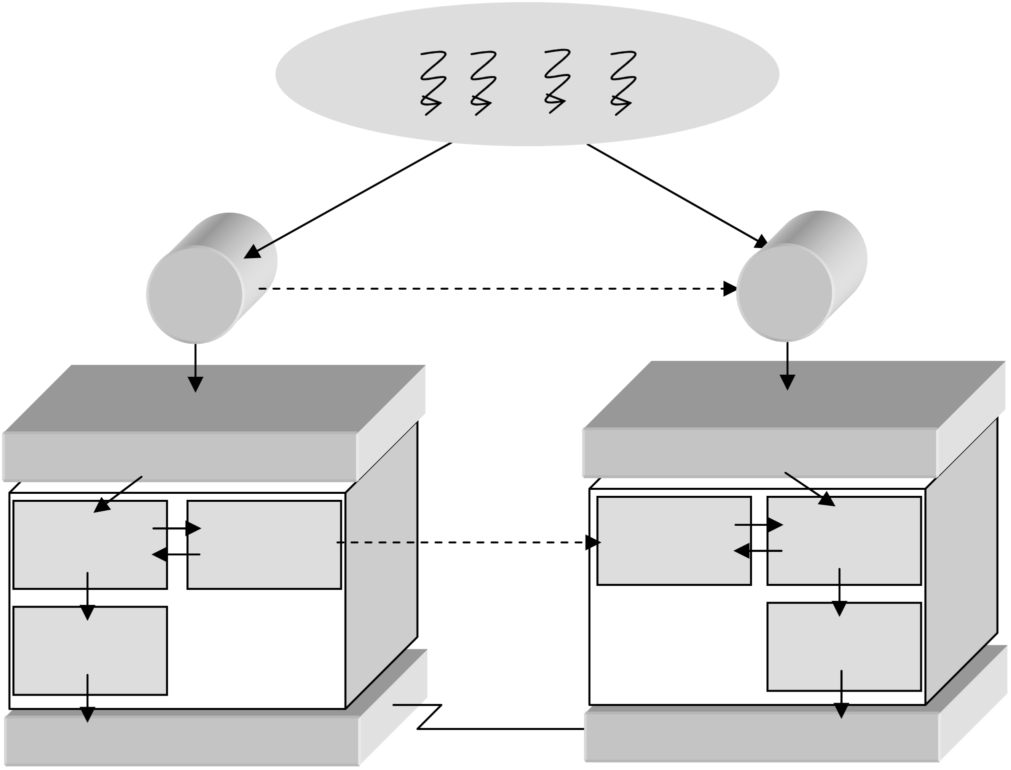


Figure 2. System Reference Model

Local schedulers apply dynamic priority scheduling algorithms such as EDF or LLF. Local resource managers are responsible for synchronizing local resources access and applying resource synchronization protocols. The coordination of local schedulers and local resource managers permits the scheduling of tasks with exclusive resource requests. Communication between resource managers is necessary for distributed nested resource requests.

3.3. Communication model

Our model assumes that all communications that occur between the segment, scheduler, and resource manager are realized via CORBA method invocations. We assume a FIFO network communication model with a maximum communication delay of δ. That is, the first message sent to a receiver is the first message received and the maximum time interval between the respective *send*() and *receive*() operations is δ.

4. Synchronization Protocol Components

In this paper we present two synchronization protocols for dynamic distributed real-time systems: the Distributed Preclaiming Priority Inheritance Protocol (DPPIP) and the Distributed Preclaiming Preemption Ceiling Protocol (DPPmCP). Before formally presenting these two protocols, we develop a conceptual description of the basic ideas around which these synchronization protocols are constructed. The three fundamental concepts that require a more detailed explanation include:

- Distributed resource preclaiming
- Distributed priority inheritance, and
- Preemption ceilings.

4.1 Distributed Resource Preclaiming

A fundamental difficulty of synchronization in a distributed system is the handling of distributed nested critical sections. That is, the system must be capable of processing a critical section that holds a local and a remote resource concurrently. A distributed nested critical section can cause distributed deadlock and unbounded priority inversion.

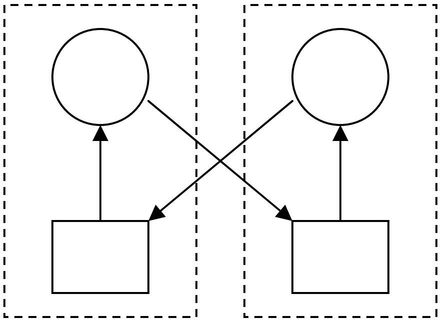


Figure 3. Distributed Deadlock

Time	Action
t = 0	τ_1 sends requests for R_1 and R_2. τ_1 is granted R_1. *ceiling*(P_1) is set to $p(\tau_0)$.
t = δ/2	T_2 sends requests for R_1 and R_2. *ceiling*(P_2) := $p(\tau_0)$.
t = δ	R_2 receives request from τ_1 and denies it because $p(\tau_1) < ceiling(P_2) = p(\tau_0)$.
t = 3δ/2:	R_1 receives request from τ_2 and denies it because $p(\tau_2) < ceiling(P_1) = p(\tau_0)$.

Applying an original PCP into a distributed system can create a distributed deadlock. To illustrate this, consider Figure N where we assume that resource R_1, R_2 are requested by three tasks τ_0, τ_1 and τ_2 with priorities $p(\tau_0) > p(\tau_1) > p(\tau_2)$. Therefore, according to the single processor PCP, *ceiling*(R_1) = *ceiling*(R_2) = $p(\tau_0)$. We assume that the network message delay is δ, R_1 and R_2 are free at t=0, and ignore local message delay. A distributed deadlock can occur as example in Figure 3:

At $t = 3\delta/2$, the three conditions for deadlock are all satisfied: non-preemptable, hold-and-wait, and cyclic waiting. A distributed deadlock is formed.

