Efficient Distributed Deadlock Avoidance with Liveness Guarantees

César Sánchez Henny B. Sipma
Zohar Manna

Computer Science Department *
Stanford University
Stanford, CA 94305, USA

Christopher D. Gill

Computer Science and Eng. Dept.
Washington University
St. Louis, MO 63130, USA

ABSTRACT

We present a deadlock avoidance algorithm for distributed systems that guarantees liveness. Deadlock avoidance in distributed systems is a hard problem and general solutions are considered impractical due to the high communication overhead. In previous work, however, we showed that practical solutions exist when all possible sequences of resource requests are known a priori in the form of call graphs; in this case protocols can be constructed that perform safe resource allocation based on local data only, that is, no communication between components is required. While avoiding deadlock, those protocols, however, did not avoid starvation: they guaranteed that some process could always make progress, but did not guarantee that every individual process would always eventually terminate.

In this paper we present a resource allocation mechanism that avoids deadlock and guarantees absence of starvation, without undue loss of concurrency. The only assumption we make is that the local scheduler is fair. We prove the correctness of the algorithm and show how it can be implemented efficiently.

Categories and Subject Descriptors

D.4.1 [**Software**]: Operating Systems—*Process management*; D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*

General Terms

Theory, Reliability, Algorithms

*This research was 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.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

Keywords

Scheduling, Deadlock Avoidance, Distributed Algorithms

1. INTRODUCTION

Computations in distributed systems often involve a distribution of method calls over multiple sites. At each site these computations need resources, in the form of threads, to proceed. With multiple processes starting and running at different sites, and a limited number of threads at each site, deadlock may arise.

Traditionally three methods are used to deal with deadlock: deadlock prevention, deadlock avoidance, and deadlock detection. In *deadlock prevention* a deadlock state is made unreachable by violating one of the necessary conditions for deadlock. For example, imposing a fixed order in which resources are acquired, such as in "monotone locking" [1], violates the condition for a cyclic dependency among resources. This strategy, however, imposes some burden on the programmer, and—often a more important concern in embedded systems—can substantially reduce performance, by artificially limiting concurrency. With *deadlock detection* methods deadlock states may occur, but are upon detection resolved by, for example, roll-back of transactions. This approach is common in databases. In embedded systems, however, this is usually not an option, especially in systems interacting with physical devices.

Deadlock avoidance methods take a middle route. At runtime a protocol is used to decide whether a request for resources is granted based on current resource availability and possible future requests of processes in the system. A resource is granted only if it is *safe*, that is, if all processes still have a strategy to complete. To make this possible processes that enter the system must inform the protocol about their expected resource usage. The best known algorithm following this strategy is Dijkstra's Banker's algorithm [4, 5, 6, 15, 13], where a process upon creation reports the maximum number of resources of each type that it can request during its execution. This information is then taken into account in the decision of whether to allow later processes to enter the system. When resources are distributed across multiple sites, however, deadlock avoidance is harder, because the 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 [14]; the communication costs involved simply outweigh the benefits gained from deadlock avoidance over deadlock prevention.

In this paper we propose a distributed deadlock avoidance algorithm that does not require any communication between sites. The algorithm is applicable to distributed systems in which processes perform method invocations at different sites and lock local resources (threads) until all remote calls have returned. In particular, if the chain of remote calls arrives back to a site previously visited, then a new resource is needed. This model arises, for example, in distributed real-time and embedded (DRE) architectures that use the *WaitOnConnection* policy for nested up-calls [12, 10, 16].

The algorithm's ability to provide deadlock avoidance using only operations over local data is made possible by providing the protocol with additional information about resource usage in the form of call graphs that represent all possible sequences of remote invocations. In DRE systems, this information can usually be extracted from the component specifications or from the source code directly by static analysis. In [9, 8] we presented a first version of such a deadlock avoidance algorithm based on an annotated global call graph and showed that the conditions imposed on these annotations were tight: a violation could lead to deadlock. That algorithm, however, did not guarantee liveness: although it guaranteed that some process could always proceed, individual processes could still starve and never terminate, independently of the schedulers' decisions. In this paper we remedy this shortcoming and present a deadlock avoidance algorithm that is still based on local data only and guarantees liveness of all individual processes, assuming a fair scheduler. Interestingly, this new algorithm is also more efficient than the previous one in that it allows more concurrency.

Liveness is guaranteed provided that the scheduler in the local node is fair in the following sense. As observed in [3], the resource assignment is implemented by a *controller* that consists of two components: an *allocation manager* and a *scheduler*. The allocation manager decides whether an incoming request is safe to be granted.

- If the allocation is safe, a unit of resource is assigned.
- It it is not, the process is inserted in a waiting queue.

Upon a resource release, the allocation manager computes the subset of the processes in the waiting queue whose pending request is now safe. It is then the job of the scheduler to pick one process among the set of safe processes. The protocol presented here guarantees liveness globally (every process in the system eventually terminates) assuming that the local schedulers are strongly fair (no process can be in the offered safe set infinitely often without being scheduled).

