Dynamic Task Scheduling in Hard Real-Time Distributed Systems

Krithivasan Ramamritham and John A. Stankovic
University of Massachusetts, Amherst

This algorithm for scheduling tasks with hard real-time constraints—that is, tasks that *must* meet their deadlines—works dynamically on loosely coupled distributed systems.

Many tasks, such as those found in nuclear power and process control applications,[1] are inherently distributed and have severe real-time constraints. Because their execution deadlines must be met, these tasks are said to have *hard* real-time constraints. Scheduling to meet their deadlines remains a major challenge in distributed system design. Although recent advances in software, hardware, and communication technology should permit more flexibility in designing these systems, most current research on scheduling tasks with hard real-time constraints has been restricted to multiprocessing systems. It is, therefore, inappropriate for distributed systems. In addition, many of the proposed algorithms, which assume that all tasks and their characteristics are known in advance, are designed for static scheduling.

Our research is directed at developing task-scheduling software for loosely-coupled systems that achieves flexibility through dynamic scheduling of tasks in a distributed, adaptive manner. We have developed

(1) a locally executed guarantee algorithm for periodic and nonperiodic tasks, which determines whether or not a task can be *guaranteed* to meet its real-time requirements;

(2) a network-wide bidding algorithm suited to real-time constraints;

(3) criteria for preempting an executing task so that it still meets its deadline; and

(4) schemes for including different types of overheads, such as scheduling and communication.

The algorithm was developed with task independence assumed, and then extended to consider precedence constraints. The only resource requirement explicitly taken into account is CPU time.

Our scheme for distributed dynamic scheduling requires one scheduler per node. The schedulers interact to determine assignment for a newly arriving task. Associated with a node in a distributed system is a set, possibly null, of periodic tasks guaranteed to execute on that node. We assume that the characteristics of periodic tasks are known in advance and that such tasks must meet their deadlines. The existence of enough processing power at a node for all periodic tasks to meet their deadlines is verified at system initialization time. In addition to the periodic tasks, we allow for the arrival of nonperiodic tasks at any time and attempt to guarantee these tasks dynamically, in the presence of periodic tasks and on a network-wide basis.

Background

Most research on scheduling tasks with hard real-time constraints has been restricted to uniprocessor and multiprocessor systems. For example, Garey and Johnson[2] described an algorithm that determines if a two-

0740-7459/84/0700/0065$01.00 © 1984 IEEE

processor schedule exists where all tasks are completed in time—given a set of tasks, their deadlines, and the precedence constraints of all tasks. Liu and Layland[3] derived necessary and sufficient conditions for scheduling periodic tasks, with preemption permitted. Their results, which hold for uniprocessor systems, were subsequently extended to include arbitrary task sets[4] and precedence constraints.[5] Teixeira[6] developed a model that considers priority scheduling and that addresses external interrupts, scheduling overhead, and preemption. Johnson and Madison[7] developed a measure of free time, similar to our notion of surplus processing power, to determine whether new jobs could be permitted to execute on multiprocessor systems. These schemes are all quite inflexible and are restricted to uniprocessor and multiprocessor systems. We are attempting to develop more flexible dynamic scheduling techniques for loosely coupled networks.

Muntz and Coffman[8] developed an efficient algorithm to determine minimal-length preemptive schedules for tree-structured computations in multiprocessor systems. Similarly, Leinbaugh has developed analysis algorithms that, when given the device and resource requirements of each task and the cost of performing system functions, determine an upper bound on the response time of each task. His initial results[9] for multiprocessor systems have been extended to distributed systems.[10] Resource requirements and some operating system overheads are accounted for in these papers, but periodic tasks are not. These approaches are useful at system design time to statically determine the upper bounds on response times, but there is no attempt at guaranteeing that a new task will meet its deadline, although it may be possible to extend the approaches in this manner.

Mok and Dertouzos[11] describe multiprocessor scheduling in a hard real-time environment by a game in which tokens are moved on a coordinate system defined by laxity and computation time. They have obtained the following results, which we use in our approach.

(1) In the case of a single processor, both earliest deadline scheduling (scheduling the task with the earliest deadline) and least laxity scheduling (scheduling the task with the least difference between its deadline and computation time) are optimal.

(2) In the multiprocessor case, neither is optimal.

(3) For two or more processors, no scheduling algorithm can be optimal without a priori knowledge of task deadlines, computation times, and start times.

In a multiprocessing environment, optimal scheduling is an NP-hard problem and thus computationally intractable.[12] The problem is even harder for loosely coupled distributed systems. Clearly, a practical scheduling algorithm must be based on heuristics to reduce scheduling costs, and must be adaptive. This is the context in which we have been studying the problem of scheduling in distributed systems.

Nature of tasks

Our research involves scheduling tasks, with real-time constraints, on processors in a network. A task is characterized by its start time, computation time, deadline, and, possibly, its period. In addition, a task may have resource requirements and precedence constraints. Here, we confine our attention to considering deadlines and precedence constraints in scheduling tasks, but we are studying the extension of our algorithm to handle resource requirements.

