and timing constraints at the level of tasking and message passing. In essence, GRMS theory guarantees that all tasks will meet their deadlines if the total system utilization of these tasks lies below a known bound, and these tasks are scheduled using appropriate algorithms. This analytic, engineering basis makes real-time systems considerably easier to develop, modify and maintain.

The generalized rate-monotonic scheduling theory begins with the pioneering work of (Liu and Layland, 1973) in which the rate-monotonic algorithm was introduced for scheduling independent periodic tasks. The rate-monotonic scheduling (RMS) algorithm gives higher priorities to periodic tasks with higher rates. RMS is an optimal static priority scheduling algorithm for independent periodic tasks when task deadlines are at period boundaries. The optimality of RMS is in the sense that if any static priority scheduling algorithm can schedule a set of independent periodic tasks with end of period deadlines, then RMS can also schedule the task set. RMS theory has since been generalized to analyze the schedulability of aperiodic tasks with both soft deadlines and hard deadlines (Sprunt and co-workers, 1989), interdependent tasks that must synchronize (Sha and co-workers, 1990b; Rajkumar and co-workers, 1988; Rajkumar, 1991), tasks with deadlines shorter than the periods (Leung and Whitehead, 1982), tasks with arbitrary deadlines (Lehoczky, 1990), and single tasks having multiple code segments with different priority assignment (Harbour and co-workers, 1991). An RMS-based technique called "period transformation" allows a task set to meet its critical deadlines even under overload conditions as long as the utilization of the critical tasks is below the schedulability bound (Sha and co-workers, 1986). RMS has also been extended to analyze scheduling of wide area networks (Sha and co-workers, 1992). RMS has also been applied to improve response times of aperiodic messages in a token-ring network (Strosnider and Marchok, 1989). The implications of RMS to Ada scheduling rules are discussed in (Sha and Goodenough, 1990), and the schedulability analysis of input/output paradigms have been treated in (Klein and Ralya, 1990). The theory has also been applied in the development of the ARTS real-time operating system (Tokuda and co-workers, 1987) and the Real-Time Mach operating system (Tokuda, 1991). Processor cache designs for real-time systems using RMS were developed in Kirk and Strosnider, 1990. Schedulability models for different operating system paradigms have been developed in Katcher and co-workers, 1991. RMS has also been applied to recover from transient hardware faults (Ramos-Thuel and Strosnider, 1991). The rate-monotonic scheduling algorithm with all its extensions is referred to as 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 (Gafford, 1990), the European Space Agency (ESA, 1990), and is supported by the IEEE Futurebus+ Standard (Futurebus+, 1990) and IEEE POSIX.4 (POSIX, 1991). A review of the basic results follows and then these results are illustrated with example applications.

11/6/00

OVERVIEW OF GENERALIZED RATE MONOTONIC SCHEDULING

The scheduling of independent periodic and aperiodic tasks begins this article. Then the issues of task synchronization and the effect of having task deadlines before the end of their period boundaries is addressed.

Scheduling Independent Periodic Tasks

A periodic task τ_i is characterized by a worst-case computation time C_i and a period T_i. Unless mentioned otherwise, assume that each instance of a periodic task must finish by the end of its period boundary when the next task instance arrives. Tasks are *independent* if they do not need to synchronize with each other. The following theorem (Liu and Layland, 1973) can be used to determine whether a set of independent periodic tasks is schedulable.

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

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

where C_i is the worst-case 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 total schedulable utilization, $n(2^{1/n}-1)$, rapidly converges to $\ln 2 = 0.69$ as n becomes large.

The bound of Theorem 1 is the least upper bound of schedulable processor utilization for the rate-monotonic scheduling algorithm. It is very pessimistic because the worst-case task set is contrived and highly unlikely to be encountered in practice. The actual boundary for task sets encountered in practice is often over 90%. The remaining utilization can still be used by background tasks with low priority. An exact schedulability test based on the *critical zone* theorem rephrased from (Liu and Layland 1973) can be used to determine whether a set of tasks having utilization greater than the bound of Theorem 1, can meet all its deadlines.

