Technical Report

CMU/SEI-95-TR-011
ESC-TR-95-011

Distributed System Design Using Generalized Rate Monotonic Theory

Lui Sha
Shirish S. Sathaye

September 1995

Technical Report
CMU/SEI-95-TR-011
ESC-TR-95-011
September 1995

Distributed System Design Using Generalized Rate Monotonic Theory

Lui Sha
Dependable Real-Time Software

Shirish S. Sathaye
Electrical and Computer Engineering
Carnegie Mellon Univeresity

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8222 or 1-800 225-3842.]

Table of Contents

List of Figures

Distributed System Design
Using Generalized Rate Monotonic Theory

Abstract: Rate monotonic theory and its generalizations have been adopted by national high technology projects such as the space station and have recently been supported by major open standards such as the IEEE Futurebus+ and POSIX.4. In this paper, we describe the use of generalized rate monotonic scheduling theory for the design and analysis of a distributed real-time system. We review the theory, examine the architectural requirements for the use of the theory, and finally provide an application example.

1. Introduction

In real-time applications, the correctness of computation depends upon not only its results but also the time at which outputs are generated. The measures of merit in a real-time system include:

- Predictably fast response to urgent events.
- High degree of schedulability. Schedulability is the degree of resource utilization at or below which the timing requirements of tasks can be ensured. It can be thought as a measure of the number of *timely* transactions per second.
- Stability under transient overload. When the system is overloaded by events and it is impossible to meet all the deadlines, we must still guarantee the deadlines of selected critical tasks.

Generalized rate monotonic scheduling (GRMS) theory allows system developers to meet the above requirements by managing system concurrency and timing constraints at the level of tasking and message passing. In essence, this theory ensures that as long as the system utilization of all tasks lies below a certain bound, and appropriate scheduling algorithms are used, all tasks meet their deadlines. This puts the development and maintenance of real-time systems on an analytic, engineering basis, making these systems easier to develop and maintain.

The generalized rate monotonic theory begins with the pioneering work in Liu [17], in which the rate monotonic algorithm was introduced for scheduling independent periodic tasks. RMS is an optimal static priority scheduling algorithm for independent periodic tasks with end of period deadlines. The rate monotonic scheduling (RMS) algorithm gives higher priorities to periodic tasks with higher rates. RMS theory has since been generalized to analyze the schedulability of aperiodic tasks with both soft deadlines and hard deadlines [26], tasks with arbitrary deadlines [15], tasks with deadlines shorter than periods [16], inter-dependent tasks that must synchronize [22, 19, 20], and single tasks having multiple code segments with different priority assignment [8]. RMS has also been used to analyze wide

area network scheduling [25], and to improve response times of aperiodic messages in a token ring network [27]. The implications of RMS on Ada scheduling rules are discussed in Sha [23], and schedulability analysis of input/output paradigms have been treated in Klein [11]. The theory has also been applied in the development of the ARTS real-time operating system [29]. Cache algorithms for real-time systems using RMS were developed in Kirk [10]. Schedulability models for different operating system paradigms have been developed in Katcher [9]. RMS has also been applied to recover from faults using transient servers [28]. Rate Monotonic Scheduling (RMS) with its extensions is henceforth called Generalized Rate Monotonic Scheduling (GRMS).

Because of its versatility and ease of use, GRMS has gained rapid acceptance. For example it is used for developing real-time software in the NASA Space Station Freedom Program [7], the European Space Agency [4] and is supported by the IEEE Futurebus+ Standard [6] and IEEE Posix.4 [18].

Uniprocessor rate monotonic scheduling and implications to Ada tasking are described in Sha [23]. This paper reviews the application of the generalized rate monotonic theory to a distributed real-time system. We review the essential elements of GRMS that are needed for the development of a distributed system,[1] discuss the system hardware and software architectural supports for using GRMS, and illustrate the application of GRMS in the design of a hypothetical distributed real-time system.

Section 2 describes a distributed real-time system model that will be used to illustrate the application of the theory in the rest of the paper. Section 3 reviews the basic elements of GRMS. Sections 4 and 5 describe architectural support for the application of GRMS. Section 6 illustrates the use of the theory. Section 7 has some concluding remarks.

[1] Additional examples and illustrations can be found in Sha [23].

2. A System Model

In this section we describe a simple model of a distributed real-time system that serves as a vehicle to illustrate GRMS theory. Figure 2-1 shows a distributed system consisting of several nodes connected by a network. Each node in the network is a multiprocessor. Each processor in the node has a CPU, memory and an operating system (OS). The processors communicate over a shared backplane bus. We assume that the OS and the backplane bus support priority scheduling. For example, the OS could be POSIX.4 [18], and the backplane could be Futurebus+ [6, 24]. The network could be a token ring [27] or a dual link network [25] that support GRMS. However, for this example we assume that the FDDI network is used [5].

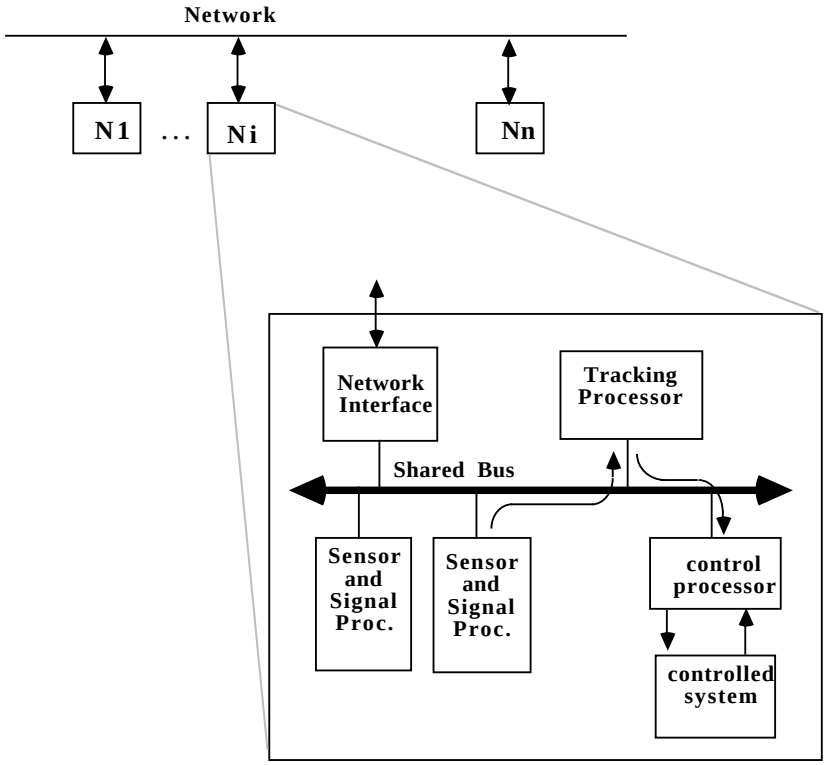

Figure 2-1: Block Diagram of Distributed System

Each node in the system consists of signal processors and control processors. In addition, nodes periodically send system status information periodically to a display node that interfaces with operators. An operator may send commands to nodes whenever the need arises. As each signal processor in a node is connected to a sensor. The results from each signal processor are periodically sent to a tracking processor, which is a high performance numeric

processor dedicated to tracking the motion of objects. The results from tracking are periodically sent over the bus to the control processor. The control processors are general purpose computers which perform feedback control tasks and communicates with operators via the network.

