Distributed Priority Inheritance for Real-Time and Embedded Systems *

César Sánchez[1], Henny B. Sipma[1],
Christopher D. Gill[2], and Zohar Manna[1]

[1] Computer Science Department
Stanford University, Stanford, CA 94305, USA
{cesar,sipma,manna}@CS.Stanford.EDU
[2] Department of Computer Science and Engineering
Washington University, St. Louis, MO 63130, USA
cdgill@CSE.wustl.EDU

Abstract. We study the problem of priority inversion in distributed real-time and embedded systems and propose a solution based on a distributed version of the priority inheritance protocol (PIP). Previous approaches to priority inversions in distributed systems use variations of the priority ceiling protocol (PCP), originally designed for centralized systems as a modification of PIP that also prevents deadlock. PCP, however, requires maintaining a global view of the acquired resources, which in distributed systems leads to high communication overhead.
This paper presents a distributed PIP built on top of a deadlock avoidance schema that requires much less communication than PCP. Since the system is already deadlock free and priority inversions can be detected locally, we obtain an efficient dynamic resource allocation system that prevents deadlocks and handles priority inversions.

Keywords: Priority Scheduling, Deadlock Avoidance, Distributed Resource Allocation, Distributed Real-Time and Embedded Systems.

1 Introduction and Related Work

Modern distributed real-time and embedded systems (DRE) are built using a real-time middleware that extends conventional middleware services with real-time QoS capabilities. It is common in real-time systems to assign priorities to processes to achieve a higher confidence that the more critical tasks will be accomplished in time. A priority inversion is produced when a high priority process is blocked by a lower priority one. State-of-the-art middleware solutions, based for example on CORBA ORBs, may suffer priority inversions [24].

* César Sánchez, Henny B. Sipma and Zohar Manna are supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939. Christopher D. Gill is partially supported by NSF CAREER Award CCF-0448562.

In centralized systems, priority inversions are usually handled using a protocol from the priority inheritance family [26, 15]. However, priority inheritance, and synchronization in general, is significantly affected by concurrent execution. Even though some variations of these centralized protocols have been proposed for multiprocessors [15] and distributed systems [15, 13], there is not yet a widely accepted general scheme. We propose here a mathematically sound method to deal with priority inversions in DRE architectures.

In this paper we consider DRE systems that consist of a set of *sites*, each of which is capable of executing a predefined set of computations or methods, connected using an asynchronous network. Processes, consisting of local and remote method calls, are created dynamically. The relevant resources are the threads (or execution contexts) to run the methods. We assume a *WaitOnConnection* [25] thread-allocation policy, that is, each method requires its own thread, including nested up-calls, and methods hold on to their thread until they finish.

Since resources are finite and we impose no restriction on the number of processes, deadlocks are reachable unless there is a mechanism in place to control those allocations. It is important to distinguish between two different kinds of deadlocks: intra-resource (caused by parallel access to a single resource) and inter-resource deadlocks (caused by interference across different allocations).

Intra-resource deadlocks: Absence of intra-resource deadlock is one of the requirements of a solution to mutual exclusion, together with exclusive access and, sometimes, lack of starvation. Several algorithms have been proposed for distributed mutual exclusion, which can be classified (see [28, 29, 17]) into:

- Permission based: A process can access a resource if there is unanimity [19] between the participants about its safety. Unanimity can be relaxed to majority quorums [8, 31, 12], or even majorities in coteries [7, 1]. The message complexities range from $2(N-1)$ in the original Ricart-Agrawala algorithm [19] to $O(log\ N)$ in the best case (with no failures) in the modern coterie-based approaches.
- Token-based: The system is equipped with a single token per resource, which is held by the process that accesses it. A distributed data-type is maintained to select the next recipient. For example, Suzuki-Kasami's approach [30] exhibits a complexity of $O(N)$ messages per access, and Raymond's [16] and Naimi-Tehel's [14] approaches, use spanning trees to obtain an average case complexity of $O(log\ N)$, still exhibiting a $O(N)$ worst case.

However, the most efficient way (in terms of message complexity) to achieve distributed mutual exclusion is to use a centralized algorithm, like a primary site approach [2]. For every resource, a distinguished site arbitrates the accesses, reducing the problem of distributed mutual exclusion to the centralized case. Thus, allocations can be resolved with one message per request. This comes at a price of lower resiliency because, if the site managing the resource fails, the resource becomes inaccessible. However, in some cases, like DRE systems and Flexible Manufacturing Systems [6, 18] resources are indeed tightly coupled to the sites where they reside, and therefore it is natural to use this site as the

primary means to resolve each request contention. This is the basic approach that we use for intra-resource arbitration.

