Reachable State Spaces of Distributed Deadlock Avoidance Protocols

CÉSAR SÁNCHEZ and HENNY B. SIPMA
Stanford University

We present a family of efficient distributed deadlock avoidance algorithms with applications to distributed real-time and embedded systems (DREs), which subsumes previously known solutions as special instances. Then we use this family to study the reachable set of states and the allocation sequences allowed by the different protocols. This result enables a new proof method for showing freedom from deadlock of variations of deadlock avoidance protocols.

Distributed deadlock avoidance 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. The most basic algorithm Basic-P, performs only one operation per request using one counter per site. The protocol Live-P, which also guarantees liveness globally, needs to performs a logarithmic number of operations per request, using a linear number of counters in each site (in terms of the total ammount of resources). The family of algorithms that we introduce here, called k-Efficient-P, subsumes these as special cases. This family allows us to compare the protocols according to the set of reachable states and their legal allocation sequences. We prove that, even though the more liberal algorithms allow more concurrency, all algorithms can reach the same set of states. We capture this set of states as a network invariant.

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

Additional Key Words and Phrases: 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, for example 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 dead-

Author's address: César Sánchez, Computer Science, Stanford University, Stanford, CA 94305.
e-mail: cesar@Theory.Stanford.EDU

This is a Technical Report. Permission is granted to make digital/hard copy of all or part of this material without fee for personal or classroom use, provided that the copies are not made or distributed for profit or commercial advantage, the copyright/server notice, the title of the publication, and its date appear.
© 2006 The Authors

lock. For example, imposing a fixed order in which resources are acquired, such as in "monotone locking" [Birrell 1989], 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 test feasable, 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 [Dijkstra 1965; Habermann 1969; Havender 1968; Stallings 1998; Silberschatz et al. 2003], where a process upon creation reports the maximum number of resources of each type that it can request during its execution. This information is used to decide whether to grant resource requests. 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 [Singhal and Shivaratri 1994]; the communication costs involved simply outweigh the benefits gained from deadlock avoidance over deadlock prevention.

In this paper we study 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 [Schmidt 1998; Schmidt et al. 2000; Subramonian et al. 2004].

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 [Sánchez et al. 2005; Sánchez et al. 2006; Sánchez et al. 2006] 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.

The rest of this document is structured as follows. Section 2 introduces the computational model and the necessary notation and shows why deadlocks can be reachable if no allocation manager is used. Section 3 presents a network invariant that deadlock avoidance algorithms maintain, restates the previously known pro-

tocols in terms of this invariants. This generalization allows the definition of the family of protocols k-EFFICIENT-P. Sections 4 and 5 compare the different algorithms according the global sequences of legal allocations they allow, and the set of global states they can reach. Finally, Section 6 explains a new proof technique for correctness of deadlock avoidance schemas derived from the results in this paper, sketches some open problems and the conclusions.

2. COMPUTATIONAL MODEL

To study deadlock avoidance protocols we view a distributed system as a set of sites that perform computations and a set of call graphs that describe how and at which site these computations are to be performed. The resources we consider are the threads. In classical operating systems terminology, the resources are guarded by counting semaphores. The call graphs provide a static representation of all possible resource usage patterns.

Formally, a distributed system is modeled by a tuple $\mathcal{S}: \langle \mathcal{R}, \mathcal{G} \rangle$ consisting of

—$\mathcal{R}: \{r_1, \ldots, r_{|\mathcal{R}|}\}$, a set of sites, and

—$\mathcal{G}: \{G_1, \ldots, G_{|\mathcal{G}|}\}$, a set of distinct call graphs.

Each call graph G_i is a finite tree $\langle V_i, \rightarrow_i \rangle$ where each node $n{:}r$ represents a method that runs in site r. We also say that node n *resides* in site r. If two nodes reside in the same site we write $n \equiv_{\mathcal{R}} m$. An edge from $n{:}r$ to $m{:}s$ denotes that method n, in the course of its execution may invoke method m in site s. We assume that every pair of call graphs is disjoint: $V_i \cap V_j = \varnothing$ and $\rightarrow_i \cap \rightarrow_j = \varnothing$ whenever $i \neq j$. We also use V for $\bigcup_i V_i$ and $\rightarrow$ for $\bigcup_i \rightarrow_i$, and call $(V, \rightarrow)$ *the* call graph of system $\mathcal{S}$.

We assume that each site has a fixed number of pre-allocated resources. Although in many modern operating systems threads can be spawned dynamically, it is common in real-time systems to bound resources. Many DRE systems pre-allocate a fixed sets of threads to avoid the relatively large and variable cost of thread creation and initialization. Each site r stores a set of local variables V_r that includes the constant $T_r \geq 1$ denoting the number of resources present in r, and a variable t_r that represents the number of available resources. 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 remove calls that the process will perform. All invocations to 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 2.1. *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 $(n_1\ r) \rightarrow (n_2\ s)$ and $(m_1\ s) \rightarrow (m_2\ r)$. If sites s and r each handle exactly one resource ($T_r = T_s = 1$) and two processes are created, one in n_1 and one in m_1, no more resources are available after each process*

starts its execution. Hence a deadlock is reached, since none of the processes can proceed. □

Our deadlock avoidance solution consists of two parts: (1) the offline computation of call-graph *annotations*, maps from call-graph nodes to natural numbers; and (2) a run time *protocol* that controls resource allocations and deallocations based on call-graph annotations. Informally, annotations measure the amount of resources that the computation of a node requires. The protocols grant a request based on the remaining local resources (and possibly other local variables) and the annotation of the requesting node.