The architecture utilizes both tasking and message passing paradigms. Application software is partitioned into *allocation units,* each of which can be allocated to a processor. An allocation unit groups together closely related application functions implemented as tasks. Tasks within an allocation unit communicate via shared variables. Tasks in different allocation units communicate via messages. Allocation units can be freely relocated as long as the resulting configuration is still schedulable.

We will illustrate the scheduling of periodic and aperiodic tasks in a general purpose computer used as a control processor. In addition, we will analyze the scheduling of a remote server via the analysis of the tracking processor. Furthermore, we will examine the scheduling of messages across task allocation units within a processor and across the backplane and network.

3. Overview of Generalized Rate Monotonic Scheduling

In this section, we review basic results which allow us to design a distributed system with features described in Section 2. We begin with the scheduling of independent periodic and aperiodic tasks. We then address the issues of task synchronization and the effect of having task deadlines before the end of their periods.

3.1. Scheduling Independent Tasks

A periodic task τ_i is characterized by a worst case computation time C_i and a period T_i. Unless mentioned otherwise we assume that a periodic task must finish by the end of its period. Tasks are *independent* if they do not need to synchronize with each other. A real-time system typically consists of both periodic and aperiodic tasks. The scheduling of aperiodic tasks can be treated within the rate monotonic framework of periodic task scheduling:

> Example 1: Suppose that we have two tasks. Let τ_1 be a periodic task with period 100 and execution time of 99. Let τ_2 be a server for an aperiodic request that randomly arrives once within a period of 100. Suppose one unit of time is required to service one request. If we let the aperiodic server execute only in the background, i.e., only after completion of the periodic task, then the average response time is about 50 units. The same can be said for a polling server that provides one unit of service time in a period of 100. On the other hand, we can deposit one unit of service time in a "ticket box" every 100 units of time; when a new "ticket" is deposited, the unused old tickets, if any, are discarded. With this approach, no matter when the aperiodic request arrives during a period of 100, it will find there is a ticket for one unit of execution time at the ticket box. That is, τ_2 can use the ticket to preempt τ_1 and execute immediately when the request occurs. In this case, τ_2's response time is precisely one unit and the deadlines of τ_1 are still guaranteed.

This is the idea behind a class of aperiodic server algorithms [13] that can reduce aperiodic response time by a large factor (a factor of 50 in this example). We allow the aperiodic servers to preempt the periodic tasks for a limited duration that is allowed by the rate monotonic scheduling formula. An aperiodic server algorithm called the *Sporadic Server* that handles hard deadline aperiodic tasks is described in Sprunt [26]. Instead of refreshing the server's budget periodically, at fixed points in time, replenishment is determined by when requests are serviced. In the simplest approach, the budget is refreshed one period after it has been exhausted, but earlier refreshing is also possible.

A sporadic server is only allowed to preempt the execution of periodic tasks as long as its computation budget is not exhausted. When the budget is used up, the server can continue to execute at background priority if time is available. When the server's budget is refreshed, its execution can resume at the server's assigned priority. There is no overhead if there are no requests. Therefore, the sporadic server is especially suitable for handling emergency aperiodic events that occur rarely but must be serviced quickly.

An effective way to implement a sporadic server is as follows. When an aperiodic request arrives, the system registers the request time. The capacity consumed by this request is replenished one sporadic period from the request time. This replenishment approach guarantees that the aperiodic response time is no greater than the sporadic period, provided that the system is schedulable and sufficient server capacity is available, i.e., aperiodic demand request within a duration of the sporadic period is no more than the server capacity. In contrast, the longest possible response time for an aperiodic request serviced by a polling server is twice the period of the polling server. This occurs when the request arrives just after the poll, so that the server waits one period for the next poll and up to another period to complete its execution. From a schedulability viewpoint, a sporadic server is equivalent to a periodic task that performs polling, except that it provides better performance.

To determine if a set of independent periodic tasks is schedulable we introduce the following theorem [17]:

> **Theorem 1:** A set of *n* independent periodic tasks scheduled by the rate monotonic algorithm will always meet their deadlines for all task start times, if
>
> $$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n}-1)$$
>
> where C_i is the execution time and T_i is the period of task τ_i.
>
> C_i/T_i is the *utilization* of the resource by task τ_i. The bound on the utilization, $n(2^{1/n}-1)$, rapidly converges to *ln 2* = 0.69 as *n* becomes large.

The bound of Theorem 1 is very pessimistic because the worst-case task set is contrived and unlikely to be encountered in practice. The actual bound is for given task sets often over 90%. The remaining utilization can still be used by background tasks with low priority. To determine if a set of tasks with utilization greater than the bound of Theorem 1 can meet their deadlines, we can use an exact schedulability test based on the *critical zone* theorem (rephrased from Liu [17]):

> **Theorem 2:** For a set of independent periodic tasks, if a task τ_i meets its *first* deadline $D_i \leq T_i$, when all the higher priority tasks are started at the same time, then it can meet all its future deadlines with any task start times.

It is important to note that Theorem 2 applies to any static priority assignment, not just rate monotonic priority assignment. To check if a task can meet its first deadline we describe the following argument from Lehoczky [14]:

Consider any task τ_n with a period T_n, deadline $D_n \leq T_n$, and computation C_n. Let tasks τ_1 to τ_{n-1} have higher priorities than τ_n. Suppose that all the tasks start at time *t = 0*. At any time *t*, the total cumulative demand on CPU time by these *n* tasks is:

$$W_n(t)=C_1\left\lceil\frac{t}{T_1}\right\rceil + \cdots + C_n\left\lceil\frac{t}{T_n}\right\rceil = \sum_{j=1}^{n} C_j\left\lceil\frac{t}{T_j}\right\rceil$$

The term $\lceil t/T_j \rceil$ represents the number of times task τ_j arrives during interval [0, t] and therefore $C_j\lceil t/T_j \rceil$ represents its demand during interval [0, t]. For example, let T_1 = 10, C_1 = 5 and t = 9. Task τ_1 demands 5 units of execution time. When t = 11, task τ_1 has arrived again and has a cumulative demand of 10 units of execution.

Suppose that task τ_n completes its execution exactly at time t before its deadline D_n. This means that the total cumulative demand from the n tasks up to time t, $W_n(t)$, is exactly equal to t, that is, $W_n(t) = t$. A method for finding the completion time of task τ_i, that is, the instance when $W_i(t) = t$ is given in Figure 3-1.

