Thread Allocation Protocols for Distributed Real-Time and Embedded Systems*

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

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

Abstract. We study the problem of thread allocation in asynchronous distributed real-time and embedded systems. Each distributed node handles a limited set of resources, in particular a limited thread pool. Different methods can be invoked concurrently in each node, either by external agents or as a remote call during the execution of a method. In this paper we study thread allocation under a *WaitOnConnection* strategy, in which each nested upcall made while a thread is waiting must be made in a different thread.

We study protocols that control the allocation of threads to guarantee the absence of deadlocks. First, we introduce a computational model in which we formally describe the different protocols and their desired properties. Then, we study two scenarios: a single agent performing sequential calls, and multiple agents with unrestricted concurrency. For each scenario we present (1) algorithms to compute the minimum amount of resources to avoid deadlocks, and (2) run-time protocols that control the allocation of these resources.

1 Introduction

In this paper we present a computational model for thread allocation in distributed real-time and embedded (DRE) systems. The model is targeted at component middleware architectures in which components make remote two-way method calls to other components. In particular, we consider the case where a remote method call f by component A to component B may result in one or more method calls from B (or other components) to A before f returns. These method calls are known as "nested upcalls".

Nested upcalls can occur in the context of a variety of middleware concurrency architectures, including the Leader/Followers [16] approach used in

* 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.

TAO [15, 10] and the Half-Sync/Half-Async [16] approach used in the variant of nORB [4] used to integrate actual and simulated system components [18]. These kinds of middleware have been used in turn to build a wide range of DRE systems for applications ranging from avionics mission computing information systems [6] to active damage detection using MEMS vibration sensor/actuators [20].

Ensuring the safety and liveness of concurrent nested upcalls in middleware for DRE systems is thus an important and compelling problem. In current practice, middleware concurrency architectures are realized in software frameworks designed according to documented best practices, but it is still possible for unforeseen factors introduced during the design to result in safety and liveness violations [17]. Hence, a more formal basis can have a significant impact on the correctness of these systems, which motivates the work presented here.

There are two main strategies to deal with nested upcalls. The first is known as *WaitOnConnection*, where component A holds on to the thread from which method call f was invoked. With this strategy any nested calls to A will have to acquire a new thread to run. The second approach relies on the use of a *reactor*, a well-known technique for multiplexing asynchronously arriving events onto one or more threads [16]. This approach is known as *WaitOnReactor*, in which component A releases the thread after the method call is made. To preserve the semantics of the two-way call, it maintains a stack to keep track of the order in which methods were invoked, such that they can be exited in reverse order. Both approaches have advantages and disadvantages. A disadvantage of the first approach is that threads cannot be reused while the reactor is waiting for the method to return, which can lead to deadlock. A disadvantage of the second approach is that the stack must be unwound in *last-in-first-out* order, resulting in blocking delays for the completion of methods initiated earlier, which can lead to deadline violations. This may be especially problematic in systems with multiple agents.

In other complementary research [19] we have examined safety and liveness for the *WaitOnReactor* approach. In this paper we only consider the *WaitOnConnection* approach, and focus on deadlock avoidance in that context. We assume we are given a set of reactors $\mathcal{R}$, hosting components of the system, and a set of call-graphs $\mathcal{G}$, representing the possible call sequences that components can invoke. We also assume that each call graph may be invoked multiple times concurrently. The goals are: (1) to determine what minimum number of threads is necessary in each reactor to avoid deadlock; and (2) to construct protocols that are deadlock free and make efficient use of the threads available (that is, they do not unnecessarily block invocations). We will consider two cases: (1) the number of concurrent processes is fixed in advance (which is common in DRE systems), and (2) the number of concurrent processes is not bounded (such as in network services).

Related Work Deadlocks have been studied in many contexts. In computer science, one of the first protocols for deadlock avoidance was Dijkstra's Banker's algorithm [5], which initiated much follow-up research [7–9, 1], and which is still the basis for most current deadlock avoidance algorithms. In the control community, deadlock avoidance mechanisms have been studied in the context of Flexible Manufacturing Systems (FMSs) [14, 3, 21, 13]. In contrast with the Banker's algorithm, in which only the maximum amount of resources is taken into account, protocols for FMSs take into account more information about the processes to maximize concurrency without sacrificing deadlock freedom.

In the distributed systems community (see, for example, [11]), general solutions to distributed deadlock tend to be considered impractical. For example, global consistency and atomicity would be a prerequisite for a "distributed version" of the Banker's algorithm. Most approaches so far have focused on deadlock detection and roll-back (e.g. in distributed databases) and deadlock prevention by programming discipline (e.g. by observing a partial order on locks acquired).

The protocols presented in this paper can be viewed as a practical solution to the case where extra information is available in the form of call graphs, which is common in DRE systems. We show that the incorporation of this information enables efficient "local" protocols (that is, no communication between distributed nodes is required at runtime) that guarantee absence of deadlock.

2 Computational Model

We define a system $\mathcal{S}$ as a tuple $\langle \mathcal{R}, \mathcal{G} \rangle$ consisting of a set of reactors $\mathcal{R} : \{r_1, \ldots, r_k\}$ and a set of distinct call graphs $\mathcal{G} : \{G_1, \ldots, G_m\}$.

