

**SCHEDULING PROBLEM IN DISTRIBUTED HARD REAL-TIME
COMPUTER SYSTEMS :**

**A Simulation Approach to Dynamic Task Scheduling
Using Focused Addressing and Bidding**

by

Z. Dilek Duman Tuzun

B.S. in Computer Engineering, Boğaziçi University, 1987

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

Boğaziçi University

1990

Y. G.

**Yükseköğretim Kurulu
Dokümantasyon Merkezi**

**SCHEDULING PROBLEM IN DISTRIBUTED HARD REAL-TIME
COMPUTER SYSTEMS :**

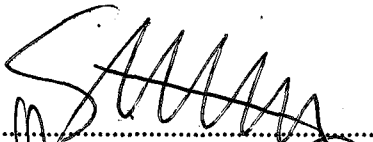
**A Simulation Approach to Dynamic Task Scheduling
Using Focused Addressing and Bidding**

APPROVED BY

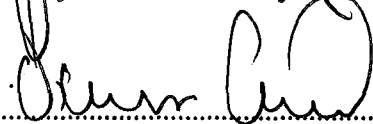
Doç. Dr. Oğuz Tosun
(Thesis Supervisor)


.....

Doç. Dr. Selahattin Kuru


.....

Doç. Dr. Ömer Cerid


.....

DATE OF APPROVALJuly...27,1990.....

ACKNOWLEDGEMENTS

I wish to express my deep gratitude to Doç. Dr. Oğuz Tosun for his guidance and help. I would like to thank to Doç. Dr. Selahattin Kuru, and to Doç. Dr. Ömer Cerid for their valuable comments. Thanks to Sema F. Akgün for sharing her experiences with me. My sincere appreciation goes to Rasim Mahmutogulları for his editorial help, and critical and detailed reading of my draft. A very large thanks is extended to my husband for his genuine understanding. Special thanks go to my baby yet unborn for the constant encouragement of working more and harder.

Z. Dilek Duman Tüzün

ABSTRACT

The unique feature, the time constraint, of hard real-time systems makes them different from the traditional computer systems because in such systems the required tasks must be executed not only functionally correctly but in a timely manner. In this thesis, the scheduling problem of hard real-time tasks in distributed systems is examined in detail. Previous work on the algorithms proposed for scheduling in hard real-time systems is reviewed. A heuristic algorithm which considers not only CPU scheduling but also general resource requirements of tasks is chosen to be evaluated. A set of heuristics that can be used by this algorithm is studied through a sequence of simulations. The heuristic function which is observed to perform the best is incorporated in the distributed scheduling algorithm. In this algorithm the determination of a good destination node for a locally nonguaranteed task, is based on a technique that combines bidding and focused addressing algorithms. Simulation studies are conducted in order to evaluate the performance of the algorithm in a wide range of application environments. The performance of the algorithm is also compared to that of three other distributed scheduling algorithms. It is observed that though this algorithm is sensitive to the characteristics of the environments, it performs well in a wide range of environments, compared with the other algorithms.

ÖZET

Katı gerçek zamanlı sistemlerin özelliği olan süre sınırlaması, bu sistemleri geleneksel bilgisayar sistemlerinden farklı kılmaktadır, çünkü bu tür sistemlerde talep edilen işler yalnızca görev bakımından doğru olarak değil, aynı zamanda vaktinde yerine getirilmelidirler. Bu tezde dağıtılmış sistemlerde katı gerçek zamanlı işlerin planlanması sorunu ayrıntılı olarak incelenmektedir. Katı gerçek zamanlı işlerin planlanması konusunda şimdiye dek önerilmiş algoritmalar gözden geçirilmektedir. Sadece MIB planlamasını değil, işlerin genel kaynak ihtiyaçlarını da dikkate alan bir buluşsal algoritma değerlendirilmek üzere seçilmiştir. Bu algoritma tarafından kullanılabilinecek bir grup buluşsal fonksiyon, bir seri benzetim çalışması ile incelenmektedir. En iyi performansı verdiği gözlenen buluşsal fonksiyon dağıtılmış planlama algoritmasında kullanılmaktadır. Bu algorithmada yerel olarak garanti edilemeyen iş için hedef düğümün seçiminde pey sürme ve direkt gönderme algoritmalarını birleştiren bir teknik esas alınmaktadır. Algoritmanın performansını gözlemek amacı ile çeşitli uygulama şartları için benzetim çalışmaları yapılmaktadır. Algoritmanın performansı başka uç değişik dağıtılmış planlama algoritmasının performansları ile de karşılaştırılmaktadır. Bu algoritmanın, uygulama alanının özelliklerine hassas olmasına rağmen, geniş bir uygulama alanı içinde, diğer algoritmalara kıyasla iyi bir performans gösterdiği gözlenmektedir.

TABLE OF CONTENTS

	<u>Page</u>
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET.....	v
LIST OF FIGURES	ix
LIST OF TABLES.....	xi
LIST OF SYMBOLS.....	xii
I. INTRODUCTION.....	1
II. SCHEDULING.....	4
2.1. Distributed Systems and Scheduling.....	4
2.2. Real-Time Systems.....	6
2.3. Scheduling in Hard Real-Time Systems	7
III. DISTRIBUTED SCHEDULING ALGORITHMS.....	10
3.1. Hierarchical Classification.....	10
3.2. Flat Classification Characteristics.....	14
3.3. Application of Taxonomy to Some Examples.....	16

IV. LITERATURE SURVEY ON SCHEDULING ALGORITHMS FOR HARD REAL-TIME SYSTEMS	18
4.1. Literature Survey for Multiprocessor Systems	18
4.1.1. Static Scheduling Algorithms for Multiprocessor Systems.....	18
4.1.2. Dynamic Scheduling Algorithms for Multiprocessor Systems.....	21
4.2. Literature Survey for Distributed Systems	22
4.2.1. Static Scheduling Algorithms for Distributed Systems.....	22
4.2.2. Dynamic Scheduling Algorithms for Distributed Systems.....	25
V. OVERVIEW OF THE SCHEDULING SCHEME	27
VI. LOCAL SCHEDULER	31
6.1. Strategy Behind the Local Scheduler.....	31
6.2. The Basic Algorithm Underlying the Guarantee Routine.....	32
6.2.1. Scheduling versus Searching.....	32
6.2.2. Data Structures.....	33
6.2.3. A Constraint on the Search.....	36
6.2.4. The Basic Algorithm.....	37
6.3. Extension to the Basic Algorithm.....	38
6.4. The Heuristic Function H	45
6.4.1. Simple Heuristics for Scheduling.....	45
6.4.2. Simulation Method and Results.....	45
6.4.3. Integrated Simple Heuristic Algorithms.....	47
6.4.4. Simulation Results of Using Integrated Simple Heuristics.....	49
6.5. Application Considerations.....	51
6.5.1. On-line Heuristic Scheduling.....	51
6.5.2. Scheduling When Tasks Arrive in a Batch.....	52

6.5.3. Non-preemptive Scheduling and the Inclusion of Periodic Tasks.....	52
VII. DISTRIBUTED SCHEDULING SCHEME	54
7.1. Generation and Transmission of the Node Surplus.....	54
7.2. Focused Addressing and Requesting for Bids.....	55
7.3. Bidding	57
7.4. Bid Evaluation	58
7.5. Response to Task Award	59
7.6. Simulation Model.....	59
7.6.1. System Model.....	59
7.6.2. Node Model.....	61
7.7. Simulation Results and Observations.....	63
7.7.1. Effect of Laxity Distribution of Tasks	65
7.7.2. Effect of Communication Delay	66
7.7.3. Effect of System's Communication Network Topology	74
7.7.4. Comparison of Algorithm FB with Algorithms NC and R.....	75
7.7.5. Comparison of Algorithm FB with Algorithm B.....	79
VIII. CONCLUSION	83
APPENDIX A. LOCAL TASK GENERATOR	85
APPENDIX B. LOCAL SCHEDULING PROGRAM	89
APPENDIX C. GLOBAL TASK GENERATOR	96
APPENDIX D. SCHEDULING PROGRAM USING BIDDING AND FOCUSED ADDRESSING.....	98
BIBLIOGRAPHY	128
REFERENCES NOT CITED	132

LIST OF FIGURES

	<u>Page</u>
FIGURE 3.1 Structure of the hierarchical classification.....	11
FIGURE 5.1 Structure of the local scheduler on a node	28
FIGURE 6.1 A search tree for a set of four tasks.....	34
FIGURE 6.2 Basic local scheduling algorithm for guarantee routine.....	38
FIGURE 6.3 The algorithm of the Limited_Backtracker.....	40
FIGURE 6.4 Data structure used to implement a task node.....	41
FIGURE 6.5 Illustration of the extended algorithm by a simplified example	44
FIGURE 7.1 Simulation system model 1 (Fully Connected)	60
FIGURE 7.2 Simulation system model 2 (Star).....	60
FIGURE 7.3 Effect of task laxity when R=16/600 and Topology=FC.....	67
FIGURE 7.4 Effect of task laxity when R=16/600 and Topology=S.....	67
FIGURE 7.5 Effect of task laxity when MD=36 and Topology=FC.....	68
FIGURE 7.6 Effect of task laxity when MD=36 and Topology=S.....	68
FIGURE 7.7 Effect of MD under M_LOAD and L_LAX.....	70
FIGURE 7.8 Effect of MD under M_LOAD and H_LAX	70
FIGURE 7.9 Effect of MD (details of FC_GNW)	72
FIGURE 7.10 Effect of MD (details of S_GNW)	72
FIGURE 7.11 Effect of MD under L_LAX and different system loads	73
FIGURE 7.12 Effect of system's communication network topology.....	74
FIGURE 7.13 Comparison of FB, R, and NC when Laxity=H_LAX and Topology=FC	76
FIGURE 7.14 Comparison of FB, R, and NC when Laxity=H_LAX and Topology=S	76

FIGURE 7.15 Comparison of FB, R, and NC when Laxity=M_LAX and Topology=FC.....	77
FIGURE 7.16 Comparison of FB, R, and NC when Laxity=M_LAX and Topology=S.....	77
FIGURE 7.17 Comparison of FB, R, and NC when Laxity=L_LAX and Topology=FC.....	78
FIGURE 7.18 Comparison of FB, R, and NC when Laxity=L_LAX and Topology=S.....	78
FIGURE 7.19 Comparison of FB and B when R=16/600 and Topology=FC.....	80
FIGURE 7.20 Comparison of FB and B when R=16/600 and Topology=S.....	80
FIGURE 7.21 Comparison of FB and B when MD=36 and Topology=FC.....	82
FIGURE 7.22 Comparison of FB and B when MD=36 and Topology=S.....	82



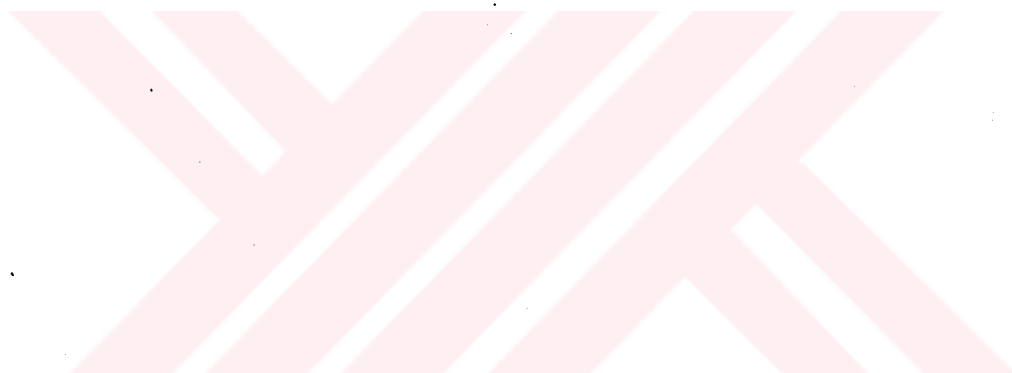
LIST OF TABLES

	<u>Page</u>
TABLE 3.1 Application of the taxonomy to some example algorithms	17
TABLE 6.1 Simulation results of using simple heuristics	48
TABLE 6.2 Simulation results of using integrated simple heuristics	50
TABLE 7.1 Nodes' local task arrival rates under different system loads	64

LIST OF SYMBOLS

$AT(T)$	arrival time of the task T
$C(T)$	worst case computation time of the task T
$D(T)$	deadline of the task T
$DRDR$	dynamic resource demand ratio for resources on a node
$DRDR_i$	dynamic resource demand ratio for resource i
EAT	earliest available times of resources on a node
EAT_i	earliest available time of resource i
$ES(T, i)$	estimated number of instances of task T that node N_i can guarantee
FAS	focused addressing surplus
$H(T)$	value of the heuristic function H for the task T
k	number of tasks in a task set
$L(T)$	laxity of the task T
$LBA(T)$	latest bid arrival time of the task T
MD	no conflict message delay
Min_C	minimum computation time first heuristic
Min_D	minimum deadline first heuristic
Min_L	minimum laxity first heuristic
Min_S	minimum start time first heuristic
$New_EAT(T)$	earliest available times of resources on a node if task T is scheduled next
$New_EAT(T)_i$	earliest available time of resource i if task T is scheduled next
N_i	node i

R	system local task arrival rate
$R(T)$	resource requirements of the task T
R_i	resource i
SGS	system-wide guarantee surplus
S_i	a set of tasks
$ST(T)$	earliest start time of the task T
T	a real-time task



I. INTRODUCTION

Recently, there has been an increased interest in hard real-time systems and such systems are becoming more and more sophisticated. Examples of this type of real-time systems are command and control systems, flight control systems, and the space shuttle avionics system.

Currently, the field of real-time scheduling is the focus of a great deal of research interest. This is because of the very frequent use of digital computers in real-time applications, growing sophistication in real-time software for the last few years, and an increased necessity in improving system performance and reliability.

In a hard real-time system, the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced. Usually, timing constraints are described in terms of deadlines by which computations of tasks must absolutely be met or the system will be considered to have failed. Further, if these real-time timing constraints are not met there may be potentially catastrophic consequences. Hence, the most critical part of supporting such new systems is the ability to guarantee that timing constraints can be met. Because of the large number of combinations of tasks that might be active at the same time and because of the continually varying demand patterns on the system, it is generally impossible to pre-calculate all possible schedules *off-line* to statically guarantee real-time timing constraints. This study concerns the scheduling algorithms for *on-line* dynamic guarantee of deadlines, in a hard real-time distributed computer system.

The problem of determining an optimal schedule is known to be NP-hard and is hence impractical for real-time task scheduling. The problem is even harder when, in addition to computation times and deadlines of tasks, their active and passive resource requirements are also accounted for. Optimal algorithms with polynomial time complexity exist only for a few restricted cases, for example, in the case where tasks having the same processing time and the same deadline are scheduled on two processors and in the case where there is only one type of resource. None of these cases represent practical situations. Moreover, it is impossible to find an optimal

schedule for a dynamic distributed system given the inherent communication delay. All of these factors necessitate a heuristic approach to scheduling.

In many hard real-time systems, tasks are scheduled dynamically and hence the scheduling algorithms used must have low run-time costs. In this study, a non-preemptive scheduling scheme is evaluated for such dynamic hard real-time distributed systems. In this scheme in addition to tasks' timing constraints their active and passive resource requirements are also taken into account. The scheme has two components : local scheduling (guaranteeing tasks that arrive dynamically at a node) and distributed scheduling (guaranteeing locally nonguaranteed tasks at remote nodes).

The heuristic algorithm developed by Zhao, Ramamritham, and Stankovic [1]* is chosen as the algorithm underlying the guarantee routine in local scheduling and is implemented with some modifications. The most critical point in local scheduling is the heuristic function used by the guarantee routine to select the task to be scheduled next. In order to keep run-time costs low, computationally simple heuristic functions are evaluated and the one which has the best performance is chosen for further exploration. The simulation results reported in Part VI show that because of the complexity of the problem, simple heuristics alone do not perform satisfactorily. However, an algorithm that uses a linear combination of simple heuristics in conjunction with limited backtracks works very well.

As mentioned before, in a hard real-time system, every task that misses its deadline can seriously degrade the performance of the system. Hence, even a small performance improvement should be considered significant in the context of hard real-time systems.

It should also be pointed out that the time complexity of this algorithm for scheduling a set of k tasks is k^2 , which is very much lower than that of an optimal exhaustive search algorithm which takes time proportional to $k!$. Hence, this is an attractive approach to overcome the exponential problem of scheduling.

When a task arriving at a node cannot be guaranteed at that node, the distributed scheduling problem comes into the picture. In that case, the local schedulers on individual nodes must interact and cooperate to determine which other node in the system can guarantee the task. The degree of this cooperation depends on

* References enclosed in brackets refer to the bibliography.

the algorithm used. In this study, an algorithm which combines bidding and focused addressing algorithms [2] is evaluated. The integrated simple heuristic, which was observed to have the best performance in Part VI, is incorporated in this distributed scheduling algorithm as the heuristic underlying the guarantee routine on each node. In the distributed scheme the guarantee routine is used both in scheduling tasks that arrive at a node, and in making bids for remote tasks locally nonguaranteed. Simulation studies are conducted on the algorithm to observe how communication delay, task laxity, system load, and system's network topology affect the overall performance of the system. The performance of the algorithm is also compared with that of three other algorithms : noncooperative, random scheduling, and direct bidding. From the simulation results reported in Part VII, it is observed that the scheme is effective and practical in a wide range of application environments. It reaps the benefits of both bidding and focused addressing, and overcomes the shortcomings in using each by itself.

The thesis consists of eight parts, including this part. In the next part, a general information about distributed systems and real-time systems is given, and a hard real-time task model is introduced. Then, in Part III, a taxonomy of different approaches to the distributed scheduling problem is presented. Part IV concerns a literature survey on scheduling algorithms in hard real-time systems. The current literatures for multiprocessor systems and for distributed systems are reviewed separately. After this general study, in Part V, an overview of the scheme of scheduling hard real-time tasks with general resource requirements in distributed systems is introduced whose local scheduling component is discussed in detail in Part VI, and distributed scheduling component is discussed in detail in Part VII. Part VIII contains the conclusion of the thesis.

Appendices A through D, contain the listings of some of the simulation programs which are introduced in Parts VI and VII. One may refer to the diskette for a complete set of programs developed for this study.

Bibliography gives a list of references used in this study and cited in the text of the thesis. References not cited are listed separately.

II. SCHEDULING

This part contains the presentation of the distributed system model adopted throughout this study and the introduction of the scheduling problem in distributed computer systems followed by a general information about real-time systems and hard real-time scheduling. A hard real-time task model is also presented.

2.1. Distributed Systems and Scheduling

A *resource* is an entity which may be demanded by tasks. It can include CPU, I/O devices, files, data structures, etc. A resource which has processing power is called an *active resource*, or *processor*. A CPU or an I/O processor is an active resource. If a resource has no processing power, it is a *passive resource*. Files are examples of passive resources. Therefore, a passive resource must be used with an active resource.

A *multiprocessor system* is a configuration of a set of resources in which the control is centralized, and processors can communicate with each other without any significant delay. According to the definition of resources, in a multiprocessor system there is at least one active resource, that is, a processor, and zero or more passive resources. In a multiprocessor system, the processors are *identical* if they are exactly the same in terms of the processing power, that is they have the same instruction set and the same speed. The processors are *uniform* if they have only the same instruction set, but different speeds.

A *distributed system* is defined as any configuration of two or more nodes, each consisting of a multiprocessor system as defined above, with control of the system being distributed among the nodes. In a distributed system, communication between nodes occurs over some communication medium, and the time of communication between nodes is often assumed to be non-negligible.

Once the system is operational, improving response time and throughput of user processes is largely the responsibility of *scheduling algorithms* which are the operating system components that function continuously to manage the processing resources in the system. Proper design of such mechanisms has a great impact on overall system performance. This design problem becomes two-dimensional in the domain of distributed computing systems since not only the question of *when* to execute, but also *where* to execute a particular task must be addressed. Towards this goal, many approaches to the problem have been attempted, with variously reported results. A taxonomy of these approaches is given in Part III.

If a distributed computer system is to exploit the multiplicity of processors and resources in the network it must contain independent *local schedulers*. The local schedulers must interact and cooperate and the degree to which this occurs can vary widely.

Stankovic in [3] suggests that a good scheduling algorithm for a distributed computer system will necessarily use *heuristics* similar to those found in "expert systems". The task of these heuristics is to effectively utilize the resources of the entire distributed system given a complex and dynamically changing environment.

Some implications of using a heuristic function for scheduling should also be pointed out :

- (a) If an optimal scheduling algorithm can come up with a feasible schedule for a set of tasks, the heuristic scheduling algorithm may be able to do the same depending on the goodness of its heuristic function.
- (b) If even an optimal scheduling algorithm is unable to schedule a set of tasks, then the heuristic scheduling algorithm definitely cannot.
- (c) When there is no feasible schedule for a set of tasks, the heuristic scheduling algorithm will be able to discover that much sooner than an optimal scheduling algorithm.

2.2. Real-Time Systems

Recently, a major area of computer application has been real-time systems. There are two types of real-time computer systems :

(a) *A Hard Real-Time System* is one in which tasks have explicit time constraints, such as deadlines, so that a task is considered to be of value only if it finishes before its deadline.

(b) *A Soft Real-Time System* is one in which tasks have to be executed as quickly as possible, but there is no explicit time constraint associated with them.

Distributed systems are suitable for hard real-time applications. This is not only because often the applications themselves are physically distributed, but also because of the potential that distributed systems have for providing good reliability, good resource sharing and good extensibility, as reported by Stankovic in [3], by Stone and Bokhari in [4], and by Kleinrock in [5].

Nuclear power plants and process control applications are inherently distributed and have severe real-time constraints and reliability requirements. These constraints add considerable complication to a distributed computer system. Airline reservation and banking applications are also distributed, but have less severe real-time and reliability constraints and are easier to build. Examples of the more demanding real-time systems include ESS [6], REBUS [7], and SIFT [8]. ESS is a software controlled electronic switching system developed by the Bell System for placing telephone calls, REBUS is a fault tolerant distributed system for industrial real-time control, and SIFT is a fault tolerant flight control system.

In the future, such real-time systems are expected to become more and more complex, have long lifetimes, and exhibit very dynamic, adaptive and even intelligent behavior.

2.3. Scheduling in Hard Real-Time Systems

In many systems, and especially in embedded systems, danger to human life or simply damage to equipment makes the violation of a task's deadline unacceptable. It follows that the main requirement of a hard real-time system is that it should be supplied with a highly efficient *task scheduler* which carefully schedules the tasks so that all the tasks meet their timing requirements.

In a hard real-time scheduling algorithm, a set of tasks is said to be *guaranteed* if and only if the algorithm derives a schedule for the set of tasks which meets the given set of time, resource and precedence constraints. In a dynamic system, because all of the task characteristics are not known a priori, a task is said to be guaranteed by a scheduling algorithm if, when the task arrives, the scheduling algorithm is capable of finding a schedule for all the tasks previously guaranteed and the new arrived task.

A major performance metric for dynamic scheduling algorithms is the *guarantee ratio*, which is defined as the total number of tasks guaranteed versus the total number of tasks arrived.

A real-time scheduling algorithm is said to be *optimal* if given a set of tasks it can always generate a schedule meeting the time, resource and precedence constraints whenever there is any algorithm which can do so.

For *sub-optimal* algorithms, one performance metric is the *success ratio*, i.e., the total number of task sets guaranteed by this algorithm versus the total number of task sets guaranteed by an optimal algorithm.

A *task*, T , in a hard real-time system is characterized by the following parameters :

(A) *The Arrival Time, AT(T)* : At this time, the task and the associated task parameters (specifications) are known to the system.

(B) *The (Earliest) Start Time, ST(T)* : Only after this time, can task T be executed.

(C) *The Worst Case Computation Time, C(T)* : In any case, the running time of task T will not be more than this amount of time. Tasks in real-time system have to be designed so that the difference between their worst case and normal execution times is

not large. Otherwise, when resources are assigned to a task for its worst case execution time, poor resource utilization will result. In this regard, a dynamic scheduling scheme has advantages since based on the input parameters of a dynamically invoked task, a lower worst case computation time can be determined (compared to a statically determined worst case computation time).

(D) **The Deadline, $D(T)$** : By that time, task T must complete its execution.

(E) **The Laxity, $L(T)$** : This is the time difference between the earliest termination time of a task T and its deadline, where earliest termination time of a task is the sum of its earliest start time and its computation time.

(F) **The Resource Requirements of the task, $R(T)$** : This is a vector, specifying the resources needed in the execution of the task. It is assumed that a task needs all its resources throughout its execution, and the resource requirements of a task are always less than or equal to the total resources in the node of the system. A task will request at least one active resource and zero or more passive resources.

It is assumed that these parameters are always feasible, that is to say,

$$0 \leq AT(T) \leq ST(T) \leq D(T) - C(T),$$

always holds.

In a hard real-time system, there are two types of tasks : *nonperiodic tasks* and *periodic tasks*. A nonperiodic task arrives at any node dynamically and has to be executed before its deadline. The existence of a periodic task with period P implies that one instance of the task should be executed once every P units of time after system initialization. The i-th instance of a periodic task with period P has deadline being $i \cdot P + D'$ where D' is the relative deadline in a period. It is generally assumed that $D' \leq P$.

In addition to resource requirements and timing constraints, tasks in real-time systems are also characterized by their *priority* and *precedence constraints*. The priority of a task encodes its level of importance relative to other tasks. There may be precedence relation among a set of tasks in the system. A task T_1 is said to precede another task T_2 , if T_1 must complete its execution before T_2 starts. It is always assumed that the precedence relation is acyclic. A task is preemptable if it can be preempted in its execution. Precedence constraints enter the picture when tasks communicate or when a complex task is viewed in terms of a number of subtasks related by precedence constraints.

This study focuses on tasks that are independent and have equal priority, because consideration of precedence and priority constraints would add new variables to the already large number of variables used, and would affect the results of the simulation studies.



III. DISTRIBUTED SCHEDULING ALGORITHMS

The study of distributed computing has grown to include a large range of applications. However, at the core of all the efforts to exploit the potential power of distributed computation are issues related to the management and allocation of system resources relative to the computational load of the system.

The notion that a loosely coupled collection of processors could function as a more powerful general-purpose computing facility has existed for quite some time. A large body of work has focused on the problem of managing the resources of a system in such a way as to effectively exploit this power. The result of this effort has been the proposal of a variety of widely differing techniques and methodologies for distributed scheduling.

In this part, a taxonomy of approaches to the scheduling problem is presented in an attempt to bring together the ideas and the common terminology used in this area, and to provide a classification mechanism necessary in addressing this problem.

Among existing taxonomies, one can find examples of flat and hierarchical classification schemes. The taxonomy presented in this part is a hybrid of these two : hierarchical as long as possible in order to reduce the total number of classes, and flat when the descriptors of the system may be chosen in an arbitrary order.

3.1. Hierarchical Classification

The structure of the hierarchical portion of the taxonomy is shown in Figure 3.1. discussion of the hierarchical portion then follows.

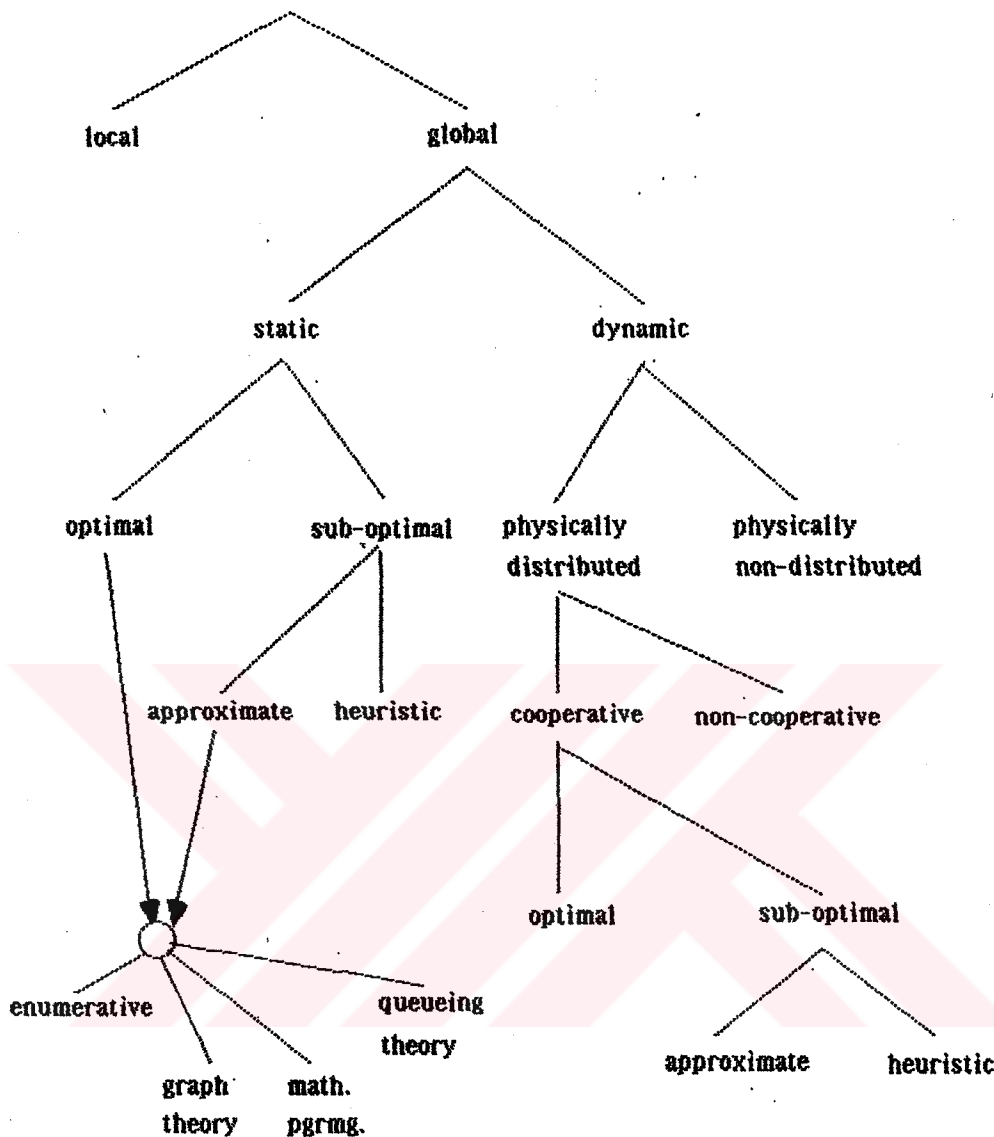


FIGURE 3.1 Structure of the hierarchical classification

(A) **Local Versus Global**: At the highest level, one may distinguish between *local* and *global* scheduling. Local scheduling is involved with the assignment of processes to the time-slices of a single processor. Global scheduling is the problem of deciding where to execute a process, and the job of local scheduling is left to the operating system of the processor to which the process is ultimately allocated. This does not imply that global scheduling must be done by a single central authority, but rather, the problems of local and global scheduling are viewed as separate issues, and (at least logically) separate mechanisms are at work solving each.

(B) **Static Scheduling**: The next level in the hierarchy (beneath global scheduling) is a choice between *static* and *dynamic* scheduling. This choice indicates the time at which the scheduling or assignment decisions are made. In the case of static scheduling, information regarding the total mix of processes in the system as well as all the independent subtasks involved in a job or task force is assumed to be available by the time the program object modules are linked into load modules. Hence, each executable image in a system has a static assignment to a particular processor, and each time that process image is submitted for execution, it is assigned to that processor.

(C) **Optimal Versus Suboptimal**: In the case that all information regarding the state of the system as well as the resource needs of a process are known, an *optimal* assignment can be made based on some criterion function. Examples of optimization measures are minimizing total process completion time, maximizing utilization of resources in the system, or maximizing system throughput. In the event that these problems are computationally infeasible, *suboptimal* solutions may be tried.

(D) **Approximate Versus Heuristic**: Within the realm of suboptimal solutions to the scheduling problem, two general categories may be encountered. The first is to use the same formal computational model for the algorithm, but instead of searching the entire solution space for an optimal solution, we are satisfied when we find a "good" one. Those solutions are categorized as *suboptimal-approximate*. The assumption that a good solution can be recognized may not be so significant, but in the cases where a metric is available for evaluating a solution, this technique can be used to decrease the time required to find an acceptable solution.

The second branch beneath the suboptimal category is labeled *heuristic*. This branch represents the category of static algorithms which make the most realistic assumptions about *a priori* knowledge concerning process and system loading characteristics. It also represents the solutions to the static scheduling problem which require the most reasonable amount of time and other system resources to perform their function. The most distinguishing feature of heuristic schedulers is that they make use of special parameters which affect the system in indirect ways.

(E) **Optimal and Suboptimal Approximate Techniques**: Regardless of whether a static solution is optimal or suboptimal-approximate, there are four basic categories of task allocation algorithms which can be used to arrive at an assignment of processes to processors:

(a) solution space enumeration and search;

(b) graph theoretic;

(c) mathematical programming;

(d) queueing theoretic.

(F) ***Dynamic Scheduling*** : In the dynamic scheduling problem, the more realistic assumption is made that very little *a priori* knowledge is available about the resource needs of a process. In the static case, a decision is made for a process image before it is ever executed, while in the dynamic case no decision is made until a process begins its life in the dynamic environment of the system.

(G) ***Distributed Versus Nondistributed*** : The next issue (beneath dynamic solutions) involves whether the responsibility for the task of global dynamic scheduling should physically reside in a single processor (*physically nondistributed*) or whether the work involved in making decisions should be *physically distributed* among the processors.

(H) ***Cooperative Versus Noncooperative*** : Within the realm of distributed dynamic global scheduling, we may also distinguish between those mechanisms which involve cooperation between the distributed components (*cooperative*) and those in which the individual processors make decisions independent of the actions of the other processors (*noncooperative*). The question here is one of the degree of *autonomy* which each processor has in determining how its own resources should be used.

In the noncooperative case individual processors act alone as autonomous entities and arrive at decisions regarding the use of their resources independent of the effect of their decision on the rest of the system.

In the cooperative case each processor has the responsibility to carry out its own portion of the scheduling task, but all processors are working toward a common system-wide goal. In other words, each processor's local operating system is concerned with making decisions in concert with the other processors in the system in order to achieve some global goal, instead of making decisions based on the way in which the decision will affect local performance only.

As in the static case, the taxonomy tree has reached a point where optimal, suboptimal-approximate, and suboptimal-heuristic solutions may be considered. The same discussion as was presented for the static case applies here as well.