Set $t_0 \leftarrow \sum_{j=1}^{i} C_j$

$t_1 \leftarrow W_i(t_0)$;
$t_2 \leftarrow W_i(t_1)$;
$t_3 \leftarrow W_i(t_2)$;
.
.
.
$t_k \leftarrow W_i(t_{k-1})$;

Stop when ($W_i(t_k) = t_k$)

Figure 3-1: Finding Minimum t, Where $W_i(t) = t$

We shall refer to this procedure as the *completion time* test. If all the tasks can be completed before their deadlines, then the task set is schedulable:

Example 2: Consider a task set with the following independent periodic tasks:

- Task τ_1: C_1 = 20; T_1 = 100; D_1 = 100;
- Task τ_2: C_2 = 30; T_2 = 145; D_2 = 145;
- Task τ_3: C_3 = 68; T_2 = 150; D_3 = 150;

The total utilization of tasks τ_1 and τ_2 is 0.41, which is less than 0.828, the bound for two tasks given by Theorem 1. Hence, these two tasks are schedulable. However, the utilization of these three tasks as given by Theorem 1 is 0.86, which exceeds 0.779, the bound, as given by Theorem 1 for the three tasks. Therefore, we need to apply the completion time test to determine the schedulability of task τ_3.

Figure 3-2 shows the time line for the execution of task τ_3. Since τ_1 and τ_2 must execute at least once before τ_3 can begin executing, the completion time of τ_3 can be no less than 118.

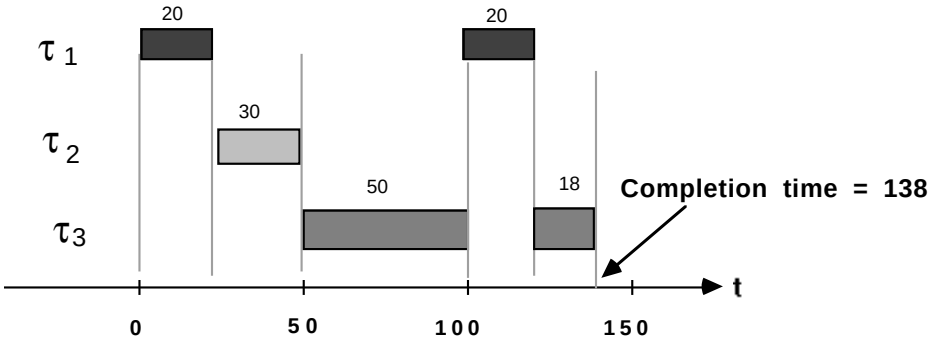

Figure 3-2: Application of Critical Zone Theorem to Task τ_3

$t_0 = C_1 + C_2 + C_3 = 20 + 30 + 68 = 118$

However, τ_1 is initiated one additional time in the interval (0,118). Taking this additional execution into consideration, $W_3(118) = 138$.

$t_1 = W_3(t_0) = 2C_1 + C_2 + C_3 = 40 + 30 + 68 = 138$

We find that $W_3(138)=138$, and thus the minimum time at which $W_3(t) = t$ is 138. This is the completion time of τ_3. Therefore, τ_3 completes its first execution at time 138 and meets its deadline of 150.

$W_3(t_1) = 2C_1 + C_2 + C_3 = 40 + 30 + 68 = 138 = t_1$

Hence, the completion time test determines that τ_3 is schedulable even though the test of Theorem 1 fails.

3.2. Task Synchronization

In the previous sections we have discussed scheduling of independent tasks. Tasks, however, do interact. In this section, we discuss how GRMS can be applied to real-time tasks that must interact. Common synchronization primitives include semaphores, locks, monitors, and Ada rendezvous. Although the use of these or equivalent methods is necessary to protect consistency of shared data or to guarantee the proper use of nonpreemptable resources, their use may jeopardize the system's ability to meet its timing requirements. In fact, a direct application of these synchronization mechanisms may lead to an indefinite period of *priority inversion*, which occurs when a high priority task is prevented from executing by a low priority task. Unbounded priority inversion can occur:

> Example 3: Let τ_1 and τ_3 share a resource and let τ_1 have a higher priority. Let τ_2 be an intermediate priority task that does not share any resource with either τ_1 or τ_3. Consider the following scenario:

1. τ_3 obtains a lock on the semaphore *S* and enters its critical section to use a shared resource,

2. τ_1 becomes ready to run and preempts τ_3. Next, τ_1 tries to enter its critical section by first trying trying to lock *S*. But *S* is already locked and hence τ_1 is blocked and moved from ready queue to the semaphore queue.

3. τ_2 becomes ready to run. Since only τ_2 and τ_3 are ready to run, τ_2 preempts τ_3 while τ_3 is in its critical section.

We would prefer that τ_1, being the highest priority task, be blocked no longer than the time for τ_3 to complete its critical section. However, the duration of blocking is, in fact, unpredictable. This is because τ_3 can be preempted by the medium priority task τ_2. As a result, task τ_1 will be blocked until τ_2 and any other pending tasks of intermediate priority are completed. The duration of priority inversion becomes a function of task execution times and is not bounded by the duration of critical sections.

The priority inversion problem can be controlled by a *priority ceiling protocol*. The *priority ceiling protocol* is a real-time synchronization protocol with two important properties [22].

Theorem 3: The priority ceiling protocol prevents mutual locks between tasks. In addition, under the priority ceiling protocol, a task can be blocked by lower priority tasks at most once.

The protocol works as follows: we define the *priority ceiling* of a binary semaphore *S* to be the highest priority of all tasks that may lock *S*. When a task τ attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceilings of all semaphores currently locked by tasks other than τ. If task τ is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking τ and hence inherits the priority of τ. As long as a task τ is not attempting to enter one of its critical sections, it will preempt every task that has a lower priority. The following example illustrates the deadlock avoidance property of the priority ceiling protocol:

Example 4: Suppose that we have two tasks τ_1 and τ_2 (see Figure 3-3). In addition, there are two shared data structures protected by binary semaphores, S_1 and S_2, respectively. Suppose task τ_1 locks the semaphores in the order S_1, S_2, while τ_2 locks them in the reverse order. Further, assume that τ_1 has a higher priority than τ_2. Since both τ_1 and τ_2 use semaphores S_1 and S_2, the priority ceilings of both semaphores are equal to the priority of task τ_1. Suppose that at time t_0, τ_2 begins execution and then locks semaphore S_2. At time t_1, task τ_1 is initiated and preempts task τ_2, and at time t_2, task τ_1 tries to enter its critical section by attempting to lock semaphore S_1. However, the priority of τ_1 is *not* higher than the priority ceiling of *locked* semaphore S_2. Hence, task τ_1 must be suspended without locking S_1. Task τ_2 now *inherits* the priority of task τ_1 and resumes execution. Note that τ_1 is blocked