The rest of this paper is structured as follows. Section 2 introduces the computational model and recalls the basics of our previous deadlock avoidance algorithms [9, 8], and shows that these protocols do not guarantee liveness. Section 3 presents a protocol schema that guarantees deadlock avoidance and liveness. Section 4 shows how this schema can be implemented efficiently. Section 5 concludes.

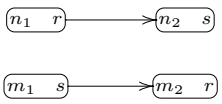
2. MODEL OF COMPUTATION

We model a distributed system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ as a set of *sites* and a *call graph* specification. Sites $\mathcal{R} : \{r_1, \ldots, r_{|\mathcal{R}|}\}$ model distributed devices that perform computations and handle a necessary and scarce local resource, for example a finite pool of threads or execution contexts. A call graph specification $\mathcal{G} : (V, \rightarrow)$ is an acyclic[1] graph that captures all the possible flows of the computations. A call graph node $n = (f : r)$ models the method call f to be performed at site r (if the method name is unimportant we simply write $n : r$). An edge from $n = (f{:}r)$ to $m = (g{:}s)$ denotes a possible remote invocation of method g at site s. If this call is performed the resource (thread or execution context) associated with n is locked at least until g returns. In the remainder of this paper we will 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. Each site r stores some local data, including a constant T_r representing the total units of resource in r, and a variable t_r whose value represents the available resources at each point in time. Initially, $t_r = T_r$.

The execution of a system consists of processes, which can be created dynamically, executing computations that only perform remote calls according to the edges in the call graph. When a new process is spawned it announces the call graph node whose outgoing paths describe the remote calls that the process will perform. All invocations of a call graph node require a new resource in the corresponding site, while call returns release the resource.

We impose no restriction on the topology of the call graph or on the number of process instances, and thus deadlocks can be reached if all requests for resources are immediately granted.

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



If sites s and r each handle exactly two resources ($T_r = T_s = 2$) and four processes are created, two running n_1 and two running m_1, no more resources are available after each process starts its execution. Hence, the resulting state is a deadlock since none of the processes can proceed. □

A deadlock avoidance algorithm implements the allocation manager ensuring that no deadlock can be reached. Our deadlock avoidance algorithms consist of two parts:

1. The offline computation of call-graph *annotations*, $\alpha : V \mapsto \mathbb{N}$, a map from nodes to natural numbers;

2. A runtime *protocol* that controls resource allocations and deallocations based on local data and call graph annotations.

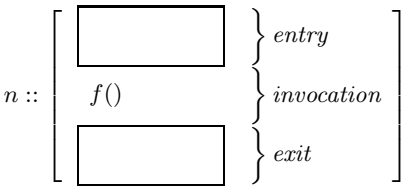
The protocol consists of two stages: one that runs when the resource is requested, and another that executes upon

[1]The restriction to acyclic graphs is relaxed in [7] but this extension is orthogonal to the discussion of this paper and can be incorporated in the liveness protocol.

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

Figure 1: Basic-P

release. A schematic view of a protocol is:



A process that is granted access into the method section is called *active*, while a process whose request is rejected is called *waiting*. We assume that the actions of the entry and exit sections of a protocol cancel each other, and that the successful execution of an entry section cannot help a waiting process in obtaining its desired resources.

Intuitively, the annotation $\alpha(n : r)$ provides a measure of how many execution contexts site r should reserve for processes executing at other sites that may perform remote calls to r with lower annotations. Thus, the annotation provides a static data structure that can be used by a protocol to ensure at runtime that there will be no cyclic dependencies between processes waiting for resources.

The simplest protocol based on this annotation is Basic-P, shown in Figure 1. It grants a resource to a process executing a call graph node $n : r$ if $\alpha(n)$ is less than the number of resources available, represented by the local variable t_r. In previous papers [9][2] we proved that this protocol avoids deadlock if the annotation is acyclic in the following sense. Given a system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ and an annotation α, the annotated call graph $(V, \rightarrow, \dashrightarrow)$ 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 [9]). *Given a system $\mathcal{S}$ and an acyclic annotation, if* Basic-P *is used to control resource allocations then all executions of $\mathcal{S}$ are deadlock free.*

Example 2. Reconsider the system described in Example 1. By granting resources whenever they are available,

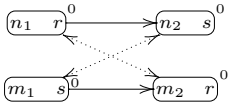
[2]In [9] the entry condition in Basic-P was $\alpha(n) \leq t_r$, because we took 1 as the lowest annotation value. In this paper we take 0 as the lowest annotation value, to simplify the presentation of the liveness protocol given in the next section. With some minor modifications of the proofs the results from [9] all carry over to this modified annotation.

$$n :: \left[\begin{array}{l} \left[\begin{array}{c}\textbf{when } 1 < t_r \textbf{ do}\\ t_r\texttt{--}\end{array}\right] \\ f() \\ \\ t_r\texttt{++} \end{array}\right] \qquad \text{If } \alpha(n) = 0.$$