Inter-resource deadlocks: A different kind of deadlock can be produced due to the interleavings between accesses to different resources, when a set of processes is waiting in a circular chain in which a process holds a resource needed by the next process in the chain. The two mechanisms commonly used to ensure deadlock-free assignment of resource allocation are deadlock prevention and deadlock avoidance. A third mechanism, deadlock detection and resolution, is common in databases, but not used in DRE systems, because it may lead to unbounded worst-case execution times.

Deadlock prevention methods eliminate one of the necessary causes of the circular contention at design time, at the price of decreasing the concurrency. For example, "monotone locking", widely used in practice [3], determines an arbitrary total order on the set of resources that is followed at run-time to acquire resources.

Deadlock avoidance methods check the current resource allocation at runtime and grant a resource only if it is *safe*, that is, if there is a way for all processes to eventually finish. This check is made possible by having processes that enter the system announce their maximum resource usage, an approach first proposed by Dijkstra in his Banker's algorithm [5, 9, 10]. When resources are distributed across multiple sites, however, deadlock avoidance is harder, because different sites may have to consult each other to determine whether a particular allocation is safe. Because of this need for distributed agreement, a general solution to distributed deadlock avoidance is considered impractical [27]. Efficient solutions do exist, however, for the type of systems considered here, namely DRE systems with a *WaitOnConnection* thread-allocation policy. In [23, 22] we demonstrated an efficient distributed deadlock avoidance method for systems for which all the possible sequences of invocations are known and available to analysis a priori. In this paper we build on this algorithm to construct an efficient distributed priority inheritance protocol.

Priority Inheritance Protocols: It is common in real-time systems to assign priorities to processes. A priority inversion is produced when a process with high priority is unnecessarily blocked by a process with lower priority. To bound priority inversions, the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) were introduced, primarily applicable to hard real-time systems with shared resources and static priorities [26]. Later, these methods were extended to dynamic priority scheduling algorithms such as Earliest Deadline First (EDF) [4]. PIP can bound blocking times if a bound on the running time of each process and all its critical sections is known. PIP does not, however, prevent deadlocks, and therefore PCP was introduced to prevent inter-resource deadlocks, at the price of some concurrency.

A distributed version of PCP was proposed in [13] to deal with priority inversion and inter-resource deadlock in distributed systems. This protocol, however, suffers from a high communication overhead: before a request is granted, the ceiling of each resource that is (globally) used must be queried. This requires

maintaining global views of the system. Having more information about the system in the form of call graphs, however, we can use the simpler and more efficient priority inheritance protocol (PIP) to deal with priority inversions and use our deadlock avoidance algorithm to guarantee absence of inter-resource deadlocks. Our priority inheritance protocol allows more concurrency and, more importantly, it involves *no communication overhead when priority inversions are not present*, which in our setting is locally testable (it requires no communication to detect a priority inversion). Moreover, when priority inversions do exist, our protocol involves only one-way communication, without the need for return messages. Once an inversion is detected and the priority inheritance protocol is run—which may inject messages into the network—the local processes can proceed immediately without compromising deadlock freedom. This leads to a more efficient solution, especially in scenarios where latencies are significant compared to the running times of methods.

Our PIP protocol enables the computation of a bound on the number of lower priority processes that can block a higher priority one. Consequently, the blocking time of a process can also be bound if the maximum running time of each method and the latency of each message is known. This solution enables the following design methodology for DRE systems. A distributed system with periodic and sporadic tasks with deadlines can be analyzed for schedulability: (1) computing initial priorities of processes statically, and (2) showing a proof that deadlines are met in all possible executions. In this paper we prove the correctness of the distributed priority inheritance protocol, and leave distributed schedulability analysis for future research.

The rest of this paper is structured as follows. Section 2 reviews our distributed deadlock avoidance algorithms. Section 3 includes the distributed priority inheritance protocol and proves its correctness, and Section 4 presents our conclusions and describes future work.

2 Distributed Deadlock Avoidance

We model a distributed system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ by a set of *sites* and a *call graph specification*. The sites $\mathcal{R} : \{r_1, \ldots, r_{|\mathcal{R}|}\}$ model distributed devices that perform computations and handle a necessary and scarce local resource, such as a finite pool of threads. A call graph specification $\mathcal{G} : \langle N, \rightarrow, I \rangle$ consists of a directed acyclic graph $\langle N, \rightarrow \rangle$, which captures all the possible sequences of remote calls that processes can perform. The set of initial nodes $I \subseteq N$ contains those methods that can be invoked when a process is spawned.

