Real-Time Computing: A New Discipline of Computer Science and Engineering

KANG G. SHIN, FELLOW, IEEE, AND PARAMESWARAN RAMANATHAN, MEMBER, IEEE

Invited Paper

This paper surveys the state of the art in real-time computing. It introduces basic concepts and identifies key issues in the design of real-time systems. Solutions proposed in literature for tackling these issues are also briefly discussed.

I. Introduction

Real-time computing has emerged as an important discipline in computer science and engineering. Many of the key issues identified in [95] have received widespread attention (e.g., scheduling), but there are still many (some old and some new) issues that need to be resolved. The main objective of this paper is to survey the state of the art in real-time computing and identify issues that warrant further investigation.

A. What is Real-Time Computing?

There are three major components and their interplay that characterize real-time systems. First, "time" is the most precious resource to manage in real-time systems. Tasks must be assigned and scheduled to be completed before their deadlines. Messages are required to be sent and received in a timely manner between the interacting real-time tasks. The correctness of a computation depends not only on the logical correctness but also on the time at which the results are produced. Second, reliability is crucial, since failure of a real-time system could cause an economical disaster or loss of human lives. Third, the environment under which a computer operates is an active component of any real-time system. For example, for a drive-by-wire system it is meaningless to consider on-board computers alone without the automobile itself.

Manuscript received July 13, 1993. This work was supported in part by the Office of Naval Research under Grants N00014-92-J-1080 and N00014-91-J-1115, and by the National Science Foundation under Grants MIP-9203895 and MIP-9009154. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

K. G. Shin is with the Department of Electrical Engineering and Computer Science, University of Michigan–Ann Arbor, Ann Arbor, MI 48109-2122.

P. Ramanathan is with the Department of Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, WI 53706-1691.

IEEE Log Number 9214151.

A real-time application is usually comprised of a set of cooperating tasks. The tasks are often invoked/activated at regular intervals and have deadlines by which they must complete their execution. In each invocation, a task senses the state of the system, performs certain computation (e.g., for derivation of a control law), and if necessary, sends commands to change and/or display the state of the system. For example, in an automobile application, a task may sense the pressure from the brake pedal and the speed of the individual wheels, perform computation to determine if a wheel is locked, and then activate antilock braking actions by changing the position of the valves in the system. Likewise, in an aircraft-control application, a task may monitor the current position of the throttle, perform computation based on the sensed position, and then change the thrust of an engine by altering the fuel injected to it.

These tasks are referred to as *periodic* tasks. A common feature of periodic tasks is that they are *time-critical* in the sense that the system cannot function without completing them in time. For instance, in the automobile application, if the task does not activate antilock braking within a short interval after a wheel is locked, the vehicle is likely to enter a spin which, in turn, could result in an accident. Similarly, in the aircraft application, if the thrust is not regulated in time, the plane may crash and result in loss of human lives. It is, therefore, very important for the computer system to ensure that the deadlines of the critical tasks are met regardless of the other conditions in the system.

Of course, not all tasks in a real-time application arrive at regular intervals. Some tasks are activated only when certain events occur, and they are commonly referred to as *aperiodic* tasks. For example, a system reconfiguration task may be activated only when an error/fault is detected by the system. Since the events may not always occur at regular intervals, the corresponding tasks also do not arrive at regular intervals. If the event is time-critical, then the corresponding aperiodic task will have a deadline by which it must complete its execution. On the other hand, if the event is not time-critical, then the corresponding aperiodic task will not have any deadline, but it must be serviced as soon as possible without jeopardizing the deadlines of the other tasks.

Based on the above discussion, deadlines of real-time tasks can be classified as either *hard*, *firm*, or *soft*. A deadline is said to be *hard* if the consequences of not meeting it can be catastrophic. Periodic tasks usually have deadlines which belong to this category. A deadline is said to be *firm* if the results produced by the corresponding task cease to be useful as soon as the deadline expires, but consequences of not meeting the deadline are not very severe. The deadlines of many aperiodic tasks belong to this category, e.g., transactions in a database system [33]. A deadline which is neither hard nor firm is said to be *soft*. The utility of results produced by a task with a soft deadline decreases over time after the deadline expires.

At this point, it is probably natural to ask where do the deadlines come from or how does one know whether a deadline is hard, firm, or soft. The deadlines come from the application. For example, consider an air-defense system that is monitoring the sky for incoming enemy missiles. Due to the nature of the application, the timing constraints are such that the incoming enemy missile must be destroyed within 15 s of detection [58]. This, in turn, imposes deadlines on other tasks which either detect, identify, engage, or launch an intercept missile. For instance, an incoming missile must be identified with 0.2 s of detection, and if necessary, an intercept missile must be engaged within 5 s after detection and launched within 0.5 s of engagement [58]. These task deadlines will in turn impose deadlines on their subtasks, which will then impose deadline on their subtasks, and so on.

The above example also highlights another important characteristic of real-time applications. Since every incoming enemy missile must be destroyed without fail, the behavior of the controlling real-time computing system must be *predictable*. That is, it should be possible to show at design time that all the timing constraints of the application will be met as long as certain system assumptions are satisfied. For example, suppose the only assumption about the system is that the total number of faults at any given time is less than or equal to a threshold f. Then, the system is predictable if one can demonstrate at design time that all the timing constraints will be satisfied as long as there are f or fewer faults. Since the need for predictability has significant impact on the design of real-time systems, it will be discussed at more length in Section I-B.

In addition to timing and predictability constraints, tasks in a real-time application also have other constraints one normally sees in traditional non-real-time applications. For example, the tasks may have:

- resource constraints: a task may require access to certain resources other than the processor, such as I/O devices, communication networks, data structures, files, and databases;
- precedence constraints: a task may require results from one or more other tasks before it can start its execution; and
- dependability/performance constraints: a task may have to meet certain reliability, availability, and/or performance requirements.

B. What is Predictability?

As indicated earlier, the notion of predictability is very important to real-time systems. However, the meaning of predictability may vary from one application to another or even from one task to another. In a simple system, predictability means that it is possible to demonstrate at design time that constraints of all tasks can be met with 100% certainty. This, however, requires one to know the exact characteristics of all tasks *a priori*. For example, one would have to know *a priori* the total number of tasks as well as the computation and resource requirements of all tasks at all time. Furthermore, one would have to know the expected changes in the environment over time because the environment can significantly affect the behavior of the system. Needless to say, it is unlikely that one would have all this information at design time.

For more complex systems, the semantics of predictability varies from one task to another. Some critical tasks may still require a 100% guarantee that their constraints will be satisfied. Periodic tasks with hard deadlines usually belong to this category. As discussed above, complete characteristics of these tasks would have to be known *a priori*. Other tasks may be satisfied with either *probabilistic* or *run-time deterministic* guarantees. The word "probabilistic guarantee" can also have multiple semantics. In some cases, it means that a certain fraction of tasks are guaranteed to meet their constraints. In other cases, it means that a given task has a certain probability of meeting its constraints. Note that, in some cases, these two notions are equivalent.

In contrast, "run-time deterministic guarantee" means that when a task is activated the system determines whether or not the task's constraints can be satisfied without jeopardizing the guarantees provided to other tasks. If the constraints can be satisfied, then the task is accepted and the task is given a 100% guarantee of meeting the constraints. On the other hand, if the constraints cannot be satisfied, the task is not accepted by the system. Consequently, at design time, one cannot predict which task will meet all its constraints. However, while the system is in operation, each task knows whether its constraints can be satisfied. This type of guarantee is often used for dynamically arriving aperiodic tasks or dynamic load sharing.

It is important to note that deadline guarantees are possible only if task characteristics like the execution and arrival times of tasks are given *a priori*. It is difficult in practice to obtain exact information of task characteristics, so the worst case values are assumed or derived from extensive simulations, testing, or other means. These values may not be "true" worst case values and the actual values may exceed them on some rare occasions. However, the

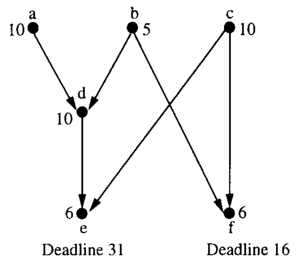


Fig. 1. Example of a real-time application.

system designer will still use the assumed worst case values, since there is no other alternative. Chodrow *et al.* [16] called such an event a *specification violation* and proposed to use an on-line monitor to record violations. This record can later be used to modify the assumed worst case values.

All aspects of the system must be appropriately designed to provide any of the above guarantees. The architecture of each node, the communication subsystem, the operating system, and the programming languages have to support the notion of guarantee at all levels of abstraction. The rest of this paper outlines how the notion of guarantee is supported by various components of a real-time system.

II. Scheduling

Given a set of real-time tasks and the resources in the system, task assignment and scheduling is the process of determining where and when each task will execute. For example, consider a real-time application with six tasks $a, b, \cdots, f$ with precedence and timing constraints as shown in Fig. 1. In this figure, the vertices represent the tasks and the directed arcs represent the precedence relation. For instance, tasks a and b must complete before task d can begin, because there are directed arcs from a to d and from b to d. Each vertex has a weight associated with it which represents the time required to execute the corresponding task. The timing constraints are such that tasks e and f must complete within 31 and 16 time units, respectively, assuming that all tasks are ready to execute at time 0 subject to their precedence constraints.

Figure 2(a) and (b) shows two possible schedules for this application on a system with two processors. In Fig. 2(a), tasks b, c, and f are assigned to one processor and the remaining tasks are assigned to the other processor. On the first processor, task b executes from time 0 to 5, task c executes from time 5 to 15, and so on. Likewise, on the second processor. Since, in this schedule, task f does not complete its execution by its deadline of 16, a key timing constraint of the application is not satisfied. However, the schedule shown in Fig. 2(b) satisfies the precedence and timing constraints of all tasks, and therefore, it is better suited for real-time applications. The problem of scheduling is to identify schedules like the one in Fig. 2(b) given the application as in Fig. 1.

Although the problem of scheduling occurs in many other areas including parallel processing, factory floor management, and high-level synthesis, there are several key con-

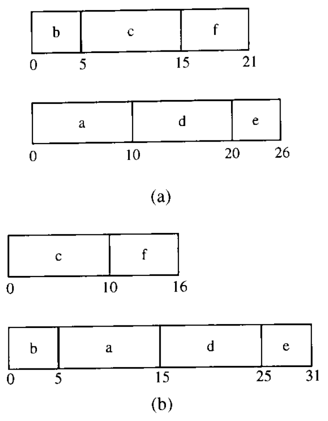


Fig. 2. Illustration of difference between real-time and non-real-time scheduling problems. (a) Infeasible schedule. (b) Feasible schedule.

straints in real-time applications which make the problem substantially different. In most non-real-time applications, the main objective of scheduling is to minimize the total time required to execute all the tasks in the application; while in a real-time application, the objective is to meet the timing constraints of the individual tasks. For example, in Fig. 2(a), all tasks complete their execution by time 26 as compared to time 31 in Fig. 2(b). Hence, the first schedule is probably preferable in a non-real-time application, whereas only the second schedule is acceptable to the real-time application because some of the timing constraints are not satisfied in the first schedule.

