REAL-TIME SYSTEMS

A real-time system can be loosely defined as a system whose response time is an important determinant of correct functioning. Let us consider a few examples. Our first example is a video game, which accepts inputs from the user, carries out some processing, and updates the state of the game on a screen. If the system is not sufficiently fast, users can lose interest. The second example is remote videoconferencing systems. These involve the transmission of images, voice, and data; and they include human interaction. The various image, voice, and data streams must be coordinated and delivered to all the participants in a timely fashion. If this is not done, the image will freeze on the screen, and voice dropouts will occur, severely degrading the system performance. A second example is a computer that is embedded in the control loop of a fly-by-wire aircraft. The computer receives signals from sensors and control inputs from the pilot. It processes them and forwards the results to the actuators (control surfaces, such as the ailerons, rudder, engines, etc.) and to the pilot display. If the computer misses too many deadlines in succession in updating control settings, the aircraft may become unstable and crash.

The common feature in all of these examples is that the system has a deadline by which to deliver its outputs. However, there is one key difference: the consequence of a failure to meet deadlines. If a video player is slow, it causes annoyance and nothing more. If an embedded fly-by-wire computer misses a lot of deadlines, it can result in a crash.

This difference is reflected in the common subdivision of real-time computers into two broad categories: *hard* and *soft.* A hard real-time system is one whose failure to meet deadines can have catastrophic consequences. A soft real-time system has no such failure consequences. In the preceding examples, the aircraft-control computer is a hard real-time system; the other two are soft.

The previous definition is subjective because the definition of what constitutes "catastrophic failure" is subjective. For example, if a stock market database is very slow in executing market transactions, that may cause events to occur that some might describe as catastrophic, and others not.

Real-time systems add the dimension of time to the design space. Every problem that the designer would confront in other computer systems is encountered here; however, the added dimension of having to meet deadlines can complicate the design process enormously. This applies especially to software. To guarantee that deadlines are met, the maximum runtimes of individual tasks must be known. Finding good upper bounds on task execution time is very difficult; indeed, we only have a few partial solutions to this problem. Runtimes are a function not only of the various possible execution paths through a task code, but also of the interaction of the application software, the executive software, and the hardware. Aspects of architecture, such as the cache and out-of-order instruction execution in pipelines, are among the complicating factors.

Another area that has resisted the most vigorous assault is proving designs and programs correct. Many real-time systems are used in life-critical applications and must be validated or formally certified before being put in use. It would be nice to have a formal way of certifying a real-time design correct; however, the existence of temporal constraints can make it very hard to prove correct any but the simplest real-time systems.

This article is organized as follows. We begin by considering what yardsticks are appropriate to evaluate the performance of real-time systems. Then we consider the problem of task assignment in real-time multiprocessors. This is followed by a discussion of real-time communication protocols, and then of fault-tolerance techniques. Finally, we briefly discuss real-time languages.

PERFORMANCE MEASURES

Performance measures used to characterize general-purpose computers will be familiar to most readers: They include throughput [e.g., in millions of instructions per second (MIPs)], reliability, and availability. These measures are not, however, suitable for real-time systems. All systems are best characterized in terms suitable to their application. In general-purpose systems, it is possible to translate the traditional measures of throughput, availability, and reliability into such terms. This is not possible in real-time systems. We will describe here two performance measures that are particularly designed for real-time systems.

Performability

This measure asks the user to specify *accomplishment levels* associated with the application (1). An accomplishment level represents a quality of performance that is distinguishable from every other level. A vector of such accomplishment levels is therefore created: $\boldsymbol{A} = (A_1\, A_2\, A_3\, \cdots\, A_n)$. Performability is then defined as the vector of probabilities $\boldsymbol{P} = (P_1\, P_2\, P_3\, \cdots\, P_n)$, where P_n is probability that the computer will perform sufficiently to permit the application to meet accomplishment level A_n.

Let us consider a simple example (see Ref. 1 for another). Suppose a video game is being created. The designer may pick the following accomplishment levels:

- A_1: The game responds to the user's input with no noticeable delay.
- A_2: Some slight delay can be noticed, but not so as to reduce significantly the quality of the game.
- A_3: The system delays are considerable and can cause annoyance.
- A_4: System delays are so considerable that most users would give up.

Once these accomplishment levels are picked, the designer then has to map them to the performance of the computer. That is, he or she has to determine what the computer response times will have to be for each of its tasks for each accomplishment level to be reached.

Cost Functions

This is a performance measure that is meant for embedded systems in the control of some process (2). It accounts for the fact that the real-time computer is in the feedback loop of the controlled process. Control theory teaches us that feedback delay increases the instability of the controlled process. This performance measure quantifies such a degradation of control.

We start by assuming the existence of a performance functional for the controlled process. Typical functionals include fuel or energy consumption, time taken to travel a given distance, and so on. Denote the performance functional by $\Omega(\boldsymbol{\xi})$, where $\boldsymbol{\xi}$ is a vector indicating the computer response time to its various tasks. Then the associated cost function is given by

$$C(\boldsymbol{\xi}) = \Omega(\boldsymbol{\xi}) - \Omega(\mathbf{0}) \tag{1}$$

where $\mathbf{0}$ is a vector of zero response times.