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. The following procedure (Lehoczky and co-workers, 1989) verifies if a task can meet its first deadline. 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:

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

The term $\lceil t/T_j \rceil$ represents the number of times task τ_i arrives during interval $[0, t]$ and therefore $C\lceil t/T_j \rceil$ represents its demand during interval $[0, t]$. For example let $T_1=10, C_1=5$. When $t=9$, task τ_1 demands 10 units of execution time. When $t=11$, task τ_1 has arrived again, and the cumulative demand becomes 15 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 below.

$$\text{Set } t_0 \leftarrow \sum_{j=1}^{i} C_j$$
$$t_1 \leftarrow W_i(t_0);$$
$$t_2 \leftarrow W_i(t_1);$$
$$t_3 \leftarrow W_i(t_2);$$
$$\vdots$$
$$t_k \leftarrow W_i(t_{k-1});$$
$$\text{Stop when } (W_i(t_k) = t_k)$$

This procedure is referred to 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 with end of the period deadlines:

- Task τ_1: $C_1=40$; $T_1=100$
- Task τ_2: $C_2=40$; $T_2=150$
- Task τ_3: $C_3=100$; $T_2=350$

The total utilization of tasks τ_1 and τ_2 is 0.67 which is less than 0.828, the bound for two tasks given by boundary Theorem 1. Hence these two tasks are schedulable. However the utilization of these three tasks is 0.90 which exceeds 0.779, Theorem 1's bound for three tasks. Therefore, apply the completion time test to determine the schedulability of task τ_3.

$$t_0=C_1+C_2+C_3=40+40+100=180$$

Tasks τ_1 and τ_2 are initiated one additional time in the interval (0, 180). Taking this additional execution into consideration,

$$t_1=W_3(t_0)=2C_1+2C_2+C_3= 80+80+100=260$$

Tasks τ_1 and τ_2 are initiated three and two times in the interval (0, 260), respectively. Taking this additional execution into consideration,

$$t_2=W_3(t_1)=3C_1+2C_2+C_3=120+80+100=300$$

Tasks τ_1 and τ_2 are still initiated three and two times in the interval (0, 300), respectively. Taking this additional execution into consideration,

$$t_3=W_3(t_2)=3C_1+2C_22+C_3= 120+80+100=300=t_2$$

Thus, task τ_3 completes at time 300 and it meets its deadline of 350. Hence the completion time test determines that τ_3 is schedulable even though the test of Theorem 1 fails.

Scheduling Periodic and Aperiodic Tasks

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. The completion time test is illustrated in the following example.

Example 2. Suppose that there are 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 1 unit of time is required to service each request. If the aperiodic server can execute only in the background, i.e., only after the completion of the periodic task, then the average response time for the aperiodic request 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, 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. Server task τ_2 can be allowed to use the ticket to preempt, 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 (Lehoczky and co-workers, 1987) that can reduce aperiodic response time by a large factor (a factor of 50 in this example). The key is to allow the aperiodic servers to preempt the periodic tasks for a *bounded* 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 and co-workers, 1989). Instead of refreshing the server's "ticket-box 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 server period from the request time. This replenishment approach guarantees that the aperiodic response time is no greater than the sporadic server period, provided that the system is schedulable and sufficient server capacity is available. That is, the worst-case aperiodic demand request within a duration of the sporadic server period is no more than the server capacity. In contrast, the worst-case response time for an aperiodic request serviced by a polling server is bounded by twice the period of the polling server. This situation occurs when the request arrives just after the poll. It 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.

Task Synchronization

Tasks have been, so far, assumed to be independent of one another. Tasks, however, do interact, and GRMS can be applied to real-time tasks that need to 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 as shown in the following example.

Example 3. Let τ_1, τ_2 and τ_3 be three tasks listed in descending order of priority. In addition, tasks τ_1 and τ_3 share a resource guarded by a binary semaphore S. Consider the following sequence of events:

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 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.

One might expect that, τ_1 being the highest priority task, will 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 unpredictably is because τ_3 is 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, thus the name, "unbounded priority inversion."

