Generalized Rate-Monotonic Scheduling Theory:

A Framework for Developing Real-Time Systems

Lui Sha
Ragunathan Rajkumar
Software Engineering Institute[1]
Carnegie Mellon University, Pittsburgh PA 15213
lrs@sei.cmu.edu, (412) 268-5875
rr@sei.cmu.edu, (412) 268-8707

Shirish S. Sathaye[2]
Electrical & Computer Engineering
Carnegie Mellon University, Pittsburgh PA 15213
sathaye@austria.ece.cmu.edu, (412) 268-7102

15 February 1993

Keywords: Real-Time Scheduling, Distributed Real-Time System, Generalized Rate-Monotonic scheduling, Software Architecture.

Abstract:

Real-time computing systems are used to control telecommunication systems, defense systems, avionics and modern factories. Generalized rate-monotonic scheduling theory is a recent development that has had large impact on the development of real-time systems and open standards. In this paper we provide an up-to-date and self-contained review of generalized rate-monotonic scheduling theory. We show how this theory can be applied in practical system development, where special attention must be given to facilitate the concurrent development by geographically distributed programming teams and the reuse of existing hardware and software components.

---

[1]The Software Engineering Institute is sponsored by the U.S. Department of Defense (DoD). This work is funded in part by the Office of Naval Research.

[2]The author is with Digital Equipment Corporation, and is currently a Ph.D. candidate at Carnegie Mellon University supported by Digital Equipment Corporation's Graduate Engineering Education Program.

1. Introduction

Real-time computing systems are critical to an industrialized nation's technological infrastructure. Modern telecommunication systems, factories, defense systems, aircraft and airports, space stations and high energy physics experiments cannot operate without them. Indeed, real-time computing systems control the very systems that keep us productive, and enable us to explore new frontiers of science and engineering.

In real-time applications, the correctness of a 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.

Real-time computing is a vibrant field. Several important research efforts are summarized in [24] and [25]. Among them, Generalized Rate-Monotonic Scheduling (GRMS) theory is a useful framework that 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 GRMS approach is cited in the Selected Accomplishments section of the National Research Council's 1992 report, A Broader Agenda for Computer Science and Engineering. The Acting Deputy Administrator of NASA, Aaron Cohen, stated in a 1992 speech titled *Charting The Future*, "Through the development of Rate-Monotonic Scheduling, we now have a system that will allow (Space Station) Freedom's computers to budget their time, to choose between a variety of tasks, and decide not only which one to do first but how much time to spend in the process." DoD's 1991 Software Technology Strategy refers to GRMS as a "major payoff" and states that "system designers can use this theory to predict whether task deadlines will be met long before the costly implementation phase of a project begins. It also eases the process of making modifications to application software."

To apply GRMS in the real-world, some practical problems must be addressed. For example, the development of large, complex systems requires several teams of programmers, often distributed across several geographical regions. To facilitate the practice of concurrent engineering, we introduce the principle of decoupling in our software architecture. Firstly, when possible we structure the system in such a way that the scheduling of each resource can be viewed as if it is a stand alone resource. Secondly, we separate the software constructs dealing with timing (e.g. tasking) from constructs that provide the functions. This helps to contain the impact of changes. Budget considerations often require that off-the-shelf hardware and software components be used in place of custom-crafted solutions. To deal with the reuse problem, we develop the notion of scheduling abstraction for hardware and software so that application developers have a consistent scheduling interface that allows them to develop applications as if every sub-system supports GRMS.

Figure 1-1 represents a high-level view of our example application, which will be presented and solved in detail later in this paper. Since there does not currently exist a widely available standard network that supports GRMS, we build our example system around the ANSI X3T9.5 FDDI network as shown. Since Futurebus+ (IEEE 896 family of standards) and POSIX.4a support the use of GRMS for real-time applications, we use them in our example as the station backplane and operating system respectively. From the application viewpoint, our example consists of both classical real-time surveillance and control applications, and emerging multimedia applications.

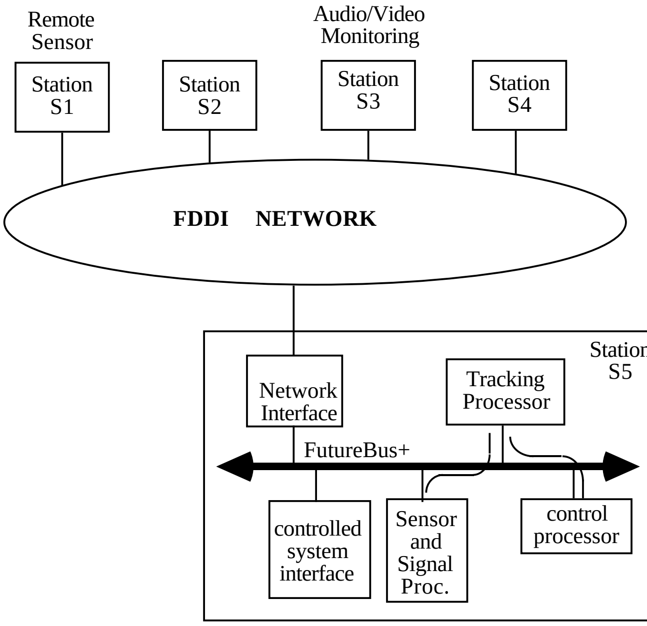

Figure 1-1: Block Diagram of Distributed Real-Time System

The rest of the paper is organized as follows. Section 2 presents an overview of generalized rate-monotonic theory for centralized systems. Section 3 describes the theoretical extensions necessary for applying GRMS to a distributed system. Section 4 discusses some key architectural concepts for real-time software, and introduces the notion of scheduling abstractions, which is a technique to use and analyze existing subsystems that are not designed to support GRMS. This is a very important aspect in the application of this theory to real-world systems. Section 5 describes a comprehensive example that illustrates task scheduling within subsystems as well as end-to-end scheduling in a large real-time system. Finally we make our concluding remarks in Section 6.

2. Overview of GRMS theory

In this section we review basic results which allow us to design a distributed real-time system. We give a brief overview of scheduling independent tasks in a centralized environment. We then address the issue of task synchronization and the effect of having task deadlines before the end of their periods.

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. By using either a simple polling procedure or a more advanced technique such as a sporadic server [23], the scheduling of aperiodic tasks can be treated within the rate-monotonic framework.

2.1. Scheduling Independent Tasks

To analyze if a set of independent periodic tasks is schedulable, we introduce the following theorem [12].