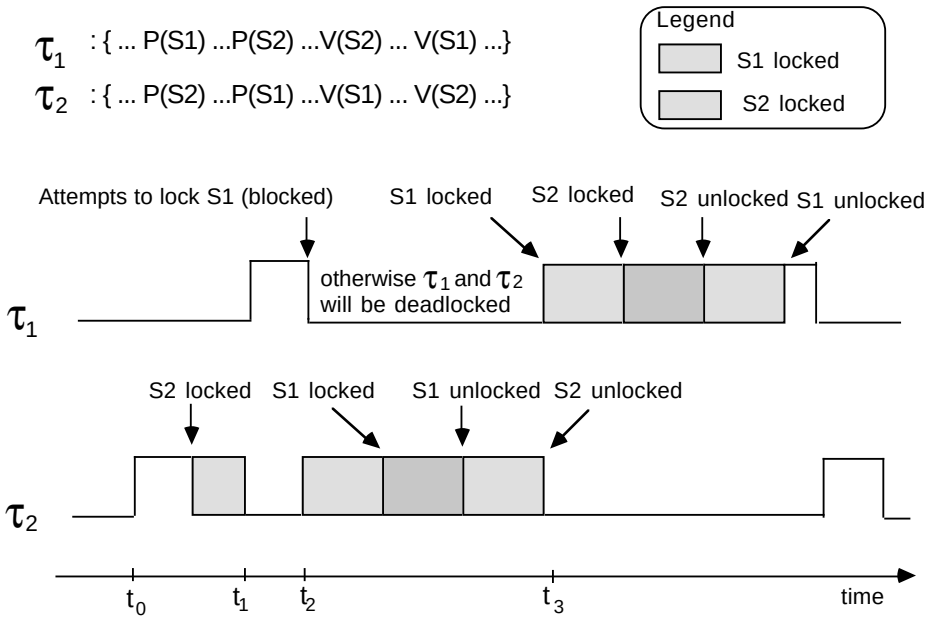

Figure 3-3: Example of Deadlock Prevention

outside its critical section. As τ_1 is not given the lock on S_1 but suspended instead, the potential deadlock involving τ_1 and τ_2 is prevented. Once τ_2 exits its critical section, it will return to its assigned priority and immediately be preempted by task τ_1. From this point on, τ_1 will execute to completion, and then τ_2 will resume its execution until its completion.

There is a simplified implementation of the the priority ceiling protocol called the priority ceiling emulation [23]. In this approach, once a task locks a semaphore, its priority is immediately raised to the level of the priority ceiling. The avoidance of deadlock and block-at-most-once results still hold, provided that a task is restricted from suspending its execution within the critical section.[2] The priority ceiling protocol has been extended to deal with dynamic deadline scheduling [3] and mixed dynamic and static priority scheduling [2].

The schedulability impact of task synchronization can be assessed as follows: Let B_i be the duration in which task τ_i is blocked by lower priority tasks. The effect of this blocking can be modeled as though task τ_i's utilization is increased by an amount B_i/T_i.

Sometimes, a task τ_i's deadline, D_i, is before the end of period. Theorem 1 was generalized to accommodate an earlier deadline. Let $\Delta_i = (D_i/T_i)$ [14].

Theorem 4: A set of *n* periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if

[2]The full implementation permits tasks to suspend within a critical section.

$$\forall\, i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} \le U(\Delta_i).$$

$$where U(\Delta_i) = \begin{cases} i((2\Delta_i)^{1/i} - 1) + 1 - \Delta_i, & 0.5 \le \Delta_i \le 1 \\ \Delta_i, & 0 \le \Delta_i \le 0.5 \end{cases}$$

The completion time test can be directly used in the case when deadlines are shorter than end of period with no modification. To accommodate blocking, we can simply add the blocking to the execution time of the task.

So far, the task priority assignment follows the rate monotonic priority assignment, that is, the shorter the period, the higher the priority. Note that this is a special case of giving tasks with narrower completion windows to complete higher priorities, since the period is the window for completion when the deadline is at the end of the period. Sometimes, tasks may have deadlines earlier than the end of periods. In generalized rate monotonic scheduling, tasks with narrower completion windows are given higher priority. Leung and Whitehead called this generalized method as deadline monotonic algorithm [16]. They showed that this generalized method is optimal for independent tasks with completion time windows less than or equal to the periods. The use of deadline monotonic priority assignment will be illustrated in Section 6.

4. Scheduling Considerations in Hardware Architecture

In this section, we examine important architectural support necessary for the use of generalized rate monotonic scheduling. Since GRMS is a priority based scheduling algorithm, the system must have an adequate number of priority levels that can be assigned to tasks, and must be free from pitfalls that lead to unbounded priority inversion.

4.1. Number of Priority Levels

The number of priority levels that can be supported in software by an operating system is essentially unlimited. In contrast, the number of priority levels that can be supported by hardware on a backplane and network is limited and therefore is an important design consideration.

When fewer priority levels are available than the number needed by the priority scheduling algorithm, the schedulability of a resource is lowered [12]. In such a case, the loss of schedulability can be reduced by employing a *constant ratio* priority grid for priority assignments. Consider a range of the task periods such as 1 msec to 100 second. A constant-ratio grid divides this range into segments such that the ratio between every pair of adjacent points is the same. An example of a constant ration priority grid is {L_1 = 1 msec, L_2 = 2 msec, L_3 = 4 msec, ...} where there is a constant ratio of 2 between pairs of adjacent points in the grid.

With a constant ratio grid, a distinct priority is assigned to each interval in the grid. For example, all tasks with periods between 1 to 2 msec will be assigned the highest priority, all tasks with periods between 2 to 4 msec will have the second highest priority and so on when using the rate-monotonic algorithm. It has been shown [12] that a constant ratio priority grid is effective only if the grid ratio is kept smaller than 2. For the rate-monotonic algorithm, the percentage loss in worst-case schedulability due to the imperfect priority representation is (1 – Relative Schedulability), which can be computed by the following formula [12]:

$$\text{Relative Schedulability} = (\ln(2/r) + 1 - 1/r)/\ln 2$$

where r is the grid ratio.

Figure 4-1 plots schedulability as a function of priority bits, relative to schedulability with as many priority levels as needed [24]. It was assumed that the ratio between the shortest and longest periods is 100,000. As can be seen, the schedulability loss is negligible with 8 encoded priority bits, which corresponds to 256 priority levels. In other words, the worst-case schedulablility obtained with 8 priority bits is close to that obtained with an unlimited number of priority levels. In many older computer backplane buses, in addition to the lack of an adequate number of priority levels, a board is given a fixed priority level. As a result, a processor cannot access the bus according to the priority of tasks or messages. Fortunately, both these problems are solved by recent bus standards such as the IEEE Futurebus+, whose real-time computing option directly supports the use of GRMS [24].