Scheduling algorithms for real-time applications can be classified along many dimensions. Some scheduling algorithms deal only with periodic tasks while others are intended only for aperiodic tasks. There are very few algorithms which deal with both types of tasks since the approach needed to deal with them differ considerably. Likewise, some scheduling algorithms can only handle preemptible tasks while others can handle nonpreemptible tasks. Criticality, independence, resource and placement constraints, and strictness of deadlines are examples of other characteristics of real-time tasks which affect the nature of the scheduling algorithm.

Scheduling algorithms also vary significantly depending on the type of computer system they are intended for. Some algorithms are for uniprocessors while others are for multiprocessor systems. Among multiprocessor systems, the scheduling algorithms can depend on whether it is a shared-memory or a message-passing system. The type of interconnection network can also affect the scheduling algorithm.

Finally, there can be difference in objectives of the scheduling algorithms. Most algorithms assume that tasks have either hard or firm deadlines. However, recently some algorithms have been proposed which assume that a task is composed of both a mandatory and an optional part [17]. The mandatory part must be completed by the deadline while the optional part may or may not.

A. Scheduling for Uniprocessor Systems

The most notable work in this area was done by Liu and Layland [51]. They proposed a static priority algorithm called *Rate Monotonic Scheduling* (RMS) and a dynamic priority algorithm called *Earliest Deadline First* (EDF) for scheduling a set of independent, preemptive, periodic tasks with hard deadlines on a uniprocessor system. They presented a simple characterization of the set of tasks schedulable by these two algorithms. They showed that RMS is optimal among all static priority algorithms in the sense that any task set schedulable by a static priority algorithm is also schedulable by RMS. Similarly, EDF was shown to be optimal among all algorithms (static or dynamic) for uniprocessor scheduling of real-time tasks with the above characteristics.

At first glance, it may seem that RMS and EDF operate on a fairly restrictive set of tasks. However, since the original work by Liu and Layland, these algorithms have been extended in various ways to deal with other constraints like dependency, periodicity, and deadline. For example, solutions have been proposed for controlled access to shared resources and also for handling aperiodic and sporadic tasks. It has been shown that the theory developed in the context of uniprocessor task scheduling is applicable in more general situations such as distributed scheduling of messages in a multiple access network like the FDDI. For a more detailed description of the extensions to deal with other constraints and the generalized theory of rate-monotonic scheduling refer to the articles by Sha *et al.* and by Ramamritham *et al.* in this issue.

B. Scheduling for Multiple-Processor Systems

The issues in multiple-processor scheduling of real-time tasks are significantly different from that in uniprocessor scheduling. The problem in multiprocessor scheduling is not only to determine when a given task executes but also where it executes. That is, task assignment and scheduling must be dealt with. There are also issues related to availability of necessary resources at the processor at which a task is scheduled to execute, contention for communication across a network, etc. These issues make the problem substantially harder to solve.

As in the case of uniprocessor systems, scheduling in a multiple-processor system can be either static or dynamic. In static algorithms, the assignment of tasks to processors and the times at which a task executes are determined *a priori* [65]. Unfortunately, even the problem of assignment of tasks to processors is difficult with or without timing constraints. The problem is NP-hard in most cases, e.g., finding optimal assignment of tasks with an arbitrary communication graph to four or more processors with different speeds is known to be NP-hard [10]. Therefore, most existing approaches try to find suboptimal solutions using some heuristics.

The most commonly used heuristics come under a category called *list scheduling*. A heuristic rule is used to first order all the tasks in the system. Tasks are considered for scheduling in this order and are often assigned and scheduled on a processor on which they can complete the earliest. For example, one can order the tasks in the increasing order of their laxities[1] and use this order for scheduling. If the deadlines of some tasks are not met using this approach, then some solutions use backtracking and rescheduling to find a feasible schedule [65], [66], [104]. Other approaches for static scheduling use optimization techniques like branch-and-bound, simulated annealing, and genetic algorithms to find a feasible schedule [56], [100].

In spite of the difficulties in finding an optimal solution, static algorithms are often used to schedule periodic tasks with hard deadlines [65]. The main advantage is that if a solution is found, then one can be sure that all the deadlines can be guaranteed. However, this approach is not applicable to aperiodic tasks whose arrival times and deadlines are usually not known *a priori*. Scheduling such tasks in a multiprocessor system requires dynamic algorithms. Dynamic scheduling algorithms can be either centralized or distributed. In the centralized scheme, all tasks arrive at a central processor from where they are distributed to other processors in the system [57], [107], [106]. The main advantage of a centralized scheme is that only the central processor needs to be aware of the load on the other processors to determine whether the deadline of an incoming task can be guaranteed.

In a distributed dynamic scheduling scheme, tasks arrive independently at each processor. When a task arrives at a processor, the local scheduler at that processor determines whether or not it can guarantee the constraints of the incoming task, which is termed the *transfer policy*. The task is accepted if the constraints can be guaranteed without jeopardizing the guarantees which have been provided earlier. If not, the local scheduler tries to find a processor which can guarantee the constraints of the task, which is called the *location policy*. The schemes in literature differ in the algorithms used to identify the processor to transfer a task [22], [37], [36], [67], [80], which is based on how to collect and maintain the state information of other nodes, termed the *information policy*.

III. Real-Time Architectures

Design of a real-time architecture involves issues at two levels: node and system levels. At the node level, each processor must provide speed and predictability in executing real-time tasks, handling interrupts, and interacting with external world. This can be accomplished by making operations like instruction execution, memory accesses, and context switching more predictable. To make these "small" operations more predictable, real-time systems seldom use virtual memory because page faults cause unpredictable or very long delays in accessing memories. Similarly, real-time systems also try to avoid the use of caches because uncertainty of cache hit/miss causes unpredictable memory access delays. However, it may be very difficult

[1] The laxity of a task is the latest time the task must begin its execution to meet its deadline.

to avoid caches because real-time systems are often built using contemporary off-the-shelf microprocessors which come with multilevel on-chip caches to optimize average performance. In fact, multiple instruction and data pipelines and branch prediction strategies commonly available in today's off-the-shelf microprocessors also make it very difficult to achieve predictability at the node level.

At the system level, internode communication and fault tolerance are two main issues which make it difficult to achieve predictability. However, these issues are also unavoidable because the high performance and high reliability of distributed systems make them attractive for real-time applications. Therefore, presented below is a brief discussion on issues and solutions related to distributed real-time architectures.

A. High-Level Architectural Issues

At the highest level, a distributed system is comprised of a set of nodes communicating through an interconnection network. Each node may itself be a multiprocessor comprised of application, system, and network processors, shared memory segment, and I/O interfaces [19], [79], [96]. Although the application processor may be an off-the-shelf product, the system and network processors usually have to be custom-designed because they provide the specialized support necessary for real-time applications [79]. Memory subsystem may also have to be carefully designed to provide fast and reliable communication between the processors at a node. For example, the memory subsystem may support a mailbox facility to support efficient inter-processor communication within a node of a distributed system.

The nodes of the system must be interconnected by a suitable communication network. For small and earlier systems, the network was a custom-designed broadcast bus with redundancy to meet the fault-tolerance requirements [19], [42], [92]. More recently, however, the interconnection is either a high-speed token ring or a point-to-point network with a carefully chosen topology. For example, the Spring system at University of Massachusetts uses a high-speed optical interconnect called Scramnet [94], whereas the HARTS project at University of Michigan uses a point-to-point interconnection network called C-wrapped hexagonal mesh topology [15], [79].

Irrespective of the exact topology, the network should support scalability, ease of implementation, and reliability. It should also have support for efficient one-to-one as well as one-to-many communications. For instance, the C-wrapped hexagonal mesh topology used in HARTS has a $\theta(1)$ algorithm for computing all the shortest paths between any two nodes in the system. The information about all shortest paths can also be easily encoded using three integers and included as part of each message so that the intermediate nodes need not do much computation. The routing algorithm can fully exploit advanced switching techniques like virtual cut-through and wormhole routing in which packets do not always have to be buffered at intermediate nodes before being forwarded to next node in the route. Broadcasting can also be done fairly efficiently and in a fault-tolerant manner using the multiple disjoint paths between any two nodes in the system [40]. Such capabilities are very important because reliable and timely exchange of information is crucial to distributed execution of any real-time application.

B. Low-Level Architectural Issues

Low-level architectural issues involve packet processing, routing, and error/flow control. In a distributed real-time system, there are additional issues related to support for meeting deadlines, time management, and housekeeping. Since support of these low-level issues impedes the execution of application tasks, nodes in a distributed real-time system usually have a custom-designed processor for handling these chores. In the description below, this special processor is referred to as the *network processor* (NP).

The main function of NP is to execute operations necessary to deliver a message from a source task to its intended recipient(s). In particular, when an application task wants to transmit a message, it provides the NP with information about the intended recipient(s) and the location of the message data and then relies on the NP to ensure that the information reaches the recipients in a reliable and timely fashion. The NP may also be responsible the functions in the transport, network, and the data link layers of the OSI reference model [99].

More specifically, at the transport layer, the NP must establish connections between the source and destination nodes. It must also handle end-to-end error detection and message retransmission. At the network layer, the NP may have to select primary and alternate routes, allocate bandwidths necessary to guarantee timely delivery, packetize the information into smaller data blocks and segments, and reassembles packets at the destination node. In point-to-point interconnections, the NP must support and choose appropriate switching method like virtual cut-through wormhole routing, store-and-forward, and circuit switching [79]. In token rings, the NP must select suitable protocol parameters to guarantee the deadlines of all messages [2], [30], [112]. At the data link layer, the NP must provide access to the network for the messages. It must perform framing and synchronization, as well as packet sequencing.

The NP must also have support for multiple levels of interrupts to manage messages with different priority levels. The hardware should provide the requisite number of levels of interrupts, so that urgent messages can be given higher priority over less urgent ones. The NP must implement buffer management policies that maximize utilization of buffer space, but guarantee the availability of buffers to the highest priority messages. Similarly, if noncritical messages hold other resources that are needed by more critical ones, NP must provide means for preemption of such resources for use by the critical messages.