The unbounded priority inversion problem can be avoided by using a *priority inheritance protocol* (Sha and co-workers, 1990b). The simplest such procotol is called the *basic priority inheritance* protocol, which requires that

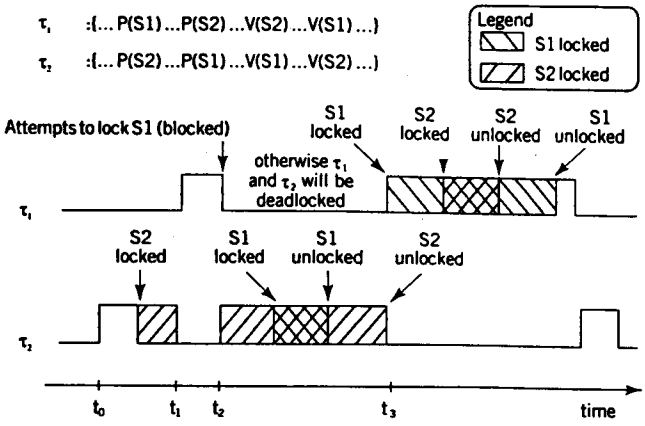

Figure 1. Example of deadlock prevention.

any task preventing one or more higher priority tasks from executing must inherit (use) the highest priority of the tasks it blocks. This simple priority, inheritance can still lead to relatively long durations of blocking for higher priority tasks. It also does not address the question of deadlocks. Unbounded priority inversion is avoided and minimized by another priority inheritance protocol called the *priority ceiling protocol.* The priority ceiling protocol is a real-time synchronization protocol with two important properties.

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 for at most one critical section.

The protocol works as follows: The *priority ceiling* of a binary semaphore S is defined 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 there are two tasks τ_1 and τ_2 (see Fig. 1). In addition, there are two shared data structures protected by binary semaphores S_1 and S_2 respectively. Suppose task τ_1 locks the two semaphores in nested fashion 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_o, 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

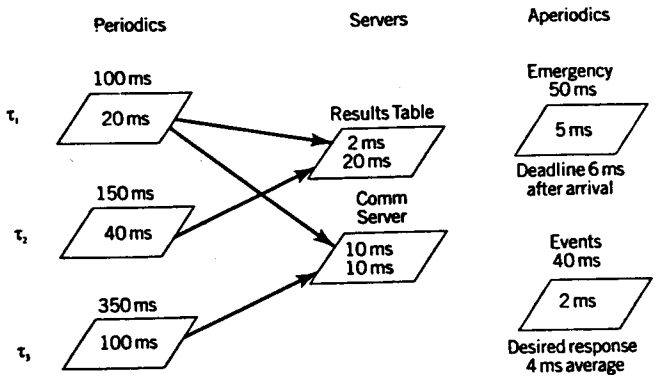

Figure 2. Task interactions for example 5. λ_2's deadline is 20 ms before the end of each period.

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 referred to as the ceiling semaphore protocol (Rajkumar and co-workers, 1988) or as priority ceiling emulation (Sha and Goodenough, 1990a). 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 hold provided that the following restriction is observed: a task cannot suspend its execution within the critical section. (The full implementation permits tasks to suspend within a critical section.) The priority ceiling protocol has been extended to deal with dynamic deadline scheduling (Chen and Lin, 1990) and mixed dynamic and static priority scheduling (Baker, 1991).

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.

Example Application

Consider the following simple example which illustrates the application of the scheduling theory.

Example 5. Consider the following task set (see Fig. 2)

1. Emergency handling task: execution time=5 ms.; worst case interarrival time=50 ms.; deadline is 6 ms. after arrival.
2. Aperiodic event handling tasks: average execution time=2 ms. (assume that it is uniformly distributed between 1 ms. to 3 ms).; average inter-arrival time=40 ms.; fast response time is desirable but there are no hard deadlines.
3. Periodic task τ_1: execution time=20 ms.; period=100 ms.; deadline is at the end of each period. In addition, τ_3 may block τ_1 for 10 ms. by using a shared communication server, and task τ_2 may block τ_1 for 20 ms. by using a shared data object.
4. Periodic task τ_2: execution time=40 ms.; period=150 ms.; deadline is 20 ms. before the end of each period.
5. Periodic task τ_3: execution time=100 ms.; period=350 ms.; deadline is at the end of each period.