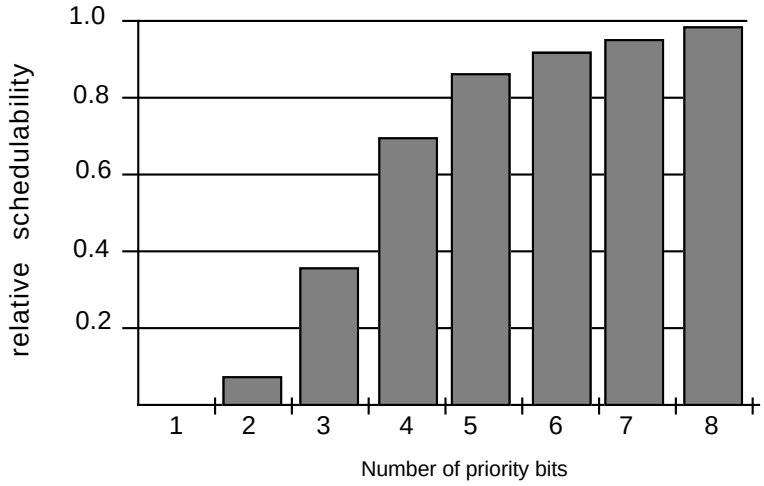

Figure 4-1: Relative Schedulability vs. The Number of Priority Bits

4.2. Multi-processor Backplane Scheduling

In addition to a sufficiently large number of priorities, it is necessary to have consistent treatment of priorities throughout the arbitration, message passing and DMA protocols. In order to support the GRMS theory, a module must request the bus based on the priority of the task or message that needs the bus. This means that the same module may request the bus at many different priorities under software control. The IEEE Futurebus+ supports such a software controlled priority arbitration paradigm. Unfortunately, most existing bus architectures staticly bind a single priority to each module.

Software controlled priority arbitration can be supported by either a distributed arbiter or a centralized arbiter [24]. In this paper, we limit ourselves to a description of the centralized arbiter because it is easier to implement. Furthermore, it can be easily used to override existing bus arbitration schemes. This maintains logical compatibility with existing products for that bus, while supporting real-time computations, such as multi-media applications. We now provide an overview of the central arbiter.

In the central arbiter model, each module contact the central arbiter via a request line, a grant line and a preemption line. Ideally, 8 encoded priority bits can be used for real-time applications. The centralized arbiter has a priority register for each module. A module that needs the bus asserts the request line. The central arbiter grants the bus to the requesting module with the highest priority value in the corresponding priority register by asserting the module's grant line. The priority level of a module can be changed by communicating its priority via a separate serial line. To minimize the number of lines, one can use the request

line as the serial line. In this case, whenever a module needs the bus, it sends its 8 bit priority code over the request line.

If there is an active bus master and the central arbiter receives a higher priority request, the arbiter asserts the preempt line on the current bus master. The module may then voluntarily suspend its ongoing transaction at a logical boundary and yield to the pending higher priority request. This scheme is particularly useful in real-time systems when a long block transfer at low priority can be preempted by a higher priority transaction. The preempted transaction can resume at a later time after a fresh request/grant cycle. If no priority change requests are transmitted, subsequent requests will have the same priority level as the last specified value from that module. This model allows each request from a module to have a different priority level, or all requests from the same module to have the same priority level.

4.3. Hardware Queues

To support message passing over a communication medium such as a backplane bus or a network, FIFO hardware queues are commonly used for both transmission and reception. Hardware FIFO queues at the receiver are acceptable if the software empties the entire queue and re-orders the messages in priority order before processing.

In contrast, a short transmission priority queue can lead to unbounded priority inversion. For example, suppose the transmission priority queue of node *A* connected to a backplane bus is filled with low priority messages. If node *A* wishes to transmit a high priority message, it cannot even enter the high priority message in its transmission queue since it is full. Also, unbounded priority inversion can occur because medium priority messages from other nodes prevent the servicing of node *A*'s queue, indefinitely holding off node *A*'s high priority message.

A practical solution is the use of a short priority queue with priority overwrite to emulate an ideal priority queue [24]. When the transmission queue is full and a higher priority message waits at the host, the higher priority message overwrites the lowest priority message in the queue. To prevent the potential loss of this lower priority message, the host must preserve each message in memory until it is successfully transmitted. The overwrite occurs at the tail of the queue and can occur concurrently with transmission from the head of the queue, thus incurring very little performance penalty.

5. Scheduling Considerations in Software Architecture

The primary objective in our software architecture considerations is to decouple the scheduling of resources. We want to analyze the scheduling of each processor in a multi-processor as if it were a stand-alone uni-processor. We also want to use the same analysis technique for message scheduling across the backplane or network. Decoupling the resources allows us to "divide and conquer" the scheduling problem and simplifies software development.

Our tool to decouple the system scheduling problem is the use of allocation units mentioned in Section 2. The use of allocation units avoids the need to synchronize distributed tasks that share variables. This greatly reduces the complexity in schedulability analysis.[3]

Another high level consideration is the need of a system wide priority scheme. The GRMS specifies only the priority order of tasks and messages and does not dictate a particular priority encoding method. However, priority encoding with local scope is not advisable. For example, consider tasks τ_1 and τ_2 with periods of 10 and 20 respectively in one allocation unit. Consider task τ_3 and τ_4 with periods 15 and 40 another allocation unit. Let priority encoding be local to allocation units. Let τ_1 and τ_2 be assigned priorities 10 and 20 in the first allocation unit, and τ_3 and τ_4 be assigned priorities 50 and 60 in the other allocation unit (larger priority numbers mean lower priority). If these two allocation units are relocated into the same processor, then rate monotonic assignment is violated since the task with a period of 15 gets a lower priority than the task with a period of 20. Hence, priorities must be assigned on a system wide basis.

In the rest of this section we consider the message passing architecture and discuss the remote server paradigm.

5.1. Message Passing Architecture

A message passing architecture for communication between allocation units may be described as follows. The sending task passes a message descriptor to the operating system (OS). The descriptor consists of a pointer to the data buffer, the sender's identification and the receiver's identification. If the receiver is in the same processor, the operating system just puts the message descriptor into the receiver's mailbox.

If the receiver is another processor connected to the same backplane, the OS assigns a bus priority to the message and passes it to the message scheduler at the bus interface unit that schedules messages on the bus. The receiving processor's bus interface unit orders the messages it receives according to their bus priority, and interrupts the receiver's OS. The OS parses the messages and determines the receiving task. The OS then constructs a message descriptor and puts it in the receiving task's mailbox.

[3] Readers interested in using shared variables across processors are referred to [19, 20].

If the receiver is across the network, then the sender sends the message across the bus to the network interface processor as described above. The OS of the network interface processor is responsible for sending the message over the network. The OS packetizes the message according to the network protocol, attaches a priority to each packet, and schedules the packets for transmission. The receiving network interface processor sends the received message across the bus as described before. The assignment of message priority for the bus is illustrated in Section 6.

5.2. Scheduling Remote Servers

Clients and remote servers are commonly used paradigms in distributed computation. Several servers can be co-located in a processor. Each server package provides a particular set of functions that can be used over the network or bus by clients. Ideally, a server for real-time applications should be multi-threaded so that high priority requests can preempt a low priority request. However, it is often impractical to provide as many threads as needed. One simple solution is to first write the service procedures and then create a few tasks each of which have a well defined response time. For example, we can have three sporadic tasks with 100 ms, 300 ms and 600 ms response time. Each task will call the same service procedures,

There is great incentive to use existing server packages whenever possible. However, servers developed for commercial use are often single threaded and come with built-in FIFO queues. Fortunately, we can still emulate the priority ceiling protocol [22], which ensures that a high priority request may wait for at most one lower priority request even if the request visits multiple servers co-located in the processor. The priority ceiling protocol emulation can be implemented as follows:

The priority ceiling of a server is defined as the highest priority request that may ever use that server. Server priority ceilings can be equal to each other since a server will always execute at the ceiling priority level. A centralized request dispatcher is necessary; otherwise, unbounded priority inversion will occur if requests are inserted directly into server queues, even if the queues are prioritized. For example, let S_M and S_H be servers with medium and high priority ceilings respectively. Let S_M have a medium priority request pending, and let S_H's queue be filled with low priority requests. S_H will keep serving the low priority requests at its queue since it executes at higher priority.