The NP may also have to monitor the state of the network in terms of traffic load and link failures. The traffic load affects the ability of the NP to send real-time messages

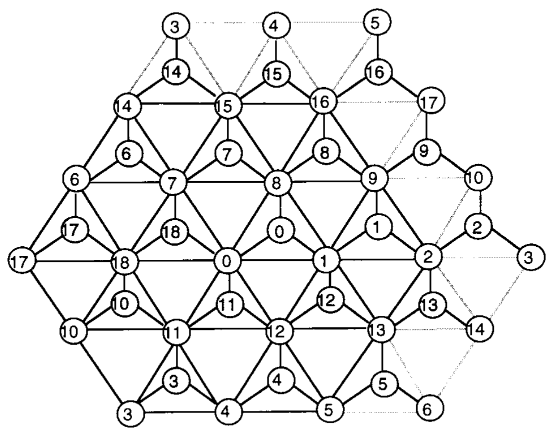


Fig. 3. I/O controller placement.

to other processors, while link failures affects the system reliability. It may also have to keep track of the processing load of its host (or hosts), and use the information for load balancing/sharing and task migration operations.

C. I/O Architecture

Most work on distributed computing systems has centered on interconnection networks, programming and communications paradigms, and algorithms. However, little has been done specifically about the I/O subsystem in a real-time environment, despite its obvious importance. Clearly, a real-time computer can process data no faster than it can acquire the data from sensors and operators. Note that I/O devices in a real-time environment are sensors, actuators, and displays, whereas they are magnetic disks and tapes for general-purpose systems. Due to the distinct timing and reliability requirements of the former, solutions to the latter are not usually applicable to the real-time environment.

To avoid the accessibility problems of nondistributed I/O, I/O devices need to be distributed and managed by relatively simple, and reliable, controllers. Moreover, to improve both accessibility (reliability) and performance, there must be multiple access paths (called *multiaccessibility* or *multiownership*) to these I/O devices.

One possible solution to provide multiaccessibility in HARTS is described in [82]. The I/O devices are clustered together and a controller is assigned to manage access to the devices in each cluster. The controller has a set of full-duplex links to certain nodes of the distributed system. In order to limit the number of links in each controller while providing multiaccessibility, a controller is connected to three nodes in the system as shown in the Fig. 3. Since each controller can be accessed by three nodes, different management protocols are proposed for handling the I/O requests. In a static scheme, one node is assigned the primary responsibility of managing the controller with the proviso that the other nodes can take over control if the primary node becomes faulty. In a dynamic scheme, all three nodes connected to a controller manage the controller using a more complicated protocol [82].

An alternative approach for connecting the I/O controllers to the nodes of the systems is described in [13], [69]. In this approach, an I/O controller is connected to only one node. However, the placement of the I/O controllers is done in such a way that a node is at most one hop away from a node which has an I/O controller connected to it. To achieve fault tolerance, schemes for placement of I/O controllers are also proposed in which a node is at most one hop away from j nodes with I/O controllers, where j is a design parameter.

Although the above solutions have some headway in dealing with the I/O architectural issue, there is a lot of work which needs to be done.

IV. Real-Time Operating Systems

Unlike the traditional operating systems, predictability is one of the main requirements of a real-time operating system (RTOS). Therefore, many of the basic paradigms found in today's general-purpose operating systems are not applicable to RTOS's. For example, it is not very important in a real-time operating system to provide extensive support for file systems, virtual memory, and security. However, it is crucial to provide support for fast context switching, quick and predictable handling of interrupts, and for scheduling with timing and dependability constraints.

Operating systems which provide such support can broadly classified into three categories: proprietary kernels, commercial operating systems with real-time extensions, and research-oriented operating systems. Proprietary kernels are small and fast. They achieve predictability by supporting only those primitives which can be shown to have a bounded execution time. They are mainly intended for small embedded applications such as instrumentation, intelligent peripherals, and simple process control. Commercial operating systems with real-time extensions, on the other hand, offer familiar interfaces with moderate support for real-time systems. They are generally slower and less predictable than the proprietary kernels. However, due to the familiarity of the interfaces, it is well-suited for software development and for large applications in which the consequences of missing a few deadlines is not very severe. In contrast to the above two categories, research-oriented operating systems focus on specific real-time issues. They are often weak on support for the issues which are not their focus. However, they play a key role in developing future operating systems. For more details on the various real-time operating systems refer to the article by Ramamritham *et al.* in this issue.

V. Real-Time Communication

Consider a real-time system with three cooperating tasks T_1, T_2, and T_3. Suppose T_1 is responsible for reading a sensor and providing the sensed data to T_2, T_2 is responsible for performing the control law computation and sending the result to T_3, and finally, T_3 is responsible for controlling the actuator based on the results from T_2. Further, suppose this entire operation has to be repeated once every R seconds. Since the sensor, the computation unit, and the actuator may not be physically close to each other, these three tasks may be executing on three different nodes of a distributed system. Therefore, T_1 and T_2 have to send messages after they finish their execution. Also, T_3 must complete its execution within a deadline $D \leq R$.

This simple example highlights the importance of time-constrained communication in a real-time system. To ensure that T_3 finishes its execution within D time units, not only must tasks T_1 and T_2 complete their execution in time, but also the information exchange between T_1 and T_2, and between T_2 and T_3 must be completed in time. Failure to complete the information exchange in time will delay the start (and hence the completion) of subsequent tasks and thus cause the application to miss its deadline. For instance, in the above example, an unacceptable delay in the delivery of information from T_1 to T_2 will delay the start and completion of T_2 and T_3 and thus possibly result in T_3 missing its deadline. Since the consequences of a task missing its deadline are severe in many of these applications, it is imperative to design the communication subsystem in such a way that all information exchange are completed within the deadline assigned to them.

Due to its importance, real-time communication has received considerable attention in literature. Presented below is a brief survey of existing work in real-time communication.

A. Multiple-Access Networks

In the traditional non-real-time Carrier Sense Multiple Access/Collision Detect (CSMA/CD) protocol, a node transmits its local messages using its own local scheduling policy. No effort is made to coordinate with other nodes to establish a network-wide policy. Even if all nodes use a First-In-First-Out (FIFO) policy for transmitting their local messages, the transmission on the network will not follow a network-wide FIFO policy due to collisions in the medium. To rectify this problem, Molle proposed a modification to the CSMA/CD protocol that implements a network-wide FIFO policy [59]. Each node has two clocks: real clock and a virtual clock. The real clock runs along the time axis and maintain elapsed time on each node. The virtual clock runs along the "arrival time" axis. It runs only when the channel is idle and when it runs it does so at a faster rate than the real clock. When a message arrives it is timestamped with its time of arrival according to the local real clock. A message is transmitted only when its timestamp equals the time on the virtual clock. Molle proves that this scheme implements a global FIFO policy except for ties in arrival time. The discrete nature of clocks also causes some ties which would not have occurred otherwise.

Zhao and Ramamritham extended Molle's scheme to real-time systems [105]. Instead of timestamping a message with its arrival time, it is stamped with a value which depends on the scheduling policy. For example, to implement a network-wide Minimum-Laxity-First policy, each message is stamped with its laxity when it arrives at a node. The message is then transmitted on the network only when the virtual clock reads the value stamped on the message. The authors of [105] compared four different network-wide scheduling policies: First-In-First-Out, Earliest-Deadline-First, Minimum-Laxity-First, and Shortest-Length-First. Through simulation they conclude that the Earliest-Deadline-First and the Minimum-Laxity-First policies perform the best in terms of certain key metrics such as the percentage of missed deadlines and effective channel utilization.

The main limitation of the virtual time protocols is that they do not account for the past history of the channel in selecting a message for transmission. To alleviate this limitation, window-based protocols were proposed for CSMA/CD networks [47], [108]. As in the case of virtual time protocols, a message is stamped with a value when it arrives at a node. However, instead of linearly searching for a message to transmit when the channel is idle, the window-based protocols use a divide-and-conquer technique on the value axis to identify a message for transmission. That is, a message will be transmitted if its arrival time falls in the time window determined by the divide-and-conquer technique. This greatly improves the performance of the scheduling policies.

Note that, due to intrinsic problems in the CSMA/CD protocol, the above schemes cannot provide a deterministic guarantee to any message. Therefore, they cannot be used for guaranteeing the deadlines of periodic messages. Token rings and/or token buses are better suited for such messages. Strosnider and Marchok extended the Rate Monotonic Algorithm (RMA) to schedule transmission of periodic messages on an IEEE 802.5 token ring network [98]. They used the priority mechanism of the IEEE 802.5 standard to implement the priority scheme of the RMA. However, since IEEE 802.5 does not provide extensive support for priorities, this scheme is limited to a maximum of four or five periodic messages.

Agrawal *et al.* [2] use the timed-token protocol to guarantee the deadlines of periodic messages in token ring networks. In a timed-token protocol, each node is guaranteed a certain duration for transmitting messages each time it receives the token. As a result, a node can predict the minimum bandwidth available to it and thus guarantee the deadlines of periodic messages. Agrawal *et al.* compare several schemes for allocating the time guaranteed to each node and they show that a scheme called the *normalized proportional allocation* can guarantee all periodic requests as long as their total utilization is less than 33%. Since then, that scheme has been extended in various ways: the scheme in [112] requires local optimization as opposed to global optimization and allows message deadlines to be arbitrary and the schemes in [1], [30] can guarantee much higher utilizations.

The problem with multiple-access networks is that they are susceptible to single-point failures. For instance, a failure in the shared medium can completely disrupt communication between any two nodes in the system. This problem can be partly overcome using redundancy, as in the case of dual-token rings in FDDI [77]. However, the system incurs a severe performance penalty if one of these rings fails.

This is in direct contrast to point-to-point interconnection networks where there are multiple disjoint routes between any two nodes in the system. Furthermore, failure of a few links or nodes has often only little impact on the performance of a point-to-point network [14], [62]. Therefore, these networks have been receiving considerable attention.

B. Point-to-Point Interconnection Networks

Unlike in multiple-access networks, not all pairs of nodes in a point-to-point interconnection topology have a direct connection between them. As a result, messages may have to be relayed by one or more intermediate nodes before reaching their destinations. This complicates the problem of guaranteeing the deadlines of real-time messages.

In point-to-point networks, Ferrari and his colleagues from the University of California at Berkeley proposed a communication abstraction called the *real-time channel* for guaranteeing the timely delivery of real-time messages [4], [5], [23]. A real-time channel is a simplex (unidirectional) connection between a source and a destination with a guaranteed end-to-end delay. A real-time channel is represented by a three-tuple (T, C, D), where T represents the minimum intermessage generation time, C the maximum transmission time per message (closely related to the maximum message length), and D the user-specified end-to-end delay bound.

The concept of real-time channel uses two techniques to guarantee the end-to-end message delay bound: admission control via the channel establishment procedure and the deadline scheduling of message transmissions. So, real-time channels are realized with two protocols: a channel establishment protocol, and a message transmission protocol.