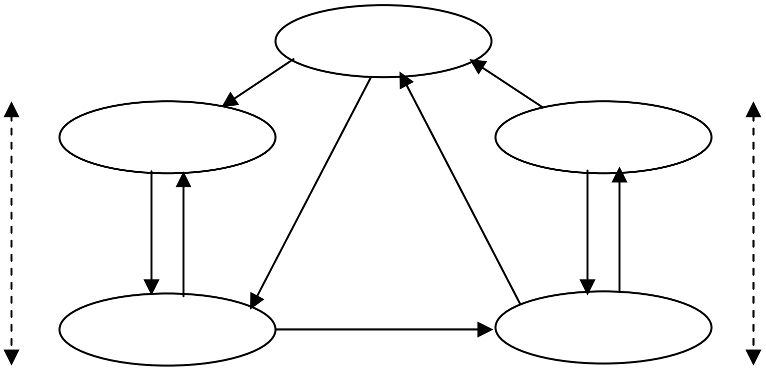
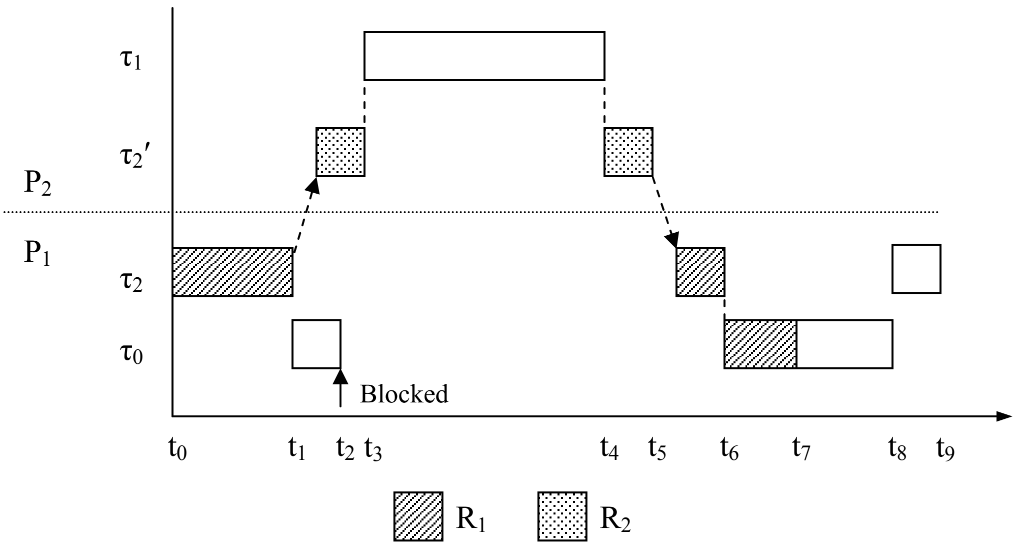


Figure 4. Segment Lifecycle under DRP

To deal with this situation, we have adopted the resource preclaiming protocol by Rhee and Martin [11]. Rhee and Martin showed that it is possible to eliminate various deadlock conditions from the resource allocation stage and the scheduling stage. They applied Distributed Resource Preclaiming (DRP) in the resource allocation stage and used this to release the non-preemptable condition from this stage (Figure 4). After all local and remote resources are allocated to a critical section through distributed resource requests and competition, the critical section enters the scheduling stage holding all resources requested. As a result, they have eliminated hold-and-wait from this stage and therefore prevented deadlock.

4.2 Distributed Priority Inheritance

Unbounded distributed priority inversion is a problem introduced by allowing jobs to lock resources on different processors simultaneously.



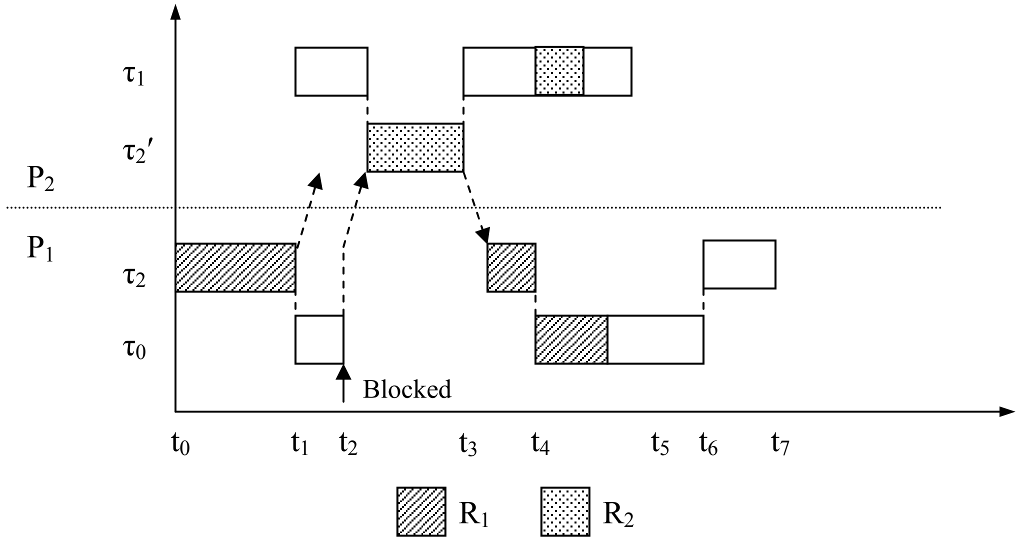
Time	Action
t_0	τ_2 request for R_1, τ_2 is granted R_1, *ceiling*(P_1) := *p*(τ_0), τ_2 starts execution in P_1.
t_1	τ_2 invokes τ_2' in P_2. τ_0 is released and starts execution in P_1.
t_1+ δ	τ_2' request for R_2, τ_2' is granted R_2, *ceiling*(P_2):=*p*(τ_2), τ_2' starts execution in P_2.
t_2	τ_0 requests for R_1, τ_0 is blocked, τ_2 inherits *p*(τ_0).
t_3	τ_2' is preempted by τ_1 in P_2.
t_4	τ_1 completes, τ_2' resumes execution.
t_5	τ_2' completes, ceiling(P_2) := 0, replies to τ_2.
t_5+ δ	τ_2 executes the remainder critical section.
t_6	τ_2 completes, ceiling(P_1) := 0, τ_2 is preempted by τ_0.
t_7	τ_0 completes critical section, still eminent, continues remainder execution.
t_8	τ_0 completes, τ_2 resumes execution of the remainder.
t_9	τ_2 completes.