2.1 Protocol

A *protocol* for controlling the resource allocation in node $n{:}r$ is implemented by a program executed in r before and after method n is dispatched. This code can be different for two call-graph nodes even if they reside in the same site. The schematic structure of a protocol for a node $n{:}r$ is:

$$
n :: \left[\begin{array}{lll}
\ell_0: & \boxed{\begin{array}{l}\textbf{when } En_n(V_r) \textbf{ do} \\ \quad In_n(V_r, V_r')\end{array}} & \left.\begin{array}{l}\\ \\ \end{array}\right\} \textit{entry section} \\
\ell_1: & f() & \left.\begin{array}{l}\\ \end{array}\right\} \textit{method invocation} \\
\ell_2: & \boxed{Out_n(V_r, V_r')} & \left.\begin{array}{l}\\ \end{array}\right\} \textit{exit section} \\
\ell_3: & &
\end{array}\right]
$$

The entry and exit sections implement the resource allocation policy. Upon invocation, the entry section checks resource availability by inspecting local variables V_r of site r. If the predicate $En_n(V_r)$, called the enabling condition, is satisfied we say that the entry section is *enabled*. In this case, the request can be granted and the resource assigned to the process updating the local variables according to the relation $In_n(V_r, V_r')$, where V_r' stands for the local variables after the action is taken. We assume that the entry section is executed atomically, as a *test-and-set*. The method invocation section executes the code of the method, here represented as $f()$, which may include remote invocations according to the edges outgoing from node n in the call graph. The method invocation terminates after all its descendants in the call graph have terminated and returned. The exit section releases the resource and may update some local variables in site r, according to the relation $Out_n(V_r, V_r')$. In_n is called the entry action and Out_n is called the exit action

2.2 Annotations

Given a system $\mathcal{S}$ and annotation α, the annotated call-graph $(V, \rightarrow, \dashrightarrow)$ is obtained from the call graph $(V, \rightarrow)$ by adding one edge $n \dashrightarrow m$ between any two nodes that reside in the same site with annotation $\alpha(n) \geq \alpha(m)$. A node n "depends" on a node m, which we represent $n \succ m$, 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.

3. A FAMILY OF LOCAL PROTOCOLS

We study protocols that prevent deadlock in all scenarios, trying to utilize the resources as much as possible, while requiring no communication between the sites. These are called local protocols. We say that a protocol is *completely local* if all the enabling conditions are determined by:

(1) the annotation of the requested node, and

(2) the number of active processes in the local site and their corresponding annotations.

For completely local protocols, the effect of the input (resp. output) action on future enabling conditions is determined by the effect of adding (resp. removing) an entry at the corresponding annotation from the set of active processes. All the deadlock avoidance protocols suggested to date are completely local. We use the following notation to for active processes: $a_r[k]$ stands for the number of active processes running in site r in a node with annotation k. We also use $A_r[k]$ as a short for $\sum_{j \geq k} a_r[j]$, that is, $A_r[k]$ stands for the number of active processes running in site r executing nodes with annotation k or higher.

The protocol Basic-P was introduced in [Sánchez et al. 2005]. An allocation attempt at node $n{:}r$ with annotation $\alpha(n) = i$, is controlled by:

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

Before assigning a resource, Basic-P checks whether the provision of resources is large enough, as indicated by the annotation i. This check ensures that processes (local or remote) that could potentially be blocked if the requested resource is granted, have actually enough resources to complete. The counter t_r keeps track of the remaining resources in site r. The correctness of Basic-P is based on the acyclicity of the annotations:

Theorem 3.1 (Annotation Theorem for Basic-P [Sánchez et al. 2005]). *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.*

The protocol Basic-P can be improved observing that processes requesting resources in nodes with annotation 1 can always terminate, in spite of any other process in the system. This observation leads to Efficient-P, which uses two counters: t_r, as before; and p_r, to keep track of the "potentially recoverable" resources, which include not only the available resources but also the resources granted to processes in nodes with annotation 1. The protocol Efficient-P that controls

the allocation at a node $n{:}r$ with annotation i is:

$$n :: \left[\begin{array}{l} \left[\begin{array}{c}\textbf{when } 1 \le t_r \textbf{ do}\\ t_r\texttt{--}\end{array}\right] \\ f() \\ t_r\texttt{++}\end{array}\right] \qquad n :: \left[\begin{array}{l} \left[\begin{array}{c}\textbf{when } i \le p_r \ \wedge\ 1 \le t_r \textbf{ do}\\ \langle p_r\texttt{--},\ t_r\texttt{--}\rangle\end{array}\right] \\ f() \\ \langle t_r\texttt{++},\ p_r\texttt{++}\rangle\end{array}\right]$$

If $i = 1$ If $i > 1$

A version of the Annotation Theorem also exists for EFFICIENT-P establishing that in the absence of dependency cycles, EFFICIENT-P can reach no deadlocks. These proofs of correctness are based on properties of the sets of active processes in the system, in states reachable by the protocols. The proof of the Annotation Theorem for BASIC-P and EFFICIENT-P relies on showing that the following invariant φ is maintained:

$$\varphi \stackrel{\text{def}}{=} \bigwedge_{r\in\mathcal{R}} \bigwedge_{k\le T_r} A_r[k] \le T_r - (k-1).$$

We use φ_r as an alias for $\bigwedge_{k\le T_r} A_r[k] \le T_r - (k-1)$, and $\varphi_r[k]$ for each of its clauses. The fact that φ is an invariant is used to show, for example, that if a request for annotation i is denied, then there must be an active process running in a node with annotation i or lower.