The cost function therefore indicates how the actual response times of the computer degrade performance, as compared to an idealized computer, which exhibits zero response time.

TASK ASSIGNMENT AND SCHEDULING

The problem of how to assign tasks to processors and schedule them is one of the most important in real-time systems. It is probably the area on which researchers have focused the greatest attention.

Let us begin by considering the various task types. Tasks can be classified in a variety of ways. One is according to their regularity: *Periodic* and *aperiodic* categories are defined. A periodic task, as its name suggests, is released periodically. Typically, it is assumed that its deadline equals its period (i.e., the deadline of a task is when its next iteration is released). There can be exceptions, however: It is not unknown for task deadlines not to equal their periods. By contrast, aperiodic tasks arrive irregularly in the system. However, they cannot arrive arbitrarily: It is assumed that there is a minimum duration that must elapse between arrivals of successive iterations of the same task.

Another classification of tasks is according to the consequences of their not meeting their deadlines. Tasks whose failure to meet deadlines can be significant are often referred to as *critical* (or *hard-real-time*) tasks; others are referred to as soft-real-time tasks.

A third classification is according to whether they are all-or-nothing tasks, or are gracefully degradable with respect to their execution time. Two examples will illustrate what we mean. Consider an algorithm that must add up some figures in your checking account before it can let you make a withdrawal. This is an all-or-nothing task: If it is terminated before it finishes adding up all the numbers, it will not be able to produce any useful output. On the other hand, consider an iterative algorithm to calculate the value of π. This algorithm quickly gets the first few significant digits for π, but could potentially go on until the numerical precision of the computer is exceeded. If we stop the processing before this happens, we will get a result for π with fewer significant digits; however, even this less accurate result is useful. This is an example of a gracefully degrading algorithm with respect to its execution time: If it is terminated prematurely, it can still produce useful results. Such tasks generally consist of mandatory portions, which have to be done before any useful result can be generated, and an optional portion. Such tasks are sometimes called *increased reward with increased service* (IRIS) or *imprecise computation.* Most of the research on scheduling such tasks has been very recent (see Ref. 3 for several algorithms for IRIS tasks).

Tasks may have precedence constraints. That is, they may require the output of other tasks to execute. However, most of the results in the literature pertain to independent tasks.

The overall task scheduling problem is as follows. Suppose we are given a set of tasks and their associated parameters. That is, we are given the task periods (for periodic tasks) or the minimum interarrival time (for aperiodic tasks). We are also given the maximum task execution times. The problem is then to develop an overall task schedule that ensures that all deadlines are met.

Such a scheduling problem can be shown to be NP complete, except under the simplest and most unrealistic conditions. Practical multiprocessor scheduling algorithms tend to work in two phases. In the *allocation* phase, tasks are assigned to processors. In the *uniprocessor scheduling* phase, a uniprocessor scheduling algorithm is executed to schedule the task assigned to each processor.

This is often an iterative process. If the allocation phase results in an assignment that cannot be scheduled successfully (i.e., so that all tasks meet their deadlines) by the scheduling phase, another allocation attempt must be made.

In the following, we outline some simple algorithms for both these phases. Unless otherwise specified, we assume that all tasks are independent and periodic, that their deadlines equal their periods, that tasks can be preempted at any time during the course of their execution, and that the cost of a task preemption is negligible.

Task Assignment

Both the algorithms we will describe are heuristics: They are not provably optimal in any sense. Their justification is that they are fairly easy to implement, and they perform quite well in most instances.

Utilization-Balancing Algorithm. This algorithm allocates tasks one by one. Each task is allocated to the processor that is least heavily utilized up to that time.

As an example, let us consider periodic tasks with execution times and periods, as shown in the following:

Task	Execution Time e_i	Period P_i	Utilization u_i
T_1	5	10	0.5
T_2	3	30	0.1
T_3	10	50	0.2
T_4	2	5	0.4

Suppose we have two processors in all, P_1 and P_2. The following lists the sequence of assignment actions. $U_b(i)$ and $U_a(i)$ denote the utilization of processor P_i before and after the indicated assignment step, respectively.

Task	$U_b(1)$	$U_b(2)$	Assign to	$U_a(1)$	$U_a(2)$
T_1	0.0	0.0	P_1	0.5	0.0
T_2	0.5	0.0	P_2	0.5	0.1
T_1	0.5	0.1	P_2	0.5	0.3
T_4	0.5	0.3	P_2	0.5	0.7

First-Fit Bin-Packing Algorithm. In this algorithm, we specify a utilization bound for each processor. A task is assigned to the first processor whose utilization bound would not be exceeded by such an assignment.

Consider again the set of tasks in our previous example. Suppose the utilization bound is set to 1 (this relates, as we shall see, to the earliest deadline first (EDF) uniprocessor scheduling algorithm). The sequence of assignment actions is shown in the following:

Task	$U_b(1)$	$U_b(2)$	Assign to	$U_a(1)$	$U_a(2)$
T_1	0.0	0.0	P_1	0.5	0.0
T_2	0.5	0.0	P_1	0.6	0.0
T_3	0.6	0.0	P_1	0.8	0.0
T_4	0.8	0.0	P_2	0.8	0.4

Uniprocessor Task Scheduling of Independent Periodic Tasks

We will describe the two best-known scheduling algorithms in this area: the rate monotonic (RM) and the EDF algorithms. Also covered briefly is the minimum laxity (ML) algorithm.