3.2. Flat Classification Characteristics

In addition to the hierarchical portion of the taxonomy already discussed, there are a number of other distinguishing characteristics which scheduling systems may have. This section deals with characteristics which do not fit uniquely under any particular branch of the tree-structured taxonomy given thus far, but are still important in the way that they describe the behavior of a scheduler. In other words, the following characteristics could be branches beneath several of the leaves shown in Figure 3.1, and in the interest of clarity are not repeated under each leaf, but are presented here as a flat extension to the scheme given thus far.

It should be noted that these attributes represent a set of characteristics, and any particular scheduling subsystem may possess some subset of this set.

(A) ***Adaptive Versus Nonadaptive***: An adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to the previous and current behavior of the system in response to previous decisions made by the scheduling system. In contrast to an adaptive scheduler, a nonadaptive scheduler would be one which does not necessarily modify its basic control mechanism on the basis of the history of system activity.

(B) ***Load Balancing***: The basic idea is to attempt to balance (in some sense) the load on all processors in such a way as to allow progress by all processes on all nodes to proceed at approximately the same rate. This solution is most effective when the nodes of a system are homogeneous since this allows all nodes to know a great deal about the structure of the other nodes. Normally, information would be passed about the network periodically or on demand in order to allow all nodes to obtain a local estimate concerning the global state of the system. Then the nodes act together in order to remove work from heavily loaded nodes and place it at lightly loaded nodes.

This is a class of solutions which relies heavily on the assumption that the information at each node is quite accurate in order to prevent processes from endlessly being circulated about the system without making much progress.

(C) ***Bidding***: In this class of policy mechanisms, a basic protocol framework exists which describes the way in which processes are assigned to processors. The resulting

scheduler is one which is usually cooperative in the sense that enough information is exchanged (between nodes with tasks to execute and nodes which may be able to execute tasks) so that assignment of tasks to processors can be made which is beneficial to all nodes in the system as a whole.

To illustrate the basic mechanism of bidding, the framework and terminology introduced by Smith [9] will be used. Each node in the network is responsible for two roles with respect to the bidding process : manager and contractor. The manager represents the task in need of a location to execute, and the contractor represents a node which is able to do work for other nodes. A single node takes on both of these roles, and there are no nodes which are strictly managers or contractors alone. The manager announces the existence of a task in need of execution, then receives *bids* from the other nodes. A wide variety of possibilities exist concerning the type and amount of information exchanged in order to make decisions.

A very important feature of this class of schedulers is that all nodes generally have full autonomy in the sense that the manager ultimately has the power to decide where to send a task from among those nodes which respond with bids. In addition, the contractors are also autonomous since they are never forced to accept work if they do not choose to do so.

(D) ***Probabilistic*** : The basic idea for this scheme is motivated by the fact that in many assignment problems the number of permutations of the available work and the number of mappings to processors are so large that examining analytically the entire solution space would require a prohibitive amount of time. Instead, the idea of randomly (according to some known distribution) choosing some process as the next to assign is used. Repeatedly using this method, a number of different schedules may be generated, and then this set is analyzed to choose the best from among those randomly generated. The fact that an important attribute is used to bias the random choosing process would lead one to expect that the schedule would be better than one chosen entirely at random. The argument that this method actually produces a good selection is based on the expectation that enough variation is introduced by the random choosing to allow a good solution to get into the randomly chosen set.

(E) ***One-Time Assignment Versus Dynamic Reassignment*** : If the entities to be scheduled are jobs in the traditional batch processing sense of the term, then the single point in time in which a decision is made as to where and when the job is to execute is considered. While this technique technically corresponds to a dynamic approach, it is static in the sense that once a decision is made to place and execute a job,

no further decisions are made concerning the job. This class is characterized as one-time assignments. In contrast, solutions in the dynamic reassignment class try to improve on earlier decisions by using information on smaller computation units.

3.3. Application of Taxonomy to Some Examples

As an illustration of the taxonomy introduced in the previous sections, some example hard real-time scheduling algorithms are taken from the published literature, and their classification characteristics are determined according to the taxonomy. Table 3.1 contains the results.

These example algorithms are discussed in detail in Section 4.2.1 and in Section 4.2.2 of Part IV concerning the literature survey on scheduling algorithms for hard real-time systems.

Since this study is focused on hard real-time distributed scheduling, example algorithms are chosen from this area of research. One may refer to the work of Casavant and Kuhl, presented in [10], for a more general annotated bibliography of scheduling algorithms for general-purpose distributed computer systems.

Developed by	Reported In	Classification Characteristics
Lo, V. M.	[11]	Global, Static, Suboptimal, Approximate, Graph theoretic.
Efe, K.	[12]	Global, Static, Suboptimal, Heuristic, Load-balancing.
Ma, P. Y. R., Lee, E. Y. S., and Tsuchiya, J.	[13]	Global, Static, Optimal, Mathematical Programming.
Ramamritham, K., and Stankovic, J. A.	[14]	Global, Dynamic, Distributed, Cooperative, Suboptimal, Heuristic, Bidding, One-time assignments.

TABLE 3.1 Application of the taxonomy to some example algorithms

IV. LITERATURE SURVEY ON SCHEDULING ALGORITHMS FOR HARD REAL-TIME SYSTEMS

In this part, the algorithms proposed for scheduling in hard real-time systems are reviewed. Most research on scheduling tasks with hard real-time constraints is restricted to uniprocessor and multiprocessor systems. As reported by Graham, Lawler, Lenstra, and Kan in [15], optimal scheduling in a multiprocessing environment is an NP-hard problem, and hence computationally intractable. The loosely coupled nature of distributed systems makes the problem even harder. Section 4.1, contains an overview of the current literature on scheduling algorithms for multiprocessor systems, and Section 4.2, surveys work on scheduling algorithms for distributed systems.

4.1. Literature Survey for Multiprocessor Systems

4.1.1. Static Scheduling Algorithms for Multiprocessor Systems

Xu and Parnas in [16], present an algorithm that finds an optimal schedule on a single processor for a given set of processors such that each process starts executing after its release time and completes its computation before its deadline, and a given set of precedence and exclusion relations are satisfied. Exclusion relations may exist between process segments when some process segments cannot be interrupted by other process segments to prevent errors caused by simultaneous access to shared resources. This algorithm can be applied to the problem of pre-run-time scheduling of such processes.

on a single processor, in hard real-time systems. Future work is required to generalize the algorithm for n processors case.

Garey and Johnson in [17], describe an algorithm to determine if a two-processor schedule exists so that all tasks are completed in time, given a set of tasks, their deadlines, and the precedence constraints of all tasks.

Liu and Layland in [18], derive necessary and sufficient conditions for scheduling periodic tasks, with preemption permitted. The first algorithm is the Rate Monotonic Priority algorithm which assigns the highest priority to the task with the fastest rate, that is, the smallest period. The second algorithm, called Deadline Driven algorithm, dynamically assigns priorities to the instances of the periodic tasks based on their deadlines. The task with the smallest deadline gets the highest priority. Their results, which hold for uniprocessor systems were extended to include arbitrary task sets and precedence constraints.

Houssine Chetto and Maryline Chetto in [19], investigate the problem of estimating localization and duration of idle times when tasks are scheduled according to the Earliest Deadline scheduling algorithm as in [11]. Their aim is to bring to light new ideas about preemptive scheduling applied to a set of real-time, independent, periodic tasks that run on a monoprocessor machine.

Teixeira in [20], develops a model that considers priority scheduling for a more general case, where the deadline of a periodic task is not necessarily equal to the length of its period.

Johnson and Madison in [21], examine single and multiple processor systems executing real-time tasks. They develop a measure of free time to determine whether new tasks can be admitted and still meet every task's response specification.

These above schemes are quite inflexible, in that they do not adapt to the changing state of the system, and do not take into account general resource requirements of the tasks.

Blazewicz, Drabowski, and Weglarz in [22], present an algorithm for determining the shortest preemptive schedule in a system with a single resource type but any number of instances of this type. The authors formulate the determination of the schedule in the form of a linear programming problem and therefore, the problem can be solved in time which is a polynomial in the number of variables. This algorithm takes an exponential time in the number of resource instances which makes it

computationally too intensive to be used for on-line scheduling. Moreover, the case of multiple resource types is not handled.

However, in the work of Leinbaugh in [23], resource requirements are dealt with. He developed a heuristic algorithm which, when given the general resource requirements of each task, determines an upper bound on the response time of each task. While this approach is useful at system design time to statically determine the upper bounds on response times, it cannot be used for on-line scheduling, because there is no attempt at *dynamically* guaranteeing a new task so that it will meet its deadline.

Zhao, Ramamritham, and Stankovic in [1], describe a heuristic algorithm which takes into account both of tasks' active and passive resource requirements, and can be used in multiprocessor systems. The heuristic function, used to guide the search of a feasible schedule if there is one, is composed of three weighted factors which explicitly consider information about real-time constraints of tasks and their utilization of resources. They also show that modifying the approach to use limited backtracking improves the degree of success.

According to Lenat [24], heuristics are informal, judgmental rules of thumb which come in two types :

- (a) *those that actively guide the system toward plausible paths to follow ;*
- (b) *those that guide the system away from the implausible paths.*

In the work of Zhao, et al. [1], both types of heuristics are used. The heuristic function used by the algorithm actively directs the scheduling process to a plausible path, and also, the search space is constrained by looking only at strongly feasible paths, preventing from looking at implausible paths. As a result, even in the worst case, this algorithm is not exponential.

Zhao, Ramamritham, and Stankovic in [25], further consider the problem of scheduling a set of preemptable tasks in a real-time system in which a passive resource can be used either in shared mode or exclusive mode. They present an algorithm which uses a heuristic function which is a combination of Minimum Deadline First heuristic and Maximum Resource Utilization First heuristic with a third factor to prevent over preemption. They show that this algorithm, in conjunction with limited backtracks, works satisfactorily.

4.1.2. Dynamic Scheduling Algorithms for Multiprocessor Systems

It should be noted that in a dynamic system there is no a priori knowledge about any characteristics of a task until it arrives. Whenever a task arrives, a new schedule needs to be determined for the tasks including those which have been in the system, but have not finished, and the newly arrived one.

Since static scheduling problems for multiprocessor systems are similar to scheduling problems in operations research, they have been attracted by the researches since the 1950's. Various algorithms have been proposed. Some of them have a small time complexity. If a system can tolerate the time complexity of a static scheduling algorithm, the algorithm may be used to determine a new schedule dynamically when a task arrives. But, there are also scheduling algorithms which are developed specially for dynamic multiprocessor systems. The followings are some examples of such scheduling algorithms :

Dertouzos in [26], shows that the Earliest Deadline algorithm is optimal, for a single processor system with independent preemptable tasks. The proof depends on the fact that for a single processor system, it is always possible to transform a feasible schedule to one which follows the Earliest Deadline algorithm. This is so because if at any time the processor executes some task other than the one which has the closest deadline, then it is possible to interchange the order of execution of these two tasks, that is, execute the task with the closest deadline first and execute the sacrificed task at a later time when the task with the closest deadline would have been executed. Since the sacrificed task has a more distant deadline, making up for its processor time before the closest deadline certainly does not violate its own deadline.

Further, Dertouzos and Mok in [27], prove that the Least Laxity algorithm is also optimal for such a system to dynamically schedule hard real-time tasks. They also point out that the above optimality proof of the Earliest Deadline algorithm does not hold in the multiprocessor case. They show that for the case when the number of processors is larger than one, no scheduling algorithm can be optimal without *a priori* knowledge of deadlines, computation times and start times of the tasks. This implies that heuristic approaches have to be taken for scheduling tasks in such systems.

Jensen, Locke, and Tokuda in [28], report that Least Laxity, and Earliest Deadline scheduling policies perform much better than others in a multiprocessor real-time system.

It should also be pointed out that the above dynamic multiprocessor scheduling algorithms do not take into account the passive resource requirements of tasks.

4.2. Literature Survey for Distributed Systems

The architecture of the network and the nature of the application programs being presented to a distributed system are often such that the communication between nodes is a significant factor in the performance of the system. Because of this, the run time control has to be distributed. Hence, each node in the system is autonomous and often has its own local scheduler to handle the tasks assigned to it. The scheduling algorithms for multiprocessor systems can be used for the scheduling tasks on a node. However, how to allocate tasks to nodes statically in a static system, and how to transfer tasks from one node to another at run time in a dynamic system are the new problems.

4.2.1. Static Scheduling Algorithms for Distributed Systems

The static scheduling algorithms for distributed systems are already known to be difficult even without time constraints on tasks.

For example, as Bokhari reports in [29], if the objective is to minimize the cost of processing and communication, the problem of assigning tasks in a distributed system with heterogeneous processors is NP-hard for a system of more than three processors. For three processors the system is open.

For two processors, an optimal algorithm is reported by Stone in [30]. This algorithm considers two kinds of costs in an assignment. One is the computational cost, the other is the cost of interprocessor communication. He shows that the problem can be solved efficiently by making use of the algorithm for finding maximum flows in commodity networks.

Lo in [11], extends Stone's algorithm into a heuristic one for arbitrary number of processors. Lo also recognizes that the use of total execution and communication costs as the criteria for optimality has no explicit advantage to concurrency. Therefore the total completion time of tasks may not be optimal as it could be. Lo introduces a new cost, *the interference cost*, to measure the cost if two tasks are assigned on the same node. Interference costs reflect the degree of incompatibility between two tasks. For example, a pair of tasks that are both highly CPU bound would have greater interference costs than a pair in which one task is CPU bound and the other is I/O bound. Similarly, if two tasks were involved in pipelining, it would be undesirable that they are assigned to the same processor. This incompatibility would be expressed in a high interference cost for that pair of tasks. With this metric, Lo's algorithm is able to make assignments with greater concurrency and less completion time than the previous ones. Further, Lo investigates the problem with the goal of minimizing the completion time of a task set. An optimal algorithm is reported for the case where all costs are constant.

Chu and Lan in [31], propose a heuristic algorithm for task assignment which consists of two phases. Phase 1, reduces modules to a number of groups each of which will be assigned as a single unit to a processor. This grouping is based on several factors, such as, precedence relationship, communication costs and accumulative execution times. In phase 2, an exhaustive search is performed for the assignment of these groups to processors, such that, the load on the most heavily loaded processor (bottleneck) is minimized. The algorithm, instead of trying to minimize the sum of processor loads, searches the assignment that yields the minimum bottleneck. They show that assignments generated by such an approach yield good task response time which is the most important performance measure for real-time systems.

Efe in [12], proposes a heuristic algorithm for static assignment of tasks in a distributed system. His algorithm works as follows :

- (a) cluster tasks according to communication costs;
- (b) assign each cluster to a processor taking the current processor load into consideration;
- (c) if the results of the assignment in the above step satisfies the load balance constraint, stop; otherwise,
- (d) identify the overloaded and underloaded processors and move some tasks from the overloaded processor to the underloaded one;

(e) repeat from c.

Although the original goal of Efe's algorithm is to balance the loads of processors, the load balance constraint can be replaced with the deadline of the task set. Consequently, the modified algorithm can be used for the static assignment of tasks with task-set deadline.

It should be noted that, the above approaches cannot take into account deadlines of individual tasks, but the algorithms that will be discussed next, do.

Leinbaugh and Yamini in [32], extend the approach in [23] into distributed cases. In their model, a task is divided into multiple segments and the segments of a task can be executed concurrently on different nodes. In this study, the worst response time of each individual task is estimated by taking into account not only the blocking times caused by other tasks, but also the communication delays. Their algorithm is useful in a hard real-time environment to determine if response times will always be met.

Ma, Lee, and Tsuchiya in [13], and Ma in [33], propose an algorithm to statically assign tasks for a distributed system taking timing-critical applications into account. The model introduced, represents an example of an optimum mathematical programming formulation employing a branch-and-bound technique to search the solution space. The goals of the solution are to minimize interprocessor communications, balance the utilization of all processors, and satisfy all other engineering application requirements. The model given defines a cost function which includes interprocessor communication costs and processor execution costs. The assignment is then represented by a set of zero-one variables, and the total execution cost is then represented by a summation of all costs incurred in the assignment. In addition to the above, the problem is subject to constraints which allow the solution to satisfy the load balancing and engineering application requirements. The algorithm then used to search the solution space (consisting of all potential assignments) is derived from the basic branch-and-bound technique.

Both Efe and Ma, use *heuristic approaches* for related scheduling problems. But, they use the second type of heuristics mentioned in Section 4.1.1. This approach of only using the second type of heuristics is limited because, in the worst case, the exponential search problem cannot be avoided.

4.2.2. Dynamic Scheduling Algorithms for Distributed Systems

The dynamic scheduling algorithms for distributed systems should *maximize the guarantee ratio*. To achieve this goal, two factors must be recognized :

(A) Suppose that tasks demand each resource with equal probability and have the computation time equal to each other. Then, the guarantee ratio will be proportional to the resource utilizations. Hence, to maximize the guarantee ratio, one should *maximize the resource utilizations*. Since, in practice, tasks will not always satisfy the above conditions, this is only a rule of thumb. As reported by Livny and Melman in [34], in a dynamic distributed system, without any mechanism for cooperation among nodes, it is very likely that one node will be idle while tasks are queued at some other nodes. Thus, to maximize resource utilization, it is necessary at run time to transfer tasks to other less loaded nodes when they cannot be guaranteed locally.

(B) Because of the real-time constraints on tasks, the scheduling algorithm itself should be very efficient. That is, to maximize the guarantee ratio, one should also *minimize the scheduling delay*. This implies that the decisions, such as where to send a task that cannot be guaranteed locally, must be made efficiently. It is not practical, if not impossible, to perform a complete search to determine the best node to send a task, in a network where communication delay is not negligible.

These factors necessitate *a heuristic approach* for scheduling hard real-time tasks in a dynamic distributed system.

As reported by Smith in [9], and by Wang and Morris in [35], two approaches below have been recognized, in the current literature, for dynamically transferring tasks in general distributed systems :

- (a) *source initiated task transfer* where a node searches for other nodes to which a task may be transferred;
- (b) *server initiated task transfer* where a node searches for other nodes from which tasks may be transferred.

Ramamritham and Stankovic in [14], adopt the ideas of source/server initiated task transfer, and suggest particular versions of them for hard real-time systems. In this work, *bidding* is implemented as source initiated task transfer, and *focused addressing* is implemented as server initiated task transfer. Briefly, in bidding, a node

is selected if the node offers the best bid. The communication costs involved in bidding are high, but selection is made based on relatively accurate state information of nodes. On the other hand, in focused addressing, a node contains some state information about the other nodes, estimates the surplus of other nodes, and selects a node to send a task to based on these estimates. Focused addressing entails less communication costs and delay than bidding, though the use of incomplete, inaccurate and out-of-date state information, increases the risk of making wrong decisions. Because of these reasons, the working domain of these schemes are limited.

Stankovic, Ramamritham, and Cheng in [36], report an approach combining bidding and focused addressing. The aim is to reap the benefits of both and to overcome the shortcomings inherent in using each by itself. They show that the working domain of the combined scheme covers both domains of bidding and focused addressing.

Kurose and Chipalkatti in [37], study analytically the relative performance of several different decentralized approaches towards load sharing, in order to determine the level of complexity for load sharing algorithms in a distributed real-time environment. In their model, it is assumed that tasks arriving at a node have to complete their execution within a fixed amount of time, after their initial arrival to the system. That is to say, deadlines, are not drawn from additional deadline distributions. They develop an approximate analytic system-level model for the entire distributed system, and use it to quantitatively study the real-time performance of two simple approaches towards real-time load sharing. In the first approach, called *quasi-dynamic load sharing*, a task which cannot meet its deadline locally is sent to a probabilistically chosen remote node. The second approach is the *probing* approach which is a simplified form of bidding. In this approach, when a task is to be transferred, a node probes some specified number of other nodes chosen at random to determine if one of them can currently guarantee it. Their performance results show that, the performance of these simple approaches is substantially better than the case of no load sharing and often close to that of a theoretically optimum algorithm.

But, all of these last three algorithms above, consider just CPU scheduling. General resource requirements of the tasks are not dealt with.

Recently, Ramamritham, Stanković and Zhao in [2], present another version of the algorithm reported in [36], in which *general task's active and passive resource requirements* are also taken into account.

V. OVERVIEW OF THE SCHEDULING SCHEME

In the design of real-time computer systems, the scheduling problem is considered to be an important one, and has been addressed by many researches as discussed in Part IV. However, most approaches are restricted to CPU scheduling only. Whereas the scheduling algorithm, which is chosen to be evaluated in this study, takes general tasks' passive and active resource requirements into account as well [2]. This part contains a brief overview of the algorithm, the details are discussed in subsequent parts.

In this scheduling scheme, the scheduling entity is a task. It is assumed that tasks may arrive dynamically at any node, and that they are independent, non-preemptable, and have equal priority. The worst case computation time, the deadline, the resource requirements of the tasks are assumed known when they arrive.

Each node in the distributed system has a *local scheduler*. Each local scheduler contains a guarantee routine, a bidder, a dispatcher, and a node surplus manager. Figure 5.1 shows how these various modules interact with each other.

The local scheduler at a node, invokes the *guarantee routine*, when a new task arrives at that node. The guarantee routine decides if the new task can be guaranteed at this node or not. The guarantee means that no matter what happens (except failures) this task will execute by its deadline, and that all previously guaranteed tasks will also still meet their deadlines. If the new task cannot be guaranteed locally, then it becomes a candidate for distributed scheduling.

The *bidder* interacts with the local schedulers on the other nodes in order to perform distributed scheduling. It is responsible for determining where a task that cannot be locally guaranteed should be sent. It does this through a combination of focused addressing and bidding.

In *focused addressing*, a task is sent directly to another node based on its partial knowledge about the surplus of the other nodes in the system.

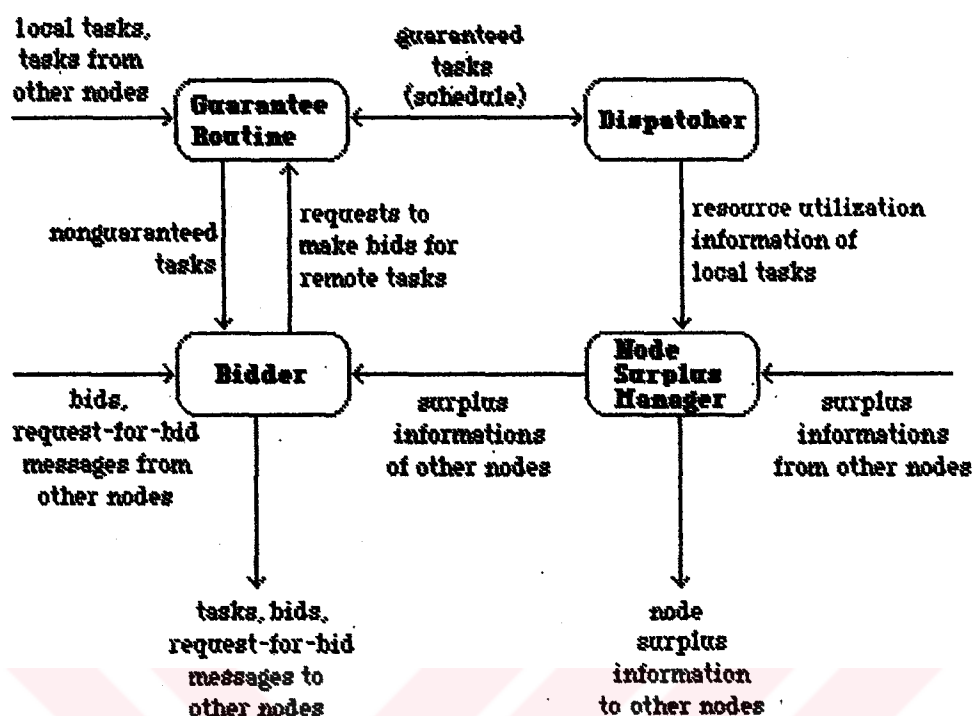


FIGURE 5.1 Structure of the local scheduler on a node

In *bidding*, the node sends out request-for-bid messages to other nodes. Nodes with sufficient surplus on resources needed for the task, respond with a bid reflecting this surplus. Then, the task is sent to the node which offers the best bid. In addition to sending its tasks to other nodes, the bidder makes bids in response to request-for-bid messages from the other nodes.

The *dispatcher* is the component that actually schedules the guaranteed tasks.

It should be pointed out that when a node bids for a task, it does not reserve CPU time for that task. Reserving CPU time ties up too many resources for a long time. Consequently, when a task finally arrives at a bidder node, the node will attempt to guarantee it. In case that this guarantee fails, the task will be considered as nonguaranteeable.

There is a separation of dispatching and guaranteeing, allowing the dispatcher and the guarantee routine to run in parallel. The dispatcher is always working with a set of tasks which have been validated to meet their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks.

One of the assumptions underlying the scheduling algorithm is that nodes can estimate the resource usage or resource surplus of other nodes. This requires that nodes keep each other informed about their surplus. This can be done by the *node surplus manager* in the following way :

The node surplus manager on each node periodically calculates the node surplus. The node surplus provides information about the available time on each resource in a previous window, by taking into account resource utilization of *local* tasks, that is to say, the tasks that directly arrived at a node from the external environment and not from the subnet. This information is used to predict the resource availability for the tasks from the other nodes in the near future. The computed node surplus is sent to a selected subset of nodes in the system. The selection is to be based on the proximity of the nodes, on who sent tasks to this node recently, and on whether the tasks were guaranteed.

The steps involved in scheduling a newly arrived task are as follows :

(A) When a local task, T , arrives at a node N_j , the local scheduler is invoked to try to guarantee the newly arrived task on the node. If the task can be guaranteed, it will be put into the *schedule* which contains all the guaranteed tasks on the node. The details of the local scheduling algorithm is discussed in Part VI.

(B) When the local scheduler of node N_j is unable to guarantee the newly arrived task, T , it attempts to find another node through focused addressing. This focused node should have sufficient surplus to guarantee the task. If a focused node is found, the task is immediately sent to the node. In addition to sending the task to the focused node, node N_j sends request-for-bid messages to a subset of the other nodes. The request-for-bid message also contains the identity of the focused node, if there is one, indicating that the bids should be sent to the focused node.

(C) When a node receives the request-for-bid message, it calculates a bid indicating the possibility that the task can be guaranteed on the node, and sends the bid to the focused node if there is one, otherwise, to the original node which issued request-for-bid.

(D) When a task reaches a focused node, it first invokes the local scheduler to try to guarantee the task. If it succeeds, all the bids for the task will be ignored. If it fails, the bids for the task will be compared and the task will be sent to the node responding with the "best bid" on condition that the bid is above a certain limit.

(E) In case there is no focused node, the original node will receive the bids for the task and will send the task to the node which offers the best bid again on condition that the bid is above a certain limit.

(F) If the focused node cannot guarantee the task and if there is no good bid available for the task, it is assumed that no node in the network is able to guarantee the task. If a task has sufficient laxity then focused addressing and bidding may be repeated. But, this will increase the scheduling and communication overheads.

The distributed scheduling scheme is discussed in detail in Part VII.



VI. LOCAL SCHEDULER

In this part, the strategy for scheduling tasks on a local node is introduced. The heuristic algorithm developed by Zhao, et al. [1], is chosen as the algorithm underlying the guarantee routine on each node, and is implemented with some modifications. Since properly choosing the heuristic function used by the guarantee routine in selecting the next task to be scheduled, is important for the performance of the algorithm, a set of heuristics is studied in Section 6.4. From the simulation studies performed in that section, it is concluded that simple heuristics do not perform satisfactorily because of the complexity of the problem. However, an algorithm that uses a combination of these simple heuristics works very well compared to an optimal algorithm that takes exponential time complexity. The heuristic function which has the best performance will be used as the heuristic for the guarantee routine in the distributed scheduling scheme described in detail in Part VII. In this scheme the guarantee routine is used both in scheduling tasks that arrive at a node, and in making a bid for a remote task which cannot be guaranteed locally.

6.1. Strategy Behind the Local Scheduler

At any given time, node N_i ($i = 1 \dots n$) has guaranteed a set of tasks S_i and has a full feasible schedule for this set of tasks. A *feasible schedule* is a list of tasks that have been guaranteed. With respect to a set of tasks, a schedule is *full*, if it contains all the tasks in the set, otherwise it is *partial*. A schedule $(T_1, T_2, \dots, T_S, T_{S+1})$ is an *immediate extension* of the schedule (T_1, T_2, \dots, T_S) .

Suppose task T comes to the local scheduler at node N_i , then the following steps are taken in order to guarantee the newly arrived task T :

(A) The guarantee routine in node N_i is called to decide whether the new task can be guaranteed or not. The new task T can be guaranteed on this node if and only if, a new

full feasible schedule exists for tasks in $S_i \cup \{T\}$. This ensures that the tasks of S_i in the original feasible schedule remain guaranteed. Also, it ensures that the new task T will meet its deadline.

(B) If T is guaranteed by node N_i (as stated above), the new full feasible schedule containing tasks in $S_i \cup \{T\}$ replaces the original one. This schedule determines the start times of the tasks in node N_i , and will not be modified until another new task is guaranteed by node N_i .

(C) If the new task T cannot be guaranteed by node N_i , that is, there is no full feasible schedule for tasks in $S_i \cup \{T\}$, the approach based on bidding and focused addressing is used to determine if another node is in a position to guarantee task T . When such a node is found, T is sent to that node. In any case, the current feasible schedule of node N_i remains unchanged.

In the remainder of this part, the first step above is explained. That is, a heuristic technique for determining whether a node's current feasible schedule can be changed in order to introduce a new task, is presented.

6.2. The Basic Algorithm Underlying the Guarantee Routine

This section describes the heuristic algorithm underlying guarantee routine. First scheduling and searching are compared, then several data structures used are presented, a constraint on the search process is motivated, and finally the basic algorithm is presented.

6.2.1. Scheduling versus Searching

The guarantee routine determines a full feasible schedule for a given set of tasks in the following way: it begins with an empty schedule and tries to extend it with one task at a time until a full feasible schedule is derived. This is, in fact, a search problem. The

structure of the search space is a *search tree*. The root of the search tree is the empty schedule. An *intermediate vertex* of the search tree is a partial schedule. A *descendant* of a vertex is an immediate extension of the schedule corresponding to the vertex. A *leaf*, a terminal vertex, is a full schedule. It should be noted that all leaves will correspond to feasible schedules. The goal of the algorithm is to search for a leaf that corresponds to a full feasible schedule. Figure 6.1 shows a search tree for a set of 4 tasks.

An optimal algorithm, in the worst case, may make an exhaustive search, which is computationally intractable. In order to make the algorithm computationally tractable even in the worst case, a heuristic approach for this search is preferred. That is, a heuristic function, H , is developed which can synthesize the various factors affecting real-time scheduling decisions to actively direct the scheduling process to a plausible path.

On each level of the search, function H is applied to each of the tasks that remain to be scheduled. The task with the minimum value of the function H is selected to extend the current partial schedule. As a result of the above directed search, even in the worst case, this scheduling algorithm is not exponential.

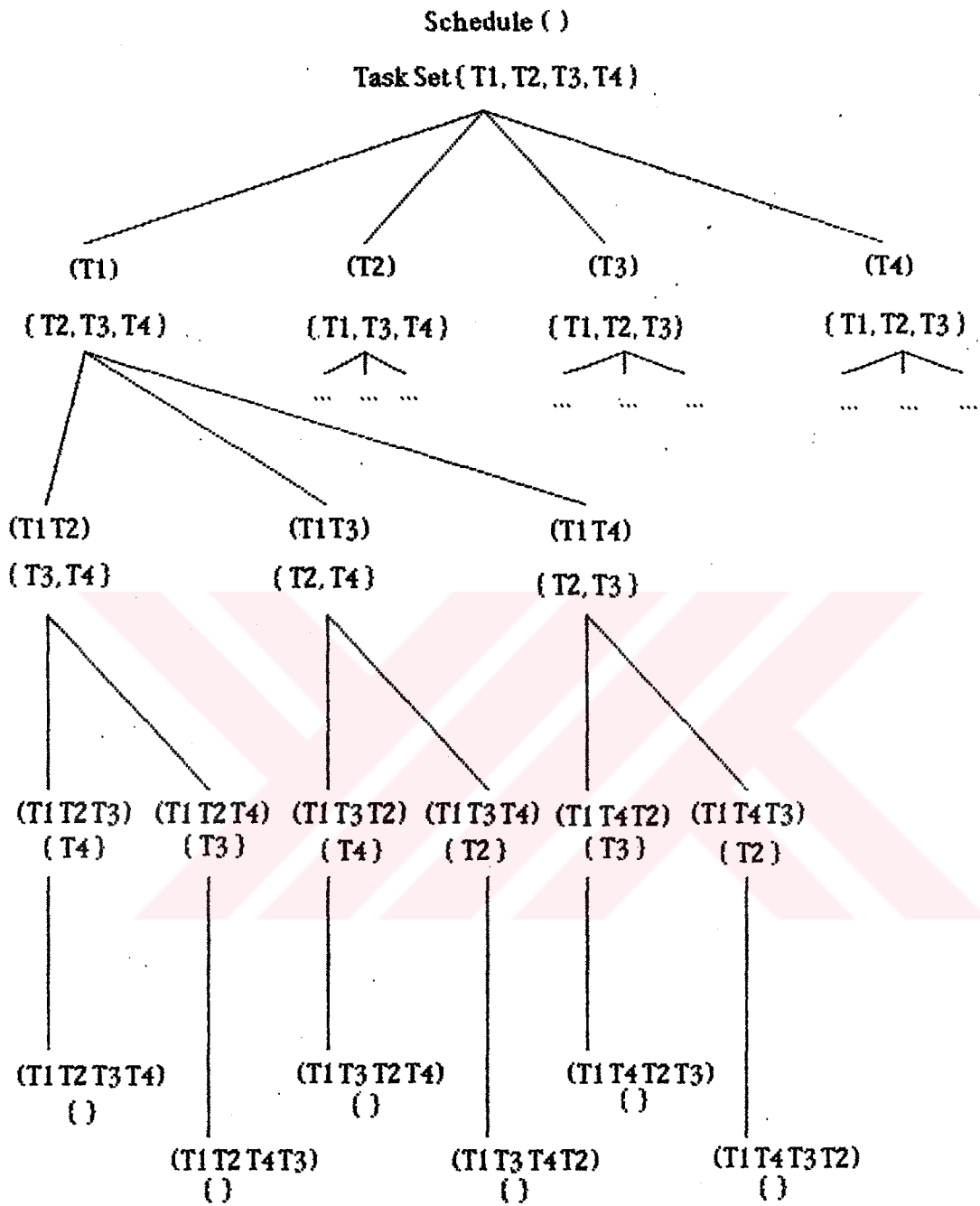
6.2.2. Data Structures

The algorithm maintains a vector EAT , to indicate the *Earliest Available Times* of resources on a node :

$$EAT = (EAT_1, EAT_2, \dots, EAT_r)$$

where EAT_i is the earliest time when resource R_i will become available. Initial values of EAT_i for all i will be the current time if the running task is preemptable. Otherwise, EAT_i will be the time when the running task finishes using it. Each time the partial schedule is extended, EAT will be updated taking into account the newly added tasks' resource requirements and completion time.