Tasks can be periodic or nonperiodic. A nonperiodic task occurs in the system just once and at an unpredictable time. Upon arrival, it is characterized by its deadline and its computation time. Such a task can be scheduled any time after its arrival, but a periodic task, say with period P, must be executed exactly once every P time units. We do not associate any other semantics with periodic tasks (such as, P time units should elapse between two consecutive executions of a periodic task). In a real-time system, there could be a number of such periodic tasks with different periods. The initial distribution of periodic tasks is assumed known, although, as explained later, our simulation model can also be used as a tool in choosing a good assignment of periodic tasks to hosts in a network. Though we do not discuss it here, it is possible to dynamically change the set of periodic tasks that execute on a node.

From a scheduler's point of view, a periodic task represents tasks with known (future) start times and deadlines, whereas a nonperiodic task can arrive at any time, can be started anytime after its arrival, and can have arbitrary deadlines. In both cases, we assume that a task's computation time is known a priori. Although we do not assume non-periodic tasks with future start times to be part of our environment, our guarantee algorithm can be easily extended to handle them.

Structure and functioning of scheduler on a node

Each node in the distributed system has a scheduler local to that node. A set (possibly null) of guaranteed periodic tasks exists at each node. Nonperiodic tasks may arrive at any node in the network. When a new task arrives, an attempt will be made to schedule the task at that node. If this is not possible, the scheduler on the node interacts with the schedulers on other nodes, using a bidding scheme, to determine the node to which the task can be sent to be scheduled. Upon arrival at the destination node, another attempt is made to schedule the task. Eventually, the task is either guaranteed and executed, or it is not guaranteed.

Underlying our scheduling algorithm is the notion of guaranteeing a task. A task is said to be guaranteed if, under all circumstances, it will be scheduled to meet its real-time requirements. Once a task has been guaranteed, we know that it will be

completed before its deadline, but not exactly when it will be scheduled. This depends on the scheduling policy, the number of guaranteed tasks waiting to be executed, the nature of periodic tasks, new arrivals, and the amount of system overhead.

Periodic tasks are guaranteed; nonperiodic tasks, after they arrive, may or may not be guaranteed. However, once guaranteed, they will definitely meet their deadlines. The intent is to guarantee all periodic tasks and as many of the nonperiodic tasks as possible, utilizing the resources of the entire network.

Figure 1 shows how the various modules—namely, the local scheduler, the dispatcher, and the bidder, as well as the guarantee subroutine—that make up the scheduler on a node interact with each other.

Bidder and local scheduler tasks. Tasks may arrive directly at a node or as a result of a bidding process. Tasks in the former category are handled by the local scheduler, while tasks in the latter category are handled by the bidder. Arriving tasks may or may not cause preemption. The conditions for both are derived below. (Since the bidder is involved in the distributed aspect of task scheduling, it is discussed in the next section.) The local scheduler first calls the guarantee routine to guarantee the new task. If it is guaranteed, the dispatcher is invoked. Otherwise, the task is placed in the bidder's queue of tasks for which bids have to be requested from other nodes, and then the dispatcher is invoked.

Dispatcher task. The dispatcher task determines which of the guaranteed periodic and nonperiodic tasks is to be executed next. As mentioned earlier, for a uniprocessor, both the earliest deadline algorithm and the least laxity algorithms are optimal. In our scheme, guaranteed tasks are executed according to the earliest-deadline-first scheme. Use of this algorithm for scheduling tasks on a single node does not guarantee an optimal schedule on the network as a whole, a computationally infeasible problem. Our aim instead is to guarantee tasks quickly and to reduce overheads. The simulation studies are an attempt to analyze the algorithm's behavior.

The dispatcher's actions are simple: whenever a task completes, the dispatcher is invoked and selects for execution the task with the earliest deadline. To expedite this selection, the list of guaranteed tasks is ordered according to task deadline. The dispatcher's runtime is included in the computation time of each task.

For the purpose of scheduling, information on periodic tasks, such as periods and computation times, is maintained in a data structure called the periodic task table. During the operation of the system, the guarantee algorithm uses the periodic task table (PTT) in conjunction with the concept of surplus to ascertain whether nonperiodic tasks can be guaranteed. Surplus is derived from information in the system task table.

System task table. Each node maintains a system task table (STT) for all local periodic and nonperiodic tasks guaranteed at any point in time. Each entry in the STT contains an arrival time, a latest start time, a deadline, and a computation time. All but the latest start time are inputs. Tasks in the STT are ordered by their

Our research involves scheduling tasks, with real-time constraints, on processors in a network.

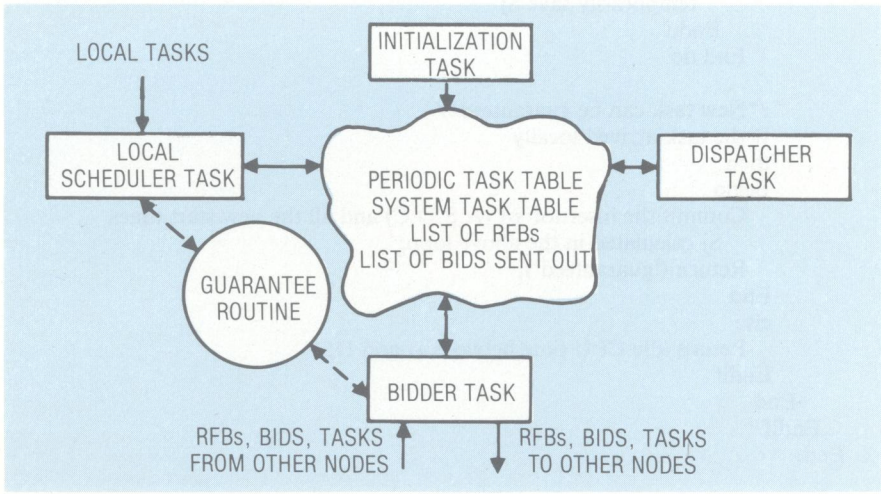


Figure 1. Structure of a scheduler.

arrival times and within each arrival time by deadlines.