Rate Monotonic Algorithm. This is a static-priority algorithm. That is, the relative priority of the tasks does not change with time.

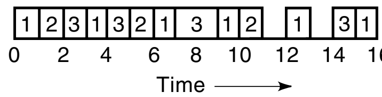

Figure 1. Example of schedule generated by rate monotonic algorithm.

In the RM algorithm, tasks are assigned preemptive priority in inverse proportion to their periods. Task T_i has higher priority than T_j if its period is less than that of T_j.

As an example, consider the following task set.

Task	Execution Time	Period
T_1	1	3
T_2	1	5
T_3	2	7

Assuming that the first iteration of each of the three tasks is released at 0, we will have task T_1 released at 0, 3, 6, 9, 12, $\cdots$; T_2 released at 0, 5, 10, 15, 20, $\cdots$; and T_3 released at 0, 7, 14, 21, 35, $\cdots$. T_1 has higher priority than T_2, which has higher priority than T_3. The first few cycles of the resulting schedule are shown in Fig 1. Whenever T_1 is ready to run, T_2 or T_3 must be preempted, if necessary. Similarly, T_2 can preempt T_3. T_3 will only run when the processor is not required by either T_2 or T_3.

There is a simple sufficiency check for the schedulability of tasks under RM. A set of tasks $T_1, T_2, \cdots, T_n$ with execution times $e_1, e_2, \cdots, e_n$ and periods $P_1, P_2, \cdots, P_n$ is guaranteed to be schedulable if

$$\frac{e_1}{P_1} + \frac{e_2}{P_2} + \cdots + \frac{e_n}{P_n} \le n(2^{1/n} - 1) \qquad (2)$$

We should emphasize that this is a *sufficient,* not a *necessary,* condition for schedulability under RM. That is, some task sets exist that do not satisfy the preceding expression but still can be scheduled successfully by the RM algorithm.

This bound, $n(2^{1/n} - 1)$, decreases monotonically as a function of n. A plot is shown in Fig. 2. The bound tends to ln 2 $\approx$ 0.693 as $n \rightarrow \infty$.

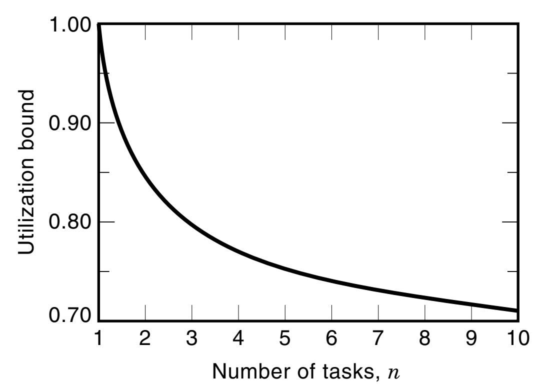

Figure 2. Utilization bound for sufficiency condition.

The necessary and sufficient schedulability conditions are as follows. Define the function

$$\Omega_i(t) = \frac{1}{t}\sum_{j=1}^{i} e_j \left\lceil \frac{t}{P_j} \right\rceil \quad (3)$$

Then task T_i will be successfully scheduled by the RM algorithm if $\Omega_i \leq 1$ (3). These conditions are derived based on the observation that the time available to execute any task is its period minus all the demands of the higher-priority tasks over that duration.

It can be proved that when the task deadlines equal their periods, RM is an optimum static-priority scheduling algorithm for uniprocessors. That is, if RM does not succeed in scheduling a set of such tasks, neither can any other static priority algorithm.

The schedulability tests for when the deadlines do not equal the periods are much more complicated and are out of the scope of this article. See Refs. 3 and 4 for information on this case.

The RM algorithm can be extended to handle aperiodic tasks. One approach is to associate a period with aperiodic tasks in general and reserve a certain amount of time every such period when pending aperiodic tasks can be run.

Earliest Deadline First Algorithm. This is a dynamic-priority algorithm. As its name suggests, it gives highest priority to the task whose deadline is the earliest among the tasks eligible to run.

When tasks are periodic and the task deadlines equal their respective periods, the schedulability test is easy: If the required overall processor utilization does not exceed one, the task set is schedulable. More precisely, a task set $\{T_1, T_2, \cdots, T_n\}$ is schedulable under the EDF algorithm if and only if

$$\frac{e_1}{P_1} + \frac{e_2}{P_2} + \cdots + \frac{e_n}{P_n} \leq 1 \quad (4)$$

Once again, the situation is much more complex when the task deadlines do not equal their respective periods: See Refs. 3 and 5 for details.

The EDF algorithm can be shown to be an optimal dynamic scheduling algorithm for uniprocessors.

Minimum Laxity Algorithm. The latest time by which a task must be started if it is to finish on time is given by $d_i - e_i$, where d_i is the absolute task deadline. This time is called the task *laxity.* As its name implies, the ML algorithm picks to run the task of minimum laxity.