Figure 5. Unbounded Priority Inversion

We assume the same task set as our previous example, with priorities *p*(τ_0)> *p*(τ_1)> *p*(τ_2) =*p*(τ_2'). R_1 in P_1 is requested by τ_0 and τ_2. R_2 in P_2 is requested by τ_2' and τ_1. We also assume that both R_1 and R_2 are both free at t_0.

In this situation, highest priority task τ_0 is blocked waiting for lower priority remote task τ_1. Since the number of remote tasks similar to τ_1 with intermediate priority is indefinite, the blocking time for τ_0 can be unbounded. Although the local blocking task τ_2 inherited the priority of τ_0, its remote nested critical segment was already running with the original priority. We extend the original priority inheritance protocol to prevent unbounded distributed priority inversion.

We now consider a second situation where distributed priority inheritance is applied. This example, shown in Figure 6, assumes the same initial condition as our last example. The events and time are listed:



Time	Action
t_0	τ_2 requests for R_1, τ_2 is granted R_1, τ_2 starts execution.
t_1	τ_1 is released and starts execution, τ_2' is invoked, τ_2' requests for R_2 and is granted, τ_2' waits in ready status. τ_0 is released and starts execution in P_1.
t_2	τ_0 requests R_1 and is blocked. τ_2 inherits $p(\tau_0)$ and sends inherit message to τ_2'.
$t_2 + \delta$	τ_2' inherits $p(\tau_0)$, preempts τ_1 and starts execution.
t_3	τ_2' ends execution and replies to τ_2. τ_1 resumes execution.
$t_3 + \delta$	τ_2 resumes execution.
t_4	τ_2 completes critical section, τ_0 is granted R_1 and preempts τ_2 for execution. τ_1 requests R_2 and is granted.
t_5	τ_1 completes.
t_6	τ_0 completes. τ_2 resumes execution of the remainder.
t_7	τ_2 completes.

Figure 6. Distributed Priority Inheritance

From this, it is apparent that one must address the possibility of multiple preemption and distributed priority inheritance iterations during one CS execution. This situation occurs when the events between t_1 and t_2 repeat with higher priority tasks replacing τ_1 and τ_0. Given the low occurrence of distributed nested resource access in common task sets, we apply the non-preemptable rule for tasks after they enter the distributed nested CS (DNCS) critical region to eliminate this situation. Note that applying non-preemption rule to the DNCS segments under all situations will cause deadlock.

One advantage of applying distributed priority inheritance in a dynamic priority system is that of flexibility. It does not require any *priori* information, including task priorities and resources accessed. Therefore, no computation and communication overhead is introduced by runtime workload changes.

4.3 Preemption Ceiling

One disadvantage of applying only the priority inheritance mechanism is the relatively loose bound of blocking for a critical segment with multiple resource accesses. We illustrate this by the situation shown in Table 3.

Assume at time t_1 we have a set of outermost CSs $\{S_1 \cdots S_n\}$ that all belong to the same processor *P*. We assume that they are granted resources $\{r_1 \cdots r_n\}$ respectively and their priorities are $p(S_1) > p(S_2) > \ldots > p(S_n)$. At t_1, S_1 enters the critical region because of its eminence. Then S_1 suspends itself during its DNCS execution and S_2 enters the critical segment since it is currently eminent. This can continuously happen and we assume that at time t_2 ($t_2 > t_1$), $\{S_1 \cdots S_n\}$ are all in the critical region and S_{n+1} comes with priority $p(S_{n+1}) > p(S_1)$ and requests $R(S_{n+1}) = \{r_1 \cdots r_n\}$. In this case, S_{n+1} will be blocked for $\sum_{i=1}^{n} E(S_n)$ even after applying the priority inheritance protocol.

Table 3. Priority Inversion Bound for CS Accessing Multiple Resources under Priority Inheritance Protocol

Applying the priority ceiling of tasks to the entrance of the critical region, which binds priority inversion to no more than one lower priority CS is efficient for fixed priority scheduling algorithms since the priority ceilings do not change during the scheduling time, given the *priori* information of fixed task priorities and resources accessed. However, for dynamic priority scheduling algorithms such as EDF, the expense of computing the dynamic priority ceilings in a distributed system can be high. When each job instance of a task finishes execution, the priority-ceilings for resources accessed by this task must be updated. After this, all conflicting tasks' priority-ceilings need to be updated. Finally, all processors' priority-ceilings that include any segment of the completed task or the conflicting tasks must be updated. This series of dynamic priority ceiling updates require network communication during the first two steps and is a regular task that the system must perform every time a CS completes.