The channel establishment protocol handles requests for the establishment of real-time channels. The protocol first selects a route between the source and the destination according to the given criterion, e.g., traffic balancing [39] or use of minimum network resources [110]. The protocol then checks whether the requested end-to-end message delivery delay bound D can be guaranteed for a real-time channel under the current network-load condition. The channel establishment request is granted only if the requested delay bound can be guaranteed and if there is enough space to buffer the messages of this channel. Note that the selection of a route affects the system in two ways. First, whether or not the given request can be established depends on the route, because it depends on the delays that each intermediate node can guarantee. Second, once a channel is established through a particular route, it reduces the likelihood of establishing future channels through any nodes in the route. Therefore, one has to be careful in selecting the route. A solution to the first problem is discussed in [40]. Regarding the second problem, Ferrari and Verma derive a simple solution under the assumption that the sum of the message transmission times over all the channels passing through a link is not larger than the minimum message interarrival times of these channels [23]. Kandlur *et al.* [40] improved this solution significantly by deriving a necessary condition for channel schedulability, i.e., delay guarantees are calculated using a prioity-based message scheduling scheme but on-line message transmissions are scheduled according to a multiclass EDD policy. More recently, Zheng and Shin have made further improvements by deriving a necessary *and* sufficient condition for the schedulability of a set of real-time channels over a link [111]. The same authors have also enhanced real-time channels to tolerate certain types of component faults while retaining their timeliness properties [110], [109].

The message transmission protocol implements the deadline scheduling of message transmissions. It specifies how a message is divided into packets, and how deadlines of a packet over the links it traverses are calculated. These two parts are closely related. The calculation of the end-to-end message delivery delay bound depends on the transmission protocol used, and the transmission protocol must be designed such that the requested delay bound can be guaranteed and easily verified. For more details on message transmission and channel establishment protocols in point-to-point interconnection networks refer to the article by Aras *et al.* in this issue.

Unfortunately, the concept of real-time channel cannot be used to service those messages without information on T and C. Hence, it is very difficult to provide a run-time deterministic guarantee to these messages. Techniques designed to increase the probability of their timely delivery have been proposed [24], [72], [73]. These techniques are based on the existence of multiple disjoint paths between any two nodes in a point-to-point interconnection network. Garcia-Molina *et al.* proposed sending two copies of all critical messages along two disjoint paths [24]. They also considered the possibility of splitting a large message into two and sending each half on different routes. In contrast to the above scheme, Ramanathan and Shin statically identified the optimal number of copies to send based on the criticality of the message, the number of hops it has to traverse, and the traffic intensity [72], [73]. They did not account for deadlines. An extension of this approach which dynamically accounts for the deadlines and also deals with networks which support more sophisticated switching schemes like virtual cut-through have been proposed in [31].

An additional issue in servicing messages in point-to-point interconnection topology is how to schedule the multiple random requests at a given node in order to maximize the likelihood of timely delivery. In multiple access networks, scheduling policies such as Earliest-Deadline-First have been shown to work well. Unfortunately, these policies do not work well in point-to-point interconnection topologies [71]. A better scheduling policy that is based on the cost the system will incur if a message misses its deadline is proposed in [71].

VI. Fault Tolerance in Real-Time Systems

A. Relationship Between Fault Tolerance and Real-Time Computing

Fault tolerance is informally defined as the ability of a system to deliver the expected service even in the presence of faults. A common misconception about real-time computing is that fault tolerance is orthogonal to real-time requirements. It is often assumed that the availability and reliability requirements of a system can be addressed independent of its timing constraints. This assumption, however, does not consider the distinguishing characteristic of real-time computing: *the correctness of a system is dependent not only on the correctness of its result, but also on meeting stringent timing requirements.* In other words, a real-time system may fail to function correctly either because of errors in its hardware and/or software or because of not responding in time to meet the timing requirements that are usually imposed by its "environment." Hence, a real-time system can be viewed as one that must deliver the expected service in a timely manner even in the presence of faults. A missed deadline can be potentially as disastrous as a system crash or an incorrect behavior of a critical task, e.g., a digital control system may lose stability.

In fact, if the logical correctness of a system may be dependent on the timing correctness of certain components, separating the functional specification from the timing specification is a very difficult task. Moreover, timeliness and fault tolerance could sometimes pull each other into opposite directions. For example, frequent extra checks and exotic error recovery routines will enhance fault tolerance but may increase the chance of missing the deadlines of application tasks.

When a system specification requires certain service in a timely manner, then the inability of the system to meet the specified timing constraint can be viewed as a failure. However, a simple approach of applying existing fault-tolerant system design methods by treating a missed deadline as a timing fault does not fully address the needs of real-time applications. The fundamental difference is that real-time systems must be predictable, even in the presence of faults. Hence, fault tolerance and real-time requirements must be considered jointly and simultaneously when designing such systems. The challenge is to include the timing and the fault-tolerance requirements in the specification of the system at every level of abstraction and to adopt a design methodology that considers system predictability even during fault detection, isolation, system reconfiguration, and recovery phases. Formal specification of the reliability requirements and their impact on meeting timing constraints is an area which requires further research. Determining the timing constraints on a system from its availability requirements is a very difficult problem.

B. Space and Time Tradeoff

The design methodologies for fault-tolerant systems have often been characterized by the tradeoff between time and space redundancy. In non-real-time systems, however, time is treated as a cheap resource and most methods concentrate on space optimization. In a real-time environment, the tendency would be to trade space for time since meeting the stringent timing constraints is essential in ensuring correct system behavior. Although time–space tradeoff forms the basis for most fault-tolerant system design methodologies, it is unclear whether it is an appropriate paradigm for characterizing fault tolerance in a real-time environment. In particular, redundancy must be considered in the context of achieving both predictability and dependability in a system. For example, tolerating transient faults by retrying a computation is an acceptable technique if the timing constraints can be met. The same assertion holds for techniques based on the notion of recovery blocks where a different version of the software module is used in the retry. However, alternative approaches must be considered when time is a scarce resource. In particular, the quality of the computation can become a third dimension in the design space. That is, in a new twist on the principle of graceful degradation, a fault in a real-time system could result in a (temporary) reduction in the quality of the services provided in order to allow the system to continue to meet critical task deadlines. Although general methods that delete or reduce the number of less critical tasks would certainly fall into this category, "quality" must ultimately be defined in terms of the detailed semantics of the application. Real-time control systems, for example, are characterized by continuous variables whose values can be approximated or estimated if time does not permit precise computation. The imprecise computation approach [52] is one technique that sacrifices accuracy for time in iteratively improving calculations. As discussed below, trading space for time also has potential limits since spatial redundancy introduces additional overhead (in time) for managing the redundancy (see [45] for an example).

C. Predictable Redundancy Management

Although advances in distributed and parallel systems provide the opportunities for achieving real-time performance while satisfying fault-tolerance requirements, using the inherent redundancy provided by these systems is not free. For example, the overhead associated with synchronization and the nondeterminism due to communication delay contribute to the complexity of building systems with predictable timing behavior. Predictable redundancy management remains an open research problem. For example, a set of identical servers on multiple processors provide fault tolerance in a system with crash or performance failures. However, predictable redundancy management requires to solve such issues as synchronization of servers, agreement on the order of request messages, and the cost of failure detection and recovery [45]. In this case, redundancy in space incurs additional cost in time. The overhead associated with managing redundancy must be quantified precisely so that certain guarantees about the real-time behavior of the system can be made.

Managing redundancy in a predictable fashion is related closely to the demands on real-time scheduling theory. Satisfying the timing requirements of a real-time system demands the scheduling of system resources such that the timing behavior of the system is "understandable, predictable, and maintainable." Most existing scheduling algorithms consider one resource at a time and ignore fault tolerance. The requirement for meeting timing constraints in the presence of faults imposes additional demands on the scheduling algorithms. New resource-allocation techniques are necessary to address the predictability and reliability requirements of complex real-time systems. A simple case of this was treated in [44].

D. Run-Time Monitoring

In designing real-time systems, we often make assumptions about the behavior of the system and its environment. These assumptions take many forms: upper bounds on interprocess communication delay and task execution time, deadlines on the execution of tasks, or minimum separations between occurrences of two events. They are often made to deal with the unpredictability of the external environment or to simplify a problem that is otherwise intractable or very hard to solve. Such assumptions may be expressed as part of the formal specification of the system or as scheduling requirements on the real-time tasks. Despite the contribution of formal verification methods and recent advances in real-time scheduling, the need to perform run-time monitoring of these systems is not diminished for several reasons: the execution environment of most systems is imperfect and the interaction with the external world introduces additional unpredictability; design assumptions can be violated at run time due to unexpected conditions such as transient overload; application of formal techniques or scheduling algorithms in turn requires assumptions about the underlying system; and it may be infeasible (or impossible) to verify formally some properties at design time, thus further necessitating run-time checks [38], [16].

Run-time monitoring of a system requires timestamping and recording of the relevant event occurrences, analyzing the past history as other events are recorded, and providing feedback to the rest of the system. The nonintrusiveness requirement of real-time monitoring often leads to use of special monitoring hardware [28], [102]. Many important issues must be addressed before run-time monitoring is fully utilized in real-time systems. Some of these issues are:

- *Language support:* What is an appropriate set of language constructs for the specification of run-time constraints? Should the specification language be tied closely to the underlying implementation language?
- *Run-time system support:* What level of support should be provided by the operating system? What internal operating system events, such as task preemption, should be made visible to the monitoring facility?
- *Scheduling support:* How intrusive is run-time system monitoring on the critical tasks within a system? Is it possible to make the intrusiveness of run-time monitoring predictable? See [101] for more on this.

In addition to detecting violation of design assumptions, run-time monitoring can be used to detect application-specific exception conditions. One can envision a system in which specification-based fault detection is done by a monitoring facility [38]. Furthermore, beside detecting exception conditions, a run-time monitoring facility can provide feedback to the rest of the system. The information collected by the monitoring facility can be used to provide feedback to the system operator, the application tasks, or the scheduler [28]. For example, exceeding the maximum computation time estimated for a task can be reported by the monitoring facility. The feedback to the scheduler can be used to build a robust system that is capable of adapting to the changes in the environment and the system load. Investigating the utilities of monitoring application tasks and operating system events as a way of providing feedback to the rest of the system is an area of ongoing research. A related topic is scheduling run-time monitoring tasks with the real-time application tasks in a system.

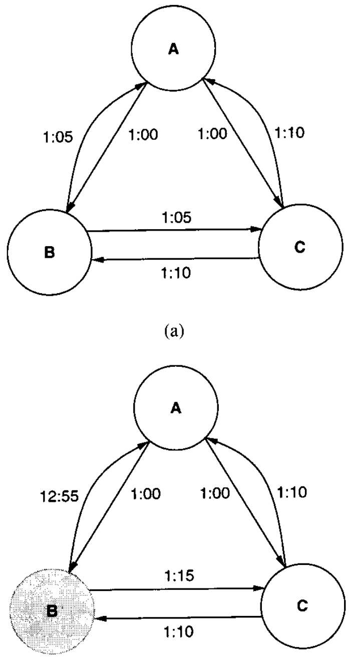


Fig. 4. Example of a Byzantine fault in clock synchronization. (a) All clocks nonfaulty. (b) Faulty clock at node B.