Solution: First, create a sporadic server for the emergency task, with a period of 50 ms. and a service time 5 ms. Since the server has the shortest period, the rate monotonic algorithm will give this server the highest priority. It follows that the emergency task can meet its deadline.

Since the aperiodic tasks have no deadlines, they can be assigned a low background priority. However, since fast response time is desirable, we create a sporadic server executing at the second highest priority. The size of the server is a design issue. A larger server (i.e., a server with higher utilization) needs more processor cycles but will give better response time. In this example, choose a large server with a period of 100 ms. and a service time of 10 ms. There are now two tasks with a period of 100 ms.—the aperiodic server and periodic task τ_1. The rate-monotonic algorithm allows us to break the tie arbitrarily, and hence let the server have the higher priority.

Now check if the three periodic tasks can meet their deadlines. Since under the priority ceiling protocol a task can be blocked by lower priority tasks at most once, the maximal blocking time for task τ_1 is B_1=max(10 ms., 20 ms.)=20 ms. Since τ_3 may lock the semaphore S_c associated with the communication server and the priority ceiling of S_c is higher than that of task τ_2, task τ_2 can be blocked by task τ_3 for 10 ms. At this point, directly apply the appropriate theorems. However, the number of steps in the analysis can be reduced by noting that period 50 and 100 are harmonics and treating the emergency server mathematically as if it had a period of 100 ms. and a service time of 10 ms., instead of a period of 50 ms. and a service time of 5 ms. There are now three tasks with a period of 100 ms. and an execution time of 20 ms., 10 ms., and 10 ms. respectively. For the purpose of analysis, these three tasks can be replaced by a single periodic task with a period of 100 ms. and an execution time of 40 ms. (20+10+10). Now there are the following three equivalent periodic tasks for analysis:

- Task τ_1: C_1=40; T_1=100; B_1=20; U_1=0.4
- Task τ_2: C_2=40; T_2=150; B_2=10; U_2=0.267
- Task τ_3: C_3=100; T_3=350; B_3=0; U_3=0.286

Note that B_3 is zero since a task can only be blocked by tasks of lower priority. Since τ_3 is the lowest priority task, it cannot be blocked. Apply the completion time test and Theorem 2.

Task τ_1: $t_0 = C_1+B_2 = 60$, which is less than the deadline 100.

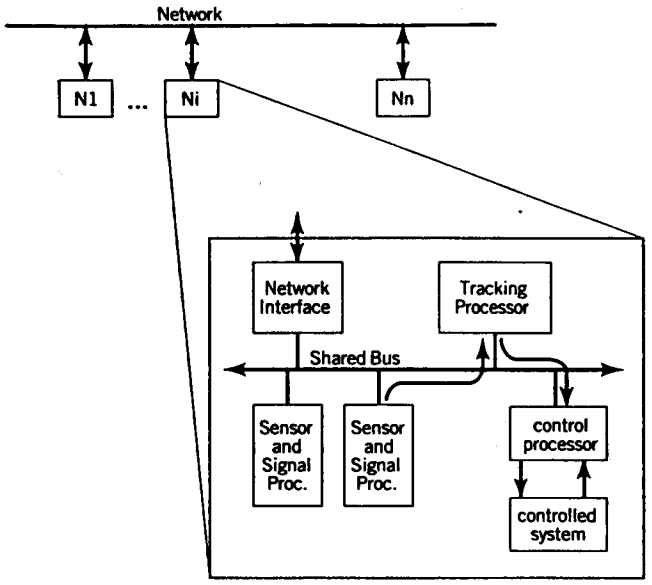

Figure 3. Distributed system configuration.

