Decentralized Decision Making For Task Reallocation In A Hard Real Time System

September 1, 1987

COINS Technical Report 87-91

John A. Stankovic[1]
Dept. of Computer and Information Science
University of Massachusetts

Decentralized Decision Making For Task Reallocation In A Hard Real Time System

John A. Stankovic[1]
Dept. of Computer and Information Science
University of Massachusetts

September 1, 1987

[1]This work began while the author was on sabbatical at CMU and was working on the Archons project. It was supported by ONR under contracts 048-716/3-22-85 and NOOO14-84-K0734, RADC Contracts 93921-39 and RI-44896X, and by NSF under DCR-8500332.

Abstract

A decentralized task reallocation algorithm for hard real-time systems is developed and analyzed. The main properties of the algorithm are that it is fast, decentralized, reliable, specifically considers deadlines of tasks, attempts to utilize all the nodes of a distributed system to achieve its objective, handles tasks in priority order, and separates policy and mechanism. An extensive performance analysis of the algorithm via simulation shows that it is quite effective in performing reallocations and that it is significantly better than a centralized approach.

Keywords: Centralized Algorithm, Cooperation, Decentralized Algorithm, Decision Making, Decentralized Decision Making, Hard Real-Time, Reallocation, Real-Time, Reconfiguration, Scheduling, Simulation.

1 Introduction

Distributed computer systems potentially provide significant advantages including good performance, high reliability, significant resource sharing, and extensibility. One particular class of distributed computer systems contains hosts that are highly integrated and being used for a *single* application such as process control, control of a nuclear power plant, or control of a submarine, aircraft carrier or space shuttle [1][13]. These dedicated distributed systems are often associated with real-time environments in which there is an especially strong requirement for meeting deadlines, for reliability, and for continued operation in the presence of failures. In this paper we are concerned with one aspect of reliability, that is, reallocating real-time tasks throughout a distributed system after a single host failure. In particular, we are interested in maximizing the success ratio of reallocated tasks. The *success ratio* is defined as the percentage of tasks that were on the failed host and which were successfully moved to another site. A *successful move* means that the task completed by its original deadline. The main contributions of this paper are the development of an efficient decentralized task reallocation algorithm tailored to real-time constraints and its analysis. We show that distributed decision making by a system-wide reallocation algorithm is a better approach than a centralized approach with backup. We also discuss the tradeoff between execution time speed of the algorithm and cooperation. By *cooperation* is meant the type and amount of information exchange between decentralized entities of the reallocation algorithm.

There has been a tremendous amount of work in developing reliable systems, and task reallocation has often been discussed. See the following texts for background information [10][14]. However, to our knowledge, the current literature has not reported on an evaluated decentralized reallocation algorithm in a hard real-time environment. A centralized algorithm was reported in [12], but it was preliminary, not evaluated by simulation, and was aimed at a considering specific resource requirements such as ports and memory constraints. Since reallocation is so similar to scheduling, some of the hard real-time scheduling literature may be considered applicable. See [3][4][5][7] and [9]. However, none of this scheduling work addresses the specific issues related to task reallocation, as is done in this paper.

The remainder of the paper is organized as follows. Section 2 presents our system model of hard real time systems, and lists our assumptions. Section 3 presents the decentralized reallocation algorithm itself. Section 4 describes the simulation program, the centralized algorithm, and contains the simulation results on the performance analysis of both the centralized and decentralized algorithms. Section 5 revisits the major steps of the algorithm to emphsize the performance versus cooperation trade-

offs made in the reallocation algorithm. Section 6 summarizes the conclusions of the paper.

2 Hard Real-Time Systems

A *Hard Real-time* system [2] is defined as a system in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Further, these real-time constraints must be met, else there are potentially catastrophic consequences. In many real-time systems one typically finds multiple levels of timing constraints. For example, processing data from a sensor might need to occur in the microsecond or millisecond range, while consistent updates of replicated files might need to occur in the seconds range, and the movement of material in an automated factory might have deadlines in the range of minutes or hours. Our overall approach assumes that tasks with tight time constraints are dealt with by using preallocated devices, processors, or time slots. The tasks with intermediate or long range timing constraints are dealt with in a dynamic manner, and it is these tasks that are the subject of possible reallocation.

2.1 Architecture of the System

We assume that the system is composed of a physically distributed collection of nodes interconnected by a subnet. Each node is a multiprocessor where one or more processors are dedicated system processors, and one or more are application processors. The system processors run the operating system which includes programs such as the distributed scheduling and distributed reallocation algorithms. The scheduling algorithm separates policy from mechanism and is composed of 4 modules. At the lowest level there exists a dispatcher. The dispatcher is the mechanism of the scheduler and it simply removes the next task from a system task table (STT) that contains all guaranteed tasks already arranged in the proper order. The second module is a local scheduler. The local scheduler is responsible for locally *guaranteeing* that a new task can make its deadline, and for ordering the tasks properly in the STT. The third module is the global scheduler which attempts to find a site for execution for any task that cannot be locally guaranteed. The final module is a meta level controller which has the responsibility of adapting various parameters of the scheduling algorithm by noticing significant changes in the environment, and also by serving as the user interface to the policies of the scheduling algorithm. The meta level controller may not always exist and is not discussed in this paper. The decentralized reallocation algorithm is a collection of modules, one on each host. These modules are separate

from the scheduling modules, but they invoke both the local scheduler and global scheduler modules.

A key ingredient to handling the real–time characteristics of the tasks is the local scheduler module which runs the guarantee algorithm. The basic notion and properties of the guarantee have been developed and analyzed elsewhere [7][8][11][15][16]. To give the reader a better understanding of this underlying mechanism which, in this paper, we assume exists, we list its major characteristics:

- conflicts over resources are *avoided* thereby eliminating the random nature of waits for resources found in timesharing systems (this same feature also minimizes context switches since tasks are not being context switched to wait for a resource),

- there is a separation of dispatching and guarantee allowing these system functions to run in parallel; the dispatcher is always working with a set of tasks which have been validated to make their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks,

- by performing the guarantee calculation when a task is invoked there may be time to reallocate the task on another host of the system via the global scheduling module or perform error handling,

- the guarantee can employ different strategies as a function of the deadline of the incoming task,

- within this approach there is notion of still possibly making the deadline even if the task is not guaranteed, that is, if a task is not guaranteed it receives any idle cycles and, in parallel, there is an attempt to get the task guaranteed on another host of the system subject to location dependent constraints; an alternative is to run a substitute task and/or error handler early, rather than only after a deadline is missed,

- some real-time systems assign fixed size slots to tasks based on their worst case execution times, we guarantee based on worst case times but any unused cpu cycles are automatically reclaimed, and

- in general, the guarantee is subject to computation time requirements, deadline or periodic time constraints, resource requirements, priority, precedence constraints, and I/O requirements.

2.2 Characteristics of Tasks

We define a *task* as the dispatchable unit of computation in the system and it is non-preemptable. We assume that tasks are independent. When a task is invoked it specifies:

- its deadline, D,
- its worst case computation time, C,
- its priority (identifies the criticality of a task; this value is independent of deadline),
- a reallocation factor, n,
- a replication factor for active copies,
- a replication factor for passive copies, and
- its type
  - single task,
  - voting task, or
  - decentralized task.