At each level of the search tree, the guarantee routine computes $ST(T)$ and $New_EAT(T)$ for each task T that remains to be scheduled. $ST(T)$ indicates the start time



A Partial Schedule : (...)

A Subset of Tasks Remaining to be scheduled : (...)

FIGURE 6.1 A search tree for a set of four tasks

of task T if it is scheduled next. Since a task T can run only when all resources it needs are available, $ST(T)$ is defined as :

$$ST(T) = \text{MAX}(EAT_i \text{ where } T \text{ needs } R_i).$$

It should be noted that for a given feasible schedule to remain feasible when extended by T,

$$ST(T) + C(T) \leq D(T)$$

must hold, where $C(T)$ is the computation time and $D(T)$ is the deadline of the task T.

$\text{New_EAT}(T)$ is a vector with the same size as EAT and contains the earliest available times of resources if task T is scheduled next. In other words, $\text{New_EAT}(T)$ will replace the current EAT if task T is scheduled. It is calculated as :

$$\text{New_EAT}(T) = ST(T) + C(T).$$

$\text{New_EAT}(T)$ should be further updated because in the system model, active resources are distinguished from passive ones. Since a passive resource must be used with active ones, no task can use a passive resource until :

$$\text{time} = \text{MIN}(\text{New_EAT}(T)_i \text{ where resource } i \text{ is an active resource})$$

where $i = 1, \dots, r$. That is, all $\text{New_EAT}(T)_i$ s for passive resources should not be less than the minimum $\text{New_EAT}_i(T)$ of active resources. Hence, $\text{New_EAT}(T)_i$ s should be further updated as :

$$\text{New_EAT}(T)_i = \text{MAX}(\text{New_EAT}(T)_i, \text{time}) \quad \text{where } i = 1, \dots, r.$$

At each level of the search, the guarantee routine also calculates a vector called $DRDR$, the *Dynamic Resource Demand Ratio*, which indicates the degree to which tasks that remain to be scheduled will demand resources :

$$DRDR = (DRDR_1, DRDR_2, \dots, DRDR_r)$$

where $DRDR_i$ is defined as :

$$DRDR_i = \frac{\sum (C(T), T \text{ remains to be scheduled and uses } R_i)}{\text{MAX}(D(T), T \text{ remains to be scheduled and uses } R_i) - EAT_i}$$

where $i = 1, \dots, r$.

For all the remaining tasks to be schedulable, every $DRDR_i$ of a DRDR associated with a partial feasible schedule should be less than or equal to one. If that is not the case, this means that there is no need to continue the search, it is not possible to find a feasible schedule for the remaining tasks with such resource requirements.

EAT, New_EATs and DRDR are updated each time the partial schedule is extended.

6.2.3. A Constraint on the Search

Using the data structures, EAT and DRDR, described above, a constraint can be imposed on the search for a full feasible schedule.

A feasible partial schedule is said to be *strongly feasible* if :

- (a) DRDR associated with the schedule has $DRDR_i \leq 1$ for $i = 1, \dots, r$, and
- (b) all of its immediate extensions are feasible, that is to say, for each task T that remains to be scheduled, there will not be any deadline violation when the current feasible schedule is extended by T.

By definition, a full feasible schedule is strongly feasible. If a schedule is not strongly feasible because one of the conditions fails, then the failed condition will also fail for all descendants, i.e., the extensions, of the non-strongly feasible schedule. Hence, none of the descendants of a non-strongly feasible schedule can be strongly feasible. On the other hand, the ancestor of a full feasible schedule must be strongly feasible, otherwise the full schedule itself will not be feasible. Therefore, only strongly feasible schedules can lead to a full feasible schedule. Considering this fact, the following constraint on the search for a full feasible schedule can be stated :

For a partial schedule to be extendible to a full feasible schedule, the partial schedule should be strongly feasible.

From the viewpoint of the algorithm, this means that it is not necessary to search through a vertex corresponding to a non-strongly feasible schedule, because a non-strongly feasible schedule will not lead to a full feasible schedule. Given the above

constraint, the search should be confined only to those subtrees whose roots correspond to strongly feasible schedules.

However, in the worst case an exhaustive search may still be required, making the search computationally intractable. In order to make the algorithm computationally tractable, even in the worst case, only one of the vertices is chosen at each level in order to expand the search tree. The vertex chosen is the one which appears to be most capable of leading to a full feasible schedule. In the next section the basic algorithm which incorporates the heuristic necessary to make this choice, is discussed.

6.2.4. The Basic Algorithm

The pseudo code for the basic local scheduling algorithm is given in the Figure 6.2. Beginning with the empty schedule, the algorithm searches the next level by expanding the current vertex (a partial strongly feasible schedule) to only one of its immediate descendants. If the immediate descendant is also a strongly feasible schedule, the search continues until a full feasible schedule is met. At this point, the searching process succeeds and all the tasks are known to be guaranteed.

If at any level, a non-strongly feasible schedule is met, the algorithm announces that the searching (scheduling) process fails and that this set of the tasks cannot be guaranteed. This implies that the new task we are trying to dynamically guarantee is not guaranteed so there is no new schedule. The previous schedule is left unaffected.

A modification is made on the original algorithm. Instead of calculating `New_EATs` just before applying the function `H` as in the original algorithm, in this study, it is preferred to calculate them before the `if` statement which checks the strong feasibility condition. In this way, while calculating `New_EATs`, possible deadline violations of tasks are detected, and this information is used by the "`strongly_feasible`" function in order to decide whether all of the immediate extensions are feasible or not (second condition of strong feasibility).

```

PROCEDURE Scheduler(VAR guaranteed : boolean);
BEGIN
  guaranteed := true;
  schedule := empty;
  WHILE NOT empty(task_set) and (guaranteed) DO
    BEGIN
      calculate ST for each task in task_set;
      calculate New_EAT for each task in task_set;
      calculate DRDR;
      if not strongly_feasible
        THEN guaranteed := false
        ELSE BEGIN
          apply function H to each task in the task_set;
          let T be the task with the minimum value of function H;
          EAT := New_EAT(T);
          remove task T from task_set;
          append task T to schedule
        END
      END
    END;

```

FIGURE 6.2 Basic local scheduling algorithm for guarantee routine

It should be noted that, it is possible to extend the algorithm to continue the search even after a failure is found, and this extension is discussed in the next section.

6.3. Extension to the Basic Algorithm

The assumptions underlying the use of the heuristic function in the basic algorithm are :

- (a) at each level of the search, there is a certain *order* among the tasks to be selected;
- (b) the order can be *identified* by a linear function such as function H used in the basic algorithm.

Though the first assumption is definitely true, the second may not always hold, so the original algorithm cannot always guarantee a set of tasks for which there is at least one full feasible schedule. To improve the success ratio, the following means were considered :

- (a) add some non-linear components to function H;
- (b) change the weight of function H dynamically;
- (c) whenever a partial non-strongly feasible schedule is met while scheduling, try to backtrack.

Since the first alternative increases the computation cost on every computation of function H, and the second could make the algorithm too complex, the third one is adopted.

The basic algorithm is extended in the following way :

Each time a non-strongly feasible schedule is found,

- (a) a procedure called *Limited_Backtracker* is invoked to withdraw the task just selected and added in the schedule, and instead attempt to schedule the task with the second minimum value of function H;
- (b) if the first step does not succeed, that is, the schedule is still non-strongly feasible, recursively backtrack to the immediate ancestor and attempt to schedule the task with the second value of function H at the ancestor level. Whenever a strongly feasible schedule is found, the *Limited_Backtracker* returns "guaranteed" to the caller, the procedure *Scheduler*. Otherwise, it continues the recursive backtrack until either it has backtracked to the root of the search tree (the empty schedule), indicating that all the ancestors have been tried; or until a counter, which counts the number of backtracks in scheduling this task set, reaches a pre-set upper bound. In these cases, the *Limited_Backtracker* returns "nonguaranteed".

The pseudo code of the algorithm for the procedure *Limited_Backtracker* is shown in Figure 6.3. The first step in the *Limited_Backtracker* is called a *pseudo backtrack* because it happens at the current search level and function H is not recalculated. The second step is called *real backtrack*. Real backtracks do increase the computation cost because they requires the recalculations of the function H at all the levels immediately below the vertex in which the real backtrack succeeds.

```

PROCEDURE Limited_Backtracker ( var guaranteed : boolean);
( This procedure is called when the partial schedule is found to be non-strongly feasible)
BEGIN
if empty(schedule)
THEN guaranteed :- false
ELSE
BEGIN ( first, pseudo backtrack )
let T1 be the last task in the schedule;
remove T1 from schedule and append it to task_set;
let T2 be the task with the second H value pointed to by the second pointer of T1;
remove T2 from task_set and append it to schedule;
IF not strongly_feasible
THEN
BEGIN ( the real backtrack starts )
guaranteed :- false;
WHILE (NOT empty(schedule)) and (counter < max_counter) and (not guaranteed) DO
BEGIN
( withdraw from the end of the schedule all the tasks, one by one, until a task
having a non-nil "second pointer" is met or there is no task left in the schedule or
the partial schedule is guaranteed.)
REPEAT
let T1 be the last task in the schedule;
remove T1 from schedule and append it to task_set
UNTIL ( T1's "second pointer" <> nil ) or ( empty ( schedule ));
IF T1's "second pointer" <> nil
THEN BEGIN
let T2 be the task pointed by T1's "second pointer";
EAT := New_EAT stored as old_EAT with T2;
remove T2 from task_set and append it to schedule;
IF strongly_feasible THEN guaranteed :- true;
counter := counter + 1
END
END ( WHILE )
END
END
END;

```

FIGURE 6.3 The algorithm of the Limited_Backtracker.

It should be noted that, if in the `Limited_Backtracker` the number of real backtracks is not limited, then in the worst case, the search process might eventually expand two vertices from each ancestor, resulting in a computation time proportional to 2^k , where k is the number of tasks.

In order to avoid some re-calculations that may be caused by possible future backtracks, each scheduled task keeps a pointer to the task with the second minimum value of function H at that level. In the original algorithm, the EAT values before the task is scheduled is also recorded. Whereas in this study, it is preferred to record the `New_EAT` values of the task with the second minimum value of function H , instead of the EAT values. The motivation is to be able to use these `New_EAT` values, when there is a backtrack which attempts to schedule the task with the second minimum value of H , without having to re-calculate them at that level.

Another modification is the following: an if statement is added at the beginning of the procedure `Limited_Backtracker`, which checks whether the schedule is empty or not. Because, a schedule can be found non-strongly feasible (any one of the two strong feasibility conditions may not hold) before any task has been scheduled. In this case, since the schedule is empty, backtracking is not possible.

Therefore, the data structure used to implement a task node has the form shown in Figure 6.4.

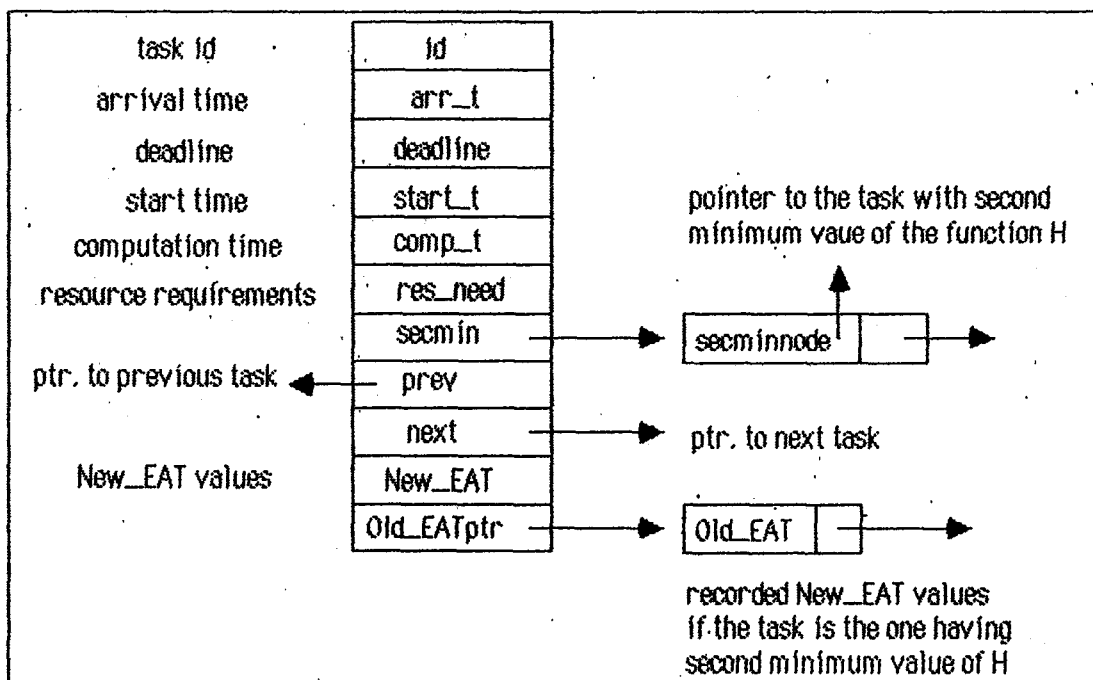
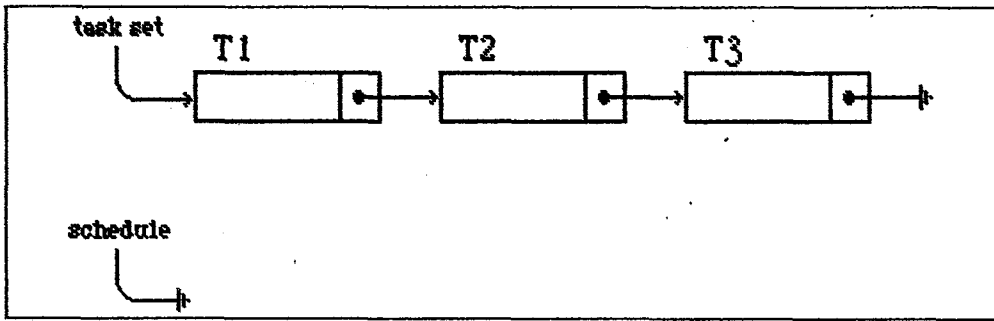


FIGURE 6.4 Data structure used to implement a task node

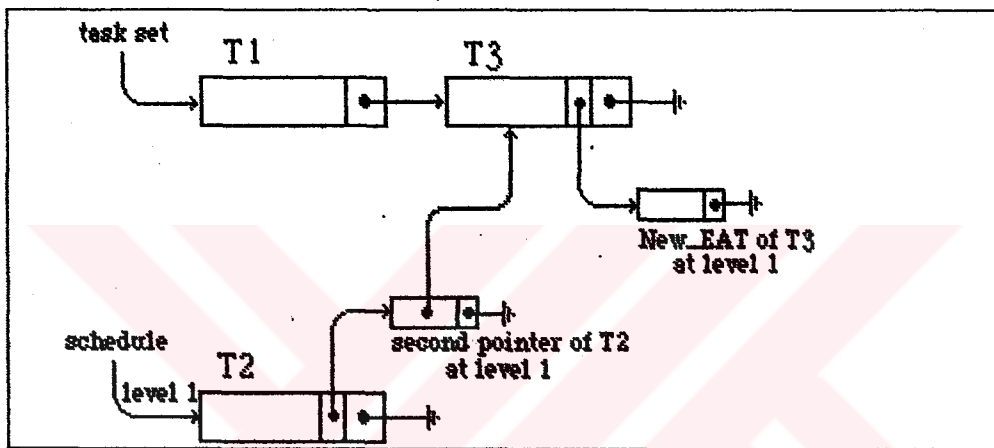
The values of *id*, *arr_t*, *deadline*, *comp_t*, and *res_need* are known when a task arrives. The values for *start_t* and *New_EAT* are calculated at each level of the search. The use of *secmin* and *Old_EATptr* will be illustrated, by a simplified example. Assume the following scenario :

- (a) let task set be { T1, T2, T3}, and let schedule be {} (Figure 6.5 (a));
- (b) let the schedule be strongly feasible, function H is applied to each task in the task set in order to select the task to be scheduled at level one : let T2 be the task with the minimum value of function H, and let T3 be the task with the second minimum value;
- (c) T2 is scheduled at level one (Figure 6.5 (b));
- (d) assume that the schedule is strongly feasible, then the next task to be scheduled at level two is selected : let T1 be the task with the minimum value of function H, and let T3 be the task with the second minimum value;
- (e) T1 is scheduled after T2 at level two (Figure 6.5 (c));
- (f) assume that the schedule is found to be non-strongly feasible;
- (g) **Pseudo Backtrack** : T1 is removed from schedule and appended to task set, since T1's "second pointer" is T3, T3 is scheduled, EAT is updated by using the recorded *New_EAT* values of T3 for level two without having to recalculate them (Figure 6.5 (d));
- (h) assume that the schedule is again non-strongly feasible;
- (i) **Real Backtrack** : T3 is removed from schedule and appended to task set, going back to level one (ancestor level) T2 is removed from schedule and appended to task set (Figure 6.5 (e));
- (j) since T2's "second pointer" is T3, T3 is scheduled, EAT is updated by using the recorded *New_EAT* values of T3 for level one (Figure 6.5 (f));
- (k) assume that the schedule is still found to be non-strongly feasible, T3 is removed from schedule and appended to task set, since further backtracks are not possible, the task set is said to be nonschedulable.

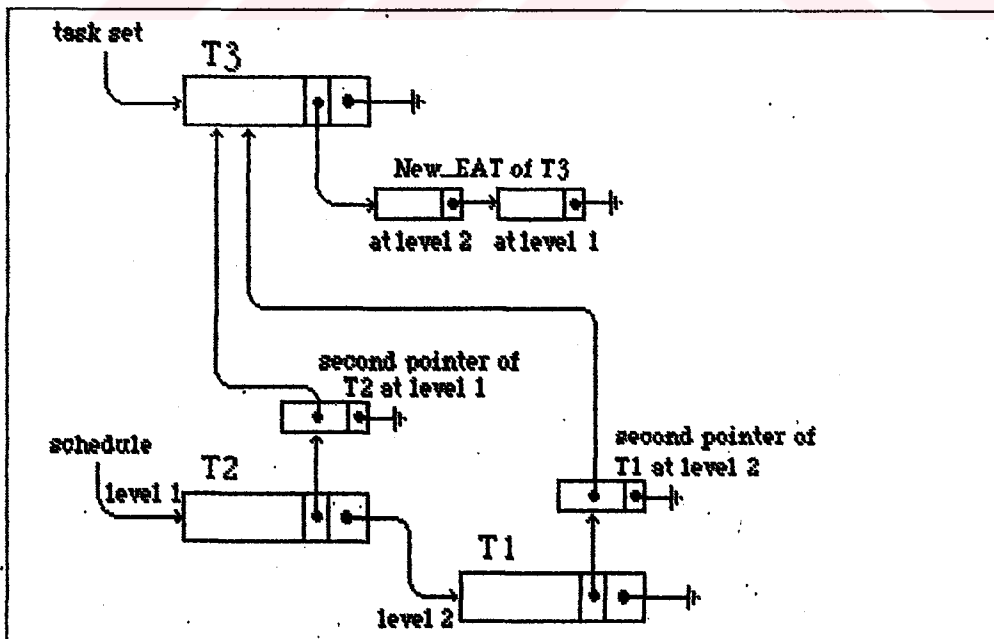
But, if this real backtrack had succeeded, the search would have continued by recalculating the function H in order to detect the task to be scheduled at level two.



(a)

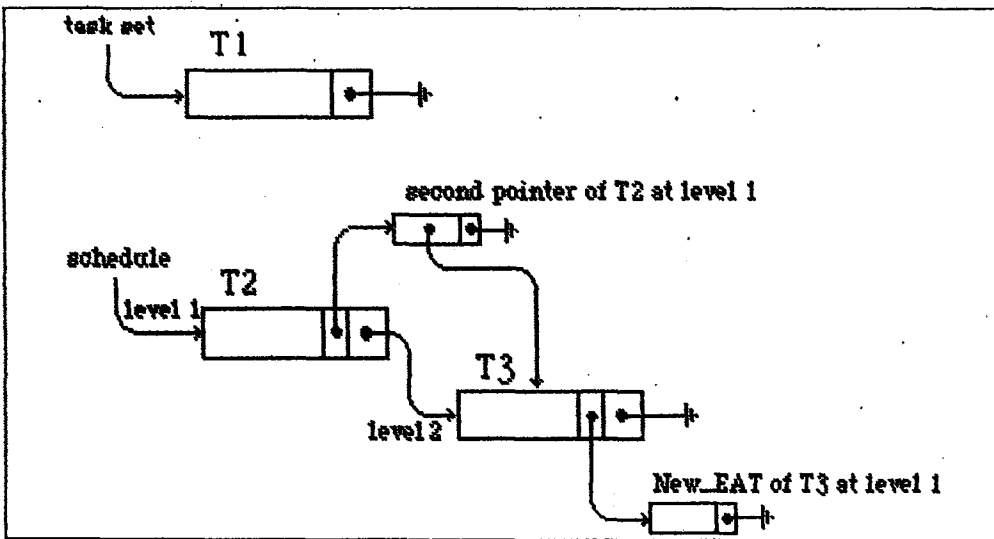


(b)

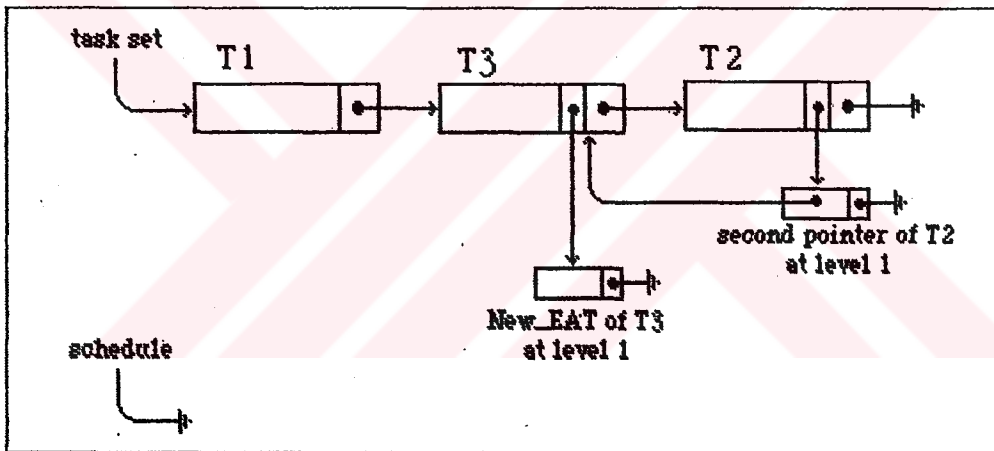


(c)

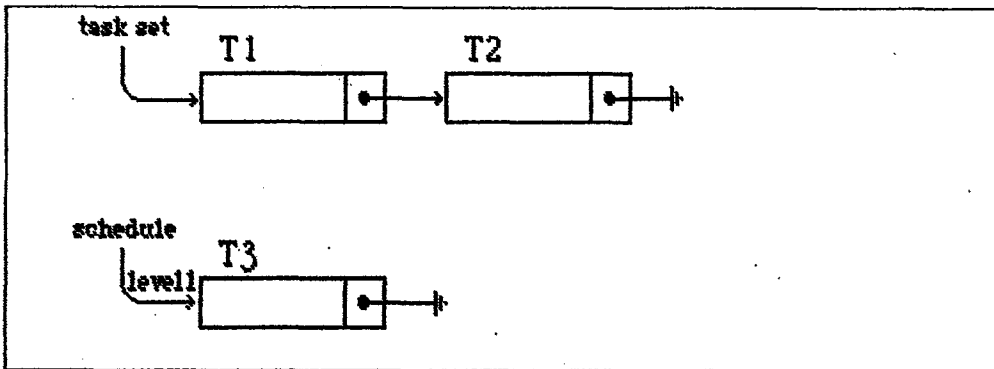
FIGURE 6.5 Illustration of the extended algorithm by a simplified example



(d)



(e)



(f)

FIGURE 6.5 Illustration of the extended algorithm by a simplified example (continued)

6.4. The Heuristic Function H

Clearly, at each level of the search, effectively and correctly identifying the immediate descendant is difficult but very important for the success of the algorithm. Function H becomes the core of the algorithm. In this section, the heuristics to construct function H are identified. First, some simple heuristics are evaluated, then integrated simple heuristics are considered. Because of the complexity of the problem, it is not expected that the use of simple heuristics alone will result in good performance. The purpose of evaluating their performance is to identify the candidates that are worthy of further exploration.

6.4.1. Simple Heuristics for Scheduling

The following is a list of simple heuristics for scheduling, and corresponding H functions defined on them :

- (a) minimum deadline first (Min_D) : $H(T) = D(T)$;
- (b) minimum start time first (Min_S) : $H(T) = S(T)$;
- (c) minimum computation time first (Min_C) : $H(T) = C(T)$;
- (d) minimum laxity first (Min_L) : $H(T) = D(T) - (S(T) + C(T))$.

6.4.2. Simulation Method and Results

The purpose of the simulation is to evaluate the performance of the different heuristics used for the function H. In each simulation, tasks are randomly generated. A number of tasks are collected as a task set. For each task set, an exhaustive search is performed to determine whether this task set has at least one feasible schedule or not. Those task sets

that are known to be schedulable are input to the local scheduling algorithm. Then, for each heuristic, the percentage of tasks sets scheduled is observed. This percentage gives the *success ratio* SR of the heuristic.

Since meeting deadlines is very important in real-time systems, the schedulability of tasks, i.e., whether or not tasks will finish before their respective deadlines, is considered as the performance metric.

For this simulation study, a local task generator program is written which given task generating parameters, generates two hundred schedulable task sets, each of which consists of six tasks. The listing of this program is given in Appendix A. It should be noted that for a set of six tasks, there are 720 permutations, each of which may or may not present a full feasible schedule. The program, after having generated a task set, performs an exhaustive search to see whether there is at least one full feasible schedule for the task set or not. If not, the task set is discarded, and a new one is generated.

The local task generator generates a task by specifying its resource requirements, its computation time and its deadline. It is assumed that the local node, has five resources : two active resources and three passive resources. The resource requirements of a task are chosen randomly with the condition that a task uses at least one active resource. A task needs a resource with probability 0.5.

The other generating parameters, to be set in the task generator program, are the mean and standard deviation values of the computation time distribution and of the laxity distribution of the tasks. The laxity distribution is used to generate the deadlines. These distributions are assumed to be normal distributions. In order to see the performance of the heuristics in different levels of *scheduling difficulties*, three different sets of tasks sets are generated by using three different laxity distributions, which indicate the tightness of the deadlines. Then, the performances of the heuristics used by the local scheduling algorithm are evaluated for each one of these.

The listing of the local scheduling program is given in Appendix B. In this program, the extension of the basic scheduling algorithm which uses the limited backtracking concept, is adopted. That is, in the scheduling process when an infeasible vertex is met, instead of simply announcing a failure, the task that has the second lowest value of the function H is tried to be appended the current feasible schedule. If this attempt fails, the program recursively backtracks to the immediate ancestor and attempts to schedule the task with the second value of the function H at that level.

The number of backtracks is limited by setting up a variable *counter* which counts the number of backtracks used in scheduling a set of tasks. If the counter exceeds a preset maximum value MC (*max_counter*); no further backtracking is allowed. In this way, even in the worst case, the time complexity of the algorithm will not be exponential. The simulation is performed for different values of MC, in order to show its effect.

The simulation results of using simple heuristics are presented in Table 6.1. In Table 6.1(a), Table 6.1(b), and Table 6.1(c), the computation time distribution is assumed to be $N(200,100^2)$ and the laxity distribution is assumed to be $N(100,100^2)$, $N(200,100^2)$, and $N(400,200^2)$ respectively.

From the tables, it can easily be seen that as deadlines become less tight, that is, as the mean of the laxity distribution increases, the difficulty in scheduling decreases, and the performance of the heuristics increases. It can also be concluded that when MC, the preset maximum value of backtracks is zero, that is, when backtracking is not allowed, none of the heuristics performs satisfactorily. Increasing the value of MC, up to 10 for example, causes a remarkable increase in the performance. But still, the performances of the heuristics are far from being good. It is also seen that increasing MC to a higher value than 10 does not make any change on the performance. In the case where laxity distribution is taken as $N(400,200^2)$ and MC is large, the heuristic *Min_D* performs reasonably well, but still does not achieve 100 per cent.

The observations from this simulation study, indicate that some traditional heuristics used in general operating systems, are not appropriate for tasks with timing constraints. For example, using *Min_C* is equivalent to using the shortest job first policy which is a heuristic sometimes adopted in nonreal-time scheduling, because it produces the minimum average waiting time for tasks. But, this simulation study shows that this heuristic does not perform satisfactorily in real-time systems.

6.4.3. Integrated Simple Heuristic Algorithms

Given that no single heuristic performs satisfactorily, integrated heuristics need to be attempted. The integrations are considered as simple as possible in order to keep the

MC	HEURISTICS			
	Min_D	Min_S	Min_C	Min_L
0	84.0%	34.0%	62.5%	48.0%
1	92.0%	38.0%	78.0%	56.0%
2	92.0%	38.0%	78.0%	60.0%
3	94.0%	40.0%	78.0%	62.0%
10	94.0%	40.0%	78.0%	62.0%
100	94.0%	40.0%	78.0%	62.0%

(a) Computation time distribution of tasks : $N(200,100^2)$,

Laxity distribution of tasks : $N(100,100^2)$;

MC	HEURISTICS			
	Min_D	Min_S	Min_C	Min_L
0	82.5%	46.0%	78.0%	56.0%
1	89.0%	49.5%	83.0%	65.5%
2	91.5%	54.0%	84.0%	68.0%
3	93.0%	54.5%	86.0%	69.0%
10	93.5%	54.5%	86.5%	71.0%
100	93.5%	54.5%	86.5%	71.0%

(b) Computation time distribution of tasks : $N(200,100^2)$,

Laxity distribution of tasks : $N(200,100^2)$;

MC	HEURISTICS			
	Min_D	Min_S	Min_C	Min_L
0	93.0%	54.5%	66.0%	77.0%
1	96.5%	60.0%	69.5%	84.5%
2	97.0%	62.5%	70.5%	87.5%
3	98.0%	63.5%	71.0%	89.0%
10	98.0%	63.5%	71.0%	90.0%
100	98.0%	63.5%	71.0%	90.0%

(c) Computation time distribution of tasks : $N(200,100^2)$,

Laxity distribution of tasks : $N(400,200^2)$;

TABLE 6.1 Simulation results of using simple heuristics

run time cost of the algorithm still low. Because Min_D performs much better than any other heuristics when used alone, it is considered to be the primary heuristic, and the others become the candidates to be combined with Min_D.

Following are the integrated simple heuristics and the corresponding definitions of H functions :

(a) Min_D and Min_C : $H(T) = D(T) + W * C(T)$;

(b) Min_D and Min_S : $H(T) = D(T) + W * S(T)$.

where W is a weight, and will be adjusted for different conditions.

Min_D and Min_L are not combined, because the information in Min_L is similar to Min_C and Min_D combined.

6.4.4. Simulation Results of Using Integrated Simple Heuristics

The same three sets of two hundred task sets generated for simple heuristics, are used to evaluate the performance of the above integrated simple heuristics. Table 6.2 shows the results.

In the table, the maximum success ratio SR, achieved by a particular H function is shown with the weight W that makes this possible. This weight that produces the maximum success ratio is determined assuming that the success ratio as a function of W has a single maximal point. Given this assumption, starting with a value of 0.5 for W and increasing it by 0.5 each time, the maximum of success ratios is determined until the success ratio starts to decrease after reaching a peak value. The value of W that produced the peak success ratio is the one shown in the tables.

It is observed that combining Min_D with Min_S improves the performance substantially. Although, Min_S does not perform well when used alone, it outperforms all others when it is integrated with Min_D. This is because Min_S by itself does not consider timing constraints and hence many tasks are liable to miss their deadlines. Combining Min_S with Min_D removes this shortcoming of the heuristic.

MC	HEURISTICS			
	Min_D+W*Min_C		Min_D+W*Min_S	
	W	SR	W	SR
0	1.0	88.0%	0.5	88.0%
1	1.0	94.0%	0.5	94.0%
2	1.0	94.0%	0.5	94.0%
3	0.5	94.0%	0.5	96.0%
10	0.5	94.0%	0.5	96.0%
100	0.5	94.0%	0.5	96.0%

(a) Computation time distribution of tasks : $N(200,100^2)$,

Laxity distribution of tasks : $N(100,100^2)$;

MC	HEURISTICS			
	Min_D+W*Min_C		Min_D+W*Min_S	
	W	SR	W	SR
0	1.0	87.5%	1.0	87.5%
1	0.5	91.5%	1.5	92.5%
2	0.5	93.0%	1.5	94.5%
3	0.5	94.0%	2.0	96.0%
10	0.5	94.5%	1.0	97.0%
100	0.5	94.5%	1.0	97.0%

(b) Computation time distribution of tasks : $N(200,100^2)$,

Laxity distribution of tasks : $N(200,100^2)$;

MC	HEURISTICS			
	Min_D+W*Min_C		Min_D+W*Min_S	
	W	SR	W	SR
0	0.5	91.0%	1.0	96.5%
1	0.5	94.5%	1.5	98.0%
2	0.5	95.0%	1.0	99.5%
3	0.5	95.5%	1.0	100.0%
10	0.5	97.0%	0.5	100.0%
100	0.5	97.0%	0.5	100.0%

(c) Computation time distribution of tasks : $N(200,100^2)$,

Laxity distribution of tasks : $N(400,200^2)$;

TABLE 6.2 Simulation results of using integrated simple heuristics

Finally, using the heuristic minimum deadline first integrated with minimum earliest start time first as the function H , along with limited backtracking makes the algorithm perform very well, close to an optimal algorithm that has an exponential time complexity.