Definition 1 (Call-graph). *Given a set of method names N, a call-graph is a finite tree $\langle V = (N \times \mathcal{R}), E \rangle$ with nodes consisting of a method name and a reactor. A node (f, r) denotes that method f runs in reactor r. An edge from (f_1, r_1) to (f_2, r_2) denotes that method f_1, in the course of its execution may invoke method f_2 in reactor r_2.*

For ease of exposition we assume that methods of child nodes are hosted in a different reactor than their parent nodes. Local calls can always be run in the same thread, implemented as conventional method calls, and are not considered here. We use the notation $f{:}r$ to represent that method f runs in reactor r.

Example 1. Figure 1 shows an example of a call-graph. The methods f_i run in reactor r, the method g runs in reactor s, and the methods h_i run in reactor t. Method f_1 may invoke the remote methods g and h_1.

We assume that each reactor r has a fixed number of pre-allocated threads. Although in many modern operating systems threads can be spawned dynamically, many DRE systems pre-allocate fixed sets of threads to avoid the relatively large and variable cost of thread creation and initialization. We assume that each reactor r has a set of local variables V_r that includes the constant $T_r \geq 1$ denoting the number of threads present in r, and a variable t_r representing the number of available threads.

A *protocol* for controlling thread allocation is implemented by code executed in a reactor before and after each method is dispatched. This code can be dependent on the call graph node. The structure of a protocol is shown in Figure 2

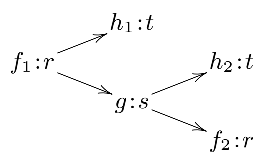


Fig. 1. A sample call graph G.

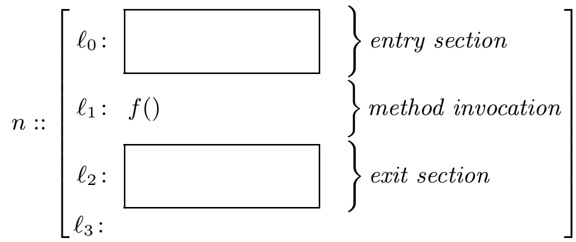


Fig. 2. Protocol schema.

for a node $n = f{:}r$. The entry and exit sections implement the thread allocation policy. Upon invocation, the entry section typically checks thread availability by inspecting local variables V_r of reactor r and assigns a thread if one is available. The method invocation section executes the code of f; it terminates after all its descendants in the call graph have terminated and returned. The exit section releases the thread and may update some local variables in reactor r.

Multiple instances of these protocols may execute concurrently, one for each invocation. Each instance is called a *task*. Thus the local variables of the reactor are shared between tasks running in the same reactor, but are not visible to tasks that reside in other reactors.

The global behavior of a system $\mathcal{S}$ is represented by sequences of (global) states, where a state $\sigma : \langle \mathcal{P}, s_{\mathcal{R}} \rangle$ contains a set of tasks together with their local states, $\mathcal{P}$, and a valuation $s_{\mathcal{R}}$ of the local variables in all reactors. To describe a task state we use the notion of labeled call graph:

Definition 2 (Labeled Call Graph). *Let ℓ_0, ℓ_1, ℓ_2, and ℓ_3 be protocol location labels representing the progress of a task, as illustrated in Figure 2. A labeled call graph (G, γ) is an instance of a call graph $G \in \mathcal{G}$ and a labeling function $\gamma : N_G \mapsto \{\perp, \ell_0, \ell_1, \ell_2, \ell_3\}$ that maps each node in the call graph to a protocol location, or to $\perp$ for method calls that have not been performed yet.*

Then, formally, the state of a task is modeled as a labeled call graph. A subtree of a labeled call graph models the state of a *sub-task*. When the context of the subtree is not relevant, we will write task to refer to the corresponding sub-task. A task is *active* if its root is labeled ℓ_1 or ℓ_2, *waiting* if it is labeled ℓ_0, and *terminated* if it is labeled ℓ_3. We use upper-case letters $P, Q, P_1, \ldots$ to denote tasks and lower case letters $n, n_1, \ldots$ to denote call-graph nodes. To simplify the presentation, given a task $P = (G, \gamma)$ we use $\gamma(P)$ as an alias of $\gamma(\text{root}(P))$. We also say that task P is in location ℓ if $\gamma(P) = \ell$.

A *system* $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ gives rise to the *state transition system* (see [12]) $\Psi : \langle V, \Theta, \mathcal{T} \rangle$ consisting of the following components:

- $V : \{I\} \cup V_R$: a set of variables, containing the variable I denoting a set of labeled call graphs (tasks and their state), and the local variables $V_R = \bigcup_{r \in \mathcal{R}} V_r$ of each reactor.