> **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
>
> $$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \le n(2^{1/n} - 1)$$
>
> 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 is unlikely to be encountered in practice. The bound is often over 90%. The remaining utilization can still be used by background tasks with low priority. To determine if tasks scheduled on a resource 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 [12]):

> **Theorem 2:** For a set of independent periodic tasks, if a task τ_i meets its *first* deadline $D_i \le T_i$, when all the higher priority tasks are started at the same time, then it meets all its future deadlines with any other 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 [10]. Consider any task τ_n with a period T_n, deadline $D_n \le T_n$, and computation C_n. Let tasks τ_1 to τ_{n-1} have higher priorities than τ_n. Note that 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 in time t and therefore $C_j \lceil t/T_j \rceil$ represents its demand in time 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 2-1.

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 2-1: Finding minimum t, where $W_i(t) = t$

We shall refer to this procedure as the *completion time* test. If all the tasks complete before their deadlines, then the task set is schedulable. For example,

Example 1: 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: $C_3 = 68$; $T_2 = 150$; $D_2 = 150$;

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 all the tasks is 0.86 which exceeds 0.779, Theorem 1's bound for three tasks. Therefore we need to apply the completion time test to determine the schedulability of task τ_3. Observe that 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.

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

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

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

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.

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

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

2.2. Task Synchronization

In the previous section, 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 non-preemptable 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 as shown in the following example.

Example 2: 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 the *priority ceiling protocol*. The priority ceiling protocol is a real-time synchronization protocol with two important properties [15, 18].

> **Theorem 3:** The priority ceiling protocol prevents mutual deadlock 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.

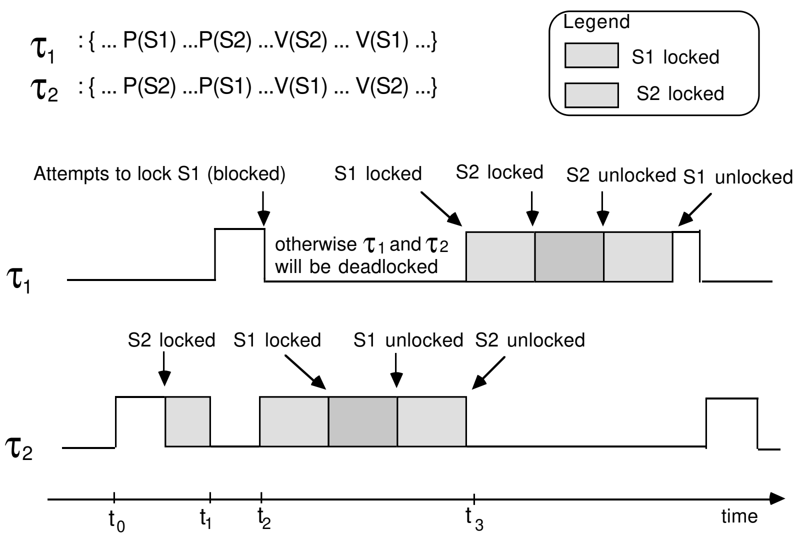

Figure 2-2: Example of deadlock prevention

Example 3: Suppose that we have two tasks τ_1 and τ_2 (see Figure 2-2). 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 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 priority ceiling protocol called *priority ceiling emulation* [19]. 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 result still holds, provided that the following restriction is observed: a task cannot suspend its execution within the critical section[3]. The priority ceiling protocol has also been extended in the context of dynamic deadline scheduling [7] and in the context of mixed dynamic and static priority scheduling [6].

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 its period. Theorem 1 can be generalized to accommodate an earlier deadline. Let $E_i = (T_i - D_i)$. That is, task τ_i has a deadline earlier by E_i. The schedulability of this task can be determined by considering it as having an end of the period deadline but being blocked by lower priority tasks for a duration of E_i. Hence, this effect can also be modeled as though task τ_i's utilization is increased by an amount E_i/T_i. The combined effect of blocking and earlier deadline can be modeled as increasing task τ_i's utilization by $(B_i + E_i)/T_i$.

Therefore the test of Theorem 1 can be augmented as follows:

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

$$\forall\, i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{(C_i + B_i + E_i)}{T_i} \le i(2^{1/i} - 1)$$

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 its blocking to the execution time of a task.

So far, the task priority assignment follows the rate-monotonic priority assignment. The period represents the window of time within which a task must initiate and complete its execution. Liu

[3]The full implementation of the priority ceiling protocol permits tasks to suspend within critical sections.

and Layland proved that it is optimal to give tasks with narrower windows higher priorities. However, a task may have its deadline before the end of its period, resulting in a window narrower than its period. Leung and Whitehead proved that it is still optimal to assign higher priorities to tasks with narrower windows. They referred to this priority assignment method as the deadline monotonic scheduling algorithm [11]. The application of deadline-monotonic assignment will be illustrated in Section 5.

2.3. Dealing with Transient Overload

In practice, a transient system overload can occur, such that not all deadlines can be met. A task assigned lower priority by either the rate or deadline-monotonic algorithms may be more important than a task assigned higher priority. Since lower priority tasks are likely to miss their deadlines first when there is an overload, the more important task may miss its deadlines before the less important task. One might try to ensure that the critical task always meets its deadline by assigning priorities according to the task's importance or by artificially assigning it a smaller deadline. However, this approach reduces system schedulability.

A better approach to address this problem is the *period transformation* technique introduced in the rate-monotonic context in [17]. However the idea can be extended to include deadline-monotonic scheduling.

Example 4: Consider two tasks with the following characteristics.

- Task τ_1: $C_1 = 3.5$; $T_1 = 10$; $D_1 = 10$; less important task.
- Task τ_2: $C_2 = 7$; $T_2 = 14$; $D_2 = 13$; critical task.

Task τ_2 has a lower rate-monotonic and deadline-monotonic priority. The completion time test can be used to determine that Task τ_2 is not schedulable. But task τ_2 is important and its deadline must be guaranteed. We can make the priority of task τ_2 higher than that of τ_1. However this will make τ_1 unschedulable. A better approach is to transform the period of task τ_2 to 7 yielding a modified task set.

- Task τ_1: $C_1 = 3.5$; $T_1 = 10$; $D_1 = 10$;
- Task τ_{2a}: $C_{2a} = 3.5$; $T_{2a} = 7$; $D_{2a} = 6$;

Note that when period transformation is used, the real task deadline is at the last transformed period. Since the real task's deadline is 13, the deadline at the second transformed period is at most 6 ($7 + 6 = 13$). Now task τ_{2a} has a higher priority by both rate monotonic and deadline monotonic assignment. Also the completion time test can be used to determine that both task τ_1 and τ_{2a} become schedulable.

3. Distributed System Extensions for GRMS

Scheduling in a network is different from scheduling in a centralized environment. In a centralized system, all resource requests are immediately known to the centralized scheduler. In some networks, distributed scheduling decisions must be made with incomplete information. From the perspective of any particular station, some requests could be delayed and some may never be seen, depending on the relative position of the station in the network. The challenge is to achieve predictability under these circumstances.