A call graph node abstracts both the computation to be performed at a site and the resource needed. Each node has a unique name (the method name) and a site associated with it, the site where the method will be executed at run-time. If node n describes resource $(f : r)$ we say that n executes computation f and *resides* in site r. We use the predicate $n \equiv_{\mathcal{R}} m$ to represent that nodes n and m reside in the same site. To simplify notation, if the method name is unimportant we use $n : r$ instead of $n = (f : r)$. We use $r, s, r_1, r_2, \ldots$ to refer to sites and $n, m, n_1, m_1, \ldots$ to refer to call graph nodes.

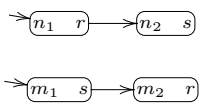
An edge $n \to m$ in the call graph denotes a possible remote invocation; in order to complete the computation modeled by n the result of a call to m may be needed. If this call is performed, the resource associated with n will be locked at least until the invocation of m returns. Note how this call semantics is not equivalent to synchronous calls since we do not assume that the caller needs a response to continue the computation but only needs a response to *complete* its execution. For example, after initiating a remote method call, the caller can immediately continue and perform more remote invocations before waiting for the reply. All our results immediately cover synchronous semantics as a particular case (in which callers do wait for a response before proceeding) and can be easily adapted to totally asynchronous semantics (where processes are even allowed to terminate without waiting for responses).

A run of a system consists of the execution of processes, created dynamically. When a new process is spawned it announces an initial call graph node whose outgoing paths capture the remote calls that the process may perform. Incoming invocations require a new resource to run, while call returns release a resource. We use the following terminology: every new method invocation is called a *process*, and the context will disambiguate between subprocesses (created by remote calls) and proper processes (corresponding to new instances entering the system). Once a process has received a resource, it holds onto it until completion, that is, there is no preemption once a resource is acquired. We also assume that all computations terminate, if their demanded resources are granted.

Even though in principle our computational model can be regarded as non-preemptive, methods that assume preemptive scheduling (the setting where classical priority inheritance with rate-monotonic scheduling was introduced [11]) are also relevant. This is because in our model, all computations occur inside critical sections (resources are nested).

Deadlocks can be reached if all resource requests are immediately granted, since resources are finite and fixed a priori in each site and—in principle—we impose no restriction on the topology of the call graph specification or the number of process instances. We use T_r for the total number of resources in site r, and t_r for a variable that keeps track of the number of resources available in r at every instant. Initially, $t_r = T_r$.

Example 1. Consider a system with sites $\mathcal{R} = \{r, s\}$, nodes $N = \{n_1, n_2, m_1, m_2\}$ with n_1 and m_1 initial nodes, and call graph

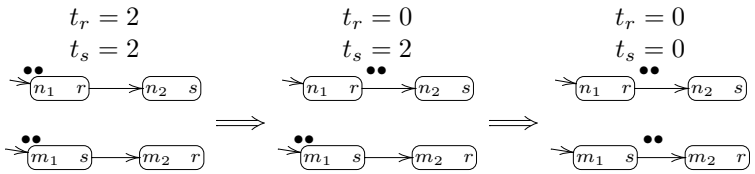


This system has reachable deadlocks if no controller is used. Let sites s and r handle exactly two threads each. If four processes are spawned, two instances of n_1 and two of m_1, all resources in the system will be locked after each process starts executing its initial node. Consequently, the allocation attempts for n_2 and m_2 will be blocked indefinitely, so no process will terminate or return a

$$n :: \left[\begin{array}{l} \left[\begin{array}{c} \textbf{when } \alpha(n) \leq t_r \textbf{ do} \\ t_r\texttt{--} \end{array}\right] \\ f() \\ t_r\texttt{++} \end{array}\right]$$

Fig. 1: The deadlock avoidance protocol BASIC-P.

resource. A possible allocation sequence is shown below,



where a • represents an existing process that tries to acquire a resource at a node (if • precedes the node) or has just been granted the resource (if • appears after the node). It is easy to see that a deadlock can still occur even if the number of threads in r and s is increased. We can simply spawn the corresponding number of processes.

In our solution to deadlock avoidance, the assignment of resources is controlled by two cooperating components: the *allocation manager* and the *scheduler*. The allocation manager decides which subset of pending requests is safe (in the sense that no continuation of the execution will reach a deadlock if granted); these requests are called *enabled*. The scheduler then chooses a process among the enabled ones, which receives the resource. This interaction between the allocation manager and the scheduler is repeated until the set of enabled processes is empty. Processes whose request is disabled are called *waiting*, while processes that hold a resource are called *active*.

A deadlock avoidance algorithm is an allocation manager that guarantees that no deadlock can be reached, independently of the scheduler used. Our deadlock avoidance algorithms consist of two parts:

1. A computation of *annotations* of call graph nodes, carried out statically. We consider maps from nodes to natural numbers $\alpha : N \mapsto \mathbb{N}$ as annotations.
2. A *protocol*: a piece of code that ensures, at runtime, that allocations and deallocations are safe. It consists of two stages: one that runs when the resource is requested, and another when the resource is released. We are seeking protocols that only inspect and modify local variables of the site.