To compute the latest start time of a task with deadline D, all guaranteed tasks with deadlines greater than or equal to D are ordered according to decreasing deadlines. The latest start time is determined by assuming that tasks are scheduled to execute just in time to meet their deadlines. For example, if the first task on the list has deadline D_1 and computation time C_1, it has a latest start time of $D_1 - C_1$. Suppose the second task on the list has a computation time C_2 and deadline D_2. If D_2 is greater than $D_1 - C_1$, the task has a latest start time of $D_1 - C_1 - C_2$; otherwise, $D_2 - C_2$. In this manner, latest start times are calculated for every guaranteed task.

```
Begin
  Let 'tick' be the current clock time;
  Let the new task be a 4-tuple, (A,S,D,C), where
       A = arrival time, S = latest start time,
       D = deadline, C = computation time;
  Let lastD be the latest deadline of the last window in the current STT;
  Do while (lastD < D)
    Append a copy of PTT to the end of STT;
    Add lastD to all the clock times of tasks in the appended copy of
      PTT;
    Update lastD;
  end do
  Find the current entry, i, in STT, such that
    (A < Ai) or (A = Ai and D < Di);
  Find the latest start time, S for the new task taking into account tasks from
    the current entry i to the end of the STT;
  If (S < tick) or (S < A)
    then return ('not guaranteed');
    else
    Begin
      Temporarily consider (A,S,D,C) as inserted prior to the entry i and be
      the
        new current entry;
      For each entry j from the current entry backwards to the first entry,
        Do
          Calculate new latest start time, Sj;
          If (Sj < tick) or (Sj < Aj)
          then
            return ('not guaranteed');
          else
            temporarily save Sj
          Endif
        End do

      /*New task can be guaranteed*/
      If the task arrived locally
      then
      Begin
        Commit the insertion of (A,S,D,C) and all the new start times
          Sj calculated in the above loop;
        Return ('guaranteed');
      End
      else
        Return idle CPU time between A and D;
      Endif
    End
  Endif
End;
```

Figure 2. Outline of the guarantee routine.

Surplus. Clearly, a newly arriving task can be guaranteed to execute at a node only if the surplus processing power at that node, between when the task arrives and its deadline, is greater than the computation time required for the task. We are interested in surplus with respect to the task about to be guaranteed or rejected. Surplus, then, is defined as the amount of computation time available on a node between the arrival time of the new unguaranteed task and its deadline.

While surplus is not explicitly calculated before guaranteeing a local task, surplus information is considered in computing latest start time, and the task is guarateed only if the latest start time is greater than the arrival time. However, as explained later, surplus is computed explicitly while responding to a request for bid.

Initialization task. Before the guarantee routine examines newly arriving tasks, its data structures have to be initialized. This is the function of the initialization task, which is executed just once—at system initialization time. Since a periodic task is known to represent regularly occurring tasks with future start times, at system initialization time it has to be determined if there is enough processing power to execute a given set of periodic tasks on a node. We define a window to be the least common multiple L of the periods $P_1, \ldots, P_N$ of all periodic tasks $1, \ldots, N$. The tasks have computation time $C_1, \ldots, C_N$. A necessary condition for periodic tasks in a window to be guaranteed is

$$\sum_{i=1}^{N} \{C_i * (L/P_i)\} \leq L$$

That is, the sum of the computation times of all periodic task instances that have to be executed within the window is less than or equal to the window itself. After this necessary condition is verified, the latest start time for every task instance in the window is calculated with the algorithm described previously. If all the tasks have latest start times no earlier than their arrival times, they are guaranteed. The calculated start times, computation times, and deadlines are then stored in the PTT for each task instance in the window.

Guarantee routine. The guarantee routine local to a node is invoked to determine if there is enough surplus processing power to execute a newly arriving task before its deadline. A task can be guaranteed only after ascertaining that this guarantee does not jeopardize either previously guaranteed nonperiodic tasks or periodic tasks with future start times. If a newly arriving task cannot be guaranteed locally, the task becomes a candidate for bidding. The guarantee routine (see Figure 2) uses information in the PTT and the STT to guarantee a newly arriving task. Recall that each entry in the STT contains an arrival time, a latest start time, a deadline, and a computation time. Note that the guarantee routine is coded, assuming that before it is called, the STT has been updated to reflect the current state of the system.

Time-overhead considerations. A prime consideration in allocating the scheduling activity among various tasks is the time spent on scheduling. This is important in hard real-time systems. Since the dispatcher has to be invoked each time any task, including the local scheduler and bidder tasks, completes execution and relinquishes the CPU, we require the dispatcher's execution time to be included in every task's computation time.