To address this challenge GRMS theory has to be extended. Certain basic concepts such as schedulability and preemption need to be revisited, and some new concepts such as system consistency need to be developed. We will demonstrate these concepts by illustrating their use in a dual-link architecture based on the IEEE 802.6 Distributed-Queue Dual-Bus (DQDB) network.

3.1. Extensions to Schedulability Concept

In a real-time system a particular activity is said to have "met its deadline" if the activity completes by its deadline. In scheduling tasks on a processor each task is said to have met its deadline if it completes execution by a certain time before the end of its period. In a communication network, the delay incurred by a message in reaching its destination is the sum of the transmission delay and the propagation delay. The transmission delay is the time between message arrival at a station and the time at which it is transmitted. The propagation delay is the time between the end of message transmission and its arrival at its destination. The transmission delay can be treated analogous to task execution on a processor. However, the propagation delay can be longer than packet transmission times causing the transmission of the next message to begin before a particular message reaches its destination. This occurs in networks such as FDDI, IEEE 802.6 DQDB, and even IEEE 802.5 token rings when early token release is used. It is therefore useful to separate transmission delay and propagation delay and consider the notion of *transmission schedulability* [20]. A set of messages is said to be *transmission schedulable* (t-schedulable) if each message can be transmitted before its deadline. Satisfaction of the end-to-end deadline of the message can be found using the relation:

$$End\text{–}to\text{–}End\ Deadline \geq Transmission\ Deadline + PropagationDelay$$

For example in an FDDI network, the worst-case propagation delay is the *walk time*, defined as the time taken by a single bit to traverse the ring if no station on the ring wanted to transmit.

3.2. Preemption Control

From the user’s viewpoint, a certain initial delay in setting up a periodic connection is acceptable. However, once the connection is set-up, users expect a steady flow of information, and hence require that C packets be delivered every period T. We will discuss the need for preemption control to achieve the above property.

Preemption is the most basic concept in priority-based preemptive scheduling. Tasks are assigned priorities according to some algorithm to maximize resource (eg: processor) utilization. It has been a long held belief that the idealized form of priority scheduling is to achieve instantaneous preemption, i.e whenever a high-priority task becomes ready, the resource is immediately taken away from any lower-priority task and given to the high-priority task.

It has been assumed that increasing preemptability always leads to a minimization of priority inversion, and that priority inversion is eliminated if a higher-priority task can always preempt a lower priority task. However, this is not true in a distributed system. In a distributed system there can be special situations when a particular preemption increases the delay experienced by lower-priority connections, but does not reduce the worst-case duration of priority inversion. We call such situations *over-preemption*, whose effect is to reduce the schedulable utilization of the network. To overcome the undesirable effect of over-preemption, a *preemption control* protocol is needed.

In the following, we use a dual-link network based on the IEEE 802.6 DQDB [21] as an example to introduce the two aspects of our preemption control protocol, namely *phase control* and *rate control*. Finally, we will address the logical relationship between preemption control and priority inversion.

The IEEE 802.6 DQDB MAC [1] protocol specifies a pair of slotted links operating in opposite directions. The links may be referred to as Flink and Rlink respectively as shown in Figure 3-1. Fixed-length slots are generated by slot generators of the corresponding links. Although the figure shows slot generators as separate functional units, the slot generation function can be embedded in stations at the end of the links. Each station is able to transmit and receive messages on both links. The selection of the link to be used for transmission depends on the physical location of the destination. Reservation for a slot on the Flink is made on the Rlink via a **request** and vice versa.

The operation of the protocol is based on a single busy bit, indicating whether the slot is used or free, and a request bit per slot for each priority level. Multiple priority levels are supported, with each priority level represented by a separate access queue. A station wishing to transmit at a certain priority on Flink, issues a request in a slot on Rlink by setting the proper request bit. It also places its own request into its access queue at the correct priority. Each station on seeing a request, enqueues it in its access queue at the correct priority. Every station on seeing a free slot discards the top request from its highest priority non-empty access queue, because the slot has been previously reserved by another station. If the top request is the station's request then it transmits in the slot on the Flink in addition to removing the request from its access queue.

> Example 5: Consider a dual-link network with three stations, as shown in Figure 3-1. Let the propagation delay between S_1 and S_2 be 10 slots and between S_2 and S_3 be 1 slot. Let S_1 and S_3 be transmitting as follows: S_1 has a low-priority connection that uses 100 slots every 10000 slots. S_3 has a medium priority connection that wants to transmit in 1 slot every 10 slots. This leads to a slot usage pattern as shown in the Figure. Slots labeled **L** are used by S_1, and the slot labeled **M** is used by S_3. Notice that S_1 has released an empty slot so that S_3 may transmit once every 10 slots as it requires. Now let S_2 start a new high-priority connection that needs to transmit in 1 slot every 4 slots. Since S_2's request has higher priority, it preempts S_3's request in S_2's queue and S_2 will transmit in the unused slots that were meant for S_3. The first of the slots released by S_1 for S_2 will take 20 units of time after S_2's first request to reach S_2. Until this time since S_2 can only transmit in slots meant for S_3, S_2 can transmit only one slot in 10 which is less than it needs. As a result, even though S_3's connection is interrupted, S_2 is not t-schedulable, resulting in an erratic connection. Therefore the preemption of S_3's request is a form of over-preemption.

To correct the problem in the above example, we need to prevent Station S_2 from using slots released for station S_3. This means that S_2 should delay its slot use for 20 slot times after its first

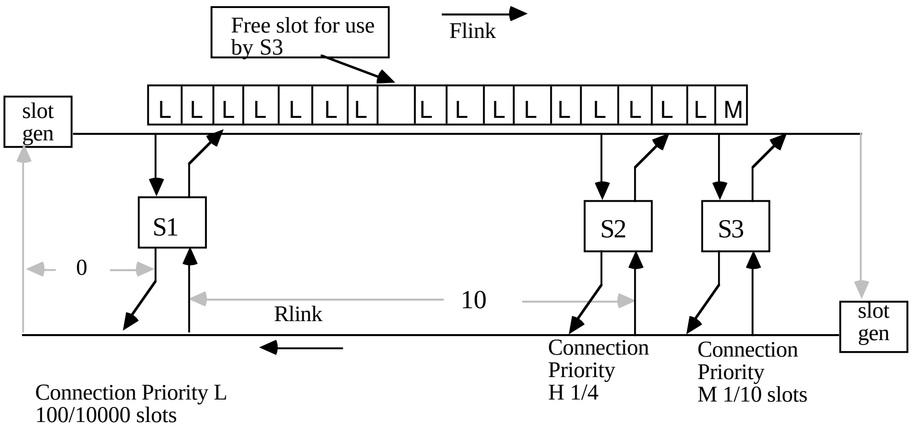

Figure 3-1: Preemption Control Example

request, which is the round trip delay between S_2 and the slot generator for the Flink. After this time, slots released by S1 in response to S2's request will reach S2. This is the phase control aspect of preemption control.