A legitimate question is whether the most liberal protocol, that only guarantees that φ is an invariant, still provides absence of deadlocks in the presence of an acyclic annotation. The answer (see [Sánchez et al. 2006]) is that this protocol, called LIVE-P, guarantees not only absence of deadlock but also liveness under some simple fairness assumptions about the local schedulers. LIVE-P simply checks whether granting a request at node $n{:}r$ with annotation i keeps the system in an φ state. LIVE-P for a node $n{:}r$ with annotation i is:

$$n :: \left[\begin{array}{l} \left[\begin{array}{c}\textbf{when } \varphi_r^{(i)} \textbf{ do}\\ a_r[i]\texttt{++}\end{array}\right] \\ f() \\ \\ a_r[i]\texttt{--} \\ \end{array}\right]$$

Here, the formula $\varphi_r^{(i)}$ stands for the formula that results from φ_r when $a_r[i]$ is incremented. That is, $\varphi_r^{(i)}$ is the weakest precondition for φ under the transition that assigns the resource. The formula $\varphi_r^{(i)}$ can easily be stated:

$$\varphi_r^{(i)} \stackrel{\text{def}}{=} \left(\begin{array}{ll} \bigwedge_{k>i} A_r[k] & \le\ T_r - (k-1) \\ & \wedge \\ \bigwedge_{k\le i} A_r[k]+1 & \le\ T_r - (k-1) \end{array}\right).$$

Again, we use $\varphi_r^{(i)}[j]$ for the clause of $\varphi_r^{(i)}$ that corresponds to annotation j. Observe that $\varphi_r^{(i)}[j]$ is equivalent to $\varphi_r[j]$ for $j > i$ since the expressions are identical.

Moreover, for $j \leq i$, the condition is strengthened: $\varphi_r^{(i)}[j] \rightarrow \varphi_r[j]$.

In order to compare the protocols, we use the notation introduced for LIVE-P to restate the definition of the protocols BASIC-P and EFFICIENT-P. The following is an invariant of BASIC-P:

$$T_r = t_r + A_r[1],$$

since $A_r[1]$ corresponds to the total number of active processes in r. The enabling condition of BASIC-P for a node $n{:}r$ with annotation i can be restated as:

$$A_r[1] \leq T_r - (i-1)$$

As we will see, this is an strengthening of $\varphi_r^{(i)}$. We define the k-th strengthening formula for a request in node $n{:}r$ with annotation i as:

$$\psi_r^{(i)}[k] \stackrel{\text{def}}{=} A_r[k] \leq T_r - (i-1)$$

It is easy to see that the following holds:

$$\psi_r^{(i)}[k] \rightarrow \varphi_r^{(i)}[j] \qquad \text{for all } k \leq j \leq i, \tag{1}$$

Therefore,

$$\psi_r^{(i)}[k] \rightarrow \bigwedge_{k \leq j \leq i} \varphi_r^{(i)}[j]. \tag{2}$$

Now, $\varphi_r^{(i)}[j]$ holds vacuously for all $i \geq j$, since the formulas for $\varphi_r^{(i)}[j]$ and $\varphi_r[j]$ are identical in this case. Hence:

$$\psi_r^{(i)}[k] \rightarrow \bigwedge_{k \leq j} \varphi_r^{(i)}[j]. \tag{3}$$

Finally, if $\varphi_r^{(i)}[j]$ is satisfied for all values at most k, and $\psi_r^{(i)}[k]$ is ensured, $\varphi_r^{(i)}$ can be concluded:

$$\begin{aligned}
(\bigwedge_{j<k} \varphi_r^{(i)}[j]) \wedge \psi_r^{(i)}[k] \;&\rightarrow\; (\bigwedge_{j<k} \varphi_r^{(i)}[j]) \wedge (\bigwedge_{j \geq k} \varphi_r^{(i)}[j]) \\
&\leftrightarrow\; \bigwedge \varphi_r^{(i)}[j] \\
&\leftrightarrow\; \varphi_r^{(i)}.
\end{aligned}$$

Therefore, if a protocol ensures $\bigwedge_{j<k} \varphi_r^{(i)}[j] \wedge \psi_r^{(i)}[k]$ for some k, then φ is an invariant.