A single task is one that has one active copy and executes in one location only, thereby using a minimum of resources. However, if the host upon which this task is executing fails, the system attempts to reallocate it, subject to the task and system characteristics at the time of the failure. The replication factor for a single task is one. The reallocation factor is ≥ 1 and it represents the number of other hosts which are responsible for reallocating this task. In addition to having ≥ 1 host responsible for reallocating the task, there must exist ≥ 1 passive copies of the task; this is the replication factor for passive copies. The copies of the task itself may or may not be at the hosts responsible for deciding where to reallocate the task, and this must be taken into account when making a reallocation decision.

A task may request that it be instantiated n times (n active copies) and run in parallel, typically for voting on the outputs. Such a task, called a voting task, is inherently more reliable, but consumes more system resources. In addition, if a host upon which one of the replicated copies is executing fails, then the system attempts to reallocate that failed instance of the task unless there are still enough copies for adequate voting. If there are enough copies, then (for efficiency) the task instantiation on the failed host need not be reallocated. The number of sites

responsible for reallocation is the reallocation factor. Passive copies of this task may also exist, but typically the number is less than for a single task because there are other active copies available. For efficiency considerations our algorithm makes use of the fact that there are multiple active copies.

A decentralized task is a collection of rtasks (replicated tasks) which cooperate in a peer relationship to achieve some system-wide goal. There is no voting on the results of each computation. Many of the system server functions such as the naming server, the scheduling algorithm, the reallocation algorithm, the directory server, the resource manager, etc. are decentralized tasks. Such tasks have a minimum replication factor specified by the replication factor for active copies. Depending on the function themselves, it is possible for new active copies of a decentralized task to be instantiated to offload work, e.g., as was done in the Medusa utilities [6]. If failures result in a particular decentralized task dropping below the minimum replication factor, then reallocation is performed, else this instantiation of the rtask is not reallocated (again improving the performance of the reallocation algorithm itself because it might have less work to perform). Work that this rtask was performing must be reassigned to some other rtask of this function.

2.3 The Reallocation Problem

The goal of this work is to develop a good task reallocation algorithm for hard real-time systems where reliability and continued performance in the presence of failures is of utmost importance. Simply letting the tasks on the failed processor be lost it not acceptable. We are interested in determining the conditions under which a decentralized algorithm or a centralized algorithm is better. In any case, the algorithm must execute fast because it is dealing with hard deadlines, must be reliable, and must perform good reallocations, i.e., as many of the tasks on the failed processor as possible must be reallocated and make their deadlines. When the reallocation algorithm decides on a new location for a task, that task must then be guaranteed at that site. If it is guaranteed, then it will make its deadline. If not, for the purposes of this paper, we consider that it fails to meet its deadline. However, in practice, our scheduling algorithm which is based on the guarantee routine would let this task use idle cycles and so it may still may make its deadline - it just won't be guaranteed to do so. Such idle cycles are attained when a guaranteed task executes for a time less than its worst case time (very common), since it was guaranteed with respect to the worst case time.

The reallocation problem is actually composed of multiple subproblems:

- detect the failure,

- notify the reallocation managers (algorithm) at each site,
- stop certain tasks and task interactions,
- decide on the new allocation of tasks,
- perform actual movement of tasks, if necessary,
- recover, and
- restart tasks

In this paper we investigate algorithms that decide on the new allocation for tasks. We assume that the other parts of the problem are handled in some effective manner.

3 The Decentralized Reallocation Algorithm

Consider that at some time T, a real-time distributed system is composed of M tasks assigned across N hosts, typically $M >> N$. Also consider that a single host fails. At this time the decentralized task reallocation algorithm is invoked to reassign tasks of the failed host. In this section we begin by describing a prerequisite condition for our algorithm, then describe the actions during normal operation, followed by a description of the algorithm during failure. We discuss various options and provide motivation for our decisions on these options. Finally, we then present the pseudo code for the actual algorithm.

3.1 Prerequisite Condition

In the most general case, when a task is assigned to a site, it is done with respect to its own requirements and the host characteristics. Its own requirements might include its deadline, requirements for communication ports, memory, and bus bandwidth, its precedence requirements, and its need for data and I/O taking into account its possible contention over those resources. Host characteristics include its current state (load on resources) and its speed, amount of memory, I/O capability, and any special devices capability. The local guarantee routine is responsible for determining whether a task can be assigned to this particular site by determining whether the task can obtain the necessary resources at this host at the correct time in order to meet its deadline. In this paper we assume that a local scheduler such as those reported in [7][11][15][16] is available.

3.2 Normal Operation

When a task is assigned to a site, n buddy sites are also chosen. The assignment and buddy information needs to be accomplished as an atomic action. A fundamental point is that some site A (or for higher reliability, n sites) must know that a given task has been assigned to execute at a given site B. The buddy sites are responsible for reallocating the original task if the processor on which the original task was assigned fails. The buddy sites can be chosen in any number of ways. Guidelines for choosing the buddy sites and the subsequent decision making strategy are as follows:

- Random: For the given task under consideration, determine the set of feasible sites to which it might be reallocated. This is done based on unique resource requirements. Then choose n unique sites at random (or less than n if the number of feasible sites is less than n). As each site is chosen create a virtual chain (order) which acts as the mechanism for the decision making strategy. That is, on failure, the first buddy site in the virtual chain makes the decision without negotiating, consensus or cooperation of any kind with the other sites in the virtual chain. The other sites in the virtual chain are only activated one at a time, if there are subsequent failures.

  This strategy is meant to make decisions quickly and efficiently and to partition the workload when a failure occurs. This scheme is very quick at determining the buddy sites (minimizing overhead during normal operation) and if we have a broadcast subnet it will only cost 1 message and one commit operation.

- Non–random: One can attempt to be much smarter about assigning buddies based on (1) resources available at a site, or (2) on trying to certify that no one site will be responsible for too many tasks from the same site. However, due to the dynamic nature of our hypothesized system, there are no assurances that the initial assignment of a buddy will be appropriate at the time of a failure. Therefore, it seems that it is not appropriate to try to make assignments based on resource availability. It is reasonable to attempt to insure that a given site is not responsible for too many tasks from the same site, B. If it were, then it would have too much work to perform if a failure of site B occurred. The random scheme outlined above should distribute work evenly most of the time.

Note that because of the random assignments of buddies, when a site fails the primary buddy sites will be able to proceed in parallel, each making reallocation decisions for some subset (possibly null) of the tasks that were active at the failed site. This is decentralized but without direct cooperation. However, because the

decentralized reallocation interacts with a decentralized scheduling algorithm, there is cooperation being achieved *implicitly* via the scheduling algorithm.

When a non-periodic task completes, then we must deactivate the buddies. This cost is one reliable broadcast message. To prevent race conditions, the deactivation of a task and its buddies must be performed as an atomic action.

3.3 On Failure of a Node

When a site failure is detected, a broadcast message is sent. This broadcast must be reliable. A failure of a host results in all other hosts attempting to reallocate all of the active tasks which existed at the failed site, except the rtasks of decentralized tasks which still have the number of instances $\geq$ the minimum replication factor, and except the replicated voting tasks which have enough members remaining to still provide a majority vote. In addition, it is necessary to update the virtual chain of buddies. Two philosophies are possible here. One can consider the original reallocation factor static. That is, if $n = 5$, then after a failure, it is reasonable to now be at level $n = 4$. The other philosophy is to dynamically attempt to retain the $n = 5$ level. We decided on the first strategy. (Actually, the same is true for the replication factor and we use the same philosophy there too.) In the static strategy that we chose, the original virtual chain may have to be modified on a failure. For example, if the site chosen to execute the task is the primary buddy site, then it can no longer remain the primary buddy for subsequent failures. The second buddy now becomes the primary via a broadcast message. Further, if one of the sites in the virtual chain fails, then it is clear that this site is no longer in the virtual chain. This condition is easily handled by having each site keep track of all failed sites.