The dispatcher maintains two priority queues. The Request Queue (RQ) maintains all the servers' pending requests, and the Active Server Record Queue (AQ) maintains the records of all active servers according to their ceilings. When a request is dispatched to a server, the server becomes active, and the record of the server is inserted in the AQ. When the server completes a request it becomes inactive and its record is removed from the AQ.

- The dispatcher runs at a priority level that is higher than all application tasks and servers.

- The dispatcher compares the head of the RQ with the head of the AQ. If the request at the head of the RQ has a higher priority than the server record at the head of the AQ, the dispatcher sends the request to the requested server.

- In case a request needs to visit more than one server it will be returned to the dispatcher and sent to the next requested server until done.

- The dispatcher suspends when either the RQ is empty, or no requests can be forwarded. The dispatcher wakes up when any server becomes inactive or when there is a new request inserted into the RQ.

Under priority ceiling protocol emulation, the server FIFO queue will not lead to priority inversion since the queue of any server has at most a single request that is being serviced.

> Example 5: Let S_M and S_H be servers with medium and high priority ceilings respectively. Let S_M be active and serving a medium priority request and let S_H be inactive. Hence the record of S_M is at the head of AQ. Low priority requests that arrive during this time cannot be forwarded to any server, because the request priority at the head of the RQ is lower than the priority of the server record at the head of AQ. On the other hand, if a high priority request for S_H arrives, it will be forwarded to server S_H immediately since the request's priority is higher than the record of S_M. As a result, S_H preempts S_M and starts serving the high priority request.

Sometimes, it may be difficult to determine the server ceilings. The default is to assign all server ceilings to the highest priority. However, the priority ceiling assignment can be refined. For example, one can have a particular "emergency only" server with ceiling higher than that of all the normal servers. In this way, the emergency requests never have to wait for normal requests. If a server's execution time is particularly long, one may decompose the request of a big job into multiple small jobs when possible. The server will then execute each small job at a time and send it back for the dispatcher for rescheduling.

6. Example Application

In this chapter, we apply the preceding theory to a concrete example. Consider the system in Figure 2-1. We assume that the priority ceiling protocol is used for task synchronization, and the message passing architecture is used for communication. We assume that we use a prioritized backplane such as the IEEE Futurebus+. However, the network used is the FDDI network, since currently no standard network fully supports GRMS.

Let the characteristics of the application be as follows: The unit of time in this example is milliseconds. Referring to Figure 2-1, the sensor takes an observation every 40. To reduce unnecessary bus traffic, the signal processing task processes the signal and averages it every 4 cycles before sending it to the tracking processor. The tracking processor has a task with a period of 160. After the task executes, it sends the result to the control processor. Task τ_3 on the control processor that uses the tracking information has a computation requirement of 30 and a period of 160. In addition, the end-to-end latency of the pipeline of data flow from the sensor to the control processor should be no more than 785. The control processor also has additional periodic and aperiodic tasks which must be scheduled. The tracking and control processors send status information across the network to a user interface node and periodically receive commands.

Let the task set on the control processor be specified as given below:

- Aperiodic event handling with an average execution time of 10 and an average interarrival time of 100. We create a sporadic server task as follows: Task τ_1: $C_1 = 20$; $T_1 = 100$;
- A periodic task for handling local feedback control with a computation requirement and a given period. Task τ_2: $C_2 = 78$; $T_2 = 150$;
- A periodic task that utilizes the tracking information received. Again the computation time and period are given. Task τ_3: $C_3 = 30$; $T_3 = 160$;
- A periodic task responsible for reporting status across the network with a given computation time and period. Task τ_4: $C_4 = 10$; $T_4 = 300$;

Tasks τ_1 and τ_2 are in one allocation unit and Tasks τ_3 and τ_4 are in another unit. Note that the scheduling of tasks in a processor is independent of allocation units.

6.1. Assigning Message and Task Deadlines

When a message is sent within a processor, it can be implemented by passing a message pointer to the receiving task and therefore can be treated as any other OS overhead. However, when a message is sent outside the processor boundary, an integrated approach to assign message and task deadlines needs to be developed. Consider the situation in Figure2-1:

- The sensor takes an observation every 40.

- The signal processing task processes the signal, averages the result every 4 cycles, and sends it to the tracking processor every 160.

- The tracking processor task executes with a period of 160. It then sends a message to the control processor.

- Task τ_3 on the control processor that uses the tracking information has a computational requirement of 30 and a period of 160, as given above. Recall that the end-to-end latency for the control processor to respond to a new observation by the sensor needs to be less than 785.

The steps involved in integrated priority assignment are as follows: First we try to use the rate monotonic priority assignment. Since rate monotonic analysis guarantees end-of-period deadlines, we assume that the end-to-end delay is the sum of the period for each resource. Since the signal processor averages four cycles, each 40 long, its delay is up to 160. Each of the other resources has a delay of up to one period which is 160. That is, the total delay using rate monotonic scheduling is bound by 4*40 + 160 + 160 + 160 + 160 = 800. If it were less than the allowable delay then rate monotonic priority assignment could be used for all the resources. However, the specified maximum allowable latency is 785. Therefore, we may need to use deadline monotonic scheduling for at least some of the resources in the path. From a software engineering viewpoint, it is advisable to give a full period delay for global resources such as the bus or the network since their workload is more susceptible to frequent changes. Since there are two bus transfers involved we attempt to assign a full period to each. We also attempt to assign a full period to the signal and tracking processors. Hence, the required completion time of the control processor task τ_3 should be no greater than $785 - 4 \times (160) = 145$.

6.2. Scheduling Tasks on the Control Processor

In this section we apply the scheduling theory to the control processor tasks. Let tasks τ_1, τ_2, and τ_3 share several data structures guarded by semaphores S_1, S_2, and S_3. Suppose the duration of critical sections accessing shared data structures are bounded by 10. Suppose the priority ceiling protocol is used. Then by Theorem 3, higher priority tasks are blocked at most once for 10 by lower priority tasks.

The task set on the control processor, is as described earlier with task τ_3 modified to have a deadline of 145. We check whether or not τ_3 completes within 145 under rate monotonic priority assignment. Under rate monotonic assignment, the completion of τ_3 is:

$$t_0 = C_1 + C_2 + C_3 = 20 + 78 + 30 = 128$$

$$t_1 = W_3(t_0) = 2C_1 + C_2 + C_3 = 40 + 78 + 30 = 148$$

$$W_3(t_1) = 2C_1 + C_2 + C_3 = 148 = t_1$$

Therefore, the completion time of τ_3 is 148. In order to meet the deadline of 145 imposed by the maximum allowable latency requirement of the previous section, we use the deadline-monotonic priority assignment. This makes the priority of task τ_3 higher than that of task τ_2, which has an end-of-period deadline of 150.

