JOURNAL OF COMPUTER AND SYSTEM SCIENCES 23, 254–278 (1981)

Upper Bounds for Static Resource Allocation in a Distributed System*

NANCY A. LYNCH

Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332

Received September 29, 1980; revised January 23, 1981

I. INTRODUCTION

The problem considered in this paper is a simplification of one arising in the study of distributed computer systems. A network is considered, in which are located a large number of "resources" and a large number of potential users of those resources. Each user requires a certain fixed set of resources (for instance, in order to execute a program). Each user's needs are assumed to be known a priori. Users originate requests for their respective sets of resources asynchronously with respect to each other. The problem is to design an algorithm guaranteeing each user eventual control over its resources. One solution to this problem is considered to be better than another if users have shorter waiting times before receiving their resources.

In an actual distributed computer system, there might be several interchangeable instances of some resources. However, one class of allocation strategies would bind each user to a particular (for example, the nearest) instance of each of its resources. The problem of this paper applies to this class of strategies.

It is assumed that there are a very large number of widely distributed users, but that the problem is nevertheless "local" in two senses. First, it is possible to locate the resources in the network in such a way that they are generally "nearby" requesting users. Second, the resource-need pattern is not very highly "connected": for instance, there are not very many resources for each user or users for each resource (at least, not very many compared to the total number which are present in the entire network). Under these two conditions, it seems reasonable that each user's waiting time should not be a function of the size of the entire network, but rather a function of local parameters only (such as the maximum number of resources for each user and the maximum number of users for each resource).

It is not always desirable to centralize control over all of the resources at one location in the network. That location would become a bottleneck, and moreover the delays inherent in long-distance communication would cause waiting time for the

*This research was supported in part by the National Science Foundation under Grants MCS77-15628, MCS78-01689, MCS79-24370 and U.S. Army Research Office Contract DAA29-79-C-0155.

0022-0000/81/050254–25$02.00/0

most distant users to depend on the size of the network. A better strategy in many cases seems to be to locate control of resources at various widely distributed points in the network, near their requesting users, and to have the users engage in a message protocol for securing their resources.

If individual resources are located at different points, one must take care, in designing an allocation algorithm, that deadlock cannot occur: for example, if each of two users requires both of the same two resources, the algorithm should not allow each user to secure one resource and then wait indefinitely for the other resource. A usual solution to this problem is to define a global linear ordering of all of the resources in the network, and insure that each user secures its resources in increasing order according to this ordering.

Although the above general strategy succeeds in preventing deadlock, it does not guarantee a fast solution to the allocation problem of this paper. Consider, for example, a large set of users, $U = \{u_i : 1 \leqslant i \leqslant n\}$ and a correspondingly large set of resources $R = \{r_i : 1 \leqslant i \leqslant n+1\}$. Suppose that the resources are ordered according to their given indices, and each user u_i requires resources r_i and r_{i+1}. Suppose that the order of events which occur is:

$$
\begin{aligned}
&u_n \text{ secures } r_n, r_{n+1},\\
&u_{n-1} \text{ secures } r_{n-1}, \text{then waits for } r_n,\\
&\qquad\vdots\\
&u_1 \text{ secures } r_1, \text{waits for } r_2.
\end{aligned}
$$

Then even after u_n releases its resources, u_1 must wait for u_{n-1}, u_{n-2},..., u_2 in turn to obtain and release their resources. A long "waiting chain" has been set up, of users waiting for each other to obtain and release resources in sequence. The waiting time for a user to obtain its resources can be proportional to the maximum possible length for such a chain, and possibly dependent on the total size of the network (even though there are a maximum of two resources for each user and two users for each resource). Since a solution is required in which the waiting time does not depend on the network size, it is apparent that the linear ordering strategy should be refined. That is, not just any linear ordering yields an efficient solution; criteria are needed for choosing a good linear ordering.

This paper presents one criterion for choosing a good linear ordering of resources, and proves worst-case upper bounds on user waiting time for an algorithm using such an ordering. The upper bounds are independent of network size, and depend only on local parameters.

The particular criterion and resulting algorithm may be of interest in themselves. Roughly similar criteria appear to be somewhat familiar to operating system designers, and the present paper makes appropriate conditions precise. Possibly a more important contribution of this paper, however, is its development of methods of analyzing time complexity of distributed alorithms. The methods used are simple,

tractable and generalizable. They are similar to those already familiar in ordinary sequential complexity theory. (See, for example, [1].)

The major problem until now with carrying out complexity analysis for an asynchronous parallel environment has been the lack of a generally accepted model for the major components of the task: stating the problem, describing the solution, and measuring the complexity. In this paper, we use a simple automata-theoretic model for distributed systems first proposed in [8, 9]. This model is tailored to the purposes of complexity analysis. (The earlier paper contains the basic automata-theoretic definitions and definitions for space measures, while the later paper also contains definitions for time measures.) Time measures used in that work are derived from those defined by Lamport [7] and Peterson and Fischer [11]. These basic tools are used in this paper to state the problem precisely, to describe the solution by showing how a "good linear ordering" of resources can be embedded into a precise algorithm, and to bound the time performance carefully.

Techniques used are quite simple (as compared to queueing theory analysis, for example). Results are obtained to describe worst-case response time and also worst-case response under certain restrictions, such as a light load on the system. In principle, the techniques are extendable to obtain throughput bounds and bounds on average performance, but this paper does not contain any interesting results of these types. The general method of performing the analysis is to find the performance bottlenecks at each point during execution and bound the time necessary for their resolution; this technique has the advantage that it helps the designer to uncover unnecessary bottlenecks. (Two improvements in the algorithm of this paper, leading to a more efficient final version, were in fact discovered in this way.) The methods are suitable for use with modular decomposition of systems into subsystems, allowing combination of analysis of subsystems. Finally, the methods seem appropriate for use in the study of many types of distributed algorithms, such as those for distributed resource allocation, message transmission, and distributed data bases.

II. Resource Problems and an Abstract Solution

In this section, we give formal definitions for the resource problems to be considered, and describe the criterion mentioned in the Introduction for a "good linear ordering" of resources. The problem and criterion are described at an abstract level, without reference to implementing algorithms. Later sections of the paper describe incorporation into an algorithm.

A *resource problem* P is a quadruple $(R(P), U(P), \mathscr{R}(P), \mathscr{U}(P))$, where $R(P)$ and $U(P)$ are disjoint, possibly infinite sets (of *resources* and *users*, respectively), where $\mathscr{R}(P)$ is a mapping from $U(P)$ to the set of finite nonempty subsets of $R(P)$ (indicating the resources required by each user), where $\mathscr{U}(P)$ is a mapping from $R(P)$ to finite nonempty subsets of $U(P)$ (indicating the users for each resource), and where $r \in \mathscr{R}(P)(u)$ if and only if $u \in \mathscr{U}(P)(r)$.

Let $rcommon(P) = \{(u, u') \in (U(P))^2 : \mathscr{R}(P)(u) \cap \mathscr{R}(P)(u') \neq \phi\}$, and *ucom-*

$mon(P) = \{(r, r') \in (R(P))^2 : \mathscr{U}(P)(r) \cap \mathscr{U}(P)(r') \neq \phi\}$. That is, rcommon($P$) describes users of common resources, and ucommon(P) describes resources with common users.

We define a graph, *graph*(P), to represent resource problem P, as follows. Let *graph*(P) be the graph with $R(P)$ as its node set and ucommon(P) as its edge set. (Note that a dual graph with $U(P)$ as the node set and rcommon(P) as the edge set is also a natural representation for P, but this dual representation will not be used in this paper.) Let *contention*(P) denote $\max_{r \in R(P)} |\mathscr{U}(P)(r)|$, the maximum number of users for any resource.

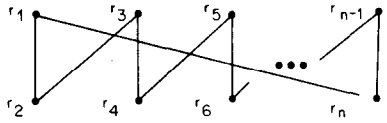
EXAMPLE 2.1. The canonical example used here for a resource problem is the Dining Philosophers problem [4], generalized to any number $n \geqslant 2$ of philosophers. Dijkstra's informally stated problem involves n philosophers sitting in a circle, a single fork between each pair of adjacent philosophers. Any philosopher may decide to eat at any time and requires both of his forks to do so, but he can only "pick up" one fork at a time. Philosophers act asynchronously. The problem is to program the philosophers in ways which guarantee certain conditions of fairness and absence of deadlock, in allowing philosophers to obtain their forks. We can formulate this problem as a resource problem as follows.

Let $R(P) = \{r_1, ..., r_n\}$, $U(P) = \{u_1, ..., u_n\}$. Let $\oplus$ represent addition "in a ring," i.e., $i \oplus j = (i + j - 1) \bmod n + 1$. Let $\mathscr{R}(P)(u_i) = \{r_i, r_{i\oplus 1}\}$ and $\mathscr{U}(P)(r_i) = \{u_{i\oplus(-1)}, u_i\}$.