Each site, upon receiving information about a processor failure, attempts to locally guarantee tasks that were executing on the failed processor and that it has primary responsibility for. It does this in priority order. If all tasks that it is responsible for can run locally, then the algorithm invocation at this site is finished. This strategy is an attempt to make *good enough* decisions quickly.

If one or more buddy tasks cannot be handled locally, then the site finds a new site to execute that task and it remains the primary buddy. Finding a new active site can be accomplished in many ways. At this point of the algorithm, it does make sense to attempt to find a good location for the task based on resource availability so that the task being reallocated has a good chance of making its deadline. A general point to make is that we also need to eliminate from further consideration, those tasks which are too close to their deadlines. Trying to process these tasks will only increase the algorithm costs but not benefits. We use the following strategies in combination, choosing the order that we employ these strategies based on policy and

current system state:

- Focussed Addressing (FA): If a site that is attempting to reallocate a task has recent information that another site is very likely to be able to execute this task, then it directly transmits the task to this site without incurring any further overhead or delays. State information needed by FA is piggybacked on other messages between communicating sites. FA has been shown to work well under these circumstances [8][11]. Overreaction of the more heavily loaded sites of the network to a lightly loaded site must be prevented. This can be partly accomplised by modifying the local state information about the site chosen as a FA site. Doing this makes it less likely that this host will send more than one task to the FA host. Various techniques to prevent different hosts from all transmitting their work to the same FA host can be implemented. Under the conditions tested in this paper, these additional techniques were not necessary.

- Bidding: We assume that the reader is familiar with the concept of bidding. Bidding is performed when a site does not have good information (e.g., too old) or all the information it does have, does not identify a good destination. In these cases, we use bidding tuned to real-time constraints [11]. Even after bidding, a requesting site may not receive any good bids. This could then require preemption. Preemption in this context means deleting a task from the STT table (essentially unguaranteeing it) in order that a higher priority task from the failed site can be guaranteed. The amount of bidding can be reduced by requesting bids on the collection of tasks still not guaranteed when bidding is entered. Various heuristics must be developed to handle multiple simultaneous bid requests from one site. As an example, suppose site 3 has 4 tasks still unassigned when the bidding is entered as part of its portion of the reallocation algorithm. Site 3 then issues a request for bids (RFB) for all 4 tasks ordering them by priority. The RFB is then a variable length message, and, of course, contains the D, C, resource requirements, etc. of each task. In the worst case, a responding site would have to make bids on every combination of the 4 tasks, i.e., bid on task 1 alone, on task 2 alone, ..., on task 1 and 2, on task 1 and 3, ..., etc. This approach is too costly. A simpler heuristic is to have the responding site bid on each alone (i.e., as if the responding site were to bid on that task by itself and none of the others, and on an accumulation of tasks from the top down (i.e., from high to low priority). For example, in the above example with 4 tasks (ordered by priority), the responding site would bid on task 1,2,3,4 each alone, as well as on tasks 1 and 2, 1, 2 and 3, and 1,2,3,4 all together. All this information could be returned in one bid message. This

heuristic reduces the combinations of tasks to be bid on and accomplishes it using priority as an important measure. The site receiving bids of this nature would be responsible for making some ultimate choice based on all received bids. The simulation implements the standard bidding technique of addressing one task at a time, and the results show that the additional complexity is not warranted.

- Preempt Locally: When it is determined that a primary buddy task, suddenly activated, cannot be handled locally, this calculation is done without considering preemption. This same task might be schedulable locally if we preempt lower priority tasks at this site. Recall, preemption refers to cancelling lower priority ready tasks, and not to interrupting tasks in execution. It is a policy decision as to what order FA, bidding, and local preemption should be performed. It is possible to make decisons on the order of execution dynamically based on the system state and on policy. For example, host A has a task B to reallocate and its view of the system is that all hosts are very busy. In this case the algorithm tries to locally preempt before FA or bidding.

- Preempt Globally: Given that a task cannot execute locally, then the criterion might be to attempt to schedule it by preempting the lower priority tasks system-wide. This scheme requires significant overhead. Since our simulations indicate that there is little need for global preemption, we do not discuss this any further.

If the primary buddy site cannot assign a task locally, we choose a preferred order for these three strategies, e.g., FA, Bidding, and Local Preemption. However, it may be a good idea to dynamically alter this order for any subsequent reallocation attempts, e.g., begin with FA but choose between the Bidding and Local Preemption based on this site's view of the network. If the network looks busy then perform Local Preemption before Bidding. If the network is not too busy then perform Bidding before any Preemption. The subsequent site strategy should not necessarily be the same as the original strategy because of the additional processing delay experienced by the task.

When a task arrives at a new site, the local scheduler must re-determine if it can execute by its deadline. If the task can be guaranteed, then the algorithm is successful. If not, the reallocation process either continues as described above, or the policy may be to run an error recovery routine for this task. Running an error handler also occurs if the task being reallocated will not make its deadline, e.g., if deadline minus computation time is greater than the current time.

An *important* issue is the effect of the reallocation algorithm on the deadlines of other tasks at this site. Since the reallocation task runs in the system processor, then application tasks already guaranteed are not affected. However, new arrivals are not being processed immediately because the system processor is now running the reallocation algorithm. The simulation results directly address this issue.

3.4 Pseudo Code

In this section we summarize the actual decentralized reallocation algorithm. First, we describe how the algorithm operates during normal system operation, and then write the pseudo code for the part of the algorithm that executes during failure mode.

During normal system operation, as each task is guaranteed, the reallocation algorithm chooses one or more buddy sites using the random scheme. When a task completes execution, the reallocation algorithm deactivates all its buddies.

The pseudo code for the decentralized reallocation algorithm during failures is as follows.

```
PROCEDURE DECENTRALIZED_REALLOCATION;

(*  ON HOST FAILURE EACH REMAINING HOST EXECUTES AS FOLLOWS *)

  var  X:string;

  find the set of tasks {R}

(* set {R} contains those tasks from the failed host for which
   this host is the buddy minus those tasks that don't need
   reallocation because they are voting tasks or rtasks and
   meet minimum replication requirements even without
   reallocation  *)

  order tasks in {R} by highest priority

  call the local scheduling module for each task in {R}

  let tasks not guaranteed locally equal set S
```

```
FOR EACH TASK IN {S}
if laxity is small (* laxity is defined as deadline minus
                      remaining computation time*)
then
   preempt locally
   if task still cannot be guaranteed locally and
      Enough_Time
   then
      perform FA        (* the site with the highest surplus
                           is chosen as the destination
                           regardless of how busy
                           that site is; decision made
                           quickly due to tight laxity *)


else (* laxity is not small *)
 BEGIN
   if this site has the view that the network is very busy
   then
     preempt locally
     if task still cannot be guaranteed
     then
        call bidding
        if no adequate bids
        then
           call error handler for this task

   else
    FOR i = 1 to 3 DO

    set X based on policy, state information, and
        previous attempts

    (* X = 'FA' or 'Bid' or 'LPE' *)

    CASE X of
      'FA':    find focussed address host
               if successful and Enough_Time
```

```
                then
                   send task to FA site
                else
                   exit case statement
      'Bid':    send out RFB
                if successful and Enough_Time
                then
                   send task to best bid site
                else
                   exit case statement
      'LPE':    perform local preemptions
                if successful
                then
                   assign this task to this site and
                      add preempted tasks to set {R}
                else
                   exit case statement
    END_CASE

    END_FOR (* try as many as three alternatives *)
   END_BEGIN

  (* if we fail on all alternatives *)

  if all alternatives are exhausted
  then
     consider task not reallocatable
            and schedule error routine;

  END_FOR; (* try next task *)
  END_PROCEDURE;


function Enough_Time (T:task):boolean;

(* called before a task is actually transmitted *)
```

```
if (D - C - Transmit_Time - Processing_Time > Current_Time)
   then
      return TRUE
   else
      return FALSE
END Enough_Time;
```