Runtime updating of the dynamic priority ceiling at the completion of job J_i, an instance of τ_i can be expressed as follows, allowing R_i to represent the resource set that τ_i requests.

1. Updating the priority ceiling of each resource in R_i, local and remote.

$$O(N_l^{(i)} + \delta N_r^{(i)}) \tag{7}$$

   Where $N_l^{(i)}$ and $N_r^{(i)}$ represent the number of local and remote resources that J_i needs respectively and $N_l^{(i)} + N_r^{(i)} = N^{(i)}$ is the size of R_i.

2. Updating the priority ceilings of conflicting tasks, non-DNCSs and DNCSs that request any resource in R_i.

$$O\left(\sum_{r\in R_i}(N_{lcs}^{(r)}+N_{dncs}^{(r)}\cdot\delta\log 2n_d)\right) \quad (8)$$

Where n_d represents the processor depth of the DNCS, i.e., the number of processors has resources requested by J_i .

3. Updating the processor priority ceiling of all processors that has conflicting tasks with J_i currently in the critical region. This step depends on the number of tasks in critical region, which is time variant. But the average can be estimated by the probability.

$$O\left(\sum_{\tau_k\to r,r\in R_i}\frac{E_k}{T_k}\right) \quad (9)$$

The overhead for updating dynamic priority ceiling in a distributed system when J_i completes is (7)+(8)+(9). With a constant cost t_u for updating a priority ceiling, we represent cost for priority ceiling at job J_i's completion as:

$$U_i=t_u\left(N_l^{(i)}+\delta N_r^{(i)}+\sum_{r\in R_i}(N_{lcs}^{(r)}+N_{dncs}^{(r)}\cdot\delta\log 2n_d)+\sum_{\tau_k\to r,r\in R_i}\frac{E_k}{T_k}\right) \quad (10)$$

This runtime overhead can be reduced by applying the preemption ceiling protocol instead of dynamic priority ceilings at the entrance to the critical region. The Preemption Ceiling Protocol (PmCP) is based on an observation that in a dynamic priority scheduling such as EDF, the potential of a task's capability of preempting another task can be decided by certain static parameter of the task that are defined as the task's preemption level. The stack-based preemption ceiling protocol (SBP) prevents deadlock and bounds priority inversion in a uniprocessor system. However, directly applying SBP to the resource allocation stage in a distributed system is not feasible because of the locality of processors' stack spaces.

We extend the definition of preemption level to distributed systems that utilize a FIFO network model for both EDF and LLF and apply the preemption ceiling protocol on the basis of these new definitions. This includes segment preemption level as well as resource preemption-ceiling and the processor preemption-ceiling.

Segment preemption level

For a deadline-driven dynamic priority system, the runtime overhead of the priority ceiling update in DPCP may be avoided. Baker [3] illustrates one approach, based on the observation that the potentials for preemption in the EDF dynamic priority system do not change with time. Therefore, a static preemption-level $\pi(\tau)$ of task τ may be substituted for the dynamic priority level $p(J)$ in resource access contention. The inverse of the relative deadline $D(\tau)$ satisfies the requirement of a static preemption-level:

$$p(J) > p(J') \text{ and } Arrival(J) > Arrival(J') \text{ then } \pi(J) > \pi(J') \quad (11)$$

in EDF scheduling. This condition can be easily proved from the relationships $p(J) \propto d(J)^{-1} = (Arrival(J) + D(\tau))^{-1}$ and $\pi^{EDF}(\tau) \propto D(\tau)^{-1}$, where *d*(*J*) is the absolute deadline of job *J*.

LLF scheduling is different from EDF scheduling in that when each job in the same periodic task is initialized, their priorities are all same which is $p(J,t_0) \propto L(J,t_0)^{-1} = (D(\tau) - E(\tau))^{-1}$. However, as the job begins its execution, its laxity value remains unchanged whenever the job is executing and decreases whenever the job is waiting. Since a job's laxity value is monotonically decreasing, defining a preemption-level for LLF scheduling that is inversely proportional to the original laxity value of the job $\pi^{LLF}(\tau) \propto L(J,t_0)^{-1} = (D(\tau) - E(\tau))^{-1}$ will satisfy the condition. We claim this from the observation that if $t > t'$ and $L(J',t) > L(J,t)$, then $L(J',t') \geq L(J',t) > L(J,t)$. If we suppose that $Arrival(J) = t$ and $Arrival(J') = t'$, we will have the preemption-level requirement in (11).

During computation for the preemption level of a DNCS, we increase its original release time by the worst-case communication delay constant δ. We have the definition of preemption level for DNCS in EDF as $\pi^{EDF}_{DNCS}(\tau) \propto (D(\tau) - \delta)^{-1}$ and preemption level for DNCS in LLF as $\pi^{LLF}_{DNCS}(\tau) \propto \left(L(J,t_0) - \delta\right)^{-1} = (D(\tau) - E(\tau) - \delta)^{-1}$. These definitions satisfy the relation of (11) under the FIFO network model assumption. A formal proof is shown in Table 4.

An equivalent expression of (11) is :

$$p(S) > p(S') \wedge Arrival(S) > Arrival(S') \rightarrow \pi(S) > \pi(S')$$

For EDF scheduling, if S′ and S belong to the same processor, the definition of preemption ceiling for EDF directly derives (11). If S′ is released locally and S is released remotely, because of the FIFO network model assumption,

$$\left.\begin{array}{l} p(S) > p(S') \rightarrow d(S) < d(S') \\ release(S) + \delta > release(S') \end{array}\right\} \rightarrow$$

$$d(S) - release(S) - \delta < d(S') - release(S') \rightarrow \pi(S) > \pi(S').$$

If S′ and S are both released remotely, from the FIFO network model

$$\left.\begin{array}{l} p(S) > p(S') \rightarrow d(S) < d(S') \\ Arrival(S) > Arrival(S') \rightarrow release(S) > release(S') \end{array}\right\} \rightarrow$$

$$d(S) - release(S) - \delta < d(S') - release(S') - \delta \rightarrow \pi(S) > \pi(S')$$

So relation (11) holds under the preemption definition for EDF scheduling in the FIFO distributed system with maximum message delay.