We motivate our general criterion for linear orderings by considering solutions for the Dining Philosophers problem. Consider first the total ordering $<$ given by "$r_i < r_j$ iff $i < j$." If each philosopher secured his two needed resources in increasing order according to $<$, then a long waiting chain similar to the one described in the Introduction can be created, of u_1 waiting for u_2 waiting for $u_3 \cdots$ waiting for u_n. If this ordering were used in an algorithm, it is reasonable to expect that u_1's worst-case waiting time would be proportional to n.

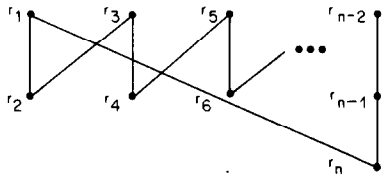
On the other hand, assume n is even and consider the total ordering $\lesssim$ given by "$r_i \lesssim r_j$ iff either (1) or (2) holds: (1) i is odd and j is even, (2) $i < j$ and $i \equiv j \bmod 2$." If each philosopher secures his resources in increasing order according to $\lesssim$, then no waiting chain containing more than three users is ever created. (For example, it is possible for u_2 to wait for u_3 (to obtain r_3), while u_3 is waiting for u_4 (to obtain r_4); however, no longer chain is possible.) This strategy corresponds to alternate philosophers picking up their left and right forks first, respectively (and is apparently a "folklore" strategy [6]).

A way to see the difference between the two total orders $<$ and $\lesssim$ is to view the essential information as a partial rather than a total order. Consider the subrelation of $\lesssim$ represented by the following Haase diagram (with minimal elements at the top):



This subrelation has an edge between any two resources with a common user. Thus, it suffices to specify the same waiting order as $\lesssim$, for each user to secure its resources. (In particular, deadlock is certainly prevented.) But in addition, the maximum length of a waiting chain for the strategy based on this partial order is bounded by (a function of) the depth of the partial order. In the case of the total ordering $<$, no sufficient nontrivial partial suborders exist.

A good solution for the Dining Philosophers problem with an odd number n of philosophers can similarly be based on the partial order



This strategy corresponds to alternate philosophers picking up their left and right forks first, respectively, except that there is one point on the circle where two consecutive philosophers both pick up their left forks first.

These observations can be generalized to arbitrary resource problems as follows. Define a *coloring* of a resource problem P to be a total mapping $c: R(P) \to N$ satisfying the condition "$(r, r') \in \text{ucommon}(P)$ implies $c(r) \neq c(r')$." Let $|c|$ denote the largest number in the range of c. Let *colors*(P) denote the minimum value of $|c|$ for any coloring c of P. (Thus, colors(P) is the chromatic number of graph(P).) Define *first*$_{P,c}: U(P) \to R(P)$ by $\text{first}_{P,c}(u) = r$ such that $r \in \mathscr{R}(P)(u)$ and $(\forall r' \in \mathscr{R}(P)(u))[c(r') \geqslant c(r)]$. Define *next*$_{P,c}: U(P) \times R(P) \to R(P)$ by $\text{next}_{P,c}(u, r) = r'$ such that $c(r) < c(r')$, $r' \in \mathscr{R}(P)(u)$ and $(\forall r'' \in \mathscr{R}(P)(u))[c(r'') \leqslant c(r) \text{ or } c(r'') \geqslant c(r')]$. That is, $\text{first}_{P,c}$ and $\text{next}_{P,c}$ list the resources of each user in increasing order. $\text{First}_{P,c}$ is a total function, while $\text{next}_{P,c}$ is partial. They are well defined since c is injective on $\mathscr{R}(P)(u)$ for each u.

A coloring c of a resource problem P can be used to define a partial ordering $<$ on the resources of P, generated by $\{(r_1, r_2) \in \text{ucommon}(P): c(r_1) < c(r_2)\}$. The strategy in which each user waits for its resources in increasing order according to $<$ certainly avoids deadlock and, moreover, it seems reasonable that this strategy will exhibit waiting times dependent on $|c|$. This strategy is basically the same as the "hierarchical resource allocation" strategy discussed in [2, Sect. 3.5.3].

Thus, a linear ordering would be expected to be "good" if it is an extension of a partial ordering defined by a coloring c of P with $|c|$ equal to (or approximately equal to) colors(P). The second locality condition mentioned in the Introduction, the limit to overlap in resource demands, is captured formally by the bounds contention(P) and colors(P).

Within this formalism, the orderings for Example 2.1 can be determined by the following colorings.

Let

$$c(r_i) = i,$$

and

$$\begin{aligned} c'(r_i) = 1 &\text{ if } n \text{ is even and } i \text{ is odd, or} \\ &\text{if } n \text{ is odd and } i < n \text{ is odd,} \\ = 2 &\text{ if } i \text{ is even,} \\ = 3 &\text{ if } n \text{ is odd and } i = n. \end{aligned}$$

Thus, c provides a linear order for the resources, while v' provides a partial order of depth $\leqslant 3$. In either case, resources with common users are comparable. However, c' is minimum in the sense that

$$\begin{aligned} |c'| = \text{colors}(P) = 2 &\text{ if } n \text{ is even,} \\ = 3 &\text{ if } n \text{ is odd.} \end{aligned}$$

We consider some further examples. The first two are simple generalizations of Example 2.1.

EXAMPLE 2.2. *k-Fork Philosophers.* Let $R(P)$ and $U(P)$ be as in Example 2.1, $\mathscr{R}(P)(u_i) = \{r_i, r_{i\oplus 1}, \ldots, r_{i\oplus(k-1)}\}$ and

$$\mathscr{U}(P)(r_i) = \{u_{i\oplus-(k-1)}, \ldots, u_{i\oplus(-1)}, u_i\}. \qquad \text{Contention}(P) = k.$$
$$\text{Colors}(P) = k+1 \qquad (\text{if } n \geqslant k^2).$$

EXAMPLE 2.3. *2-Dimensional Philosophers.* The resource requirement pattern in a distributed system might not have a 1-dimensional structure such as those of Examples 2.1 and 2.2. As a simplified example of a 2-dimensional pattern, let $R(P) = \{r_i : i = (i_1, i_2) \in Z^2 \text{ and } i_1 + i_2 \text{ is even}\}$, $U(P) = \{u_i : i = (i_1, i_2) \in Z^2 \text{ and } i_1 + i_2 \text{ is odd}\}$, and $\mathscr{R}(P)(u_i) = \{r_j : \sum_{l=1}^{2} |j_l - i_l| = 1\}$. Contention$(P)$ = colors$(P) = 4$. Graph(P) is a diagonal grid.

The remaining two examples will be used later to demonstrate situations in which our algorithm approaches its upper bound; they do not describe "local" resource requirement patterns for which the algorithm is well suited.

EXAMPLE 2.4. *k-Tree.* Let A be an alphabet of k elements, a any distinguished element of A. Let $R(P) = \{r_i : i \in A^*, |i| < k\}$, $U(P) = \{u_i : i \in A^k\}$, and let $\mathscr{U}(P)(r_i)$ consist of all $u_{ia'a^{k-|i|-1}}$ for all $a' \in A$. The resources can be envisioned as forming a tree. For instance, if $k = 3$, $A = \{1, 2, 3\}$ and $a = 1$, the resources form a 3-ary tree, with users as indicated in Fig. 1.

Contention(P) $= k$ and colors(P) $= k$. (Letting $c(r_i) = |i| + 1$ shows colors(P) $\leqslant k$.

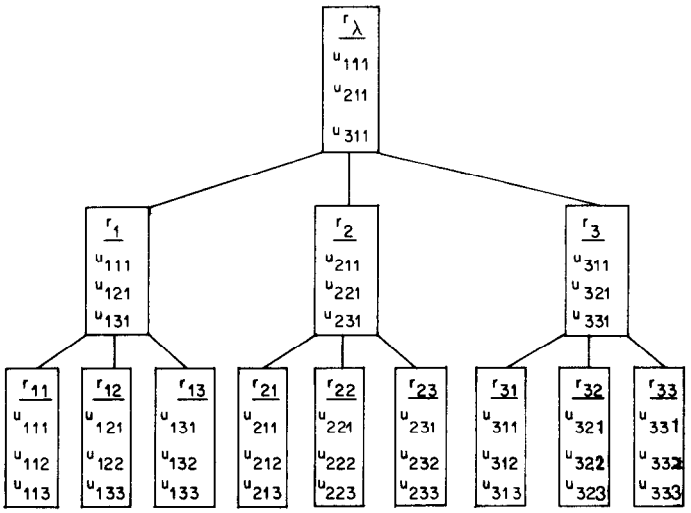


FIGURE 1

Colors(P) cannot be less than k because $|\mathscr{R}(P)(u_{ak})| = k$.) Graph(P) includes the tree as a subgraph.