The deadlock avoidance protocol BASIC-P [23] for node $n = (f : r)$ is shown in Fig. 1. The *entry section* that precedes the access to the method call $f()$ consists of a guard and an action that operate on local variables of site r. The guard captures the enabling condition of the request for node n. We assume that the entry section is executed atomically, as a test-and-set operation. A requesting process executing node $n : r$ is enabled if there are at least $\alpha(n)$

resources available in r. Note, however, that only one resource is acquired. The *exit section* is executed when the method terminates and consists of an action that updates local variables, in this case increasing t_r. The execution of the exit section triggers the allocation manager to re-evaluate the entry condition of the waiting processes. Those requests for resources found safe, if any, are handed over to the scheduler, which chooses one to be granted.

The most important property that protocols must enforce is freedom from deadlock. The following is a characterization of deadlock:

Definition 1 (Deadlock). *A deadlock is a global state in which there is a non-empty set of disabled processes that continue to be disabled even if all the other processes in the system return their acquired resources.*

BASIC-P avoids deadlocks if the annotation is acyclic in the following sense. Given a system $\langle \mathcal{R}, \mathcal{G} \rangle$ and an annotation α, the annotated call graph $(N, \rightarrow, \dashrightarrow)$ adds to $\mathcal{G}$ one edge $n \dashrightarrow m$ for every pair of nodes n and m that reside in the same site and $\alpha(n) \geq \alpha(m)$. A node n depends on a node m, represented as $n \succ m$, if there is a path in the annotated graph from n to m that follows at least one $\rightarrow$ edge. The annotated graph is acyclic if no node depends on itself, in which case we say that the annotation is acyclic.

Theorem 1 (Annotation Theorem for Basic-P [23]). *Given a system and an acyclic annotation, if* BASIC-P *is used in every node to control resource allocations then all executions of the system are deadlock free.*

Example 2. Reconsider the system from Example 1. The left diagram below shows an annotated call graph with $\alpha(n_1) = \alpha(n_2) = \alpha(m_2) = 1$ and $\alpha(m_1) = 2$. It is acyclic, and thus by Theorem 1, if BASIC-P is used with these annotations, the system is deadlock free.

Let us compare this with Example 1 where a resource is granted simply if it is available. This corresponds to using BASIC-P with the annotated call graph above on the right, with $\alpha(n) = 1$ for all nodes. In Example 1 we showed that a deadlock is reachable, and indeed this annotated graph is not acyclic; it contains dependency cycles, for example $n_1 \rightarrow n_2 \dashrightarrow m_1 \rightarrow m_2 \dashrightarrow n_1$. Therefore Theorem 1 does not apply. In the diagram on the left all dependency cycles are broken by the annotation $\alpha(m_1) = 2$. Requiring the presence of at least two resources for granting a resource at m_1 ensures that the last resource available in s can only be obtained at n_2, which breaks all possible circular waits.

The priority inheritance protocol presented in the next section is based on BASIC-P. Its correctness relies on the following property.

Theorem 2 (Reachable state space [20]). *The set of global states reachable by* BASIC-P *contains precisely those states in which, for all sites r and annotations k*

$$\varphi: \quad A_r[k] \leq T_r - (k-1) \ ,$$

where $A_r[k]$ denotes the number of active processes in site r executing call graph nodes with annotation k or higher.

The global states that satisfy φ are called φ-states. Since BASIC-P guarantees deadlock free operation, Theorem 2 implies, for example, that if a system is in a φ-state and BASIC-P is used as an allocation manager in every site, then there is some enabled process. In [21, 20] we introduced more efficient protocols (EFFICIENT-P, k-EFFICIENT-P and LIVE-P) and proved annotation theorems similar to Theorem 1, but BASIC-P is simpler and it is enough to illustrate the present discussion. In the remainder of the paper we will assume that BASIC-P is used to allocate the resources.

3 Priority Inheritance

In this section we develop a priority inheritance mechanism and show how it helps to alleviate priority inversions. A priority specification extends a system specification with a description of the possible priorities at which processes can run. The fixed set $\mathcal{P}$ of priorities is a finite and totally ordered set. Without loss of generality, we take $\mathcal{P} = \{1, \ldots, p_m\}$, where lower value means higher priority: 1 represents the highest priority and p_m the lowest.