In this reallocation algorithm we have considered the tradeoffs between making a decision quickly using the current information a site has about the state of the network versus accepting additional delay and cost of obtaining more up-to-date information. This decision was made with respect to the facts that (a) tasks being reallocated have deadlines, and (b) the reallocation routine itself must be fast because its execution time will affect the ability for tasks being reallocated to make their deadline. Our strategy was to make the decisions quickly for tasks with close deadlines, and *possibly* trade off some delay for improved information when a task has a longer laxity.

Alternative algorithms could attempt more exchange of information. This ranges from each site informing each other site about its state and its set of tasks to be reallocated, and then a single final decision is made for all the tasks. Another algorithm might use multiple rounds of messages. The type, amount, and frequency of data exchanged all could be a function of the load and/or deadlines of the tasks under consideration for reallocation. This would get quite complicated. Our simulations indicate that this additional complexity is not necessary.

4 Evaluation

4.1 Brief Description of the Simulation Programs

The main idea behind the simulation programs is to set up the state of the network just prior to the failure (called configuring the network), cause the failure, and then run the decentralized (or centralized) reallocation algorithm on each non-failed host from the point of the failure onwards to some future time where the statistics stabilize. In the simulation, we assume that each host has one application processor, one smaller system processor, and that there is only one host failure. The system processor runs the reallocation modules for both the centralized and decentralized algorithms. The size of the network is an input parameter, and the results presented here are for sizes 4 and 8. The subnet is modeled as a bus with potentailly different delays for RFB's, BIDS, and task movement.

Configuring the network means that we a) generate a load for each host including the host that will fail, b) assign buddies for each generated task in the system, and c) for each host, generate a view of the loads at the other hosts of the network (to be used in Focussed Addressing). Consider each of these aspects of configuring the network in a little more depth.

a) Generate a load: There is no good single measure for non-periodic load in a real-time system because each task has a discrete and different deadline. We deal with non-periodic load in the following way. First, we generate a load. The computation time of a task is drawn from a uniform distribution between 15-50 units of time. The deadline of a task is computed by drawning from another uniform distribution set between 5-200 units of time and adding this value to the computation time. A collection of tasks is assigned to a host so that all of them make their deadlines and the SUM(C/200) = %LOAD, where %LOAD is a parameter of the test and 200 is an arbitrary time interval long enough to have 4–10 guaranteed tasks. The problem is that %LOAD by itself is not a good indicator of load because it does not indicate the number of tasks needed to create a given load, nor the effect of deadlines. Consequently, we use two other metrics: number of tasks and %LOAD(D). %LOAD(D) is a load factor that accounts for deadlines. It is given by

```
%LOAD(D) = 100% * [[C(1)/D(1) + (C(1)+C(2)/D(2)) + ...
                                    (C(1)+C(2)+...C(N)/D(N))]/N]
```

%LOAD(D) measures the laxity of tasks at a host.

It is our belief that no single metric, to date, gives a good understanding of the true load. Rather than arbitrarily choosing one metric, we simply present all three metrics side–by–side.

An example will better clarify the meaning of the 3 load indicators. Let there be two hosts, one with 5 tasks and another with 6 tasks, all of which are guaranteed. The table below lists task numbers, worst case computation times, and deadlines of tasks, and the three load indicators. This example is taken from an actual simulation run:

TABLE 1: Example of Load Indicators

HOST A			HOST B		

TASK	C	D
1	23	67
2	49	85
3	34	112
4	34	174
5	17	181

	157	

%LOAD = 157/200 = 76%
No. of Tasks = 5
%LOAD(D) = 76%

TASK	C	D
1	28	70
2	28	72
3	24	90
4	16	195
5	32	198
6	50	219

	178	

%LOAD = 178/200 = 89%
Number of Tasks = 6
%LOAD(D) = 66%

All tasks make their deadlines.

On host A the %LOAD and %LOAD(D) metrics turned out to be identical, but this is rare as you will see from the data of the simulation runs. More typical is host B, where there is considerable difference between the two metrics.

b) Assigning buddies: Assigning buddies is trivial and is done with a random number generator. We generate one buddy per task in these simulation tests.

c) Network View: A host's view of the network is determined by taking the actual %LOAD of each host, generated as described above, and perturbing it by some amount chosen from a uniform distribution in the interval +− VIEW_SIZE_PERTUR about the actual %LOAD. Each host's view of each other host is determined in this manner.

Two main input parameters of the simulation were determined by looking at the generic requirements of a real command and control application currently under development. In this application there are to be 20-100 general purpose processors, and 4-8 tasks active per host. Given this basic information, we then studied a wide range of external arrival rates, computation time requirements, and deadline values. Changing the seeds of the random number generators give different loads, so it is very difficult to present confidence intervals. Instead, hundreds of simulation tests were performed with the decentralized algorithm always outperforming the centralized algorithm.

Many overheads are accounted for in the simulation program including the cost of performing the local check to determine if a task can execute at the site of the buddy (includes the cost of the guarantee routine $O(n^2)$, the cost of performing the focussed addressing, and the cost of bidding. The overheads associated with bidding

are substantial and include:

- RFB processing,
- transport of RFB,
- wait for dispatcher at receiving site,
- process incoming RFB and create a bid,
- transmit the bid,
- wait for dispatcher at original site after (some) bids are returned or a timer fires,
- process bids, and
- reassign task.

While the reallocation algorithm is executing, current tasks are also executing in parallel, and new external arrivals may occur. The external arrival rates are parameters of the individual test.

A second simulation program was implemented for the *centralized model.* The same scheme was used to configure the network so that identical situations existed for both algorithms. The centralized algorithm ran on the system processor of host 1. We assumed that the algorithm had up to date information about all the hosts of the network and performed the reallocation for all the tasks on the failed processor using the same guarantee algorithm used at each site in the decentralized algorithm. Since the guarantee algorithm is $O(n^2)$, we modelled the cost of the centralized algorithm execution as $O(n^2)$. This is the only overhead of the centralized algorithm modeled in the simulations. If the node on which the centralized algorithm was to executes fails, we assume that there is a backup. The cost of keeping the backup site up-to-date is not accounted for in the simulations.

4.2 Performance Results

This section contains the description of the main simulation results. The simulation of a decentralized reallocation algorithm in a hard real-time environment is quite complicated and many parameters are of interest. Due to space limitations we cannot present results with respect to each parameter. Instead, we discuss the results of the most important and interesting ones. We present these results in several categories: 1) Basic comparison of the decentralized and centralized algorithms, 2) modifying

unit costs of the algorithms, 3) the effect of reallocation on external arrivals, 4) the effect of the worst case computation time, C, distribution, 5) the effect the quality of state information has on focussed addressing, and 6) larger networks.