6.5. Application Considerations

In this section, how the algorithm can be applied to the following cases, is discussed :

- (a) on-line heuristic scheduling;
- (b) scheduling when tasks arrive in a batch;
- (c) non-preemptive scheduling and the inclusion of periodic tasks.

6.5.1. On-line Heuristic Scheduling

As noted before, this heuristic approach is used to decide whether a new schedule exists for the tasks that have already been scheduled to execute on a node plus the task that just arrived at that node. Now, a technique for making this decision will be presented :

Suppose n tasks are scheduled to execute on a node, i.e., there is a full feasible schedule for the n tasks. Suppose m of these tasks begin execution and then task T arrives. Because task preemption is not allowed, the m tasks in execution will be allowed to run to completion. Let EAT be the vector indicating the earliest available times for all the resources, taking into account the fact that m tasks are in execution. With this EAT, if a full feasible schedule is found for the $(n-m)$ tasks plus the newly arrived task T , then T can be said to be guaranteed. In this way, this scheduling algorithm can be used to decide whether a task which arrives during the execution of m tasks on a node can be scheduled to execute on that node.

The method just described for on-line scheduling assumes that to decide whether the new task T is schedulable, a full feasible schedule has to be determined for $(n-m)+1$ tasks, that is, the scheduling algorithm has to be executed on the $(n-m)+1$ tasks, given the EAT.

6.5.2. Scheduling When Tasks Arrive in a Batch

Another issue is how to perform on-line scheduling when a number of tasks arrive in a batch. Assume that p tasks have been guaranteed but not yet begin execution, when q tasks arrive. Augmenting the schedule for the p tasks with the $q (>1)$ tasks becomes difficult. Suppose the heuristic algorithm is used to determine a schedule for the $p+q$ tasks. If such a schedule does not exist, this means that *not all* of the q tasks are schedulable. But, a subset of the q tasks may be schedulable. To find out this subset, the heuristic algorithm has to be repeatedly applied to subsets of the q tasks. The problem here is to determine which task is to be discarded from a given set before the algorithm is re-applied.

The best thing to do is the following : when tasks arrive in a batch, each of them should be considered one by one, in some order, say, earliest-deadline-first. If a full schedule is found when a task is added, the task is kept in the new schedule. If a full feasible schedule cannot be found for this task, it is nonguaranteed, and it becomes a candidate to be sent to some other node.

6.5.3. Non-preemptive Scheduling and the Inclusion of Periodic Tasks

This scheduling algorithm is developed assuming that tasks cannot be preempted. Two reasons for this are as follows :

(A) Suppose the first task in a schedule is dispatched and then a new task arrives. The requirements of the tasks and of the newly arrived task may be such that even if the

currently running task is preempted to run the newly arrived task, all tasks will meet their deadlines. Whether this is true or not can be checked easily when only CPU requirements of the tasks are taken into account as in [36]. Inclusion of the general resource requirements considerably increases the complexity of the check.

(B) Preemption also introduces the need to take into account the consistency of resources. For example, if R_1 is a file and both T_1 and T_2 modify the file, then a schedule where T_2 preempts T_1 may result in R_1 becoming inconsistent. Hence, once preemption is allowed, considerations such as this enter the picture.

Primarily for these reasons, in this study, the heuristic scheduling without task preemption is discussed. It should also be recognized that when preemption is not permitted, resource utilization may decrease and the number of tasks guaranteed may also decrease.

Another implication of doing non-preemptive scheduling is that a task may not be schedulable mainly because of its arrival time. For example, suppose a task T_1 with deadline 200 and computation time 100 is the first task in a schedule, and begins execution at time equals zero. At time one, a task T_2 with deadline 100 and computation time 80 arrives. If T_1 were not in execution, T_2 may be schedulable. If it was known that T_2 would arrive at time one, it might be possible to schedule all tasks in the current schedule plus the new task T_2 , such that they all finish before their deadlines.

In any dynamic system, such information about future task arrivals will not be available. However, for an important type of tasks, called *periodic tasks*, such information is available and can be used to perform intelligent scheduling, because periodic tasks are tasks that have to be executed at regular intervals specified by their periods. In general, each periodic task will be generated at the beginning of its period. The following technique is advised to be utilized in case of periodic tasks: if a *nonperiodic task*, arriving before the beginning of the next period, has a deadline in or beyond the next period, the next periodic task will be generated and sent to the scheduler before the nonperiodic one. Each periodic task has an *earliest start time* equal to the beginning of its period so that it cannot be scheduled before that time. Therefore, the definition of ST needs a slight change with the inclusion of periodic tasks, it should be redefined as:

$$ST(T) = \text{MAX}(EAT_i \text{ where } T \text{ needs } R_i, \text{ and the earliest start time of task } T).$$

The earliest start time for a nonperiodic task is defined as its arrival time, so that it can start any time after its arrival.

VII. DISTRIBUTED SCHEDULING SCHEME

In this part, the strategy for scheduling tasks dynamically in a distributed hard real-time system is presented. The distributed scheduling algorithm developed by Ramamritham, Stankovic, and Zhao [2], is chosen as the algorithm to study on, and is implemented with some modifications.

Since the local scheduling algorithm, explained in Part VI, with the heuristic function minimum deadline first integrated with minimum start time first :

$$\text{Min_D} + W * \text{Min_S},$$

has been shown to be highly successful, it is incorporated in the distributed scheduling scheme as the local scheduling algorithm underlying the guarantee routine on each node.

The performance of the overall system heavily depends on how distributed scheduling is done, that is to say, on how the node to send a task which cannot be guaranteed locally, is detected. In this part, the details of the distributed scheduling algorithm are considered first. Then a sequence of simulation studies is performed in order to observe how the system performs under different conditions. The performance of the algorithm is also compared with that of three other algorithms.

7.1. Generation and Transmission of the Node Surplus

The purpose of generation and transmission of node surplus from a node is to help other nodes to correctly make the decision about which node a task should be sent to during focused addressing and which nodes the request-for-bid messages should be sent to during bidding. Obviously, it is neither practical nor possible to let nodes have precise state information about other nodes because of the communication delay involved.

The notion of the surplus of a node, as used in this distributed scheduling algorithm is its ability to guarantee tasks from the other nodes. A node's surplus is in reality a vector, with one entry per resource on that node. Each entry indicates the total amount of time, in past window, during which a resource is not used by the *local* tasks.

Each node periodically calculates its node surplus and sends it to a subset of the remaining nodes. A node sorts other nodes according to the number of tasks received from them that were guaranteed on this node in a past time window. Then, according to this sorted node list, a node selects a subset of nodes to send information on its own current node surplus. The subset is chosen such that nodes in the subset will potentially use this information in deciding whether or not to send a task to this node. Therefore, the nodes, which recently sent more tasks to this node, will more likely to be selected.

Broadcasting the node surplus information in large network is not suggested, because it causes heavy traffic and therefore can increase communication overheads. Because of the fact that communication takes non-negligible time delay, and that resource requirements of tasks from different nodes may be different, the surplus information from a node may not always be useful for some other nodes. Sending a node's surplus information to a subset of other nodes, reduces the communication traffic, and lets a node send its surplus information only to those nodes where its surplus information is potentially needed. These nodes will typically be those that have tasks which require the resources that are less utilized by the local tasks on the sending node.

Of course, if the network is small, the surplus information can be sent to all the other nodes.

7.2. Focused Addressing and Requesting for Bids

When a task, T , arrives at a node N_j , the local scheduler is invoked to try to schedule the newly arrived task on the node. If it is impossible to schedule the task locally, node N_j 's bidder comes into the picture which is responsible for doing focused addressing and requesting bids.

For $j = 1, \dots, n$ and $j \neq i$, the bidder on node N_i estimates $ES(T, j)$ which is the number of instances of task T that node N_j can guarantee.

This estimation is made according to the node surplus information available on node N_i and provides a good indication of the likelihood of a site being able to guarantee a given task.

For example, assume that the computation time of task T is 250 time units. Suppose, node N_s is estimated to have a minimum surplus of 400 time units on each of the resources needed by T . Then, the surplus of N_s with respect to the resources needed by task T is 400, and the estimated number of instances of task T that node N_s can guarantee is $400/250$ which is 1.6.

In the original algorithm [2], it is suggested to continue the process as below:

Node N_i sorts other nodes according to their $ES(T, j)$, in descending order. The first k nodes are selected to participate in focused addressing and bidding. The value of k is decided such that the sum of $ES(T, j)$ of the k nodes is larger than or equal to SGS , the *System-Wide Guarantee Surplus*. This is a tunable parameter of the system. If the first node N_f among the k nodes has its $ES(T, f)$ larger than FAS , the *Focused Addressing Surplus*, another tunable parameter, node N_f is the focused node. The task is immediately sent to that node. The remaining $k-1$ nodes are sent request-for-bid messages in parallel, to handle the case where the focused node cannot guarantee the task.

Whereas, in this study, it is preferred to modify this process as follows:

The node N_j , having the maximum value of $ES(T, j)$ is selected as the focused node on condition that $ES(T, j)$ is larger than FAS . The task is immediately sent to node N_j , and request-for-bid messages are sent to each one of the remaining nodes in parallel.

The purpose of this modification is to increase the chance of being guaranteed of task T at another node, in case that it cannot be guaranteed at the focused node. Since in focused addressing, out-of-date state information of the nodes is used, there is a risk of making wrong decisions. Consequently, a task T may not be guaranteed, not because there are no nodes that can guarantee it, but because the nodes that can guarantee it, are not sent request-for-bid messages. By sending request-for-bid messages to all the other nodes, this risk can be tolerated. But, it should also be kept in mind that, this method is preferable as long as the network is small. Because when there are too many

nodes in the network, there will be too many transmitted messages which will increase the communication overhead.

A request-for-bid message includes information about the deadline, the computation time and the resource requirements of the task as well as the latest bid arrival time, that is, the time by which bids should reach the focused or requesting node to be eligible for further consideration. The latest bid arrival time for a task T , $LBA(T)$, is estimated as follows:

$$LBA(T) = D(T) - C(T) - (TD + SD),$$

where $D(T)$ is the deadline of T , $C(T)$ is the computation time of T , TD is the network-wide average transmission delay between two nodes, and SD is the average scheduling delay on a node. Thus, on the average, before $LBA(T)$ there will be sufficient time to send the task to a bidder node, for it to be scheduled there and then be executed before its deadline.

7.3. Bidding

When a node receives a request-for-bid message, it calculates a bid for the task provided that there is enough time for bidding. Each request-for-bid message contains a deadline for response (latest arrival time of a bid). If the responding node estimates that it cannot deliver the bid to the requesting node on time, it does not bid. Therefore, only viable bids will reach the requesting host and the communication overhead is reduced.

The *bid* is purely a number which indicates the number of instances of the task the bidder node can guarantee. The calculation is done in two steps:

First, an upper bound of the bid, Max-Bid is calculated by the below formula:

$$\text{Max-Bid} = \frac{\text{Task Deadline} - \text{Estimated Earliest Arrival Time of the Task}}{\text{Task Computation Time}}$$

The earliest arrival time of the task to the bidder node is estimated in an optimistic manner to be the sum of current time, the minimum message delay in transmitting the bid, and the minimum delay in sending the task to this node. Max-Bid is the best possible bid that this node can make assuming ideal availability of resources that the task needs.

In the second step, the actual bid is calculated by performing a binary search between zero and Max-Bid. In each step of the binary search, a given number of instances of task T are temporarily inserted into the current schedule of this node, and the guarantee routine is called to see if the inserted instances can also be guaranteed. At the end of the search, the maximum number of instances of the remote task T that this node can actually guarantee without endangering previously guaranteed tasks, is obtained. This number, if above a predefined limit, becomes the bid. The bid is sent to the node which was selected for focused addressing if there is one. Otherwise, the bid is sent to the original node which issued the request-for-bid message. The inserted instances of the remote task are removed from the schedule on a bidder's node. Therefore, the schedule on the bidder's node is not affected by the bid it makes. This implies that a node does not reserve the resources needed by the tasks for which it bids since a node will typically bid for multiple tasks and multiple bids will be received for a task, reservation of resources will result in pessimistic bids and therefore may reduce the system performance.

7.4. Bid Evaluation

When a node receives a bid for a given task, and the bid is higher than a certain limit, high-bid (HB), the node awards the task to the bidding node immediately and all other bids for this task, that arrived earlier or may arrive later, are discarded. If all the bids, that have arrived, for a given task are lower than the high-bid, the node postpones making the awarding decision until the latest bid arrival time of the task. At that time, the task will be awarded the highest bidder if any. All the bids that arrive later will be discarded.

7.5. Response to Task Award

When the awarded task arrives at the highest bidder, the local scheduler on that node is invoked to see if the task can be guaranteed. It should be noted that the state of the node may change after making a bid and since resources needed by the task were not reserved, the task may or may not be guaranteed. If the task is not guaranteed, it is rejected.

7.6. Simulation Model

In this section, the simulation model on which a sequence of simulation studies are conducted, is introduced. The results and observations of these studies are presented in Section 7.7.

7.6.1. System Model

The system model is assumed to be physically distributed and composed of a network of five nodes (multiprocessors) each of which has its own local memory. All internode distances are considered to be the same.

The nodes in a network can be physically connected in a variety of ways, namely *communication topologies*. In order to see the performance of the algorithm in different conditions, the simulation studies are performed on two different network communication topologies :

(A) **Fully Connected Communication Network** : In such a network, each node is directly linked with all other nodes in the system. The basic cost of this configuration is very high, since a direct communication line must be available between every nodes. The basic cost grows as the square of the number of nodes. In this environment,

however, messages between the nodes can be sent very fast. The first simulation system model with such communication topology is shown in Figure 7.1.

(B) **Star Communication Network** : In a star network, one of the nodes in the system is connected to all other nodes. None of the other nodes is connected to each other. The basic cost of this system is linear in the number of nodes. The communication cost is also low, since a message from Node_i to Node_j requires at most two transfers. This speed may be somewhat misleading, however, since the central node may become a bottleneck. Consequently, even though the number of message transfers needed is low, the time required to send these messages may be high. Figure 7.2 shows the second simulation system model with such communication topology. In this model, the central node, S, is completely dedicated to the message switching task.

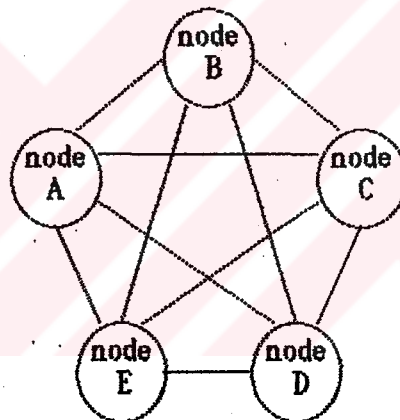


FIGURE 7.1 Simulation system model 1 (Fully Connected)

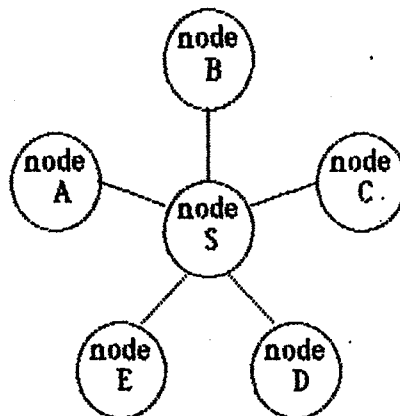


FIGURE 7.2 Simulation system model 2 (Star)

Messages pass through a communication line in a pipe-lined fashion with only one message occupying a channel at a given time, in a given direction. When a message is in a line, if there is another message that needs to be transmitted, the latter message must wait until the first has left. This situation is called a *conflict*. The total time for transmitting a message from one node to another without any conflict is denoted as *no conflict message delay* (MD). In star network, since a message from one node to another passes through two communication lines, the time taken by a message to pass through a line, without any conflict, is half of the message delay.

The delay involved in transferring a task through the network is assumed to be the message delay plus 10 per cent of the computation time of the task. That is, it is assumed that transferring a task requires higher communication overheads than a message, and this overhead is proportional to the computation time of the task. Again in star network, the time taken by a task to pass through one communication line is half of this amount.

In the simulation program, since the network is sufficiently small, a node sends its surplus information to all the other nodes in the system. When the network is large, the node surplus information should be sent to only a subset of selected nodes. Nodes that potentially need such information should be selected. A good selection policy will reduce the number of messages transmitted in the network, while letting the nodes obtain such information if needed.

The message traffic created by the transmission of surplus information as well as all other messages generated in the course of scheduling is also taken into account in the simulation model.

These two network topologies and the communication protocol just described are chosen for simulation in order to observe the effect of the communication overhead on the performance of the algorithm.

7.6.2. Node Model

It is assumed that a stream of tasks arrives locally to each node as a Poisson process. The nodes are considered to be heterogeneous in the sense that each node may

have a different arrival rate of local tasks, but homogeneous in the sense that a task submitted to any other node in the network can be executed there. The fact that local task arrival rates on different nodes may be different, results in differences in the loads of the nodes. In the simulation studies, the term *system local task arrival rate*, R , is used to refer to the sum of the local task arrival rates of all the nodes in the system.

In the simulation model, two of the five nodes (node A and node B) are assumed to have equal loads which are higher than the remaining three nodes. Given the system arrival rate, R , the local task arrival rate for each node is considered to be as follows :

(a) for nodes A and B : $0.375R$;

(b) for node C : $0.125R$;

(c) for nodes D and E : $0.0625R$.

Each node is assumed to contain five *resources* which may be demanded by tasks, including two active resources (processors) and three passive resources. A resource can be serially shared by tasks. The resource requirements of a task are determined randomly, provided that a task needs a resource with probability 0.5. Moreover, each task requires at least one of the active resources and zero or more passive resources.

Both the *computation time* and *laxity* of tasks are considered to be normally distributed.

Since the *dispatcher* has to be invoked each time any task completes execution, the run-time cost of the dispatcher is included in the computation time of every task.

The simulation model also assumes that the scheduler tasks such as the *bidder* and the *local scheduler* are executed on a co-processor dedicated to scheduling.

The model is based on the assumption that there is a *communication module*, executing on a co-processor which 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 different tasks execute.

The purpose of using such co-processors (or system processors) is to offload the scheduling algorithm and the other operating system overhead from the application

tasks both for speed, and so that this overhead does not cause uncertainty in executing already guaranteed tasks.

7.7. Simulation Results and Observations

The distributed scheduling algorithm explained in the previous sections, is implemented and tested under different conditions, using the simulation model presented in Section 7.6. Appendix D contains the listing of the simulation program implementing this algorithm. Since the algorithm uses a technique that combines bidding and focused addressing, the term *FB* will be used to describe this algorithm. Before presenting the simulation results, a general information about what kind of simulation studies are performed will be given, discussions on the observations then follow.

In the simulation studies, the computation time distribution of tasks is considered to be normally distributed having a mean of 200 and a standard deviation of 100, denoted as $N(200,100^2)$.

The cases with three different laxity distributions are tested in order to study the effect of tasks laxity distributions on the performance. These cases are :

- (a) **Low laxity** (L_LAX) : laxity distribution of tasks is $N(300,150^2)$;
- (b) **Medium laxity** (M_LAX) : laxity distribution of tasks is $N(450,150^2)$;
- (c) **High laxity** (H_LAX) : laxity distribution of tasks is $N(600,150^2)$.

In order to observe the changes in system's performance under different system loads, the simulation is performed under light, moderate and heavy system loads:

- (a) **Light load** (L_LOAD) : system arrival rate, R , is 8 tasks per 600 time units;
- (b) **Moderate load** (M_LOAD) : system arrival rate, R , is 16 tasks per 600 time units;
- (c) **Heavy load** (H_LOAD) : system arrival rate, R , is 24 tasks per 600 time units.

Consequently, the local task arrival rates for each node, under these different system loads are as shown in Table 7.1.

SYSTEM LOAD	LOCAL TASK ARRIVAL RATE					
	NODE A	NODE B	NODE C	NODE D	NODE E	R
L_LOAD	3/600	3/600	1/600	0.5/600	0.5/600	8/600
M_LOAD	6/600	6/600	2/600	1/600	1/600	16/600
H_LOAD	9/600	9/600	3/600	1.5/600	1.5/600	24/600

TABLE 7.1 Nodes' local task arrival rates under different system loads

Three different cases for task laxity distributions and three different cases for system load, result in a combination of nine different cases, each of which has a specified task laxity distribution and a specified system task arrival rate. Hence, nine groups of tasks are generated by the global task generator program in order to be used during the simulation studies. The listing of the global task generator program is given in Appendix C.

The performance of the algorithm is tested under different no conflict message delay, MD, values as well. The purpose is to examine how communication delay affects the system performance.

In the simulation studies, the performance of the algorithm FB, is also compared to that of three other algorithms explained below :

(a) *Noncooperative scheduling algorithm* (NC) : In this algorithm, whenever a task cannot be guaranteed locally, the task is discarded. No attempt is made to send the task to other nodes.

(b) *Random scheduling algorithm* (R) : In this algorithm, when a task cannot be guaranteed by the local node at which it arrives, the node randomly selects another node and directly sends the task to that node. The advantage of this algorithm is that, it uses the minimum communication overhead to determine where to schedule a task in the network. The disadvantage is that, it is easy to send a task to an improper node because of the randomness.

(c) *Bidding* (B) : This algorithm, whenever a task fails, do not select a focused node to send the task, as in the algorithm FB, but sends a request-for-bid message to the other nodes, and then sends the task to the node which offers the best bid. If there is no good

bid available for the task, it is assumed that no node in the network is able to guarantee the task.

The listings of the simulation programs which implement these three scheduling algorithms are not given because of the space limitations. One may refer to the diskette for the program files.

In order to observe the effect of different network topologies on the performance of the algorithms, all these simulations are performed on both of the below communication network topologies, explained in Section 7.6.1 :

- (a) ***Fully connected communication topology*** (FC);
- (b) ***Star communication topology*** (S).

7.7.1. Effect of Laxity Distribution of Tasks

The purpose of this study is to examine how the differences in the laxity distribution of tasks, affect the performance of the distributed scheduling algorithm FB.

The term *percentage of nonguaranteed tasks*, denoted as "% NG," is used to indicate the system performance.

Three different laxity distributions (L_LAX : $N(300,150^2)$, M_LAX : $N(450,150^2)$, and H_LAX : $N(600,150^2)$) are tested as follows :

- (a) under moderate system load where system arrival rate (R) is 16 tasks per 600 time units, and with different no conflict message delay values (Figure 7.3 and Figure 7.4 show the simulation results for fully connected network topology and for star network topology respectively);
- (b) under three different system loads (L_LOAD : $R=8/600$, M_LOAD : $R=16/600$, H_LOAD : $R=24/600$), with a constant no conflict message delay (MD) value which is taken to be 36 time units (Figure 7.5 and Figure 7.6 show the simulation results for fully connected network topology and for star network topology respectively).

From the simulation results, it is easily observed that the task laxity does affect the system performance.

As seen from Figures 7.3, 7.4, 7.5 and 7.6, when the mean of tasks' laxity distribution increases, the percentage of tasks nonguaranteed decreases significantly.

From Figure 7.3, it is observed that when laxity increases from L_LAX to H_LAX, the percentage of tasks nonguaranteed decreases by an amount between nine and 14 per cent, for different values of MD. But as seen from Figure 7.4, on star topology this decrease is not very significant for high values of MD. For example, when MD is 96, there is only a decrease of three per cent on the number of nonguaranteed tasks. This implies that, when message delay is very large, the increase in laxity does not affect the system performance on star communication topology, as much as it does on fully connected communication topology.

Figures 7.5 and 7.6 show that, increasing the task laxity, decreases the percentage of tasks nonguaranteed under each one of the different system loads, on both of the communication topologies. This decrease is more obvious when the system load is light or moderate than when the system load is heavy. This reflects the fact that, when the system arrival rate is high, there are so many tasks to be scheduled in the system that increasing the mean of the task laxity distribution does not result in a significant increase in system performance.

7.7.2. Effect of Communication Delay

In this section, how the communication delay affects the system performance of the algorithm FB, is examined. In the simulation studies, the term *percentage of guaranteed tasks*, denoted as "% G," is used to indicate the system performance.

The first set of simulation studies with different no conflict message delay (MC) values, is performed under moderate system load ($R=16/600$), on two groups of tasks having different laxity distributions. One group of tasks is generated by using a low laxity distribution ($N(300,150^2)$), and the other by using a high laxity distribution ($N(600,150^2)$). The performance observations of these two groups, with different MD values are shown in Figures 7.7 and 7.8.

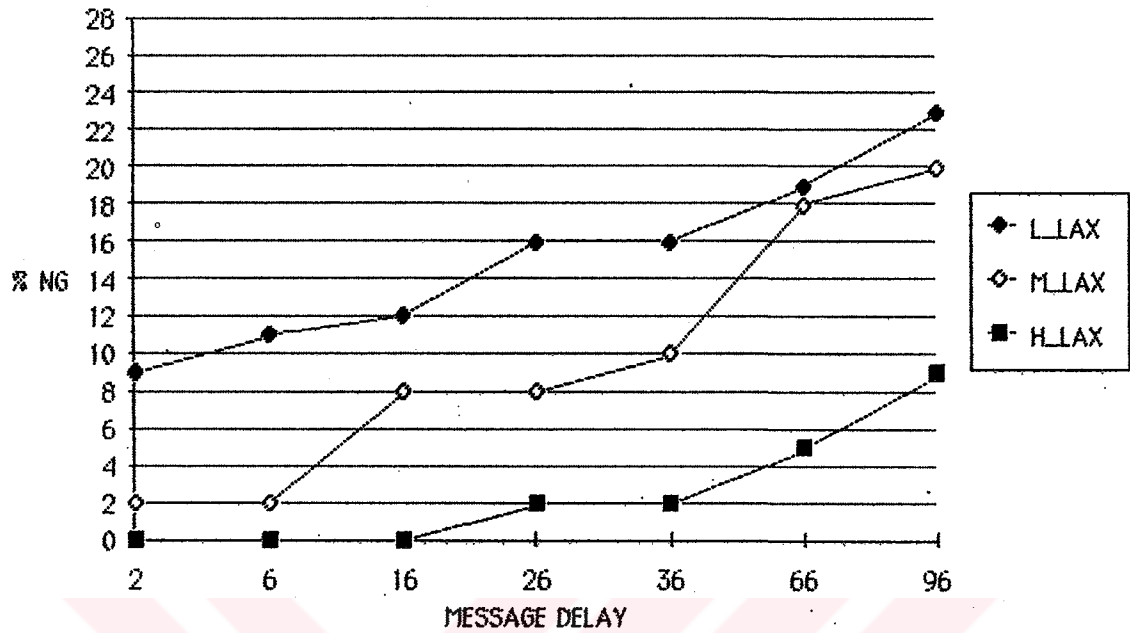


FIGURE 7.3 Effect of task laxity when $R=16/600$ and Topology=FC

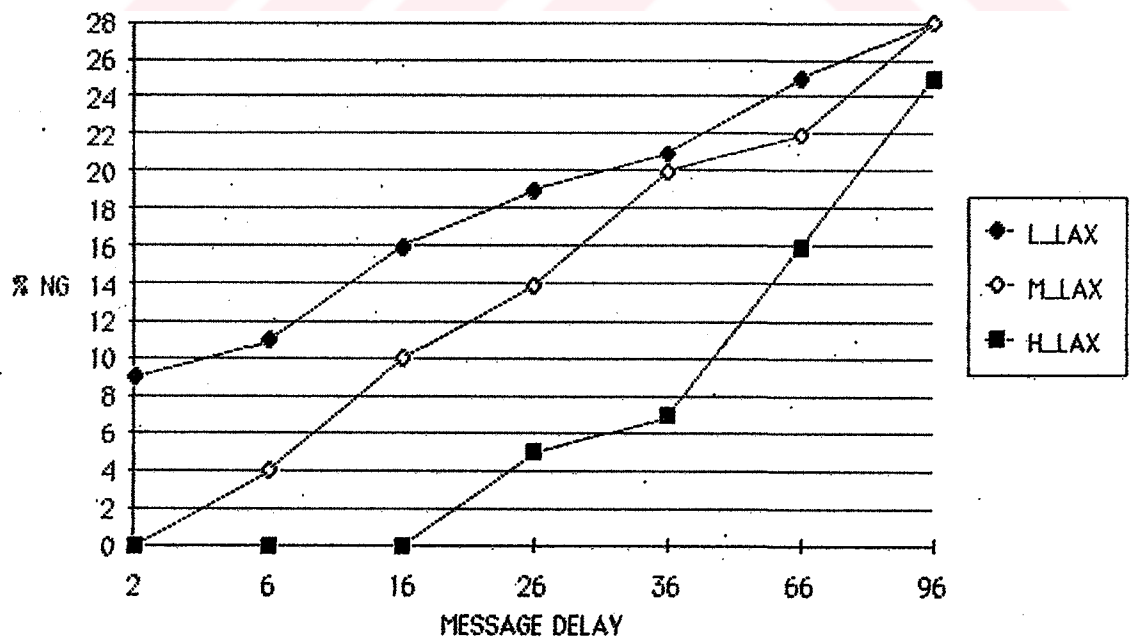


FIGURE 7.4 Effect of task laxity when $R=16/600$ and Topology=S

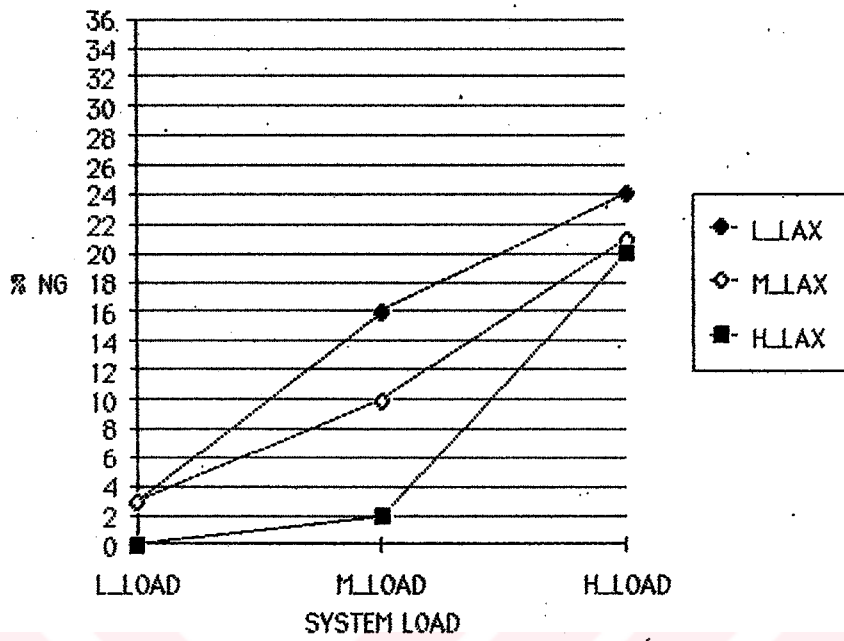


FIGURE 7.5 Effect of task laxity when MD=36 and Topology=FC

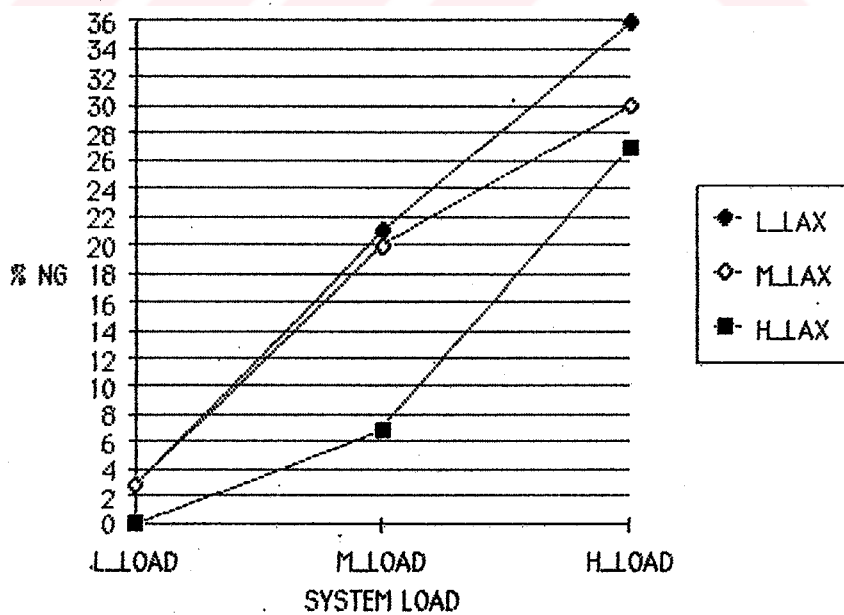


FIGURE 7.6 Effect of task laxity when MD=36 and Topology=S

on star topology than it does on fully connected topology. This implies that the communication network topology of a system, is also an important factor in tolerating high communication delays.