EXAMPLE 2.5. *k-Nested Sets.* $R(P) = \{r_i : 1 \leqslant i \leqslant k\}$, $U(P) = \{u_i : 1 \leqslant i \leqslant k\}$, and $\mathscr{R}(P)(r_i) = \{u_j : j \leqslant i\}$. Contention(P) = colors(P) = k, and graph(P) is the complete graph on k vertices.

It seems reasonable that one could design an efficient algorithm for a resource problem P, based on the partial ordering constructed from a small coloring of P. However, there is still quite a lot of work remaining to be done in describing a precise algorithm and in proving bounds on its performance. At the very high level of description of this section, many details have been ignored which could have significant impact on the running time of an algorithm. For example, the solution implemented in a network will probably utilize some type of message delivery system. It is likely that message system characteristics such as buffer sizes, message delivery time and message pickup time will affect the algorithm's running time. Queueing policies for the various resources can also be significant, as well as other "implementation details" such as relative order of certain communication and computation steps. One would hope that not too many such details would need to be considered, but until sufficient experience is gained in analyzing distributed systems to know which of these factors are important, it seems that all of these factors must be considered.

The remaining sections of the paper carry out the tasks of formalizing the algorithm and proving upper bounds on its performance.

III. Formulation of the Computational Problem

A Model for Systems of Asynchronous Processes

The formal model used to state the computational problem and describe the algorithm is that of [8, 9]. Only a brief description is provided in this paper; the reader is referred to [8, 9] for a rigorous treatment.

The model is very low level (on the level of Turing machines). Since it is designed for complexity analysis, nothing is hidden which could affect execution costs. In particular, there is no built-in synchronization or queueing. The model is automata-theoretic in style. It is suitable for separate description of interface behavior and implementation of systems, so allows separation of the descriptions of problems and their solutions.

The basic entities are *processes* and *variables* (for communication). An atomic *execution step* of a process involves accessing one variable and possibly changing the process' state or the variable's value or both. A system of processes is a set of processes, with certain of its variables designated as *internal* and the others as *external*. Internal variables are to be used only by the given system. External variables are assumed to be accessible to an "environment" which can change the values between steps of the given system.

The execution of a system of processes is described by a set of *execution sequences*. Each sequence is a (finite or infinite) list of steps which the system could perform when interleaved with appropriate actions by the environment.

For describing the external behavior of a system of processes, certain information in the execution sequences is irrelevant. The *external behavior* of a system of processes is the set of sequences derived from the execution sequences by "erasing" information about process identity, changes of process state and accesses to internal variables. What remains is just the history of accesses to external variables.

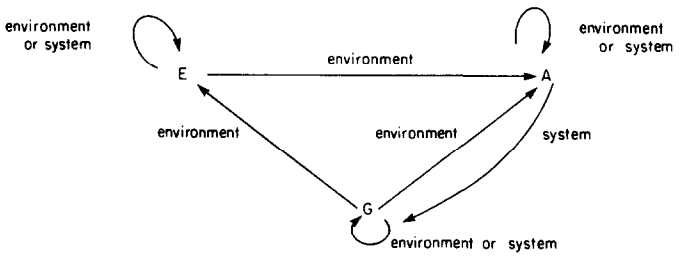
A *distributed problem* is any set of sequences of accesses to variables. A system is said to solve the problem if its external behavior is *any subset* of the given problem. (We do not require that the system exhibit all possible correct sequences, but only that every sequence which it does exhibit is correct.)

One method for specifying a distributed problem, used in the examples of [8, 9], is to describe first the set of correct sequences of accesses to the external variables by the environment and the system together, tagging each access by the label "environment" or "system" as appropriate, second the assumed external behavior of the environment of a system, and third the initialization of the external variables. Then a sequence of system accesses to external variables is *acceptable* provided when it is interleaved consistently with a correct sequence of environmental actions, the resulting sequence is correct for the environment and system together. The distributed problem is the set of acceptable sequences, and a system thus solves the problem if all of its external behavior sequences are acceptable. We follow this method for describing the requirements of our problem.

Interface Description for Resource Problems

Our model is best suited for specification of interface behavior of systems and their components, rather than the "eating and thinking region" behavior described by Dijkstra. Direct formalization of region behavior for arbitrary resource problems does not seem to be particularly natural. Therefore, we formulate the problem in terms of external behavior.

Fix resource problem P. For each $u \in U(P)$, there is an external variable EXT_u, having values 'E' (empty), 'A' (ask) and 'G' (grant). We first describe the correct sequences of accesses to the external variables by the environment and system together. Allowed transitions at any EXT_u are as in the following diagram:



(That is, the environment can ask for resources, the system can grant resources, and the environment can return granted resources by resetting EXT_u to empty (or immediately asking for the resources once again).) If the value of any EXT_u stops changing, then the final value is 'E'. (That is, the system must grant all requests, and the environment must return all resources.) Finally, if two variables, EXT_u and $\text{EXT}_{u'}$, are ever simultaneously equal to 'G', then $(u, u') \notin \text{rcommon(P)}$. (That is, the same resource cannot be simultaneously granted at two different external variables.)

Next, we describe correct external behavior for the environment: a correct environment is one that only makes allowed transitions as described above and which does not leave the variable at 'G'.

All external variables are initialized as 'E'. The set of sequences of system accesses to the external variables which combine consistently with correct environment sequences to yield behavior satisfying the interface description comprises the distributed problem to be solved.

Network Constraints

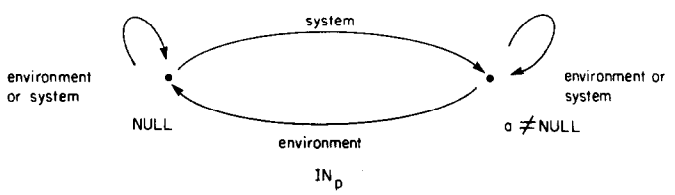
In a distributed environment, there are usually network constraints in addition to interface requirements. These constraints involve number and location of processes, network connectivity and communication time. For the problem at hand, we constrain the solution to consist of *user processes*, one accessing each external variable, and (a disjoint set of) *resource processes*, one for each resource. For notational convenience, these processes are identified with the elements of $U(P) \cup R(P)$. (Of course, one might imagine that several users or resources might be

located at the same place in the network, and might be the responsibility of a single process. It is simpler, and not essentially different, to allow one process for each user and resource, however.)

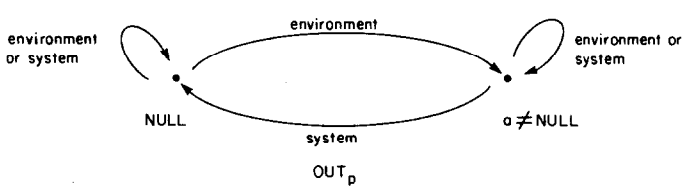
These processes need to communicate, but the means of communication are not the subject of this paper. We would like to suppress as many details as possible about the communication. However, some information about the communication might be important for determining the performance of the algorithm. A compromise which is natural in this model is to specify another interface, of the user and resource processes with a "message system." Then correctness of the complete algorithm will depend on correctness of the message system, and also complexity analysis of the complete algorithm will depend on certain complexity assumptions about the message system.

Thus, an interface is assumed between the user and resource processes and a *message system.* Each process p communicates with the message system by two external variables, IN_p and OUT_p. If M is a message alphabet, then the values of IN_p are $M \cup \{\text{'NULL'}\}$, while the values of OUT_p are $M \times 2^{\mathscr{R}(P)(p)} \cup \{\text{'NULL'}\}$ if $p \in U(P)$ and $M \times (\mathscr{U}(P)(p) \cup \mathscr{R}(P)(\mathscr{U}(P)(p)) \cup \{\text{'NULL'}\}$ if $p \in R(R)$. (Intuitively, the second component of OUT values is an address or set of addresses. User processes can simultaneously broadcast the same message to any subset of the associated resource processes, while resource processes can only send messages one at a time to associated processes. The associated processes for resource processes are the processes of their users, and also the processes of all of the resources of those users.)

We first describe the allowable sequences of accesses to the external variables IN_p and OUT_p by the environment (i.e., the user and resource processes) and the system (i.e., the message system) together. Allowed transitions at any IN_p are in the following diagram:



Allowed transitions at any OUT_p are as in the following diagram:



If the value of any variable stops changing, the final value is 'NULL'. Finally, messages "get delivered"—in any execution sequence, the "writes" by the message system to any IN_p variable must be of message values which are some permutation of the message values "read" (i.e., changed to '*NULL*') by the system from the *OUT* variables, addressed to process p. (We do not specify any order for delivery, nor do we care how the message system operates.)

Next, we describe correct external behavior for the message system's environment: a correct environment is one that only makes allowed transitions and does not leave any IN_p variables $\neq$ '*NULL*'.

All variables are initialized as '*NULL*'. The acceptable system sequences are then defined as before.

The reason for this particular choice of message system interface is that it seems to be the minimum natural interface needed to make our solution work as efficiently as it should. Thus, we have made this interface description part of the given conditions on the problem solution.