For LLF scheduling, by subtracting the communication delay from the initial laxity for preemption level computation, we have the relation: $p\left(S, Arrival(S)\right) \leq \pi(S)$

Due to the fact that a lower priority *S′* can block *S* from execution under DPPmCP only when it is in critical region, we only need the relation (11) when *S′* is in the critical region at *Arrival*(*S*).

$$\left.\begin{array}{l} p(S, Arrival(S)) \le \pi(S) \\ p(S', Arrival(S)) \ge \pi(S') \ge p(S', Arrival(S')) \end{array}\right\} \rightarrow$$

$$\pi(S) \ge p(S, Arrival(S)) > p(S', Arrival(S)) \ge \pi(S') \rightarrow \pi(S) > \pi(S')$$

So relation (11) holds under the preemption definition for LLF scheduling in the FIFO distributed system with maximum message delay.

Hence, relation (11) holds for both the definition of preemption level in EDF and in LLF when preemption-ceiling rule applies. Therefore, we have the definition of preemption level for DNCS in EDF as

$$\pi_{DNCS}^{EDF}(\tau) \propto (D(\tau) - \delta)^{-1}$$

and preemption level for DNCS in LLF as

$$\pi_{DNCS}^{LLF}(\tau) \propto \left(L(J, t_0) - \delta\right)^{-1} = (D(\tau) - E(\tau) - \delta)^{-1}.$$

Table 4. Proof of Preemption Level Property in the FIFO Distributed System with Maximum Message Delay

Preemption ceilings for resources, critical segments and processor

For both EDF and LLF scheduling, we define the preemption ceiling of resource r bounded to processor P to be $\bar{\pi}(r) = \max\{\pi(S_i) \mid r \in R_i \wedge S_i \in P\}$. The resource preemption ceiling does not need to perform an update during segment activation or dynamic segment priority change (inheritance or updates). However, when new segments accessing r are registered, the preemption ceiling requires recalculation with a complexity of $O(1)$.

We also define the preemption ceiling of the critical segment to be the highest preemption ceiling of all of the resources in the critical segment and all its nested critical segments needs. That is, $\bar{\pi}(S_i) = \max\{\bar{\pi}(r) \mid r \in R_i \vee (r \in R_j \wedge S_i \supset S_j)\}$. It is obvious that an outmost critical segment has a preemption ceiling higher or equal to all of its nested critical segments according to this definition. The critical segment preemption ceiling is only updated when one of the resource preemption ceiling changes.

We define the processor preemption ceiling to be: $\hat{\pi}(P)$ = $\max\{\bar{\pi}(S) \mid$ all $CS \in P$ currently in critical region$\}$. The processor preemption ceiling requires updates when any critical segment enters or leaves the critical region. However, since segment preemption levels are static values given a fixed set of tasks and resources, these updates are pure local calculation of searching for maximum value no more complicated than looking up a preemption levels table. It requires more updates when any local resource's preemption ceiling changes, such as when the system workload or resources pool experience runtime changes. This process is similar to distributed priority ceiling updates yet it not a regular system task.

5. The Synchronization Protocols and a Proof of Correctness

We now present the Distributed Preclaiming Priority Inheritance Protocol (DPPIP) and the Distributed Preclaiming Preemption Ceiling Protocol (DPPmCP) for synchronization in a DSRT CORBA environment. DPPIP applies DRP and distributed priority inheritance schema while DPPmCP applies preemption ceiling schema in addition to DRP and distributed priority inheritance. The DRP schema is adapted to the DSRT CORBA model with the appropriate terminology modification. Distributed priority inheritance extends the priority inheritance schema to distributed systems. The non-preemptable rule for DNCSs in critical region is invoked to bind the priority inversion tighter. Instead of applying preemption ceiling in the resource allocation stage (preparing region), this schema is modified to restrict the entrance into the scheduling stage (critical region) for seamless cooperation with DRP and distributed priority inheritance. We provide a formal description for the Distributed Preclaiming Subprotocol, the Distributed Priority Inheritance Subprotocol and the Preemption Ceiling Subprotocol.

Rules of Distributed Preclaiming Subprotocol

1. *Resource Preclaiming Rule*: When a DT invokes its critical segment CS, the Local Scheduler (LS) of the CS's host processor puts the CS in a preparing status and sends request messages, on behalf of the CS, to the respective Resource Managers for all needed resources. The CS will remain in the preparing status until the LS receives all grant messages from the respective Resource Managers for all nested CSs and will then change its status to ready.
2. *Resource Allocation Rule*: CS will be granted resource R by Resource Manager
   a. If R is free.
   b. If the priority of current holder CS′ that is in the preparing or ready status has a priority such that $p(CS') < p(CS)$. CS′ is then preempted for resource R, returning to the preparing status if it is in the ready status.
   c. Otherwise, CS is blocked and enters the waiting queue of R in Resource Manager.
3. *Resource Release Rule*: when a critical segment exits, the LS will send release messages to all the respective Resource Managers to release all the resources that CS hold, except for resources still necessary for other CSs nested into the same outmost CS.