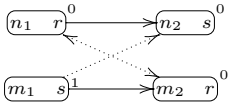
$$n :: \left[\begin{array}{l} \left[\begin{array}{c}\textbf{when } \alpha(n) < p_r \ \wedge \ 1 < t_r \textbf{ do}\\ \langle p_r\texttt{--},\ t_r\texttt{--}\rangle\end{array}\right] \\ f() \\ \\ \langle t_r\texttt{++},\ p_r\texttt{++}\rangle \end{array}\right] \qquad \text{If } \alpha(n) > 0.$$

Figure 2: Efficient-P

it implicitly assumes the following annotation graph,



in which all nodes have annotation 0. Clearly, this graph has a dependency cycle, and thus deadlock avoidance is not guaranteed. If we set $\alpha(m_1) = 1$, this dependency cycle is eliminated, resulting in the acyclic annotated call graph,

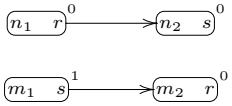


Deadlock is avoided by always reserving at least one resource in site s for a remote call from a process P executing node n_1, so that P can complete. □

The protocol Basic-P, however, unnecessarily restricts concurrency. It can be refined into a more efficient protocol, called Efficient-P, shown in Figure 2. Each site maintains two local variables: t_r, representing the number of currently available resources, and p_r, the number of "potentially available" resources. Initially both p_r and t_r are set to the total number of resources T_r. A process requesting a resource to execute a node with annotation 0 is granted the resource whenever a resource is available, as in Basic-P. Since this execution is guaranteed to terminate independently of other competing processes, this resource is potentially recoverable, and thus p_r is not decremented in this case. When a process requests a resource to execute a node with annotation higher than 0, both t_r and p_r are decremented to ensure that resources are reserved for remote invocations from processes trying to execute nodes with lower annotations. In [9] it was shown that, similar to Basic-P, Efficient-P guarantees deadlock avoidance if the annotation graph has no dependency cycles.

The following example shows that Basic-P does not guarantee liveness.

Example 3. Consider the following system, with resources $T_r = T_s = 2$:



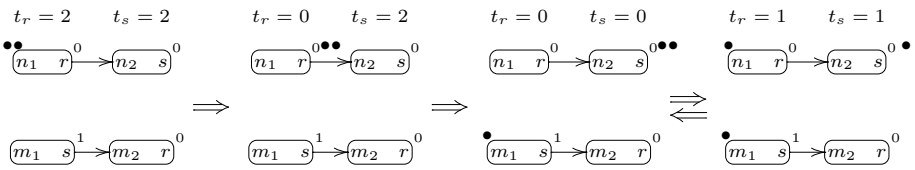


Figure 3: A run of with starvation using Basic-P as allocation protocol.

Figure 3 shows an execution for this system that violates liveness. It begins with two processes that start their execution in n_1. After these processes perform the remote call to n_2 and become active, all the resources are in use. At this point, a new process P is spawned to execute m_1. As $t_s = 0$, it cannot start. Even if one of the processes executing n_2 finishes and releases its resources, P cannot start, because it requires the availability of two resources in s. A new process executing n_1, however, can start, because it only requires one resource in r and then one in s, which are available. Repeating this pattern results in an execution in which $t_s \leq 1$ for all future states, hence the entry condition for P is never enabled, and thus P will wait forever, independent of any scheduler.

Notice that in the scenario in the above example starvation is avoided if we use EFFICIENT-P, assuming a fair scheduler. Unfortunately, in general, EFFICIENT-P does not guarantee liveness either. Indeed, with a call graph specification with annotations of 2 or higher, a similar scenario as that in Example 3 can be constructed for EFFICIENT-P, in which a process will wait forever to execute a node with $\alpha(n) = 2$.

The idea behind EFFICIENT-P, however, can be generalized to obtain a protocol that does guarantee liveness, as we will demonstrate in the rest of the paper.

3. A LIVENESS PROTOCOL

An acyclic annotation of a call graph specification provides, for every node n, a measure of the number of (directly or indirectly) dependent nodes that execute at the same site as n. As we will show, deadlock is avoided if, in every site r and for every annotation value k, there are never more than $T_r - k$ active processes executing nodes with annotation value k or higher. It is the task of the protocol to preserve this property by denying access to processes that would violate this condition. We use ϕ_r to denote this property for site r. We propose a protocol that preserves ϕ_r in all sites by denying access to processes that would violate this condition, ensuring it is a network invariant. Enforcing ϕ_r to be a network invariant implies, for example:

- there can be at most one active process with annotation $T_r - 1$, since $T_r - (T_r - 1) = 1$.
- there can be at most T_r active processes in total, since all of them have annotation at least 0 and $T_r - 0 = T_r$. Every admissible protocol must satisfy this, since no more resources than available can be granted.
- there can be a maximum of $T_r - 1$ active processes with annotation 1 or higher. In other words, processes with annotations higher than 0 cannot exhaust the resources. There is always a resource "reserved" for processes of annotation 0. In EFFICIENT-P this was stated as $t_r \geq 0 \wedge p_r \geq 1$.