- $\Theta: \ I = \varnothing \wedge \bigwedge_{r \in \mathcal{R}} \Theta_r$: the initial condition, specifying initial values for the local reactor variables and initializing the set of tasks to the empty set.
- $\mathcal{T}$: a set of state transitions consisting of the following global transitions:
  1. **Creation:** A new task P, with $\gamma(n) = \perp$ for all nodes n in its call graph, is added to I:
  $$\tau_1 : I' = I \cup \{P\} \wedge \mathit{pres}(V_R)$$
  where $\mathit{pres}(V_R)$ states that all variables in V_R are preserved.
  2. **Method invocation:** Let $P \in I$ and Q be a sub-task of P such that either (a) $Q = P$ or (b) $\gamma(Q) = \perp$ and its parent node is in ℓ_1. A function invocation changes the annotation of Q to ℓ_0:
  $$\tau_2 : \gamma(Q) = \perp \wedge \gamma'(Q) = \ell_0 \wedge \mathit{pres}(V_R)$$
  3. **Method entry:** Let Q be a waiting task whose enabling condition is satisfied. The method entry transition marks Q as ℓ_1 and updates the local variables in its reactor according to the protocol of the node $n = \mathrm{root}(Q)$. Formally, let $\mathit{En}_n(V_r)$ be the enabling condition of the entry of the protocol of Q, and $\mathit{Act}_n(V_r, V_r')$ represent the change in variables of reactor r after the entry is executed; then:
  $$\tau_3 : \gamma(Q) = \ell_0 \wedge \mathit{En}_n(V_r) \wedge \gamma'(Q) = \ell_1 \wedge \mathit{Act}_n(V_r, V_r') \wedge \mathit{pres}(V_R - V_r)$$
  4. **Method execution:** Let Q be a task in ℓ_1 such that all its descendants are labeled $\perp$ or ℓ_3. This transition denotes the termination of Q. The status of Q is updated to ℓ_2:
  $$\tau_4 : \ \gamma(Q) = \ell_1 \wedge \bigwedge_{R \in \mathit{descs}(Q)} (\gamma(R) = \perp \vee \gamma(R) = \ell_3) \wedge \gamma'(Q) = \ell_2 \wedge \mathit{pres}(V_R)$$
  5. **Method exit:** Let Q be a task in ℓ_2; the method-exit transition moves Q to ℓ_3 and updates the variables in its reactor according to the exit of the protocol for $n = \mathrm{root}(Q)$. Formally, let $\mathit{Out}_n(V_r, V_r')$ represent the transformation that the exit protocol for n performs on the variables of reactor r; then
  $$\tau_5 : \ \gamma(Q) = \ell_2 \ \wedge \ \gamma'(Q) = \ell_3 \ \wedge \mathit{Out}_n(V_r, V_r') \wedge \mathit{pres}(V_R - V_r)$$
  6. **Deletion:** A task P in ℓ_3 is removed from I:
  $$\tau_6 : \gamma(P) = \ell_3 \ \wedge \ I' = I - \{P\}$$
  7. **Silent:** All variables are preserved: $\quad \tau_7 : \mathit{pres}(V)$

All transitions except **Creation** and **Silent** are called *progressing* transitions, since they correspond to the progress of some existing task. The system as defined is a nondeterministic system. It assumes an external environment that determines creation of new tasks, and a scheduler that selects which task progresses. In particular, the scheduler decides which task in the entry transition proceeds to the method section. If any progressing transition *can* occur, then some progressing transition will be taken in preference to any non-progressing transition. Therefore, unless the system is deadlocked, an infinite sequence of silent transitions cannot occur because a progressing transition will occur eventually. If in state σ there are k active tasks corresponding to methods in reactor r, then we say that there are k threads allocated in r.

Definition 3 (Run). *A* run *of a system Ψ is a sequence of $\sigma_0, \sigma_1, \ldots$ of states such that σ_0 is an initial state, and for every i, there exists a transition $\tau \in \mathcal{T}$ such that σ_{i+1} results from taking τ from σ_i.*

3 Properties

In this section we formally define some properties to study the correctness of the protocols. Most properties will be presented as *invariants*. Figure 3(a) illustrates the simplest possible protocol, Empty-P, in which the entry and exit sections do nothing with the reactor variables.

Definition 4 (Invariant). *Given a system $\mathcal{S}$, an expression φ over the system variables of $\mathcal{S}$ is an invariant of $\mathcal{S}$ if it is true in every state of every run of $\mathcal{S}$.*

An expression can be proven invariant by showing that it is inductive or implied by an inductive expression. An expression φ is *inductive* for a transition system $\mathcal{S} : \langle V, \Theta, \mathcal{T} \rangle$ if it is implied by the initial condition, $\Theta \rightarrow \varphi$, and it is preserved by all its transitions, $\varphi \wedge \tau \rightarrow \varphi'$, for all $\tau \in \mathcal{T}$.

Definition 5 (Adequate). *A protocol is adequate if the number of threads allocated in every reactor r never exceeds the total number of threads in r, T_r.*

Adequacy is a fundamental property, required in every reasonable protocol, since no more resources than available can possibly be granted. Adequate-P is a simple adequate protocol, shown in Figure 3(b), in which the entry section (ℓ_0) blocks further progress until the guard expression $1 \leq t_r$ evaluates to true. Its adequacy is a consequence of the following invariants:

$$
\begin{aligned}
\psi_1 &: \ \forall r \in \mathcal{R} \,.\, t_r \geq 0 \\
\psi_2 &: \ \forall r \in \mathcal{R} \,.\, T_r = t_r + at\_\ell_{1,r} + at\_\ell_{2,r}
\end{aligned}
$$

where, $at\_\ell_{1,r}$ and $at\_\ell_{2,r}$ denote the total number of active sub-tasks in reactor r. It is easy to show that ψ_1 and ψ_2 are inductive.