Minimum laxity is not more successful than EDF in ensuring that all task deadlines are met: After all, EDF is, as we said previously, an optimal dynamic priority algorithm for uniprocessors. However, EDF does have one drawback, which ML does not. In cases when the entire task set cannot be successfully scheduled, EDF tends to discriminate against tasks with longer execution times. Such tasks miss their deadlines disproportionately often. The ML algorithm is fairer.

Priority Inversion. Priority inversion is a troublesome side-effect of using critical sections of code. A critical section cannot be held by more than one processor at any one time (6). Priority inversion can cause a task to wait needlessly for a lower-priority task to finish.

The canonical example of priority inversion is as follows. Consider three tasks, T_1, T_2, T_3, in descending order of priority. Suppose both T_1 and T_3 require the use of critical section, S. T_3 arrives at some time, say time t_0, and starts running. At time t_1, it enters S. At some subsequent time, t_2, T_1 arrives and preempts T_3. Note that T_3 has not yet relinquished its lock on S; it has just been pushed aside by T_1. T_1 runs until, at t_3, it wants S. It cannot proceed any further because T_3 is in possession of it. So it waits and lets T_3 execute. At time t_4, T_2 arrives. Because it has higher priority than T_3, it preempts T_3, and runs to completion (T_2 does not require S). Only after T_2 has finished, at t_5, can T_3 resume execution. At t_5, T_3 exits S and is immediately preempted by T_1. Now T_1 has been made to wait for T_3 to exit S and for T_2 to execute. The wait for T_3 is unavoidable: It arises from the constraint imposed by the critical section. However, the time spent by T_1 in waiting for T_2 to execute is entirely avoidable: In fact, T_2 has, for all practical purposes, been treated as if it had higher priority than T_1. This is called priority inversion.

To avoid priority inversion, we have the priority ceiling algorithm (7). The priority ceiling of the semaphore guarding a critical section, S, is the maximum priority of any task that wishes to access it. Let $S_{max}(t, T)$ be the highest-priority ceiling of all the semaphores that are locked at time t by tasks other than T. Then task T cannot enter *any* critical section at time t if its priority is less than $S_{max}(t, T)$. When a task is in a critical section and is blocking higher-priority task(s), it inherits the highest priority of the task(s) it is blocking.

It is possible to show that, under the priority ceiling algorithm, no task will be blocked by more than one lower-priority task. This allows us to bound the blocking time that could be suffered by any task. If b_i is the maximum blocking time that task T_i can suffer, it is easy to show that the task set is schedulable under the RM algorithm if

$$\frac{e_1}{P_1} + \frac{e_2}{P_2} + \cdots + \frac{e_i}{P_i} + \frac{b_i}{P_i} \leq i(2^{1/i} - 1)\,\forall\, 1 \leq i \leq n \quad (5)$$

As with Eq. (2), this is a sufficient, not a necessary, condition.

COMMUNICATION ALGORITHMS

The aim of real-time communication algorithms is to ensure that messages are delivered within a specified bound of being sent. There is a large number of such algorithms available: For a good survey, the reader should consult Ref. 8. We will describe two such algorithms, one designed for optical rings and another for store-and-forward networks. In the discussion that follows, we will assume that the reader has some familiarity with communication networks; if not, a reference such as Ref. 9 should be consulted.

Fiber Distributed Data Interface

Fiber Distributed Data Interface (FDDI) is a token-based protocol meant to run on optical ring topologies (10,11). A token circulates on the ring, and whichever node currently holds the token has the right to transmit on the ring. The algorithm owes its real-time characteristics to the bound that is imposed on the token-holding time at each node.

Traffic is classified into *synchronous* and *asynchronous* categories. Synchronous traffic is that which has a deadline associated with it, while asynchronous traffic is handled on a "best-effort" basis. Every node is assigned a quota of synchronous traffic: It is guaranteed the right to transmit this quota every time it receives the token.

Central to the operation of the algorithm is the *target token rotation time* (TTRT). This is the desired average cycle time of the token. It has an important part to play in maintaining the real-time characteristics of this protocol.

The TTRT determines whether the token is *early* or *late* at any stage. The token is said to be late if its current cycle time exceeds the TTRT; it is said to be early otherwise.

If the token arrives late at any node, that node only transmits up to its synchronous quota on the ring before passing the token to the next node. If the token arrives x seconds early, the node may transmit not only its assigned synchronous quota, but also up to x seconds' worth of other traffic.

It has been shown that the bound on the token cycle time is $2 \times TTRT$. That is, each node is guaranteed that it can transmit up to its synchronous quota every $2 \times TTRT$ seconds. This is the special case of a result that says that the time for K consecutive cycles cannot exceed $(K + 1) \times TTRT$.

Let us now turn to a procedure for setting the $TTRT$ value and the per-node synchronous traffic quota (12). We will begin by defining some notation. Consider the (periodic) synchronous traffic, S_i, emerging from node i. Such traffic is characterized by the 3-tuple, $S_i = (c_i, P_i, d_i)$; c_i is the size of the traffic generated per period P_i, and d_i is its relative transmission-start deadline (i.e., the time following its arrival by which it has to *start* transmitting). Define $u_i = c_i/\min(P_i, d_i)$: u_i can be regarded as a measure of the *utilization* of the ring by stream S_i.