Rules of Priority Inheritance Subprotocol

1. *Priority Inheritance Rule*: The blocking critical segment and all of its lower priority nested critical segments in the preparing region inherits the priority of the highest priority critical segment blocked.
2. *Non-preemptable DNCSs*: A DNCS in the critical region is not preemptable.
3. *Scheduling Rule*: The highest priority segment S in the runners queue executes preemptively when no local DNCS is in the critical region.

Rule of Preemption Ceiling Subprotocol

1. *Preemption Ceiling Rule*: The highest priority critical segment *S* in the ready queue can enter the critical region when its preemption level is higher than the processor's preemption ceiling.

Table 5. Rules of DPPIP and DPPmCP subprotocols

The three subprotocols apply during different stage of the segment life cycle. *Distributed Preclaiming Subprotocol* applies during the preparing region and when a segment terminates. *Priority Inheritance Subprotocol* applies during the critical region of the segment life cycle, while *Preemption Ceiling Subprotocol* applies at the entrance from preparing region into critical region in DPPmCP. Separation of the representation of the three subprotocols enables us to clarify how these mechanisms coordinate with each other to provide all the properties we need for the synchronization of dynamic priority tasks in the distributed environment.

We now provide a formal proof of correctness for both DPPIP and DPPmCP. Three properties must be satisfied in order to ensure valid resource synchronization in a distributed real-time system: (1) mutual exclusion, (2) deadlock prevention, and (3) bounded priority inversion.

Mutual Exclusion:

Mutual exclusion is ensured through Theorem 1, shown below. Here *grant_set*$_i$ represents for the set of resources currently granted to critical segments *CS*$_i$. *R*$_i$ represents for the set of resources necessary for *CS*$_i$. $\overline{R_i}$ represents for the set of resources necessary for *CS*$_i$ or any of its nested critical segments.

Theorem 1:

Under control of *Distributed Preclaiming Subprotocol*, when *CS*$_i$ is in its critical region, there is no conflicting *CS*$_j$ ($j \neq i$) in the critical region.

Proof: CS segment CS$_i$ enters the critical region only when

$$grant\_set_i = \overline{R_i} = \bigcap_{\{k|R_i \supset R_k\}} R_k$$

and remains the granted_set of $\overline{R_i} \supseteq grant\_set_i \supseteq R_i$ during the critical region. Here, *grant_set*$_i$ includes a member *R* only when it receives the *grant*($R_{,i}$) message from resource manager *RM* guarding *R*. Since the *RM* only sends out a *grant*($R_{,i}$) message when *R* is free or when *R* is successfully preempted from another *CS*$_j$ in preparing region, i.e., when *release*(*R, k*) message is successfully received, we have the conclusion that if $R \in R_i$ and $i \neq j$, then $R \notin R_j$. Therefore, two conflicting critical segments cannot be in the critical region concurrently. The theorem follows.

Table 6. Proof of Mutual Exclusion Property for DPPIP and DPPmCP

Deadlock Prevention:

Deadlock prevention follows from Theorem 2, which relies on two lemmas.

Theorem 2

Under the control of the *Distributed Preclaiming Subprotocol*, no deadlocks will be formed when using priority-driven scheduling.

Lemma 2.1

Under control of the *Distributed Preclaiming Subprotocol*, a critical segment will not be blocked after it begins execution. Hence, no deadlock will be formed that involves a critical segment in the critical region.

Proof: *Distributed Preclaiming Subprotocol* prevents the formation of a distributed deadlock in the critical region by denying hold-and-wait. Since all critical segments request (preclaim) all resources needed, including nested critical segments' needed resources before execution, they are executed without resource requests. Together with the mutual exclusive resource access given by Theorem 1, no deadlock is formed involving a critical segment in the critical region. The lemma follows.

Lemma 2.2

Under the control of *Distributed Preclaiming Subprotocol*, no deadlock is formed that involves a critical segment in the preparing region.

Proof: The three conditions for deadlock formation are non-preemptive, hold-and-wait and cyclic waiting. In the preparing region, all preclaims for non-preemptive resources are preemptable. This is possible because the preclaim holder has not yet entered the critical region for execution. We now give the formal proof of the above.

Assume that at time *t*, *S* is the critical segment that has the highest priority among its conflicting critical segments. *S* sends out the request messages to respective processors' Resource Managers. We distinguish a starvation situation from a deadlock situation by assuming that no higher priority segments arrive after *t* and before S enters the critical region and no critical segments are currently in the critical region.

When the request of S arrives at the specific Resource Manager (*RM*) guarding r, three cases are possible:

1. r is free. Then S is granted r immediately and receives the *grant*(r) message no later than $t+2\delta$.
2. r is granted to a lower priority conflicting critical segment who is still in the preparing region. Then S preempts r and receives the *grant*(r) message no later than $t+4\delta$ with the *preempt*(r) message sent and the *release*(r) message received.
3. r is granted to a lower priority conflicting critical segment S' who is already in the critical region. From Lemma 4.2.1, S' will finish its execution in a bounded time Δt. Hence, S is granted r immediately after S' exits critical region and receives the *grant*(r) message no later than $t+ \Delta t +2\delta$.

Therefore, S will eventually be granted all the resources that it has requested after a bounded waiting time in the preparing status and a bounded waiting time in the ready status, allowing all higher priority segments in the critical region to finish and then enter the critical region for execution. Lemma 2.2 follows.

Concluding Lemma 2.1 and Lemma 2.2, no deadlock will be formed that involves segments in either preparing region or critical region when using priority-driven scheduling. Hence Theorem 2 follows.