Definition 2. *Given a system $\langle \mathcal{R}, \mathcal{G} \rangle$ and set of priorities $\mathcal{P}$, a* priority assignment *is a map from initial nodes I to sets of priorities:*

$$\Pi : I \to 2^{\mathcal{P}}.$$

A priority specification $\langle \mathcal{R}, \mathcal{G}, \Pi \rangle$ *equips the system with a priority assignment.*

In the prioritized setting, when a process is created, it declares both the initial node i—as in the unprioritized case before— and its initial priority from $\Pi(i)$, called the *nominal priority* of the process. Informally, a process L will run at its nominal priority, and "accelerate" to a higher priority when some higher priority process H is waiting for some resource that L holds. This will prevent processes running at intermediate priorities from making L wait and blocking H indirectly. We now define the distributed priority inheritance protocol:

(PI.1) A process maintains a *running priority*, which is initially its nominal priority.

(PI.2) Let P be a process, running at priority p, that is denied access to a resource in site r, and let Q be an active process in r running at a priority lower than p. Q and all its subprocesses set their priority to p or their current running priority, whatever is higher. We say that Q is *accelerated* to p.

(PI.3) When a (sub)process is accelerated it does not decrease its running priority until completion.

(PI.2) may require sending acceleration messages for subprocesses running in remote sites. (PI.2) does not require changing the priority of ancestor processes

since the acceleration of a caller cannot help the callee to finish earlier. (PI.3) states that a (sub)process cannot decelerate. In general, decelerations compromise deadlock freedom.

It is easy to see that a subprocess either runs at priority at least as high as any of its ancestors, or there are undelivered acceleration messages.

We now calculate the sets of possible priorities at which a call graph node can be executed. A node $n{:}r$ can be executed at a priority p either if $p \in \Pi(i)$ for some initial ancestor of n, or if a process can execute n running at priority lower than p and block another process running at p. This block can be produced either if there is some node in r that can be executed at p, or if some ancestor of n can block some process executing at p. Formally, the set of pairs (n, p) representing priorities p at which a node n can run is the smallest set $N^{pr} \subseteq N \times \mathcal{P}$ containing:

1. (n, p) for every n that descends from $i \rightarrow^* n$, $i \in I$ and $p \in \Pi(i)$.
2. (n, p) for every $(m, p) \in N^{pr}$ with $n \equiv_{\mathcal{R}} m$, and $(n, q) \in N^{pr}$ for some $q \geq p$, and
3. (n, p) for some ancestor $m \rightarrow^+ n$ that can also run at p, $(m, p) \in N^{pr}$.

Example 3. Consider the call graph

and the priority assignment $\Pi(n) = \{1\}, \Pi(m) = \{2\}, \Pi(o_1) = \{3\}$. The set of potential priorities is:

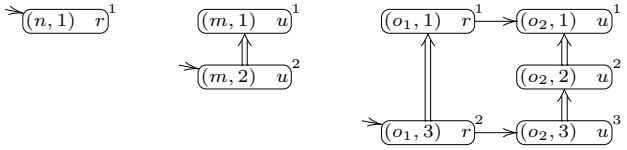
n	m	o_1	o_2
$\{1\}$	$\{1, 2\}$	$\{1, 3\}$	$\{1, 2, 3\}$

Node o_1 can run at priority 1 because o_1 resides in the same site as n and n can run at 1. Since o_2 is a descendant of o_1, o_2 can also run at 1, and since m resides in the same site as o_2, m can also run at 1. Moreover, m can run at 2, higher than o_2 running at 3, so o_2 can also run at 2. Thus $N^{pr} = \{(n, 1), (m, 1), (m, 2), (o_1, 1), (o_1, 3), (o_2, 1), (o_2, 2), (o_2, 3)\}$.

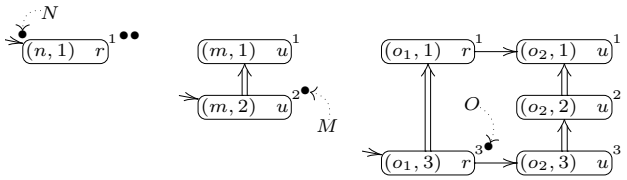
We extend the state transition system that models the global behavior of our model of distributed systems [23] with a new transition called *acceleration*. When an acceleration transition is taken, a process P running at priority p accelerates to higher priority q ($q < p$). It is easy to see that, if the priority inheritance protocol is used, and a process running in n can accelerate from priority p to q, then both (n, p) and (n, q) are in N^{pr}.

The notion of annotation can be adapted to prioritized specifications. A prioritized annotation α is a map from N^{pr} to the natural numbers. It *respects priorities* if for every two pairs (n, p) and (m, q) in N^{pr}, with $n \equiv_{\mathcal{R}} m$, $\alpha(n, p) > \alpha(m, q)$ whenever $p > q$, that is, if *higher priorities receive lower annotations*. As with unprioritized call graphs, we define an annotated call graph by adding an edge relation $\overset{pr}{\dashrightarrow}$ connecting $(n, p) \overset{pr}{\dashrightarrow} (m, q)$ whenever n and m reside in the same site and $\alpha(n, p) \geq \alpha(m, q)$. If there is a path from (n, p) to (m, q) that contains at least a $\overset{pr}{\rightarrow}$ edge we say that (n, p) depends on (m, q), and we write $(n, p) \succ (m, q)$. An annotation is *acyclic* if no pair depends on itself.