However, phase control by itself is insufficient. During the 20 unit delay, 5 cells are buffered at S_2. After the 20 unit delay, only 1 slot in 4 will be released for use by S_2. Hence if S_2 attempts to transmit all 5 slots as soon as possible, then the connection from S_3 will again be disrupted without improving S_2's worst-case end-to-end latency. Observe that the 20 unit delay will add to S_2's worst-case delay irrecoverably. After the connection is set-up, the destination expects 1 cell every 4 slots. Hence attempting transmission of all 5 cells as soon as possible does not improve S_2's worst-case performance. Hence S_2 should only transmit one slot every 4, after the round-trip delay of 20 slot times. This is the rate control aspect of preemption control. With phase and rate control both S_2 and S_3's connections will be *transmission schedulable*.

Finally, we want to point out that from an implementation viewpoint, the preemption control occurs at a layer higher than the priority queueing mechanism. Prioritized packets released by the preemption control protocol into the MAC layer will follow the usual priority queueing rules as in a centralized system.

3.3. System Consistency

As discussed before, stations in a distributed system may have incomplete or delayed information of the system state. This may lead to inconsistent views of the system state as illustrated by the following example.

> Example 6: Consider three stations S_c and S_b and S_a in a dual-link network as shown in Figure 3-2. Suppose S_b enters its own request R_b in its transmission queue and then attempts to make a request on the Rlink. Let S_b be prevented by making a request on Rlink by higher-priority requests until request R_a by station S_a passes by. On the request stream R_a precedes R_b while in S_b's transmission queue R_b precedes R_a. After the requests are registered in station S_c, the transmission queue of S_c will have R_a preceding R_b which is inconsistent with the queue of station S_b.

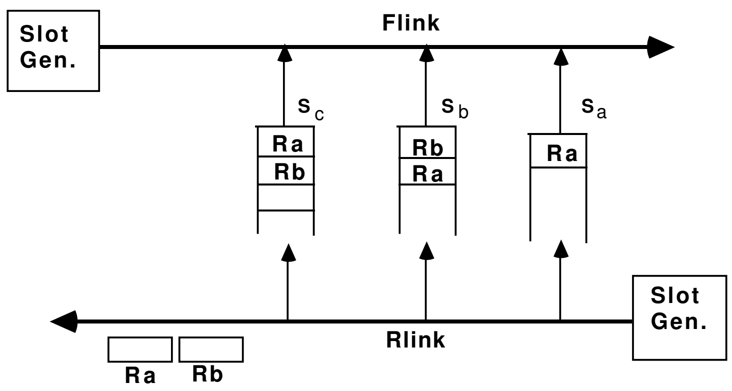

Figure 3-2: Inconsistent Station Queues in an IEEE 802.6 DQDB Network

To address this problem we introduce the concept of system consistency. System consistency can be defined as follows: In a distributed system it is possible for two request entries to exist in multiple queues. For example in a dual-link network two requests can simultaneously exist in multiple station queues [1]. A system is said to be *consistent* if and only if the order of the same entries in different station queues is consistent with each other. For example in a dual-link network, if request R_1 and request R_2 both exist in queue Q_a and queue Q_b, and if R_1 is ahead of R_2 in Q_a, then R_1 must also be ahead of R_2 in Q_b.

The inconsistency problem can lead to conflicts between distributed scheduling actions. Inconsistency can be avoided by the following rule: A station is not permitted to enter its request in its own queue until it has successfully made the request on the link. This makes the entries in each queue consistent with the ordering of requests on the link. Therefore all the queues will be consistent with each other. In the above example, station S_b cannot enter its request in its queue until it can make a request on the link. Hence S_b's request will be after S_a's request, both on the link and in S_b's queue.

In the preceding paragraphs we have highlighted fundamental new issues in distributed real-time system design. These issues are intrinsic to a wide area network where communication delays are long and scheduling has to be carried out in parallel by distributed stations with partial or delayed information. Any distributed real-time system protocol must address at least some of these issues.

A formal description of the above concepts is described as a coherent reservation protocol (CRP) and a preemption-control protocol [21]. The important theoretical result of this work is summarized by the following theorem.

> **Theorem 5:** For a given a set of periodic connections in a dual-link network that follows CRP, if the set of connections is schedulable in a centralized preemptive priority-driven system with zero (negligible) propagation delay, then the set of connections is transmission schedulable in a dual-link network.

The importance of this theorem is that even in a wide area network with incomplete information scheduling decisions can be made as though it is a centralized system. This allows us to seamlessly use GRMS in the analysis of such systems.

4. Software Architecture

In this section, we discuss some key architectural issues in using GRMS. The purpose of our real-time software architecture is to facilitate the use of GRMS in the development of real-time applications. Architectural issues comprise of two aspects: system level support and application software architecture. The system level requirements to support GRMS include:

- support to maintain reliable and accurate time at each subsystem;
- primitives to schedule periodic tasks at exact intervals;
- primitives to support the use of sporadic servers [23] for scheduling aperiodic tasks;
- primitives to support real-time synchronization protocols [15, 18] such as the priority inheritance or the more advanced priority ceiling protocol;
- primitives to enforce timing assumptions and to raise exceptions when they are violated;
- primitives to support monitoring and trouble-shooting.

A comprehensive and faithful implementation of these system primitives requires coordination between the hardware, the operating system and the programming language. Fortunately, key support is already available in IEEE Futurebus+ [5], POSIX.4 [13], Ada [2] and Ada 9x [3]. Since system level support can be obtained from commercially available industry standard components, we shall focus on application architecture issues in this paper. In particular, we shall focus upon the following:

- the containment of the impact of timing requirement changes,
- the creation of hardware scheduling abstractions for existing hardware components, and
- the creation of scheduling abstractions for existing software components.

We shall now examine each of these issues in turn.

4.1. Minimizing The Impact of Timing Requirement Changes

The primary objective in our application-level software architecture considerations is to minimize the impact of timing requirement changes. The three basic steps to achieve this goal are the decoupling of resource scheduling, providing application independent real-time I/O services for all shared sensors, actuators and other special devices, and separating the timing control structures from code providing the computational functionality.

First, we need to decouple the scheduling of resources so that we can analyze the scheduling of each resource in a system as if it were a stand-alone uniprocessor. We achieve this by allocating whenever possible a full period delay for activities on every subsystem. We will illustrate the application of this guideline in our example in Section 5. In addition, we use software *allocation units* to avoid the need to synchronize distributed tasks that share variables. One or more allocation units can reside in a processor but an allocation cannot be split across processors. Within an allocation unit, communication and synchronization is carried out by the use of shared variables for efficiency reasons. Between allocation units, synchronization and communication will be achieved by means of message passing.