Since the cycle time is upper bounded by $2 \times TTRT$, we must set $TTRT = \min d_i/2$. Now comes the task of setting the synchronous quotas. It can be shown that assigning the following synchronous quota per node will satisfy the need to transmit c_i bits of data every P_i seconds, to meet transmission-start deadline d_i:

$$Q_i = \frac{u_i d_i}{\lfloor d_i/TTRT - 1 \rfloor} \tag{6}$$

so long as

$$\sum_{i=1}^{n} Q_i + \tau \leq TTRT \tag{7}$$

where τ is the overhead associated with token passing. That is, $TTRT - \tau$ is the time available for transmitting packets.

The Stop-and-Go Protocol

The Stop-and-Go protocol is meant for multihop networks. The protocol works by bounding the delay at each hop. Knowing the route that a message takes from input to output allows us to bound the total time taken.

The time axis at each link is subdivided into *frames.* The best way to think about frames is to imagine (virtual) interframe markers transmitted at regular intervals by a node on its outgoing links. As the marker travels down the link, it defines the end of one frame and the beginning of another. It should be stressed that these markers are imaginary and meant for conceptual purposes only. The frames that they define, however, are very real and lie at the heart of the Stop-and-Go protocol.

Multiple traffic classes are supported by this protocol, and associated with each class is a frame size. The protocol is as follows. When a class-i packet arrives at an intermediate node (en route to its destination), it becomes eligible for forwarding by that node to the next node in its path only upon the beginning of the next outgoing frame following its arrival. To make this clear, consider Fig. 3. The figure shows class-i frames incoming and outgoing at a node. When a packet arrives at a node, it becomes eligible for forwarding by that node at the beginning of the outgoing frame indicated by the arrows. We call the incoming-outgoing frame pairs as *conjugate frames.*

Packets eligible for transmission are transmitted according to a non-preemptive order. The priority of a class is inversely related to its frame size. For example, if $f_1 = 3, f_2 = 5$, eligible packets in class 1 will have priority over eligible packets in class 2.

It can be shown that so long as the traffic intensities do not exceed a given bound, incoming traffic on a frame will always be able to be transmitted in the outgoing (conjugate) frame in which it becomes eligible; we will describe this bound later. What this result means is that the maximum delay of any class-i traffic in any node is given by $3f_i + d$, where f_i is the frame size associated with class-i traffic and d is the overhead for handling the packet at the node. This is derived as follows. The earliest a packet can arrive in a frame is at its very beginning; the latest it leaves is at the end of its conjugate outgoing frame. This accounts for $2f_i$ time. Furthermore, there is no requirement that the incoming and outgoing frames be aligned with respect to one another. The worst case arises when an outgoing frame begins momentarily before an incoming frame ends. This can lead to up to f_i further delay. Putting all this together with the processing overhead at the node, we get $3f_i + d$.

It only remains for us to specify the traffic intensity bounds under which this protocol will work correctly. Let $C_l(i)$ denote the total load on link l imposed by class-i traffic, and γ denote the maximum packet size. Let B_l denote the total bandwidth of link l, and n the total number of traffic classes. Then the protocol requires that the following inequalities be satisfied for the preceding delay bound to work:

$$\sum_{i=j}^{n} C_\ell(i)\left(1 + \left\lceil \frac{f_j}{f_i} \right\rceil\right)\frac{f_i}{f_j} - C_\ell(j) \leq \begin{cases} B_\ell - \gamma/f_j & \text{if } j = 2, \ldots, n \\ B_\ell & \text{if } j = 1 \end{cases} \tag{8}$$

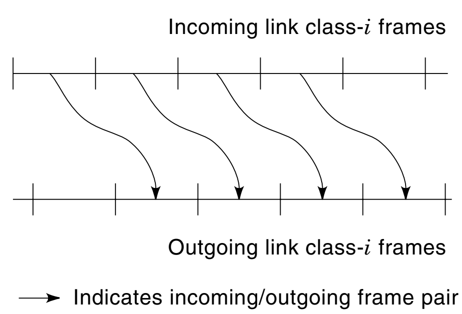

Figure 3. Illustrating frames in the Stop-and-Go protocol.

It can also be shown that the total buffer required at per link l for traffic-class i is upper bounded by $3C_l(i)f_i$.

The designer must subdivide the traffic suitably into classes, pick appropriate frame sizes, and set the link bandwidths.

FAULT TOLERANCE

The article in this encyclopedia on fault tolerance covers general-purpose fault-tolerant techniques. In this section, we limit ourselves largely to fault-tolerant issues specific to real-time systems.

To tolerate faults, a system must have redundancy. Redundancy may be in hardware, software, or time. Hardware redundancy takes the form of additional processors and communication links; software redundancy is implemented in having multiple versions of software executing the same function, and time redundancy exists whenever there is slack in the schedule. Of these, hardware redundancy is a general fault-tolerance technique, so we do not discuss it further here.