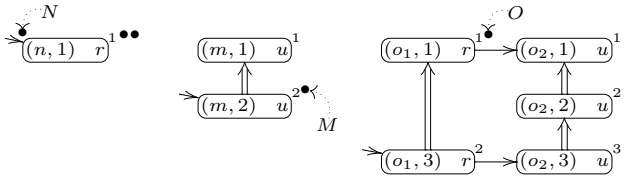
Example 4. The following diagram represents the annotated call graph of Example 3 with ⇒ arrows representing accelerations. This annotated call graph is acyclic.



Example 5. This example shows how priority inheritance bounds the blocking time caused by priority inversions. Reconsider the annotated call graph of Example 4. Let the total number of resources be $T_r = 3$ and $T_u = 3$. Let σ be a state in which two active processes are running in n at priority 1, one active process M is running in m at priority 2, and one active process O is running in o_1 at priority 3. Thus the available resources in σ are given by $t_r = 0, t_u = 2$. Let N be a new process spawned to run in n with nominal priority 1. The resulting state is shown below.



N is blocked trying to access $(n, 1)$. There is a priority inversion since O holds an r resource in o_1, while running at lower priority 3. If no acceleration is performed, then the remote call of O to o_2 is blocked until M completes, so N will be blocked indirectly by M (see Fig. 2 (a)). Even worse, if there are several processes waiting in $(m, 2)$, all these processes will block O and indirectly N, causing an unbounded blocking delay (see Fig. 2 (b)). With priority inheritance in place, O inherits the priority 1 from N, and the resulting state after the acceleration is:

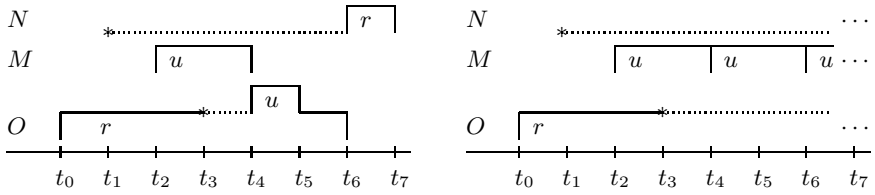


In this state, O will be granted the resource in o_2 in spite of M (and potentially other priority 2 processes waiting at m) and O will terminate, freeing the resource demanded by N. In this case the blocking time of N is bounded by the running time of O at priority 1, as shown in Fig. 3.

The following results hold in spite of when and how accelerations are produced:

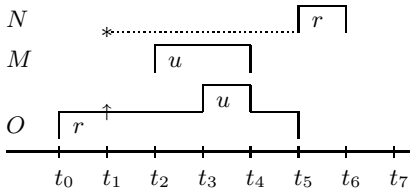
Lemma 1. *If an annotation respects priorities, then accelerations preserve φ.*

Proof. Let P be a process that accelerates from priority p to q. If P is waiting, the result holds immediately since the global state does not change. If P is active at a node $n : r$, its annotation $\alpha(n, p) > \alpha(n, q)$ decreases. Therefore, all terms $A_r[k]$ are either maintained or decreased, and φ is preserved. □



(a) Blocking due to Priority Inversions (b) Unbounded Blocking

Fig. 2: Time diagram (a) shows an execution of the scenario in Example 5 with blocking priority inversions. Diagram (b) shows an execution with unbounded blocking time due to priority inversions. In both diagrams, at instant t_0, process O is created, and its request for an r-resource is granted. At t_1, process N is created, but due to the existence of O and two active processes in n_1, its request is denied, indicated by $*$. At t_2, M is spawned to run m and its request for a u-resource is granted. This causes the request of O to execute o_2 at t_3 to be denied. O can only execute o_2 when some resource in u is freed. Therefore M is blocking N indirectly. In diagram (a) this blocking is restricted to the interval (t_3, t_4), while in diagram (b) the blocking delay is unbounded.



(a) No Blocking with Priority Inheritance

Fig. 3: At time t_1, process O inherits priority 1 from process N, depicted by $\uparrow$. This allows O to acquire the u-resource and execute o_2 at t_3, in spite of the existence of M or other processes trying to execute m at priority 2. Consequently, N can start executing n at t_5, with no blocking delay.