Task τ_2: $t_0=C_1+C_2+B_2=40+40+10=90$ and $W_2(t_0)=90=t_0$. That is, task τ_2 finishes at 90 which is earlier than its deadline 130 (150−20).

Task τ_3: The analysis here is identical to the analysis for Task 3 of example 1. It follows that all three periodic tasks can meet their first deadlines. By Theorem 2, they will meet all their deadlines.

The response time for the aperiodics can be determined. The server capacity is 10% and the average aperiodic workload is 5% (2/40). Because the emergency task rarely runs and most of the aperiodic arrivals can find "tickets," a good response time would be expected. Indeed, simulation indicates that the average 4 msec response-time requirement can be satisfied. The results derived for this example, show how the scheduling theory puts real-time programming on an *analytic engineering* basis.

End-to-End Delays in Distributed Systems

Consider the system in Figure 3 which uses a message-passing architecture for communication. Assume that the network used is FDDI, and a prioritized backplane such as the IEEE Futurebus+ is used. The sensor makes periodic observations, and a signal processing task processes the incoming signals and averages it every 4 cycles before sending it to the tracking processor. A tracking processor task processes the data and sends the result to the control processor. Task τ_3 on the control processor uses the tracking information. The unit of time in this example is milliseconds.

It is required that the end-to-end latency of the dataflow pipeline from the sensor to the control processor be no more than 785. 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. 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: $C_4=10$; $T_4=300$.

Assigning Message and Task Deadlines

An integrated approach to assigning deadlines to tasks and interprocessor messages is to be used. Let the sensor take an observation every 40. The signal processing task processes the signal averaging every 4 cycles and sends it to the tracking processor every 160. The tracking processor task with a period of 160 sends a message to the control processor. Task τ_3 on the control processor uses the tracking information and has $C_3=30$ and $T_4=160$ 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.

Assume for the moment 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 including the back plane has a delay up to one period which is 160. That is, the total delay using rate-monotonic scheduling is bounded by 4*40+160+160+ 160+160=800. However, the specified maximum allowable latency is 785. Hence, some tasks may have to be assigned deadlines earlier than their period boundaries. 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, attempt to assign a full period to each. 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 $785-4\times(160)=145$.

Scheduling Tasks on the Control Processor

The scheduling analysis of the control processor tasks is similar to that presented in the Example of Figure 3 except for one variation. The task set on the control processor is as described earlier with task τ_3 modified to have a deadline of 145. The completion time test for τ_3 shows that its completion time is 148. In order to meet the deadline of 145 imposed by the maximum allowable latency requirement, assign τ_3 a higher priority than τ_2 which has an end-of-period deadline of 150. This priority assignment is known as the deadline monotonic algorithm, an optimal generalization of RMS when deadlines are earlier than the periods (Leung and Whitehead, 1982). If deadline-monotonic priority assignment is different from the rate-monotonic assignment, Theorem 1 cannot be used directly. However, the completion time test can be used without modification for

the deadline-monotonic priority assignment. Its application shows that τ_3 now meets its deadline of 148. A similar test shows that tasks τ_2 and τ_4 are also schedulable.

Scheduling Messages across a Network

The problem of guaranteeing message deadlines in the FDDI network has been addressed in (Agrawal and co-workers, 1992). FDDI employs a timed-token protocol that results in a bounded token rotation time. A protocol parameter called Target Token Rotation Time (TTRT) is negotiated at network initialization. Time-critical messages must only use synchronous mode of transmission, and each station can transmit once every TTRT for an amount equal to an assigned synchronous capacity H_i. The capacity to each station is allocated proportionally using the following formula (Agrawal and co-workers, 1992):

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

where H_i is the capacity allocated to station S_i, U_i is the network bandwidth utilized by station S_i, $U=U_1+\ldots+U_n$, and D is the token round-trip propagation delay when the network is idle.

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

- Message τ_1: $C_1=7$; $T_1=100$.
- Message τ_2: $C_2=10$; $T_2=145$.
- Message τ_3: $C_3=15$; $T_3=150$.

The utilization of the above message set, $U=0.239$. Applying the above formula $H_1=2.05$, $H_2=2.02$, $H_3=2.93$.