The decoupling of resource scheduling greatly reduces the complexity of schedulability analysis. This decoupling is particularly important when the software system is developed concurrently by several geographically distributed teams. However, the decoupling is not without its cost. A more tightly coupled system can significantly reduce the total delay experienced by tasks, especially when a task has to serially use many resources. Readers interested in using shared variables or synchronization across processors are referred to [14, 15].

Secondly, we need to provide an application-independent real-time I/O service for I/O and other special devices. In large real-time systems such as the Space Station program, multiple applications run simultaneously and need to access the same set of sensors and actuators. Open standards such as POSIX.4 only provide generic interfaces to message-passing services and device management facilities. We recommend an I/O service layer built on top of these generic facilities solely to perform I/O functions. Individual tasks and applications can send or receive data to and from various devices by communicating with this service. A configuration manager maintains a mapping between logical names of entities and the actual entities so that I/O can be redirected without changes in the tasks that use these devices. For real-time I/O service, special care must be taken to minimize potential priority inversions. For example, the common practice of chaining multiple I/O transactions together to reduce overhead can introduce serious priority inversion because data items at different priority levels may be packaged together.

Thirdly, it is important to separate the software structure used to manage concurrency and timing from the code used to perform computations. A system's timing behavior depends upon many factors including the speed of the hardware, the structure of the runtime system, the optimization provided by a compiler and the tasking control structure. However, the logical behavior is largely independent of these factors and is more portable. Thus, it is important to separate the timing aspects from the logical aspects of the application. Furthermore, to facilitate the use of GRMS, we want to centralize all the timing parameters used to control system timing behavior. For example, the implementation of this centralized package can use templates. Such a template for implementing a periodic task is shown in Figure 4-1. The functional code for the periodic task is implemented as a procedure in a separate package that is called from within the periodic timer code. At the start of every period, the functional procedure is called and then a timer is set to wake up at the end of the period.

```
-- initialize timing control variable
NextStart = GetCurrentTime();
loop

 -- perform I/O (through real-time I/O service layer)
 Get_Data_From_IO_Server();

 -- the task functionality is computed by separately developed
 -- packages which contain the code for computations.

  Invoke_Computation_Procedure();

 -- timing control parameters such as NextStart and
 -- TaskPeriod are maintained separately in a centralized package.

 NextStart := NextStart + TaskPeriod;

 -- SleepUntil suspends invoking procedure until the absolute
 -- time specified by its parameter is reached.
 SleepUntil( NextStart );
end loop;
```

Figure 4-1: A Modular Implementation of a Periodic Task.

4.2. Hardware Scheduling Abstraction

To use GRMS for the development of real-time computing systems, we would like to use sub-systems that support the use of GRMS such as Futurebus+, POSIX.4 and Ada 9x. However, we may have to use some components that do not support GRMS. Furthermore, certain existing software packages that do not conform to GRMS may need to be used. In these cases, we need to develop *scheduling abstractions* for these components so that they can be treated as if they support GRMS. Although the scheduling abstraction allows the use of GRMS, it comes at cost of reduced schedulability due to the lack of direct support. With scheduling abstractions, we can provide application developers a consistent scheduling interface that allows them to develop applications as if every sub-system supports GRMS. In the following, we demonstrate the scheduling abstraction concept by demonstrating its application to the FDDI timed token protocol.

FDDI is a 100 Mbit/s Local/Metropolitan Area Network that has gained recent popularity. FDDI is a token ring protocol that uses a timed-token access method [8]. In a token rotation media access protocol, stations are connected to form a ring. All messages move around the ring and are repeated by each station through which they pass. A station reading its own address as the destination, copies the packet and then passes the packet to the next station in the ring. Once the frame reaches the source station, it is removed from the ring. The permission to transmit is granted to a station that is in possession of a special type of frame called a *token*. The time for a token to traverse an idle ring is called the *walk time*, denoted here as W_T.

Under this protocol, stations on the network choose a *target token rotation time* (TTRT). A station in the FDDI protocol can transmit in either synchronous or asynchronous mode. Each station is allocated a *synchronous capacity*, which is the maximum time a station is permitted to transmit in synchronous mode every time it receives the token. Synchronous capacities of each station are restricted to a pre-allocated fraction of $(TTRT-W_T)$, such that the cumulative synchronous capacity of the entire network is bounded by $(TTRT-W_T)$. When a station receives a token it first transmits its synchronous traffic for an amount of time bounded by its synchronous capacity. Then it may transmit asynchronous traffic only if the the time since the previous token departure from the same station is less than TTRT. This protocol forces the token to rotate at a speed such that the time between two consecutive token visits is bounded by 2*TTRT [16]. In a network that uses only synchronous mode, the time between consecutive token arrivals is bounded by TTRT.

Real-time scheduling analysis of FDDI for the case of one periodic connection per station has been developed [4]. In this paper, using the normalized proportional allocation scheme in [4] we create a scheduling abstraction when there are more than one periodic connections per station. In the development of this abstraction we need to consider priority inversion, system consistency, pre-emption control, and transmission schedulability. We will now develop the abstraction and describe how we address each of the above issues.

In an FDDI network that uses only synchronous mode, each station S_i can transmit once every TTRT for an amount equal to an assigned synchronous capacity H_i. Therefore the resource (network) is allocated to stations in a time-division multiplexed fashion, with no priority between stations. However, the order of message transmissions from each station may be prioritized. Furthermore, if each station locally implements sufficient priorities to use its dedicated portion of the bandwidth, there is no schedulability loss within stations. However, since there is no priority arbitration between stations, a station with the token can transmit lower-priority messages even when high-priority messages are waiting. In this sense, it is a bounded priority inversion and limits the schedulable utilization of the network.

The conditions that a message must satisfy, for it to be schedulable in an FDDI network that operates in synchronous mode are as follows:

- Each connection's period T_i must satisfy the relation $T_i \geq TTRT$.
- Each station S_i must be allocated enough synchronous capacity H_i so that each connection in the station is t-schedulable.

A simple scheme for synchronous bandwidth allocation is the normalized proportional scheme suggested by [4]. The total available bandwidth on each token rotation is given by $(TTRT - W_T)$. The normalized proportional allocation scheme gives each station a fraction of this bandwidth, consistent with that stations contribution to the total network utilization. Therefore, the bandwidth H_i allocated to station S_i is given by:

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

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 W_T is the walk time. The following example demonstrates this formula:

Example 7: Suppose we have three periodic messages τ_1, τ_2 and τ_3, to be transmitted from three stations S_1, S_2, and S_3 respectively, on an FDDI network. Let the TTRT be 8 and the walk time W_T be 1.