Software Redundancy

Software faults are essentially design faults. Unlike hardware, software does not wear out as time goes on, and there is no point replicating software modules in the same way as hardware is replicated in *N*-modular redundancy. To implement software fault tolerance, we need multiple versions of software, written by independent teams of programmers. The hope is that since they are written independently, the versions will not suffer correlated failure (i.e., they will not fail on the same set of inputs).

There are two ways of implementing software fault-tolerance. The first is similar to *N*-modular redundancy in hardware fault-tolerance. Called *N*-version programming (13), it consists of *N* versions of software independently written for the same algorithm. These versions are executed in parallel, and their outputs are voted on. So long as a majority of the versions run successfully, there will be a correct output from the system.

The second approach is to use *recovery blocks* (14). Again, multiple versions of software are used; however, only one version is ever run at any one time. The sequence of events is as follows. One version is run, and its results passed through an *acceptance test.* This test checks to see if the output falls within the expected range. If the test is passed, the output is accepted by the system; if not, another version is made to execute. Its output is similarly run through an acceptance test. The process continues until either a version is executed that passes the acceptance test (success) or we run out of versions or miss the task deadline (failure).

The major drawback of software redundancy approaches is cost. Software costs dominate the development costs of most large systems. Generating independent replicates of the critical tasks can increase costs even more. Another problem is that even if the versions are developed independently without the development teams exchanging ideas, it is possible to have correlated failures. For example, different teams may interpret ambiguities in the specification in the same way, or certain types of mistakes may simply be so common that they occur in multiple versions. If the same algorithm is implemented, numerical instabilities in it can cause further correlations. The existence of correlated faults severely degrades the reliability that can be obtained from software redundancy. Not much is known about the extent to which industrial-grade replicates of software modules suffer correlated failure: Most experiments on software fault tolerance have been carried out in universities, where students can be used as programmers.

Time Redundancy

Time redundancy consists of having sufficient slack in the schedule, so that after a failure is detected, the system is still able to meet the deadline of the affected tasks. Time redundancy is most often exploited in the handling of *transient* faults. As the term implies, these are faults that occur and then go away after some time. Such faults have many causes. One of the most common is the impact of alpha-particle radiation. When alpha particles go through a memory cell, they sometimes have enough charge to change their state from 0 to 1 or vice versa. This fault is transient because the cell has not been physically damaged; it goes away when it is overwritten.

Checkpointing is frequently done to render time redundancy more efficient. The state of the process is stored regularly in a safe place. If faulty behaviour is discovered, the process is simply rolled back to the last checkpoint and resumed. This avoids having to restart the process from the beginning.

The question arises as to how to place the checkpoints. Typically, they are placed at equal intervals along the execution trajectory. The question then is how many checkpoints should be used. The greater this number, the smaller the distance between them, and hence the less the time taken for a rollback. In general-purpose systems, the checkpoints are placed so as to minimize the average execution time. By contrast, in real-time systems, they should be placed so as to reduce the chances of missing a hard deadline, even if this entails increasing the average execution time (15).

Fault-Tolerant Clock Synchronization

Clock synchronization allows for faster communication between processors. The simplest clock synchronization method consists of distributing a single clocking signal to all the processors. If the length of the path from the root of the clocking tree to the processors is roughly the same, the clocks will be fairly well synchronized. However, this approach is not fault tolerant, since the failure of the common clocking source will bring down the entire clocking system.

We present in this section two approaches to fault-tolerant clock synchronization. First, we provide some background information.

All clocks can be regarded mathematically as a mapping from the fictitious “real time” to something called “clock time.” For example, if at real time of 10:00 UTC (coordinated universal time) my watch says 10:02, my clock time at a real time of 10:00 is 10:02. Real clocks drift (i.e., they go faster or slower than a perfect clock would). Their maximum drift rate (i.e., the rate at which they run fast or slow) varies with the clock technology. Clocks based on quartz crystals typically have drift rates of about 10^{-6} (i.e., they may gain or lose about a second for every million seconds). The clocks at the Bureaus of Standards around the world are about a million times more accurate.

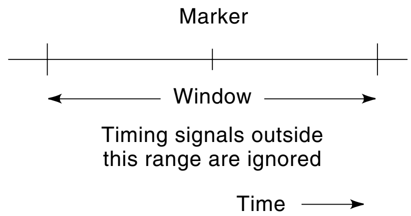

Figure 4. Interactive convergence algorithm.

If two clocks are synchronized at time 0 and then left to run freely, at time t they will diverge by at most $2\rho t$, where ρ is the maximum drift rate. This is because in the worst case, one clock can run fast at rate $(1 + \rho)$, while the other runs slow at the rate $(1 - \rho)$.

If, whenever a clock fails, it simply stops sending out timing signals, clock synchronization would be a very simple problem. However, this is not always the case: Often, when a clock fails, it sends out incorrect timing information, or even inconsistent information (e.g., it could say "it is 2:00 PM" to one processor and "it is 3:00 PM" to another processor at the same time). Failures that result in such contradictory outputs are called *Byzantine* or *malicious* failures. The two algorithms we present next are designed to work in the face of such a failure mode. In general, it can be shown that if up to f maliciously faulty clocks are to be tolerated, the system must consist of at least $N = 3f + 1$ clocks.