Newly arriving nonperiodic tasks must be examined soon after they arrive. But interrupting a running task to guarantee a newly arriving task could cause the running task to miss its deadline. To prevent this, we use the following scheme: after the dispatcher chooses the next task to run, using the SST, it checks the surplus to verify that running the bidder task or the local scheduler task, after preempting the newly dispatched task, will not cause guaranteed tasks to miss their deadlines. If this is true for the bidder (local scheduler) task, then the dispatcher sets the invoke-bidder bit (invoke-local-scheduler bit). Even if a running task is not preemptable, thereby preventing a new task from being guaranteed soon after its arrival, the newly arriving task should be examined without undue delay. To facilitate this, both the bidder and the local scheduler tasks are executed as periodic tasks. The period and computation time of these tasks is determined a priori by the nature of tasks, for instance, the estimates of their laxity and frequency of arrival, as well as by the nature of communication from other nodes in the system, for instance, the frequency of request for bids.

Time spent on scheduling is important in hard real-time systems.

The above scheme is based on the assumption that there is a communication module, executing on a processor separate from the CPU on which tasks are scheduled, that is responsible for receiving communication from local sources as well as from other nodes. Based on the type of communication, this module stores received information in the appropriate data structures so that they will be looked at when the different tasks execute. In addition, if a task arrives from another node and the invoke-bidder bit has been enabled, the currently running task is preempted and the bidder task is executed. If, instead, a task arrives locally and the invoke-local-scheduler bit is set, the currently running task is preempted and the local scheduler task is executed.

A logical extension to the above scheme, which allocates a separate processor for the communication module, is an architecture for a real-time distributed system that allocates a separate processor with limited processing capabilities for scheduling. In this scheme, scheduling overheads are incurred only by the additional processor, since the main processing unit is used only for task execution.

Distributed task scheduling

Interaction between schedulers on different nodes is based on a bidding scheme[13]: the scheduler with a task that needs to be scheduled, but which cannot be scheduled on that node itself, requests bids from nodes with surplus processing power. The bidder component of a scheduler utilizes the knowledge contained in each scheduler regarding the estimated surplus processing power in other nodes. This surplus information is derived from information passed between nodes in bids or is explicitly requested from other nodes.

Nodes making bids do not reserve resources needed to execute the task for which they are bidding. When a task arrives at a node, as the result of a bid being accepted, it is checked again to determine if it can be guaranteed. The node may be unable to guarantee the task if its surplus has changed since it sent the bid. This occurrence can be due to local task arrivals and to the arrival of tasks as a result of previous bids. One solution to this problem is for nodes to reserve resources, specifically CPU time slots, for the task for which they are bidding. We have not adopted this solution because of the poor resource utilization that it is likely to entail: a node may bid for more tasks than it will be awarded and there may be multiple bids for a task.

During the bidding process, it is necessary to consider communication delays. Communication delays depend on the pairs of processes involved, for instance, on the distance separating them and on the com-

munication from other nodes in the system to the two nodes. We plan to estimate the time in the following way: every communication will be time-stamped by the sending node. The receiving node will then be able to compute the delay due to that communication by subtracting the time-stamp from the time of receipt. Subsequent communication delays will be estimated, based on a linear relationship between message length and communication delay. By utilizing the latest known delay, this computation can adapt to changing system loads.

In our scheme, we assume that the clocks on different nodes are synchronized. The only effect of asynchrony in the clocks will be that the estimates, for example, of task arrival times and communication delays, will be inaccurate. When tasks are independent, a node can guarantee a task solely on local information, and asynchrony in the clocks will not affect such tasks. (The next section discusses our scheme for handling precedence constraints, which does require approximately synchronized clocks.)

We now describe in detail the different phases of the bidding process. The bidder first checks whether it can send the task to another node via *focused addressing*, which utilizes surplus information to reduce overheads in the bidding process. If focused addressing can determine that a particular node has a surplus significantly greater than the new task's computation time and, thus, a high probability of guaranteeing it, the new task can be sent directly to that node without bidding.

We propose to do this in the following manner: Estimate the arrival time AT of the task at the selected node. If the estimated surplus of that node, between AT and the deadline D of the task, is greater than the computation time C of the task by FP percent, the task is sent to the node. The receiving node, as stated earlier, uses the guarantee routine to check whether it can guarantee the arriving task. FP is an adaptive parameter used in focused addressing.

If there is no node with a significant amount of surplus, bidding is invoked. In this case, the main functions of the bidder are sending out a request for bids for a task that cannot be guaranteed locally, evaluating bids, and responding to the request for bids from other nodes.

Request for bids. For a task that cannot be locally guaranteed, the bidder broadcasts to all nodes a request-for-bid message containing the task's computation time C, deadline D, and size S; the time T at which the message is being sent; and a deadline for responses R. R is the time after which the requesting process will examine the bids to choose the best bidder. R should be such that after R there is sufficient time (1) for the requesting process to evaluate the bids, (2) for the task to reach the best bidder node, (3) for the best bidder to guarantee and schedule the task, and (4) once scheduled, for the task to complete computations and meet its deadline. Thus,

$$R = D - (P + E + W + C)$$

where P is the period of the task that evaluates bids and hence the maximum waiting time before bids are recognized; E is the estimate of the average time taken for the task to reach the best bidder; and W is a window representing the estimated time after arrival that the task might begin computation. Both E and W adapt to the load on the communication network as well as to the surplus of the receiving node.