The schedulability of each task can be checked as follows: Task τ_1 can be blocked by lower priority tasks for 10 , i.e., B_1 = 10. The schedulability test for task τ_1 is a direct application of Theorem 4:

$$\frac{C_1}{T_1} + \frac{B_1}{T_1} = 0.2 + 0.1 = 0.3 \leq 1(2^{1/1} - 1) = 1.0$$

The sporadic server task τ_1 is schedulable. The average response time for aperiodic events handled by τ_1 can be calculated as follows: The server capacity is 20% (20/100) and the average aperiodic workload is 10% (10/100). Referring back to the ticket box analogy of Example 1, because most of the aperiodic arrivals can find "tickets," we would expect a good response time. Simulation indicates the average response time is about 20.

Task τ_3 is the second highest priority task. Since τ_3 has a deadline shorter than its period, the schedulability test for τ_3 can be checked as given in Theorem 4. Here $\Delta_3 = (D_3/T_3)$ = 145/150 = 0.967. Also, in the schedulability test of τ_3, the utilization of task τ_2 does not appear, since τ_2 has a lower priority and does not preempt τ_3. Because τ_2 has a lower priority, its critical section can delay τ_3 by 10 . Therefore B_3 = 10.

$$\frac{C_1}{T_1} + \frac{C_3}{T_3} + + \frac{B_3}{T_3} = 0.2+0.188+0.0625 = 0.4505 \leq 2((2\Delta_3)^{1/2} - 1) = 0.781$$

Now consider the third highest priority task τ_2. From the view point of the rate monotonic assignment, the deadline monotonic assignment is a "priority inversion". Therefore in the schedulability test for task τ_2, the effect of blocking has to include τ_3's execution time. The blocking time is $B_2 = C_3+0$. The zero indicates that there can be no lower priority task blocking τ_2.

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} = 0.2+0.52+0.2 = 0.92 > 2(2^{1/2} - 1) = 0.828$$

The schedulability test of Theorem 4 fails for τ_2. The schedulability of τ_4 can be checked by the following simple test since there is neither blocking or deadline before its end of period.

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + \frac{C_4}{T_4} = 0.2+0.52+0.188+0.033 = 0.941 > 4(2^{1/4} - 1) = 0.757$$

Note that the schedulability test of Theorem 4 fails for both tasks τ_2 and τ_4. To determine their schedulability we use the completion time test. Since τ_1 and τ_3 must execute at least once before τ_2 can begin executing, the completion time of τ_2 can be no less than 128:

$$t_0 = C_1 + C_2 + B_2 = 20 + 78 + 30 = 128$$

However, τ_1 is initiated one additional time in the interval (0,128). Taking this additional execution into consideration, $W_2(128) = 148$:

$$t_1 = W_2(t_0) = 2C_1 + C_2 + B_2 = 40 + 78 + 30 = 148$$

Finally, we find that $W_2(148)=148$ and thus the minimum time at which $W_2(t) = t$ is 148. This is the completion time for τ_2. Therefore τ_2 completes its first execution at time 148 and meets its deadline of 150.

$$W_2(t_1) = 2C_1 + C_2 + B_2 = 40+ 78 + 30 = 148 = t_1$$

Similarly, we can check the schedulability of task τ_4 using the completion time test. It turns out to be schedulable.

6.3. Scheduling Messages across a Network

In previous sections we have discussed system support for scheduling resources. The importance of such support is that it permits us to treat the scheduling of all resources in a uniform way. That is, we can analyze scheduling of any resource, similar to processor scheduling. The total end to end delay is then be bounded by the sum of delays at each resource. At this writing, the support of GRMS scheduling can be found in standard processors and OS such POSIX.4 [18] and in multi-processor backplanes such as IEEE Futurebus+ [6], but not in standard networks.

The problem of guaranteeing message deadlines in the FDDI network has been addressed in [1]. In a simple token passing protocol, the amount of time between consecutive token visits may be unbounded. Due to this, deadline guarantees cannot be made. FDDI employs a timed token protocol that results in a bounded token rotation time. There is a protocol parameter called Target Token Rotation Time (TTRT) which is negotiated at network initialization. Sevcik and Johnson [21] have shown that the time between two consecutive token visits is bounded by 2*TTRT. A node in the FDDI protocol can transmit in either synchronous or asynchronous mode. Without describing the details of these modes, we observe that time critical messages should use the synchronous mode. In a network that uses only synchronous mode, each station can transmit once every TTRT for an amount equal to an assigned synchronous capacity H_i.

To support real-time applications the FDDI network should be appropriately configured. That is, each station only uses a preallocated portion of the network bandwidth on every token arrival. The capacity to each station is allocated proportionally using the following formula [1]:

$$H_i = \frac{U_i}{U}(TTRT - D)$$

where H_i is the capacity allocated to station S_i. U_i is the network bandwidth utilized by station S_i and $U=U_1+ \ldots +U_n$. *TTRT* is the target token rotation time and *D* is the walk time (the token round trip propagation delay when the network is idle).

This allocation can be directly realized using only synchronous mode of transmission. However, it can also be realized when using only the asynchronous mode of transmission, provided each station only transmits a preallocated number of frames on every token arrival. Finally, message queues should be prioritized.

Suppose we have three periodic messages to be transmitted on an FDDI network with the default TTRT of 8ms. Let the token propagation delay D be 1ms:

- Message τ_1: $C_1 = 7$; $T_1 = 100$;
- Message τ_2: $C_2 = 10$; $T_2 = 145$;
- Message τ_2: $C_3 = 15$; $T_2 = 150$;

In the utilization of the above message set, $U = 0.239$.

Applying the above formula $H_1 = 2.05$, $H_2 = 2.02$, and $H_3 = 2.93$. The proportional allocation scheme is directly supported by the synchronous mode of operation.

The schedulability analysis can be carried out as follows. Let there be at least four message priority levels. The messages are first processed in the Network Operating System (NOS), that executes on the end stations. The total application level delay is the sum of the processing delay at the sender's NOS, the delay in the FDDI ring, and the processing delay in the receiver's NOS.

The requirement on the application level delays are given in the table below: There are four message types with following timing requirements:

Message	*Type*	*Average Latency*
M_1	Emergency	21ms
M_2	Alert	24ms
M_3	Fast	30ms
M_4	Normal	48ms

Table 6-1: Latency Metrics

To meet the timing requirements, polling or sporadic servers for each level can be created. For example, to meet the average timing requirement, four polling servers can be created with periods T_1, T_2, T_3, and T_4, given by 7 ms, 8 ms, 10 ms, and 16 ms, respectively. Each server has a full period at the sending NOS, the FDDI and the receiving NOS. If the total traffic is schedulable according to the RMS formula, then we expect the delays will be under 21 ms, 24 ms, 30 ms, and 48 ms most of time. The absolute worst case are 42 ms, 48 ms, 60 ms, and 96 ms since polling guarantees a responsiveness of twice the period. If sporadic server is used, the worst case performance will be 21 ms, 24 ms, 30 ms, and 48 ms.