The formal correspondence between resources and resource processes is a matter of convenience and is not intended to imply that the resources must be "located at" or "controlled by" the corresponding processes. The space of allowable solutions includes solutions in which control over the granting of a resource is shared by many different resource processes, and solutions in which one resource process controls many resources.

The first locality condition mentioned in the Introduction stated that "it is possible to locate the resources in the network in such a way that they are generally nearby requesting users." We will interpret this condition formally by hypothesizing a small time bound for the restricted communication guaranteed by the message system. Since the message system only guarantees communication with certain associated processes, it is reasonable to suppose that the worst-case time bound for such communication might be independent of the size of the entire network. In order to state this condition formally, it is necessary first to introduce a notion of "time" into the model.

A Time Measure

We base our time measure on those of [7, 10, 11]; a real-valued "time" is assigned to each step of an execution sequence, subject to certain constraints such as upper bounds on the duration of certain events. There may be many distinct allowable assignments of times for each execution sequence. (Note that if no *lower* bounds on duration of events are hypothesized, the set of possible execution sequences of the system is not restricted in any way by the constraints on time.) When an upper bound on time for required system behavior is proved, it must be shown to be an upper bound for all execution sequences and also for all allowable assignments of times for those executions.

Formally, let R^+ represent the nonnegative reals. A *timing* is a nondecreasing total mapping $t: N \to R^+$. (It is intended to represent an assignment of times to successive steps of an execution sequence.) Let P denote a fixed resource problem. Let, $\mathscr{S}$, $\mathscr{M}$,

and $\mathscr{V}$ denote arbitrary implementations within our model of a correct resource allocation system, a correct message system and a correct environment for a resource allocation system, respectively, for P. (Correctness for $\mathscr{M}$ and $\mathscr{V}$ involves interface behavior only, while correctness for $\mathscr{S}$ implies also that its set of processes is $U(P) \cup R(P)$ with external variables accessible to those processes specified in the previous subsection.) Let $\mathscr{C}$ be the system constructed by combining $\mathscr{M}$, $\mathscr{S}$, and $\mathscr{V}$. (In the notation of [8, 9], the combined system is $\text{consist}_{Y,f}(\mathscr{S} \oplus \mathscr{M} \oplus \mathscr{V})$, where

$$Y = \{\text{EXT}_u : u \in U(P)\} \cup \{IN_p : p \in U(P) \cup R(P)\} \cup \{\text{OUT}_p : p \in U(P) \cup R(P)\},$$
$$f(\text{EXT}_u) = \text{'}E\text{'}$$

for all $u \in U(P)$ and $f(\text{IN}_p) = f(\text{OUT}_p) = \text{'NULL'}$ for all $p \in U(P) \cup R(P)$.) Let e denote an execution sequence of $\mathscr{C}$.

Let *sent*(e, i, a, p, p') denote the number of times message 'a' is placed in OUT_p, addressed to process p' (including broadcasts in which p' is included among the addressees), in execution sequence e up to and including step i. Let *collected*(e, i, a, p, p') be the number of times changes are made in variable OUT_p from values in which message "a" is addressed to process p', up to and including step i. Let *sentfrom*(e, i, a, p) denote $\sum_{p'} \text{sent}(e, i, a, p, p')$, and similarly for *collectedfrom*(e, i, a, p). Let *sentfrom*(e, i, p) denote $\sum_a \text{sentfrom}(e, i, a, p)$, and similarly for *collectedfrom*(e, i, p). Let *sentto*(e, i, a, p) denote $\sum_{p'} \text{sent}(e, i, a, p', p)$ and similarly for $\text{collectedto}(e, i, a, p)$. Let *deliveredto*$(e, i, a, p)$ be the number of times a transition to 'a' is made in IN_p, up to and including step i. Let *sentto*(e, i, p) denote $\sum_a \text{sentto}(e, i, a, p)$, and similarly for *collectedto*(e, i, p) and *deliveredto*(e, i, p). Finally, let *sent*(e, i) denote $\sum_p \text{sentto}(e, i, p)$, and similarly for *collected*(e, i) and *delivered*(e, i).

Let *requests*(e, i, u) (resp. *grants*(e, i, u), *returns*(e, i, u)) denote the number of changes to 'A' (resp. to 'G', from 'G') at variable EXT_u in execution sequence e, up to and including step i. If $V \subseteq U(P)$, then let *requests*$(e, i, V) = \sum_{u \in V} \text{requests}(e, i, u)$, and similarly for *grants*(e, i, V) and *returns*(e, i, V). Let *requests* $(e, i) =$ *request*$(e, i, U(P))$ and similarly for *grants*(e, i) and *returns*(e, i).

It is now possible to state the necessary constraints on timings. Let $\mathscr{A} = (\sigma, \nu, \gamma, \delta) \in (R^+)^4$, and let t be a timing. Then t is *$\mathscr{A}$-admissible for e* provided (a)–(d) hold.

(a) *σ is an upper bound on process step time.* Let $p \in U(P) \cup R(P)$ and let the execution steps involving actions of process p be indexed by $p_1, p_2, \ldots$. Then $t(p_1) \leqslant \sigma$ if p_1 exists. Also, $t(p_{i+1}) - t(p_i) \leqslant \sigma$ for each i for which p_i and p_{i+1} are both defined.

(b) *ν is an upper bound on time for a user to return a granted resource.* If $\text{grants}(e, i, u) = k$, if $\text{returns}(e, j, u) = k$ and $\text{returns}(e, j-1, u) < k$, then $t(j) - t(i) \leqslant \nu$.

(c) *γ is an upper bound on message collection time.* If $\text{sentfrom}(e, i, p) = k$, if $\text{collectedfrom}(e, j, p) = k$ and $\text{collectedfrom}(e, j-1, p) < k$, then $t(j) - t(i) \leqslant \gamma$.

(d) *δ is an upper bound on message delivery time.* If sentto$(e, i, a, p) = k$, if deliveredto$(e, j, a, p) = k$ and deliveredto$(e, j-1, a, p) < k$, then $t(j) - t(i) \leqslant \delta$.

The first locality condition mentioned in the Introduction is captured formally by the bound δ. δ is to be thought of as much smaller than the worst-case transmission time for a message system that could send messages between all processes in the entire distributed network.

Note that, in general, some of these parameters might depend on others; for example, γ and δ might be functions of σ, since they might depend on the time taken by a process to pick up a delivered message.

We now define the "worst-case response time" to be measured. Let $T_{\mathscr{C}}(\mathscr{A}, u)$ denote the supremum, for all execution sequences e of combined system $\mathscr{C}$ and all $\mathscr{A}$-admissible timings t for e, of the quantity $t(j) - t(i)$, where requests$(e, i, u) = k$, where grants$(e, j, u) = k$ and grants$(e, j-1, u) < k$. Let $T_{\mathscr{C}}(\mathscr{A})$ denote $\sup_u T_{\mathscr{C}}(\mathscr{A}, u)$.

IV. The General Solution and Its Worst-Case Analysis

General Strategy

We consider solutions in which each resource process maintains a *FIFO* queue of waiting users. It is easy to see that deadlock is prevented in a distributed resource allocation system if the resources are partially ordered by a coloring c of the resource problem P (as defined in Section II), if each user waits on queues for all of its resources in increasing order of resources, if it only waits for one resource at a time (i.e., until reaching the front of the associated queue), and if it remains on all its queues until it is first on all of them. In fact, if all granted resources are eventually returned, it is clear that each user eventually obtains all of its resources.

A High-Level Language

The high-level language used is (almost) the same as that used in [3, 5]. Computation occurring within a lock–unlock pair occurs within a single execution step in the formal model. In the formal model, every step involves access to a variable. The local computation appearing in our language is combined into the previous lock–unlock pair in the formal model. (In [5], this computation was combined into the following lock–unlock pair, an alternative which would change the complexity analysis of our algorithm slightly.) The construct "waitfor (condition);" is an abbrevation for "A: lock; if $\neg$ (condition) then [unlock; goto A];." Subscripts are omitted from EXT, IN and OUT variables.

The Algorithm

The complete code follows. It is a straightforward programming of the strategy already described.

Code for user process u *local*: STATUS init '*E*'

```
do forever
  if STATUS = 'E'
  then [lock; if EXT = 'A' then STATUS := 'A'; unlock]
  else [waitfor (OUT = 'NULL'); OUT := (u, {first_{p,c}(u)}); unlock;
      waitfor (IN = 'G'); IN := 'NULL'; unlock;
      lock; EXT := 'G'; unlock;
      waitfor (EXT ≠ 'G'); STATUS := EXT; unlock;
      waitfor (OUT = 'NULL'); OUT := ('RETURN', ℛ(P)(u)); unlock];
```

STATUS is a flag which has value '*A*' if *u* knows of a pending request to be serviced, '*E*' otherwise. When *u* discovers a request to be serviced, it waits for its OUT variable to become empty, and then sends the identifier *u* to the first resource process, $\text{first}_{p,c}(u)$. Then *u* waits for its IN variable to receive the information that all of *u*'s resources have been granted. It passes this information to the environment, waits for a return and broadcasts a 'RETURN' message to *all* associated resource processes.