If R is insufficient for the requests to reach other nodes and for the nodes to send their bids, the bidder then resorts to focused addressing. In this case, FP, the adaptive parameter used in focused addressing, is adjusted to augment the chances of finding a node with surplus. If a node with surplus still cannot be found, the task cannot be guaranteed.

For a task that cannot be locally guaranteed, the bidder broadcasts to all nodes a request-for-bid message.

A possible improvement over the above scheme for requesting bids, one in which RFBs are broadcast to all nodes, would be to send RFBs only to nodes whose estimated surplus matches the requirements of the task to be guaranteed, thus avoiding potentially unnecessary communication. This approach, however, would require time to check the node-surplus information to determine potential bidders and could prevent bidding by nodes with surplus if the available information was inaccurate.

Bidding. When bidding in response to a request for bids, the bidder first estimates that its response will reach the requestor before the response deadline. It proceeds with further actions only if the time of response plus the transit time for the response is less than the indicated deadline.

Once a node decides to respond, it first computes ART, the estimated arrival time of the task if, indeed, it is awarded the task. Computation of ART takes into account the following: (1) the fact that bids at the requesting node are evaluated after the response deadline; (2) the average delay in evaluating bids (estimated to be one half the bidding period); and (3) the estimated time for the task to arrive at the bidder's node.

Whether the bidder can execute the new task is determined by the second component of the bid, $SARTD$, which is the surplus at the bidder's node between ART and the task deadline D. The surplus information takes into account

(1) future instances of periodic tasks (to prevent jeopardizing guaranteed tasks);

(2) processing time for tasks that may arrive as a result of previous bids (to ensure that nodes requesting bids are aware of other bids by a node and

minimize the probability of a node being awarded tasks with conflicting requirements or being awarded too many tasks, creating an unstable situation); and

(3) processing time needed for nonperiodic tasks that may arrive locally in the future (to minimize the probability of a task arriving as a result of a bid not being guaranteed due to the arrival of a local task with similar real-time requirements).

While accurate information is available concerning (1), information needed for (2) and (3) is estimated, based on the past behavior of the node. In our case, we keep track of *PNB,* which is the percentage of CPU time used by nonperiodic tasks arriving as a result of bidding, and *PNL,* which is the percentage of CPU time used by nonperiodic tasks arriving locally. These percentages are maintained for the most recent *PT* time units, where *PT* is an adaptive parameter.

Let *S* be the surplus between *ART* and *D,* computed taking into account only tasks already guaranteed. Then *SARTD,* the surplus between *ART* and *D,* taking into account all three factors mentioned above, is given by

$$S-(PNL+PNB)*(D-ART).$$

The bidder responds to the RFB only if *SARTD* is greater than the task's computation time *C. ART, SARTD, PNL,* and *PNB* are sent by the bidder to the requester.

Before we describe how the requestor picks the best bidder—that is, the scheduler's bid processing aspects—let's consider some possible improvements to the bidding process.

Suppose a node receiving an RFB, utilizing its estimates of other nodes' surpluses, determines that another node has a higher probability of being awarded the task. Communication and computation costs incurred in bidding could be reduced if the node decided not to respond to the RFB. Of course, the accuracy of this decision would depend on the accuracy of the surplus information possessed by the node.

It is quite possible for a node to have so many local processing requirements that it does not want to respond to RFBs. It could intimate to all other nodes that it has no surplus until some future time. This information could be used, as indicated under requesting for bids, in choosing nodes to which RFBs should be sent. Alternatively, a node could temporarily disable the mechanism that recognizes RFBs from other nodes.

Bid processing. Bids are processed by the node that originally sent the request for bids. A bid processor queues all bids until the response deadline *R*. Once the deadline passes, the bid processor computes the task's estimated time of arrival at each bidder's node. For each bidder it estimates *SETAD,* the surplus between *ETA* and *D,* assuming the same rate of surplus indicated by the bidder.

Bids are processed by the node that originally sent the request for bids.

SETAD is computed using the following formula:

$$SETAD = SARTD*(D-ETA)/(D-ART)$$

The bid processor then chooses the one with the greatest *SETAD* as the best bidder.

To the best bidder, the task is sent. The identity of the second best bidder, if any, is also communicated to the best bidder. So that bidders will know the response to their bids and can purge unnecessary information, the bid processor intimates to all but the best bidder that their bids were not accepted. A possible alternative, which avoids the traffic resulting from the responses, is for bidders to time-out after a predetermined interval.

One final note about the information sent on bids: A node utilizes it to track surpluses in other nodes, and this surplus information is used for focused addressing and for sending RFBs to nodes with high surplus. In responding to an RFB for a task, a bidder sends the estimated arrival time and the estimated surplus between the arrival time and the task's deadline. If a node does not respond to an RFB, it is assumed that it does not have sufficient surplus. (In reality, a node may not have sent its bid because it estimated that its bid will not arrive before the deadline for responses.) Since information received via bids is bound to be fragmented, a node may explicitly request surplus information from other nodes within a specific time interval. Whichever scheme is followed for obtaining surplus information, due to the nature of system activities, it will often be outdated by the time it is used. Hence, such information should be viewed as estimates that will be updated when nodes bid.