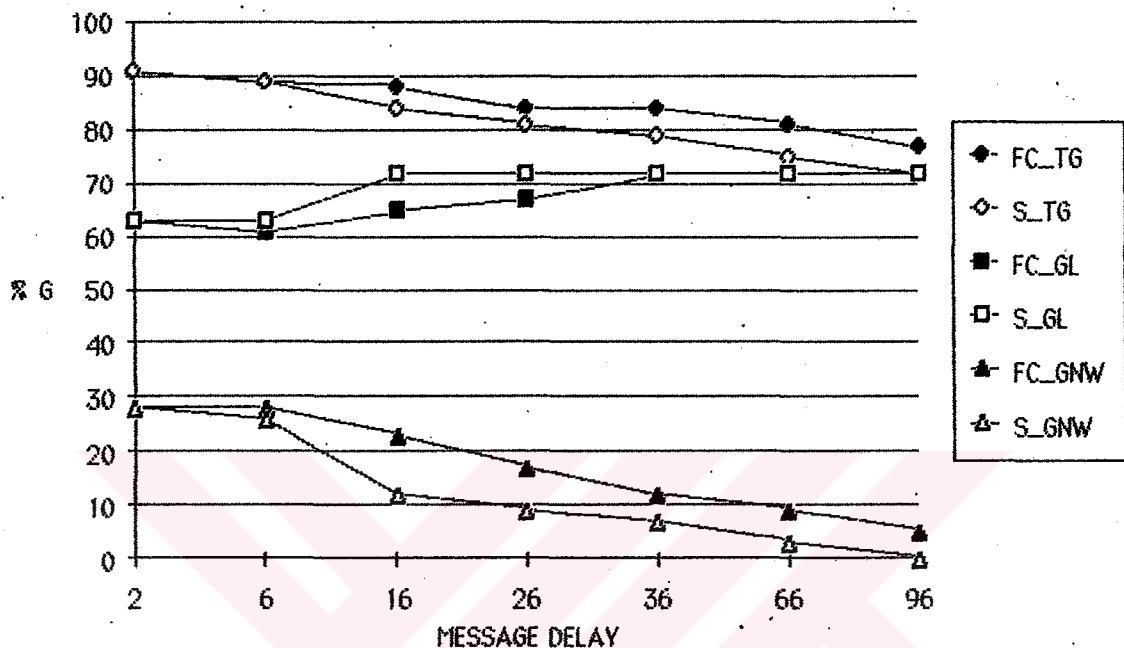


FIGURE 7.7 Effect of MD under M_LOAD and L_LAX

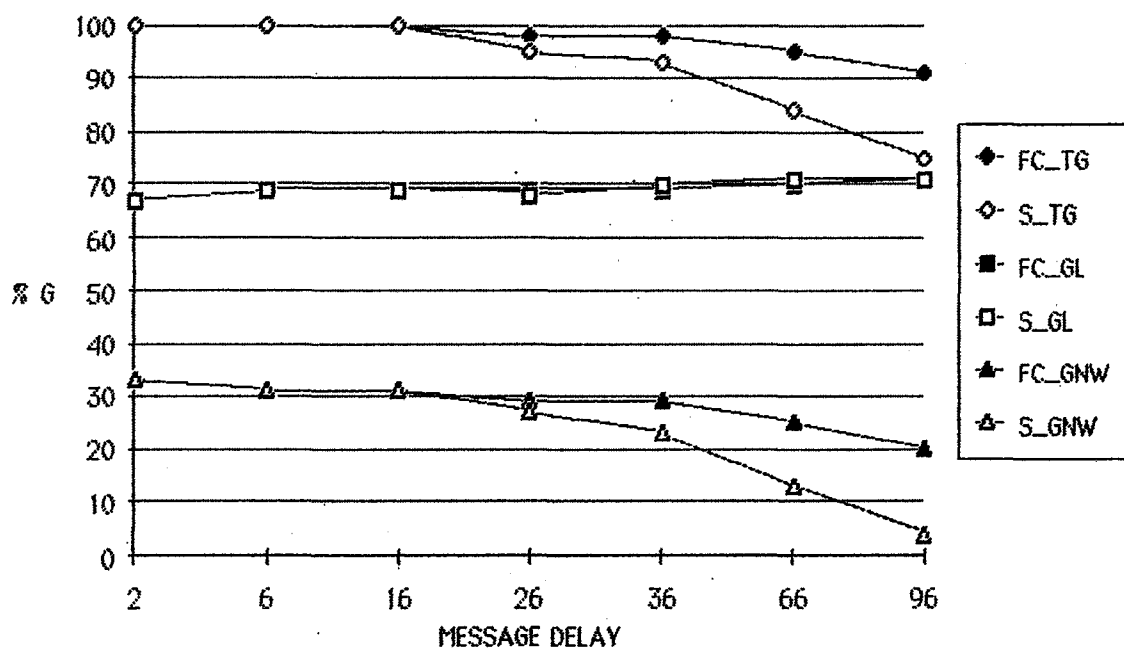


FIGURE 7.8 Effect of MD under M_LOAD and H_LAX

As mentioned before, in Figure 7.7, the lines "FC_GNW" and "S_GNW" indicate the percentage of tasks guaranteed network wide, on two different topologies, under moderate load and low laxity. Figures 7.9 and 7.10, further present the details about how these tasks guaranteed at remote nodes are actually guaranteed. Figure 7.9 shows the results for fully connected system model, and Figure 7.10, for star system model.

According to the distributed scheduling algorithm FB, there are three possible ways for a task to be guaranteed network wide :

(A) When a task cannot be guaranteed locally, the local scheduler, if it finds a node having sufficient surplus to guarantee it, sends the task to that node through focused addressing. Hence, a task can be guaranteed at the focused node. This first way of guarantee is named as *guaranteed by focused addressing* and denoted as "G_FA" in the figures.

(B) The local scheduler, in addition to sending the task to the focused node, sends request-for-bid messages to the remaining nodes, to handle the case where the task cannot be guaranteed at focused node. If this happens, the focused node evaluates the bids arrived for this task, and sends the task to the best bidder, if there is any, so that the task has a chance of being guaranteed at this "second step" node. This way of guarantee is called *guaranteed by focused addressing and bidding* and denoted as "G_FAB" in the figures.

(C) In case that there is no focused node having sufficient surplus to guarantee the task, the local scheduler starts the bidding process, and then sends the task to the node which offers the best bid. So a task can be guaranteed at the bidder node. This third way of guarantee is named as *guaranteed by direct bidding* and denoted as "G_B" in the figures.

As seen from Figures 7.9 and 7.10, there is no task guaranteed by focused addressing and bidding (FAB), when $MD \geq 66$ time units on fully connected topology, and when $MD \geq 16$ time units on star topology. Because when communication delay is high, it is very difficult to find enough time to attempt to schedule a task which is not guaranteed at focused node, at a second step node. This effect of MD, is much more obvious on star system model.

These figures also show that, at high message delays, guaranteeing by direct bidding (B), becomes difficult as well. No task is guaranteed by direct bidding when $MD \geq 96$ time units for fully connected system model (Figure 7.9), and when $MD \geq 66$ time units on star system model (Figure 7.10). This reflects the fact that at high

communication delays the message traffic required by bidding process creates an overhead.

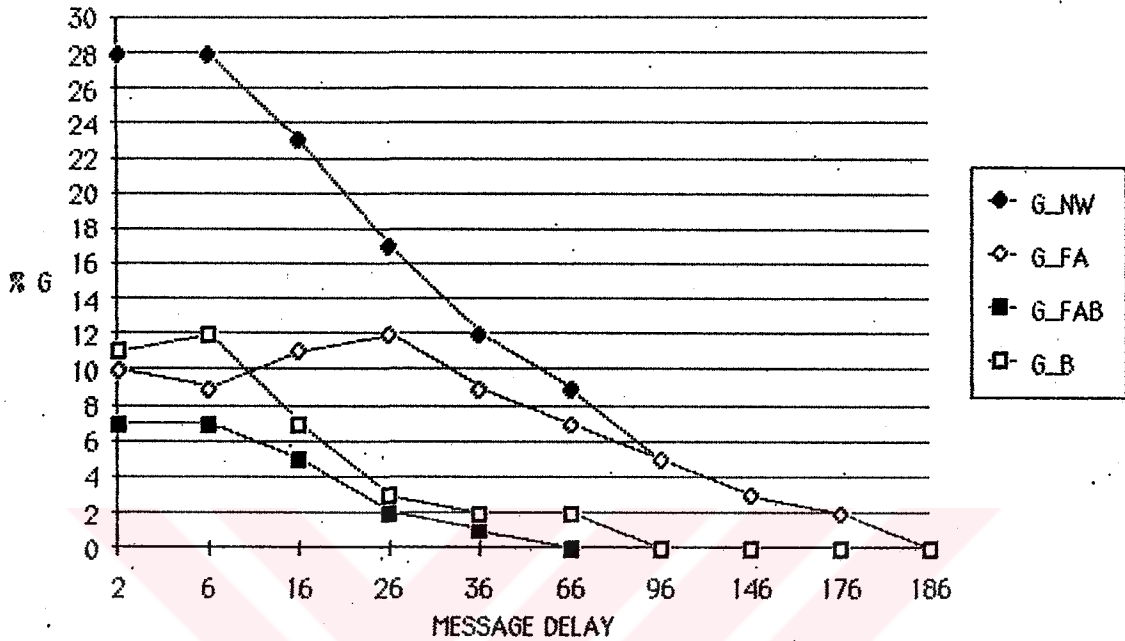


FIGURE 7.9 Effect of MD (details of FC_GNW)

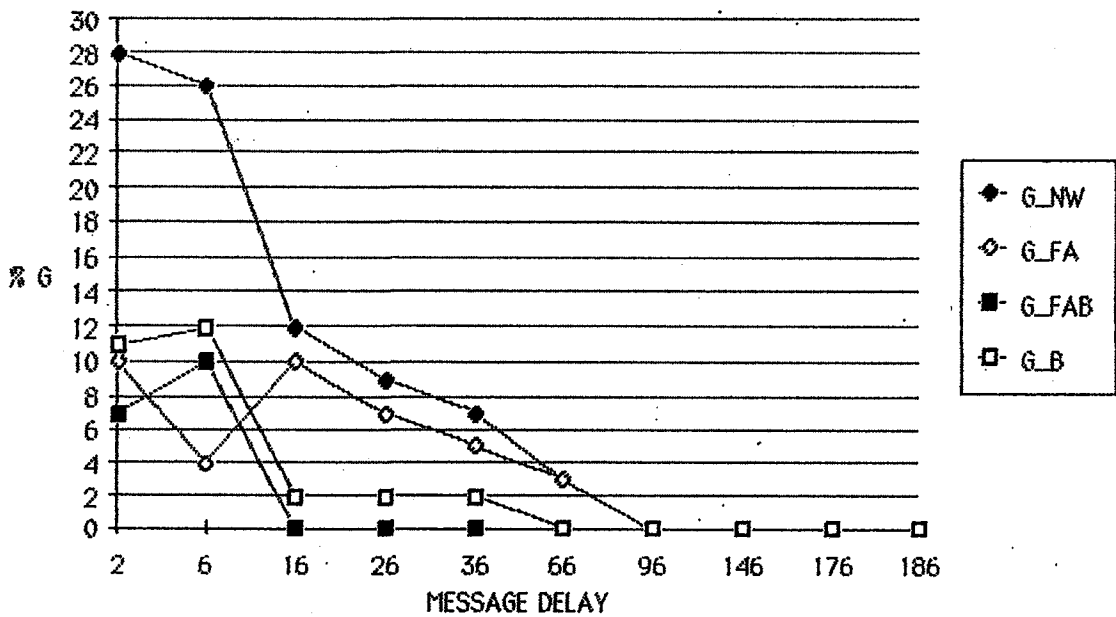


FIGURE 7.10 Effect of MD (details of S_GNW)

The system performance on fully connected topology is not as sensitive to MD as it is on star topology. For example, on star topology, there is no task guaranteed network wide when $MD \geq 96$ time units, whereas, on fully connected one, at $MD=96$ time units, the percentage of tasks guaranteed network wide is five. Moreover, this percentage remains positive for much higher values of MD, and finally becomes zero at $MD=186$ time units.

In order to observe the effect of the communication delay under different system loads, a set of simulations is performed. In these studies, tasks' laxity distribution is chosen to be low laxity ($L_LAX : N(300,150^2)$), and the performance of the algorithm is tested under light, moderate, and heavy system loads ($L_LOAD : R=8/600$, $M_LOAD : R=16/600$, and $H_LOAD : R=24/600$), on both of the fully connected and star system models (FC and S). The results obtained are presented in Figure 7.11.

It is observed that, as MD increases from two to 96 time units, the decrease in the percentage of guaranteed tasks is :

- (a) under light load : six per cent for FC topology, and 10 per cent for S topology;
- (b) under moderate load : 14 per cent for FC topology, and 19 per cent for S topology;
- (c) under heavy load : 22 per cent for FC topology, and 24 per cent for S topology.

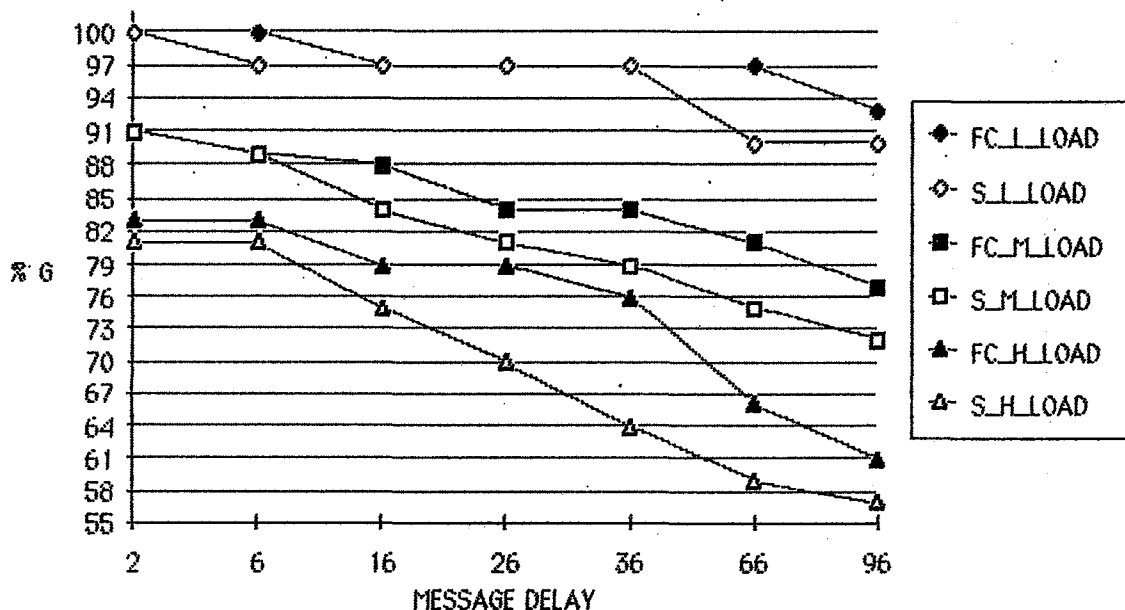


FIGURE 7.11 Effect of MD under L_LAX and different system loads

Hence, according to these results, it can be concluded that the effect of MD on the system's performance becomes more significant as the load of the system becomes heavier, and also, this effect is more explicit on star system model than it is on fully connected one.

7.7.3. Effect of System's Communication Network Topology

In order to observe the effect of system's network topology on the performance of the algorithm FB, the algorithm is tested on both of fully connected and star system models, under light, moderate, and heavy system loads (L_LOAD : R=8/600, M_LOAD : R=16/600, and H_LOAD : R=24/600), for each of the three different laxity distributions of tasks (L_LAX : $N(300,150^2)$, M_LAX : $N(450,150^2)$, and H_LAX : $N(600,150^2)$). The results obtained are as shown in Figure 7.12. During these simulation studies, no conflict message delay, MD, value of the system is taken to be 36 time units.

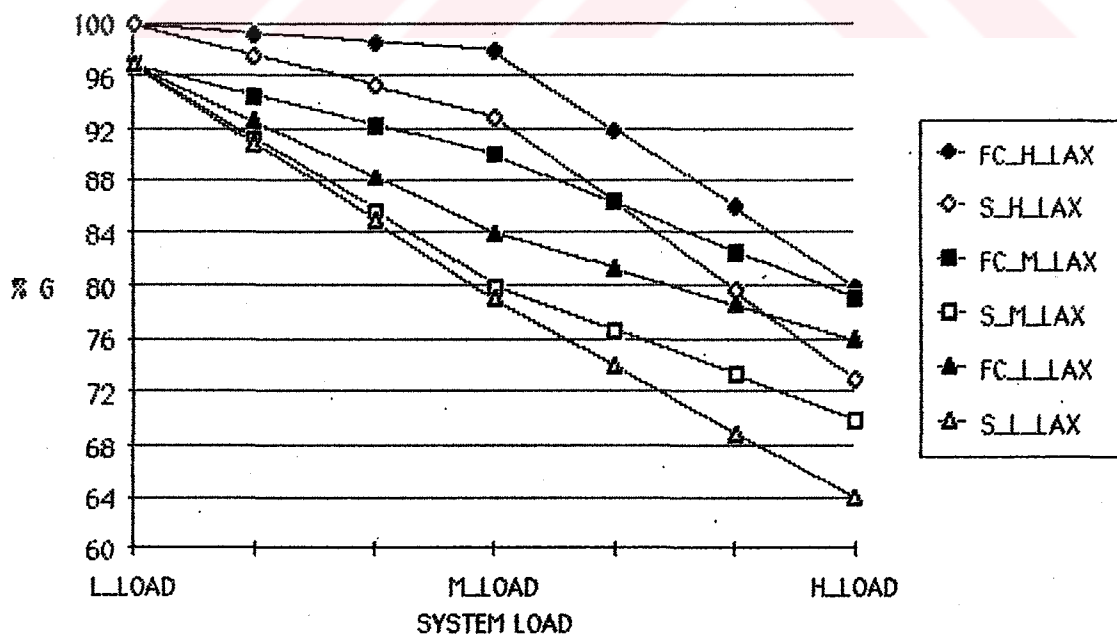


FIGURE 7.12 Effect of system's communication network topology

From the simulation studies, it is observed that when the system load is light, the performance of the algorithm FB is the same on both of the topologies, for each of the cases. But, when the system load is moderate, the difference between the performance of fully connected system model and that of the star system model is five per cent at H_LAX, 10 per cent at M_LAX, and five per cent at L_LAX. Further, when the system load is heavy, this difference is seven per cent at H_LAX, nine per cent at M_LAX, and 12 per cent at L_LAX. Hence, as the system load becomes heavier, and tasks' deadlines become more tight, the algorithm FB performs better on fully connected topology than it does on star topology.

It should also be added that, as mentioned in Section 7.7.2, the performance of the algorithm FB, on fully connected topology, is not as sensitive to communication overheads as it is on star topology (see Figure 7.11).

7.7.4. Comparison of Algorithm FB with Algorithms NC and R

In order to compare the performance of the algorithm FB to the performances of the noncooperative scheduling algorithm, NC, and of the random scheduling algorithm, R, three cases with different task laxity distributions (L_LAX : $N(300,150^2)$, M_LAX : $N(450,150^2)$, and H_LAX : $N(600,150^2)$) are tested. The results are shown in Figures 7.13 through 7.18. In each case, the performances of the algorithms NC, R, and FB, are observed under light, moderate, and heavy system loads (L_LOAD : $R=8/600$, M_LOAD : $R=16/600$, and H_LOAD : $R=24/600$). During these simulation studies, the system's message delay, MD, is taken to be 26 time units. The performances of the algorithms FB and R are evaluated on both fully connected (Figures 7.13, 7.15, 7.17) and star (Figures 7.14, 7.16, 7.18) system models.

As seen from the figures, in most cases the performance of the algorithm FB is much better than the lower bound offered by the algorithm NC. The percentage of guaranteed tasks of the algorithm FB is higher than that of the algorithm NC, by an amount between five and 24 per cent on star system model, and by an amount between five and 27 per cent on fully connected one. This proves the fact that distributed scheduling improves the performance of a hard real-time system.

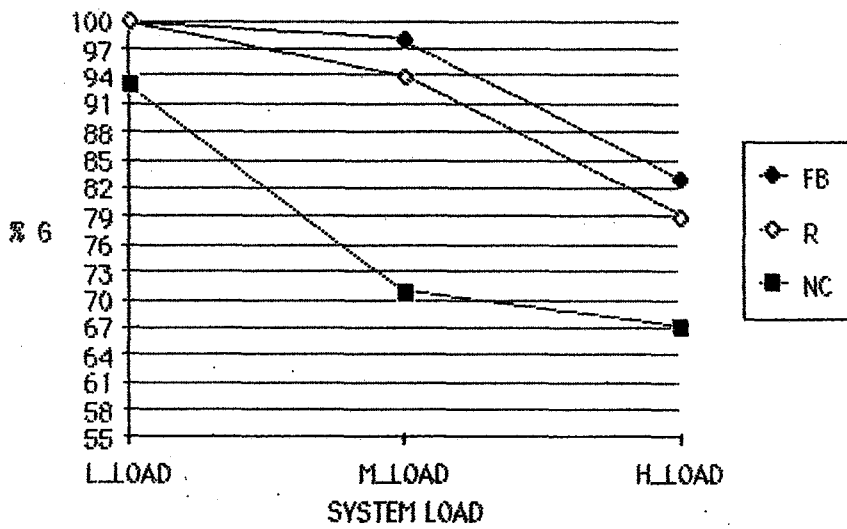


FIGURE 7.13 Comparison of FB, R, and NC when Laxity=H_LAX and Topology=FC

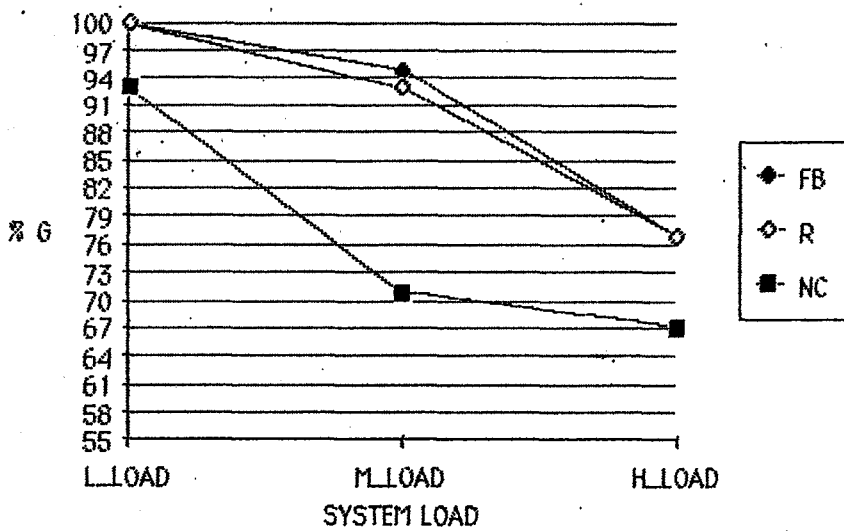


FIGURE 7.14 Comparison of FB, R, and NC when Laxity=H_LAX and Topology=S

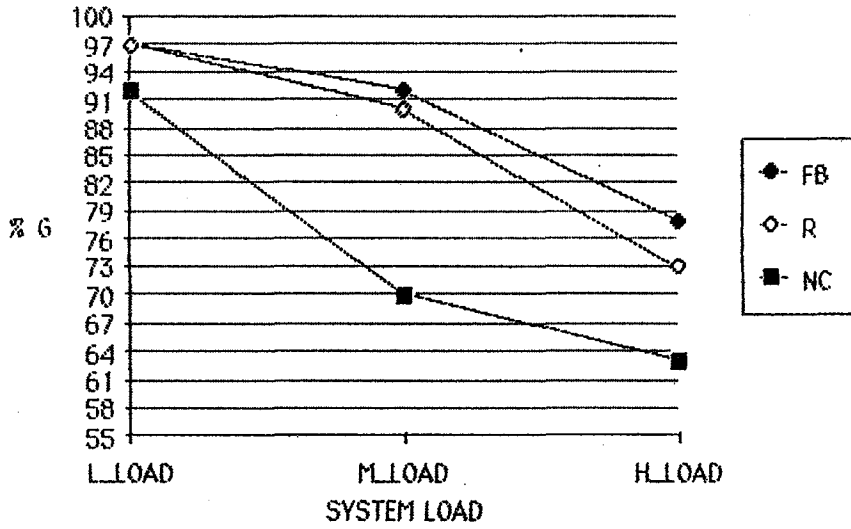


FIGURE 7.15 Comparison of FB, R, and NC when Laxity=M_LAX and Topology=FC

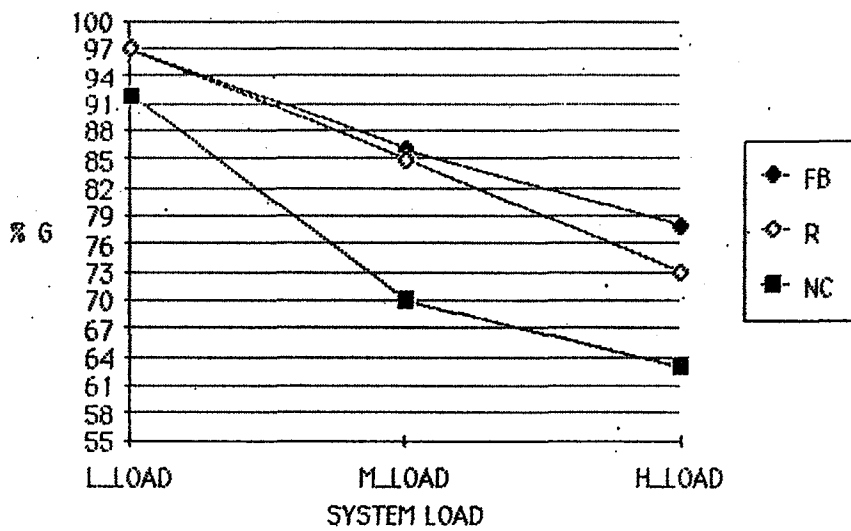


FIGURE 7.16 Comparison of FB, R, and NC when Laxity=M_LAX and Topology=S

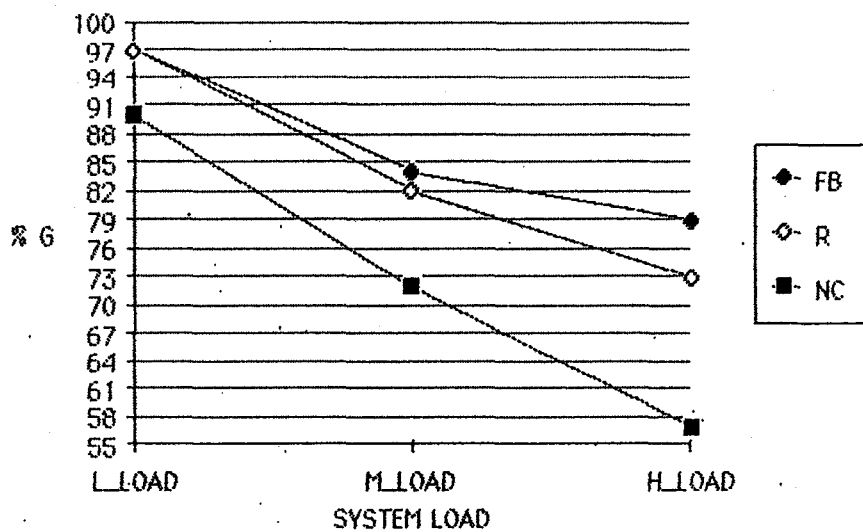


FIGURE 7.17 Comparison of FB, R, and NC when Laxity=L_LAX and Topology=FC

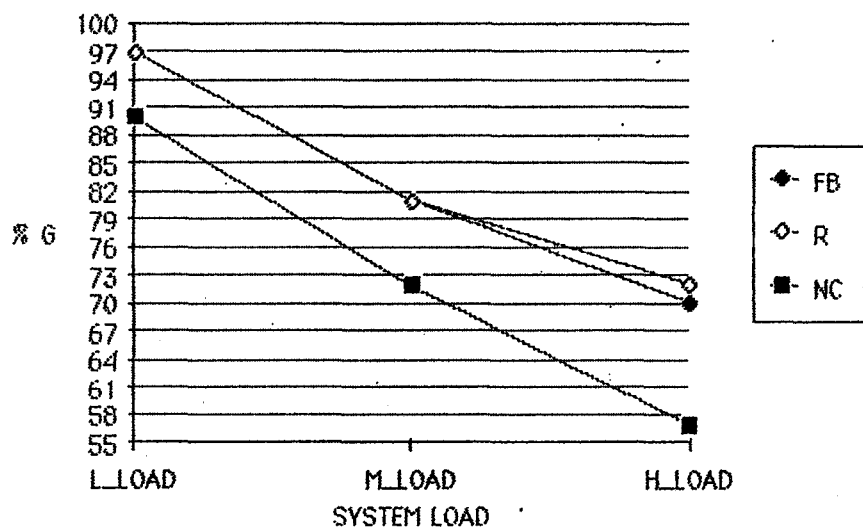


FIGURE 7.18 Comparison of FB, R, and NC when Laxity=L_LAX and Topology=S

In all cases, on fully connected topology, the performance of the algorithm FB is better than that of the algorithm R. This is expected in most cases, since the decisions about to which node to send the tasks locally nonguaranteed, are made by using the network wide surplus information in the algorithm FB, whereas in the algorithm R, they are made randomly.

Except one case, the performance of the algorithm FB is better on star topology as well. But, at the point where the system load is heavy (H_LOAD) and tasks' deadlines are tight (L_LAX), the performance of the algorithm R is observed to be higher than that of the algorithm FB, by an amount of two per cent. Since the algorithm FB requires much more communication than the algorithm R, when the system load is heavy, the non-negligible message delay MD, which was taken to be 26 time units, results in a performance lower than that of the algorithm R.

When the system load is light, no performance difference is observed between the algorithms FB and R. This reflects the fact that when the load is light, most of the nodes have enough surplus so that any node selected randomly is as good as any other node.

7.7.5. Comparison of Algorithm FB with Algorithm B

In order to compare the performance of the algorithm FB which combines focused addressing and bidding, with that of the algorithm B which uses bidding only, a set of simulation studies is performed. First, under moderate system load (M_LOAD : $R=16/600$), the performance of the algorithms is evaluated with different no conflict message delay values, for each of the three different laxity distributions (L_LAX : $N(300,150^2)$, M_LAX : $N(450,150^2)$, and H_LAX : $N(600,150^2)$), on both of fully connected and star network topologies. Figures 7.19 and 7.20 show the simulation results for fully connected and star system models respectively. Supported by these results, it is easily concluded that the algorithm FB performs better than B.

As seen from Figures 7.19 and 7.20, the communication overhead does have an explicit effect on the performance of both of the algorithms.

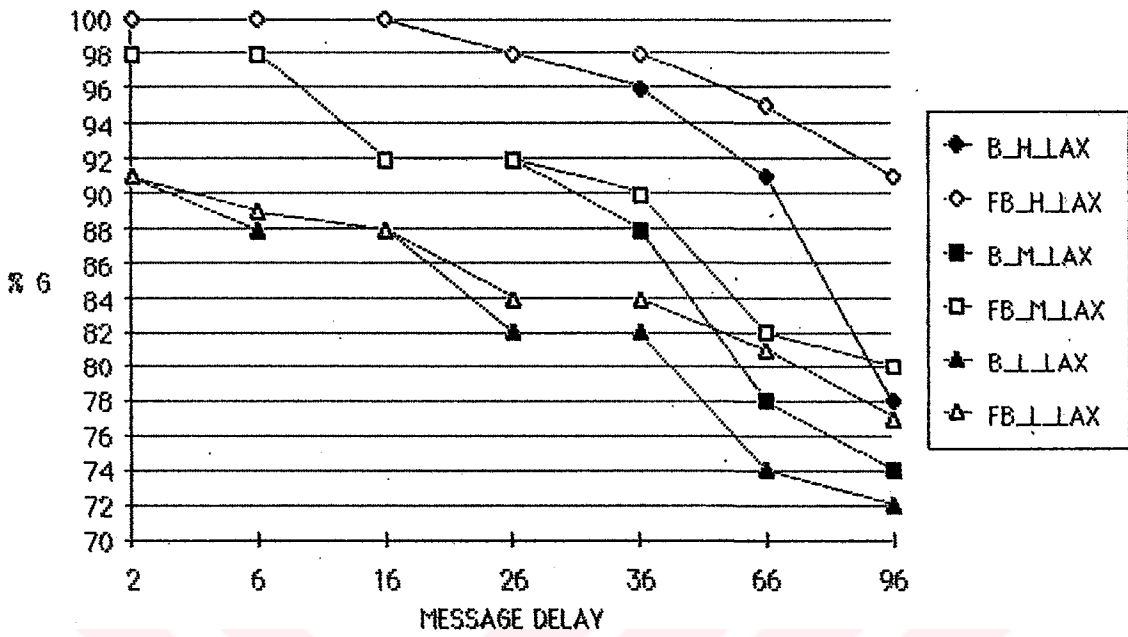


FIGURE 7.19 Comparison of FB and B when R=16/600 and Topology=FC

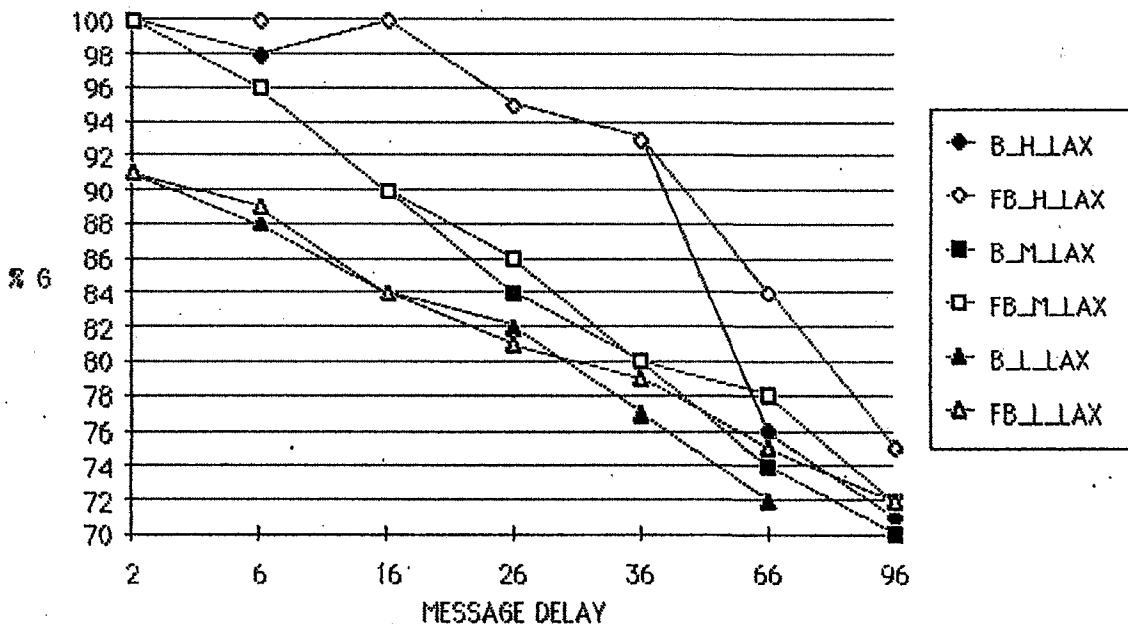


FIGURE 7.20 Comparison of FB and B when R=16/600 and Topology=S

When MD is small, the performance of the algorithm B is close to that of the algorithm FB. As MD increases, the difference between the performances of the two schemes increases. For example, on fully connected system model, although the performances of the two algorithms are the same at MD=16 time units, at MD=96 time units, the percentage of guaranteed tasks by the algorithm FB is higher than that of the algorithm B, by an amount of 13 per cent at H_LAX, six per cent at M_LAX, and five per cent at L_LAX. This difference is not so significant on star system model, because in this model, the performance of the algorithm FB also decreases explicitly at MD=96 time units.