In general, the lower the value of the strengthening point k, the less computation is needed to compute the predicate (the number of comparison are reduced) but the less liberal the enabling condition becomes. In the case of $k = 1$ the strengthening

is $\psi_r^{(i)}[1] \rightarrow \varphi_r^{(i)}$, and the following protocol, equivalent to BASIC-P, is obtained:

$$n :: \left[\begin{array}{l} \left[\begin{array}{c} \textbf{when } \psi_r^{(i)}[1] \textbf{ do} \\ a_r[i]\texttt{++} \end{array} \right] \\ f() \\ \\ a_r[i]\texttt{--} \end{array} \right]$$

It must be noted that this protocol is *logically* equivalent to BASIC-P: the result of the enabling condition, and the effect of the input and output actions on future tests are equivalent. The direct implementation of BASIC-P introduced earlier in this section uses a single counter t_r, while in this restated version, several counters are apparently needed: $a_r[i]$ and $A_r[1]$. However, the effect of the increments and decrements of $a_r[i]$ on $A_r[1]$ are the same, independently of i. Therefore, these actions can be implemented as $A_r[1]\texttt{++}$ and $A_r[1]\texttt{--}$ respectively.

With a strengthening point of $k = 2$ we obtain a protocol equivalent to EFFICIENT-P:

$$n :: \left[\begin{array}{l} \left[\begin{array}{c} \textbf{when } \varphi_r^{(i)}[1] \wedge \psi_r^{(i)}[2] \textbf{ do} \\ a_r[i]\texttt{++} \end{array} \right] \\ f() \\ \\ a_r[i]\texttt{--} \end{array} \right]$$

In general, if an arbitrary k is picked as the strengthening point, the following protocol, that we call k-EFFICIENT-P is obtained:

$$n :: \left[\begin{array}{l} \left[\begin{array}{c} \textbf{when } \left(\bigwedge_{j<k} \varphi_r^{(i)}[j] \right) \wedge \psi_r^{(i)}[k] \textbf{ do} \\ a_r[i]\texttt{++} \end{array} \right] \\ f() \\ \\ a_r[i]\texttt{--} \end{array} \right]$$

In the extreme, if the strengthening point is higher than the number of resources T_r, then LIVE-P is obtained:

$$n :: \left[\begin{array}{l} \left[\begin{array}{c} \textbf{when } \left(\bigwedge \varphi_r^{(i)}[j] \right) \textbf{ do} \\ a_r[i]\texttt{++} \end{array} \right] \\ f() \\ \\ a_r[i]\texttt{--} \end{array} \right]$$

The protocol k-EFFICIENT-P can be directly implemented using k counters, obtaining a running time of $O(k)$ and needing a storage space of $O(k\ log\ T_r)$ bits. Using an *active tree* data-structure, (see [Sánchez et al. 2006]), k-EFFICIENT-P can be implemented in $O(log\ k)$ running time per allocation and still needing $O(k\ log\ T_r)$ bits in each site.

We remark here that the *In* and *Out* actions of all these protocols (BASIC-P, EFFICIENT-P, k-EFFICIENT-P and LIVE-P) are identical. Since this protocols are completely local, the effects that they have in the successor states are the same as with the standard presentations using the extra variables t_r, p_r, etc.

4. ALLOCATION SEQUENCES

Here we establish the basic foundations to compare the reachable states and the possible allocation sequences allowed by a deadlock avoidance protocol. We define a language whose strings correspond to sequences of (global) allocations and deallocations. Then we define the language that a protocol accepts as the legal sequences allowed by the protocol.

Given a call graph $\mathcal{G} : (V, \rightarrow)$ we define the alphabet $\Sigma = V \cup \overline{V}$ where $\overline{V}$ contains one element $\overline{n}$ for every node n in V. We assume that V and $\overline{V}$ are disjoint, $V \cap \overline{V} = \varnothing$. We refer to the elements of V as allocation symbols, to the elements of $\overline{V}$ as deallocation symbols and the elements of Σ simply as symbols. Given a string s in Σ^* and a symbol v, we use s_v to denote the number of occurrences of v in s, and $|s|_n$ as a short for $s_n - s_{\overline{n}}$.

DEFINITION 4.1 (WELL-FORMED ALLOCATION STRINGS). *A string s in Σ^* is called well-formed if for every prefix p of s, and for every allocation symbol n,*

$$p_n \geq p_{\overline{n}},$$

or, equivalently, $|p|_n \geq 0$.

An *admissible* allocation sequence is one that corresponds to a possible execution of the system, which not only requires the string to be well-formed but it also requires that every allocation of a non-root node is preceded by a corresponding annotation in its parent node. Essentially, a process must be executing in the calling node for the actual call to exist. Different semantics for remote calls are possible, and each is captured by a different language restriction. For example, *asynchronous* calls allow a process to continue running while a remote call is being executed, while *synchronous* calls block the caller until the call returns. *Multiple* call semantics allow a remote call—expressed by a $\rightarrow$ edge—to be repeated indefinitely, while *exactly-once* semantics establish that all descendants must be invoked, and using *at-most-once* a call can either be ignored or produced once[1]. We describe here asynchronous, at-most-once calls. Many of these results can also be obtained for other call semantics.

DEFINITION 4.2 (ADMISSIBLE STRINGS). *A well-formed allocation string s is called admissible if for every prefix p of s, and every remote call $n \rightarrow m$:*

$$|p|_n \geq |p|_m.$$

[1]The only restriction all semantics must satisfy is that all processes must terminate if scheduled infinitely often.

Admissible strings ensure that the number of children processes (callees) is not more than the number of parents (caller processes), so that there is a possible match. For brevity, we simply use *string* to refer to "admissible string".