Definition 6 (Deadlock). *A state σ is a deadlock if some task is in ℓ_0, but only non-progressing transitions are enabled.*

If a deadlock is reached, the tasks involved cannot progress. Intuitively, each of the tasks has locked some resources—threads in our case—that are necessary for other tasks to complete, but none of them has enough resources to terminate. The following example shows that Adequate-P does not guarantee absence of deadlock.

$$n :: \begin{bmatrix} \ell_0: & \textbf{skip} \\ \ell_1: & f() \\ \ell_2: & \textbf{skip} \\ \ell_3: & \end{bmatrix}$$

(a) The protocol EMPTY-P.

$$n :: \begin{bmatrix} \ell_0: & \begin{bmatrix} \textbf{when } 1 \leq t_r \textbf{ do} \\ t_r\texttt{--} \end{bmatrix} \\ \ell_1: & f() \\ \ell_2: & t_r\texttt{++} \\ \ell_3: & \end{bmatrix}$$

(b) The protocol ADEQUATE-P.

Fig. 3. Protocols EMPTY-P and ADEQUATE-P for node $n = (f\!:\!r)$.

Example 2. Consider the system $\mathcal{S}: \langle \{r, s\}, \{G_1, G_2\} \rangle$ with two reactors r and s, and two call graphs

$$G_1 : n_{11} : \langle f\!:\!r \rangle \longrightarrow n_{12} : \langle g_2\!:\!s \rangle \quad \text{and} \quad G_2 : n_{21} : \langle g\!:\!s \rangle \longrightarrow n_{22} : \langle f_2\!:\!r \rangle .$$

Both reactors have one thread ($T_r = T_s = 1$). Assume the protocol for all nodes is ADEQUATE-P. Let $\sigma : \langle \{P_1, P_2\}, t_r = 0, t_s = 0 \rangle$ be a state with two tasks: P_1 an instance of G_1 with $\gamma(n_{11}) = \ell_1$ and $\gamma(n_{12}) = \ell_0$, and P_2 an instance of G_2 with $\gamma(n_{21}) = \ell_1$ and $\gamma(n_{22}) = \ell_0$. It is easy to see that σ is a deadlock: no progressing transition is enabled. Furthermore, σ is reachable from an initial state and hence appears in some run.

In a deadlock state, independent of the protocol used, any task that is active must have a descendant task that is waiting for a thread, as expressed by the following lemma.

Lemma 1. *In a deadlock state, any active task has a waiting descendant.*

Proof. Let σ be a deadlock state and P an active task. Then $\gamma(P) = \ell_1$, since for $\gamma(P) = \ell_2$, transition τ_5 is enabled, contradicting deadlock. We prove that P has at least one waiting descendant by induction on the position of P in the call graph. For the base case, let P correspond to a leaf node. But then transition τ_4 is enabled, contradicting deadlock. Thus a leaf node cannot be active in a deadlock state. For the inductive case, let $Q_1, \ldots, Q_n$ be the descendants of P. If some Q_i is waiting we are done. If some Q_i is active, by the inductive hypothesis, it has a waiting descendant, and hence P has a waiting descendant. Otherwise for all Q_i, $\gamma(Q_i) = \perp$ or $\gamma(Q_i) = \ell_3$. But then τ_4 is enabled, contradicting deadlock. $\square$

Another desirable property of thread allocation protocols is absence of starvation, that is, every task eventually progresses. A task P starves in a run of a system if, after some prefix run, the labeling of P never changes thereafter. A system prevents starvation if no task starves in any of its runs.

Deadlock implies starvation, but the converse does not hold. For example, if other tasks are scheduled in preference to it throughout the run, but the task could have made progress had it been scheduled, that constitutes starvation but not deadlock. This raises the question of schedulers to enforce particular (possibly application-specific [2]) policies for process scheduling, but a detailed consideration of that issue is outside the scope of this paper.

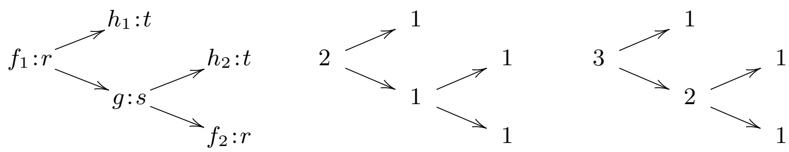


Fig. 4. Sample call graph (left) and its local-height (center) and height (right) annotations.

4 Deadlock-avoidance Protocols

This section introduces several protocols and studies whether they prevent deadlocks in different scenarios. The protocols have the structure shown in Figure 2, where the entry and exit sections may be different for different nodes in the call graph. More precisely, the protocols are parameterized by an annotation of the call graphs, $\alpha : V \mapsto \mathbb{N}^+$, that maps nodes of all call graphs to the positive natural numbers. Intuitively, the annotation provides a measure of the resources—threads in our case—needed to complete the task corresponding to the node. We consider two annotations: *height* and *local height*. Height of a node in a call graph is the usual height of a node in a tree. Local height only takes into account nodes in the same reactor.

Definition 7 (Height). *Given a call graph G, the height of a node n in G, written $h(n)$, is*

$$h(n) = \begin{cases} 1 & \textit{if } n \textit{ is a leaf, and} \\ 1 + max\{h(m) \mid n \rightarrow m\} & \textit{otherwise.} \end{cases}$$

where $n \rightarrow m$ denotes that m is a child node of n.

Definition 8 (Local Height). *Given a call graph G, the local height of a node $n = f{:}r$ in G, written $lh(f{:}r)$ is*

$$lh(f{:}r) = \begin{cases} 1 & \textit{if } f{:}r \textit{ is a leaf, and} \\ 1 + max\{lh(g{:}s) \mid f{:}r \rightarrow^+ g{:}s \textit{ and } r = s\} & \textit{otherwise.} \end{cases}$$

where $n \rightarrow^+ m$ denotes that m is a descendant of n.