Response to task award. Once a task is awarded to a node, the awardee node treats it as a task that has arrived locally and takes action to guarantee it. If the task cannot be guaranteed, the node can request for bids and can determine if some other node has the surplus to guarantee it. However, given that the task was sent to the best bidder and that the task's deadline will be closer than before, the chances of there being another node with sufficient surplus are small. Hence, we decided to send not only the task but also the identity of the second-best bidder to the best bidder. If the best bidder cannot guarantee the task, it sends the task to the second-best bidder, if any. Otherwise, the task is rejected.

The environment that submitted the task will be responsible for appropriate action if a task is not guaranteed. One such action could be to resubmit the task with a later deadline. If there are specifications concerning, for example, the percentage of tasks that should be guaranteed, then the system should be designed to meet them. The simulation model, described later, can be used as a tool in this regard. The

simulation can be tailored to model a given system to determine whether it can meet the specifications and, if so, how the periodic tasks should be allocated to individual nodes.

Precedence constraints

In the previous discussion of the bidding algorithm, all tasks were independent of each other, but a real system often has precedence constraints among tasks. The bidding algorithm can be extended to handle tasks with precedence contraints. The scheme is more general than the single example, presented below, indicates.

Consider that task A has been guaranteed on node 1, task B has been guaranteed on node 2, and task C arrives at node 3. Task C with deadline D_C has the following precedence requirements: A should precede C, and B should precede C. Also assume that $D_A < D_B$. The procedure followed consists of four steps.

(1) The local guarantee routine attempts to guarantee C at node 3, assuming a start time of $D_B + T$, where D_B is the latest deadline of all preceding tasks and T is the maximum time required for the outputs from task B to reach task C. If this local guarantee is successful, the procedure is finished.

(2) If task C cannot be guaranteed at its arrival site (node 3 in this example), node 3 broadcasts a request for bids with an indication that bids should be returned, not to itself, but to the node containing the preceding task with the latest deadline (node 2 in this example, because $D_B > D_A$). Also, node 3 sends the task itself to node 2. This approach provides additional parallelism and should increase the successful guarantee rate.

(3) Once the task arrives at node 2, there is an attempt to guarantee it at that node. If this is not successful, the local guarantee algorithm attempts to modify D_B to D'_B so that $D'_B < D_B$, $D_A < D'_B$, task B remains guaranteed, and task C can be guaranteed. This artificial movement of task B's deadline to an earlier time and subsequent processing is all done locally and in parallel with waiting for returning bids. Many modifications to this part of the procedure are possible. For example, if moving the deadline of task B to an earlier time fails to guarantee task C, it is sometimes possible to attempt to move tasks that task B might depend on to an earlier time in a recursive fashion (none in this example). The problem is that these subsequent tasks might be on other nodes, causing additional message traffic and complications. Therefore, we restrict the process of moving deadlines to an earlier time to the local site, but it is, of course, subject to network-wide precedence constraints.

(4) Finally, if task C is still not guaranteed, node 2 waits for returning bids and chooses the best bidder as in the normal bidding process.

The bidding algorithm can be extended to handle sets of tasks.

However, if in step 3, D_B was moved up to D'_B, task B retains this new deadline to increase the chances of guaranteeing task C at its new location.

Although this article does not consider problems that arise in determining maximum communication delay T, it is possible to determine T if, for example, the subnet uses a deterministic algorithm. In addition, the requirement of synchronization among clocks becomes crucial if precedence constraints are included. This is because, in the above example, C's guarantee is based on the fact that B will finish before its deadline—that is, before time D_B according to the clock local to B's node. We assume that clocks are synchronized, using one of the proposed algorithms (for an example, see Lamport and Melliar-Smith[14]) and take maximum clock asynchrony into account while estimating communication delays.

Evaluation of the scheduling algorithm

The task scheduling algorithm proposed in this article consists of two main algorithms, the local guarantee algorithm and the bidding algorithm. These algorithms are being evaluated in two stages of simulation. The first stage, which is complete, implemented the local guarantee algorithm in an event-driven simulation model. Stage 2, which extends this model to handle the bidding algorithm, has been implemented and debugged, but only preliminary tests have been run. The overall simulation model, the planned evaluation process, and the stage 1 simulation results are described below.

Simulation model. The program, written in GPSS and Fortran, simulates a (variable) number of nodes connected by an arbitrary topology. All model parameters can be easily changed to facilitate evaluation, and the model is extensible so that precedence constraints and additional resource requirements (other than the CPU) can be added later.

In the simulation model each node is described by its processing speed. The model runs all the scheduling algorithm's modules (local scheduler, dispatcher, and bidder), and it executes periodic as well as nonperiodic tasks. As described in our algorithm, tasks can sometimes be preempted. This facility is also included in the simulation model. Tasks are characterized by their period, if any, their computation time, and their deadline. Specific values for these parameters are generated by GPSS functions.