Table 7. Proof of Deadlock Prevention Property for DPPIP and DPPmCP

Bounded priority inversion:

The priority inversion time for DPPIP and DPPmCP is the time a higher priority critical segment in the preparing status must wait for conflicting lower priority critical segments to exit the critical region and release the resources. We first analyze the priority inversion time of DPPIP and then give the bound of priority inversion time of DPPmCP on the basis of DPPIP analysis.

<u>Bounded priority inversion for DPPIP:</u>

In DPPIP, the invoked lower priority DNCSs in the ready status can at most experience priority inversion due to message delay through multicasting before they inherit the blocked segment's higher priority and preempt all intermediate priority segments to enter non-preemptable execution. Higher priority segments, critical or non-critical, can be blocked by a lower priority DNCSs that is in the critical region due to the non-preemptable property. Therefore, the worst-case priority inversion for a critical segment is the execution time of one conflicting outmost critical segments plus the duration of lower priority DNCSs in the local processor plus multicast delay.

Multicast delay is represented as $\delta\cdot\log(2n_d)$ where n_d is the nesting depth of the blocking lower priority DNCS. The logarithm comes from the fact that we can apply multicasting algorithms such as that of Message Passing Interface (MPI) that assumes there is no multicasting hardware available in the system. We have the bounded priority inversion property given by the following theorem.

Theorem 3

Under the control of DPPIP and priority-driven scheduling, a non-critical segment can be blocked by at most one lower priority conflicting outmost critical segment for each requested resource, lower priority local DNCSs and inheritance message multicast delay.

Proof: When DPPIP applies to task set without DNCSs, the rules of DPPIP subprotocol are identical to the basic priority inheritance protocol. Under this scenario, DPPIP inherits properties of the basic priority inheritance protocol proved in section III.B of [12]. According to "Lemma 5: Under the basic priority inheritance protocol, a job J_i can encounter blocking by at most one critical section in $\zeta^*_{i,k}$ for each semaphore S_k, $1 \le k \le m$, where m is the number of distinct semaphores. ", where $\zeta^*_{i,k}$ is the set of lower priority jobs conflicting jobs of job J_i over semaphore S_k, we derive Theorem 3 where the inheritance message multicast delay is zero.

When DPPIP applies to task set with DNCSs, the extension of DPPIP of the basic priority inheritance protocol applies when the DNCS *CS* of task τ conflicts with two tasks with higher and lower priority τ_H and τ_L respectively in two different processors. This scenario is a generalization of the example given in Figure 6. There are two possibilities for blocking of τ_H:

1. τ_L is in its critical section of CS_L and during the critical region, then according to the Distributed Preclaiming Subprotocol, *CS* of task τ cannot be in the critical region at the same time. Then τ_H is only blocked by the length of CS_L. No distributed priority inheritance occurs under this scenario.

2. τ is in its critical section of *CS* and during the critical region, then according to the Distributed Preclaiming Subprotocol, CS_L of task τ_L cannot be in the critical region at the same time. Resource Manager of τ send out multicast inheritance message to all nested segments of *CS*. If τ_L has higher priority than τ in their shared processor, then τ will resume running status from waiting status when it receives the inheritance message. Then τ_H is blocked by the length of *CS* plus inheritance message multicast delay.

Concluding the DPPIP property under task set with or without DNCSs, Theorem 3 follows.

Table 8. Proof of Bounded Priority Inversion for DPPIP

As is noted from our analysis in Section 4.1.2, one advantage of applying the distribute priority inheritance schema is its flexibility. Since it has no requirement for any *priori* information of the tasks and therefore no extra cost at the runtime workload change.

The runtime overhead of DPPIP is due to the multicast delay of DNCS priority inheritance messages. Although deeply DNCSs can lead to a high priority inversion time in a distributed system without multicast support, this case is extremely rare and should be intentionally avoided in a real-time system. A message delay from priority inheritance of uninvoked lower priority DNCSs does not occur. This is due to the fact that the priority of an uninvoked DNCS may be transferred at segment invocation that does not introduce any extra message delay.

Hence, the performance of DPPIP is good for regular cases. Its performance is especially good for a task set with a low number of average resource access per task. For task sets with high number of average resource access per task, introduction of the Preemption Ceiling Subprotocol, i.e., application of the DPPmCP is suggested.

Bounded Priority Inversion for DPPmCP:

As noted in Section 4.1.3, one advantage of DPPmCP is providing a tighter bound of priority inversion. DPPmCP does not apply the preemption ceiling to the resource allocation as it may induce deadlock and unbounded priority inversion. However, applying the preemption ceiling for the critical segment's entrance into the critical region can effectively bound priority inversion. The following theorem bounds DPPmCP's priority inversion time.

Theorem 4

Under control of DPPmCP for dynamic priority scheduling, a critical segment can be at most blocked for the duration of one lower priority conflicting outmost critical segment, one non-conflicting lower priority outmost critical segment with higher preemption ceiling and lower priority local DNCSs and multicast delay.

Proof: Assume that S_i arrives at time t and is the eminent outmost CS. At time $t+\delta$, all Resource Managers have received the resource requests from S_i. For a specific request of resource r, there are three possibilities.

1. If r is free, it is granted immediately.
2. If r is held by a lower priority segment in the preparing region, it is preempted and granted at t+4δ.
3. Otherwise r is held by a lower priority critical segment S_j in the critical region and S_i is blocked. We now prove that under (iii), S_i cannot be blocked by another lower priority conflicting critical segment S_k in the same processor.