Example 3. Figure 4 shows a call-graph (left) and its local height (center) and height (right) annotations. Here, $n = f_1 : r$ has local height 2, since f_1 may indirectly call f_2 in the same reactor through a nested call.

4.1 Single Agent

We first consider the scenario of a single agent sequentially activating tasks, also studied in [17]. This scenario represents an environment that cannot make any

$$n :: \left[\begin{array}{ll} \ell_0: & \left[\begin{array}{c}\textbf{when } \alpha(n) \le t_r \textbf{ do}\\ t_r\texttt{--}\end{array}\right] \\ \ell_1: & f() \\ \ell_2: & t_r\texttt{++} \\ \ell_3: & \end{array}\right]$$

Fig. 5. The protocol Basic-P for call-graph node $n = (f\!:\!r)$.

concurrent calls. In terms of our model, this corresponds to systems in which the number of tasks in any state is at most 1, that is, systems for which $|I| \le 1$ is an invariant. In this scenario the following theorem establishes a necessary and sufficient condition to guarantee absence of deadlocks.

Theorem 1 (from [17]). *To perform a call with local-height* n *with absence of deadlock, at least* n *available threads in the local reactor are needed.*

Theorem 1 provides a simple, design-time method to compute the minimum number of threads, T_r, needed in each reactor r to guarantee absence of deadlock: T_r must be at least the maximum local height for any node in any call graph whose method call resides in r. The condition is necessary, because if it is violated a deadlock will occur, independent of the protocol used. The condition is sufficient, because if it is met, no deadlock will occur. Thus, in the single agent case the trivial protocol Empty-P, shown in Figure 3(a), will guarantee absence of deadlock, provided all reactors have the required number of threads.

4.2 Multiple Agents: Generic Protocols

In case of multiple agents performing multiple concurrent calls, the condition expressed in Theorem 1 is necessary but not sufficient to guarantee the absence of a deadlock (using Empty-P) as illustrated by the following example.

Example 4. Consider again the system of Example 2. This system satisfies the condition of Theorem 1: the local heights of the nodes are $lh(n_{11}) = lh(n_{12}) = lh(n_{21}) = lh(n_{22}) = 1$ and $T_r = T_s = 1$. A deadlock is produced however, if P_1, an instance of G_1, takes the thread in r and P_2, an instance of G_2, takes the thread in s.

Indeed, it can be shown that no number of pre-allocated threads in reactors r and s in the above example can prevent deadlock in the presence of an unbounded number of multiple concurrent calls. Thus more sophisticated protocols are needed to control access to the threads.

We propose two such protocols: Basic-P and Efficient-P, both parameterized by the annotation functions $\alpha_G : G \mapsto N,\ G \in \mathcal{G}$. In this section we present some properties for generic annotations α_G; in the next section we analyze deadlock avoidance and resource utilization for some specific height annotations.

The first protocol, Basic-P, is shown in Figure 5. The reactor variable t_r is used to keep track of the threads currently available in the reactor. In the entry section access is granted only if the number of resources indicated by the annotation function at that node, $\alpha(n)$, is less than or equal to the number of threads available. When access is granted, t_r is decremented by one, reflecting that one thread has been allocated. Note that not all resources requested are reserved.

The second protocol, EFFICIENT-P, shown in Figure 6, exploits the observation that every task that needs just one resource can always, independently of other tasks, terminate once it gets the resource. The protocol has two reactor variables, t_r and p_r, where t_r, as in BASIC-P, keeps track of the number of threads currently available, and p_r tracks the threads that are potentially available. The difference with BASIC-P is that the number of *potentially* available threads is not reduced when a thread is granted to a task that needs only one thread. With EFFICIENT-P fewer tasks are blocked, thus increasing potential concurrency and improving resource utilization.

Example 5. Consider the system $\mathcal{S}: \langle\{r, s\}, \{G_1, G_2\}\rangle$ with $T_r = T_s = 2$ and

$$G_1 : n_{11} : \langle f_1 {:} r\rangle \quad \text{and} \quad G_2 : n_{21} : \langle f_2 {:} r\rangle \longrightarrow n_{22} : \langle g {:} s\rangle .$$

with annotations $\alpha(n_{11}) = \alpha(n_{22}) = 1$ and $\alpha(n_{21}) = 2$. Assume the following arrival of tasks: P_1: an instance of G_1 and P_2 an instance of G_2. With BASIC-P, P_2 is blocked until P_1 is finished and has released the thread, while with EFFICIENT-P, P_2 can run concurrently with P_1.