It was shown in [9] that both BASIC-P and EFFICIENT-P preserve ϕ_r as an essential step to show that these protocols avoid deadlock. These protocols, however, do not keep track of how many processes are currently active at each annotation level, and therefore, in deciding whether to grant a resource or not, have to assume the worst case, namely that all currently active processes have annotations equal or higher than the requesting process. If in fact some currently active processes have lower annotations, this decision unnecessarily denies access, thereby limiting concurrency and compromising liveness.

In EFFICIENT-P this restriction is partially solved by adding the extra variable p_r, which is used to keep track of the number of active processes with annotation 1 or higher. For systems with maximum annotation 1, EFFICIENT-P does not limit concurrency and, in fact, guarantees liveness.

In the remainder of this section we propose a protocol schema that is a generalization of EFFICIENT-P in that it keeps track of the the number of active processes at each annotation level. This extra information allows it to grant resources that BASIC-P or EFFICIENT-P would have to deny. We prove that any protocol that implements this new protocol schema guarantees absence of deadlock and provides liveness as well. In the next section we propose an efficient implementation of the schema.

3.1 Protocol Schema

The property to be preserved by the protocol in site r is that at each annotation level k there are never more than $T_r - k$ processes executing call graph nodes with annotation value k or higher, where T_r is the total number of threads in site r. To express this property more formally we first introduce some notation. Throughout we assume a fixed site r in which the protocol runs and drop the subscript r.

Let $a[k]$ stand for the number of processes executing nodes with annotation value k that are active in r and let $A[k]$ stand for $\sum_{j \geq k} a[j]$. Let $\phi[k]$ be the property that the number of active processes executing nodes with annotation value k or higher does not exceed $T - k$, that is,

$$\phi[k] \stackrel{\text{def}}{=} A[k] \leq T - k \ .$$

The property the protocol must maintain is

$$\phi \stackrel{\text{def}}{=} \bigwedge_k \phi[k]$$

for k ranging over all annotation levels in the annotated call graph.

Let $A'_i[j]$ and $a'_i[j]$ represent the values of $a[j]$ and $A[j]$

$$n :: \left[\begin{array}{l} \left[\begin{array}{l}\textbf{when } \phi'_i \textbf{ do}\\ \quad a[i]\texttt{++}\end{array}\right] \\ f() \\ \\ a[i]\texttt{--} \end{array}\right] \qquad \text{for } \alpha(n) = i.$$

Figure 4: Live-P

after $a[i]$ is incremented, that is:

$$a'_i[j] \stackrel{\text{def}}{=} \begin{cases} a[j] & \text{if } j > i \\ a[j]+1 & \text{if } j = i \\ a[j] & \text{if } j < i \end{cases} \qquad A'_i[j] \stackrel{\text{def}}{=} \begin{cases} A[j] & \text{if } j > i \\ A[j]+1 & \text{if } j = i \\ A[j]+1 & \text{if } j < i \end{cases}$$

Then the condition that ϕ is preserved if a resource were granted to a process requesting access to a node with annotation i is given by the property ϕ'_i defined by

$$\begin{array}{lcl} \phi'_i[k] & \stackrel{\text{def}}{=} & A'_i[k] \le T - k \\ \phi'_i & \stackrel{\text{def}}{=} & \bigwedge_k \phi'_i[k] \end{array}$$

The protocol schema Live-P can now be given as shown in Figure 4. It is a schema, because the actual implementation of checking ϕ'_i and performing the operations $a[i]$++ and $a[i]$-- is left unspecified. Several implementations are possible, ranging from a brute force implementation using tables to store $a[\cdot]$ and repeated computations of $A[\cdot]$, to more efficient implementations presented in the next section. Any correct implementation of Live-P, however, guarantees absence of deadlock and liveness, as we prove below.

3.2 Deadlock avoidance

To show that Live-P guarantees absence of deadlock we first prove an auxiliary lemma.

Lemma 1. *If ϕ holds and a clause $\phi'_i[j]$ does not hold, then there is at least one active process with annotation j.*

Proof. From the fact that ϕ holds it follows that

$$\begin{array}{lcl} A[j] & \le & T - j \\ A[j+1] & \le & T - (j+1) \;<\; T - j \end{array}$$

From the fact that $\phi'_i[j]$ does not hold, we know

$$A'_i[j] \;=\; A[j] + 1 \;>\; T - j$$

which, with $A[j] \le T - j$, gives

$$A[j] \;=\; T - j$$

and thus, with $A[j+1] < T - j$, we have

$$A[j+1] \;<\; A[j]$$

Since $A[j] = a[j] + A[j+1]$ we have $a[j] > 0$ as desired. □

Corollary 1. *If, for some process with annotation i, ϕ'_i is not satisfied then there is some process with annotation at most i that is active (i.e., $\sum_{j \le i} a[j] \ge 1$).*

Proof. Immediate consequence of Lemma 1, by observing that if ϕ holds, and ϕ'_i does not, there must be some offending clause for some $j \le i$. □

Theorem 2 (Annotation Theorem for Live-P). *Given a system $\mathcal{S}$ and an acyclic annotation, if every site uses* Live-P *to decide allocations then all executions of $\mathcal{S}$ are deadlock-free.*