Code for resource process r *local*: QUEUE init ∅, MSG init 'NULL', *J* init 'NULL', STATUS init '*E*'

```
do forever
  [lock; if IN ≠ 'NULL' then [MSG := IN; IN :='NULL']; unlock;
  if MSG ∈ U then append MSG to QUEUE;
  if MSG = 'RETURN' then delete front of QUEUE;
  if (MSG ∈ U and |QUEUE| = 1) or (MSG = 'RETURN' and |QUEUE| ⩾ 1)
  then [STATUS := 'A'; J := front(QUEUE)];
  if STATUS = 'A'
  then [lock; if OUT = 'NULL' then OUT := if next_{P,c}(J, r) is defined then
    (J, next_{P,c}(J, r)) else ('G', J); unlock; STATUS := 'E'];
  MSG := 'NULL']
```

r maintains a QUEUE of users awaiting resource *r*, and processes all messages as it receives them. Whenever *r* receives a user identifier message, it appends that identifier to the end of the QUEUE. Whenever *r* receives a 'RETURN' message, it deletes the first element of the QUEUE. Whenever a new user identifier *u* reaches the front of *r*'s QUEUE, *r* sets a STATUS flag to indicate that *r* must send a message for *u*, either requesting *u*'s next resource or granting *u*'s request (in the event that *r* is *u*'s last needed resource). Note that *r* can be delayed in sending this message because its OUT variable is not NULL. In this event, *r* continues picking up and processing its own inputs while waiting to send the message.

It is easy to see that deadlock is avoided by this solution, and that each request eventually gets granted (provided all granted resources are eventually returned). In addition, if $|c|$ is small, this solution appears to limit the lengths of chains of waiting proceses, thereby providing an upper bound on running time. The remaining sections prove results to this effect.

Worst-Case Performance

We obtain a general theorem giving an upper bound on performance of our solution, a bound which is not directly dependent on total network size or total number of users. Let $\mathscr{C}(P, c)$ denote the combined system composed of our solution for resource problem P using coloring c, and any arbitrary correct message system and correct environment for a resource system. Let $\mathscr{A} = (\sigma, \nu, \gamma, \delta)$.

THEOREM 1. $T_{\mathscr{C}(P,c)}(\mathscr{A}) \leqslant (\text{contention(P)}^{|c|} - 1)\ \nu + 0(|c|\ \text{contention(P)}^{|c|}(\sigma + \gamma + \delta))$.

Proof. Since processes operate asynchronously, it is generally the case during execution that some parts of the system are waiting for work to be accomplished by other parts. (The waiting processes might be busy-waiting, or might be performing a considerable amount of work.) In the analysis, it is crucial that the key parts of the system be identified at each time during execution.

The general analysis strategy is quite straightforward. The details, however, can be slightly tricky. This is because an asynchronous system provides no guarantees on how long it takes even "trivial" events to occur; thus, one must be careful to bound the time for each necessary event explicitly. For this reason, we provide a detailed analysis.

Classify the resource processes into *levels*, each resource process r at level $c(r)$. For $1 \leqslant i \leqslant |c|$, $\leqslant j \leqslant$ contention(P), let $G_{i,j}$ denote the supremum, over all execution sequences e and $\mathscr{A}$-admissible timings t, of the time from when the identifier for any user u reaches position j from the front of a level $\geqslant i$ resource process QUEUE, until the resources are next granted to u. A system of recurrences is obtained.

First, consider arbitrary i and $j \geqslant 2$. Let a_1 be an execution step in which u reaches position j from the front of the QUEUE of a level $\geqslant i$ process, r. Let u' be the immediate predecessor of u on r's QUEUE. By induction, within time at most $G_{i,j-1}$ from $t(a_1)$ (as measured by timing t), the environment at u' is granted its resources. Then within time at most ν, the environment returns the resources (because of $\mathscr{A}$-admissibility), and then within time σ, user process u' detects the return. Also, within time γ of $t(a_1)$, the value $(u', \{\text{first}_{P,c}(u')\})$ arising from this u' request is removed from $\text{OUT}_{u'}$. Thereafter, within time σ, u' broadcasts a 'RETURN' message, and then within δ the 'RETURN' reaches IN_r. Within σ, the 'RETURN' is read by r and u' is removed from r's QUEUE, making u first. Then within $G_{i,1}$, u is granted its resources. We see that $G_{i,j} \leqslant \max(\gamma, G_{i,j-1} + \nu + \sigma) + 2\sigma + \delta + G_{i,1}$, for $j \geqslant 2$.

Next, consider $i \geqslant 2$ and $j = 1$. Consider an execution step in which u reaches the front of the QUEUE of a level $\geqslant i$ process, r. Within time $\gamma + 2\sigma$, a value ('G', u) or $(u, \text{next}_{P,c}(u, r))$ is placed in OUT_r. If the value is ('G', u), then within $\delta + 2\sigma$, the environment at u is granted its resources. If the value is $(u, \text{next}_{P,c}(u, r))$, then within $\delta + 2\sigma$, u is appended to the QUEUE of a level $\geqslant i + 1$ process, r'. At that moment, u is in position $\leqslant$ contention(P) + 1 on the QUEUE of r'; each contender for resource r' can appear at most once, with the single exception that the first user on the r' QUEUE might appear twice. (This is because the first user might have its resources

granted, return them, and then request them again. The new request might arrive at r' before the 'RETURN' message.) However, within time $\delta + \sigma$, u reaches a position $\leqslant$ contention(P) on the r' QUEUE. Then within $G_{i+1,\,\text{contention(P)}}$, (the environment at) u is granted its resources. We see that $G_{i,1} \leqslant \gamma + 2\sigma + \max(\delta + 2\sigma,\ \delta + 2\sigma + \delta + \sigma + G_{i+1,\,\text{contention(P)}})$, $i < |c|$. That is, $G_{i,1} \leqslant \gamma + 2\delta + 5\sigma + G_{i+1,\,\text{contention(P)}}$, for $i < |c|$.

Next, it is easy to see that $G_{|c|,1} \leqslant \gamma + 2\sigma + \delta + 2\sigma = \gamma + \delta + 4\sigma$.

Finally, consider $T_{\mathscr{C}(P,c)}(\mathscr{A})$. Let a_1 be an execution step in which (the environment at) u makes a request. Within time σ from $t(a_1)$, the request is detected by user process u. Also, within time γ from $t(a_1)$, OUT_u becomes 'NULL'. Thus, within time $\max(\gamma, \sigma) + \sigma$ of $t(a_1)$, the value $(u, \{\text{first}_{P,c}(u)\})$ is placed in OUT_u. Then (as above) within $\delta + 2\sigma + \delta + \sigma$, u reaches position $\leqslant$ contention(P) on some QUEUE. Thus, $T_{\mathscr{C}(P,c)}(\mathscr{A}) \leqslant \max(\gamma, \sigma) + \sigma + \delta + 2\sigma + \delta + \sigma + G_{1,\,\text{contention(P)}}$. That is, $T_{\mathscr{C}(P,c)}(\mathscr{A}) \leqslant \max(\gamma, \sigma) + 2\delta + 4\sigma + G_{1,\,\text{contention(P)}}$.

To summarize the inequalities, let σ' denote $\sigma + \gamma + \delta$. Then for some constant k, we have $G_{i,j} \leqslant k\sigma' + v + G_{i,j-1} + G_{i,1}$ for $j \geqslant 2$, $G_{i,1} \leqslant k\sigma' + G_{i+1,\,\text{contention(P)}}$ for $i < |c|$, $G_{|c|,1} \leqslant k\sigma'$, and $T_{\mathscr{C}(P,c)}(\mathscr{A}) \leqslant k\sigma' + G_{1,\,\text{contention(P)}}$. Letting a denote contention(P), we have $G_{i,j} \leqslant (j-1)(k\sigma' + v) + j\,G_{i,1}$ for i, j, and so $G_{i,a} \leqslant (2a-1)(k\sigma') + (a-1)\,v + a\,G_{i+1,a}$ for $i < |c|$. Also, $G_{|c|,a} \leqslant (2a-1)(k\sigma') + (a-1)\,v$. Thus, $G_{1,a} \leqslant (1 + a + a^2 + \cdots + a^{|c|-2})((2a-1)(k\sigma') + (a-1)(v)) + a^{|c|-1}G_{|c|,a} = (1 + a + a^2 + \cdots + a^{|c|-1})((2a-1)(k\sigma') + (a-1)(v))$. Then $T_{\mathscr{C}(P,c)}(\mathscr{A}) \leqslant k\sigma' + (1 + a + a^2 + \cdots + a^{|c|-1})((2a-1)(k\sigma') + (a-1)(v)) \leqslant 2k\,|c|\,a^{|c|}\sigma' + (a^{|c|} - 1)v$, as required. ∎

Since we do not hypothesize any lower bounds on time for events to occur, there is no limit on the *number* of times competing users can get ahead of a particular user. However, Theorem 1 shows that the only way large numbers of processes can get ahead is by going fast; there is still a limit on the total *time* any particular user waits.