Corollary 1. *The set of reachable states of a prioritized system that uses* Basic-P *as an allocation manager with an acyclic annotation that respects priorities is a subset of the φ-states.*

In the rest of this section we show that if Basic-P is used to control allocations, and accelerations are produced according to the priority inheritance protocol, deadlocks are not reachable. When a process inherits a new priority, all its existing subprocesses, including those in other sites produced as a result of remote invocations, must accelerate as well. A message is sent to all sites where a subprocess may be running. When the message is received, if the process exists, then it is accelerated. If it does not, the acceleration is recorded as a future promise. We first show deadlock-freedom if all acceleration requests are delivered immediately with global atomicity. Then, we complete the proof for asynchronous delivery in general.

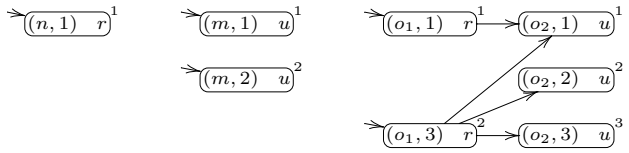
3.1 Priority Inheritance with Global Atomic Accelerations

Given a prioritized system $\mathcal{S}$ we build an (unprioritized) system and show that if $\mathcal{S}$ has reachable deadlocks so does the derived one.

Definition 3 (Flat call graph). *Given a priority specification $\langle \mathcal{R}, \mathcal{G}, \Pi \rangle$, a flat call graph is $\mathcal{G}^\flat : \langle N^{pr}, \xrightarrow{pr}, I^{pr} \rangle$, where N^{pr} is the set of potential priorities, there is an edge $(n,p) \xrightarrow{pr} (m,q)$ if $p \geq q$ and $n \to m$ occurs in the original call graph, and $(i,p) \in I^{pr}$ if i is initial in $\mathcal{G}$.*

We use $\mathcal{S}^\flat : \langle \mathcal{R}, \mathcal{G}^\flat \rangle$ for the (unprioritized) flat system that results from the flat call graph. It is easy to see that the set of reachable states of a process (the resources and running priorities of the process and each of its active subprocesses) is the same in a system and in its flat version. Moreover, if an annotation α of a prioritized specification is acyclic and respects priorities then α, when interpreted in the flat call graph, is acyclic.

Example 6. The flat call graph for the annotated specification in Example 3 is



Theorem 3. *Given a prioritized system $\mathcal{S}$ and an acyclic annotation that respects priorities, every global state reachable by $\mathcal{S}$ is also reachable by $\mathcal{S}^\flat$, if* Basic-P *is used as an allocation manager.*

Proof. Follows directly from Corollary 1 and Theorem 2. □

It is important to note that Theorem 3 states that for every sequence of requests and accelerations that leads to state σ in $\mathcal{S}$, there is a—possibly different but also legal— sequence that leads to σ in $\mathcal{S}^\flat$. Theorem 3 does not imply, though, that every transition in $\mathcal{S}$ can be mimicked in $\mathcal{S}^\flat$, which is not the case in general for accelerations. A consequence of Theorem 3 is that deadlocks are not reachable in $\mathcal{S}$, since the same deadlock would be reachable in $\mathcal{S}^\flat$, which is deadlock free by the Annotation Theorem.

Corollary 2. *If α is an acyclic annotation that respects priorities, and* Basic-P *is used as a resource allocation manager in every node, then all runs of $\mathcal{S}$ when accelerations are executed in global atomicity are deadlock free.*

The following section shows that the requirement for global atomicity in the previous corollary is actually not necessary.

3.2 Priority Inheritance with Asynchronous Communication

When an arbitrary asynchronous communication subsystem is assumed, with no guarantees of globally atomic delivery of messages, the proof of deadlock freedom is more challenging. In this case, the flat system does not directly capture the reachable states of the system with priorities, since subprocesses may accelerate later than their ancestors.

Theorem 4 (Annotation Theorem for Prioritized Specifications). *If α is an acyclic annotation that respects priorities, and* Basic-P *is used as a resource allocation manager for every call graph node, then all runs of $\mathcal{S}$ are deadlock free.*

Proof (sketch). Assume, by contradiction, that deadlocks are reachable, and let σ be a state in which a set $\{P\}$ of processes forms a deadlock. Note that σ need not be a reachable state of the flat system $\mathcal{S}^\flat$. Consider an arbitrary continuation of the run, and let σ' be the first state in which there is no undelivered message containing an acceleration of a process in the set $\{P\}$. Such a state exists since the set of possible accelerations is finite and all messages are eventually delivered. In σ' if all the processes not involved in the deadlock (i.e., not in $\{P\}$) return their resources the resulting state becomes a φ-state, and therefore some process (in $\{P\}$) can progress if Basic-P is used as an allocation manager. This contradicts the assumption that σ is a deadlock state. □

4 Conclusions and Further Work

We have presented a distributed priority inheritance protocol built using a deadlock avoidance mechanism, and proved its correctness. This protocol involves less communication overhead than a distributed PCP, since inversions can be detected locally, while PCP requires a global view of the resources allocated. Our approach enables the calculation of bounds on the maximum blocking times, which is necessary for schedulability analysis.