4.2.1 Basic Comparison

Figures 1, 2, and 3 show results for different load patterns. Figure 1's load pattern is that Host 1 is lightly loaded, and the others are approximately equally loaded. Figure 2's load pattern is a more arbitrary unbalanced situation with no lightly loaded host, and Figure 3's load pattern is that all hosts are approximately equally loaded. Within each load pattern, a range of loads is tested, and labelled A), B), C), etc. In these figures, the %LOAD, %LOAD(D) and number of tasks metrics are all presented for hosts 1-4 inclusive. Host 4 is the host that fails. These figures also show the success ratio of the decentralized versus the centralized algorithms. For the decentralized algorithm the figures indicate the number of tasks that each host are responsible for (RES), the number of tasks locally guaranteed (LG), the number of tasks guaranteed via focussed addressing (FA), and the number of tasks guaranteed via bidding (BID). Some later figures also show the number of tasks locally preempted (LPE) whenever this situation actually occurs. The success ratio in % of tasks from the failed host that are subsequently guaranteed by all means is presented under TOTAL, for both the centralized and decentralized algorithms. Note, that without a reallocation algorithm all tasks on the failed host will miss their deadline so the success ratio is zero. A major advantage of our approach is that it does not affect currently guaranteed tasks, so running the centralized or decentralized reallocation algorithm is a net improvement as long as at least 1 task is successfully reallocated, and this reallocated task does not cause one or more future external arrivals to not be guaranteed (see subsection below on external arrivals). Now consider each of the Figures 1, 2 and 3, in turn.

Figure 1 shows that for specific loads A) and B) all tasks are guaranteed locally (100%) for the decentralized algorithm, while the centralized algorithm only guarantees 75% of the tasks. This type of result was typical for over a hundred runs made with non severe conditions. That is, it was typical for the decentralized algorithm (which nicely partitions the responsibility for tasks) to guarantee all, or nearly all, of the tasks while the centralized algorithm did not. Note that there is always the possibility that an individual task cannot be guaranteed because it had a very close deadline when the failure occurred. Under more demanding conditions, such as in load C) in Figure 1, Hosts 2 and 3 are not able to perform local guarantees for all the tasks (for example, Host 2 is responsible for 2 tasks and locally guarantees only 1), but subsequently make use of FA (since Host 1 is lightly loaded) to guarantee

FIGURE 1: UNBALANCED - ONE VERY LIGHTLY LOADED HOST

		LOAD					SUCCESS RATIO DECENTRALIZED			CENTRALIZED
		H1	H2	H3	H4		H1	H2	H3	ALL HOSTS
	%L	.24	.51	.60	.60	RES.	1	3	0	
(A)	%L(D)	.55	.38	.52	.54	LG	1	3	0	
	NO.	1	3	4	4	FA	0	0	0	
						BID	0	0	0	
						TOTAL		100%		75%
	%L	.24	.80	.78	.70	RES.	1	0	3	
(B)	%L(D)	.55	.50	.77	.52	LG	1	0	3	
	NO.	1	5	4	4	FA	0	0	0	
						BID	0	0	0	
						TOTAL		100%		75%
	%L	.24	.82	.87	.88	RES.	2	2	2	
(C)	%L(D)	.55	.53	.81	.67	LG	2	1	0	
	NO.	1	5	5	6	FA	3	0	0	
						BID	0	0	0	
						TOTAL		100%		33%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 2: UNBALANCED: NO LIGHTLY LOADED HOST

		LOAD					SUCCESS RATIO			
							DECENTRALIZED			CENTRALIZED
		H1	H2	H3	H4		H1	H2	H3	ALL HOSTS
	%L	.49	.64	.87	.88	RES.	2	2	2	
(A)	%L(D)	.75	.44	.81	.67	LG	2	2	0	
	NO.	2	4	5	6	FA	0	0	0	
						BID	1	0	0	
						TOTAL		83%		33%
	%L	.49	.76	.89	.82	RES.	2	0	4	
(B)	%L(D)	.75	.50	.80	.66	LG	1	0	2	
	NO.	2	4	6	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		50%		16.6%
	%L	.49	.64	.87	.88	RES.	2	2	2	
(C)	%L(D)	.75	.44	.81	.67	LG	2	2	0	
	NO.	2	4	5	6	FA	0	0	0	
						BID	1	0	0	
						TOTAL		83%		33%
	%L	.68	.67	.89	.82	RES.	2	0	4	
(D)	%L(D)	.72	.46	.80	.66	LG	1	0	2	
	NO.	3	4	6	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		50%		16.6%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 3: BALANCED

		LOAD					SUCCESS RATIO			
							DECENTRALIZED			CENTRALIZED
		H1	H2	H3	H4		H1	H2	H3	ALL HOSTS
	%L	.49	.51	.58	.59	RES.	2	1	1	
(A)	%L(D)	.75	.37	.61	.51	LG	2	1	1	
	NO.	2	3	4	4	FA	0	0	0	
						BID	0	0	0	
						TOTAL		100%		75%
	%L	.68	.67	.70	.70	RES.	1	3	1	
(B)	%L(D)	.72	.46	.74	.41	LG	0	3	1	
	NO.	3	4	4	5	FA	0	0	0	
						BID	0	0	0	
						TOTAL		80%		80%
	%L	.90	.83	.86	.88	RES.	1	3	2	
(C)	%L(D)	.76	.54	.55	.88	LG	0	2	2	
	NO.	4	5	6	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		66%		20%
	%L	.90	.95	.89	.88	RES.	1	3	2	
(D)	%L(D)	.76	.59	.59	.67	LG	1	2	1	
	NO.	4	6	5	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		66%		20%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

its second task. In total, 3 tasks are guaranteed via FA, so again 100% of the tasks are guaranteed. Note that bidding is not needed, so its overhead is not incurred. For load C) the centralized algorithm only guarantees 33% of the tasks, even though the system is capable of guaranteeing 100% of the tasks. This is strictly due to the computation time needed by the centralized algorithm to make its decision since its work is not partitioned. We made several additional runs (not shown in the Figure) where the centralized cost was modelled as linear. We did this to determine the effect on the results if the average cost of running the centralized algorithm was linear and not $O(n^2)$. The success ratio for the centralized algorithm was 75%, 75% and 83% for loads A), B) and C) respectively. This is better, but still not as good as the decentralized case which includes a squared cost at each site (but the work is partitioned), and all the costs of FA, bidding, and local preemption.

Figure 2 again shows the utility of partitioning the reallocation responsibility because a large percentage of the tasks are locally guaranteed under a different load pattern. For this load pattern, FA does not help because there is no host viewed as being lightly loaded, so no tasks are assigned via FA. Bidding is beneficial under loads A) and C) where an additional task is guaranteed, even with the high cost of bidding. It is typical, under this load pattern to find bidding useful about 50% of the time, while FA is not useful. From Figure 2, the decentralized algorithm again clearly outperforms the centralized algorithm, i.e., the success ratio was 83% to 33%, 50% to 16.6%, 83% to 33%, and 50% to 16.6% for loads A), B), C) and D), respectively, in favor of the decentralized algorithm. When linear costs were assumed for the centralized algorithm, the success ratio improved to 66% for each of the four loads A), B), C) and D). This is better than the decentralized algorithm for cases B) and D) where the success ratio was only 50%. This anomaly is due to the fact of random assignment of responsibility in the decentralized algorithm where Host 3 was responsible for 4 tasks at squared cost. For a fairer comparison, we need to model a linear cost for the decentralized algorithm too. After doing this, the success ratio for the decentralized algorithm with linear cost improved to 100%, 83%, 83% and 83% for the four loads A), B), C), and D), respectively. Again, the decentralized algorithm is better than the centralized algorithm for all loads. The loads B) and D) show that our random assignment of buddies sometimes causes and imbalance of responsibility and a possible loss in performance. If this fairly rare condition is deemed important to avoid, it is easy to enforce a balanced level of responsibility. However, a balanced level of responsibility is not without cost, because a certain amount of coordination costs would be required to enforce a balance at all times. Our view is that this extra coordination cost is not worth it, because in all but a few rare cases we get excellent results with the random assignment of responsibility at zero coordination costs.