Hence, it is observed that the performance of the algorithm FB is not as sensitive to MD as that of the algorithm B. This is because bidding always requires more message traffic. Also, the overhead of processing the request-for-bid messages and bids, affects the performance of the algorithm B at high MD values. Although the algorithm FB also uses bidding scheme, it has the advantage of being able to send a locally nonguaranteed task immediately to a focused node, using network wide surplus information of the previous window. This feature prevents the algorithm FB from decreasing in performance as much as the algorithm B does, at high communication delays.

As a result, it is concluded that the algorithm FB, compared to the algorithm B, performs well over a large range of no conflict message delays.

Further, the performances of these two algorithms are compared under heavy, moderate, and light system loads (L_LOAD : R=8/600, M_LOAD : R=16/600, and H_LOAD : R=24/600), with low, medium, and high laxity distributions of tasks, at a constant no conflict message delay which is taken to be 36 time units. The results obtained are presented in Figure 7.21 for fully connected system model, and in Figure 7.22 for star system model. These observations show that the algorithm FB performs better than the algorithm B under different system loads and tasks' laxities, on both of the network topologies.

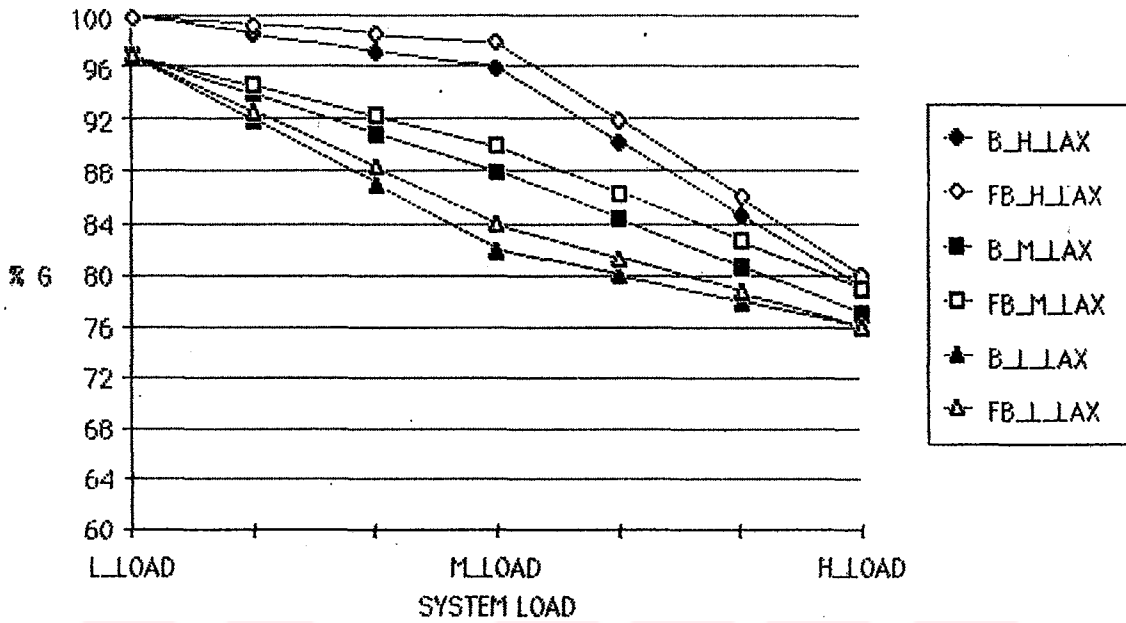


FIGURE 7.21 Comparison of FB and B when MD=36 and Topology=FC

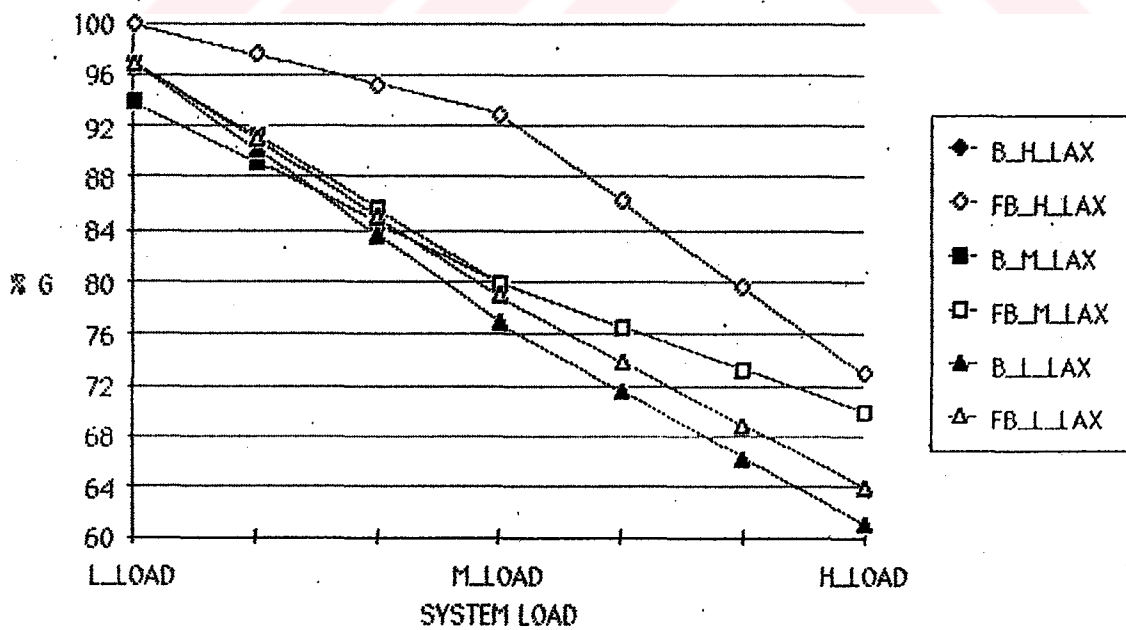


FIGURE 7.22 Comparison of FB and B when MD=36 and Topology=S

VIII. CONCLUSION

In a hard real-time processing or control environment, each task must be completed within a specified amount of time after being requested. If any task fails to complete in time, the entire system fails. Hence, one of the most important steps in designing a real-time computer system is to supply it with an efficient task scheduler. In a real-time context, efficiency is essential both for achieving the best use of the computer and for adhering with severe timing constraints relating to task executions.

Considerable research effort has been contributed to the subject of scheduling algorithms for hard real-time systems for decades. However, for most applications, the problem is often hard. For most cases, the problem of determining a static optimal schedule for a multiprocessor system is known to be NP-Complete. The problem is further complicated when dynamic distributed systems are dealt with, in which tasks can arrive dynamically at any nodes and the communication delay among the nodes is inherent and non-negligible.

In this thesis, the problem of dynamic scheduling of hard real-time tasks with resource requirements in distributed computer systems is considered. A heuristic scheduling approach for solving the problem is studied. Needless to mention, since heuristics are built into the algorithm, it is not optimal. Heuristics are necessary given the computationally hard nature of the scheduling problem. An optimal algorithm, in the worst case, may make an exhaustive search which is computationally intractable. In order to make the algorithm computationally tractable even in the worst case, a heuristic approach has to be taken. That is, on each level of the search, a heuristic function is applied to each of the tasks that remain to be scheduled. The task with the minimum value of this heuristic function is selected to extend the current schedule. Therefore, even in the worst case, the algorithm is not exponential.

The simulation studies performed on this algorithm, in Part VI, with different sets of tasks indicate that combination of simple heuristics with small number of backtracks performs very close to the optimal algorithm that uses exhaustive search. Hence, this is an attractive approach to on-line scheduling in dynamic real-time systems.

The heuristic function is invoked $\sum_{i=1, \dots, k} i$ ($i=1, \dots, k$, k being the size of the task set) times, resulting in a time complexity of k^2 . Pseudo backtracks do not increase the computational complexity. Moreover, the computation cost increased by real backtracks cannot effect the total complexity, so long as the upper bound of real backtracks is pre-set to less than k^2 . The time complexity k^2 of the algorithm is fairly low compared to that of an exhaustive search algorithm which takes time proportional to $k!$. The other features of the algorithm are that it takes both tasks' active and passive resource requirements into account, is dynamic, and is distributed.

Of course, there is the question of cost versus performance of the heuristics proposed in Part VI. The improved performance that results from the use of complex mechanisms, such as backtracking, may be offset by the computational overheads introduced by such mechanisms. Such overheads may be tolerated if a separate specially designed coprocessor is used for scheduling. In case such a processor is not used, one should use the simplest algorithm appropriate for the application under consideration.

This introduces the issue of selection of heuristics appropriate for a given situation. For example, as observed in the course of discussions of the simulation results, in Part VI, simple (single) heuristics may suffice if the deadlines of tasks being scheduled are not very tight.

The cooperation among the nodes, needed when a node is unable to guarantee a task, occurs through a combined scheme of bidding and focused addressing as explained in Part VII. It should be pointed out that bidding and focused addressing techniques are refined forms of the traditional source-initiated and server-initiated scheduling techniques. The combined scheme functions satisfactorily in spite of imprecise and incomplete global state information of the system.

The results of the simulation studies show that in a wide range of application environments (defined by task characteristics, system loads, communication delays, system network topologies, etc.), this scheme is effective and practical, and has a performance better than the other three algorithms that it is compared with: bidding only, random scheduling, and noncooperative scheduling algorithms. It is observed that the system performance improves when bidding is used in conjunction with focused addressing. In fact, focused addressing is an intelligent form of random scheduling in that it takes into account node's surplus information in choosing a node to send a task. By using a scheme that incorporates focused addressing and bidding, the benefits of both schemes are reaped.

APPENDIX A. LOCAL TASK GENERATOR

```

program task_generation;
{This program using the preset generating parameters, generates a number of
task sets so that each of them has at least one full feasible schedule.
Those schedulable task sets will be used as input data by the Local
Scheduler Program which checks the performance of various heuristics.}

const
  number_of_task_sets = 200;           {number of task sets to be generated}
  number_of_tasks = 6;                 {number of tasks in one task set}
  mu_compt = 200;
  sig_compt = 100;                     {mean and standard deviation of tasks' computation time}
  mu_laxity = 400;
  sig_laxity = 200; {mean and standard deviation of tasks' laxity distribution}
  r = 5;                               {number of resources on a node}
  rr = 7;

type
  taskset = array[1..number_of_tasks,1..rr] of integer;

var
  Tset : taskset;                      {contains specifications of tasks in a task set}
  count : integer;
  schedulable : boolean;
  tasksfile : text;

procedure generate_task_set(var T:taskset);
{create a task set, by generating task specifications for each task in it}
var
  counter,i,j : integer;
  n : real;
begin
  for i:=1 to number_of_tasks do
    for j:=1 to r+2 do T[i,j]:=0;           {initialize task set}
    counter:=1;                             {generate computation times for each task}
    repeat
      n:=0;
      for i:=1 to 12 do n:=n+RANDOM;
      n:=sig_compt*(n-6)+mu_compt;
      if trunc(n)>0
        then begin
          T[counter,r+1]:=trunc(n);
          counter:=counter+1
        end
    until counter=number_of_tasks+1;
  counter:=1;                               {generate laxities and calculate deadlines using them}
  repeat
    n:=0;
    for i:=1 to 12 do n:=n+RANDOM;
    n:=sig_laxity*(n-6)+mu_laxity;
    if trunc(n)>0

```

```

then begin
    T[counter,r+2]:=trunc(n)+T[counter,r+1];
    counter:=counter+1
end
until counter=number_of_tasks+1;
for i:=1 to number_of_tasks do
    repeat
        for j:=1 to r do
            if RANDOM<=0.5
            then T[i,j]:=1
            else T[i,j]:=0
        until ((T[i,1]<>0) or (T[i,2]<>0))
    end; {generate_task_set}
    {generate resource requirements}
end;

```

```

procedure exhaustive_search(T:taskset; var schedulable: boolean);
{perform an exhaustive search which checks all the possible permutations
of the task set, one by one, until a full feasible schedule is found. If there
is not any full feasible schedule, the task set is nonschedulable.}
label stop;

```

```

type
    eat_array = array[1..r] of integer;
var
    i,j,k,l,m,n : integer;
    EAT,EATi,EATj,EATk,EATl,EATm : eat_array;
    passdeadline : boolean;

```

```

procedure init;

```

```

var
    z : 1..r;
begin
    for z:=1 to r do
        begin
            EATi[z]:=0;
            EATj[z]:=0;
            EATk[z]:=0;
            EATl[z]:=0;
            EATm[z]:=0
        end
    end; {init}

```

```

procedure schedule(T:taskset;tt:integer;var EAT:eat_array;var pass:boolean);

```

```

var
    max,min,z,start_t : integer;
    New_EAT : eat_array;
begin
    max:=0;
    pass:=false;
    for z:=1 to r do New_EAT[z]:=0;
    for z:=1 to r do
        if T[tt,z]<>0
        then if EAT[z]>max
            then max:=EAT[z];
        start_t:=max;
        min:=9999;
        for z:=1 to r do
            begin
                if T[tt,z]<>0
                then begin
                    New_EAT[z]:=start_t+T[tt,r+1];
                    if New_EAT[z]>T[tt,r+2] then pass:=true
                    end
                else New_EAT[z]:=EAT[z];
            end
        end;

```


APPENDIX B. LOCAL SCHEDULING PROGRAM

program local_scheduling;
 {This program, given a number of schedulable task sets, determines the performance of the local scheduling algorithm. For each of the simple and integrated heuristic functions H, the number of task sets scheduled by the algorithm is calculated.}

```
const
  MC = 3;                {max counter used in real backtracking}
  r = 5;                {number of resources on each node}
  rr = 7;
  number_of_task_sets = 200; {number of task sets to be processed}
  number_of_tasks = 6;      {number of tasks in one task set}
  number_of_sim_types = 6; {number of different heuristic functions}
```

```
type
  resource_ar = array [1..r] of integer;
  real_ar = array [1..r] of real;
  bool_ar = packed array [1..r] of boolean;
  Old_EATptrtype = ^Old_EATtype;
  Old_EATtype = record
    Old_EAT : resource_ar;
    link : Old_EATptrtype;
  end;
  nodeptr = ^nodetype;
  secminptr = ^secminptrtype;
  secminptrtype = record
    secminnode : nodeptr;
    nextsecmin : secminptr;
  end;
  nodetype = record
    id : zstring[2];
    arr_t, deadline, start_t, comp_t : integer;
    res_need : bool_ar;
    secmin : secminptr;
    prev, next : nodeptr;
    New_EAT : resource_ar;
    Old_EATptr : Old_EATptrtype;
  end;
  st_type = 1..number_of_sim_types;
  task_sets_range = 0..number_of_task_sets;
```

```
var
  try : 0..5;
  guaranteed_task_sets, previous_value : task_sets_range;
  active : bool_ar;
  noincrease : boolean;
  sim_type : st_type;
  W : real;
```

```
procedure init;
```

```

begin
    active[1]:=true;active[2]:=true;           (indicate active and passive resources)
    active[3]:=false;active[4]:=false;active[5]:=false;
end; {init}

```

```

procedure get_tasks(var infile:text;var first_task_ptr:nodeptr);
{read task_set's specifications from the input file,and create a task queue}
var

```

```

    ii,i : integer;
    p,q : nodeptr;
    rn : array [1..rr] of integer;
begin
    new(q);
    q^.prev:=nil;q^.secmin:=nil;
    q^.Old_EATptr:=nil;first_task_ptr:=q;
    for ii:=1 to number_of_tasks do
        begin
            new(p);
            with p^ do
                begin
                    str(ii:2,id);start_t:=0;
                    next:=nil;prev:=q;secmin:=nil;
                    Old_EATptr:=nil;
                    for i:=1 to rr do read(infile,rn[i]);
                    readln(infile);
                    deadline:=rn[i]+21;
                    comp_t:=rn[i+1];
                    for i:=1 to r do
                        if rn[i]<> 0 then res_need[i]:=true
                            else res_need[i]:=false;
                    for i:=1 to r do New_EAT[i]:=0
                end;
                q^.next:=p;q:=p
            end
        end;
end; {get_tasks}

```

```

procedure delete_queue(var fn:nodeptr);
var
    n,pn : nodeptr;
begin
    pn:=fn;n:=pn^.next;
    repeat
        dispose(pn);
        pn:=n;
        if n<>nil then n:=n^.next
    until pn=nil
end; {delete_queue}

```

```

procedure scheduler(sim_type:st_type;var guaranteed:boolean;ll:real;
    var first_task_ptr:nodeptr; EAT:resource_ar);

```

```

var
    empty,passdeadline : boolean;
    schedule,s,f,ptr,ss : nodeptr;
    temp_ptr : secmin_ptr;
    DRDR : real_ar;
    counter : integer;

```

```

procedure calculate_ST;
{calculate the start time of the task if it is scheduled next}

```

```

var
    max,i : integer;
    p : nodeptr;

```

```

begin
  p:=first_task_ptr^.next;
  while p<>nil do
    begin
      max:=0;
      for i:=1 to r do
        if p^.res_need[i] then if EAT[i]>=max then max:=EAT[i];
      p^.start_t:=max;
      p:=p^.next
    end
  end; (calculate_ST)

procedure calculate_DRDR(var DRDR:real_ar);
{calculate Dynamic Resource Demand Ratio, which indicates the degree to which
tasks that remain to be scheduled will demand resources}
var
  fraction,tot_comp_t,max,i : integer;
  p : nodeptr;
begin
  for i:=1 to r do
    begin
      tot_comp_t:=0;
      max:=0;
      p:=first_task_ptr^.next;
      while p<>nil do
        begin
          if p^.res_need[i]
            then begin
              tot_comp_t:=tot_comp_t+p^.comp_t;
              if p^.deadline>=max then max:=p^.deadline
            end;
          p:=p^.next
        end;
      fraction:=max-EAT[i];
      if fraction=0 then DRDR[i]:=0
        else if max=0 then DRDR[i]:=0
          else DRDR[i]:=tot_comp_t/fraction
    end
  end; (calculate_DRDR)

function strongly_feasible(DRDR:real_ar):boolean;
var
  i : integer;
begin
  strongly_feasible:=true;
  for i:=1 to r do if DRDR[i]>=1 then strongly_feasible:=false;
  if passdeadline then strongly_feasible:=false
end; {strongly_feasible}

procedure calculate_New_EAT;
{calculate the EAT values of the task if it is scheduled next}
var
  min,i : integer;
  p : nodeptr;
begin
  p:=first_task_ptr^.next;
  passdeadline:=false;
  while (p<>nil) and (not passdeadline) do
    begin
      min:=maxint;
      for i:=1 to r do
        begin

```

```

    if p^.res_need[i] then begin
        p^.New_EAT[i]:=p^.start_t+p^.comp_t;
        if p^.New_EAT[i]>p^.deadline
            then passdeadline:=true
            end
        else p^.New_EAT[i]:=EAT[i];
    if active[i] then if p^.New_EAT[i]<=min
        then min:=p^.New_EAT[i]
    end;
    for i:=1 to r do
        if p^.New_EAT[i]<min then p^.New_EAT[i]:=min;
    p:=p^.next
    end
end; {calculate_New_EAT}

procedure calculate_min_H(var ptr:nodeptr;sim_type:st_type;H:real);
{detect which tasks, among the tasks that remain to be scheduled, have minimum
and second minimum values of the function H}
var
    temp,i : integer;
    secondmin,min,H : real;
    secpointer,q : nodeptr;
    sec : secminptr;
    p,pp : Old_EATptrtype;
begin
    min:=maxint; secondmin:=maxint;
    ptr:=nil; q:=first_task_ptr^.next;
    while q<>nil do
        begin
            case sim_type of
                1 : H:=q^.deadline;
                2 : H:=q^.start_t;
                3 : H:=q^.comp_t;
                4 : H:=q^.deadline-(q^.start_t+q^.comp_t);
                5 : H:=q^.deadline+H*q^.comp_t;
                6 : H:=q^.deadline+H*q^.start_t
            end;
            if H<=min then begin
                secondmin:=min;
                min:=H;
                secpointer:=ptr;
                ptr:=q
            end
            else if H<=secondmin then begin
                secondmin:=H;
                secpointer:=q
            end;
            q:=q^.next
        end;
    if secpointer <> nil
        then begin
            new(sec);
            sec^.secminnode:=secpointer;
            sec^.nextsecmin:=ptr^.secmin;
            ptr^.secmin:=sec;
            new(p);
            for i:=1 to r do p^.Old_EAT[i]:=secpointer^.New_EAT[i];
            pp:=secpointer^.Old_EATptr;
            p^.link:=pp;
            secpointer^.Old_EATptr:=p
        end
    end; {calculate_min_H}

```



```

procedure update_EAT(pointer:nodeptr);
{replace EAT values by New_EAT values of the task just scheduled}
var
  i : integer;
begin
  for i:=1 to r do EAT[i]:=pointer^.New_EAT[i];
end; {update_EAT}

procedure delete_from_task_set(pointer:nodeptr);
{remove the task to be scheduled from the task queue}
begin
  pointer^.prev^.next:=pointer^.next;
  if pointer^.next<>nil then begin
    pointer^.next^.prev:=pointer^.prev;
    pointer^.next:=nil
  end;

  pointer^.prev:=nil
end; {delete_from_task_set}

procedure add_to_schedule(var s:nodeptr;pointer:nodeptr);
begin
  pointer^.prev:=s;      {add the task to be scheduled to the schedule queue}
  s^.next:=pointer;
  s:=pointer
end; {add_to_schedule}

procedure delete_from_schedule(var s,pointer:nodeptr);
begin
  pointer:=s;      {remove the last task scheduled from the schedule queue}
  s:=pointer^.prev;
  pointer^.prev:=nil;
  s^.next:=nil
end; {delete_from_schedule}

procedure put_back_to_task_set(pointer:nodeptr);
{add the task, removed from schedule queue, to task queue}
var
  p : nodeptr;
begin
  p:=first_task_ptr^.next;
  p^.prev:=pointer; pointer^.next:=p;
  first_task_ptr^.next:=pointer;
  pointer^.prev:=first_task_ptr
end; {put_back_to_task_set}

procedure get_Old_EAT(pointer:nodeptr);
{since there is a backtrack attempting to schedule the task which has second
minimum value of the function H, EAT values should be replaced by New_EAT of
this task which was recorded as Old_EAT}
var
  i : integer;
  p : Old_EATptrtype;
begin
  p:=pointer^.Old_EATptr;
  for i:=1 to r do EAT[i]:=p^.Old_EAT[i];
  pointer^.Old_EATptr:=p^.link;
  dispose(p); p:=nil;
end; {get_Old_EAT}

procedure schedule_second_minimum;
{schedule the task which has second minimum value of function H}

```

```

begin
  tempptr:=ptr^.secmin;
  ptr^.secmin:=tempptr^.nextsecmin;
  ptr:=tempptr^.sacminnode;
  dispose(tempptr); tempptr:=nil;
  delete_from_task_set(ptr);
  add_to_schedule(s,ptr);
  get_Old_EAT(ptr);
  calculate_ST;
  calculate_New_EAT;
  calculate_DRDR(DRDR)
end; {schedule_second_minimum}

procedure limited_backtracker(var guaranteed:boolean);
begin
  if schedule^.next=nil
  then guaranteed:=false
  else begin
    {perform Pseudo Backtrack}
    delete_from_schedule(s,ptr);
    put_back_to_task_set(ptr);
    schedule_second_minimum;
    if not strongly_feasible(DRDR)
    then begin
      {perform Real Backtrack}
      guaranteed:=false;
      empty:=false;
      while (not empty) and (counter<MC)
        and (not guaranteed) do
      begin
        repeat
          delete_from_schedule(s,ptr);
          put_back_to_task_set(ptr);
        until (ptr^.secmin<>nil) or (s=schedule);
        if ptr^.secmin<>nil
        then begin
          schedule_second_minimum;
          if strongly_feasible(DRDR)
          then guaranteed:=true;
        end
        else empty:=true;
        counter:=counter+1
      end {while}
    end
  end
end; {limited_backtracker}

begin {scheduler}
  new(s); s^.next:=nil; s^.prev:=nil;
  s^.secmin:=nil; schedule:=s; counter:=0; guaranteed:=true;
  empty:=false; f:=first_task_ptr;
  while (f^.next<>nil) and guaranteed do
  begin
    calculate_ST;
    calculate_New_EAT;
    calculate_DRDR(DRDR);
    if not strongly_feasible(DRDR)
    then limited_backtracker(guaranteed)
    else begin
      calculate_min_H(ptr,sim_type,W);
      update_EAT(ptr);
      delete_from_task_set(ptr);
      add_to_schedule(s,ptr)
    end
  end
end

```

```

end; {while}
delete_queue(schedule)
end; {scheduler}

```

```

procedure check_task_sets(var guaranteed_task_sets:task_sets_range);
var

```

```

  i : integer;
  infile : text;
  EAT : resource_ar;
  guaranteed : boolean;
  first_task_ptr : nodeptr;
  count_task_sets : task_sets_range;
begin
  assign(infile,'tasks.dat'); reset(infile);
  guaranteed_task_sets:=0; count_task_sets:=0;
  repeat
    for i:=1 to r do EAT[i]:=0;
    get_tasks(infile,first_task_ptr);
    scheduler(sim_type,guaranteed,W,first_task_ptr,EAT);
    if guaranteed
      then guaranteed_task_sets:=guaranteed_task_sets+1
      else delete_queue(first_task_ptr);
    count_task_sets:=count_task_sets+1
  until count_task_sets=number_of_task_sets;
  close(infile)
end; {check_task_sets}

```

```

begin {main}
  W:=0; init;
  for sim_type:=1 to number_of_sim_types do
    if sim_type<5
      then begin
        check_task_sets(guaranteed_task_sets);
        writeln;
        writeln('ST = ',sim_type:1,' NUMBER OF GUARANTEED TASK SETS = ',
          guaranteed_task_sets:3)
      end
    else begin
      try:=1; W:=0.5;
      previous_value:=0;
      noincrease:=false;
      repeat
        check_task_sets(guaranteed_task_sets);
        writeln;
        writeln('ST = ',sim_type:1,' W = ',W:3:1,
          ' NUMBER OF GUARANTEED TASK SETS = ',guaranteed_task_sets:3);
        if (guaranteed_task_sets=number_of_task_sets)
          or (guaranteed_task_sets<previous_value)
          then noincrease:=true
        else begin
          W:=W+0.5;
          if guaranteed_task_sets=previous_value
            then try:=try+1
            else try:=1;
          if try>4 then noincrease:=true;
          previous_value:=guaranteed_task_sets
        end
      until noincrease
    end
  end. {main}

```

APPENDIX C. GLOBAL TASK GENERATOR

program gen_task_queues;
 {This program, given the preset generating parameters, generates tasks that arrive locally to each node of the system model, and creates task queues which will be used as input data by the simulation programs which study the performance of various distributed scheduling algorithms.}

```

type
  string_type = string[51];

var
  tasksfile : text;

procedure gen_tasks(ch:char;sig_compt,mu_compt,sig_laxity,
  mu_laxity:integer;lambdarrt:real);

const
  r = 5;
  rr = 8;
  SIM_TIME = 2500;
type
  tasktype = array [1..rr] of integer;
var
  T : tasktype;
  counter,i,arr_t : integer;
  n : real;
  stop : boolean;

begin
  for i:=1 to rr do T[i]:=0;
  counter:=1;
  arr_t:=0;
  stop:=false;
  repeat
    T[r+3]:=arr_t;
    repeat
      n:=0;
      for i:=1 to 12 do n:=n+RANDOM;
      n:=sig_compt*(n-6)+mu_compt
    until trunc(n)>>0;
    T[r+1]:=trunc(n);
    {generate computation time}
    repeat
      n:=0;
      for i:=1 to 12 do n:=n+RANDOM;
      n:=sig_laxity*(n-6)+mu_laxity;
    until trunc(n)>>0;
    T[r+2]:=T[r+3]+T[r+1]+trunc(n);
    {generate deadline}
    repeat
      for i:=1 to r do
        if RANDOM<=0.5 then T[i]:=1
        else T[i]:=0
    until ((T[1]<>0) or (T[2]<>0));
    {generate resources requirements}
  
```

```

if T[r+2] > SIMTIME
  then stop:=true
  else begin
    for i:=1 to r+3 do write(tasksfile,T[i]:5);
    writein(tasksfile,' ',ch:1,counter);
    counter:=counter+1;
    arr_t:=trunc(arr_t-ln(RANDOM)/lambda_arrr) {generate arrival time}
  end
until stop
end; {gen_tasks}

procedure create_data_file(fl:string_type;ch:char;
  sig_c,mu_c,sig_l,mu_l:integer;
  lambda_arrr:real);
begin
  assign(tasksfile,fl);
  rewrite(tasksfile);
  gen_tasks(ch,sig_c,mu_c,sig_l,mu_l,lambda_arrr);
  close(tasksfile)
end; {create_data_file}

begin {main}
  randomize;
  {generate task queues for the nodes A through E}
  create_data_file('A.dat','A',100,200,150,300,1/100);
  create_data_file('B.dat','B',100,200,150,300,1/100);
  create_data_file('C.dat','C',100,200,150,300,1/300);
  create_data_file('D.dat','D',100,200,150,300,1/600);
  create_data_file('E.dat','E',100,200,150,300,1/600)
end. {main}

```

APPENDIX D. SCHEDULING PROGRAM USING BIDDING AND FOCUSED ADDRESSING

ADDRESSING

This part contains the listing of the distributed scheduling program which uses a technique which combines bidding and focused addressing schemes. First, the listings of the include files are given, the listing of the main program then follows.