We define a global state of a system as a the number of the active processes for each site and annotation. This is enough to capture all effects of completely local protocols. The initial condition Θ of the system establishes that there are no active process: $a_r[i] = 0$ for all sites r and annotations i. If a state σ satisfies a predicate θ we write $\sigma \vDash \theta$.

A protocol P can reach a system state σ after executing the sequence of operations described by the string s (which we represent by $\sigma = P(s)$) if:

(1) $s = \epsilon$ and $\sigma \vDash \Theta$.

(2) $s = s_1 \cdot n$, $\sigma_1 = P(s_1)$, and $\sigma = In_n(\sigma_1)$ with $\sigma_1 \vDash En_n$.

(3) $s = s_1 \cdot n$, $\sigma_1 = P(s_1)$, and $\sigma = \bot$ with $\sigma_1 \nvDash En_n$.

(4) $s = s_1 \cdot \overline{n}$, $\sigma_1 \in P(s_1)$, and $\sigma = Out_n(\sigma_1)$.

Initiation is encoded by (1): before any operation is executed the systems is the initial state. Consecution of an allocation is encoded by (2) if the allocation is possible, or (3) if it is not a legal allocation. Deallocations are always legal, and encoded by (4). The special state $\bot$ is introduced here to indicate that s does not represent a legal sequence of operations for protocol P. We extend the input and output actions to preserve $\bot$, and force all enable conditions to be false in $\bot$.

A string s is *accepted* by a protocol P, and we write $s \in \mathcal{L}(P)$ if $\bot \neq P(s)$. We define a partial order $\sqsubseteq$ on protocols based on language inclusion: $P \sqsubseteq Q$ whenever $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$.

LEMMA 4.3. *The following are equivalent:*

(1) $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$.

(2) For all strings s *and allocation symbols* n*, if* $En_n^P(P(s))$ *then* $En_n^Q(Q(s))$.

PROOF. Assume $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$, let s be an arbitrary string and n an arbitrary symbol, with $En_n^P(P(s))$. First, $s \in \mathcal{L}(P)$ and consequently $s \in \mathcal{L}(Q)$; second, $s \cdot n \in \mathcal{L}(P)$ and then $s \cdot n \in \mathcal{L}(Q)$. Hence, $En_n^Q(Q(s))$.

Assume now (2). We show by induction on strings that if $s \in \mathcal{L}(P)$ then $s \in \mathcal{L}(Q)$.

—First, both $\epsilon \in \mathcal{L}(P)$ and $\epsilon \in \mathcal{L}(Q)$.

—Let $s \cdot n \in \mathcal{L}(P)$. Then $En_n^P(P(s))$, so also $En_n^Q(Q(s))$. Hence, $s \cdot n \in \mathcal{L}(Q)$.

—Let $s \cdot \overline{n} \in \mathcal{L}(P)$. This implies $s \in \mathcal{L}(P)$ and by inductive hypothesis $s \in \mathcal{L}(P)$. Then $s \cdot \overline{n} \in \mathcal{L}(P)$, as desired.

Therefore (1) and (2) are equivalent. □

Let P, Q be any two of BASIC-P, EFFICIENT-P, k-EFFICIENT-P and LIVE-P. We showed in Section 3 that the input and exit actions are identical for all these protocols. Therefore, if s is in the language of both P and Q then the states reached are the same, i.e., $P(s) = Q(s)$. It follows that if for all states σ, $En_n^P(\sigma)$ implies $En_n^Q(\sigma)$, then $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$. Consequently, comparing the enabling conditions for arbitrary states allows to conclude language containments.

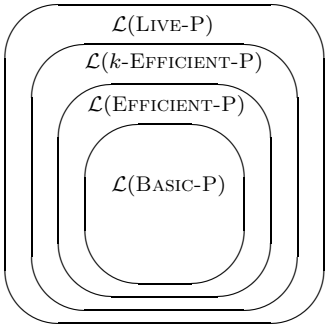


Fig. 1. Allocation sequences accepted by the different protocols.

4.1 Comparing the enabling conditions

The following lemma shows that the enabling condition of k-Efficient-P becomes stronger as k grows. This implies that the enabling condition of Basic-P is stronger than that of Efficient-P, which in turn is stronger than k-Efficient-P, which is stronger than Live-P.

Lemma 4.4. *If j-Efficient-P allows an allocation then k-Efficient-P also allows the allocation, provided $j \leq k$.*

Proof. It follows from the definition that if $j \leq k$ then $\psi_r^{(i)}[j]$ implies $\psi_r^{(i)}[k]$. Moreover, by (3), $\psi_r^{(i)}[j]$ implies $\bigwedge_{j \leq l \leq k} \varphi_r^{(i)}[l]$. Consequently,

$$\underbrace{\bigwedge_{l<j} \varphi_r^{(i)}[l] \wedge \psi_r^{(i)}[j]}_{En_n^{j\text{-Efficient-P}}} \rightarrow \underbrace{\bigwedge_{l<k} \varphi_r^{(i)}[l] \wedge \psi_r^{(i)}[k]}_{En_n^{k\text{-Efficient-P}}}$$