COROLLARY 1. *For some* c, $T_{\mathscr{C}(P,c)}(\mathscr{A}) \leqslant (\text{contention(P)}^{\text{colors(P)}} - 1)\,v + 0(\text{colors(P)} \cdot \text{contention(P)}^{\text{colors(P)}}(\sigma + \gamma + \delta))$.

EXAMPLE 4.1. *Dining Philosophers.* Recall the colorings c and c' from Example 2.1. c yields a worst-case running time of $(2^n - 1)\,v + 0(n2^n(\sigma + \gamma + \delta))$, while c' yields the much better running time $7v + 0(\sigma + \gamma + \delta)$. Intuitively, the ordering yielded by c allows length n waiting chains to form, but c' does not allow chains of length greater than 3.

EXAMPLE 4.2. *k-Fork Philosophers.* A worst-case bound of $(k^{k+1} - 1)\,v + 0(k^{k+2}(\sigma + \gamma + \delta))$ is obtained. If, however, the worse coloring $c(r_i) = i$ is used, one obtains a bound of $(k^n - 1)\,v + 0(nk^n(\sigma + \gamma + \delta))$.

EXAMPLE 4.3. *2-Dimensional Philosophers.* The bound is $0(v + \sigma + \gamma + \delta)$.

EXAMPLE 4.4. *k-Tree.* The bound is $(k^k - 1)\nu + 0(k^k(\sigma + \gamma + \delta))$.

EXAMPLE 4.5 *k-Nested Sets.* The bound is $(k_k - 1)\nu + 0(k^k(\sigma + \gamma + \delta))$.

V. REALIZING THE UPPER BOUND

It is not always clear how to produce "bad" execution sequences and "bad" $\mathscr{A}$-admissible timings for which the bound derived in Theorem 1 is (approximately) realized. For instance, it does not seem possible to exhibit exponential dependence on n in Example 4.1 (Dining Philosophers). In this section, we sketch how to construct bad execution sequences and timings for k-Trees, k-Nested Sets and k-Fork Philosophers. In Section VI, we prove an alternative upper bound theorem which implies that such bad execution sequences and timings cannot be constructed for Example 4.1.

EXAMPLE 5.1. *k-Tree.* Consider an execution sequence for a request from user u_{a^k} in which, whenever a user $y_{ia^{k-|i|}}$ arrives on the QUEUE for a resource r_i, all of the other contenders for r_i have just arrived very shortly before. This execution involves u_{a^k} waiting for $k^k - 1$ distinct other users to obtain (sequentially) their resources. Thus, if a timing is constructed to maximize waiting times, a response time of at least $(k^k - 1)\nu$ is realized. Note that whenever the specified users arrive on the specified QUEUEs, it is possible that the other contenders can all arrive as required. This is because these other contenders are only being required to arrive at their first resources, which they can do independently.

EXAMPLE 5.2. *k-Nested Sets.* Let $c(r_i) = i$, $1 \leqslant i \leqslant k$. Construct an execution sequence for a request for user u_1 in which whenever a user u_i arrives on the QUEUE for a resource r_j, $j > i$, it is the case that u_j has just arrived very shortly before. This execution involves u_1 waiting for $2^{k-1} - 1$ other requests to be granted. Unlike Example 5.1, however, many of these requests are from the same users. (For instance, for $k = 5$, the order of granted requests is given by $u_5u_4u_5u_3u_5u_4u_5u_2u_5u_4u_5$ $u_3u_5u_4u_5u_1$.) Thus, if a timing is constructed to maximize waiting times, a response time of at least $(2^k - 1)\nu$ is realized. Again, the required arrivals are possible because we are only requiring contenders to arrive at their first resources. There is sufficient independence in the operation of the system to allow the arrivals to occur.

In both of these examples above, a permutation of the values of c will make it impossible to construct execution sequences and timings with exponential dependence on k. (For instance, for Example 5.2, simply reversing the order of the resources will make the dependence on k quadratic, as we will show in Section VI.)

Neither of the examples above is of the "local" type for which this algorithm is intended. However, one can easily construct an example with a "local" flavor which approaches the upper bound by patching together multiple instances of Examples 5.1 or 5.2.

EXAMPLE 5.3. *k-Fork Philosophers.* Let $c(r_i) = i$, $1 \leqslant i \leqslant n$, as in Example 4.3.

We construct an execution sequence for a request of u_1, using only $u_1,\ldots,u_{n-k+1}$. Whenever a user u_i, $1 \leqslant i \leqslant n-k$, arrives on the QUEUE for a resource r_j, $i < j \leqslant n-k+1$, it is the case that u_j has just arrived. Then, for example, if $k=3$ and $n=7$, the order of granted requests is $u_5u_4u_5u_3u_5u_4u_2u_5u_4u_5u_3u_1$. In general, if $f(k, n)$ is the number of requests for which u_1 waits, then one can calculate $f(k, n)$ as follows.

Consider a slightly modified resource problem P' having $R(P') = \{r_1,\ldots,r_n\}$, $U(P') = \{u_1,\ldots,u_n\}$, and $\mathscr{U}(P')(r_i) = \{u_j : i-k+1 \leqslant j \leqslant i\}$. (Thus, modular arithmetic is eliminated and so the first few resources have fewer than k users if $k > 1$). Let $c(r_i) = i$. We construct an execution sequence for a request of u_1: whenever u_i, $1 \leqslant i \leqslant n-1$, arrives on the QUEUE for a resource r_j, $i < j \leqslant n$, it is the case that u_j has just arrived. If $g(k, n)$ denotes the total number of requests granted in this execution up to and including the initial u_1 request, then $f(k, n) = g(k, n-k+1) - 1$ for $n \geqslant 2k-1$. Then we can see that $g(k, 1) = 1$, $g(k, n) = \sum_{i=1}^{n-1} g(i, i) + 1$ for $n \leqslant k$, and $g(k, n) = \sum_{i=n-k+1}^{n-1} g(k, i) + 1$ for $n > k$. This Fibonacci-style bound shows that $f(k, n)$ is $\Omega(k^{n/k})$, so that a response time of $\Omega(k^{n/k})(v)$ is realized.

VI. A Special Case

A case analysis for the coloring c of the Dining Philosophers Example 2.1 shows (in contrast with Example 5.3) that no execution exhibiting exponential dependence on $|c|$ is possible. Also, for example, changing only the ordering of the colors in Example 5.2 changes the dependence on $|c|$ from exponential to quadratic. Thus, while Theorem 1 yields the required independence of network size, it does not tell the entire story.

Theorem 1 allows for the possibility that a user will have to wait for the maximum number of competing processes on each queue. However, if two users contend for two different resources, then neither will ever have to wait for the other for the second of the two resources. Moreover, Theorem 1 does not take into account any special limitations on the resources needed by any particular user. The second theorem takes these factors into account.

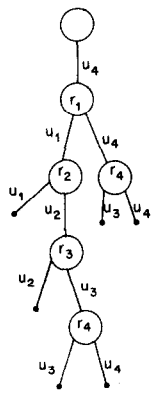
Define a tree *waittree*(P, c, u) for a resource problem P, a coloring c and $u \in U(P)$ as follows.

(1) *pretree*(P, c, u)

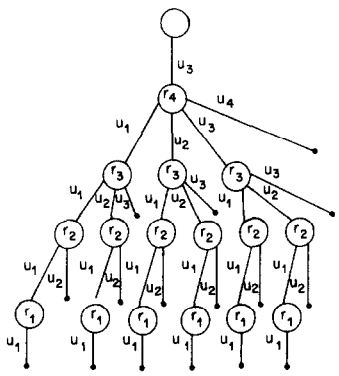
The root node has a single son labelled by the resource $\text{first}_{P,c}(u)$. The edge joining the root node to this son is labelled by u.

For any node x labelled by any r, and any $u' \in \mathscr{U}(P)(r)$ with $\text{next}_{P,c}(u', r)$ defined, there is a son of node x labelled by $\text{next}_{P,c}(u', r)$. If $\text{next}_{P,c}(u', r)$ is undefined, there is a son of node x which is a dummy node. In either case, the edge joining x to this son is labelled by u'.

EXAMPLE 6.1. *Dining Philosophers.* Let $n = 4$. Pretree(P, c, u_4) is as follows.