Proof. We first observe that, in the absence of cyclic dependencies, the relation $\succ$ is a partial order on call graph nodes. By contradiction, suppose that there is a reachable deadlock state. Let P be a process involved in the deadlock, blocked in a node n that is minimal in $\succ$. Let r be the site of n, and i its annotation. We consider the two possible cases:

(1) P is active. In this case a subsequent call to some node m must be blocked, but then m is smaller than n in $\succ$ which contradicts the minimality of n.
(2) P is waiting and ϕ'_i is false. By Corollary 1 there must be an active process executing the method section of some node n_1 with annotation $j \le i$. Since this process is active, it must be blocked in some subsequent call (to some node n_2). Then $n \dashrightarrow n_1 \mapsto^+ n_2$, so $n \succ n_2$ contradicting again the minimality of n.

Therefore, no deadlock is reachable. □

3.3 Liveness

Any implementation of Live-P prevents starvation, provided the local schedulers are fair in the sense that they will always eventually select a process to run if its entry condition is true infinitely often. To prove absence of starvation in the presence of a fair scheduler, it is sufficient to show that every waiting process will eventually be enabled, that is, the entry condition in Live-P will eventually be true. This guarantees that every process progresses and, since the invocations described in the call-graph can be performed at most once, that each process terminates.

Lemma 2. *If $k \ge i$ then ϕ'_k implies ϕ'_i.*

Proof. First, $\phi'_k[j] \equiv \phi'_i[j]$ for all $j \ge k$ and for all $j < i$ since the formulas are syntactically identical. Now, take an arbitrary j within $i \le j < k$. In this case,

$$\begin{array}{lcl} \phi'_k[j] & \equiv & \big(A[j]+1 \;\le\; T - j\big) \\ \phi'_i[j] & \equiv & \big(A[j] \;\le\; T - j\big), \end{array}$$

and if $\phi'_k[j]$ holds, so does $\phi'_i[j]$. □

Corollary 2 (Maximal enabled annotation). *At every instant, there exists $0 \le i \le T$ such that all processes with annotation lower than i are enabled, and all processes with annotation at least i are disabled.*

The protocols Basic-P and Efficient-P also provide a notion of a maximal enabled annotation: t_r and p_r (or 0 if t_r is 0) resp. In general, these are smaller than the one provided by Live-P.

Theorem 3 (Liveness). *Given a system $\mathcal{S}$ and an acyclic annotation, if every site uses* Live-P *to decide allocations then in every run all waiting processes are eventually enabled.*

Proof. By contradiction, consider a run with some starving process and let P starve in some node n that is minimal in $\succ$ among all nodes with starving processes. Let r be the site where n resides and i its annotation. After some prefix of the run, P will be continuously disabled. We call every such subsequent state an "offending state."

In every offending state, the formula ϕ'_i is not satisfied in site r, so there must be some $j \leq i$ for which $\phi'_i[j]$ does not hold, i.e.,

$$A'_i[j] \not\leq T - j.$$

Given an offending state, let j be the largest annotation that causes a violation to ϕ'_i, which by Lemma 1 satisfies $a[j] \geq 1$. We call the pair $(j, a[j])$ the *characteristic* of P at that state. Note that if $(j, a[j])$ is the characteristic of P, then all waiting processes of annotations $k \geq j$ must also be disabled, by Lemma 2.

Let σ be an offending state with minimum characteristic, when compared according the lexicographic order. Since the set of characteristics is finite and totally ordered, there is one such state. Now we consider the two possibilities:

(1) no process Q with annotation j that is active in σ terminates. Since Q is active, it must be performing a remote invocation that does not terminate. Therefore, some of Q nested invocations must be starving in a node m with $n \dashrightarrow\!\!\!\!\mapsto^{+} m$, which contradicts the minimality of n in $\succ$.

(2) some active process Q with annotation j terminates. In this case, since P is continuously disabled by assumption, all waiting processes with annotation j or higher are also blocked, by Lemma 2. Then, when Q terminates the value of the pair $(j, a[j])$ decreases. Either P is then enabled, or its new characteristic has a smaller j or it is $(j, a[j] - 1)$. This holds since after Q has released its resource, no process with annotation j or higher can be granted a resource, unless P becomes enabled too. This contradicts that σ is a state with minimal characteristic.

Consequently, P will be eventually enabled. □

4. IMPLEMENTATION

In this section we study how to implement efficiently a controller that guarantees liveness of every process. In Section 4.1 and 4.2 we study how to implement LIVE-P. In Section 4.3 we sketch an implementation of a strongly fair scheduler. Finally, in Section 4.4 we describe how to integrate LIVE-P as an allocation manager, together with a strongly fair scheduler to build a controller that guarantees liveness.