Therefore if j-Efficient-P allows the request so does k-Efficient-P. □

Lemma 4.4 directly implies that an allocation allowed by Basic-P is also allowed by Efficient-P, and that an allocation allowed by Efficient-P is also allowed by k-Efficient-P with $k \geq 2$. Moreover:

Corollary 4.5. *If k-Efficient-P allows an allocation then so does Live-P.*

4.2 Protocol Comparison by Allocation Sequences

As an immediate consequence of lemmas 4.4 and 4.5 we conclude that the languages accepted by these protocols are partially ordered as follows:

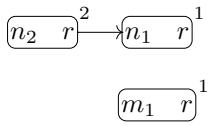
$$\text{Basic-P} \sqsubseteq \text{Efficient-P} \sqsubseteq \ldots \sqsubseteq k\text{-Efficient-P} \sqsubseteq \ldots \sqsubseteq \text{Live-P}$$

The following examples shows that these language containments are strict:

$$\text{Basic-P} \not\supseteq \text{Efficient-P} \not\supseteq \ldots \not\supseteq k\text{-Efficient-P} \not\supseteq \ldots \not\supseteq \text{Live-P}$$

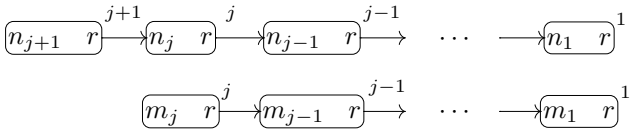
which is depicted in Fig 1.

EXAMPLE 4.1. *Consider the following call-graph, with initial resources* $T_r = 2$.



The string $m_1 n_2$ *is accepted by* EFFICIENT-P *but not by* BASIC-P.

EXAMPLE 4.2. *The previous example can be generalized to show that there is a string accepted by* k-EFFICIENT-P *but not by* j-EFFICIENT-P (*for* $j < k$). *Consider the following annotated call graph, with initial resources* $T_r = j + 1$.



The string $m_j n_{j+1}$ *is accepted by* k-EFFICIENT-P, *but is not accepted by* j-EFFICIENT-P.

Example A.1 in Appendix A generalizes these examples to a single scenario that shows all the proper containments.

5. REACHABLE STATE SPACES

In this section we study the notion of state space reachable by a protocol, and study a containment relation between the protocols based on their reachable state spaces.

Let P be an completely local protocol. The reachable state space of P, denoted by $\mathcal{S}(P)$, is the set of global states that P can reach following some legal allocation sequence. It follows immediately that if the actions of two protocols P and Q are equivalent, then $P \sqsubseteq Q$ implies that every state reachable by P is also reachable by Q: any string that witnesses the reachability of a state for P also witnesses that the same state is reachable for Q.

LEMMA 5.1. *For every two protocols* P *and* Q *with the same input and output actions, if* $P \sqsubseteq Q$ *then* $\mathcal{S}(P) \subseteq \mathcal{S}(Q)$.

Consequently, the reachable state spaces are ordered by:

$$\mathcal{S}(\text{BASIC-P}) \subseteq \mathcal{S}(\text{EFFICIENT-P}) \subseteq \ldots \subseteq \mathcal{S}(k\text{-EFFICIENT-P}) \subseteq \ldots \subseteq \mathcal{S}(\text{LIVE-P})$$

We show in the rest of this section that this containments are not proper, by proving that BASIC-P can reach all φ-states, which together with LIVE-P $\subseteq \varphi$-states, implies that all reachable spaces are actually the same: all φ-states.

5.1 Topological Orders of Annotated Call Graphs.

Given a call graph, a total order $>$ in its nodes is called topological if it respects the descendant relation, that is, if for every pair of nodes n and m, if $n \to m$ then $n > m$. Analogously, we say that an order $>$ respects an annotation α if for every pair of nodes n and m residing in the same site, if $\alpha(n) > \alpha(m)$ then $n > m$. A total order that is topological and respects annotations is called compatible.

LEMMA 5.2. *Every call graph annotated with an acyclic annotation has a compatible order.*

PROOF. The proof proceeds by induction in the number of call-graph nodes. The result trivially holds for the empty call-graph. Assume the result holds for all call-graphs with at most k nodes and take an arbitrary call-graph with $k+1$ nodes.

First, there must be a root node whose annotation is the highest among all the nodes residing in the same site. Otherwise a dependency cycle can be formed: take the maximal nodes for all sites, which are internal by assumption, and their root ancestors. For every maximal (internal) node there is $\rightarrow^+$ path reaching it, starting from its corresponding root. Similarly, for every root there is an incoming $\dashrightarrow$ edge from the maximal internal node that resides in its site. A cycle exist since the (bipartite) subgraph of roots and maximal nodes is finite, and every node has a successor (a $\rightarrow^+$ for root nodes, and a $\dashrightarrow$ for maximal nodes). This contradicts that the annotation is acyclic.

Now, let n be a maximal root node, and let $>$ be a compatible order for the graph that results by removing n, which exists by inductive hypothesis. We extend $>$ by adding $n > m$ for every other node m. The order is topological since n is a root. The order respects annotations since n is maximal in its site. □

5.2 Reachable States

A global state of a distributed system is *admissible* if all existing processes (active or waiting) in a node n are also existing, and active, in every node ancestor of n. That is, if the state corresponds to the outcome of some admissible allocation sequence.