The communication subnet is modeled by a delay per packet per hop. The delay is described by a random variable with a distribution function chosen at runtime. Various types of information move through the subnet, each experiencing a different delay as a function of its size. The types of information include requests for bids (one packet), the bids themselves (a variable number of packets), and the tasks (a variable

number of packets). Arrival rates for the nonperiodic tasks are Poisson and occur independently at each node of the network.

The set of periodic tasks for each node must be chosen at the start of the test, but can be easily changed from run to run. Execution of the initialization task ensures that all periodic tasks can be guaranteed. If it doesn't, initialization terminates immediately with an error condition.

The statistics accumulated include node utilization, task response time, node queue length, percentage of nonperiodic tasks that meet or do not meet their deadline, percentage of nonperiodic tasks that move to another site and then meet or do not meet their deadline, and the number of tasks moved via focused addressing and the percentage of these that do or do not meet their deadline. Using a log file, it is also possible to determine which tasks were or were not guaranteed.

Evaluation process. Our testing determines how these statistics change as we vary the characteristics of the periodic and nonperiodic tasks (computation time and deadline), their periods and arrival rates, respectively, the delays in the subnet, and the size of the network. Since much of the scheduler runs as a periodic task, we can easily determine the effect, say, of running the bidder at various periods.

In addition, we would like to determine which parameters of our algorithm have the greatest effect on performance. For example, determining the usefulness of focused addressing, or determining the difference between an adaptive or nonadaptive policy in setting a wait time before processing arriving bids, or the effectiveness of various types of information in the bids themselves. The simulation will also be used to study the performance of the various possibilities for improvement discussed earlier.

The various simulation runs will be compared to each other as well as to a (lower-bound) baseline model where 100 percent accurate data about other nodes' schedules are assumed to be known.

Stage 1 results. The simulation of the guarantee algorithm on a single node provides some insight into the percentage of tasks guaranteed under various conditions (set of base periodic tasks, arrival rates, task characteristics, etc.). For example, Table 1 shows the simulation results where computation time (CPU time) and deadline characteristics are varied for a specific set of two periodic tasks and a given arrival

Table 1.
Simulation results—guarantee algorithm.

	CPU Time	$\hat{D}$ Deadline	Tasks Guaranteed	Tasks Not Guaranteed	Average CPU Utilization
Case 1	Uniform distribution (1-10)	Exponential distribution u = 4	65.9%	34.1%	70.1%
Case 2	Normal distribution $N(4,1)$	Same as case 1	81.6%	18.4%	60.9%
Case 3	Normal distribution $N(6,1)$	Same as case 1	64.8%	35.2%	70.2%
Case 4	Normal distribution $N(8,1)$	Same as case 1	51.7%	48.3%	73.7%
Case 5	Normal distribution $N(9,1)$	Same as case 1	45.5%	54.5%	73.9%
Case 6	Normal distribution $N(10,1)$	Same as case 1	39.4%	60.6%	72.9%
Case 7	Normal distribution $N(12,1)$	Same as case 1	29.9%	70.1%	69.5%
Case 8	Normal distribution $N(14,1)$	Same as case 1	22.6%	77.4%	65.5%
Case 9	Same as case 3	Uniform distribution (1-25)	87.3%	12.7%	85.0%
Case 10	Same as case 3	Normal distribution $N(15,4)$	87.9%	12.1%	85.2%

rate. The characteristics of the periodic tasks are

Period	CPU Time
7	1
19	2

Window size = 133

The two periodic tasks take 24.8 percent of the total CPU time.

A single arrival rate for nonperiodic tasks is utilized and assumed to be Poisson distributed. The mean of the interarrival time is nine units of time. The simulation runs for 4000 units of time. During this period, 783 instances of periodic tasks are executed and 495 nonperiodic tasks are generated. Six different combinations of computation time and deadline distributions are shown in Table 1. Note that the deadline is assigned to tasks according to the formula:

$$D = \text{current clock} + \text{computation time} + \hat{D}$$

where $\hat{D}$ is taken from a distribution function.

Based on the results of case 1 to case 8, we conclude for this set of data that (1) given the same $\hat{D}$ distribution, if most of the tasks have lower computation time requirements, the percentage of guaranteed tasks is higher, and (2) the CPU utilization is higher when most tasks have higher computation time requirements, but reaches a maximum of 73.9 percent when the computation time distribution is $N(9, 1)$, then goes down as the mean of the distribution is even higher. The percentage of guaranteed tasks is higher for lower computation time requirements because they can more easily be guaranteed.

Lower CPU utilization for tasks of lower computation time requirements is due to two factors: (1) even though more tasks are guaranteed, total computation time is still less, and (2) CPU time is fragmented by a larger number of small pieces of tasks, which causes more idle CPU time slots. When the computation time distributions go beyond $N(9, 1)$, the CPU utilization drops because too few tasks are guaranteed. Note that, in case 8, the CPU utilization is still more than that in case 2, even though the percentage of tasks guaranteed is much less. In cases 3 and 9 we see that, by changing the $\hat{D}$ distribution, the amount of the guaranteed tasks and the CPU utilization are improved by 35 percent and 21 percent, respectively. From this comparison, we conclude that, with the same computation time distribution, more tasks are guaranteed if tasks have more laxity.