VII. Clock Synchronization

A global time base has been widely recognized as an important requirement in distributed systems. It can simplify the design of many fault-tolerant algorithms used for interprocess communication, checkpointing and rollback recovery, resource allocation, and transaction processing. It can also facilitate the use of deadlines and timeouts that are essential for correct operation of any distributed real-time system.

A global time base can be established by synchronizing all the clocks in the system. This would not have been a serious problem had all the clocks including the faulty ones behaved consistently with one another. However, when some of the faulty clocks can behave in any arbitrary manner, synchronizing all the clocks can pose some serious problems. This problems is best illustrated by the example in Fig. 4. Figure 4 shows a three-node system in which each node has a clock of its own. Clocks are synchronized by adjusting each to the median of the three clock values. This "intuitively correct" algorithm works fine as long as all the clocks are consistent in their behavior as illustrated in Fig. 4(a). However, if one of the clocks is faulty and lies to the other two clocks, then the two nonfaulty clocks cannot be synchronized. For example, in Fig. 4(b), the faulty clock B lies to clocks A and C. As a result, clocks A and C do not make any corrections because both think that each is the median clock.

Lamport and Melliar-Smith were the first to study the three-clock synchronization problem in the presence of arbitrary fault behavior [49]. They coined the term *Byzantine fault* to refer to the fault model in which a faulty clock can exhibit arbitrary behavior including, but not limited to, intentionally and maliciously lying about its value to other clocks in the system. They showed that in the presence of Byzantine faults there is no algorithm that can guarantee synchronization of the nonfaulty clocks in a three-node system. They also showed that $3m+1$ clocks are sufficient to ensure synchronization of the nonfaulty clocks in the presence of m Byzantine faults. This condition was later proved to be necessary as well as sufficient to ensure synchronization in the presence of Byzantine faults [20].

Since the initial study by Lamport and Melliar-Smith, the problem of clock synchronization in the presence of Byzantine faults has been studied extensively by several researchers. A survey of these solutions can be found in [74]. The solutions proposed in literature can be categorized as either a software or a hardware approach. The software approach is flexible and economical but requires additional messages to be exchanged solely for synchronization [29], [49], [54], [93]. The basic idea of software synchronization algorithms is as follows. Each node has a logical clock that provides a time base for all the activities on that node. This logical clock is derived from the hardware clock on that node, but it usually has a much larger granularity (than the hardware clock). The nodes periodically exchange their clock value and adjust their local clock based on the values of the other clocks. The software synchronization algorithms differ in the manner in which the nodes exchange their clock values and in the way they readjust their clocks.

In convergence-averaging algorithms, the nodes pre-agree on the times for resynchronization. When a node's clock reaches one of these resynchronization times, it broadcasts its clock value and then waits for a pre-agreed duration to receive clock values from other nodes. At the end of the waiting period, the node estimates its clock skew with respect to each of the other nodes based on the times at which its receives their clock values. The node then computes a fault-tolerant average of the estimated skews and uses this average to correct the local clock. For example in [49], an arithmetic mean of the estimated skews is used to correct the clocks. To limit the impact of malicious clocks on the mean, the estimated skew with respect to each node is compared against a threshold and skews greater than the threshold are set to zero before computing the mean. In contrast, in [54], each node limits the impact of malicious clocks by first discarding the m highest and the m lowest estimated skews and then uses the midpoint of the remaining skews as its correction, where m is the maximum number of faulty clocks to be tolerated.

On the other hand, a class of software synchronization algorithms called the *convergence non-averaging* do not use the principle of averaging to synchronize their clocks [29], [93]. Instead, the nodes pre-agree on the times for resynchronization. When a node's local clock reaches one of these times, it tries to become the "synchronizer" of the entire system by sending a message to other nodes asking them to adjust their clocks. Upon receiving this message, other nodes check the validity of the message and then set their clocks to the pre-agreed resynchronization time if the message is deemed valid. The exact nature of the validity check depends on the algorithm.

A third class of software algorithms are often called the *consistency-based* algorithms. These algorithms treat clock values as data and try to ensure agreement among the clock values using an interactive consistency algorithm [49]. Unlike the convergence averaging and convergence non-averaging algorithms, these algorithms do not need initial synchronization among clocks to maintain synchronization. This is a major advantage, but the overhead of the consistency-based algorithms is very high as compared to the other algorithms.

The main problem with the software synchronization algorithms is that they rely on message exchanges for their synchronization. As a result, the worst case skews guaranteed by most of these algorithms are greater than the difference between the maximum and minimum message transit delay between any two nodes in the system [49], [70]. Since, in general, the difference between the maximum and the minimum message transit delay can be very large, the corresponding worst case skews are also very large, i.e., the synchronization is not very tight.

This problem is not present in hardware synchronization algorithms [41], [46], [91], [103]. These algorithms use special hardware at each node to achieve a very tight synchronization. The principle of hardware synchronization is that of a phase-locked loop. The hardware clock at each node is an output of a voltage-controlled oscillator. The voltage applied to the oscillator comes from a phase detector whose output is proportional to the phase error between the phase of its clock (i.e., the output of the voltage controlled oscillator it is controlling) and a reference signal generated by using the other clocks in the system. Thus by continuously adjusting the phase of each clock, all clocks are kept in lock-step. The key issue in hardware synchronization algorithms is the scheme used to select the reference signal. Care must be taken in selecting this reference signal because of the existence of faulty clocks. It can be shown that intuitive solutions like always selecting the median signal do not work in the presence of Byzantine faults.

One of the first hardware synchronizations that is resilient to Byzantine faults was proposed by Smith in [91]. The main problem with this scheme is that it can only synchronize a maximum of four clocks. This algorithm was later extended to any number of clocks by Krishna and Shin [46], Kessel [41], and also by Vasantavada and Marinos [103]. However, these algorithms had two major limitations. The first limitation is that they all assume a fully connected network of clocks, i.e., each clock sends its hardware clock signal to all other clocks. This is not a major problem in small systems. However, as systems get large, the total number of interconnections in a fully connected network will be so large that the reliability of synchronization will be determined by the failure rates of these interconnections rather than the failure rate of the clocks. Furthermore, there will be problems of fan-in and fan-out caused by the large number of interconnections. To alleviate this limitation Shin and Ramanathan proposed a clustering scheme which requires substantially fewer number of interconnections as compared to a fully connected network. The second limitation with the above mentioned hardware synchronization algorithms is that they assume negligible propagation delay in sending a clock signal from one node to the other. This can again be major problem in a large system where the physical separation between two clocks can be considerable enough to result in nonnegligible propagation delays. This limitation can be eliminated using the scheme proposed in [87]. When combined with the solutions in [86], [87], hardware synchronization algorithms can achieve a very tight synchronization even in large distributed system with very little overhead. However, the cost of additional hardware at each node precludes their use in large distributed systems unless a very tight synchronization is absolutely essential.

To overcome the cost limitation, a hybrid approach was proposed in [70]. This approach strikes a balance between the tightness of synchronization and the hardware requirement at each node. The basic idea is to augment the software algorithms with some hardware assistance. The hardware assist is used to estimate the message transit delays, which are then used to correct the estimate of skews made in the software algorithm. When the corrected skews are used in adjusting the local clock, reasonably tight skews can be achieved. In particular, the worst case skew achieved by the hybrid approach is much smaller than the difference between the maximum and minimum message transit delays between any two nodes.

Another approach for achieving a balance between tightness of synchronization and the cost of synchronization is referred to as *probabilistic synchronization* [18], [63]. Basically, this approach is to assume that the probability distribution function of message transit delay is known and let each node make several attempts to read the other clocks. At the end of each attempt, a node calculates the maximum error that might occur if the clock value obtained in that attempt is used to determine the correction. If the estimated maximum error is greater than a specified threshold, then the node makes another attempt to read the other node's clock. If one limits the maximum number of tries a node can make (to limit the message and the time overhead of synchronization), then there is a nonzero probability that a node cannot obtain another node's clock to a specified precision. This can lead to loss of synchronization. That is, unlike other schemes discussed above, in this approach, the worst case skews can be made as small as desired. However, depending on the desired worst case skew, there is a nonzero probability of loss of synchronization. Neither does it deal with fault tolerance as directly as the others.

VIII. Applications

There are numerous real-time applications including almost all of defense systems, automated factories, industrial process control, life-support systems, utility distribution and monitoring, and so on. These applications can often require the computer system to close feedback control loops, be intelligent, and provide real-time access to databases. In this section, we will briefly sketch some of the important application-related subjects: real-time control, real-time artificial intelligence, and real-time databases.

A. Real-Time Control Systems

Digital computers are commonly used in real-time control systems due mainly to their improved performance and reliability in dealing with increasingly complex controlled processes. A digital computer in the feedback loop of such a control system calculates the control input by executing a sequence of instructions, thereby introducing an unavoidable delay—called the *computation-time delay*—to the controlled process. This is an extra delay in addition to the system delay commonly seen in the control literature. The computation-time delay is an important part of the delay in the feedback loop, which also includes the other parts of delay related to measurement or sensing, A/D and D/A conversion, and actuation. However, these other parts of delay are usually constant, and thus easy to deal with.

Due to data-dependent loops and conditional branches, and unpredictable delays in sharing resources during the execution of control programs (that implement control algorithms), the computation-time delay is a *continuous* random variable which is usually much smaller than one sampling period T_s if no failure occurs in the controller computer. When a component failure or environmental disruption such as an electromagnetic interference (EMI) occurs, the time taken for error detection, fault location, and recovery must be added to the execution time of a control program, thus increasing the computation-time delay significantly. This in turn seriously degrades the system performance and may even lead to catastrophe, or a *dynamic failure* if the delay exceeds a certain limit called the *hard deadline* [85].

Several researchers attempted to analyze the effects of computation-time delay on the performance or stability of a control system. The sufficient (necessary) conditions of stability with a feedback delay and the delay effects on quadratic performance indices were presented for a linear control system [26] (for a nonlinear robot control system [81]). In [76], a more detailed analysis of the stability of a digital control system with a feedback delay was carried out by modifying the state difference equation. However, all these analyses are based on the assumption that the feedback delay is fixed or constant. Although the stability problem with a variable-feedback delay was investigated in [34], it was still based on a regular and periodic (i.e., thus deterministic) pattern of delay. In [8], a control system with a random time-varying delay in the feedback loop was modeled with a stochastic-delay differential equation, and sufficient conditions were derived for the almost-sure sample stability under which almost every possible differential equation in an ensemble of stochastic systems has a stable solution. However, this result did not give any explicit relation between the performance (or stability) and the magnitude of delay, but, instead, gave a condition of the coefficients of the state equations and the average rate-of-change of delay for sample stability. Furthermore, this work assumed a delay to be bounded by the "worst case" intersample period. In [85], the hard deadline in controlling the elevator deflection of the aircraft landing problem was obtained numerically by using the concept of allowed state space.