In both algorithms, we assume a system model in which each processor has its own clock. These clocks interchange clocking signals, and the clocking signal used by each processor is a function of these. We will also assume that the propagation time for clock signals is negligible.

Phase-Locked Clocks. Each processor (more accurately, its clocking subcomponent) receives inputs (i.e., square-wave signals) from all the clocks in the system, including its own. The clocking network is a fully connected graph (i.e., each clock has a line to every other clock in the system). If up to f faulty clocks are to be tolerated, each clock averages the timing signal from the $(f + 1)$th and $(N - f)$th signals it receives (according to the order in which it receives them). It speeds up, or slows down, its own clock to try to align it with this average signal. This approach can be shown to ensure very tight synchronization if there are at least $N \geq 3f + 1$ clocks in the system (16).

A completely connected network can be quite expensive if N is large, since the number of links grows quadratically with N. It is possible to use a sparser interconnection network to propagate the clocking signals, by subdividing the network into a hierarchy of completely connected clusters. The clusters themselves are more sparsely connected to one another. This can substantially reduce the network cost, although it can result in tripling the maximum clock skew between clocks in different clusters. See Ref. 17 for further details.

An Interactive Convergence Synchronization Algorithm. This is a software synchronization technique (18). Every time it reads a multiple of R seconds, a clock sends out a message (marker) announcing its current time to the other clocks. Each clock therefore has a sequence of timing messages coming in. It ignores timing signals that fall outside a certain window of its own clocking signal and averages the clocking signals that fall within it. This is the time value that is used (Fig. 4).

REAL-TIME PROGRAMMING LANGUAGES

In this section, we describe some of the features one looks for in a real-time programming language. This treatment is necessarily brief; for more complete coverage, the reader should consult either a language manual or books devoted to real-time programming languages (19,20).

Most of the desired features in real-time languages are the same as those for a general-purpose language and are omitted from this section. We concentrate instead on those features that are much more important to the real-time programmer than to his or her general-purpose counterpart. Many of our examples are from the Ada programming language (Ada is a trademark of the US Department of Defense).

Subtypes and Derived Types

Real-time languages should be strongly typed and permit the programmer to construct *subtypes.* A subtype has to follow the rules of its parent type and be restricted to a given range. Thus, for example, the programmer might say

```
type DEPTH is new int range 0..500
```

DEPTH is of type `int` and has the additional restriction that its value should lie between 0 and 500. If, at any time during execution, it strays beyond this limit, the system will report an error.

Subtypes can be mixed in expressions. For example, if we define subtypes of `int`, `DEPTH`, and `ALTITUDE`, we can have a statement `A = DEPTH + ALTITUDE`. It is possible to define types that cannot be mixed in this way: These are called *derived* types. For example, we may define

```
type PRESSURE is new int
type TEMPERATURE is new int
```

We cannot now mix `PRESSURE` and `TEMPERATURE` in the same expression. Just as with subtypes, derived types can also be given a range.

Numerical Precision

Every C programmer knows that `double` is supposed to give a higher precision than `float`. However, the exact level of precision varies from one machine to the next. It is important to be able to specify exactly how much precision one wants. In Ada, for example, one can say

```
type xyz is digits 8 range −1e5..1e5
```

Then `xyz` is a type with eight decimal digits of precision, with range between `−1e5` and `1e5`.

Supporting Time

One of the most difficult things for a language to do is to specify that one event must take place `x` milliseconds after some other event. Practically, no languages exist that do this precisely. Languages such as Ada allow us to specify a delay, although it is implemented as a lower bound. That is, we can specify only that two events must be separated in time by *at least* `x` milliseconds.

We should also mention that at least one language tries to make it easier to estimate program runtimes. As we pointed out earlier, such estimates are extremely difficult to make.

Euclid, an experimental language, disallows `while` loops on the grounds that it is not always possible to bound the number of iterations in such loops. This makes it easier to bound at least the number of executed instructions in a program and takes one partway toward being able to bound program runtimes.

Exception Handling

When things go wrong, it is often important for the real-time system to respond quickly and try to compensate. A real-time language should have a rich set of exception-handling features. Let us consider some examples from Ada. This language has three built-in exceptions:

- `CONSTRAINT_ERROR`: This flag is raised whenever a variable strays outside its designated range or when the program tries to access an array outside its bounds.
- `NUMERIC_ERROR`: This exception is raised whenever a computation occurs that cannot deliver the prescribed level of precision.
- `STORAGE_ERROR`: This exception indicates that the dynamic storage allocator has run out of physical storage.

In addition, the programmer can define his or her own exceptions, through the `raise` command.

DISCUSSION

In this article, we have briefly surveyed some aspects of real-time systems. Real-time systems are becoming increasingly prevalent, with computers involved in the control of cars, aircraft, nuclear reactors, as well as in multimedia, videoconferencing, and command and control systems. It is increasingly being recognized that the addition of response time as a performance criterion can dramatically change the outcome of design tradeoffs.