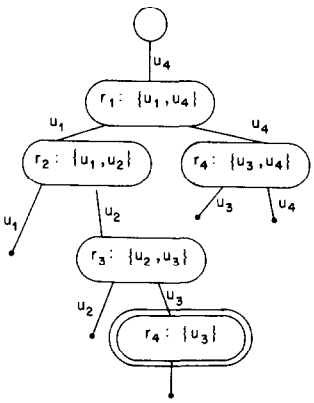
EXAMPLE 6.2. *k-Nested Sets.* Let $k = 4$ and $c'(r_i) = 5 - i$. Pretree(P, c', u_3) is as follows.



(2) *waittree*(P, c, u)

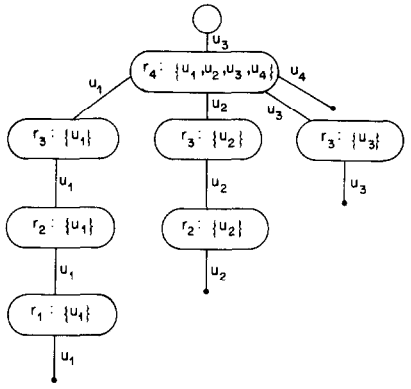
The non dummy, nonroot, nodes of pretree(P, c, u) are assigned tags consisting of sets of processes. If node x has label r, then the tag, A_x, for x is the set of all $u' \in \mathscr{U}(P)(r)$ with the following property. For all ancestors y of x, where y is labelled by r', if $u' \in \mathscr{U}(p)(r')$ then u' labels the out-edge leaving y in the direction of node x. The resulting tagged tree is then pruned so that the only edges leaving any node are those labelled by processes included in the tag of that node.

EXAMPLE 6.3. *Dining Philosophers.* Waittree(P, c, u_4), for Example 6.1, is as follows:



The only user omitted from a tag is u_4, omitted from the tag of the double circled node. u_4 is omitted because $u_4 \in \mathscr{U}(P)(r_1)$ and u_4 does not label the edge leaving the son of the root in the direction of the double circled node. Only one (dummy) node is pruned from the tree.

EXAMPLE 6.4. *k-Nested Sets.* Waittree(P, c', u_3) is as follows.



A considerable amount of pruning occurs for this tree.

Let *subtree*(P, c, u, x, a), where $a \in A_x$, denote the subtree of waittree(P, c, u) which has root x, a single edge e leaving x toward the leaves with label a, and contains all edges of waittree(P, c, u) below e. Let *weight*(P, c, u, x, a) denote the number of edges in subtree(P, c, u, x, a). Let *weight*(P, c, u, x, B), where $B \subseteq A_x$, denote $\sum_{a \in B}$ weight(P, c, u, x, a). Let *weight*(P, c, u, x) denote weight(P, c, u, x, A_x) and let *weight*(P, c, u) denote weight(P, c, u, x), where x is the son of the root.

THEOREM 2. $T_{\mathscr{C}(P,c)}(\mathscr{A}, u) = 0$ ((weight$(P, c, u)) \cdot (\sigma + \nu + \gamma + \delta)$).

Proof. Some additional vocabulary is required. Note that each request generates a

set of QUEUE entries which persist until 'RETURN' messages are received. A QUEUE entry is said to be *active* at step i of execution sequence e provided the request which generated it has not yet been granted at (i.e., immediately after) step i. The following is obvious.

Claim 1. At any step i of an execution sequence e, any user u' who has an entry active on the QUEUE for any resource r also has an entry which is first on the QUEUEs for all $r' \in \mathscr{R}(P)(u')$ with $c(r') < c(r)$.

We next relate waittree(P, c, u) to the execution of the algorithm for a request for user u.

Step i of execution sequence e is *consistent* with node x of waittree(P, c, u) provided each user which labels an edge above x has an entry which is active on the QUEUE of the resource labelling the lower (son) endpoint of that edge, at (i.e., immediately after) step i. (There can be at most one such active entry for each QUEUE.) The following claim shows how the pruning of pretree(P, c, u) carried out in defining waitree(P, c, u) preserves a sufficient portion of the tree to describe situations arising in actual execution.

Claim 2. Let x be a node of waittree(P, c, u) labelled by resource r. Assume step i of execution sequence e is consistent with x. Let u' be a user with an active entry on r's QUEUE at step i. Then $u' \in A_x$.

Proof of Claim 2. Assume $u' \notin A_x$. Then by definition of A_x, there is an ancestor y or x, labelled by a resource r', with $u' \in \mathscr{U}(P)(r')$ and u'' the label of the outedge leaving y in the direction of node x, $u'' \neq u'$. By the definition of consistency, u'' has an entry active on the QUEUE for some resource with a higher number than $c(r')$, at step i. By Claim 1, u'' has an entry which is first on the QUEUE of r' at step i. However, Claim 1 also implies that u' has an entry which is first on the QUEUE of r' at step i, a contradiction.

The next claim, proved inductively, describes the relationship between the sizes of certain unions of subtrees of waittree(P, c, u), and the time until the granting of certain requests. Write $\sigma' = \sigma + \nu + \gamma + \delta$.

Claim 3. Let x be a node of waittree(P, c, u) labelled by resource r. Let u' be the label of the edge immediately above x. Assume step i of execution sequence e is consistent with x, and that u'' is a user having an active entry a (not necessarily proper) predecessor of the active entry for u' on r's QUEUE at step i. Let B be the set of users having active entries which are predecessors of this entry of u'' (including u'' itself) on r's QUEUE at step i. (By Claim 2, $B \subseteq A_x$.) Let j be the step at which the request of u'' which generated this active entry is granted. Let t be an $\mathscr{A}$-admissible timing for e. Then $t(j) - t(i)$ is $0((\text{weight}(P, c, u, x, B))(\sigma'))$.

Proof of Claim 3. We use induction on the nodes x of waittree(P, c, u), starting at the lowest nodes and working towards the root. For each node x, we use induction on subsets B of A_x, ordered by containment. Assume e, i, x, r, u', u'' and B are as above.

There are three cases.

(1) The given active entry of u'' is the first active entry on r's QUEUE at step i, and $\text{next}_{P,c}(u'', r)$ is undefined.

Then since u'' has reached the front of the QUEUE for its last needed resource, the request of u'' is granted within time $0(\sigma')$, as needed.

(2) The given active entry of u'' is the first active entry on r's QUEUE at step i, and $\text{next}_{P,c}(u'', r) = r'$.

Then within time $0(\sigma')$, a step i' is reached at which the same request of u'' has generated an active entry on the QUEUE for r'. Then step i' is consistent with node y, where y is the son of x reached by following the edge labelled by u''. Thereafter, by induction on nodes and by Claim 2, the time until the request of u'' is granted is $0((\text{weight}(P, c, u, y))(\sigma'))$. The total time is, therefore, $0((\text{weight}(P, c, u, y) + 1)(\sigma'))$, as needed.

(3) The first active entry on r's QUEUE at step i is generated by $u''' \neq u''$.

Then by induction on subsets of A_x, within time $0((\text{weight}(P, c, u, x, \{u'''\}))(\sigma'))$, a step i' is reached at which the request of u''' is granted, so that the given u''' entry is inactive at (i.e., immediately after) step i'. Step i' is still consistent with x, and $B' = B - \{u'''\}$ is the set of users having active entries which are predecessors of the given entry of u'' at step i'. Thereafter, by induction on subsets, the request of u'' is granted within time $0((\text{weight}(P, c, u, x, B'))(\sigma'))$. The total time is, therefore, $0((\text{weight}(P, c, u, x, \{u'''\}) + \text{weight}(P, c, u, x, B'))(\sigma')) = 0((\text{weight}(P, c, u, x, B))(\sigma'))$, as needed.

Thus, Claim 3 is true. Now, consider any request of u. Within time $0(\sigma')$ of initiation, a step i is reached at which u obtains an active entry on the QUEUE for $\text{first}_{P,c}(u)$. Step i is consistent with the son of the root of waittree(P, c, u). Claim 3 yields the result. ∎

EXAMPLE 6.5. *Dining Philosophers.* Generalizing Example 6.3, we see that coloring c provides a running time of $0(n(\sigma + \nu + \gamma + \delta))$, because of the size of the waittrees. This is in contrast to the exponential bound of Example 5.3.

EXAMPLE 6.6. *k-Nested Sets.* Generalizing Example 6.4, we see that the coloring c' provides a running time of $0(n^2(\sigma + \nu + \gamma + \sigma))$. This is in contrast to Example 5.2.

There are, of course, cases in which Theorem 2 does not provide improvement over Theorem 1. For example, for a k-tree, the waittree for coloring c of Example 2.4 and user u_{a^k} follows the structure of the k-tree itself. (See the example for $k = 3$ in Example 2.4.) Thus, the waittree has more than k^k edges. We also note that the given upper bound proportional to the size of the waittree cannot always be realized; by ad hoc arguments, it is often possible to eliminate still more waiting possibilities.