To study the properties of different annotations, we first establish some properties that hold for EFFICIENT-P [3] for any annotation α. We first introduce some notation and abbreviations. Let $at\_\ell_{i\_j,r}$ denote the number of tasks in a reactor r that are at location ℓ_{i_j}. Then the number of active tasks P with annotation $\alpha(P) = 1$ in r is equal to $act_{1,r} \stackrel{\text{def}}{=} at\_\ell_{1\_1,r} + at\_\ell_{2\_1,r}$, and the number of tasks in r with $\alpha(P) > 1$ is $act_{>1,r} \stackrel{\text{def}}{=} at\_\ell_{1\_2,r} + at\_\ell_{2\_2,r}$. Let $act_{>k,r}$ denote the number of tasks in r with annotation greater than k. With initial condition $\Theta_r: \ t_r = p_r \wedge T_r = t_r$, it is easy to verify that the following are inductive invariants for all reactors.

$$\begin{aligned}
&\varphi_1 : t_r \geq 0 \\
&\varphi_2 : p_r \geq 1 \\
&\varphi_3 : p_r = t_r + act_{1,r} \\
&\varphi_4 : T_r = p_r + act_{>1,r}
\end{aligned}$$

The following lemmas apply to all reactors.

Lemma 2. *If $t_r = 0$ then there exists at least one active task with annotation 1 in r, that is, $\varphi_5: \ t_r = 0 \rightarrow act_{1,r} \geq 1$ is an invariant.*

Proof. Follows directly from φ_2 and φ_3. □

[3] The same properties hold for BASIC-P, but we will not prove them here.

$$n :: \begin{bmatrix} \ell_{0\_1}: & \begin{bmatrix} \textbf{when } 1 \le t_r \textbf{ do} \\ t_r\texttt{--} \end{bmatrix} \\ \ell_{1\_1}: & f() \\ \ell_{2\_1}: & t_r\texttt{++} \\ \ell_{3\_1}: & \end{bmatrix} \qquad n :: \begin{bmatrix} \ell_{0\_2}: & \begin{bmatrix} \textbf{when } \alpha(n) \le p_r \ \wedge \ 1 \le t_r \textbf{ do} \\ \langle p_r\texttt{--},\ t_r\texttt{--} \rangle \end{bmatrix} \\ \ell_{1\_2}: & f() \\ \ell_{2\_2}: & \langle t_r\texttt{++},\ p_r\texttt{++} \rangle \\ \ell_{3\_2}: & \end{bmatrix}$$

If $\alpha(n) = 1$ If $\alpha(n) > 1$

Fig. 6. The protocol Efficient-P for node $n = (f\!:\!r)$.

Lemma 3. *The number of active tasks P with annotation $\alpha(P) > k$, for $0 \le k \le T_r$, is less than or equal to $T_r - k$, that is, $\varphi_6 :\ act_{>k,r} \le T_r - k$ is an invariant.*

Proof. To show that φ_6 is an invariant, it suffices to show that in a state where $act_{>k,r} = T_r - k$, a waiting task Q with $\alpha(Q) > k$ cannot proceed. For $k = 0$ we have $act_{>0,r} = T_r$. By φ_3 and φ_4, $T_r = t_r + act_{>1,r} + act_{1,r} = t_r + act_{>0,r}$, and thus $t_r = 0$. Consequently, the transitions for both ℓ_{0_1} and ℓ_{0_2} are disabled for Q. For $k > 0$, note that $act_{>k,r} \le act_{>1,r}$, and thus in a state where $act_{>k,r} = T_r - k$, we have $T_r - k \le act_{>1,r}$. By φ_4, $p_r = T_r - act_{>1,r}$, and thus $p_r \le k$. Consequently, transition ℓ_{0_2} is disabled for k, as by assumption $\alpha(Q) > k$. □

Lemma 4. *If a task P is in location ℓ_{0_2} in r and the transition for ℓ_{0_2} is not enabled for P, then there is an active task Q in r with annotation $\alpha(Q) \le \alpha(P)$.*

Proof. Note that $\alpha(P) > 1$. Transition ℓ_{0_2} is disabled for P if $t_r = 0$ or if $\alpha(P) > p_r$. If $t_r = 0$, then by Lemma 2 there exists an active task Q with annotation 1, and hence $\alpha(Q) < \alpha(P)$. If $\alpha(P) > p_r$ then by φ_4, $act_{>1,r} > T_r - \alpha(P)$. However, by Lemma 3, $act_{>\alpha(P),r} \le T_r - \alpha(P)$. Thus there must at least be one active task Q in r such that $\alpha(Q) \le \alpha(P)$. □

4.3 Protocols Based on Height and Local Height

In Section 4.2 we introduced the protocol Efficient-P for a generic call graph annotation α that provides a measure of the number of resources required to complete the task. In this section we analyze two such measures: local height (Def. 8) and height (Def. 7). Local height requires the least resources. Unfortunately it does not guarantee freedom of deadlock. We prove that using height does guarantee absence of deadlock. However, for many designs it is too conservative in its requirements for resources. Therefore in Section 4.4 we propose a less conservative annotation, based on the combination of all graphs rather than on individual call graphs, that still guarantees deadlock freedom.

Local Height Using local height, as defined in Def. 8 for α in the protocols Basic-P and Efficient-P does not guarantee absence of deadlock. A simple counterexample is provided by Example 4 for both protocols.

Height We will now prove that using height, as defined in Def. 7 for α guarantees absence of deadlock. We assume that for every reactor the number of threads T_r is greater than or equal to the highest annotation of any node that runs in r in any call graph in the system. We first prove one more auxiliary lemma.