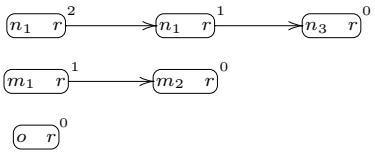
4.1 A Tempting (but Incorrect) Implementation

One tempting implementation of LIVE-P would only check the clause that corresponds to the annotation of the requesting process, called the BAD-P protocol (for a node n residing in site r with annotation $\alpha(n) = i$):

$$n :: \left[\begin{array}{l} \left[\begin{array}{c} \textbf{when } 1 \leq t_r \ \wedge \ A[i] < (T_r - i) \textbf{ do} \\ t_r\texttt{--} \ ; \ a[i]\texttt{++} \end{array} \right] \\ f() \\ \\ t_r\texttt{++} \ ; \ a[i]\texttt{--} \end{array} \right]$$

Unfortunately, the protocol BAD-P is not correct in the sense that it compromises deadlock freedom.

Example 4. Consider a scenario with $T_r = 3$ and the following call-graph:



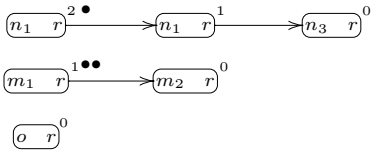
Let a sequence of allocation requests in site r be 0, 1, 1, $\overline{0}$, 2 where $\overline{0}$ indicates a resource release. The following table shows the values of $a[\cdot]$ after each allocation or deallocation is performed

Active	$a[0]$	$a[1]$	$a[2]$	$A[0]$	$A[1]$	$A[2]$
{}	0	0	0	0	0	0
0	1	0	0	1	0	0
0, 1	1	1	0	2	1	0
0, 1, 1	1	2	0	3	2	0
1, 1	0	2	0	2	2	0
1, 1, 2	0	2	1	3	3	1

All requests satisfy the conditions of the entry section of protocol BAD-P, so they are immediately granted. The last row, though, corresponds to a state that does not satisfy the invariant clause $\phi[1]$:

$$A[1] = 3 \qquad \not\leq \qquad T - 1 = 3 - 1 = 2.$$

This illegal state was reached after a process with annotation 2 requested a resource, but granting this request causes a violation of $\phi[1]$ but not a violation for $\phi[2]$. Even more, all previous requests for annotation 1 were granted rightfully. The illegal state is depicted:



At this state all resources are used, $t_r = 0$ continuously, all processes incur in a deadlock, and none will ever terminate. □

4.2 An Efficient and Correct Implementation

We describe here an efficient implementation of LIVE-P. The key idea is to use a data-structure, called *active tree*, that stores the number of active processes for each annotation (denoted as $a[\cdot]$ above) with efficient operations of:

(1) inserting a process,
(2) removing a process, and
(3) obtaining the highest annotation of a processes that can become active without violating ϕ. By Corollary 2 this value is unique.

We describe here how to implement this data-structure using a binary search tree with annotation as key, and where each node also stores (in a field named *count*) the number of active processes with that annotation. This data-structure can be maintained:

- in $O(T)$ space and $O(log\ T)$ time per insertion and removal using a complete binary tree, or
- in $O(d)$ space and $O(log\ d)$ time per insertion and removal using a balanced tree (for example a Red-Black tree), where d is the number of different annotations among all active processes (this parameter is called the *diversity load*).

When a processes with annotation i is granted access, if a node with key i exists in the tree, its *count* field is incremented, otherwise a new node with key i is added to the tree with *count* 1.

In order to obtain an efficient calculation of the maximum legal annotation, the search tree is augmented with extra information in each node, based on the following observation. If the active processes were linearly ordered according to their annotation, a violation of ϕ would be witnessed by a process with annotation i located further than $T-i$ positions to the end of the list. Similarly, the value of the minimum illegal annotation corresponds to the process with smallest i that is precisely $T-i$ positions to the end of the list. We maintain enough information in each node to retrieve the smallest such offending annotation in time proportional in the height of the tree. In the following description we use $tree(x)$ to denote the (sub)-tree rooted at node x, and $left(x)$ and $right(x)$ for the left and right subtrees resp. If *foo* is a field, the instance of *foo* at node x is represented by $x.foo$. Each node in the tree stores:

1. *key*: the annotation of the processes that the node describes.
2. *count*: the number of active processes with that annotation.
3. *size*: the total number of processes in $tree(x)$, including all the $x.count$.
4. *larger*: the maximum number of processes with annotations larger than the largest key in $tree(x)$, that could be added (or that exist in the super-tree containing $tree(x)$) without causing a ϕ violation in any of the nodes in $tree(x)$.
5. *larger_me*: the maximum number of processes with annotations larger than the largest key in $tree(x)$, that could be added (or that exist in the super-tree containing $tree(x)$) without causing a ϕ violation in x itself.
6. *larger_left*: the maximum number of processes with annotations larger than the largest key in $tree(x)$, that could be added (or that exist in the super-tree containing $tree(x)$) without causing a ϕ violation in $left(x)$. Note that $x.count$ and all the processes described in $right(x)$ are already present and higher than any annotation in $left(x)$.
7. *larger_right*: the maximum number of processes with annotation larger or equal than the largest key in $tree(x)$, that could be added (or that exist in the super-tree containing $tree(x)$) without causing a ϕ violation in $right(x)$.