The structure for schedulability analysis is as follows: We consider four message processing tasks at the NOS level. Let task M_i have a processing requirement of C_i per period T_i. This

is determined by the number of messages the task processes per period. For example, if the processing of one message in task M_1 takes 0.5 ms, and there are three messages to be processed per period, then C_1 = 1.5. Since the NOS executes on the host processor, the message processing tasks (C_1, T_1), (C_2, T_2), (C_3, T_3), and (C_4, T_4) are scheduled along with other application tasks if any. The schedulability analysis is the standard rate monotonic analysis for processor. The schedulability analysis of the tasks in the receiving NOS can be analyzed similarly.

The approach to message scheduling on FDDI can be as follows: There are four message transmission tasks with transmission time C_i for task with a period of T_i. For example, if 4 Kbyte packets are used, each packet will take 0.33 ms to transmit. If a message task M_1 has to transmit 3 packets, its transmission time C_1 is 0.99 ms.

Consider the scheduling of messages in a particular station S_i. Let the capacity allocated to the station be H_i. The station can transmit for up to H_i ms every TTRT. This can be treated as having another high priority task with message transmission time (TTRT – H_i) and period TTRT. We refer to this task as Token Rotation task M_{tr}. If TTRT = 6 ms and H_i = 2 ms, then M_{tr} = (C_{tr} = 4, T_{tr} = 6). The task set for station S_i is then (4, 6),(C_1, T_1), (C_2, T_2), (C_3, T_3), and (C_4, T_4). This task set can be analyzed in the standard rate monotonic framework.

Finally, note that although the Token rotation task behaves like the highest priority task at each station, it actually may be comprised of transmission of lower priority messages from other stations. In this sense, it is a priority inversion and limits the schedulable utilization of the network.

7. Conclusion

In this paper, we have described the use of generalized rate monotonic scheduling theory for the design and analysis of a distributed real-time system. We have reviewed the basic elements of the theory and provided references for further study. We have described hardware architectural support such required number of priority levels, and the design of hardware queues. We have described several important software techniques such as message passing interface between tasks, the scheduling of remote servers and management of transient overload. Finally, we have provided an application example to illustrate the assignment of message and task deadlines, task scheduling and message scheduling.

8. Acknowledgement

The authors want to thank Mark Klein and John Goodenough for their suggestions and comments.

References

1. Agrawal, G., Chen, B., Zhao, W., Davari, S. "Guaranteeing Synchronous Message Deadlines in High Speed Token Ring Networks with Timed Token Protocol". *To appear in the Proceedings of IEEE International Conference on Distributed Computing Systems* (1992).

2. Baker, T. "Stack-Based Scheduling of Realtime Processes". *Journal of Real-Time Systems 3*, 1 (March 1991), 67--100.

3. Chen, M., Lin, K.J. "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-time Systems". *Journal of Real-Time Systems 2*, 4 (November 1990), 325--346.

4. ESA. "Statement of Work, Hard Real-Time OS Kernel". *On-Board Data Division, European Space Agency* (July, 1990).

5. *FDDI Token Ring Media Access Control -- ANSI Standard X3T9.5/83-16 .* 1986.

6. *Futurebus P896.1,2,3 Specifications.* IEEE, 345 East 47th St., New York, NY 10017, 1991. P896 Working Group of the Microprocessor Standards Committee.

7. Gafford, J. D. "Rate Monotonic Scheduling". *IEEE Micro* (June 1990).

8. Harbour, M. G., Klein M. H., and Lehoczky, J. P. "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority". *Proceedings of IEEE Real-Time Systems Symposium* (December 1991).

9. Katcher, D., Arakawa, H., and Strosnider J. K. Engineering and Analysis of Fixed Priority Schedulers. Tech. Rept. CMUCDS-91-10, Center for Dependable Systems, Carnegie Mellon University, Pittsburgh, PA, December 1991.

10. Kirk, D. and Strosnider, J. K. "SMART (Strategic Memory Allocation for Real-Time) Cache Design Using MIPS R3000". *Proceedings of IEEE Real-Time Systems Symposium* (1990).

11. Klein, M. H., and Ralya, T. An Analysis of Input/Output Paradigms for Real-Time Systems. Tech. Rept. CMU/SEI-90-TR-19, ADA226724, Software Engineering Institute, July 1990.

12. Lehoczky, J. P. and Sha, L. "Performance of Real-Time Bus Scheduling Algorithms". *ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1* (May, 1986).

13. Lehoczky, J. P., Sha L and Strosnider, J. "Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment". *IEEE Real-Time System Symposium* (1987).

14. Lehoczky, J.P., Sha, L., and Ding, Y. "The Rate Monotonic Scheduling Algorithm --- Exact Characterization and Average Case Behavior". *Proceedings of IEEE Real-Time System Symposium* (1989).

15. Lehoczky, J. P. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines". *IEEE Real-Time Systems Symposium* (December 1990).

16. Leung, J. and Whitehead, J. "On the complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks". *Performance Evaluation 2* (1982).

17. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM 20 (1)* (1973), 46 - 61.

18. *IEEE Standard P1003.4 (Real-time extensions to POSIX).* IEEE, 345 East 47th St., New York, NY 10017, 1991.

19. Rajkumar, R., Sha, L., and Lehoczky, J.P. Real-Time Synchronization Protocols for Multiprocessors. Proceedings of the Real-Time Systems Symposium, IEEE, Huntsville, AL, December, 1988, pp. 259-269.

20. Rajkumar, R. "Real-Time Synchronization Protocols for Shared Memory Multi-Processors". *Proceedings of The 10th International Conference on Distributed Computing* (1990).

21. Sevcik, K.C., and Johnson. M.J. "Cycle Time Properties of the FDDI Token Ring Protocol". *IEEE Transactions on Software Engineering SE-13 No. 3 pp 376-385* (1987).

22. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transaction On Computers* (Sept., 1990).

23. Sha, L. and Goodenough, J. B. "Real-Time Scheduling Theory and Ada". *IEEE Computer* (Apr., 1990).

24. Sha, L., Rajkumar, R., and Lehoczky, J. "Real-Time Applications Using IEEE Futurebus+". *IEEE Micro* (June 1990).

25. Sha, L., Sathaye, S., and Strosnider J. K. Analysis of Reservation Based Dual Link Networks for Real-Time Applications. Software Engineering Institute, March 1992.

26. Sprunt, B., Sha, L., and Lehoczky, J. P. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems 1* (1989), 27-60.

27. Strosnider, J. K. and Marchok, T. E. "Responsive, Deterministic IEEE 802.5 Token Ring Scheduling". *Journal of Real-Time Systems 1* (1989), 133--158.

28. Ramos-Thuel, S., Strosnider J. "The Transient Server Approach to Scheduling Time-Critical Recovery Operations". *Proceedings of IEEE Real-Time Systems Symposium* (December 1991).

29. Tokuda, H., Sha, L. and Lehoczky, J. P. "Towards Next Generation Distributed Real-Time Operating Systems". *Abstracts of IEEE Fourth Workshop on Real-Time Operating Systems* (1987).