Lemma 5. *With the use of* EFFICIENT-P *with the height annotation, every task P with $h(P) = 1$ can complete.*

Proof. Note that P can always progress when it is active, since it is a leaf node. Thus it is sufficient to show that it can eventually progress when it is waiting at ℓ_{0_1}. If $t_r \geq 1$ it can obviously progress. If $t_r = 0$, then by Lemma 2, there exists an active task Q in r with $h(Q) = 1$. This task can terminate, thereby incrementing t_r, and thus unblocking P. $\square$

Theorem 2. EFFICIENT-P *with height annotation guarantees absence of deadlock.*

Proof. By contradiction, suppose that σ is a reachable deadlock state. Let $P \in I$ be a task in σ such that $h(P)$ is minimal in I. Consider two cases: (1) $h(P) = 1$. By Lemma 5, P can eventually progress, contradicting deadlock. (2) $h(P) > 1$. If P is active, then by Lemma 1 it must have a waiting descendant, contradicting that P has minimal height. If P is waiting, then by Lemma 4 there exists an active task Q with $h(Q) \leq h(P)$. Again $h(Q) < h(P)$ contradicts the minimality of P. If $h(Q) = h(P)$, then Q, being active, must have a waiting descendant by Lemma 1, contradicting the minimality of P. $\square$

Thus, if every call graph is annotated with height and every reactor r is provided with at least as many threads as the maximum height of a node that runs in r, then EFFICIENT-P guarantees deadlock-free operation. The disadvantage of using height as an annotation is that the number of threads to be provided to each reactor can be much larger than is strictly necessary. This not only wastes resources, it may also make some systems impractical, as illustrated by the following example.

Example 6. Consider a system $\mathcal{S} : \langle \mathcal{R}, \{G_1, \ldots, G_m\} \rangle$. A simple way to force invocations of $G_1, \ldots, G_m$ to be performed sequentially is to introduce a new reactor r— called the *serializer*—and to merge $G_1, \ldots, G_m$ into a single call graph by adding a root node n, and making $G_1, \ldots, G_m$ its subtrees. When using EFFICIENT-P with the height annotation, the new node node n is annotated with $h(n) = max(h(G_1), \ldots, h(G_m)) + 1$, which may be large. Now r needs to be provided with this many threads, while one would have sufficed.

Clearly, using height may be wasteful of resources. In the next section we propose a more efficient annotation that addresses this problem.

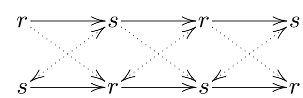
4.4 A More Efficient Annotation

Deadlock is caused by cyclic dependencies. Using EFFICIENT-P with an annotation without cyclic dependencies prevents deadlock. Example 4 showed that the

$G_1: \quad r^3 \longrightarrow s^2 \longrightarrow r^2 \longrightarrow s^1$

$G_2: \quad s^2 \longrightarrow r^2 \longrightarrow s^1 \longrightarrow r^1$

(a) Annotated call graphs G_1 and G_2.



(b) Resulting global call graph.

Fig. 7. Global call graph for two call graphs G_1 and G_2.

deadlock produced with local height as annotation was caused by the interaction of multiple call graphs. Thus, a check for cyclic dependencies must consider the union of all call graphs.

Definition 9 (Global Call Graph). *Given a system* $\mathcal{S} : \langle \mathcal{R}, \{G_1, \ldots, G_m\} \rangle$ *with* $G_i : \langle V_i, E_i \rangle$*, where* V_i, V_j *are assumed to be disjoint for* $i \neq j$*, and annotation function* $\alpha : \bigcup_i V_i \mapsto \mathbb{N}$*, the global call graph for* $\mathcal{S}$ *and* α*,* $G_{\mathcal{S},\alpha} : \langle V_{\mathcal{S}}, E_D, E_A \rangle$ *consists of*

- $V_{\mathcal{S}}$*:* $\bigcup_i V_i$*;*
- E_D*:* $\bigcup_i \rightarrow_i^+$*, the union of the descendant relations of all call graphs, where* $\rightarrow^+$ *is the transitive closure of* E_i*;*
- E_A*:* $\{(v_1 : \langle f : r \rangle, v_2 : \langle g : s \rangle) \mid \alpha(v_1) \geq \alpha(v_2)$ *and* $r = s\}$ *where* v_1 *and* v_2 *may belong to different call graphs* G_i*.*

Example 7. Figure 7(b) shows the global call graph for two annotated call graphs G_1 and G_2, where the solid lines indicate edges in E_D (no composed edges are shown) and the dotted lines indicate edges in E_A.

Definition 10 (Dependency Relation). *Given global call graph* $G_{\mathcal{S},\alpha} : \langle V_{\mathcal{S}}, E_D, E_A \rangle$*,* $v_1 \in V_{\mathcal{S}}$ *is dependent on* $v_2 \in V_{\mathcal{S}}$*, written* $v_1 \succ v_2$*, if there exists a path from* v_1 *to* v_2 *consisting of edges in* $E_A \cup E_D$ *with at least one edge in* E_D*.*

A global call graph has a *cyclic dependency* if for some node v, $v \succ v$.

Theorem 3 (Annotation). *Given a system* $\mathcal{S}$ *and annotation* α*, if the global call graph* $G_{\mathcal{S},\alpha}$ *does not contain any cyclic dependencies, then* Efficient-P *used with* α *guarantees absence of deadlock for* $\mathcal{S}$*.*

Proof. We first observe that, in the absence of cyclic dependencies, the dependency relation $\succ$ is a partial order on the nodes in all call graphs, the proof closely follows that of Theorem 2.