It is well-known (see, for example [2], Theorem 15.1) that an augmented Red-Black tree can be maintained, with the regular operations of insertion and removal still in $O(log\ n)$, if all fields can be computed from simpler fields of the node and all the fields of the children nodes. This augmentation result obviously holds for complete binary trees as well. Our augmentations satisfy this property, since:

1. *key* and *count* are primitive fields, not depending on other fields in any node in the tree.
2. *size* can be computed from the values of the keys of the children nodes:
$$x.size = left(x).size + right(x).size + x.count.$$
3. *larger* is just the minimum of the other three augmentation fields:
$$x.larger = min(x.larger\_me, x.larger\_left, x.larger\_right).$$
4. *larger_me* can be computed using
$$x.larger\_me = T - x.key - (x.count + right(x).size).$$
This is because if there are $x.larger\_me + right(x).size$ active processes with annotation higher than $x.key$, then the total number of processes with annotation $x.key$ or higher is
$$A[x.key] = x.count + x.larger\_me + right(x).size,$$
and then $A[x.key]$ would be $T - x.key$. This is the largest value allowed by ϕ.
5. *larger_right* is directly the largest value of the right sub-tree:
$$x.larger\_right = right(x).larger.$$
6. Finally, *larger_left* can be computed by subtracting the size of the right subtree and the root node from the minimum of the values of the left child:
$$x.larger\_left = left(x).larger - (right(x).size + x.count).$$

In all the definitions above, if the left (resp. right) subtrees are missing, then $left(x).size$ is 0, and $left(x).larger = \infty$. A tree stores a legal configuration of active processes if the value of *root.larger* is non-negative. Finally, the following program can be used to calculate the maximum value of a legal insertion:

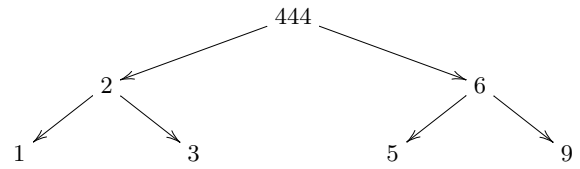
```
1: CalcMax(x, extra)
2: if (x.larger_left − extra = 0) then
3:    return CalcMax(left(x), extra+right(x).size+x.count)
4: else if (x.larger_me − extra = 0) then
5:    return x.key − 1
6: else if (x.larger_right − extra = 0) then
7:    return CalcMax(right(x), extra)
8: else
9:    return T − 1
10: end if
```

The initial call is $\textsc{CalcMax}(root, 0)$. The algorithm traverses the tree seeking for the leftmost occurrence of a node x satisfying the following condition:

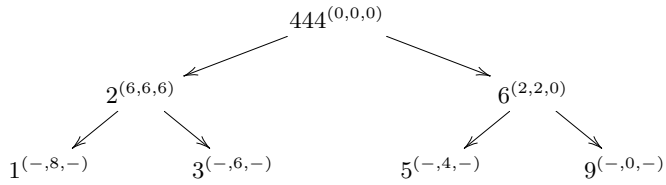
$$(x.larger\_me - extra) = 0 \qquad (1)$$

Since the parameter *extra* passes the number of nodes actually larger than x in the whole tree, condition (1) captures precisely whether a new insertion of a value larger or equal $x.key$ would cause a $\phi[x.key]$ violation. This search can be clearly performed in a number of steps proportional to the height of the tree, which gives a complexity of $O(log\ d)$ where d is the size of the tree (the diversity load) with the use of balanced trees, and $O(log\ T)$ with the complete tree.

Example 5. Consider a site with $T = 10$ resources, and the following tree, which is a possible active tree representing the set of active processes $\{1, 2, 3, 4, 4, 4, 5, 6, 9\}$:



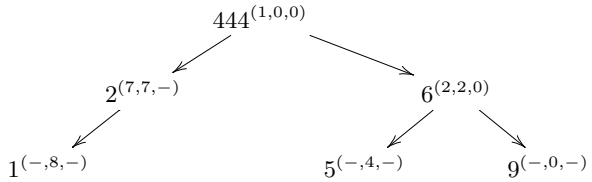
The values of (*larger_left*, *larger_me*, *larger_right*) are:



CalcMax($root, 0$) returns 0, after performing the sequence of calls:

$$\begin{aligned}\textsc{CalcMax}(444^{(0,0,0)}, 0) &\mapsto \textsc{CalcMax}(2^{(6,6,6)}, 6)\\ &\mapsto \textsc{CalcMax}(1^{(-,8,-)}, 8)\\ &\mapsto 1 - 1 = 1\end{aligned}$$

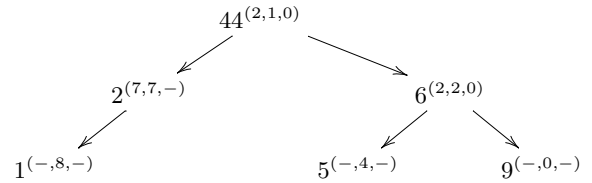
The maximum annotation of a process with an enabled entry section is 0 since it can be legally inserted, and any insertion of 1 or higher would cause a violation in the node $1^{(-,8,-)}$. Suppose that the process with annotation 3 releases its resource, and that the resulting tree is:



In this case the maximum legal annotation is 3 since:

$$\textsc{CalcMax}(444^{(1,0,0)}, 0) \mapsto 4 - 1 = 3$$

Finally, if one of the processes with annotation 4 releases its resource, the resulting tree is:



The maximum annotation is 8 as indicated by:

$$\begin{aligned}\textsc{CalcMax}(44^{(2,1,0)}, 0) &\mapsto \textsc{CalcMax}(6^{(2,2,0)}, 0)\\ &\mapsto \textsc{CalcMax}(9^{(-,0,-)}, 0)\\ &\mapsto 9 - 1 = 8\end{aligned}$$

□

The asymptotic running time of the three methods presented to implement Live-P are summarized in the table:

data-structure	*time*	*space*
Array	$O(T)$	$O(T)$
CompleteBinaryTree	$O(log\ T)$	$O(T)$
Red-Black Tree	$O(log\ d)$	$O(d)$

Our experimental simulations reveal that the simplest array implementation is the best choice only for small resource sets. The Red-Black tree is only the preferred choice when memory is heavily constrained, or when the resource set managed is large but the load is not.

4.3 Implementation of a Fair Scheduler

We sketch how to implement a fair scheduler based on an *oldest process first* policy. An earliest deadline first policy could similarly be used. The implementation is based on a data structure, called *waiting tree*, that can perform three operations:
(1) insert a process,
(2) remove a process,
(3) obtain the oldest process with a certain annotation or smaller.

Similarly to the discussion of the previous section, an efficient waiting tree can be implemented using a binary search tree, with annotation as key but this time including a priority queue to store the waiting processes with the node's annotation. Each node is also augmented with the oldest process in the left and right subtrees. These augmentations only depend on the values of the corresponding children nodes, so its maintenance is efficient. Using a Red-Black tree, this data-type can be maintained in $O(log\ w + log\ m)$, where w is the number of different annotations with some waiting process, and m is the maximum size of any priority queue (maximum number of waiting processes for the worst annotation). If a complete tree is used then a running time of $O(log\ T + log\ m)$ is obtained.

4.4 Implementation of the Controller

Finally, the controller can be built by combining the waiting tree that implements the scheduler, and the active tree that implements deadlock avoidance algorithm as follows:

- **allocation request**: check whether the annotation of the requesting process is at most CalcMax($root, 0$).
  - If the check succeeds, grant the resource and insert the process in the active tree.
  - If the check fails, insert the process in the waiting tree.
- **resource release**: remove the process from the active tree, and recalculate $N =$ CalcMax($root, 0$). Obtain the oldest process P with annotation N or smaller from the waiting tree (if any); extract it, and perform an allocation request. This allocation is guaranteed to be successful.

5. CONCLUSIONS

We have presented an efficient distributed deadlock avoidance mechanism that guarantees liveness. The construction is possible under the assumption that all possible call graphs are known a priori, and that processes announce the call graph they are going to execute when they are created. These are reasonable assumptions in distributed real-time and embedded systems. We have proved the correctness of the protocol and presented the different trade-offs to implement it in practice.

Our distributed deadlock-avoidance liveness protocol can

serve as a basis for several new developments. Ongoing and future research include a distributed priority inheritance protocol that serves as a mathematically sound basis to deal with priority inversions in DREs, an important problem as indicated in [11].

In this paper we have assumed that every site models a type of resource (a different thread pool) and is placed in a separate distributed node. In practice, different sites can be mapped into the same processor, so there is a potential optimization using shared memory. While in the extreme case a purely centralized controller can be synthesized (see [3]) if all sites are mapped to the same processor—without the need of annotations—it is worth investigating mixed approaches for partially distributed placement.

We have shown that if all local schedulers are strongly fair then the controller obtained using an implementation of our deadlock avoidance algorithm guarantees liveness globally. However, it will be interesting to investigate what is the effect of local scheduling policies in the global scheduling goals.

6. REFERENCES

[1] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[3] Luca de Alfaro, Vishwanath Raman, Marco Faella, and Rupak Majumdar. Code aware resource management. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT'05)*, pages 191–202. ACM Press, 2005.

[4] Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[5] Arie N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12:373–377, 1969.

[6] James W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.

[7] César Sánchez, Henny B. Sipam, and Zohar Manna. On efficient deadlock avoidance for distributive recursive processes. Submitted for publication.

[8] 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 *Proc. of the 20th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE Computer Society Press, 2006.

[9] 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. Springer-Verlag, October 2005.

[10] Douglas C. Schmidt. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Communications of the ACM Special Issue on CORBA*, 41(10), October 1998.

[11] 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, June 1998.

[12] 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.

[13] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., Sixth edition, 2003.

[14] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. McGraw-Hill, Inc., 1994.

[15] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, Third edition, 1998.

[16] Venkita Subramonian, Guoliang Xing, Christopher D. Gill, Chenyang Lu, and Ron Cytron. Middleware specialization for memory-constrained networked embedded systems. In *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*. IEEE Computer Society Press, May 2004.