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

where C_i is the transmission time and T_i is the period of message τ_i. The utilization of the above message set, $U = 0.239$. Applying the above formula, $H_1 = 2.05$, $H_2 = 2.02$, $H_3 = 2.93$.

Consider the application of GRMS to an FDDI network that transmits only in synchronous mode. Let synchronous bandwidths be allocated to stations by some technique such as the above formula. Notice that if each station uses its allocated synchronous bandwidth, the actual token rotation time (TRT) (time between consecutive token arrivals at the station) attains its maximum possible value TTRT. However if any station does not use its allotted bandwidth, then the token rotates faster than TTRT.

Consider any station S_i in the network. Let the capacity allocated to the station be H_i. Let the station be a source of periodic messages $\tau_{1i} = (C_{1i}, T_{1i})$, $\tau_{2i} = (C_{2i}, T_{2i})$, $\ldots$, $\tau_{ni} = (C_{ni}, T_{ni})$. The station can transmit for up to H_i units of time, every TTRT.

GRMS can be applied to scheduling messages in this station as follows: In addition to the message set that must be transmitted, the station can be considered to have another "message" to transmit. The period of this message is the actual token rotation time TRT. The "transmission time" of this message is given by $(TRT - H_i)$. We refer to this task as Token Rotation Message τ_{tr}. The token rotation message represents the time that the station is prevented from transmitting every token rotation. The longest period of the token rotation task is TTRT. A necessary condition for schedulability is that the period of the highest frequency message must be longer than TTRT. Note that the actual token rotation time can be shorter than TTRT if other stations do not completely use their synchronous allocations. However, station S_i is guaranteed H_i amount of bandwidth in every token rotation cycle. Hence if connections in S_i are schedulable in the longest cycle (TTRT) they are also schedulable in any shorter cycle.

The FDDI scheduling abstraction described above has two levels of scheduling. At the higher level, the resource capacity is allocated between applications in a time division multiplexed (TDM) manner. Within each allocation, the resource schedules activities using GRMS. This abstraction can be directly used for sharing a processor between real-time and non-real-time applications. In this case, we create a cycle and allocate portions of the cycle to real-time and non-real-time activities respectively. Observe that similar to FDDI, the cycle has to be no greater in length than the period of the highest frequency real-time task. The TDM switching overhead can be treated similar to the walk-time in the FDDI case.

Finally, consistency between station queues is not an issue since each station has its own dedicated bandwidth. Preemption control is still necessary when a new connection has to be established and synchronous bandwidth has to be reallocated. In this case, the connection should first exercise phase control and avoid transmitting until bandwidth is allocated for the new connection. Furthermore, the new allocation should exercise rate control and not transmit all its accumulated packets. Finally, the concept of transmission schedulability is directly applicable in this abstraction as will be discussed in Section 5.

4.3. Software Scheduling Abstraction

Scheduling abstractions may also be needed to model existing software components that do not conform to GRMS principles. The goal of the hardware scheduling abstraction is to simulate support for preemptive priority scheduling in a system. However, assuming that a preemptive operating system is in use, the primary concern of the software scheduling abstraction is to restructure the interactions of software components to simulate support for the set of synchronization protocols in GRMS. In the following, we demonstrate the scheduling abstraction concept by demonstrating its application to a numerical package running on a remote server.

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 application 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 the number of priorities. One simple solution is to partition the priorities into a few classes such as high, medium and low, with similar schedulability impact.

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. A server will always execute at the ceiling priority level. We must construct a centralized request dispatcher. If requests are directly inserted into server queues, unbounded priority inversion will occur 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 8: 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.

5. Example Application

We have reviewed the theory for applying GRMS to a distributed system and developed a scheduling abstraction for FDDI, the only component in our system that does not directly support GRMS. This example illustrates how to apply GRMS extensions to schedule a distributed system that consists of both real-time control activities and multimedia communication. This example will illustrate the following concepts:

- management of end-to-end deadline by partitioning into subsystem deadlines,
- sharing of a processor by both real-time and non-real-time activities,
- application of the FDDI scheduling abstraction, and
- management of propagation delay and jitter.

5.1. Description of Example

Consider the system in Figure 1-1. It is built around an FDDI network. Station S_5 is a multiprocessor built around a Futurebus+ backplane. The control processor in station S_5 receives a variety of sensor, audio and video information, from both local and remote sources. There exists an end-to-end deadline from each source to the control processor. The end-to-end deadline is the permissible delay between the instant the information is captured at the sensor, to the instant the system outputs its response to the information. We assume that the priority ceiling protocol is used for task synchronization. A high-level description of the data flow in this example system is as follows. All time-units in the following example are milliseconds.

Traffic Across Network:

- *Station S_1*: Remote sensor information is captured and transmitted across the network to the control processor in Station S_5. The station transmits 1.0 Mbits of data every 150. Also sends 5 Mbits of data every 100 to Stations 2 and 4.
- *Station S_3*: Video monitoring station captures audio and video information and transmits it over the network to the control processor in Station 5. The required end-to-end deadline is 100 . The video source is 1024x768 pixels per frame at 24 bits/pixel and 30 frames/sec. Three CD quality audio channels sampled at 44.1 Khz with 32 bits/sample are also transmitted.

Workload in Station S_5

- *Signal Processor Tasks*:The local 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 tasks:* After the task executes it sends the result to the control processor with a period of 160. Task τ_3 on the control processor uses this tracking information. 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. We shall assume that the tracking processor provides commercial numerical computation packages and that the remote server scheduling abstraction is implemented.
- *Control Processor Tasks:* The control processor has additional periodic and aperiodic tasks which must be scheduled.
    - Aperiodic event handling with an execution time of 5 and an average interarrival time of 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. 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: $C_4 = 10$; $T_4 = 300$;
- In addition there is an existing non-real-time application which requires 9% of the CPU cycles to meet its performance goals.

5.1.1. Partitioning of End-to-End Deadlines

We now discuss the assignment of deadlines. When a message is sent within a station, it can be implemented by passing a message pointer to the receiving task and hence 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 Figure 1-1.

- The sensor takes an observation every 40.
- The signal processing task processes the signal and every 4 cycles it averages the result 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 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.

A guiding principle in partitioning the deadline is to try and minimize the impact of workload changes in a subsystem and to contain the impact within the subsystem. If each resource is allowed a full period delay, each subsystem can be analyzed as if it is an independent resource. An alternate approach is to determine the completion time at each resource and the end-end delay is the sum of the completion times. This approach is more sensitive to workload changes.