The field has developed unevenly. Task assignment and scheduling are mature subfields, with hundreds of papers devoted to them. By contrast, real-time databases and the formal verification of real-time systems are still in an early stage of development. In the case of formal verification techniques, it is not for want of trying but rather because of the extreme difficulty of the problem. Powerful formal validation procedures are badly needed since computers are increasingly used in life-critical applications, where failure of the computer can lead to loss of life.

FURTHER READING IN REAL-TIME SYSTEMS

There are several books on real-time systems. The books by Kopetz (21) and by Krishna and Shin (3) provide a general description of real-time systems. There are three collections of important papers from the real-time literature that are worth reading (22–24).

The main conference in this field is the *IEEE Real-Time Systems Symposium.* The chief specialist journal is *Real-Time Systems,* published by Kluwer Academic Publishers. Real-time papers also regularly appear in the journals of the IEEE Computer Society.

BIBLIOGRAPHY

1. J. F. Meyer, On evaluating the performability of degradable computing systems, *IEEE Trans. Comput.,* **C-29**: 720–731, 1980.
2. C. M. Krishna and K. G. Shin, Performance measures for control computers, in A. K. Agrawala and S. K. Tripathi (eds.), *Performance '83,* Amsterdam: North-Holland, 1983, pp. 229–250.
3. C. M. Krishna and K. G. Shin, *Real-Time Systems,* New York: McGraw-Hill, 1997.
4. J. P. Lehoczky, Fixed priority scheduling of periodic task sets with arbitrary deadlines, *Proc. IEEE Real-Time Syst. Symp.,* Lake Buena Vista, FL, 1990, pp. 201–209.
5. S. K. Baruah, A. K. Mok, and L. E. Rosier, Preemptively scheduling hard-real-time sporadic tasks on one processor, *Proc. IEEE Real-Time Syst. Symp.,* Lake Buena Vista, FL, 1990, pp. 182–190.
6. A. Tannenbaum, *Operating Systems: Design and Implementation,* Englewood Cliffs, NJ: Prentice-Hall, 1987.
7. L. Sha, R. Rajkumar, and J. P. Lehoczky, Priority inheritance protocols: An approach to real-time synchronization, *IEEE Trans. Comput.,* **39**: 1175–1185, 1990.
8. C. M. Aras et al., Real-time communication in packet-switched networks, *Proc. IEEE,* **82**: 122–139, 1994.
9. A. Tannenbaum, *Computer Networks,* Englewood Cliffs, NJ: Prentice-Hall, 1996.
10. R. M. Grow, A timed token protocol for local area networks, *Proc. Electro/82 Token Acces Protocols,* 1982, Paper 17/3.
11. K. C. Sevcik and M. J. Johnson, Cycle time properties of the FDDI token ring protocol, *IEEE Trans. Softw. Eng.,* **SE-13**: 376–385, 1987.
12. N. Malcolm, S. Kamat, and W. Zhao, Real-time communication in FDDI networks, *Real-Time Syst.,* **10** (1): 75–107, 1996.
13. J. P. J. Kelly and S. Murphy, Dependable distributed software, in Y.-H. Lee and C. M. Krishna (eds.), *Readings in Real-Time Systems,* Cupertino, CA: IEEE Computer Society Press, 1993, pp. 146–173.
14. B. Randell, System structure for software fault-tolerance, *IEEE Trans. Softw. Eng.,* **SE-1**: 220–232, 1975.
15. C. M. Krishna, K. G. Shin, and Y.-H. Lee, Optimization criteria for checkpointing, *Commun. ACM,* **27**: 1008–1012, 1984.
16. N. Vasanthavada and P. N. Marinos, Synchronization of fault-tolerant clocks in the presence of malicious failures, *IEEE Trans. Comput.,* **C-37**: 440–448, 1988.
17. K. G. Shin and P. Ramanathan, Clock synchronization of a large multiprocessor system in the presence of malicious faults, *IEEE Trans. Comput.,* **C-36**: 2–12, 1987.
18. L. Lamport and P. M. Melliar-Smith, Synchronizing clocks in the presence of faults, *J. ACM,* **32**: 52–78, 1985.
19. A. Burns and A. Wellings, *Real-Time Systems and their Programming Languages,* Reading, MA: Addison-Wesley, 1987.
20. S. J. Young, *Real-time Languages: Design and Development,* Chichester, UK: Ellis Horwood, 1982.
21. H. Kopetz, *Real-Time Systems,* Boston: Kluwer, 1997.
22. Y.-H. Lee and C. M. Krishna, *Readings in Real-Time Systems,* Cupertino, CA: IEEE Computer Society Press, 1993.
23. J. A. Stankovic and K. Ramamritham, *Hard Real-Time Systems,* Cupertino, CA: IEEE Computer Society Press, 1988.
24. J. A. Stankovic and K. Ramamritham, *Advances in Real-Time Systems,* Cupertino, CA: IEEE Computer Society Press, 1993.

C. M. KRISHNA
University of Massachusetts

RECEIVER PROTECTORS. See MICROWAVE LIMITERS.
RECEIVERS. See DEMODULATORS; MICROWAVE RECEIVERS; UHF RECEIVERS.
RECEIVERS, RADAR. See RADAR SIGNAL DETECTION.
RECEIVING AND SHIPPING. See WAREHOUSE AUTOMATION.