VII. Remarks on Quantities Other Than Worst-Case Response Time

One might be interested in measures other than worst-case response time. For example, one can obtain an upper bound on "worst-case throughput" by measuring the rate at which requests are granted, assuming that each user always initiates a new request within some time ε after the preceding request is returned. In outline, if $\mathscr{A} = (\sigma, \nu, \gamma, \delta, \varepsilon) \in (\mathscr{R}^+)^5$, then a timing is *$\mathscr{A}$-admissible* provided σ, ν, γ, and δ are as before and, in addition, the first request of each user is within ε of the beginning, and each subsequent request is within time ε of the previous return by that user. Then let $T'_{\mathscr{C}}(\mathscr{A})$ denote the supremum of the quantity $\limsup_{i\to\infty} t(i)/\text{grants}(e, i)$ for all execution sequences e of $\mathscr{C}$ and all $\mathscr{A}$-admissible timings t for e. An easy corollary to Theorem 1 says that $T'_{\mathscr{C}(P,c)}$ is

$$0 \left(\frac{|c|\ \text{contention}(\mathrm{P})^{|c|}(\sigma + \nu + \gamma + \delta) + \varepsilon}{n}\right) \qquad \text{if} \quad |U(P)| = n.$$

Another interesting measure for many distributed algorithms is worst-case performance under assumptions of limited concurrency. Intuitively, if at most k requests are concurrent with a given request, then worst-case response time for the given request might be better than worst-case with unlimited concurrency, for small values of k. Analysis techniques for deriving such bounds are somewhat different from those used for Theorems 1 and 2.

For the problem of this paper, it is probably not appropriate to consider limitations on concurrent requests throughout the network. Since the problem has a local flavor, it might be more appropriate to seek a worst-case bound in the presence of at most k concurrent "nearby" requests. For $u \in U(P)$, define *pred*(u) to be the set of users appearing as edge labels in waittree(P, c, u). (Thus, pred(u) represents all users which could delay the granting of a request of u, together with u itself.) Let $\mathscr{A} = (\sigma, \nu, \gamma, \delta)$ and use the definition of $\mathscr{A}$-admissibility in Section III. Let $T''_{\mathscr{C}}(a, u, k)$ denote the supremum, for all execution sequences e and $\mathscr{A}$-admissible timings t, of the quantity $t(j) - t(i)$, where u makes a request at step i which is granted at step j, and where requests$(e, j, \text{pred}(u)) \leqslant k + \text{returns}(e, i, \text{pred}(u))$. That is, there are at most k requests involving users in pred(u) active in the interval from i to j. It is not difficult to verify the following claim about our system $\mathscr{C}(P, c)$: if at any step of any execution sequence there is a request of user u pending, and if there is no request of a user in pred(u) currently granted, then within time $0(|c|(\sigma + \gamma + \delta))$, *some* request by a user in pred(u) gets granted. Therefore, if there is a bound of k on such requests, the total time to grant the request of u is $(k-1)\,\nu + 0(k|c|(\sigma + \gamma + \delta))$.

A refinement on the analysis outlined above might attempt to use the fact that the message system might also be guaranteed to perform at better than its worst-case performance under the limited usage deducible from the given limit on concurrent nearby user requests. The message system is performing a significant part of the work of the entire system, and its improved performance under light usage conditions might be expected to have a significant impact on the calculated bounds. In order to obtain

such a sharpened analysis, one needs to include more detailed bounds on the behavior of the message system in the admissibility vector, rather than just γ and δ. Let $\mathscr{A} = (\sigma, \nu, \gamma, \delta, \mu) \in (R^+)^5$, and redefine a timing to be *$\mathscr{A}$-admissible* for an execution sequence e provided σ, ν, γ, and δ are as in Section 3, and μ satisfies the following. For all $k \geqslant 1$ and all p, if $\text{sentto}(e, j, p) \leqslant k + \min(\text{deliveredto}(e, i, p), \text{collectedto}(e, i, p))$ and if for no l, $i < l < j$ is the case that $\text{sentto}(e, l, p) = \text{collectedto}(e, l, p) = \text{deliveredto}(e, l, p)$, then $t(j) - t(i) \leqslant k\mu$. That is, we bound the length of time taken to collect and deliver at most k messages to a single process p.

For simplicity, we assume a bound of the form $k\mu$, where μ is to be thought of as much smaller than γ and δ. Let $T'''_{\mathscr{C}}(\mathscr{A}, u, k)$ be defined to be the same as $T''_{\mathscr{C}}(\mathscr{A}, u, k)$, except that the new definition of $\mathscr{A}$-admissibility is used. We require another definition and a lemma in order to analyze our system $\mathscr{C}(P, c)$. For $u \in U(P)$, define *res*(u) to be the set of resources appearing as node labels in waittree(P, c, u). (Thus, res(u) represents all resources which could delay the granting of a request of u.)

LEMMA 7.1. *If* $r \in \text{res}(u)$ *and* $u' \in \mathscr{U}(P)(r)$, *then* $u' \in \text{pred}(u)$.

Proof. Consider particular $r \in \text{res}(u)$, $u' \in \mathscr{U}(P)(r)$. Let y be any node of waittree(P, c, u) labelled by r.

Let x be the highest node on the path from the root of waittree(P, c, u) to y such that $\text{label}(x) \in \mathscr{R}(P)(u')$. Then $u' \in A_x$, be definition of A_x. ∎

$T'''_{\mathscr{C}(P,c)}(\mathscr{A}, u, k)$ can be bounded as follows. First, there may be an initial interval of $0(\sigma + \gamma + \delta)$ before all old 'RETURN' messages from nonconcurrent requests have been collected, delivered and processed. We analyze the remaining interval I until u's request is granted. First imagine that all messages overlapping I which are addressed to users in pred(u) or to resources in res(u) take time zero until collection and delivery. With this assumption, the time for I is at most $(k-1)\nu + 0(k|c|\sigma)$. (The analysis is the same as that for $T''_{\mathscr{C}(P,c)}(\mathscr{A}, u, k)$.) We must add to this bound the total time taken for all the relevant messages to be collected and delivered, which we calculate as follows.

The given requests concurrent with the original request of u produce a set of at most $k(2|c| + 1)$ messages, all addressed either to users in pred(u) or to resources in res(u). Moreover, *all* messages which overlap interval I and are addressed to these users and resources are among this set of messages. This is because the only messages ever sent to a user process during the execution of the algorithm are 'G' messages originating from its own requests, and also the only messages ever sent to a resource process are 'RETURN' messages and requests from its users and from lower numbered resources of its users. But using Lemma 7.1, we see that all of these messages must be generated by requests from users in pred(u). Because I does not begin until the initial interval has elapsed, these requests must all be among the given concurrent requests.

Now consider any particular destination process p in pred(u) or res(u), and assume that l of the $\leqslant k(2|c| + 1)$ messages above are addressed to p. By admissibility, the total time for the l messages to p is at most $l\mu$. Summing over all of the processes in

pred$(u) \cup$ res(u) yields a total message time of at most $k(2|c|+1)\mu$. Thus, the total time is at most $(k-1) + 0(\sigma+\gamma+\delta) + 0(k|c|(\sigma+\mu))$.

ACKNOWLEDGMENTS

This work is part of two larger projects—a joint project with Professor Michael J. Fischer on theory of asynchronous systems and a Georgia Tech project for design of fully distributed processing systems. Thanks for many ideas and discussions go to the members of both projects, especially Mike Fischer, Jim Burns and Nancy Griffeth.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass. June 1976.
2. P. BRINCH-HANSEN, "Operating System Principles," Prentice–Hall, Englewood Cliffs, N.J., 1973.
3. A. B. CREMERS AND T. N. HIBBARD, "Arbitration and Queueing Under Limited Shared Storage Requirements," University of Dortmund Technical Report, 1979.
4. E. W. DIJKSTRA, Hierarchical ordering of sequential processes, *Acta Inform.* **1** (1971), 115–138.
5. M. FISCHER, N. LYNCH, J. BURNS, AND A. BORODIN, Resource allocation with immunity to limited process failure, *in* "Proceedings, Annual Symposium on Foundations of Computer Science," pp. 234–254, 1979.
6. D. HAREL, On folk theorems, *Comm. ACM* **23**, No. 7 (July 1980), 379–389.
7. L. LAMPORT, Private communication.
8. N. LYNCH AND M. FISCHER, On describing the behavior and implementation of distributed systems, *in* "Semantics of Concurrent Computation, Proceedings, Evian, France," Lecture Notes in Computer Science No. 70, pp. 147–171, Springer-Verlag, Berlin/New York, 1979.
9. N. LYNCH AND M. FISCHER, On describing the behavior and implementation of distributed systems, *in* "Theoretical Computer Science," North-Holland, Amsterdam, 1981.
10. G. PETERSON, "The Complexity of Parallel Algorithms," Ph.D. thesis, Computer Science Department, University of Washington, 1979.
11. G. PETERSON AND M. FISCHER, Economical solutions to the critical section problem in a distributed system, *in* "Proceedings of the Ninth Annual ACM Symposium on Theory of Computing," pp. 91–97, 1977.