Shin and Kim [84] proposed a method for analyzing computation-time delay effects on system stability and state constraints for linear, time-invariant control systems. Specifically, they derived the hard deadline for such control systems—the critical value of computation-time delay beyond which a dynamic failure occurs—under the assumption that the computation-time delay is stochastically stationary. This assumption corresponds to the characteristics of transient computer failures caused by, for example, electromagnetic interferences. The system dynamics are modified first according to the assumed maximum delay NT_s and the probability distribution of delays whose occurrence periods $\leq NT_s$, where N is changed from 1 to the actual maximum delay (or hard deadline), denoted by DT_s. The pole positions of the modified state equation will then be tested to derive necessary conditions for (asymptotic) system stability. Moreover, the state and input constraints are used to derive the allowed state space from which the hard deadline is derived as a function of time and the system state. This analysis is useful for the one-shot delay model in [84], where a single event—a long-lasting failure—may cause a dynamic failure. The authors of [84] used this approach to derive deadlines for numerous example control systems and then applied the knowledge of hard deadline to the design of error recovery in a triple-modular-redundant (TMR) controller computer [83].

B. Real-Time Artificial Intelligence

As Artificial Intelligence (AI) techniques become mature, there has been growing interest in applying these techniques to controlling complex real-world systems which involve hard deadlines. In such systems, the controller is required to respond to certain inputs within rigid deadlines, or the system may fail catastrophically. Since the number of possible domain situations is too large to be fully enumerated, and the consequences of failure are so severe, testing alone is insufficient to guarantee the required real-time performance [48], [97]. These control problems require systems which can be *proven* to meet the hard deadlines imposed by the environment. Unfortunately, many AI techniques and heuristics are not suited to analyses that would provide guaranteed response times [21]. Even when AI techniques can be shown to have predictable response times, the variance in these response times is typically so large that providing timeliness guarantees based on the worst case performance would result in severe underutilization of the computational resources during normal operations [64].

Thus there is an apparent conflict between the nature of AI and the needs of real-world, real-time control systems. While AI methods are characterized by unpredictable or high-variance performance, real-time control systems require constant, predictable performance. Most research on "real-time AI" focuses either on restricted AI techniques that have predictable performance characteristics [9], [35], [43] or on reactive systems that retain little of the power of traditional AI [3], [11]. Several researchers are investigating systems which combine reactive and traditional AI methods [6], [32], [55], [88]. These approaches have concentrated on retaining both reactive and unpredictable mechanisms, but do not address the guarantees required by hard real-time tasks.

To combine unrestricted AI techniques with the ability to make hard performance guarantees, Musliner *et al.* [60] proposed a Cooperative Intelligent Real-time Control Architecture (CIRCA) (see Fig. 5). In this architecture, an AI subsystem reasons about task-level problems that require its powerful but unpredictable reasoning methods, while a separate real-time subsystem uses its predictable performance characteristics to deal with control-level problems that require guaranteed response times. The key difficulty with this approach is allowing the subsystems to interact without compromising their respective performance goals. CIRCA is based on a scheduling module and a structured interface that allow the unconstrained AI subsystem to asynchronously direct the real-time subsystem without violating any response-time guarantees.

1) Performance Tradeoffs and Bounded Reactivity: The responses of an intelligent control system can be rated along four dimensions: completeness, precision, confidence, and timeliness [50]. Completeness means that responses are produced for all possible inputs; timeliness means that the responses are produced before any associated deadlines. Precision and confidence together determine the "quality" of a solution, or how accurate the output is, to the best of the system's knowledge. An ideal intelligent control system could guarantee that any possible sequence of inputs would elicit optimal responses from the system, within all timing requirements.

Some systems strive for this ideal by assuming they have unlimited processing resources. For example, the subsumption architecture [11] assigns each reactive element to a separate processor. Such assumptions limit scalability: it would be highly impractical to build a subsumption system to control an oil drilling platform, which can make up to 20 000 signals available to its operators [7], [48].

Other systems recognize that processor limitations make realistic control systems subject to the same "bounded

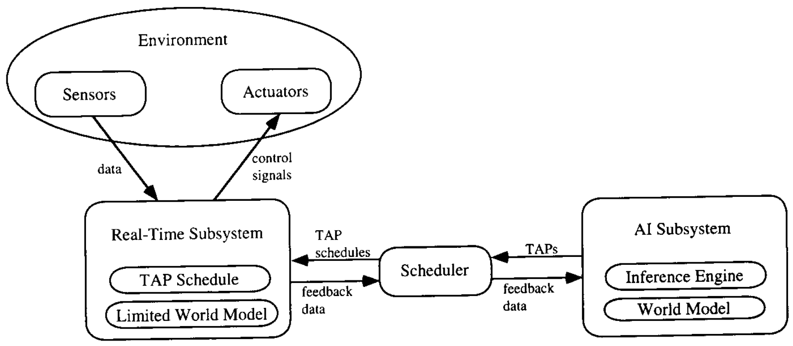


Fig. 5. The Cooperative Intelligent Real-Time Control Architecture.

rationality" [89] as humans, pushing ideal performance out of reach. To deal with bounded rationality, these systems provide differing levels of guarantees for the four performance dimensions. The guarantees that a system provides are often defined by the conditions that determine when its control algorithm returns a result. We call methods that halt when they reach a certain threshold along a dimension "any-<dimension>" algorithms [61]. For example, "any-time" algorithms can be terminated at any time, yielding some result, possibly with reduced precision, confidence, or completeness [9], [17], [35]. If "any-time" algorithms are interrupted before the deadline for every response, they guarantee timeliness and completeness. Many iterative numerical methods [12] are "any-precision" algorithms that terminate when a result with a certain precision has been achieved. Similarly, algorithms that halt when the confidence in a solution rises above a threshold are examples of "any-confidence" algorithms.

These types of systems are inappropriate for hard real-time control tasks because they cannot guarantee *acceptable* results within a deadline. Even "any-time" systems are inappropriate, because they have no control over the degree of response quality degradation which may occur. A hard real-time control system might only guarantee a subset of tasks, but that subset requires guaranteed timeliness, precision, and confidence to ensure that the system does not fail catastrophically. Realistic systems must also recognize that, in addition to processor limitations, sensor and actuator limitations constrain intelligent control systems. Even if a system's processors are fast enough, its sensors and actuators might not be able to provide ideal performance. Thus a system must recognize its "*bounded reactivity*" as well as its bounded rationality.

CIRCA [60] was designed to meet these demands by guaranteeing that it will produce a precise, high-confidence response in a timely fashion *to a limited set of inputs*. In other words, the architecture can sacrifice completeness in order to achieve precision, confidence, and timeliness. CIRCA reasons about its bounded reactivity within the AI subsystem (AIS) and the Scheduler, which cooperate to decide which responses the real-time subsystem (RTS) can and should guarantee. The AIS and the Scheduler form an "any-completeness" system, searching for a subset of guaranteed responses that will cover the inputs which are expected to occur in the domain at each moment. If the AIS can move the subset of guaranteed responses over the complete response set properly, CIRCA will provide guaranteed ideal performance.

2) Making Performance Guarantees: How can a real-time AI system provide any performance guarantees when its AIS uses high-variance or unpredictable computations? CIRCA answered this question based on the distinction we draw between task-level goals and control-level goals. This distinction is largely a functional one: in CIRCA, the RTS uses predictable methods to achieve control-level goals, while the AIS has unpredictable AI techniques to decompose task-level goals into control-level goals. CIRCA is designed to reason about guaranteeing its control-level goals, but not necessarily its task-level goals.

The choice of which specific goals are assigned to which category is largely up to the system designer: the "control-level" and the "task-level" are somewhat arbitrary divisions along a continuous range of problem complexities. However, since CIRCA's guarantees are based on worst case performance assumptions, assigning goals which need high-variance algorithms to the RTS results in a decreased capacity to guarantee other control-level goals. Thus the system designer must decide which types of goals will require guarantees, and which can be left less predictable. Given that separation, the AIS and Scheduler attempt to guarantee the control-level goals.

Figure 5 illustrates the CIRCA. The RTS is responsible for implementing the actual guaranteed responses; the AIS and the Scheduler cooperate to adjust the subset of responses that the RTS is supporting, attempting to ensure that the overall system meets hard deadlines and also achieves system goals as closely as possible.

The RTS executes a cyclic schedule of simple test-action pairs (TAP's) with known worst case execution times. Since the RTS performs no other functions, it can guarantee that the scheduled tests and actions are performed within predictable time bounds. The AIS reasons about the RTS's bounded reactivity, attempting to find a subset of TAP's which can be guaranteed to meet the control-level goals and make progress towards the task-level goals. In cooperation with the AIS, the Scheduler reasons about the limited execution resources available to the RTS, and builds the schedule of TAP's. Since the AIS and RTS run asynchronously, the AIS need not conform to the rigid performance restrictions which the RTS uses to guarantee meeting hard deadlines. Thus the AIS can utilize unpredictable, high-variance heuristics without compromising the overall system's ability to meet real-time deadlines.

C. Real-Time Databases

There are many real-time applications requiring database systems that support stringent timing constraints. Such database systems would facilitate the implementation of extensible and open software architectures for real-time systems and would help real-time applications manage large volumes of data and information sharing between tasks. These issues are fundamental to the goals of next generation real-time systems. Therefore, it is critical that databases capable of supporting real-time applications be developed.

The chief difficulty in applying database technology to real-time systems is that conventional database architectures are not designed to provide the performance levels or response time guarantees needed by real-time systems. Most conventional database systems are disk-based and use transaction logging and two-phase locking protocols to ensure transaction atomicity and serializability. These characteristics preserve data integrity, but they also result in relatively slow and unpredictable response times. Thus it is not feasible to simply connect a conventional multiuser database system, such as Oracle or Sybase, to a real-time manufacturing machine controller. The relatively slow and unpredictable database response times would cause time-critical tasks to miss deadlines [78], [90].

The inadequacy of conventional database systems for real-time applications has spawned the field of real-time database systems (RTDBS's) [25], [27], [53], [68], [75]. For more detailed information on these and on other work in the area of real-time databases refer to the article by Yu *et al.* in this issue.

IX. Conclusion

The growing use of digital computers in a great number of applications has driven real-time computing to become one of important disciplines of computer science and engineering. We have thus far highlighted the various issues in this new area of real-time computing. Existing solutions to many of the issues were also discussed. In this section, we identify important research issues which warrant further investigation and present a few possible directions for the future.

Experimental prototyping: Although basic research has produced many significant results, not much work has been done to validate those results on a real application. Existing real-time applications may need to be re-engineered with state-of-the-art results to validate the underlying assumptions and feed back ideas and problems to basic research. The prototypes will enable researchers to compare different solutions in a "real" environment and identify truly promising solutions.