Listing of Include File SCH.PAS :

```

procedure get_node(var pointer:nodeptr);
var
  i : integer;
begin
  new(pointer);
  with pointer do
    begin
      next:=nil;
      prev:=nil;
      secmin:=nil;
      Old_EATptr:=nil;
      arr_t:=0;
      deadline:=0;
      start_t:=0;
      comp_t:=0;
      for l:=1 to r do
        begin
          res_need[l]:=false;
          New_EAT[l]:=0
        end
      end
    end; {get_node}

procedure copy_info(newnode,oldnode:nodeptr);
begin
  newnode^.id:=oldnode^.id;
  newnode^.arr_t:=oldnode^.arr_t;
  newnode^.comp_t:=oldnode^.comp_t;
  newnode^.deadline:=oldnode^.deadline;
  newnode^.res_need:=oldnode^.res_need;
  newnode^.New_EAT:=oldnode^.New_EAT
end; {copy_info}

procedure get_tasks(var first_task:nodeptr);
var
  p,q : nodeptr;
  rn : array [1..rr] of integer;
  ch : char;

```

```

i : integer;
begin
  get_node(q);
  first_task:=q;
  while not eof(infile) do
    begin
      new(p);
      with p^ do
        begin
          start_t:=0;
          next:=nil;
          prev:=q;
          secmin:=nil;
          Old_EATptr:=nil;
          for i:=1 to rr do read(infile,rn[i]);
          read(infile,ch,id);
          readln(infile);
          arr_t:=rn[r+3];
          deadline:=rn[r+2];
          comp_t:=rn[r+1];
          for i:=1 to r do
            if rn[i]> 0 then res_need[i]:=true
              else res_need[i]:=false;
          for i:=1 to r do New_EAT[i]:=0
          end;
          q^.next:=p;
          q:=p
        end
      end;
    end;
  end; {get_tasks}

```

```

procedure scheduler(q,DQ:nodeptr;EAT:resource_ar; var guaranteed:boolean;
  var first_task_ptr,schedule:nodeptr;quantity:integer);
{check whether a newly arrived task can be guaranteed or not}
var

```

```

  passdeadline,empty : boolean;
  s,f,p,pp,ptr,ss : nodeptr;
  temptr : secminptr;
  DADR : real_ar;
  counter,i : integer;

```

```

procedure calculate_ST;

```

```

var
  max,i : integer;
  p : nodeptr;
begin
  p:=first_task_ptr^.next;
  while p<>nil do
    begin
      max:=p^.arr_t;
      for i:=1 to r do
        if p^.res_need[i]
          then if EAT[i]>max
            then max:=EAT[i];
      p^.start_t:=max;
      p:=p^.next
    end
  end; {calculate_ST}

```

```

procedure calculate_DADR(var DADR:real_ar);
{calculate the degree to which tasks to be scheduled will demand resources}
var
  fraction,tot_comp_t,max,i : integer;

```

```

p : nodeptr;
begin
  for i:=1 to r do
    begin
      tot_comp_t:=0;
      max:=0;
      p:=first_task_ptr^.next;
      while p<>nil do
        begin
          if p^.res_need[i]
            then begin
              tot_comp_t:=tot_comp_t+p^.comp_t;
              if p^.deadline>max
                then max:=p^.deadline
              end;
              p:=p^.next;
            end;
          fraction:=max-EAT[i];
          if fraction=0
            then DRDR[i]:=0
            else if max=0 then DRDR[i]:=0
            else DRDR[i]:=tot_comp_t/fraction
          end
        end;
      {calculate_DRDR}

function strongly_feasible(DRDR:real_ar):boolean;
var
  i : integer;
begin
  strongly_feasible:=true;
  for i:=1 to r do
    if DRDR[i]>=1 then strongly_feasible:=false;
    if passdeadline then strongly_feasible:=false
  end; {strongly_feasible}

procedure calculate_New_EAT;
{calculate EAT values of the task if it is scheduled next}
var
  min,i : integer;
  p : nodeptr;
begin
  p:=first_task_ptr^.next;
  passdeadline:=false;
  while (p<>nil) and (not passdeadline) do
    begin
      min:=maxint;
      for i:=1 to r do
        begin
          if p^.res_need[i]
            then begin
              p^.New_EAT[i]:=p^.start_t+p^.comp_t;
              if p^.New_EAT[i]>p^.deadline
                then passdeadline:=true
              end
            else p^.New_EAT[i]:=EAT[i];
          if active[i]
            then if p^.New_EAT[i]<=min
              then min:=p^.New_EAT[i]
            end;
          for i:=1 to r do
            if p^.New_EAT[i]<min
              then p^.New_EAT[i]:=min;

```



```

    p:=p^.next
  end
end; {calculate_New_EAT}

procedure calculate_min_H(var ptr:nodeptr);
{detect the tasks having minimum and second minimum values of H}
const
  W = 0.5;
var
  temp,i : integer;
  secondmin,min,H : real;
  secpointer,q : nodeptr;
  sec : secminptr;
  p,pp : Old_EATptrtype;
begin
  min:=maxint;
  secondmin:=maxint;
  ptr:=nil;
  q:=first_task_ptr^.next;
  while q<>nil do
    begin
      H:=q^.deadline + W * q^.start_t;
      if H<=min
        then begin
          secondmin:=min;
          min:=H;
          secpointer:=ptr;
          ptr:=q
        end
        else if H<=secondmin
          then begin
            secondmin:=H;
            secpointer:=q
          end;
      q:=q^.next
    end;
  if secpointer <> nil
    then begin
      new(sec);
      sec^.secminnode:=secpointer;
      sec^.nextsecmin:=ptr^.secmin;
      ptr^.secmin:=sec;
      new(p);
      for i:=1 to r do
        p^.Old_EAT[i]:=secpointer^.New_EAT[i];
      pp:=secpointer^.Old_EATptr;
      p^.link:=pp;
      secpointer^.Old_EATptr:=p
    end
  end; {calculate_min_H}

procedure update_EAT(pointer:nodeptr);
var
  i : integer;
begin
  for i:=1 to r do EAT[i]:=pointer^.New_EAT[i];
end; {update_EAT}

procedure delete_from_task_set(pointer:nodeptr);
begin
  pointer^.prev^.next:=pointer^.next;
  if pointer^.next<>nil

```

```

    then begin
        pointer^.next^.prev:=pointer^.prev;
        pointer^.next:=nil
    end;
    pointer^.prev:=nil
end; {delete_from_task_set}

procedure add_to_schedule(var s:nodetype;pointer:nodetype);
begin
    pointer^.prev:=s;
    s^.next:=pointer;
    s:=pointer
end; {add_to_schedule}

procedure delete_from_schedule(var s,pointer:nodetype);
begin
    pointer:=s;
    s:=pointer^.prev;
    pointer^.prev:=nil;
    s^.next:=nil
end; {delete_from_schedule}

procedure put_back_to_task_set(pointer:nodetype);
var
    p : nodetype;
begin
    p:=first_task_ptr^.next;
    p^.prev:=pointer;
    pointer^.next:=p;
    first_task_ptr^.next:=pointer;
    pointer^.prev:=first_task_ptr
end; {put_back_to_task_set}

procedure get_Old_EAT(pointer:nodetype);
var
    p : Old_EATptrtype;
    i : integer;
begin
    p:=pointer^.Old_EATptr;
    for i:=1 to r do EAT[i]:=p^.Old_EAT[i];
    pointer^.Old_EATptr:=p^.link;
    dispose(p);
    p:=nil
end; {get_Old_EAT}

procedure initial_values;
begin
    counter:=0;
    empty:=false;
    guaranteed:=true;
    get_node(s);
    schedule:=s;
    get_node(f);
    first_task_ptr:=f
end; {initial_values}

procedure form_tasks_queue;
var
    i:integer;
begin
    p:=DQ^.next;
    while p<>nil do
        {copy dispatcher queue (DQ) to tasks queue}

```

```

begin
  get_node(pp);
  copy_info(pp,p);
  f^.next:=pp;
  pp^.prev:=f;
  f:=pp;
  p:=p^.next
end;
for i:=1 to quantity do      {add new task (or multiples of it) to tasks_queue}
begin
  get_node(pp);
  copy_info(pp,q);
  f^.next:=pp;
  pp^.prev:=f;
  f:=pp
end;
end; {form_tasks_queue}

procedure calculations;
begin
  calculate_ST;
  calculate_New_EAT;
  calculate_DRDR(DRDR)
end;

begin {scheduler}
  initial_values;
  form_tasks_queue;
  f:=first_task_ptr;
  while (f^.next<>nil) and guaranteed and (not empty) do
  begin
    calculations;
    if not strongly_feasible(DRDR)
    then if schedule^.next=nil
    then empty:=true
    else begin
      {Pseudo Backtrack}
      delete_from_schedule(s,ptr);
      put_back_to_task_set(ptr);
      temptr:=ptr^.secmin;
      ptr^.secmin:=temptr^.nextsecmin;
      ptr:=temptr^.secminnode;
      dispose(temptr); temptr:=nil;
      delete_from_task_set(ptr);
      add_to_schedule(s,ptr);
      get_Old_EAT(ptr);
      calculations;
      if not strongly_feasible(DRDR)
      then begin
        {Real Backtrack}
        guaranteed:=false;
        empty:=false;
        while (not empty) and (counter<MC) and
          (not guaranteed) do
        begin
          repeat
            delete_from_schedule(s,ptr);
            put_back_to_task_set(ptr)
          until (ptr^.secmin<>nil) or (s=schedule);
          if ptr^.secmin<>nil
          then begin
            temptr:=ptr^.secmin;
            ptr^.secmin:=temptr^.nextsecmin;
            ptr:=temptr^.secminnode;

```

```

dispose(temp_ptr);
temp_ptr:=nil;
delete_from_task_set(ptr);
add_to_schedule(s,ptr);
get_Old_EAT(ptr);
calculations;
if strongly_feasible(DRDR)
then guaranteed:=true;
end
else empty:=true;
counter:=counter+1
end(while)
end
end
else begin
calculate_min_H(ptr);
update_EAT(ptr);
delete_from_task_set(ptr);
add_to_schedule(s,ptr)
end
end;
if empty then guaranteed:=false
end; {scheduler}

```

Listing of Include File BID.PAS :

```

procedure indicate_line(nodeid,dest:char;var bid_arr,no:integer);
begin
detect_index(nodeid,dest,no);
case no of
1: bid_arr:=line4;
2: bid_arr:=line3;
3: bid_arr:=line2;
4: bid_arr:=line1
end
end; {indicate_line}

procedure calculate_MAXBID;
var
l,min_task_sending_delay,max,estimated_start_t : integer;
begin
min_task_sending_delay:=b^.comp_t div 10+message_delay;
{ calculate earliest estimated arrival time}
b^.arr_t:=clock+message_delay+min_task_sending_delay;
for i:=1 to r do
if real_EAT[i]<clock
then temp_EAT[i]:=clock
else temp_EAT[i]:=real_EAT[i];
max:=b^.arr_t;
for i:=1 to r do
if b^.res_need[i]
then if temp_EAT[i]>max then max:=temp_EAT[i];
estimated_start_t:=max;
MAXBID:=(b^.deadline-estimated_start_t) div b^.comp_t
end; {calculate_MAXBID}

procedure binary_search_for_BID;
var
low,hi,mid : integer;
guaranteed : boolean;

```

```

begin
  BID:=0;
  low:=1;
  hi:=MAXBID;
  while (low<=hi) do
    begin
      mid:=(low+hi) div 2;
      scheduler(b,DQ,temp_EAT,guaranteed,TQ2,SQ,mid);
      deletequeue(TQ2);
      TQ2^.next:=nil;
      deletequeue(SQ);
      SQ^.next:=nil;
      if guaranteed
        then begin
          BID:=mid;
          low:=mid+1
        end
        else hi:=mid-1
      end
    end; {binary_search_for_BID}

procedure bidding_C;
begin
  case no of
    1: send_bid(b^.id,IBQ_4,1,BID,line4);
    2: send_bid(b^.id,IBQ_3,2,BID,line3);
    3: send_bid(b^.id,IBQ_2,3,BID,line2);
    4: send_bid(b^.id,IBQ_1,4,BID,line1)
  end
end; {bidding_C}

procedure bidding_S;
var
  bn,pbn,bid_node : bid_nodeptr;
begin
  get_bid_node(bid_node);
  bid_node^.t_id:=b^.id;
  bid_node^.value:=BID;
  case no of
    1: begin
        bid_node^.ind:=1;
        if nodeid<>'A'
          then bid_node^.dest:=pred(nodeid)
          else bid_node^.dest:='E'
        end;
    2: begin
        bid_node^.ind:=2;
        if (nodeid='A') or (nodeid='B')
          then bid_node^.dest:=succ(succ(succ(nodeid)))
          else bid_node^.dest:=pred(pred(nodeid))
        end;
    3: begin
        bid_node^.ind:=3;
        if (nodeid='D') or (nodeid='E')
          then bid_node^.dest:=pred(pred(pred(nodeid)))
          else bid_node^.dest:=succ(succ(nodeid))
        end;
    4: begin
        bid_node^.ind:=4;
        if nodeid<>'E'
          then bid_node^.dest:=succ(nodeid)
          else bid_node^.dest:='A'
        end;
  end;
end;

```

```

    end
  end;
  update_line(message_delay, line1);
  bid_node^.arr:=line1;
  insert_into_S_BIDQ(bid_node);
end; {bidding_S}

```

Listing of Include File BIDEVAL.PAS :

```

procedure sending_task(var b:inc_bids_nodeptr;
  RFBQ,task_queue:nodeptr;ch:char;var line:integer);
var
  p : nodeptr;
  delay : integer;
begin
  find_the_task(b,RFBQ,p);
  if p<>nil
  then begin
    delay:=p^.comp_t div 10+message_delay;
    if TOPOLOGY='S'
    then p^.id:=ch+p^.id;
    send(p,task_queue,line,delay);
    delete_task(p)
  end
end; {sending_task}

procedure nonguaranteed(var b:inc_bids_nodeptr;RFBQ:nodeptr);
var
  p : nodeptr;
begin
  find_the_task(b,RFBQ,p);
  if p<>nil
  then begin
    tasks_nonguaranteed:=tasks_nonguaranteed+1;
    delete_task(p)
  end
end; {nonguaranteed}

procedure bid_evaluating(IBQ:inc_bids_nodeptr;
  RFBQ,New_TQ1,New_TQ2,New_TQ3,New_TQ4:nodeptr;
  ch1,ch2,ch3,ch4:char;
  var line1,line2,line3,line4:integer);
var
  okay : boolean;
  b : inc_bids_nodeptr;
  l,latest_bid_arr,max,no : integer;

procedure sender(no:integer);
begin
  case no of
    0: nonguaranteed(b,RFBQ);
    1: sending_task(b,RFBQ,New_TQ1,ch1,line1);
    2: sending_task(b,RFBQ,New_TQ2,ch2,line2);
    3: sending_task(b,RFBQ,New_TQ3,ch3,line3);
    4: sending_task(b,RFBQ,New_TQ4,ch4,line4)
  end
end; {sender}

begin {bid_evaluating}
  b:=IBQ^.next;
  {evaluate bids arrived at the Incoming Bids Queue}

```

```

while b<>nil do
  with b^ do
    begin
      if clock=latest_bid_arr
      then begin
        max:=0;
        no:=0;
        for i:=1 to 4 do
          begin
            if (arrs[i]<=clock) and (bids[i]>max)
            then begin
              max:=bids[i];
              no:=i
            end
          end;
        sender(no)
      end
    else begin
      okay:=true;
      for i:=1 to 4 do
        if (arrs[i]=0)or(arrs[i]>clock) then okay:=false;
      if okay
      then begin
        max:=0;
        no:=0;
        for i:=1 to 4 do
          begin
            if bids[i]>max
            then begin
              max:=bids[i];
              no:=i
            end
          end;
        sender(no)
      end
      else if (bids[1]>HB) and (arrs[1]<=clock)
      then sender(1)
      else if (bids[2]>HB) and (arrs[2]<=clock)
      then sender(2)
      else if (bids[3]>HB) and (arrs[3]<=clock)
      then sender(3)
      else if (bids[4]>HB) and (arrs[4]<=clock)
      then sender(4)
      else b:=b^.next
    end
  end
end; (bid_evaluating)

```

Listing of Include File FROMS.PAS :

```

procedure send_RFbMs_from_S;
var
  ch : char;
  q : nodeptr;
  stop : boolean;
begin
  stop:=false;  {transfer request for bid messages queued at the central node}
  repeat
    q:=S_RFbM^.next;
    if (q<>nil) and (q^.arr_t=clock)

```

```

then begin
    S_IRFBM^.next:=q^.next;           {get message from queue}
    if q^.next<>nil
        then begin
            q^.next^.prev:=S_IRFBM;
            q^.next:=nil
        end;
    q^.prev:=nil;
    ch:=q^.id[1];
    q^.id:=copy(q^.id,2,3);
    case ch of
        'A': transfer_node(q,A_IRFBM,lineSA,message_delay);
        'B': transfer_node(q,B_IRFBM,lineSB,message_delay);
        'C': transfer_node(q,C_IRFBM,lineSC,message_delay);
        'D': transfer_node(q,D_IRFBM,lineSD,message_delay);
        'E': transfer_node(q,E_IRFBM,lineSE,message_delay)
    end
end
else stop:=true
until stop
end; {send_RFBMs_from_S}

procedure send_BIDs_from_S;
var
    bn : bid_nodeptr;
    stop : boolean;
begin
    stop:=false;           {transfer bid informations queued at the central node}
    repeat
        bn:=S_BIDQ^.next;
        if (bn<>nil) and (bn^.arr=clock)
            then begin
                S_BIDQ^.next:=bn^.next;           {get bid node from queue}
                if bn^.next<>nil
                    then begin
                        bn^.next^.prev:=S_BIDQ;
                        bn^.next:=nil
                    end;
                bn^.prev:=nil;
                case bn^.dest of
                    'A': send_bid(bn^.t_id,A_IBQ,bn^.ind,bn^.value,lineSA);
                    'B': send_bid(bn^.t_id,B_IBQ,bn^.ind,bn^.value,lineSB);
                    'C': send_bid(bn^.t_id,C_IBQ,bn^.ind,bn^.value,lineSC);
                    'D': send_bid(bn^.t_id,D_IBQ,bn^.ind,bn^.value,lineSD);
                    'E': send_bid(bn^.t_id,E_IBQ,bn^.ind,bn^.value,lineSE)
                end;
                dispose(bn)
            end
        else stop:=true
    until stop
end; {send_BIDs_from_S}

procedure send_tasks_from_S;
var
    ch : char;
    q : nodeptr;
    stop : boolean;
    delay : integer;
begin
    stop:=false;           {transfer tasks queued at the central node}
    repeat
        q:=S_TQ^.next;

```



```

if (q<>nil) and (q^.arr_t=clock)
  then begin
    S_TQ^.next:=q^.next;           {get task node from task queue}
    if q^.next<>nil
      then begin
        q^.next^.prev:=S_TQ;
        q^.next:=nil
      end;
    q^.prev:=nil;
    delay:=q^.comp_t div 10+message_delay;
    ch:=q^.id[1];
    q^.id:=copy(q^.id,2,3);
    case ch of
      'A': transfer_node(q,A_tasks_ptr,lineSA,delay);
      'B': transfer_node(q,B_tasks_ptr,lineSB,delay);
      'C': transfer_node(q,C_tasks_ptr,lineSC,delay);
      'D': transfer_node(q,D_tasks_ptr,lineSD,delay);
      'E': transfer_node(q,E_tasks_ptr,lineSE,delay)
    end
  end
else stop:=true
until stop
end; {send_tasks_from_S}

```

Listing of The Main Program :

```

program bidding_and_focused_addressing;
{This program tries to schedule real-time tasks in a distributed system model.
Whenever a task cannot be guaranteed by the local node at which it arrives, it
is sent to a remote node by means of focused addressing and bidding scheme.}

const
  r = 5;           {number of resources on each node}
  rr = 8;
  MC = 3;         {max count used in real backtacking}
  TD = 45;       {average transmission delay between nodes}
  SD = 4;        {average scheduling delay on a bidder node}
  HB = 2;        {high bid}
  SIM_TIME = 2500; {simulation time}
  TOPOLOGY = 'C'; { 'C' for FULLY CONNECTED, 'S' for STAR}
  message_delay = 25; {system's no conflict message delay}
  window_lenght = 500; {period for surplus exchange}
  windows = 5;   {number of windows}

type
  id_type = string[4];
  string_type = string[5];
  resource_ar = array [1..r] of integer;
  real_ar = array [1..r] of real;
  bool_ar = packed array [1..r] of boolean;
  Old_EATptrtype = ^Old_EATtype;
  Old_EATtype = record
    Old_EAT : resource_ar;
    link : Old_EATptrtype
  end;
  nodeptr = ^nodetype;
  secminptr = ^secminptrtype;
  secminptrtype = record
    secminnode : nodeptr;
    nextsecmin : secminptr

```

```

        end;
nodetype = record
    id : id_type;
    arr_t,deadline,start_t,comp_t : integer;
    res_need : bool_ar;
    foc_ind : char;
    secmin : secminptr;
    prev,next : nodeptr;
    New_EAT : resource_ar;
    Old_EATptr : Old_EATptrtype
end;
bid_nodeptr = ^bid_nodetype;
bid_nodetype = record
    t_id : id_type;
    arr,value,ind : integer;
    dest : char;
    prev,next : bid_nodeptr;
end;
inc_bids_nodeptr = ^inc_bids_nodetype;
inc_bids_nodetype = record
    t_id : id_type;
    latest_bid_arr : integer;
    bids,arrrs : array [1..4] of integer;
    prev,next : inc_bids_nodeptr;
end;
inc_surplus_nodeptr = ^inc_surplus_nodetype;
inc_surplus_nodetype = record
    n_id,dest : char;
    arr : integer;
    surp : resource_ar;
    prev,next : inc_surplus_nodeptr;
end;
nd_range = 'A'..'E';
report_array = array[1..2,nd_range,1..3] of integer;
info_array = array[1..windows] of resource_ar;
surplus_array = array[nd_range] of resource_ar;

```

(busy times)
(fraction of free times)

```

var
infile : text;
active : bool_ar;
lwrange : set of char;
S_BIDQ : bid_nodeptr;
A_EAT,B_EAT,C_EAT,D_EAT,E_EAT : resource_ar;
A_IBQ,B_IBQ,C_IBQ,D_IBQ,E_IBQ : inc_bids_nodeptr;
A_tasks_ptr,A_disp,A_IRFBM,A_RFBQ,B_tasks_ptr,B_disp,B_IRFBM,B_RFBQ,
C_tasks_ptr,C_disp,C_IRFBM,C_RFBQ,D_tasks_ptr,D_disp,D_IRFBM,D_RFBQ,
E_tasks_ptr,E_disp,E_IRFBM,E_RFBQ,S_TQ,S_IRFBM : nodeptr;
lineAB,lineAC,lineAD,lineAE,lineBA,lineBC,lineBD,lineBE,lineCA,
lineCB,lineCD,lineCE,lineDA,lineDB,lineDC,lineDE,lineEA,lineEB,
lineEC,lineED,clock,tasks_guaranteed,tasks_disp_locally,
tasks_disp_nw_wide,tasks_nonguaranteed,lineAS,lineBS,lineCS,
lineDS,lineES,lineSA,lineSB,lineSC,lineSD,lineSE : integer;
A_report,B_report,C_report,D_report,E_report : report_array;
A_surplus,B_surplus,C_surplus,D_surplus,E_surplus : surplus_array;
A_info,B_info,C_info,D_info,E_info : info_array;
A_ISIQ,B_ISIQ,C_ISIQ,D_ISIQ,E_ISIQ,S_ISIQ : inc_surplus_nodeptr;

procedure Init;
var
j : nd_range;
i,ii : integer;
begin

```

```

active[1]:=true; active[2]:=true;           {set initial values}
active[3]:=false; active[4]:=false; active[5]:=false;
clock:=-1;
lineAB:=0; lineAC:=0; lineAD:=0; lineAE:=0;
lineBA:=0; lineBC:=0; lineBD:=0; lineBE:=0;
lineCA:=0; lineCB:=0; lineCD:=0; lineCE:=0;
lineDA:=0; lineDB:=0; lineDC:=0; lineDE:=0;
lineEA:=0; lineEB:=0; lineEC:=0; lineED:=0;
lineAS:=0; lineBS:=0; lineCS:=0; lineDS:=0; lineES:=0;
lineSA:=0; lineSB:=0; lineSC:=0; lineSD:=0; lineSE:=0;
tasks_guaranteed:=0; tasks_disp_locally:=0;
tasks_disp_nw_wide:=0; tasks_nonguaranteed:=0;
for i:=1 to r do
  begin
    A_EAT[i]:=0; B_EAT[i]:=0; C_EAT[i]:=0; D_EAT[i]:=0; E_EAT[i]:=0
  end;
for i:=1 to 2 do
  for j:='A' to 'E' do
    for ii:=1 to 3 do
      begin
        A_report[i,j,ii]:=0; B_report[i,j,ii]:=0;
        C_report[i,j,ii]:=0; D_report[i,j,ii]:=0; E_report[i,j,ii]:=0
      end
    end; {init}

($! sch.paz)

procedure init_foc;
var
  j : char;
  i,ii : integer;
begin
  lwrange:=['a'..'e'];
  for ii:=1 to windows do
    for i:=1 to r do
      begin
        A_info[i,ii]:=0; B_info[i,ii]:=0;
        C_info[i,ii]:=0; D_info[i,ii]:=0; E_info[i,ii]:=0
      end;
    for j:='A' to 'E' do
      for i:=1 to r do
        begin
          A_surplus[j,ii]:=0; B_surplus[j,ii]:=0;
          C_surplus[j,ii]:=0; D_surplus[j,ii]:=0; E_surplus[j,ii]:=0
        end
      end;
end; {init_foc}

procedure getting_tasks(fl:string_type;ch:char;var task_queue:nodeptr);
begin
  assign(infile,fl);
  reset(infile);
  get_tasks(task_queue);
  close(infile)
end; {getting_tasks}

procedure get_bid_node(var bn:bid_nodeptr);
begin
  new(bn);
  bn^.next:=nil;
  bn^.prev:=nil
end; {get_bid_node}

```

```

procedure get_inc_bids_node(var b:inc_bids_nodeptr);
var
  i : integer;
begin
  new(b);
  for i:=1 to 4 do
    begin
      b^.bids[i]:=0;
      b^.arrs[i]:=0
    end;
  b^.next:=nil;
  b^.prev:=nil
end; {get_inc_bids_node}

```

```

procedure get_inc_surplus_node(var s:inc_surplus_nodeptr);
begin
  new(s);
  s^.next:=nil;
  s^.prev:=nil
end; {get_inc_surplus_node}

```

```

procedure delete_inc_bids_node(var b:inc_bids_nodeptr);
var
  bb : inc_bids_nodeptr;
begin
  bb:=b;
  b:=bb^.next;
  bb^.prev^.next:=b;
  if b<>nil then b^.prev:=bb^.prev;
  dispose(bb)
end; {delete_inc_bids_node}

```

```

procedure getting_nodes(var DQ,RFBQ,IRFBM:nodeptr;
                       var IBQ:inc_bids_nodeptr;var ISIQ:inc_surplus_nodeptr);
begin
  get_node(DQ);
  get_node(RFBQ);
  get_node(IRFBM);
  get_inc_bids_node(IBQ);
  get_inc_surplus_node(ISIQ)
end; {getting_nodes}

```

```

procedure get_tasks_initialize_all;
begin
  init;
  init_foc;
  getting_tasks('A.dat', 'A', A_tasks_ptr);
  getting_nodes(A_disp, A_RFBQ, A_IRFBM, A_IBQ, A_ISIQ);
  getting_tasks('B.dat', 'B', B_tasks_ptr);
  getting_nodes(B_disp, B_RFBQ, B_IRFBM, B_IBQ, B_ISIQ);
  getting_tasks('C.dat', 'C', C_tasks_ptr);
  getting_nodes(C_disp, C_RFBQ, C_IRFBM, C_IBQ, C_ISIQ);
  getting_tasks('D.dat', 'D', D_tasks_ptr);
  getting_nodes(D_disp, D_RFBQ, D_IRFBM, D_IBQ, D_ISIQ);
  getting_tasks('E.dat', 'E', E_tasks_ptr);
  getting_nodes(E_disp, E_RFBQ, E_IRFBM, E_IBQ, E_ISIQ);
  get_node(S_TQ);
  get_node(S_IRFBM);
  get_bid_node(S_BIDQ);
  get_inc_surplus_node(S_ISIQ)
end; {get_tasks_initialize_all}

```

```

procedure deletequeue(var ptr:nodeptr);
var
  pointer : nodeptr;
begin
  repeat
    pointer:=ptr;
    ptr:=pointer^.next;
    dispose(pointer);
    pointer:=nil
  until ptr=nil
end; {deletequeue}

procedure insert_into_queue(qq,queue:nodeptr);
var
  ppointer,pointer : nodeptr;
begin
  ppointer:=queue;
  pointer:=queue^.next;
  while (pointer<>nil) and (pointer^.arr_t<=qq^.arr_t) do
    begin
      ppointer:=pointer;
      pointer:=pointer^.next
    end;
  if pointer<>nil then begin
    pointer^.prev:=qq;
    qq^.next:=pointer
  end;

  ppointer^.next:=qq;
  qq^.prev:=ppointer
end; {insert_into_queue}

procedure update_line(delay:integer;var line:integer);
begin
  case TOPOLOGY of
    'C':if line<clock then line:=clock+delay
        else line:=line+delay;
    'S':if line<clock then line:=clock+delay div 2
        else line:=line+delay div 2
  end
end; {update_line}

procedure insert_into_ISIQ(s,ISIQ:inc_surplus_nodeptr);
var
  ppointer,pointer : inc_surplus_nodeptr;
  i : integer;
begin
  ppointer:=ISIQ;
  pointer:=ISIQ^.next;
  while (pointer<>nil) and (pointer^.arr<=s^.arr) do
    begin
      ppointer:=pointer;
      pointer:=pointer^.next
    end;
  if pointer<>nil
  then begin
    pointer^.prev:=s;
    s^.next:=pointer
  end;

  ppointer^.next:=s;
  s^.prev:=ppointer
end; {insert_into_ISIQ}

```

```

procedure s_surplus(s, ISIQ:inc_surplus_nodeptr; var line:integer);
begin
  update_line(message_delay, line);
  s^.arr:=line;
  insert_into_ISIQ(s, ISIQ)
end; {s_surplus}

procedure send_surplus(surplus:surplus_array; nodeid, dest:char;
  ISIQ:inc_surplus_nodeptr; var line:integer);
var
  s : inc_surplus_nodeptr;
begin
  get_inc_surplus_node(s);
  s^.n_id:=nodeid;
  s^.dest:=dest;
  s^.surp:=surplus[nodeid];
  s_surplus(s, ISIQ, line)
end; {send_surplus}

procedure sending_surplus_C(surplus:surplus_array; id, d1, d2, d3, d4:char;
  ISIQ_1, ISIQ_2, ISIQ_3, ISIQ_4:inc_surplus_nodeptr;
  var line1, line2, line3, line4:integer);
begin
  send_surplus(surplus, id, d1, ISIQ_1, line1);
  send_surplus(surplus, id, d2, ISIQ_2, line2);
  send_surplus(surplus, id, d3, ISIQ_3, line3);
  send_surplus(surplus, id, d4, ISIQ_4, line4)
end; {sending_surplus_C}

procedure sending_surplus_S(surplus:surplus_array;
  id, d1, d2, d3, d4:char; var line:integer);
begin
  send_surplus(surplus, id, d1, S_ISIQ, line);
  send_surplus(surplus, id, d2, S_ISIQ, line);
  send_surplus(surplus, id, d3, S_ISIQ, line);
  send_surplus(surplus, id, d4, S_ISIQ, line)
end; {sending_surplus_S}

procedure surplus_exchange;
var
  i, wno : integer;
begin
  wno:=clock div window_lenght;
  for i:=1 to r do
    begin
      A_surplus['A', i]:=window_lenght-A_info[wno, i];
      B_surplus['B', i]:=window_lenght-B_info[wno, i];
      C_surplus['C', i]:=window_lenght-C_info[wno, i];
      D_surplus['D', i]:=window_lenght-D_info[wno, i];
      E_surplus['E', i]:=window_lenght-E_info[wno, i]
    end;
  case TOPOLOGY of
    'C':begin
      sending_surplus_C(A_surplus, 'A', 'B', 'C', 'D', 'E', B_ISIQ,
        C_ISIQ, D_ISIQ, E_ISIQ, lineAB, lineAC, lineAD, lineAE);
      sending_surplus_C(B_surplus, 'B', 'C', 'D', 'E', 'A', C_ISIQ,
        D_ISIQ, E_ISIQ, A_ISIQ, lineBC, lineBD, lineBE, lineBA);
      sending_surplus_C(C_surplus, 'C', 'D', 'E', 'A', 'B', D_ISIQ,
        E_ISIQ, A_ISIQ, B_ISIQ, lineCD, lineCE, lineCA, lineCB);
      sending_surplus_C(D_surplus, 'D', 'E', 'A', 'B', 'C', E_ISIQ,
        A_ISIQ, B_ISIQ, C_ISIQ, lineDE, lineDA, lineDB, lineDC);
      sending_surplus_C(E_surplus, 'E', 'A', 'B', 'C', 'D', A_ISIQ,

```

```

    B_ISIQ,C_ISIQ,D_ISIQ,lineEA,lineEB,lineEC,lineED)
end;
'S':begin
    sending_surplus_S(A_surplus,'A','B','C','D','E',lineAS);
    sending_surplus_S(B_surplus,'B','C','D','E','A',lineBS);
    sending_surplus_S(C_surplus,'C','D','E','A','B',lineCS);
    sending_surplus_S(D_surplus,'D','E','A','B','C',lineDS);
    sending_surplus_S(E_surplus,'E','A','B','C','D',lineES)
end
end
end; {surplus_exchange}

procedure find_delete_inc_bids_node(task_id:id_type;IBQ:inc_bids_nodeptr);
var
    b : inc_bids_nodeptr;
begin
    b:=IBQ^.next;
    while (b<>nil) and (b^.t_id<>task_id) do
        b:=b^.next;
    if b<>nil then delete_inc_bids_node(b)
end; {find_delete_inc_bids_node}

procedure check_foc_add(q:nodeptr;noda:id,ch1,ch2,ch3,ch4:char;
    var focnode:char;surplus:surplus_array;
    var possible:boolean);

const
    FAS = 0.5;
var
    max : real;
    i,time,ind : integer;
    factor : array[1..4] of real;

function freetime(q:nodeptr;res_surp:resource_ar):integer;
var
    i,ft : integer;
begin
    ft:=window_lenght;
    for i:=1 to r do
        if q^.res_need[i]
            then if ft>res_surp[i]
                then ft:=res_surp[i];
        freetime:=ft
    end; {freetime}

begin
    possible:=true;    {check whether there is a node for focused addressing}
    factor[1]:=freetime(q,surplus[ch1]);
    factor[2]:=freetime(q,surplus[ch2]);
    factor[3]:=freetime(q,surplus[ch3]);
    factor[4]:=freetime(q,surplus[ch4]);
    if q^.comp_t>window_lenght
        then time:=window_lenght
        else time:=q^.comp_t;
    for i:=1 to 4 do
        if factor[i]<>0
            then factor[i]:=factor[i]/time;
    ind:=0;
    max:=0;
    for i:=1 to 4 do
        if factor[i]>max
            then begin
                max:=factor[i];

```

```

        ind:=i
    end;
    if max>FRS
    then case ind of
        1: focnode:=ch1;
        2: focnode:=ch2;
        3: focnode:=ch3;
        4: focnode:=ch4
    end
    else begin
        possible:=false;
        q^.foc_nd:='X'
    end
end; {check_foc_add}

procedure transfer_node(q,queue:nodptr;var line:integer; delay:integer);
begin
    update_line(delay,line);
    q^.arr_t:=line;
    insert_into_queue(q,queue)
end; {transfer_node}

procedure send(q,queue:nodptr;var line:integer;delay:integer);
var
    qq : nodptr;
begin
    get_node(qq);
    copy_info(qq,q);
    qq^.foc_nd:=q^.foc_nd;
    transfer_node(qq,queue,line,delay)
end; {send}

procedure create_an_inc_bids_node(task_id:id_type;latest:integer;
                                var lBQ,b:inc_bids_nodptr);
var
    bb,bbb : inc_bids_nodptr;
begin
    get_inc_bids_node(b);
    b^.t_id:=task_id;
    b^.latest_bid_arr:=latest;
    bb:=lBQ;
    bbb:=bb^.next;
    while bbb<>nil do
        begin
            bb:=bbb;
            bbb:=bb^.next
        end;
    bb^.next:=b;
    b^.prev:=bb
end; {create_an_inc_bids_node}

function upc(ch:char):char;
begin
    upc:=chr(ord(ch)-ord('a')+ord('A'))
end;

function lwc(ch:char):char;
begin
    lwc:=chr(ord(ch)-ord('A')+ord('a'))
end;

procedure add_to_RFBQ(q,RFBQ:nodptr;nodeid:char);

```



```

var
  qq : nodeptr;
begin
  get_node(qq);
  copy_info(qq,q);
  qq^.foc_nd:=q^.foc_nd;
  insert_into_queue(qq,RFBQ)
end; {add_to_RFBQ}

procedure send_RFBM_C(q,IRFBM_1,IRFBM_2,IRFBM_3,IRFBM_4:nodeptr;
  ch1,ch2,ch3,ch4:char;var line1,line2,line3,line4:integer);
begin
  send(q,IRFBM_1,line1,message_delay);
  send(q,IRFBM_2,line2,message_delay);
  send(q,IRFBM_3,line3,message_delay);
  send(q,IRFBM_4,line4,message_delay)
end; {send_RFBM_C}

procedure send_RFBM_S(q:nodeptr;ch1,ch2,ch3,ch4:char;var line:integer);
begin
  q^.id:=ch1+q^.id;
  send(q,S_IRFBM,line,message_delay);
  q^.id[1]:=ch2;
  send(q,S_IRFBM,line,message_delay);
  q^.id[1]:=ch3;
  send(q,S_IRFBM,line,message_delay);
  q^.id[1]:=ch4;
  send(q,S_IRFBM,line,message_delay)
end; {send_RFBM_S}

procedure detect_index(nd,dest:char;var ind:integer);
begin
  if (succ(nd)=dest) or (nd=subc(succ(succ(dest))))
    then ind:=4
  else if (succ(succ(nd))=dest) or (nd=succ(succ(succ(dest))))
    then ind:=3
  else if (succ(succ(succ(nd)))=dest) or
    (nd=succ(succ(dest)))
    then ind:=2
  else ind:=1
end; {detect_index}

procedure form_an_inc_bids_node(id:id_type;l:integer;
  foc_IBQ:inc_bids_nodeptr;nd,fn:char);
var
  ind : integer;
  b : inc_bids_nodeptr;
begin
  create_an_inc_bids_node(id,l,foc_IBQ,b);      {create an incoming bids node}
  detect_index(nd,fn,ind);                      {at the focused node}
  b^.bids[ind]:=-1;                             {no bid is expected from the sender node}
  b^.arrs[ind]:=-1
end; {form_an_inc_bids_node}

procedure start_foc_add_C(q,foc_TQ,IRFBM1,IRFBM2,IRFBM3:nodeptr;
  l:integer;foc_IBQ:inc_bids_nodeptr;
  var lfn,lbn1,lbn2,lbn3:integer;
  nd,fn,bn1,bn2,bn3:char);

begin
  form_an_inc_bids_node(q^.id,l,foc_IBQ,nd,fn);
  q^.foc_nd:=fn;