The results presented in Figure 3 indicate that when the network is balanced, neither FA nor bidding improve the success ratio. This is due to the fact that if a host cannot locally guarantee a task, other hosts are just as busy, and they might have even increased their load by their own local guarantees. Of course, in our many runs we did see isolated situations where FA and bidding increased the success ratio even in a balanced system, but such results are rare. The results if these tests indicate that a more adaptive version of our algorithm might attempt to recognize a balanced network load, ignore application of FA and bidding under these conditions, and instead, go directly to local preemption. Comparison of the success ratios of the decentralized and centralized algorithms (Figure 3) again show the superiority of the decentralized algorithm under this load pattern, i.e., 100% to 75%, 80% to 80%, 66% to 20%, and 66% to 20% in favor of the decentralized algorithm in spite of the fact that FA and bidding are not helpful. Hence the improvement is due to lower cost of the decentralized algorithm because it partitions the work.

4.2.2 Unit Costs

In all the previously presented results, the decentralized algorithm outperforms the centralized algorithm. Obviously, the comparison is heavily dependent on the costs assumed for each of the algorithms. We were realistic about these costs. For example, since the algorithms must attempt a guarantee for each task for which it is responsible, and the guarantee algorithm is $O(n^2)$ we used a squared cost for each algorithm. However, implicit in applying this cost is some unit cost which is then squared. In the previous runs the unit cost was 5 units. Figure 4 shows the results when we reduce this unit cost to 3 units and then to 1 unit. The decentralized algorithm is still better than the centralized algorithm at all unit costs, although the difference is smaller at smaller unit costs. The results shown in Figure 4 span the three load patterns presented in Figures 1-3 inclusive. For example, load A) in Figure 4 is a representative of the imbalanced load pattern, and the results are 66.6% to 50%, 50% to 50% and 50% to 16.6% for unit costs 1, 3, 5, respectively, in favor of the decentralized algorithm. Similar results are shown in Figure 4 for the other two load patterns, i.e., load B) is the balanced load pattern, and load C) is the "1 host lightly loaded, the others approximately equally loaded" load pattern.

4.2.3 External Arrivals

Reallocation adds extra work to each site. If this extra work causes subsequent external arrivals to miss their deadlines, then the net effect of the reallocation is diminished or possibly even negative. Two of the more important parameters in de-

FIGURE 4: UNIT COSTS

		LOAD				UNIT COSTS	SUCCESS RATIO	
							DECENT.	CENTR.
		H1	H2	H3	H4			
	%L	.49	.76	.89	.82	1	66.6%	50%
(A)	%L(D)	.75	.50	.80	.66	3	50%	50%
	NO.	2	5	6	6	5	50%	16.6%
	%L	.68	.79	.78	.74	1	60%	40%
(B)	%L(D)	.72	.52	.60	.66	3	60%	40%
	NO.	3	5	5	5	5	40%	20%
	%L	.24	.68	.61	.63	1	100%	75%
(C)	%L(D)	.55	.46	.69	.54	3	100%	100%
	NO.	1	4	3	4	5	100%	75%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

termining the effect of reallocation on external arrivals, is the load on the network at the time of the failure, and the external arrival rate. We assume that the interarrival times are exponential random variables with averages 90, 70, 60, 50, 40 and 30 time units at each host. Since the average service time is 38 units, the runs with 40 and 30 time units as averages are very demanding, and overloaded, respectively. For network loads similar to the ones found in earlier portions of this paper, the effect of reallocation on external arrivals was minimal (almost 0) for average interarrival times of 50, 60, 70, and 90 time units. This is primarily due to the fact that the extra load imposed by reallocation is not considerable compared to the laxity of a new external arrival. In other words, the new arrivals tend to let the reallocated tasks run first, and there still exists enough time for them to execute before their deadlines. This condition does not hold when external arrivals are very frequent. As a typical example consider the interarrival rate of 40 units. A typical test result was that the decentralized algorithm success ratio was 5 out of 8 tasks, but subsequently 2 external arrivals were lost - a net gain of only 3 tasks, not 5. For the same test, the centralized algorithm successfully reallocated 4 out of 8, but lost 5 external arrivals for a net loss of 1 task. When the interarrival time was decreased to 30 units, the net effect for the decentralized algorithm was negative (5 reallocated and 6 lost external arrivals), and neutral for the centralized algorithm (4 reallocated and 4 lost external arrivals). Tests results such as these substantiate an intuitive notion, that if the system is overloaded, there is no net worth in performing reallocations, and, in fact, it can be harmful if the algorithm doesn't have mechanisms to turn itself off. Consequently, we recommend adding a factor to our algorithm that would attempt to monitor the load on the system, and turn off reallocation when the load is too high. This simple scheme would have to be slightly modified if it was policy that the priority of tasks was the prime factor during reallocation.

4.2.4 Computation Time Distribution

The laxity (deadline minus remaining computation time) of a task is an extremely important factor in the success of any on-line algorithm. If all laxities are very small, then there is no point to on-line decision making; as laxities increase we can expect better results from on-line algorithms. Consequently, if we run tests with increasing the computation time requirements with fixed deadlines (thereby decreasing laxity), then we expect a net decrease in the number of tasks successfully reallocated. This result was observed over many tests. However, another issue is the relative performance of the centralized to the decentralized algorithms over a range of computation times. Recall that most of the results presented earlier are with a computation time for a task given by a uniform distribution in the range of 15-50 units. In adition, we

tested under uniform distributions over the ranges 30-60, 40-70, 10-90, and 30-100 units. The decentralized algorithm performed better than the centralized algorithm for all three load patterns and over all computation time distributions tested. To save space we will not present the actual data from which this conclusion was reached. Of course, particular tasks with high computation time requirements were not often successfully reallocated in either case, as is to be expected.

4.2.5 Quality of State Information

Focussed addressing is one aspect of our reallocation algorithm which requires approximate information about the state of other hosts. In these tests we used an approximation of the %LOAD factor as an indicator of load. We then tested the effect on performance of an error in a host's view of this %LOAD factor. All previous tests used a +- 10% factor, meaning that a random number in the interval +- 10% about the true %LOAD was used as a host's view of the load at another site. We also then tested +-20%, +-30%, and +-40% for each of the three load patterns.

Figure 5 shows that when the load pattern was all hosts equally busy and the load was approximately 75% at each site, the success ratio of 40% was unaffected by poor state information. This is, in part, due to the fact that in most cases a host's view of another host was that this other host was too busy for it to be considered as a FA host, and, in part, to the fact that when a task was sent to a perceived FA host it was not accepted due to the heavy load at that site.