Using the method of contradiction, we assume that at time $t'(t' > t+\delta)$, S_j and S_k are both in the critical region blocking S_i. We have the following condition.

$$\exists r, r',\ r \rightarrow S_j \wedge r' \rightarrow S_k \wedge r \in R(S_i) \wedge r' \in R(S_i).$$

Therefore, from the definition of preemption ceiling

$$\overline{\pi}(r) \geq \pi(S_i) \wedge \overline{\pi}(r) \geq \pi(S_j) \wedge \overline{\pi}(r') \geq \pi(S_i) \wedge \overline{\pi}(r') \geq \pi(S_k)$$

is true. Without losing generality, we can assume that S_k enters the critical region after S_j. Hence S_k can enter the critical region only if $\pi(S_k) > \overline{\pi}(r) \geq \pi(S_j)$. From the definition of preemption level, we can establish the following relation for the preemption level:

$$p(S_i) > p(S_k) \wedge Arrival(S_i) > Arrival(S_k) \rightarrow \pi(S_i) > \pi(S_k).$$

From this, we derive the following contradiction $\overline{\pi}(r) \geq \pi(S_i) > \pi(S_k) > \overline{\pi}(r)$.

Therefore, S_i can only be blocked by at most one lower priority conflicting critical segment S_k in the same processor.

As for a non-conflicting lower priority critical segment S_k with higher preemption ceiling, S_k can only block S_i when it enters the critical region before *Arrival*(S_i). There is not another critical segment S_j with lower priority but higher preemption ceiling level that will enter the critical segment before S_k leaves the critical region and S_i enters the critical region. Because if $p(S_j) < p(S_i)$ and $Arrival(S_j) < Arrival(S_i)$, then $\pi(S_j) < \pi(S_i) < \overline{\pi}(S_k)$. Therefore S_j cannot enter the critical region before S_k leaves the critical region and S_i enters the critical region.

From the above proof we can derive that the eminent critical segment S_i can be at most blocked by one lower priority conflicting critical segment, one non-conflicting lower priority outmost critical segment with higher preemption ceiling and lower priority DNCSs in any processor.

From the *Distributed Preclaiming Subprotocol*, S_i multicasts all it resource requests at *Arrival*(S_i) which will be all received at $Arrival(S_i) + \delta\ log(2n_d)$. Therefore, S_i is granted all resources and enters the critical segment after being blocked for the duration of the longest lower priority conflicting critical segment. Theorem 4 follows.

Table 9. Proof of Bounded Priority Inversion for DPPmCP

5. Conclusion

In this paper, we have examined the problem of real-time resource synchronization with a goal of providing efficient support for the message-based distributed and dynamic priority scheduling middleware environment of DSRT CORBA. We presented two new resource synchronization protocols for use in this environment. These protocols allow dynamic priority end-to-end RT CORBA tasks to simultaneously lock multiple resources on distributed processors. Both synchronization protocols may be applied to either EDF or LLF dynamic priority scheduling algorithms. Two fundamental research questions have been answered in this paper.

1. *How to support different dynamic scheduling algorithms?* We have chosen two candidate mechanisms: distributed priority inheritance and distributed preemption ceiling. The disagreement of preemption feature inherent to EDF and LLF is overcome by assigning preemption levels according to the dynamic scheduling algorithm while maintaining a consistent synchronization mechanism. This provides the resource manager with the capability to adapt to various scheduling algorithms. Therefore, the resource manager applying the protocol can be integrated with algorithm-adaptive dynamic schedulers, such as the one utilized in the TAO model of Washington University.
2. *How to minimize the runtime overhead?* We applied distributed priority inheritance and static preemption levels to avoid the expensive cost of dynamic priority ceilings runtime updates in a message based distributed system. With association of pessimistic DRP resource allocation, the context switch overhead is minimized with bounded priority inversion and low degree of parallelism tradeoff.

An interesting extension of the research is to provide concurrency control for RT CORBA transactional services. The importance of middleware and databases interoperability in network QoS has motivated a search for an optimal resource synchronization protocol. This interoperability depends on a CORBA Common Object Service (COS) known as the Object Transaction Service (OTS). This service supports operations with distributed transactional semantics. OTS in turn relies on Concurrency Control Service (CCS) to control transactions concurrency. In spite of the significance of real-time OTS and CCS in supporting middleware-database interoperability, their implementation lacks algorithms and protocols support. Our proposed protocols potentially support both real-time CCS and OTS. Future work will focus on the generation of algorithms and protocols supporting real-time CCS and real-time OTS.

References

1. **Common Object Request Broker Architecture (CORBA), v2.4.1, OMG, 2001.**
2. **Real-time CORBA 2.0: Dynamic Scheduling Specification, OMG, 1999.**
3. **Baker, T.P., A Stack-Based Resource Allocation Policy for Realtime Processes. in *11th IEEE Real-Time Systems Symposium*, (1990).**
4. **Chen, M.I. and Lin, K.J. Dynamic Priority Ceiling: A Concurrency Control Protocol for Real-Time Systems. *Real-Time Systems*, *2*. 325-346.**
5. **Dertouzos, M.L., Control Robotics: the Procedural Control of Physical Processes. in *IFIP '74*, (1974), 807-813.**
6. **Hermant, J.-F., Leboucher, L. and Rivierre, N. Real-Time Fixed And Dynamic Priority Driven Scheduling Algorithms: Theory And Experience, 1996, 142.**
7. **Leung, J.Y.-T. and Whitehead, J. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, *2*. 237-250.**
8. **Liu, C.L. and Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the Association for Computing Machinery*, *20* (1). 46-61.**
9. **Mok, A.K. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment *EECS*, MIT, 1983.**
10. **Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.**
11. **Rhee, I. and Martin, G., A Scalable Realtime Synchronization Protocol for Distributed Systems. in *16th IEEE Real-time Systems Symposium*, (1995).**
12. **Sha, L., Rajkumar, R. and Lehoczky, J.P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, *39* (9). 1175-1185.**