```

```

send(q, IRFBM1, lbn1, message_delay);
send(q, IRFBM2, lbn2, message_delay);
send(q, IRFBM3, lbn3, message_delay);
send(q, foc_TQ, lfn, (q^.comp_t div 10)+message_delay)
end; {start_foc_add_C}

```

```

procedure start_foc_add_S(q:nodeptr;l:integer; var foc_IBQ:inc_bids_nodeptr;
var line:integer;nd,fn,bn1,bn2,bn3:char);

```

```

begin
form_an_inc_bids_node(q^.id,l,foc_IBQ,nd,fn);
q^.foc_nd:=fn;
q^.id:=bn1+q^.id;
send(q,S_IRFBM,line,message_delay);
q^.id[11]:=bn2;
send(q,S_IRFBM,line,message_delay);
q^.id[11]:=bn3;
send(q,S_IRFBM,line,message_delay);
q^.id[11]:=fn;
send(q,S_TQ,line,(q^.comp_t div 10)+message_delay)
end; {start_foc_add_S}

```

```

procedure start_foc_add(q, foc_TQ1, foc_TQ2, foc_TQ3, foc_TQ4, IRFBM_1,
IRFBM_2, IRFBM_3, IRFBM_4:nodeptr;
var foc_IBQ1, foc_IBQ2, foc_IBQ3, foc_IBQ4:inc_bids_nodeptr;
var line1, line2, line3, line4:integer;
nd, fn, ch1, ch2, ch3, ch4:char; latest:integer);

```

```

begin
q^.id[11]:=lwc(q^.id[11]);
if fn=ch1
then
case topology of
'C': start_foc_add_C(q, foc_TQ1, IRFBM_2, IRFBM_3, IRFBM_4, latest,
foc_IBQ1, line1, line2, line3, line4, nd, fn, ch2, ch3, ch4);
'S': start_foc_add_S(q, latest, foc_IBQ1, line1, nd, fn, ch2, ch3, ch4)
end
else
if fn=ch2
then
case topology of
'C': start_foc_add_C(q, foc_TQ2, IRFBM_3, IRFBM_4, IRFBM_1, latest,
foc_IBQ2, line2, line3, line4, line1, nd, fn, ch3, ch4, ch1);
'S': start_foc_add_S(q, latest, foc_IBQ2, line1, nd, fn, ch3, ch4, ch1)
end
else
if fn=ch3
then
case topology of
'C': start_foc_add_C(q, foc_TQ3, IRFBM_4, IRFBM_1, IRFBM_2, latest,
foc_IBQ3, line3, line4, line1, line2, nd, fn, ch4, ch1, ch2);
'S': start_foc_add_S(q, latest, foc_IBQ3, line1, nd, fn, ch4, ch1, ch2)
end
else
case topology of
'C': start_foc_add_C(q, foc_TQ4, IRFBM_1, IRFBM_2, IRFBM_3, latest,
foc_IBQ4, line4, line1, line2, line3, nd, fn, ch1, ch2, ch3);
'S': start_foc_add_S(q, latest, foc_IBQ4, line1, nd, fn, ch1, ch2, ch3)
end
end; {start_foc_add}

```

```

procedure start_bidding(q, RFBQ, IRFBM_1, IRFBM_2, IRFBM_3, IRFBM_4:nodeptr;
nd, ch1, ch2, ch3, ch4:char; var IBQ:inc_bids_nodeptr;
var line1, line2, line3, line4:integer; latest:integer);

```

```

var
  b : inc_bids_nodeptr;
begin
  create_an_inc_bids_node(q^.id, latest, IBQ, b);
  add_to_RFBQ(q, RFBQ, nd);
  case TOPOLOGY of
    'C': send_RFBM_C(q, IRFBM_1, IRFBM_2, IRFBM_3, IRFBM_4, ch1, ch2, ch3, ch4,
                    line1, line2, line3, line4);
    'S': send_RFBM_S(q, ch1, ch2, ch3, ch4, line1)
  end
end; {start_bidding}

procedure scheduling
  (nodeid:char; real_EAT:resource_ar; var RFBQ, DQ:nodeptr;
   var IBQ, foc_IBQ1, foc_IBQ2, foc_IBQ3, foc_IBQ4:inc_bids_nodeptr;
   TQ, foc_TQ1, foc_TQ2, foc_TQ3, foc_TQ4, IRFBM_1, IRFBM_2, IRFBM_3,
   IRFBM_4:nodeptr; ch1, ch2, ch3, ch4:char;
   var line1, line2, line3, line4:integer;
   var report:report_array; surplus:surplus_array);
var
  q : nodeptr;
  focnode, ch : char;
  i, latest, dim : integer;
  SQ, TQ2 : nodeptr;
  EAT : resource_ar;
  stop, guaranteed, possible : boolean;
begin
  stop:=false;
  repeat
    q:=TQ^.next;
    if (q<>nil) and (q^.arr_t=clock)
    then
      begin
        TQ^.next:=q^.next;
        if q^.next<>nil then begin
          q^.next^.prev:=TQ;
          q^.next:=nil
        end;
        q^.prev:=nil;
        for l:=1 to r do
          if real_EAT[l]<clock then EAT[l]:=clock
            else EAT[l]:=real_EAT[l];
          if q^.id[l] in lwrange then begin
            ch:=upc(q^.id[l]);
            if q^.foc_nd=nodeid
            then dim:=1
            else dim:=2
          end
          else begin
            ch:=q^.id[l];
            dim:=3
          end;
          report[l, ch, dim]:=report[l, ch, dim]+1;
          scheduler(q, DQ, EAT, guaranteed, TQ2, SQ, 1);
          if guaranteed
          then
            begin
              tasks_guaranteed:=tasks_guaranteed+1;
              report[2, ch, dim]:=report[2, ch, dim]+1;
              if q^.id[l] in lwrange
              then find_delete_inc_bids_node(q^.id, IBQ);
              deletequeue(DQ);
            end;
          end;
        end;
      end;
  until stop;
end; {delete Dispatcher Queue}

```

```

DQ:=SQ;                                     {Schedule Queue becomes DQ}
dispose(TQ2);
TQ2:=nil
end
else
begin
latest:=q^.deadline-q^.comp_t-(TD+SD);
if latest<=clock
then tasks_nonguaranteed:=tasks_nonguaranteed+1
else
begin
q^.New_EAT[1]:=latest;
for i:=2 to r do q^.New_EAT[i]:=0;
if q^.id[1]=nodeid
then {if the task is local send it to another node}
begin
check_foc_add(q,nodeid,ch1,ch2,ch3,ch4,focnode,
surplus,possible);
if possible
then start_foc_add(q,foc_TQ1,foc_TQ2,foc_TQ3,
foc_TQ4,IRFBM_1,IRFBM_2,IRFBM_3,IRFBM_4,
foc_IBQ1,foc_IBQ2,foc_IBQ3,foc_IBQ4,line1,
line2,line3,line4,nodeid,focnode,ch1,ch2,
ch3,ch4,latest)
else start_bidding(q,RFBQ,IRFBM_1,IRFBM_2,
IRFBM_3,IRFBM_4,nodeid,ch1,ch2,ch3,ch4,
IBQ,line1,line2,line3,line4,latest)
end
else if ((q^.id[1] in lrange) and (nodeid=q^.foc_nd))
then add_to_RFBQ(q,RFBQ,nodeid) {try bidding}
else tasks_nonguaranteed:=tasks_nonguaranteed+1
end;
deletequeue(TQ2); TQ2^.next:=nil;
deletequeue(SQ); SQ^.next:=nil;
dispose(q); q:=nil
end
end
else stop:=true
until stop
end; {scheduling}

```

```

procedure update_info(p:nodeptr;var info:info_array);
var
i,start,termination,wno,duration,limit : integer;
begin
start:=clock;
duration:=p^.comp_t;
termination:=start+duration;
wno:=(start div window_lenght)+1;
while duration<>0 do
begin
limit:=wno*window_lenght;
if termination<=limit
then begin
for i:=1 to r do
if p^.res_need[i]
then info[wno,i]:=info[wno,i]+duration;
duration:=0
end
else begin
for i:=1 to r do
if p^.res_need[i]

```

```

        then info[wno,i]:=info[wno,i]+limit-start;
        start:=limit;
        duration:=termination-limit;
        wno:=wno+1
    end
end
end; {update_info}

procedure dispatching(nodeid:char;DQ:nodeptr;
    var real_EAT:resource_ar;var info:info_array);
var
    p,pp : nodeptr;
    i : integer;
begin
    pp:=DQ;
    p:=pp^.next;
    while p<>nil do
        if p^.start_t=clock
            then begin
                if nodeid=p^.id[1]
                    then begin
                        tasks_disp_locally:=tasks_disp_locally+1;
                        {increment number of tasks dispatched locally}
                        update_info(p,info)
                    end
                else tasks_disp_nw_wide:=tasks_disp_nw_wide+1;
                    {increment number of tasks dispatched network_wide}
                    for i:=1 to r do real_EAT[i]:=p^.New_EAT[i];
                    pp^.next:=p^.next;
                    if p^.next<>nil then p^.next^.prev:=pp;
                    dispose(p);
                    p:=pp^.next;
                end
            else begin
                pp:=p;
                p:=p^.next;
            end
        end
    end; {dispatching}

procedure send_bid(task_id:id_type;IBQ:inc_bids_nodeptr;
    index,BID:integer;var line:integer);
var
    p : inc_bids_nodeptr;
    i : integer;
begin
    update_line(message_delay,line);
    if task_id[2] in lwrange then task_id:=copy(task_id,2,3);
    p:=IBQ^.next;
    {search the node in the Incoming Bids Queue}
    while (p<>nil) and (p^.t_id<>task_id) do
        p:=p^.next;
    if p<>nil
        then begin
            p^.arrs[index]:=line;
            p^.bids[index]:=BID
        end;
end; {send_bid}

procedure insert_into_S_BIDQ(bn:bid_nodeptr);
var
    ppointer:pointer : bid_nodeptr;
begin
    ppointer:=S_BIDQ;

```

```

pointer:=S_BIDQ^.next;
while (pointer<>nil) and (pointer^.arr<=bn^.arr) do
begin
  ppointer:=pointer;
  pointer:=pointer^.next
end;
if pointer<>nil
then begin
  pointer^.prev:=bn;
  bn^.next:=pointer
end;
ppointer^.next:=bn;
bn^.prev:=ppointer
end; {insert_into_S_BIDQ}

procedure bidding(nodeid:char;real_EAT:resource_ar;IRFBM,DQ:nodeptr;
  IBQ_1,IBQ_2,IBQ_3,IBQ_4:inc_bids_nodeptr;
  var line1,line2,line3,line4:integer);
var
  b : nodeptr;
  SQ,TQ2 : nodeptr;
  stop,first : boolean;
  temp_EAT : resource_ar;
  MAXBID,BID,latest_bid_arr,bid_arr,no : integer;

  {$I bid.pas}

begin
  stop:=false;
  repeat
    b:=IRFBM^.next;
    if ((b<>nil) and (b^.arr_t=clock))
    then begin
      IRFBM^.next:=b^.next;           {get task from IRFBM queue}
      if b^.next<>nil
      then begin
        b^.next^.prev:=IRFBM;
        b^.next:=nil
      end;
      b^.prev:=nil;
      if b^.id[1] in lwrange then b^.id:=b^.foc_nd+b^.id;
      indicate_line(nodeid,b^.id[1],bid_arr,no);
      latest_bid_arr:=b^.New_EAT[1];
      update_line(message_delay,bid_arr);
      if TOPOLOGY='S' then bid_arr:=bid_arr+message_delay div 2;
      if latest_bid_arr>=bid_arr
      then begin
        calculate_MAXBID;
        binary_search_for_BID;
        case TOPOLOGY of
          'C':bidding_C;
          'S':bidding_S
        end
      end;
      dispose(b)
    end
  else stop:=true
  until stop
end; {bidding}

procedure delete_task(var p:nodeptr);
var

```

```

pp : nodeptr;
begin
  pp:=p;
  p:=pp^.next;
  pp^.prev^.next:=p;
  if p<>nil then p^.prev:=pp^.prev;
  dispose(pp)
end; {delete_task}

procedure find_the_task(var b:inc_bids_nodeptr;RFBQ:nodeptr;var p:nodeptr);
begin
  p:=RFBQ^.next;
  while (p<>nil) and (p^.id<>b^.t_id) do p:=p^.next;
  if p=nil then b:=b^.next
  else delete_inc_bids_node(b)
end; {find_the_task}

{$i bideval.pas}

procedure updating_surplus_info;

procedure update_surplus_info(ISIQ:inc_surplus_nodeptr;nodeid:char;
                             var surplus:surplus_array);
var
  s,ss : inc_surplus_nodeptr;
begin
  s:=ISIQ^.next;
  while s<>nil do
    if s^.arr=clock
      then begin
        surplus[s^.n_id]:=s^.surp;
        ss:=s;
        s:=ss^.next;
        ss^.prev^.next:=s;
        if s<>nil then s^.prev:=ss^.prev;
        dispose(ss);
        ss:=nil
      end
    else s:=s^.next
  end; {update_surplus_info}

begin
  update_surplus_info(A_ISIQ,'A',A_surplus);
  update_surplus_info(B_ISIQ,'B',B_surplus);
  update_surplus_info(C_ISIQ,'C',C_surplus);
  update_surplus_info(D_ISIQ,'D',D_surplus);
  update_surplus_info(E_ISIQ,'E',E_surplus)
end; {updating_surplus_info}

procedure fully_connected_topology;
begin
  repeat
    clock:=clock+1;
    if (clock mod window_lenght)=1
      then if clock<>1
        then surplus_exchange;
    scheduling('A',A_EAT,A_RFBQ,A_disp,A_IBQ,B_IBQ,C_IBQ,D_IBQ,E_IBQ,
              A_tasks_ptr,B_tasks_ptr,C_tasks_ptr,D_tasks_ptr,E_tasks_ptr,
              B_IRFBM,C_IRFBM,D_IRFBM,E_IRFBM,'B','C','D','E',lineAB,
              lineAC,lineAD,lineAE,A_report,A_surplus);
    dispatching('A',A_disp,A_EAT,A_info);
    bidding('A',A_EAT,A_IRFBM,A_disp,B_IBQ,C_IBQ,D_IBQ,E_IBQ,lineAB,

```

```

    lineAC, lineAD, lineAE);
bid_evaluating(A_IBQ, A_RFBQ, B_tasks_ptr, C_tasks_ptr, D_tasks_ptr,
    E_tasks_ptr, 'B', 'C', 'D', 'E', lineAB, lineAC, lineAD, lineAE);
scheduling('B', B_EAT, B_RFBQ, B_disp, B_IBQ, C_IBQ, D_IBQ, E_IBQ, A_IBQ,
    B_tasks_ptr, C_tasks_ptr, D_tasks_ptr, E_tasks_ptr, A_tasks_ptr,
    C_IRFBM, D_IRFBM, E_IRFBM, A_IRFBM, 'C', 'D', 'E', 'A', lineBC,
    lineBD, lineBE, lineBA, B_report, B_surplus);
dispatching('B', B_disp, B_EAT, B_info);
bidding('B', B_EAT, B_IRFBM, B_disp, C_IBQ, D_IBQ, E_IBQ, A_IBQ, lineBC,
    lineBD, lineBE, lineBA);
bid_evaluating(B_IBQ, B_RFBQ, C_tasks_ptr, D_tasks_ptr, E_tasks_ptr,
    A_tasks_ptr, 'C', 'D', 'E', 'A', lineBC, lineBD, lineBE, lineBA);
scheduling('C', C_EAT, C_RFBQ, C_disp, C_IBQ, D_IBQ, E_IBQ, A_IBQ, B_IBQ,
    C_tasks_ptr, D_tasks_ptr, E_tasks_ptr, A_tasks_ptr, B_tasks_ptr,
    D_IRFBM, E_IRFBM, A_IRFBM, B_IRFBM, 'D', 'E', 'A', 'B', lineCD,
    lineCE, lineCA, lineCB, C_report, C_surplus);
dispatching('C', C_disp, C_EAT, C_info);
bidding('C', C_EAT, C_IRFBM, C_disp, D_IBQ, E_IBQ, A_IBQ, B_IBQ, lineCD,
    lineCE, lineCA, lineCB);
bid_evaluating(C_IBQ, C_RFBQ, D_tasks_ptr, E_tasks_ptr, A_tasks_ptr,
    B_tasks_ptr, 'D', 'E', 'A', 'B', lineCD, lineCE, lineCA, lineCB);
scheduling('D', D_EAT, D_RFBQ, D_disp, D_IBQ, E_IBQ, A_IBQ, B_IBQ, C_IBQ,
    D_tasks_ptr, E_tasks_ptr, A_tasks_ptr, B_tasks_ptr, C_tasks_ptr,
    E_IRFBM, A_IRFBM, B_IRFBM, C_IRFBM, 'E', 'A', 'B', 'C', lineDE,
    lineDA, lineDB, lineDC, D_report, D_surplus);
dispatching('D', D_disp, D_EAT, D_info);
bidding('D', D_EAT, D_IRFBM, D_disp, E_IBQ, A_IBQ, B_IBQ, C_IBQ, lineDE,
    lineDA, lineDB, lineDC);
bid_evaluating(D_IBQ, D_RFBQ, E_tasks_ptr, A_tasks_ptr, B_tasks_ptr,
    C_tasks_ptr, 'E', 'A', 'B', 'C', lineDE, lineDA, lineDB, lineDC);
scheduling('E', E_EAT, E_RFBQ, E_disp, E_IBQ, A_IBQ, B_IBQ, C_IBQ, D_IBQ,
    E_tasks_ptr, A_tasks_ptr, B_tasks_ptr, C_tasks_ptr, D_tasks_ptr,
    A_IRFBM, B_IRFBM, C_IRFBM, D_IRFBM, 'A', 'B', 'C', 'D', lineEA,
    lineEB, lineEC, lineED, E_report, E_surplus);
dispatching('E', E_disp, E_EAT, E_info);
bidding('E', E_EAT, E_IRFBM, E_disp, A_IBQ, B_IBQ, C_IBQ, D_IBQ, lineEA,
    lineEB, lineEC, lineED);
bid_evaluating(E_IBQ, E_RFBQ, A_tasks_ptr, B_tasks_ptr, C_tasks_ptr,
    D_tasks_ptr, 'A', 'B', 'C', 'D', lineEA, lineEB, lineEC, lineED);
updating_surplus_info;
until clock=SIM_TIME
end; {fully_connected_topology}

```

```
{$i fromS.pas}
```

```
procedure send_surplus_infos_from_S;
```

```
var
```

```
  s : Inc_surplus_nodeptr;
```

```
  stop : boolean;
```

```
begin
```

```
  stop:=false;           {transfer surplus informations queued at the central node}
```

```
  repeat
```

```
    s:=S_IBQ^.next;
```

```
    if (s<>nil) and (s^.arr=clock)
```

```
      then begin
```

```
        S_IBQ^.next:=s^.next;
```

```
                          {get surplus info from queue}
```

```
        if s^.next<>nil
```

```
          then begin
```

```
            s^.next^.prev:=S_IBQ;
```

```
            s^.next:=nil
```

```
          end;
```

```
        s^.prev:=nil;
```



```

        case s^.dest of
            'A': s_surplus(s, A_ISIQ, lineSA);
            'B': s_surplus(s, B_ISIQ, lineSB);
            'C': s_surplus(s, C_ISIQ, lineSC);
            'D': s_surplus(s, D_ISIQ, lineSD);
            'E': s_surplus(s, E_ISIQ, lineSE)
        end
    end
    else stop:=true
until stop
end; {send_surplus_infos_from_S}

procedure transfers_from_S;
begin
    send_surplus_infos_from_S;
    send_RFBMs_from_S;
    send_BIDs_from_S;
    send_tasks_from_S
end; {transfers_from_S}

procedure star_topology;
var
    x : nodeptr;
    z : inc_bids_nodeptr;
begin
    x:=nil; z:=nil;
    repeat
        clock:=clock+1;
        if (clock mod window_lenght)=1
            then if clock<>1
                then surplus_exchange;
            scheduling('A', A_EAT, A_RFBQ, A_disp, A_IBQ, B_IBQ, C_IBQ, D_IBQ, E_IBQ,
                A_tasks_ptr, x, x, x, x, x, x, x, x, 'B', 'C', 'D', 'E', lineAS, lineAS,
                lineAS, lineAS, A_report, A_surplus);
            dispatching('A', A_disp, A_EAT, A_info);
            bidding('A', A_EAT, A_IRFBM, A_disp, z, z, z, z, lineAS, lineAS, lineAS, lineAS);
            bid_evaluating(A_IBQ, A_RFBQ, S_TQ, S_TQ, S_TQ, S_TQ, 'B', 'C', 'D', 'E',
                lineAS, lineAS, lineAS, lineAS);
            scheduling('B', B_EAT, B_RFBQ, B_disp, B_IBQ, C_IBQ, D_IBQ, E_IBQ, A_IBQ,
                B_tasks_ptr, x, x, x, x, x, x, x, x, 'C', 'D', 'E', 'A', lineBS, lineBS,
                lineBS, lineBS, B_report, B_surplus);
            dispatching('B', B_disp, B_EAT, B_info);
            bidding('B', B_EAT, B_IRFBM, B_disp, z, z, z, z, lineBS, lineBS, lineBS, lineBS);
            bid_evaluating(B_IBQ, B_RFBQ, S_TQ, S_TQ, S_TQ, S_TQ, 'C', 'D', 'E', 'A',
                lineBS, lineBS, lineBS, lineBS);
            scheduling('C', C_EAT, C_RFBQ, C_disp, C_IBQ, D_IBQ, E_IBQ, A_IBQ, B_IBQ,
                C_tasks_ptr, x, x, x, x, x, x, x, x, 'D', 'E', 'A', 'B', lineCS, lineCS,
                lineCS, lineCS, C_report, C_surplus);
            dispatching('C', C_disp, C_EAT, C_info);
            bidding('C', C_EAT, C_IRFBM, C_disp, z, z, z, z, lineCS, lineCS, lineCS, lineCS);
            bid_evaluating(C_IBQ, C_RFBQ, S_TQ, S_TQ, S_TQ, S_TQ, 'D', 'E', 'A', 'B',
                lineCS, lineCS, lineCS, lineCS);
            scheduling('D', D_EAT, D_RFBQ, D_disp, D_IBQ, E_IBQ, A_IBQ, B_IBQ, C_IBQ,
                D_tasks_ptr, x, x, x, x, x, x, x, x, 'E', 'A', 'B', 'C', lineDS, lineDS,
                lineDS, lineDS, D_report, D_surplus);
            dispatching('D', D_disp, D_EAT, D_info);
            bidding('D', D_EAT, D_IRFBM, D_disp, z, z, z, z, lineDS, lineDS, lineDS, lineDS);
            bid_evaluating(D_IBQ, D_RFBQ, S_TQ, S_TQ, S_TQ, S_TQ, 'E', 'A', 'B', 'C',
                lineDS, lineDS, lineDS, lineDS);
            scheduling('E', E_EAT, E_RFBQ, E_disp, E_IBQ, A_IBQ, B_IBQ, C_IBQ, D_IBQ,
                E_tasks_ptr, x, x, x, x, x, x, x, x, 'A', 'B', 'C', 'D', lineES, lineES,
                lineES, lineES, E_report, E_surplus);
    end
end

```

```

dispatching('E',E_disp,E_EAT,E_info);
bidding('E',E_EAT,E_IRFBM,E_disp,z,z,z,lineES,lineES,lineES,lineES);
bid_evaluating(E_IBQ,E_RFBQ,S_TQ,S_TQ,S_TQ,S_TQ,'A','B','C','D',
lineES,lineES,lineES,lineES);
transfers_from_S;
updating_surplus_info;
until clock=SIM_TIME
end; {star_topology}

procedure print_report(report:report_array);
var
  dim : 1..3;
  nd : nd_range;
begin
  write('tasks_arrived ');
  for nd:='A' to 'E' do
    begin
      for dim:=1 to 3 do
        write(report[1,nd,dim]:3);
        write(' ');
      end;
      writeln; write('tasks_dispatched:');
      for nd:='A' to 'E' do
        begin
          for dim:=1 to 3 do
            write(report[2,nd,dim]:3);
            write(' ');
          end;
          writeln
        end;
      end; {print_report}

procedure print_rep;
begin
  writeln;
  writeln('NODE A : : local : from B : from C : from D : from E :');
  print_report(A_report);
  writeln;
  writeln('NODE B : : from A : local : from C : from D : from E :');
  print_report(B_report);
  writeln;
  writeln('NODE C : : from A : from B : local : from D : from E :');
  print_report(C_report);
  writeln;
  writeln('NODE D : : from A : from B : from C : local : from E :');
  print_report(D_report);
  writeln;
  writeln('NODE E : : from A : from B : from C : from D : local :');
  print_report(E_report);
end; {print_rep}

function calc(i1:integer):integer;
var
  j : nd_range;
  n : integer;
begin
  n:=0;
  for j:='A' to 'E' do
    n:=n+A_report[2,j,i1]+B_report[2,j,i1]+C_report[2,j,i1]
    +D_report[2,j,i1]+E_report[2,j,i1];
  calc:=n
end; {calc}

```

```
procedure writing_the_results;
var
  n_by_foc,n_by_foc_bid,n_by_bid : integer;
begin
  print_rep;
  n_by_foc:=calc(1);
  n_by_foc_bid:=calc(2);
  n_by_bid:=calc(3)-tasks_disp_locally;
  writeln;
  writeln('NUMBER OF TASKS :',(tasks_guaranteed+tasks_nonguaranteed):5);
  writeln('NUMBER OF TASKS GUARANTEED :',tasks_guaranteed:5);
  writeln('NUMBER OF TASKS DISPATCHED LOCALLY :',tasks_disp_locally:5);
  writeln('NUMBER OF TASKS DISPATCHED NETWORK WIDE :',tasks_disp_nw_wide:5);
  writeln('          BY FOC_NODE :',n_by_foc:5);
  writeln('          BY SECOND_STEP_NODE :',n_by_foc_bid:5);
  writeln('          BY DIRECT BIDDING :',n_by_bid:5);
  writeln('NUMBER OF TASKS NONGUARANTEED :',tasks_nonguaranteed:5);
end; {writing_the_results}

begin {main}
  writeln;
  get_tasks_initialize_all;
  case TOPOLOGY of
    'C':fully_connected_topology;
    'S':star_topology;
  end;
  writing_the_results
end. {main}
```

BIBLIOGRAPHY

1. Zhao, W., Ramamritham, K., and Stankovic, J.A., "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 5, pp. 564-576, May 1987.
2. Ramamritham, K., Stankovic, J.A., and Zhao, W., "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, pp. 1110-1123, August 1989.
3. Stankovic, J.A., "A Perspective on Distributed Computer Systems," *IEEE Transactions on Computers* Vol. C-33, No. 12, pp. 1102-1115, December 1984.
4. Stone, H., and Bokhari, S. H., "Control of Distributed Processes," *IEEE Computer*, Vol. 11, No. 7, pp. 97-106, July 1978.
5. Kleinrock, L., "Distributed Systems," *Communications of the ACM*, Vol. 28, No. 11, pp. 1200-1213, November 1985.
6. Barclay, D. K., Byrne, E. R., and Ng, F. K., "A Real-Time Database Management System for No.5 ESS," *Bell Syst. Tech. J.*, Vol. 61, No. 9, November 1982.
7. Ayache, J. M., Courtiat, J. P., and Diaz, M., "REBUS, A Fault Tolerant Distributed System for Industrial Control," *IEEE Transactions on Computers*, Vol. C-31, July 1982.
8. Melliar-Smith, P. M., and Schwartz, R. L., "Formal Specification and Mechanical Verification of SIFT," *IEEE Transactions on Computers*, Vol. C-31, July 1982.
9. Smith, R.G., "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers*, Vol. C-29, No. 12, pp. 1104-1113, December 1980.

10. Casavant, Thomas L., and Kuhl, Jon G., "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 2, pp. 141-154, February 1988.
11. Lo, V.M., "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, Vol. C-37, No. 11, pp. 1384-1397, November 1988.
12. Efe, K., "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, pp.50-56, June 1982.
13. Ma, R.P., Lee, E.Y.S., and Tsuchiya, M., "A Task Allocation Model for Distributed Computer Systems," *IEEE Transactions on Computers*, Vol. C-31, No. 1, pp. 41-47, January 1982.
14. Ramamritham, K., and Stankovic, J.A., "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, Vol. 1, No. 3, pp. 65-75, July 1984.
15. Graham, R.L., Lawler, E.L., Lenstra, J.K., and Kan, A.H.G.R., "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey," *Annals of Discrete Mathematics*, 5, 1979.
16. Xu, J., and Parnas, D.L., "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, pp. 360-369, March 1990.
17. Garey, M.R., and Johnson, D.S., "Scheduling Tasks with Nonuniform Deadlines on Two Processors," *Journal of the ACM*, Vol. 23, No. 3, pp. 461-467, July 1976.
18. Liu, C.L., and Layland, J., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61, January 1973.
19. Chetto, H., and Chetto, M., "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Transactions on Software Engineering*, Vol. 15, No. 10, pp. 1261-1269, October 1989.
20. Teixeira, T., "Static Priority Interrupt Scheduling," *Proceedings of the Seventh Texas Conference on Computing Systems*, November 1978.
21. Johnson, H., and Madison, M.S., "Deadline Scheduling for a Real-Time Multiprocessor," NTIS (N76-15843), Springfield, VA, May 1974.

22. Blazewicz, J., Drabowski, M., and Weglarz, J., "Scheduling Multiprocessor Tasks to Minimize Schedule Length," *IEEE Transactions on Computers*, Vol. C-35, No. 5, pp. 389-393, May 1986.
23. Leinbaugh, D.W., "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, pp. 85-90, January 1980.
24. Lenat, Douglas B., "The Nature of Heuristics," *Artificial Intelligence*, 19, 1982.
25. Zhao, W., Ramamritham, K., and Stankovic, J.A., "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers*, Vol. C-36, No. 8, pp. 949-960, August 1987.
26. Dertouzos, M., "Control Robotics: The Procedural Control of Physical Process," *Proc. of the IFIP Congress*, 1974.
27. Dertouzos, M., and Mok, A.K., "Multiprocessor On-line Scheduling of Hard Real-Time Tasks," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pp. 1497-1506, December 1989.
28. Jensen, E.D., Locke, C.D., and Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of IEEE Real-Time Systems Symposium*, December 1985.
29. Bokhari, S.H., "On the Mapping Problem," *IEEE Transactions on Computers*, Vol. C-30, No. 3, pp. 207-214, March 1981.
30. Stone, H., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, pp. 85-93, January 1977.
31. Chu, W.W., and Lan, M.T., "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Transactions on Computers*, Vol. C-36, No. 6, pp. 667-679, June 1987.
32. Leinbaugh, D.W., and Yamini, M., "Guaranteed Response Times in a Distributed Hard Real-Time Environment," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 12, pp. 1139-1143, December 1986.

33. Ma, R.P., "A Model to Solve Timing-Critical Application Problems in Distributed Computer Systems," *IEEE Computer*, Vol. 17, pp. 62-68, January 1984.
34. Livny, M., and Melman, M., "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proceedings of ACM Computer Network Performance Symposium*, April 1982.
35. Wang, Y., and Morris, R., "Load Sharing in Distributed Systems," *IEEE Transactions on Computers*, Vol. C-34, No. 3, pp. 204-217, March 1985.
36. Stankovic, J.A., Ramamritham, K., and Cheng, S., "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers*, Vol. C-34, No. 12, pp. 1130-1143, December 1985.
37. Kurose, J.F., and Chipalkatti, R., "Load Sharing in Soft Real-Time Distributed Computer Systems," *IEEE Transactions on Computers*, Vol. C-36, No. 8, pp. 993-1000, August 1987.

REFERENCES NOT CITED

- Bach, Maurice J. *The Design of the UNIX Operating System*. London : Prentice/Hall International, Inc., 1986.
- Casavant, Thomas L., and Kuhl, Jon G., "Effect of Response and Stability on Scheduling in Distributed Computing Systems," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, pp. 1578-1588, November 1988.
- Eager, Derek L., Lazowska, Edward D., and Zahorjan, John, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, pp. 662-675, May 1986.
- Lewis, T.G., and Smith, B.J. *Computer Principles of Modelling and Simulation*. Houghton Mifflin Company, 1979.
- Liestman, Arthur L., and Campbell, Roy H., "A Fault-Tolerant Scheduling Problem," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 11, pp. 1089-1095, November 1986.
- Lorin, Harold. *Aspects of Distributed Computer Systems*. John Wiley & Sons, Inc., 1988.
- Maekawa, M., Oldehoeft A.E., Oldehoeft R.R. *Operating Systems: Advanced Concept*. The Benjamin/Cummings Pub. Co., 1987.
- Pasquale, Joseph, "Using Expert Systems to Manage Distributed Computer Systems," *IEEE Network*, pp. 22-28, September 1988.
- Peterson, James L., and Silberschatz, Abraham. *Operating System Concepts*. Reading, Massachuttes : Addison-Wesley Pub. Co., 1985.
- Schrott, Gerhard, "A Generalized Task Concept for Multiprocessor Real-Time Systems," *Microprocessing and Microprogramming*, 20, pp. 85-90, 1987.

- Stankovic, John A., "Decentralized Decision Making for Task Reallocation in a Hard Real-Time System," *IEEE Transactions on Computers*, Vol. 38, No. 3, pp. 341-355, March 1989.
- Stankovic, John A., "Stability and Distributed Scheduling Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, pp. 1141-1152, October 1985.
- Tanenbaum, Andrew S. *Computer Networks*. London : Prentice/Hall International, Inc., 1981.