Figure 6 illustrates an *improvement* in success ratio from 33% to 50% when there is poor information. This is typical under this load pattern because under heavy but unbalanced loads, a site often erroneously considers a site as lightly loaded, sends work, and luckily, there is sometimes enough free time within the specific time bounds of this task to guarantee it. The FA decision is made quickly at low cost. Bidding, because of its higher cost, is not able to guarantee these extra tasks in this case (but it can in other cases - see next paragraph). We have found similar results in our other decentralized scheduling work [8][11], and the solution is to perform FA on the host considered most lightly loaded, but in parallel proceed with bidding. Employing this scheme we were able to attain better overall results in normal task scheduling and would expect the same results with respect to reallocation.

Figure 7 is for the load pattern where one site is lightly loaded and the others are heavily loaded. Here FA is used extensively (i.e., 50% of the tasks successfully reallocated were via FA) when the quality of the information is good (as in the +-10% case). This data appears in the section labelled 10) in Figure 7. As the quality of information degrades, fewer tasks are moved by FA (from 3 tasks, to 2 tasks, to 2 tasks, then to only 1 task as the view degrades from 10-20-30-40%), but the slack is

FIGURE 5: FOCUSSED ADDRESSING - LOAD A

		LOAD				%PERTURB.	SUCCESS RATIO %SR	%FA
		H1	H2	H3	H4			
	%L	.68	.79	.78	.74	+-10	40%	0%
(A)	%L(D)	.72	.52	.60	.66	+-20	40%	0%
	NO.	3	5	6	6	+-30	40%	0%
						+-40	40%	0%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 6: FOCUSSED ADDRESSING - LOAD B

		LOAD				%PERTURB.	SUCCESS RATIO %SR	%FA
		H1	H2	H3	H4			
	%L	.68	.67	.89	.82	+-10	33%	0%
(B)	%L(D)	.72	.46	.80	.66	+-20	33%	0%
	NO.	3	4	6	6	+-30	50%	33%
						+-40	50%	33%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 7: FOCUSSED ADDRESSING - LOAD C

ERROR		LOAD					SUCCESS RATIO DECENTRALIZED		
		H1	H2	H3	H4		H1	H2	H3
	%L	.24	.82	.87	.88	RES.	2	2	2
±10)	%L(D)	.55	.53	.81	.67	LG	2	1	0
	NO.	1	5	5	6	FA	3	0	0
						BID	0	0	0
						TOTAL		100%	
	%L					RES	2	2	2
±20)	%L(D)		SAME			LG	0	0	0
	NO.					FA	2	0	0
						BID	2	0	0
						TOTAL		66.6%	
	%L					RES	2	2	2
±30)	%L(D)		SAME			LG	2	0	0
	NO.					FA	1	0	0
						BID	2	0	0
						LPE	0	1	0
						TOTAL		100%	
	%L					RES	2	2	2
±40)	%L(D)		SAME			LG	2	0	0
	NO.					FA	0	0	1
						BID	3	0	0
						TOTAL		100%	

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

taken up by bidding which increases from 0 tasks, to 2 tasks, to 2 tasks, to 3 tasks as the view degrades from 10-20-30-40%. It is for this reason that we cannot just use FA, but need to proceed with FA and bidding in parallel. In this test the deadlines of the tasks to be reallocated were sufficiently long to allow bidding to be successful. It is easy to hypothesize other situations where the poor quality of state information would reduce FA, as in this figure, but where subsequent bidding would then not be successful because deadlines are too close. This is the situation which would be most affected by poor quality of state information. In summary, we believe that our algorithm is quite robust to state information sharing since it is used in limited contexts, and the affect of its degradation is usually ameliorated by bidding.

4.2.6 Larger Networks

Having performed the majority of our tests simulating a four host network, we increased the size of the network to 8, where host 8 fails. Over many tests we observed the same general result, the decentralized algorithm significantly outperforms the centralized algorithm. Figures 8.1, 8.2 and 8.3 show typical test results. Figure 8.1 contains the data for loads A and B, Figure 8.2 for loads C and D, and Figure 8.3 for load E. For load A), 66% of the tasks are reallocated via local guarantee (and none by FA, bidding or local preemption), while the centralized algorithm is successful with only 16.6%. In load B) our decentralized algorithm is successful with 100% of the tasks, but does preempt a low priority task at host 1. This compares to only a 50% success ratio for the centralized algorithm under the same conditions. Load C) produces as 100% success ratio for the decentralized algorithm, primarily by FA, while the centralized algorithm only guarantees 33%. Load D) is even more dramatic, producing a 100% to 0% result in favor of the decentralized algorithm. However, in load D) the decentralized algorithm does preempt 3 low priority tasks to achieve this success ratio, one each at sites 2, 4 and 5. Local preemption occurred because tasks were sent to sites 2, 4 and 5 via FA, but were subsequently not guaranteed. At this point, bidding was not attempted because there was not enough time for bidding, so preemption was tried and was successful. If bidding is attempted before preemption and bidding is not successful, then chances are good that local preemption will also not be successful. That is, the algorithm will have waited too long before preempting. This is why we rarely see any local preemptions in these test results. It is a policy decision that the designers of the system under consideration must make - the order in which to employ the various parts of the reallocation algorithm.

Finally, the results from load E) are presented to emphasize that the local guarantee is quite useful, and that bidding is also sometimes useful. Load E) results in a 100% to 33% success ratio in favor of the decentralized algorithm. Note that while

FIGURE 8.1 - LARGER NETWORK - LOADS A AND B

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.90	.95	.89	.88	.84	.86	.74	.91
A)	%L(D)	.76	.59	.72	.67	.75	.79	.53	.72
	NO.	4	6	5	6	5	6	5	6

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES	2	1	0	1	1	0	1	
LG	2	1	0	1	0	0	0	
FA	0	0	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
TOTAL			66.6%					16.6%

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.24	.82	.87	.88	.83	.85	.85	.60
B)	%L(D)	.55	.53	.81	.67	.74	.63	.66	.72
	NO.	1	5	5	6	4	5	5	4

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES	1	0	0	1	1	1	0	
LG	1	0	0	0	1	1	0	
FA	0	0	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
LPE	1	0	0	0	0	0	0	
TOTAL			100%					50%

FIGURE 8.2 - LARGER NETWORKS - LOADS C and D

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.49	.47	.72	.79	.78	.90	.79	.55
C)	%L(D)	.75	.35	.75	.54	.56	.70	.71	.65
	NO.	2	3	3	4	4	5	6	3

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES	0	0	1	0	0	1	1	
LG	0	0	0	0	0	0	0	
FA	0	2	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
TOTAL				100%				33%

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.45	.48	.60	.55	.49	.52	.30	.57
D)	%L(D)	.76	.59	.78	.78	.71	.79	.72	.84
	NO.	4	6	7	7	6	6	4	7

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES	2	2	0	0	1	0	2	
LG	1	1	0	0	1	0	1	
FA	0	0	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
LPE	0	1	0	1	1	0	0	
TOTAL				100%				0%

FIGURE 8.3 - LARGER NETWORKS - LOAD E

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.45	.48	.75	.79	.78	.90	.82	.87
E)	%L(D)	.31	.64	.73	.58	.57	.70	.51	.68
	NO.	3	3	5	6	6	6	5	6

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES	0	1	2	0	2	0	1	
LG	0	1	2	0	1	0	1	
FA	0	0	0	0	0	0	0	
BID	1	0	0	0	0	0	0	
TOTAL				100%				33%

FOR FIGURES 8.1, 8.2, and 8.3
C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

bidding is not always useful, it is useful in certain circumstances where otherwise there would be no chance at success.