THEOREM 5.3. *The set of reachable states of a system using* LIVE-P *is precisely the set of φ-states.*

PROOF. It follows directly from the specification of the LIVE-P protocol that all reachable states satisfy φ. Therefore, we only need to show that all φ-states are reachable. Let $>$ be a topological order in the nodes, compatible with the annotation as guaranteed by Lemma 5.2. Given a global state σ let $\sigma(n)$ indicate the utilization of node n (the number of active processes running node n in state σ). We partially order the set of global states, by having $\sigma_1 \succ \sigma_2$ whenever:

(1) $\sigma_1(n) > \sigma_2(n)$ for some node n, and
(2) $\sigma_1(m) = \sigma_2(m)$ for all prior nodes $m < n$.

Since the maximum utilization of any node is bounded, the order $\succ$ is finitely-based, and consequently well-founded. We show by induction in $\succ$ that every φ-state is reachable.

The base of induction is the global state with no utilization in any node, which is the initial state of the system and therefore it is reachable. Consider now an arbitrary φ-state σ with some utilization. Let n be an arbitrary node with an active process P that has no active descendant. Such a process is guaranteed to exist since call-graphs are finite, and leafs have no descendants (active or otherwise). Let σ' be the state that is obtained from σ by removing P from being an active process in node n. The state σ' is strictly smaller than σ in $\succ$, and is also a φ-state. By induction hypothesis, σ' is reachable. Since σ is obtained from σ' by an allocation that preserves φ, σ is also reachable, as desired. □

In terms of the allocation strings, Theorem 5.3 is equivalent to show that for every sequence s that arrives to a φ-state there is a sequence s' arriving at the same state

for which all prefixes also reach to φ-states. The sequence s' is in the language of Live-P.

Perhaps more surprisingly, the set of reachable states of Basic-P is also the set of all φ-states. To prove this we first need an auxiliary lemma.

Lemma 5.4. *In every φ-state, an allocation request in site r with annotation k has the same outcome using* Basic-P *and* Live-P*, if there is no active process in r with annotation strictly smaller than k.*

Proof. First, in every φ-state, if Basic-P grants a resource so does Live-P, by Lemma 4.4. We show that in every φ-state, if Live-P grants a request of k and $a_r[j] = 0$ for all $j < k$, then Basic-P also grants the request. In this case,

$$T_r - t_r = A_r[1] = \sum_{j=1}^{T_r} a_r[j] = \sum_{j=k}^{T_r} a_r[j] = A_r[k], \tag{4}$$

and since Live-P grants the request, then $A_r[k]+1 \leq T_r-(k-1)$ and $A_r[k] \leq T_r-k$. Using (4), $T_r - t_r \leq T_r - k$, and $t_r \geq k$, so Basic-P also grants the resource. □

Theorem 5.5. *The set of reachable states of a system using* Basic-P *is precisely the set of φ states.*

Proof. The proof is analogous to the characterization of the reachable states of Live-P, except that we also observe that the constructed run to the desired state satisfies that at every transition the request is performed in a node with no active process of lower annotation. By Lemma 5.4 the request is granted because it is granted by Live-P (both starting and reaching states are φ-states) and the allocation picked to be in a node whose sites have no smaller active processes. □

Theorem 5.5 can also be restated in terms of allocation sequences. For every admissible allocation string that arrives to a φ-state there is an admissible allocation string that, arrives at the same state and (1) contains no deallocations, and (2) all the nodes occur according to some compatible order. It follows from Theorem **??** that $\mathcal{S}$Basic-P = φ-states, so:

$$\mathcal{S}(\text{Basic-P}) = \mathcal{S}(\text{Efficient-P}) = \ldots = \mathcal{S}(k\text{-Efficient-P}) = \ldots = \mathcal{S}(\text{Live-P})$$

as depicted in Fig 2.

6. CONCLUSIONS AND OPEN PROBLEMS

The fact that the reachable state space of k-Efficient-P is all the φ-states can be used as a proof technique: if the decision of some protocol can be stated as a transition into an φ-state, then the protocol still provides deadlock avoidance. This is used in [Sánchez et al. 2006] to create an efficient distributed priority inheritance mechanism where annotations are used to encode priorities; inheriting a priority is carried out by an annotation decrease. This does not compromise deadlock freedom because decreasing the annotation of an existing process in a φ-state results into an φ-state.

The following conjectures are still open problems:

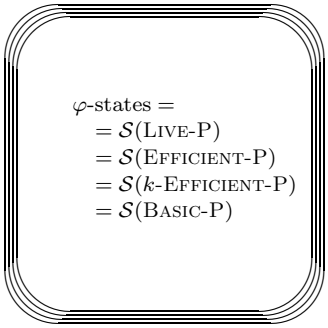


Fig. 2. Reachable state spaces of the different protocols.

(1) Given a completely local protocol P that guarantees deadlock avoidance for all initial resources, there is a minimal annotation for which $P \sqsubseteq$ Live-P. This would imply an optimality result for Live-P and an indication that the φ captures the essence of deadlock avoidance.

(2) Let P be a completely local protocol that guarantees deadlock avoidance for all initial resources, and whose enabling condition and input and exit actions can be computed using $O(log\ T_r)$ bits at each site r. Then for some minimal annotation $P \sqsubseteq k$-Efficient-P for some k. This result would imply that k-Efficient-P is optimal among protocols that use only a constant number of counters.