Message Complexity: The message complexity of a naïve implementation of the priority inheritance protocol described here is given by the number of different sites of the set of descendant nodes, which in the worst case is $|\mathcal{R}|$. However, this communication is one-way, in the sense that once the message is sent to the network, the local process can immediately accelerate, increasing its running priority. Moreover, broadcast can be used when available. Also, under certain semantics for remote calls this worst case bound can be improved. For example, with synchronous remote calls (the caller is blocked until the remote invocation returns), one can build, using a pre-order traversal of the descendant sub-tree, an order on the visited sites. Then, a binary search on this order can be used to find the active subprocess where the nested remote call is executing. This gives a worst case ($log\ |\mathcal{R}|$) upper-bound on the number of messages needed for each priority inheritance.

Dynamic Priorities: Most dynamic priority scheduling algorithms, like EDF, require querying for the current status of existing processes to define their relative priorities. Our priority inheritance mechanism can be used with dynamic priorities if there is some static discretization of the set of priorities that processes may run at. To ensure the correctness of the priority inheritance protocol shown here, subprocesses must only increase (never decrease) their priorities while running. Note that in this kind of scheduling algorithm, accelerations would not only be caused by a priority inversion but also by the decision of a process to increase its priority to meet a deadline.

References

1. Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, February 1991.
2. Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*, pages 562–570, Los Alamitos, CA, 1976. IEEE Computer Society Press.
3. Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.
4. Giorgio C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, New York, NY, 2005.
5. Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
6. Maria Pia Fanti and MengChu Zhou. Deadlock control methods in automated manufacturing systems. *IEEE Transactions on Systems, Man and Cybernetics–Part A: Systems and Humans*, 34(1):347–363, January 2004.
7. Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.
8. David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles (SOSP '79)*, pages 150–162, Pacific Grove, CA, 1979. ACM Press.
9. Arie N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12:373–377, 1969.
10. James W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.
11. C. J. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
12. Mamory Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.
13. Frank Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Proceedings of 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 340–349, Phoenix, AZ, December 1999. IEEE Computer Society.
14. Mohamed Naimi, Michel Trehel, and André Arnold. A $log(n)$ distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.
15. Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A priority inheritance approach*. Kluwer Academic Publishers, 1991.

16. Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, February 1989.
17. Michel Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986.
18. Spyros A. Reveliotis. *Real-time Management of Resource Allocation Systems: a Discrete Event Systems Approach*. International Series In Operation Research and Management Science. Springer, 2005.
19. Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.
20. César Sánchez and Henny B. Sipma. Reachable state spaces of distributed deadlock avoidance algorithms. Technical Report REACT-TR-2006-01, Stanford Computer Science, REACT Group, June 2006. Available from `http://theory.stanford.edu/~cesar`.
21. César Sánchez, Henny B. Sipma, Zohar Manna, and Christopher Gill. Efficient distributed deadlock avoidance with liveness guarantees. In *To appear in the Proceedings of the 6th Annual ACM Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, 2006. ACM Press.
22. César Sánchez, Henny B. Sipma, Zohar Manna, Venkita Subramonian, and Christopher Gill. On efficient distributed deadlock avoidance for distributed real-time and embedded systems. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodas, Greece, 2006. IEEE Computer Society Press.
23. César Sánchez, Henny B. Sipma, Venkita Subramonian, Christopher Gill, and Zohar Manna. Thread allocation protocols for distributed real-time and embedded systems. In Farn Wang, editor, *25th IFIP WG 2.6 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, volume 3731 of *LNCS*, pages 159–173, Taipei, Taiwan, October 2005. Springer-Verlag.
24. Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha S. Gokhale. Alleviating priority inversion and non-determinism in real-time CORBA ORB core architectures. In *Proc. of the Fourth IEEE Real Time Technology and Applications Symposium (RTAS'98)*, pages 92–101, Denver, CO, June 1998. IEEE Computer Society Press.
25. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.
26. Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
27. Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. McGraw-Hill, Inc., New York, NY, 1994.
28. Julien Sopena, Luciana Arantes, Marin Bertier, and Pierre Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Proceedings of Euro-Par'05*, volume 3648 of *LNCS*, pages 654–663, Lisboa, Portugal, 2005. Spring-Verlag.
29. Pradip K. Srimani and Sunil R. Das, editors. *Distributed Mutual Exclusion Algorithms*. IEEE Computer Society Press, 1992.
30. Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.
31. Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.