5 Reallocation Algorithm Revisited

Since we are interested in decentralized control (coordination), reliability and performance, let's revisit the main steps of the algorithm with respect to these issues. See Tables 2 and 3.

During normal system operation, steps 1 and 2 (Table 2), the algorithm is highly autonomous, reliable and performs well. There is no need for coordination, but in some sense there has been an a priori agreement (a protocol) made by each site. The same argument applies during host failure processing for steps 3, 4, 5, and 6a, b, c. See Table 3. Only in step 6d does explicit cooperation occur, but at a high cost. Based on the simulation results, the decision to treat bidding as a last resort seems to be a good one.

TABLE 2: LEVEL OF COOPERATION - NORMAL MODE

Algorithm Step	Dec.Control	Reliability	Performance
DURING NORMAL SYSTEM OPERATION			
1)choose buddies	each site on a per task basis; decisions made decentralized; no coordination	all up sites can continue	ovhd of choosing buddies is small, distributed; ovhd of broadcast and commit acceptable
2)deactivate buddies	task completion at any site invokes the deactivate operation; decisions made decentralized; no coordination;	at task level; all up sites can continue	ovhd is small

TABLE 3: LEVEL OF COOPERATION - FAILURE MODE

ON HOST FAILURE

Algorithm Step	Dec.Control	Reliability	Perf.
3)find set resp. for	n sites, roughly in in parallel determine this; no cooperation	each site can perform autonomously;all up sites can cont.	fast;no delay for cooperation
4)attempt local guarantee	same	same	same
5)for small laxity preempt locally	same	same	same
6) a) for not small lax., check state, apply policy	same	same	same
b) -FA	same	same	same
c) -Preempt Locally	same	same	same
d) -bidding	n-pairwise negot.	high	slow

Tables 2 and 3 illustrate that any sophisticated *decentralized* algorithm will contain many steps, and/or phases being executed by multiple distributed agents. Each step of the algorithm at each agent of the decentralized algorithm could exhibit a different degree of decentralization. There is a basic tradeoff between information exchange used for coordination and/or negotiation among the decentralized agents running the algorithm and speed of execution (i.e., of making the decision). As in the above reallocation algorithm, it seems that most decentralized system algorithms will contain *points of automony* and *points of coordination.* For example, it is easy to modify the above algorithm by changing points of cooperation. Suppose we add a step 5.5 which required that every host exchange state information. This cooperation would occur after local guarantees, but before we attempt any distributed reallocation. This exchange would increase the quality of the state information available, but cause additional delays. If one eliminates coordination altogether, you end up with a totally decentralized algorithm where each agent is making decisions based only on

local information. Finding the cost-performance tradeoff curve for a given algorithm in a given environment so that the algorithm has the "best" amount of coordination to achieve good performance under acceptable cost is the design problem generally faced. We have attempted to make these tradeoff choices in designing the algorithm presented above.

6 Summary

We have developed and analyzed a decentralized task reallocation algorithm for hard real-time systems. The main properties of the algorithm are that it is fast, decentralized, reliable, specifically considers deadlines of tasks, attempts to utilize all the nodes of a distributed system to achieve its objective, handles tasks in priority order, and separates policy and mechanism. Another significant advantage of our approach is that it allows continued operation of the guaranteed tasks on the non-failed processors while reallocation is in progress. The reallocation algorithm itself is $O(n)$, but it invokes the guarantee algorithm which is $O(n^2)$. Consequently, the reallocation process is $O(n^2)$. An extensive performance analysis of the algorithm via simulation has shown that it is quite effective in performing reallocations, and is significantly better than a centralized approach. In particular, the bias that the algorithm has towards making decisions quickly is beneficial in a hard real–time environment. The algorithm neatly partitions the workload, and this simple step provides significant performance improvement. We also show that FA, bidding and local preemption are useful features, but that global preemption and groups bidding are probably not necessary. We evaluated our algorithm under many conditions including various loads, computation time distributions, network sizes, and unit costs. We studied the effect of the quality of state information on the FA part of the algorithm, and the effect of the reallocation algorithm on new external arrivals. In all cases, the decentralized algorithm was effective and outperformed the centralized algorithm. In our tests, the number of tasks to be reallocated varied from 3–8. In other environments where significantly larger number of tasks are to be reallocated, the decentralized algorithm should perform even better than the centralized algorithm. Finally, while no simulation results for multiple host failures are reported in this paper, our decentralized algorithm does automatically handle multiple host failures.

7 Acknowledgements

I would like to thank John Lehoczky and Lui Sha for various discussions on this material early in its development.

References

[1] Carlow, G., Architecture of the Space Shuttle Primary Avionics Software System, *CACM*, Vol. 27, No. 9, September 1984.

[2] Dasarathy, B., Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them, *IEEE Trans on Software Engineering*, Vol. SE-11, No. 1, January 1985.

[3] Leinbaugh, D., Guaranteed Response Times in a Hard Real-Time Environment, *IEEE Trans on Software Engineering*, Vol. SE-6, January 1980.

[4] Liu, C, and J. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *JACM*, Vol. 20, No. 1, January 1973.

[5] Mok, A., and L. Dertouzos, Multiprocessor Scheduling in a Hard Real-Time Environment, *Proc. Seventh Texas Conf. on Computer Systems*, Nov. 1978.

[6] Ousterhout, J., D. Scelza, and P. Sindhu, Medusa: An Experiment in Distributed Operating System Structure, *CACM*, Vol. 23, No. 2, February 1980.

[7] Ramamritham, K. and J. Stankovic, Dynamic Task Scheduling in Distributed Hard Real-Time Systems, *IEEE Software*, Vol. 1, No. 3, July 1984.

[8] Ramamritham, K., J. Stankovic, and W. Zhao, Distributed Scheduling of Hard Real Time Tasks Under Resource Constraints in the Spring System, submitted to *IEEE Trans on Computers*, February 1986.

[9] Sha, Lui, J. Lehoczky, and R. Rajkumar, Solutions for Some Practical Problems in Prioritized Preemptive Scheduling, *Proc. 1986 Real-Time Systems Symposium*, December 1986.

[10] Siewiorek, D., and R. Schwarz, *The Theroy and Practice of Reliable System Design*, Digital Press, 1982.

[11] Stankovic, J, K. Ramamritham, and S. Cheng, Evaluation of a Bidding Algorithm for Hard Real-Time Distributed Systems, *IEEE Trans on Computers*, Vol. C-34, No. 12, December 1985.

[12] Stankovic, J and D. Towsley, Dynamic Reallocation in a Highly Integrated Real-Time Distributed System, *Proc. Sixth Int. Conf. on Distributed Computing Systems*, May 1986.

[13] Stankovic, J. *Reliable Distributed System Software*, IEEE Computer Society Press, 1985.

[14] Ward, Paul, and S. Mellor, *Structured Development for Real-Time Systems*, Yourdan Press, Vol. 1 and Vol. 2, N.Y., N.Y., 1985.

[15] Zhao, W., K. Ramamritham, and J. Stankovic, Scheduling Tasks with Resource Requirements in Hard Real-Time Systems, *IEEE Trans on Software Engineering*, Vol. SE-13, No. 5, May, 1987.

[16] Zhao, W., K. Ramamritham, and J. Stankovic, Preemptive Scheduling Under Time and Resource Constraints, *IEEE Trans on Computers*, Vol. C-36, No. 8, August 1987.