Formal specification and verification: Specification and verification are other areas of research which need further investigation. At present, most of this work is still in preliminary stages. A few formal languages capable of dealing with timing constraints have been developed. These languages have been used to specify and verify small toy problems. There is, however, a serious need for better understanding of the fundamental problems in this area and their scalability. Such an understanding will lead to better specification languages. These languages must then be used to specify and verify real applications, not just a few toy problems.

Assignment of task and message deadlines: Although deadlines are crucial to most real-time systems, not much work has been done to formalize the assignment of deadlines to tasks and messages in a real-time application. Most task and message scheduling algorithms assume that the deadlines are given and they usually do not worry about how these deadlines were arrived at in the first place. However, in practice, task and message deadlines must be derived from the application under consideration. For example, an automotive company may set a goal to complete the assembly of a car in 20 h based on various reasons. Based on this deadline, one must derive individual station deadlines, operation deadlines, interstation transfer deadlines, etc. These subdeadlines may have to be further divided into task and message deadlines based on the capabilities of the stations in the system. Currently, derivation of such deadlines is done in an *ad hoc* fashion. It is essential to develop a systematic approach to the derivation of these deadlines in conjunction with specification of the application.

Characterization of task execution times: Similar to task and message deadlines, scheduling algorithms in real-time systems often assume knowledge of the task execution times. However, very little has been reported on how to determine the task execution times. There are several factors that make this problem very difficult: data-dependent loops, workload-dependent resource-sharing delays, conditional branches, and system dependency on hardware, operating systems, languages, and/or compilers. There is a need for tools which analyze the task description and determine the task execution times under different conditions.

Integration of fault-tolerance and/or security: Current real-time systems often treat fault-tolerance and security constraints in isolation. However, due to growing complexity of real-time systems, it is no longer possible to separate and treat timing, fault tolerance, and security as independent constraints. Therefore, there is a need to develop integrated approaches which systematically address the impact of fault-tolerance and security techniques on the timing constraints in a real-time system. Predictable error detection, location, and reconfiguration techniques need to be developed.

Real-time architectures: As discussed earlier, there are two levels of architecture that need to be considered for enhancing predictability: processor and system levels. At the single processor level, we need research into making interrupt handling and memory access predictable. However, contemporary processors are designed for high average performance using multiple pipelines and multilevel cache. At the system level, research is needed to make interprocessor communication predictable, help making multiprocessor scheduling feasible, and tolerate component failures using the component multiplicity as redundancy.

Real-time databases: This area is an important subdiscipline of real-time computing. One of the problems with RT databases is that databases by their very nature exhibit unpredictable response times. This arises from many sources like disk I/O, resource contention (blocking/aborts), and the inability to know exactly how many data objects a query will access. Important issues include transaction scheduling to meet deadlines, explicit semantics for specifying constraints (especially timing constraints), and checking the database system's ability of meeting transaction deadlines during application initialization.

Real-time communications: Gigabit networking technology is currently beginning to be embraced by industry in ATM switches. Real-time scheduling of packet/message transmissions for gigabit switches will allow us to dynamically select and correlate distributed sensor information on demand. Users of these systems can virtually visit different geographical locations, and get a "first-hand view" of the situation. Global assessment can be facilitated by gathering information from different points at a single decision point. Such capabilities can greatly enhance the functionality and flexibility of C3I systems, air-traffic control systems, modern mass-transportation systems, and automated factories. However, in order to fully utilize this potential, rapid strides need to made in the area of real-time communication.

Integration and scheduling of I/O: A real-time system contains not only disks but also a large number of sensors, actuators, and display devices. These devices must be integrated and scheduled with the processors to meet all timing constraints.

Real-time artificial intelligence: There is an inherent conflict between real-time computing and AI requirements. The former requires predictability and the latter requires "intelligence" by handling as much uncertainties as possible. Research is needed to integrate both RTC and AI by resolving this conflict.

In recent years, considerable research efforts have been directed towards real-time computing, especially because of the growing number of applications and the Office of Naval Research's five-year initiative on time-critical computing (which began in 1988). These efforts have resulted in significant progress in many of the subareas of real-time computing, yet the area is still young, exciting, and growing rapidly.

References

[1] G. Agrawal, B. Chen, and W. Zhao, "Optimal synchronous capacity allocation for hard real-time communication with the timed-token protocol," in *Proc. Real-Time Systems Symp.*, Dec. 1992, pp. 198–207.

[2] G. Agrawal, B. Chen, W. Zhao, and S. Davari, "Guaranteeing synchronous message deadlines with the timed token protocol," in *Proc. Distributed Computing Systems*, June 1992, pp. 468–475.

[3] P. E. Agre and D. Chapman, "Pengi: An implementation of a theory of activity," in *Proc. Nat. Conf. on Artificial Intelligence.* Morgan Kaufmann: 1987, pp. 268–272.

[4] D. P. Anderson, "The DASH project: An overview," Tech. Rep. UCB/CSD, Dept. of Elec. Eng. Comput. Sci., Univ. of California, Berkeley, Feb. 1988.

[5] ——, "A software architecture for network communication," in *Proc. Distributed Computing Systems*, June 1988, pp. 376–383.

[6] R. C. Arkin, "Integrating behavioral, perceptual, and world knowledge in reactive navigation," in *Robotics and Autonomous Systems 6*, 1990, pp. 105–122.

[7] J. E. Arnold, "Experiences with the subsumption architecture," in *Conf. on Artificial Intelligence Applications*, 1989, pp. 93–100.

[8] A. Belleisle, "Stability of systems with nonlinear feedback through randomly time-varying delays," *IEEE Trans. Automat. Contr.*, vol. AC-20, no. 1, pp. 67–75, Feb. 1975.

[9] M. Boddy and T. Dean, "Solving time-dependent planning problems," in *Proc. Int. Joint Conf. on Artificial Intelligence*, Aug. 1989, pp. 979–984.

[10] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Eng.*, vol. 7, no. 6, pp. 583–589, 1981.

[11] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE J. Robotics Automat.*, vol. RA-2, no. 1, pp. 14–22, Mar. 1986.

[12] R. L. Burden and J. D. Faires, *Numerical Analysis.* PWS-KENT Pub., 1989.

[13] H.-L. Chen and N.-F. Tzeng, "Fault-tolerant resource placement in hypercube computers," in *Proc. Int. Conf. on Parallel Processing*, Aug. 1991, pp. 517–524.

[14] M.-S. Chen and K. G. Shin, "Depth-first search approach for fault-tolerant routing in hypercube multicomputers," *IEEE Trans. Parallel Distributed Syst.*, vol. 1, no. 2, pp. 152–159, Apr. 1990.

[15] M.-S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. Comput.*, vol. 39, no. 1, pp. 10–18, Jan. 1990.

[16] S. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Proc. Real-Time Systems Symp.*, Dec. 1991, pp. 74–83.

[17] J.-Y. Chung, J. W. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1156–1174, Sept. 1990.

[18] F. Cristian, "Probabilistic clock synchronization," Tech. Rep. RJ 6432 (62550), IBM Almaden Research Center, Sept. 1988.

[19] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz, "The real-time operating system of MARS," *ACM Operating Syst. Rev.*, vol. 23, no. 3, pp. 141–157, July 1989.

[20] D. Dolev and J. Halpern, "On the possibility and impossibility of achieving clock synchronization," in *Proc. Symp. on Theory of Computing*, Apr. 1984, pp. 504–511.

[21] E. H. Durfee, "A cooperative approach to planning for real-time control," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, Nov. 1990, pp. 277–283.

[22] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, no. 5, pp. 662–675, May 1986.

[23] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE J. Selected Areas Commun.*, vol. 8, no. 3, pp. 368–379, Apr. 1990.

[24] H. Garcia-Molina, B. Kao, and D. Barbara, "Aggressive transmissions over redundant paths," in *Proc. Distributed Computing Systems*, 1991, pp. 198–207,

[25] GDX, sales literature of Firmware Associates, Inc., West Chester, PA, 1992.

[26] A. Gosiewski and A. Olbrot, "The effect of feedback delays on the performance of multivariable linear control systems," *IEEE*

Trans. Automat. Contr., vol. AC-25, no. 4, pp. 729–734, Aug. 1980.

[27] M. H. Graham, "Issues in real-time data management," *J. Real-Time Syst.*, vol. 4, no. 3, pp. 185–202, Sept. 1992.

[28] D. Haban and K. G. Shin, "Application of real-time monitoring for scheduling tasks with random execution times," *IEEE Trans. Software Eng.*, vol. 16, no. 12, pp. 1374–1389, Dec. 1990.

[29] J. Y. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-tolerant clock synchronization," in *Proc. Principles of Distributed Computing*, 1984, pp. 89–102.

[30] M. Hamdaoui and P. Ramanathan, "Selection of timed token protocol parameters to guarantee message deadlines," Tech. Rep. ECE-92-10, Univ.of Wisconsin–Madison, Nov. 1992.

[31] ——, "A dynamic multiple copy approach for message passing in virtual cut-through environment," in *Proc. Int. Parallel Processing Symp.*, Apr. 1993, pp. 757–761.

[32] S. Hanks and R. J. Firby, "Issues and architectures for planning and execution," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, Nov. 1990, pp. 59–70.

[33] J. R. Haritsa, M. J. Carey, and M. Livny, "Data access scheduling in firm real-time database systems," *J. Real-Time Syst.*, vol. 4, no. 3, pp. 203–241, Sept. 1992.

[34] K. Hirai and Y. Satoh, "Stability of a system with variable time delay," *IEEE Trans. Automat. Contr.*, vol. AC-25, no. 3, pp. 552–554, June 1980.

[35] E. J. Horvitz, "Reasoning about beliefs and actions under computational resource constraints," in *Proc. Workshop on Uncertainty in AI*, 1987.

[36] C.-J. Hou and K. G. Shin, "Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems," in *Proc. Real-Time Systems Symp.*, Dec. 1991, pp. 94–103.

[37] ——,"Incorporation of optimal timeouts into distributed real-time load sharing," in *Proc. Hawaii Int. Conf. on System Sciences*, Jan. 1993, pp. 603–612

[38] F. Jahanian and A. Goyal, "A formalism for monitoring real-time constraints at run-time," in *Proc. Fault-Tolerant Computing Symp. (FTCS-20)*, June 1990, pp. 148–155.

[39] D. D. Kandlur and K. G. Shin, "Traffic routing for multicomputer networks with virtual cut-through," *IEEE Trans. Comput.*, vol. 41, no. 10, pp. 1257–1270, Oct. 1992.

[40] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," in *Proc. Distributed Computing Systems*, May 1991, pp. 300–307.

[41] J. L. W. Kessels, "Two designs of a fault-tolerant clocking system," *IEEE Trans. Comput.*, vol. C-33, no. 10, pp. 912–919, Oct. 1984.

[42] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 398–405, Apr. 1988.