REFERENCES

Birrell, A. D. 1989. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center.

Dijkstra, E. W. 1965. Cooperating sequential processes. Tech. Rep. EWD-123, Technological University, Eindhoven, the Netherlands.

Habermann, A. N. 1969. Prevention of system deadlocks. *Communications of the ACM 12*, 373–377.

Havender, J. W. 1968. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal 2*, 74–84.

Sánchez, C., Sipma, H. B., and Manna, Z. 2006. Efficient distributed deadlock avoidance with liveness guarantees. Submitted for publication.

Sánchez, C., Sipma, H. B., Manna, Z., and Gill, C. D. 2006. Priority inheritance for distributed real-time and embedded systems. In preparation.

Sánchez, C., Sipma, H. B., Manna, Z., Subramonian, V., and Gill, C. 2006. 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)*. IEEE Computer Society Press.

Sánchez, C., Sipma, H. B., Subramonian, V., Gill, C., and Manna, Z. 2005. Thread allocation protocols for distributed real-time and embedded systems. In *25th IFIP WG 2.6 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, F. Wang, Ed. LNCS, vol. 3731. Springer-Verlag, 159–173.

Schmidt, D. C. 1998. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Communications of the ACM Special Issue on CORBA 41,* 10 (Oct.), 54–60.

Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York.

Silberschatz, A., Galvin, P. B., and Gagne, G. 2003. *Operating System Concepts*, Sixth ed. John Wiley & Sons, Inc.

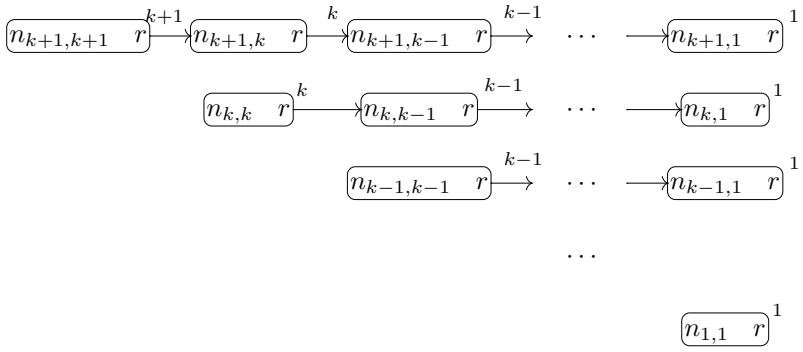
Singhal, M. and Shivaratri, N. G. 1994. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. McGraw-Hill, Inc.

Stallings, W. 1998. *Operating Systems: Internals and Design Principles*, Third ed. Prentice Hall.

Subramonian, V., Xing, G., Gill, C. D., Lu, C., and Cytron, R. 2004. 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.

A. MISSING PROOFS AND EXAMPLES

EXAMPLE A.1. *Let k be arbitrary, and consider the following annotated call graph, where all nodes reside in site r, and $T_r = k + 1$.*



The string $n_{k,k}n_{k+1,k+1}$ is accepted by LIVE-P*, but is not accepted by k-*EFFICIENT-P*. For arbitrary $j < k$, the following string is accepted by k-*EFFICIENT-P *but not by j-*EFFICIENT-P*:*

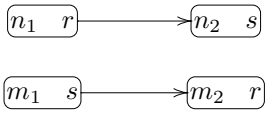
$$\underbrace{n_{j,j} \ldots n_{j,j}}_{T_r - j \text{ times}} n_{j+1,j+1}$$

This includes the following string, which is accepted by EFFICIENT-P *but not by* BASIC-P*:*

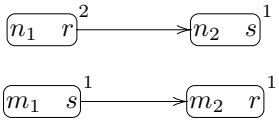
$$\underbrace{n_{1,1} \ldots n_{1,1}}_{T_r - 1 \text{ times}} n_{2,2}$$

B. A PROTOCOL THAT ACCEPTS A RUN THAT LIVE-P DOES NOT ACCEPT

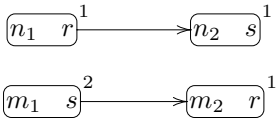
Consider the call graph:



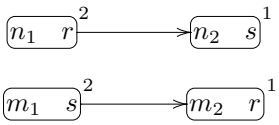
The protocol starts running as LIVE-P with the annotation:



The first time there is a process running in n_2 but no process running in m_1, it blocks all requests in m_1 until the process in n_2 finishes. This is a local operation controlled by a boolean variable in s. When the process in n_2 finishes, the output action *piggybacks* a token to n_1 as a part of the return call, and changes the annotation of m_1 to 2. Node n_1, when receiving the token changes its annotation to 1, effectively continuing as LIVE-P with annotation:



Observe that there is an interval during which the system is running with the annotation:



until the message of the return call that contains the token arrives to n_1.

We conjecture that this run, for large enough number of resources cannot be accepted by LIVE-P with any annotation whatsoever. However, this conjecture does not immediately contradict the optimality of LIVE-P, as stated in some section above, since the protocol presented here is not local: it exchanges messages as piggybacks.

CHANGE LOG

June 1, 2006.	Submitted as technical report
July 25, 2006.	Modified (correction of typos)