The schedulability analysis can then 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 requirements on the application-level delays are given in the table below. There are four message types with the following timing requirements.

For meeting 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 the delays are expected to be under 21 ms, 24 ms, 30 ms and 48 ms most of the time. The absolute worst-case delays are 42 ms, 48 ms, 60 ms and 96 ms since polling guarantees a responsiveness of twice the period. If a sporadic server is used, the worst-case performance will be 21 ms, 24 ms, 30 ms and 48 ms.

Table 1. Latency Metrics

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

For schedulability analysis, 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 sequence is determined by the number of messages the task processes per period. For example, assume that the processing of one message in task M_i takes 0.5 ms, and there are three messages to be processed per period. Hence $C_1=1.5$. 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 on the processor side and in the receiving NOS correspond to standard rate-monotonic analysis.

For message scheduling on FDDI, there can be 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. This task is referred to as the Token Rotation task M_{tr}. Suppose 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 the transmission of lower priority messages from other stations. In this sense, it constitutes priority inversion and limits the schedulable utilization of the network.

CONCLUSION

The rate-monotonic scheduling theory and its generalizations provide an analytic engineering basis for building predictable real-time systems. This framework has 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 IEEE POSIX. This article summarizes the basic elements of the theory and illustrates them with examples. Extensive references are provided for further study.

BIBLIOGRAPHY

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

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

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

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

Futurebus+ P896.1 Specification, Draft 1.0, IEEE, New York, 1990. (Prepared by the P896.2 Working Group of the Microprocessor Standards Committee)

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

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

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

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

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

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

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

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

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

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

IEEE Standard P1003.4 (Real-time extensions to POSIX), IEEE, New York, 1991.

R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the Real-Time Systems Symposium*, IEEE, Huntsville, Ala., Dec. 1988, pp. 259–269.

R. Rajkumar, L. Sha, J. P. Lehoczky, "An Experimental Investigation of Synchronization Protocols," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May, 1988.

R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

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

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

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

L. Sha and S. Sathaye, "Distributed System Design Using Generalized Rate Monotonic Theory," *The Proceedings of The 2nd International Conference on Automation, Robotics, and Computer Vision*, 1992.

L. Sha, S. Sathaye, and J. Strosnider, "Scheduling Real-Time Communication on Dual Link Networks," *Proceedings of the IEEE Real-Time System Symposium*, 1992.

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

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

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

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

H. Tokuda, "The Real-Time Mach Operating System," Tech. report, Carnegie Mellon University, Pittsburgh, Pa., Sept. 1991.

LUI SHA
RAGUNATHAN RAJKUMAR
Software Engineering Institute
Carnegie Mellon University

RAULT, JEAN-CLAUDE (1937–)

Jean-Claude Rault ranks among French key pioneers in software engineering to which he contributed seminal work and which he helps by promoting through consulting, publishing and organizing conferences.

He is currently chairman of the EC2 company, a consulting and publishing firm he founded in 1987. EC2 is dedicated to software engineering, natural language processing, system and software dependability, CAD-CAM and CIM, an advanced information processing technologies such as neural networks, fuzzy systems, or genetic algorithms.

In this context, he is editor of several authoritative technical magazines and information newsletters in the French language, namely the quarterly *Génie Logiciel* (Software Engineering), *La Lettre de l'IA* (The AI Letter); he also is the publisher of the *PCTE Newsletter*. He is the founder and chairman of two successful yearly major European events, namely the Toulouse Conference (since 1988), the Avignon Conference (since 1981) dedicated to the industrial applications of, respectively, software engineering and artificial intelligence. He consults, on an international basis, for industrial companies, and government departments and international programs such as Esprit and Eureka.

From 1980 to 1987 he was with Agence de l'Informatique, a French Government Agency, with similarities with ARPA, NIST, and NSF, where he was responsible for R&D and technology transfer regarding software engineering, artificial intelligence, and CAD/CAM. In this capacity he is known to have been instrumental in launching several key projects, in promoting within French industry