By contradiction, suppose that σ is a reachable deadlock state. Let $P \in I$ be a task in the set of tasks in σ such that P resides in reactor r and is minimal with respect to $\succ$. Consider three cases: (1) P is active. Then, by Lemma 1, P must have a waiting descendant Q, but then $P \succ Q$, contradicting the minimality of P. (2) P is waiting and $\alpha(P) = 1$. Then $t_r = 0$ (otherwise P could proceed, contradicting deadlock), and by Lemma 2, there exists an active task Q in r with annotation 1, and thus there exists an edge in E_A from P to Q. But by Lemma 1, Q has a waiting descendant R, and thus $P \succ R$, contradicting minimality of P. (3) P is waiting and $\alpha(P) > 1$. By Lemma 4, there exists an active task Q with $\alpha(Q) \leq \alpha(P)$, and, as for case (2) there exists a task R such that $P \succ R$, contradicting minimality of P. $\square$

It is easy to see that the conditions posed by Theorem 3 require the annotation to subsume local height. On the other hand, height clearly satisfies the conditions. For many systems, however, the annotation can be significantly lower than height. For example, in Example 6, the serializer node can safely be given annotation 1, instead of the maximum height of all call graphs.

5 Conclusions and Future work

We have formalized thread allocation in DRE systems and proposed several "local" protocols that guarantee absence of deadlock with respect to availability of threads. We have assumed static call graphs, which are the norm in many DRE systems.

These protocols are of practical as well as theoretical interest: they can be implemented (1) *transparently*, *e.g.*, by using the protocols to filter which enabled handles in a reactor's handle set will be dispatched in each iteration of a reactor's event loop [16]; (2) *efficiently*, *e.g.*, by storing pre-computed call graph annotation constants and references to protocol variables for each method in a hash map, to allow constant time lookup at run-time; and (3) *effectively*, *e.g.*, by checking for cyclic dependencies in annotations, as we have done previously for scheduling dependencies between avionics mission computing operations [6].

As future work we will examine optimizations where the protocol can reduce pessimism dynamically at run-time, *e.g.*, upon taking particular branches that require fewer threads than other alternatives. This will involve (1) maintaining a representation of the call graph and its annotations, as objects register and deregister with each reactor at run-time; (2) distributed consistent communication of relevant changes to the graph, annotations, and task progress variables; and (3) re-evaluation of safety and liveness properties at each relevant change.

References

1. Toshiro Araki, Yuji Sugiyama, and Tadao Kasami. Complexity of the deadlock avoidance problem. *2nd IBM Symposium on Mathematical Foundations of Computer Science*, pages 229–257, 1971.
2. Tejasvi Aswathanarayana, Venkita Subramonian, Douglas Niehaus, and Christopher Gill. Design and performance of configurable endsystem scheduling mechanisms. In *Proceedings of 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.
3. Zbignew A. Banaszak and Bruce H. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flow. *IEEE Transactions on Robotics and Automation*, 6(6):724–734, December 1990.

4. Center for Distributed Object Computing. nORB—Special Purpose Middleware for Networked Embedded Systems. `http://deuce.doc.wustl.edu/nORB/`, Washington University.
5. Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
6. Christopher D. Gill, Jeanna M. Gossett, Joseph P. Loyall, Douglas C. Schmidt, David Corman, Richard E. Schantz, and Michael Atighetchi. Integrated Adaptive QoS Management in Middleware: An Empirical Case Study. *Journal of Real-time Systems*, 24, 2005.
7. Arie N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12:373–377, 1969.
8. James W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.
9. Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4:179–196, 1972.
10. Institute for Software Integrated Systems. The ACE ORB (TAO). `http://www.dre.vanderbilt.edu/TAO/`, Vanderbilt University.
11. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaugmann, 1996.
12. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Sytems: Safety*. Springer, 1995.
13. Philip M. Merlin and Paul J. Schweitzer. Deadlock avoidance in store-and-forward networks–I: Store-and-forward deadlock. *IEEE Transactions on Communications*, 28(3), March 1980.
14. Spiridon A. Reveliotis, Mark A. Lawley, and Placid M. Ferreira. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Transactions on Automatic Control*, 42(10):1344–1357, October 1997.
15. Douglas C. Schmidt. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Communications of the ACM Special Issue on CORBA*, 41(10), October 1998.
16. 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.
17. Venkita Subramonian and Christopher D. Gill. A generative programming framework for adaptive middleware. In *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04)*. IEEE, 2004.
18. Venkita Subramonian and Christopher D. Gill. Middleware Design and Implementation for Networked Embedded Systems. In Richard Zurawski, editor, *Embedded Systems Handbook*, chapter 30, pages 1–17. CRC Press, Boca Raton, FL, 2005.
19. Venkita Subramonian, Christopher D. Gill, César Sánchez, and Henny B. Sipma. Composable timed automata models for real-time embedded systems middleware. `http://www.cse.seas.wustl.edu/techreportfiles/getreport.asp?440`, Washington University CSE Department Technical Report 2005-29.
20. Venkita Subramonian, Guoliang Xing, Christopher D. Gill, Chenyang Lu, and Ron Cytron. Middleware specialization for memory-constrained networked embedded systems. In *Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.
21. Ke-Yi Xing, Bao-Sheng Hu, and Hao-Xun Chen. Deadlock avoidance policy for petri-net modeling of flexible manufacturing systems with shared resources. *IEEE Transactions on Automatic Control*, 41(2):289–295, February 1996.