Finally, when a task is scheduled on multiple resources in series, it may arrive at the next resource well before its deadline on the current resource. If we schedule the task immediately upon its arrival, it will create the *jitter* problem as illustrated below:

> Example 9: Consider two resources R_1 and R_2 connected in series. Assume task τ_1 has a period of 10. Furthermore τ_1 is allocated a full period on each resource, and it uses each of the two resources for 5 units. Let task τ_2 only use the second resource for 3 units, with a period of 12 units. Let the first instance of τ_1 arrive at R_1 at t = 0, and let the first instance of τ_2 arrive at R_2 at t = 10.
>
> Suppose the first instance of τ_1 at resource R_1 completes its execution and arrives at R_2 at t = 10. Since τ_1 has higher priority than that of τ_2, it will immediately use R_2, preempting τ_2. Observe that the second instance of τ_1 arrives at R_1 at t = 10. Suppose this instance is not delayed at R_1. Then at t = 15 the second instance of τ_1 will begin to use R_2, further preempting τ_2's starting time to t = 10. As a result τ_2 will miss its deadline at t = 12.

The jitter effect can be easily controlled by a simple rule: A task becomes ready to use a resource at the beginning of a new period. Using this rule in the above example, the second instance of τ_1 will be buffered and will become ready to use R_2 only at t = 20. In the following discussion we assume that this rule is enforced. It should be noted that jitter control is a special case of the phase control aspect of preemption control.

The steps involved in deadline partitioning 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 units long, its delay is up to 160. Each of the other resources has a delay up to one period which is 160. That is, the total delay using rate-monotonic scheduling is bound by $4 \times 40 + 160 + 160 + 160 + 160 = 800$. 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. Hence 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 $785 - 4 \times (160) = 145$. We therefore assign a deadline of 145 to control processor task τ_3.

5.1.2. Scheduling Tasks on the Control Processor

We will concentrate on scheduling analysis of tasks in the control processor using the completion time test. Scheduling the backplane and other processors is similar. First, we need to create two virtual processors to separate the real-time and non-real-time applications. We select the length of the TDM cycle to be the same as the shortest period among the real-time tasks, that is, 100. Let the virtual processor switching overhead be 0.5. Out of 100, 9 will be allocated to non-real-time activities and 90 to real-time virtual processor, and one unit is lost in switching overhead.

Let the real-time task set in control processor execute on the real-time virtual processor. The effect of the TDM cycle spent in non-real-time and overhead processing can be modeled as a high-priority task with a period of 100 and execution of 10. Consider the requirement for aperiodic event handling with an execution time of 5 and an average inter-arrival time of 100. We create an equivalent sporadic server task with 10 units execution and a duration of 100 which has the highest priority. A simple approximate analysis consists of two parts.

- First with 90% probability the aperiodic arrives during the real-time virtual processor operation. Since we have allocated twice the average required bandwidth, we assume that the probability of an aperiodic arriving when there is no server capacity is negligibly small. Together with the fact that the aperiodic task has the highest priority, we can use a a simple M/D/1 queueing formula. We have the following result [9].
- $W = \dfrac{\rho C}{2(1-\rho)} + C = 5.132$

- where ρ = 5/100, the utilization by the aperiodic task, and C = 5.
- Secondly, with 10% probability the aperiodic arrives during the non-real-time virtual processor operation. Since the average aperiodic inter-arrival time is ten times longer than the duration of the non-real-time virtual processor, we assume that at most one aperiodic message can arrive when the virtual processor is executing. In this case the aperiodic must wait on average for half the duration of the non-real-time processor including switching overhead. In this case the response time of the aperiodic message is 5+5.132=10.132.

Finally, considering both cases the response time of the aperiodic task is: $0.9\times5.132+0.1\times10.132=5.632$. It is important to note that the analysis of aperiodic tasks is in general complex and may require simulation.

Since the TDM cycle and sporadic server have the same period they may be considered as a single task: Task τ_1: C_1 = 20; T_1 = 100; Therefore tasks on the control processor are as shown in Table 5-1.

Task	**C**	**T**	**D**
τ_1	20	100	100
τ_2	78	150	150
τ_3	30	160	145
τ_4	10	300	300

Table 5-1: Tasks on the control processor

Let tasks τ_1, τ_2 and τ_3 share several data structures guarded by semaphores. Suppose the duration of critical sections accessing shared data structures are bounded by 10 units. Suppose the priority ceiling protocol is used. Since the priority ceiling protocol is assumed to be implemented, higher-priority tasks are blocked at most once for 10 by lower-priority tasks.

We now check whether or not τ_3 completes within 145 under rate monotonic priority assignment. Under rate-monotonic assignment, τ_1 and τ_2, have higher-priority than τ_3. Hence the completion of τ_3 can be calculated using the completion time test as follows:

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

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

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

Therefore the completion time of τ_3 is 148 which is later than the required completion time of 145. 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 task τ_3's priority higher than task τ_2's priority, since τ_3 has the shorter deadline.

Under this priority assignment, 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.

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

Therefore task τ_1 is schedulable. 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 follows. Let $E_3 = (T_3 - D_3)$. 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 of τ_2 has a lower priority its critical section can delay τ_3 by 10 . Therefore $B_3 = 10$.

$$\frac{C_1}{T_1} + \frac{C_3}{T_3} + \frac{E_3}{T_3} + \frac{B_3}{T_3} = 0.2+0.188+0.094+0.0625 = 0.545 \le 2(2^{1/2}-1) = 0.828$$

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.

5.2. Scheduling Messages on FDDI

The messages that exist on the FDDI network are as follows:

Station S_1: Sensor data is collected and stored. Every 150, the station transmits 1.0 Mbits of data. Let the station also transmit 5 Mbits of data every 100 to Stations S_2 and S_4.

Station 3: Video information: The required end-to-end deadline is assumed to be 100. As an example we assume a video source of 1024x768 pixels per frame with 24 bits/pixel at 30 frames/second, compressed with a ratio of 16:1. There also exist 3 channels of CD quality audio. Each channel is sampled at 44.1 Khz with 32 bits/sample. The end-to-end deadline for audio is also 100.

Consider Scheduling of Messages at Station 3. We need to partition the end-to-end deadlines into subsystem deadlines. The resources that need to be scheduled along the path between the source and the control processor is as follows: The source interface processor, the network, the destination network interface, the backplane and the control processor itself. As discussed in Section 5.1.1, the simplest was to partition the end-to-end deadline is to allow a delay up to a period on each resource

First consider the video task. Its natural period at 30 frames/sec is 33. If we spend an entire period on each of the five resources, the end-to-end delay will exceed the limit of 100. Hence we transform the sending period to 60 Hz, i.e we send half a frame every 16.5. For the resolution given above this works out to no more than 6 of transmission time every period of 16.5.

Now consider the audio task. Its natural period is roughly one sample every 22 microseconds. This period is too short for the network as well as the packetization processing. Hence we transform the transmission period to 11. That is we accumulate 500 samples every 11 units for each of the three sources. This bundling results in efficient network utilization, but requires the destination to buffer and regulate the delivery of the voice packets at the source frequency. This yields no more than 0.5 of transmission time every period. The end to end delay over 5 resources will be no more than 55.