In case 9 versus case 10, improvement in the percentage of guaranteed tasks is negligible. This is because the percentage of tasks with urgent deadlines, which are not easy to guarantee, are about the same in both cases. Other similar runs with different sets of periodic tasks have been made. As expected, as a higher percentage of time is allocated to periodic tasks, fewer and fewer nonperiodic tasks are guaranteed and executed. While stage 1 of the simulation reveals little about the overall approach, it does show that our simulation program can be used to help identify a "good" set of periodic tasks for a node, given a known nonperiodic arrival rate.

Stage 2 plans. Stage 2 has been implemented, and simulations are just beginnning. Due to the large number of parameters involved, extensive tuning is necessary. We have observed, for example, that if all nodes have the same set of periodic tasks and the same characteristics for nonperiodic tasks, bidding is not effective. This is because a task that cannot be locally guaranteed will, on the average, find the same conditions in other nodes, and significant time will have already passed. Thus, the task will not be guaranteed. To test our heuristics, the parameters must be chosen to produce unbalanced loads in the system.

To summarize, our algorithm for scheduling tasks with hard real-time constraints in a loosely-coupled distributed system has the following main features.

(1) It is a distributed scheduling technique; no node has greater importance, as far as scheduling is concerned, than any other.

(2) The algorithm is designed to schedule both periodic tasks as well as nonperiodic tasks. The latter may arrive at any time.

(3) A bidding approach is used for the selection of nodes on which tasks execute. We have worked out the details of the various phases involved in the bidding process.

(4) Overheads involved in scheduling are taken into consideration. This includes the time spent on executing scheduling tasks at a node and the communication delays between nodes.

(5) Heuristics are built into the algorithm. This is necessary given the computationally hard nature of the scheduling problem.

(6) The algorithm can be extended to take into account other types of constraints, in particular, precedence and resource constraints.

(7) Simulation studies are being performed in order to evaluate many of the alternatives available. ■

Acknowledgment

This work was supported in part by the US Army CECOM, CENCOM under grant DAAB07-82-K-J015 and by the NSF under grant MCS 82-02586.

References

1. J. D. Schoeffler, "Distributed Computer Systems for Industrial Process Control," *Computer*, Vol. 17, No. 2, Feb. 1984, pp. 11-19.
2. M. R. Garey and D. S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM J. of Computing*, 4, 1975.

3. C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM,* Vol. 20, No. 1, Jan. 1973.

4. M. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Proc. IFIP Congress,* 1974.

5. R. Henn, "Antwortzeitgesteuerte Prozesourzuteilung unter Strengen Zeitbedingungen," *Computing,* Vol. 19, 1978.

6. T. Teixeira, "Static Priority Interrupt Scheduling," *Proc. Seventh Texas Conf. Computing Systems* Nov. 1978.

7. H. H. Johnson and M. S. Madison, "Deadline Scheduling for a Real-Time Multiprocessor," NTIS (N76-15843), Springfield, Va., May 1974.

8. R. R. Muntz and E. G. Coffman, "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *JACM,* Vol. 17, No. 2, Apr. 1970.

9. D. W. Leinbaugh, "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Trans. Software Engineering,* Vol. SE-6, Jan. 1980.

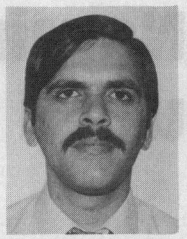
10. D. W. Leinbaugh and M. Yamini, "Guaranteed Response Times in a Distributed Hard-Real-Time Environment," *Proc. Real-Time Systems Symposium,* Dec. 1982.

11. A. K. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Seventh Texas Conf. Computing Systems,* Nov. 1978.

12. R. L. Graham et al., "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics,* 5, 1979, pp. 287-326.

13. R. G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *IEEE Trans. Computers,* Vol. C-29, No. 12, Dec. 1980.

14. L. Lamport and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," SRI International, Mar. 1982.

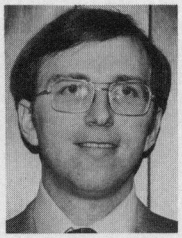


Krithivasan Ramamritham is an assistant professor in the Department of Computer and Information Science at the University of Massachusetts, Amherst. His research interests include software engineering, operating systems, and distributed computing.

He received the B.Tech degree in electrical engineering and the M.Tech degree in computer science from The Indian Institute of Technology, Madras, India, in 1976 and 1978, respectively, and the PhD in computer science from the University of Utah in 1981.

Ramamritham is a member of ACM and the IEEE Computer Society.

The authors may be reached at their respective departments at the University of Massachusetts, Amherst, MA 01003.



John A. Stankovic is an assistant professor in the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. Active in distributed systems research since 1976, Stankovic now serves as the vice-chairman of the IEEE Computer Society's Technical Committee on Distributed Operating Systems. His current research includes various approaches to process scheduling on loosely-coupled networks and recovery protocols for distributed databases.

Stankovic received the 1983 Outstanding Junior Faculty Award for the School of Engineering at the University of Massachusetts. He received his ScB in electrical engineering in 1970, and his ScM and PhD in computer science in 1976 and 1979, respectively, all from Brown University. He is a member of ACM, IEEE, and Sigma Xi.