Each source of traffic first has to packetize the information. The schedulability analysis of the tasks running on the source network interface processor is simpler than the analysis of the control processor tasks since there is no complex data-sharing between tasks. Hence we will omit the analysis and assume that it is schedulable.

Let the TTRT be 8 and let the walk time W_T be 1. The approach to scheduling traffic on FDDI is as follows. Consider the scheduling of messages in a particular station S_i with allotted synchronous bandwidth H_i. Therefore, the station can transmit for up to H_i every TTRT. As discussed in Section 4, 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 τ_{tr}. Using this framework, schedulability of traffic at each station can be independently analyzed.

Let us apply the above technique to messages at Station 3. Let H_3 = 4, then τ_{tr} = (C_{tr}=8–4=4, T_{tr} = 8). The message set for Station 3 is then

- Token Rotation Message $\tau_{\mathrm{tr}3}$: $C_{\mathrm{tr}3} = 4$; $T_{\mathrm{tr}3} = 8$;
- Audio Message τ_{13}: $C_{13} = 0.5$; $T_{13} = 11$;
- Video Message τ_{23}: $C_{23} = 6$; $T_{23} = 16.5$;

The schedulability of this message set can be checked using the completion time test. The token rotation message is obviously schedulable. Consider completion of message τ_1:

$t_0 = C_{tr3} + C_{13} = 4 + 0.5 = 4.5$

$t_1 = W_{13}(t_0) = C_{tr3} + C_{13} = 4 + 0.5 = 4.5$

Hence τ_{13} is schedulable: Consider completion of message τ_{23}:

$t_0 = C_{tr3} + C_{13} + C_{23} = 4 + 0.5 + 6 = 10.5$

$t_1 = W_{23}(t_0) = 2C_{tr3} + C_{13} = 8 + 0.5 + 6 = 14.5$

$t_2 = W_{23}(t_1) = 2C_{tr3} + 2C_{13} = 8 + 1.0 + 6 = 15.0$

$t_3 = W_{23}(t_2) = 2C_{tr3} + 2C_{13} = 8 + 1.0 + 6 = 15.0 = t_2$

Hence τ_{23} is schedulable.

Similarly we can test the schedulability of messages at station 1. If station 1 is allotted a synchronous bandwidth of H_1 = 3, the message set at station 1 can be written as:

- Token Rotation Message $\tau_{\mathrm{tr}1}$: $C_{tr3} = 5$; $T_{\mathrm{tr}3} = 8$;
- τ_{11}: $C_{11} = 10$; $T_{11} = 100$;
- τ_{21}: $C_{21} = 15$; $T_{21} = 150$;

Note that this message set is also schedulable. The utilization of the network is 60 %.

6. Conclusion

The rate-monotonic scheduling theory and its generalizations have been adopted by projects such as the Space Station and has recently been supported by major open standards such as the IEEE Futurebus+ and POSIX. The focus of this paper is to demonstrate the use of GRMS in the design and analysis of practical real-time systems. We first provided an up-to-date and self-contained review of generalized rate-monotonic scheduling theory. We then showed how this theory can be applied in practical system development. We discussed software architecture aspects designed to facilitate the use of GRMS and the joint development of software by geographically distributed teams. We showed how existing subsystems can be reused by developing scheduling abstractions for FDDI and an existing remote server package.

References

[1] *IEEE 802.6 Distributed Queue Dual Bus - Metropolitan Area Network - Draft Standard - Version P802.6/D15*
1990.

[2] *Reference Manual for the Ada Programming Language*
U.S. Department of Defense, Washington, D.C., 1983.

[3] *Ada9X Mapping/Revision Team, Real-Time Systems Annex (draft document)*
Intermetrics, Inc., Cambridge, MA., 1992.

[4] Agrawal, G., Chen, B., Zhao, W., Davari, S.
Guaranteeing Synchronous Message Deadlines in High Speed Token Ring Networks with Timed Token Protocol.
Proceedings of IEEE International Conference on Distributed Computing Systems , 1992.

[5] ANSI/IEEE Std 896.1.
IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus+.
IEEE Standard Board and American National Standards Institute , 1990.

[6] Baker, T.
Stack-Based Scheduling of Realtime Processes.
Journal of Real-Time Systems 3(1):67--100, March 1991.

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

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

[9] Kleinrock, L.
Queueing Systems, Vol I.
John Wiley & Sons , 1975.

[10] Lehoczky, J. P., Sha, L. and Ding, Y.
The Rate Monotonic Scheduling Algorithm --- Exact Characterization and Average-Case Behavior.
IEEE Real-Time Systems Symposium , Dec, 1989.

[11] Leung, J. Y., and Whitehead, J.
On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks.
Performance Evaluation 2(4):237-250, Dec., 1982.

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

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

[14] Rajkumar, R., Sha, L., and Lehoczky J.P.
Real-Time Synchronization Protocols for Multiprocessors.
Proceedings of the IEEE Real-Time Systems Symposium :259-269, 1988.

[15] Rajkumar, R.
Synchronization in Real-Time Systems: A Priority Inheritance Approach.
Kluwer Academic Publishers, 1991.
ISBN 0-7923-9211-6.

[16] 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.

[17] Sha, L., Lehoczky, J. P. and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
IEEE Real-Time Systems Symposium , 1986.

[18] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
IEEE Transactions on Computers :1175-1185, September, 1990.

[19] Sha, L. and Goodenough, J. B.
Real-Time Scheduling Theory and Ada.
Computer , May, 1990.

[20] Sha, L., Rajkumar, R., and Lehoczky, J.
Real-Time Applications Using IEEE Futurebus+.
IEEE Micro , June 1990.

[21] Sha, L., Sathaye, S., and Strosnider J. K.
Scheduling Real-Time Communication on Dual-Link Networks.
13th IEEE Real-Time Systems Symposium , December 1992.

[22] Sha, L., Rajkumar, R. and Locke, D.
Real-Time Applications Using Multiprocessors: Scheduling Algorithms and System Supports.
Technical Report, SEI, CMU , 1990.

[23] Sprunt, H.M.B., Sha, L., and Lehoczky, J.P.
Aperiodic Task Scheduling on Hard Real-Time Systems.
The Real-Time Systems Journal , June, 1989.

[24] J. Stankovic.
Real-Time Computing Systems: The Next Generation.
IEEE Tutorial on Hard Real-Time Systems , 1988.

[25] van Tilborg, A., and Koob, G., .
Foundations of Real-Time Computing: Scheduling and Resource Management.
Kluwer Academic Publishers , 1991.

Table of Contents

List of Figures

List of Tables