[43] R. E. Korf, "Real-time search for dynamic planning," in *Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[44] C. M. Krishna and K. G. Shin, "On scheduling tasks with a quick recovery from failure," *IEEE Trans. Comput.*, vol. C-35, no. 5, pp. 448–455, May 1986.

[45] C. M. Krishna, K. G. Shin, and R. W. Butler, "Synchronization and fault-masking in redundant real-time systems," in *Dig. Papers, FTCS-14*, June 1984, pp. 152–157.

[46] ——,"Ensuring fault tolerance of phase-locked clocks," *IEEE Trans. Comput.*, vol. C-34, no. 8, pp. 752–756, Aug. 1985.

[47] J. F. Kurose, M. Schwartz, and Y. Yemini, "Controlling window protocols for time-constrained communication in multiple access networks," *IEEE Trans. Commun.*, vol. 36, no. 1, pp. 41–49, Jan. 1988.

[48] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, "Real-time knowledge-based systems," *AI Mag.*, vol. 9, no. 1, pp. 27–45, 1988.

[49] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. Assoc. Comput. Mach.*, vol. 32, no. 1, pp. 52–78, Jan. 1985.

[50] V. R. Lesser, J. Pavlin, and E. Durfee, "Approximate processing in real-time problem solving," *AI Mag.*, vol. 9, no. 1, pp. 49–61, 1988.

[51] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[52] J. Liu, K.-J. Lin, W.-K. Shih, A. Yu, A.-Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *Computer*, vol. 24, no. 5, pp. 58–69, May 1991.

[53] V. B. Lortz and K. G. Shin, "MDARTS: A multiprocessor database architecture for real-time systems," Tech. Rep. CSE-TR-155-93, CSE Div., Dep. Elec. Eng. Comput. Sci., Univ. of Michigan, Ann Arbor, MI, Mar. 1993.

[54] J. Lundelius-Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Informat. and Computat.*, vol. 77, no. 1, pp. 1–36, 1988.

[55] D. P. Miller and E. Gat, "Exploiting known topologies to navigate with low-computation sensing," in *Proc. SPIE Sensor Fusion Conf.*, Nov. 1990.

[56] H. Mitra and P. Ramanathan, "A genetic approach for scheduling non-preemptive tasks with precedence and deadline constraints," in *Proc. Hawaii Int. Conf. on System Sciences*, Jan. 1993.

[57] A. K. Mok and M. L. Dertouzos, "Multiprocessor scheduling in a hard real-time environment," in *Proc. Texax Conf. on Computing Systems*, Nov. 1978.

[58] J. J. Molini, S. K. Maimon, and P. H. Watson, "Real-time system scenarios," in *Proc. Real-Time Systems Symp.*, Dec. 1990, pp. 214–225.

[59] M. L. Molle and L. Kleinrock, "Virtual time CSMA: Why two clocks are better than one," *IEEE Trans. Commun.*, vol. COM-33, no. 9, pp. 919–933, Sept. 1985.

[60] D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: A cooperative intelligent real-time control architecture," *IEEE Trans. Systems, Man, Cybern.*, vol. 23, no. 6, Nov./Dec. 1993 (in press).

[61] D. J. Musliner, E. H. Durfee, and K. G. Shin, "Any dimension algorithms and real-time AI," Tech. Rep. CSE-151-92, CSE Div., Dep. Elec. Eng. Comput. Sci., Univ. of Michigan, Ann Arbor, MI, Dec. 1992.

[62] A. Olson and K. G. Shin, "Message routing in HARTS with faulty components," in *Proc. Fault-Tolerant Computing Symp.*, June 1989, pp. 331–338.

[63] ——, "Probabilistic clock synchronization in large distributed systems," in *Proc. 11th Int. Conf. on Distributed Computer Systems*, May 1991, pp. 290–297.

[64] C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider, "Reducing problem-solving variance to improve predictability," *Commun. ACM*, vol. 34, no. 8, pp. 81–93, Aug. 1991.

[65] D. Peng and K. G. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," in *Proc. 9th Int. Conf. on Distributed Computer Systems*, June 1989, pp. 190–198.

[66] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," in *Proc. Int. Conf. on Distributed Computing Systems*, May 1990, pp. 108–115.

[67] K. Ramamritham and J. A. Stankovic, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. Comput.*, vol. 38, no. 8, pp. 1110–1123, Aug.1989.

[68] K. Ramamritham, "Real-time databases" (Invited Paper)*Int. J. Distributed and Parallel Databases*, to be published, 1992.

[69] P. Ramanathan and S. Chalasani, "Resource placement in k-ary n-cubes," in *Proc. Int. Parallel Processing Symp.*, pp. II-133–II-140, Aug. 1992.

[70] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 514–524, Apr. 1990.

[71] P. Ramanathan and G. M. Rupnick, "Deadline constrained message scheduling in point-to-point interconnection," in *Proc. Systems Design Synthesis Technology Workshop*(Silver Spring, MD, Naval Surface Warfare Center, Sept. 1991), pp. 183–192.

[72] P. Ramanathan and K. G. Shin, "A multiple copy approach for delivering messages under deadline constraints," in *Proc. Fault-Tolerant Computing Symp.*, June 1991, pp. 300–307.

[73] ——, "Delivery of time-critical messages using a multiple copy approach," *ACM Trans. Comput. Syst.*, vol. 10, no. 2, pp. 144–166, May 1992.

[74] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-tolerant clock synchronization in distributed systems," *IEEE Comput.*, vol. 23, no. 10, pp. 33–42, Oct. 1990.

[75] *RTA Introduction & Overview*, Real Time Computer software GmbH, 1992.

[76] Z. V. Rekasius, "Stability of digital control with computer

interruption," *IEEE Trans. Automat. Contr.*, vol. AC-31, no. 4, pp. 356–359, Apr. 1986.

[77] F. E. Ross, "FDDI—A tutorial," *IEEE Commun. Mag.*, vol. 24, pp. 46–61, May 1986.

[78] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Concurrency control for distributed real-time databases," *SIGMOD Rec.*, vol. 17, no. 1, pp. 82–98, Mar. 1988.

[79] K. G. Shin, "HARTS: A distributed real-time architecture," *IEEE Comput.*, vol. 24, no. 5, pp. 25–35, May 1991.

[80] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state-change broadcasts," *IEEE Trans. Comput.*, vol. 38, no. 8, pp. 1124–1142, Aug. 1989.

[81] K. G. Shin and X. Cui, "Effects of computing time delay on real-time control systems," in *Proc. 1988 American Control Conf.*, 1988, pp. 1071–1076.

[82] K. G. Shin and G. L. Dykema, "Distributed {I/O} architecture for {HARTS}," in *Proc. 17th Int. Symp. on Computer Architectures*, June 1990, pp. 332–342.

[83] K. G. Shin and H. Kim, "A time redundancy approach to tmr failures using fault-state likelihoods," *IEEE Trans. Comput.* (in press).

[84] ——, "Derivation and application of hard deadlines for real-time control systems," *IEEE Trans. System, Man, Cybern.*, vol. 22, no. 6, pp. 1403–1413, Nov./Dec. 1992.

[85] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaliating real-time computer controller and its application," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 4, pp. 357–366, Apr. 1985.

[86] K. G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious faults," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 2–12, Jan. 1987.

[87] ——, "Transmission delays in hardware clock synchronization," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1465–1467, Nov. 1988.

[88] R. Simmons, "An architecture for coordinating planning, sensing, and action," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, Nov. 1990, pp. 292–297.

[89] H. A. Simon, *Models of Bounded Rationality.* Boston, MA: M.I.T. Press, 1982.

[90] M. Singhal, "Issues and approaches to design of real-time database systems," *SIGMOD Rec.*, vol. 17, no. 1, pp. 19–33, Mar. 1988.

[91] T. B. Smith, "Fault-tolerant clocking system," in *Proc. Fault-Tolerant Computing Symp.*, 1981, pp. 262–264.

[92] T. B. Smith and J. H. Lala, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer: FTMP principles of operation," Contractor Rep.166071, NASA, Langley Res. Ctr., May 1983.

[93] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *J. Assoc. Comput. Mach.*, vol. 34, no. 3, pp. 626–645, July 1987.

[94] J. Stankovic, "SpringNet: A scalable architecture for high performance, predictable, distributed, real-time computing," Tech. Rep.91-74, Univ. of Massachusetts, Oct. 1991.

[95] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next generation systems," *IEEE Comput.*, vol. 21, no. 10, pp. 10–19, Oct. 1988.

[96] J. A. Stankovic and K. Ramamritham, "The design of Spring kernel," in *Proc. Real-Time Systems Symp.*, Dec. 1987, pp. 146–157.

[97] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *IEEE Comput.*, vol. 21, no. 10, pp. 10–19, Oct. 1988.

[98] J. K. Strosnider, T. Marchok, and J. P. Lehoczky, "Advanced real-time scheduling using the IEEE 802.5 token ring," in *Proc. Real-Time Systems Symp.*, Dec. 1988, pp. 42–52.

[99] A. L. Tannenbaum, *Computer Networks* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[100] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: An NP-hard problem made easy," *Real-Time Syst.*, vol. 4, no. 2, pp. 145–165, June 1992.

[101] H. Tokuda, M. Koreta, and C. Mercer, "A real-time monitor for a distributed real-time o.s.," *ACM SIGPlan Notices*, vol. 24, no. 1, pp. 68–77, Jan. 1989.

[102] J. P. Tsai, K.-Y. Fang, and H.-Y. Chen, "A noninvasive architecture to monitor real-time distributed systems," *Comput.*, vol. 23, no. 3, pp. 11–23, Mar. 1990.

[103] N. Vasanthavada and P. N. Marinos, "Synchronization of fault-tolerant clocks in the presence of malicious failures," *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 440–448, Apr. 1988.

[104] J. P. C. Verhoosel, E. J. Luit, and D. K. Hammer, "A static scheduling algorithm for distributed hard real-time systems," *Real-Time Syst.*, vol. 3, no. 3, pp. 227–246, Sept. 1991.

[105] W. Zhao and K. Ramamritham, "Virtual time {CSMA} protocols for hard real-time communications," *IEEE Trans. Software Eng.*, vol. SE-13, no. 8, pp. 938–952, Aug. 1987.

[106] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Trans. Comput.*, vol. C-36, no. 8, pp. 949–960, Aug. 1987.

[107] ——, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.*, vol. SE-13, no. 5, pp. 564–577, May 1987.

[108] W. Zhao, J. A. Stankovic, and K. Ramamritham, "A window protocol for transmission of time constrained messages," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1186–1203, Sept. 1990.

[109] Q. Zheng and K. G. Shin, "Establishment of isolated failure immune real-time channels in harts," *IEEE Trans. Parallel Distrib. Syst.* (in press).

[110] ——, "Fault-tolerant real-time communication in distributed computing systems," in *Proc. FTCS-22*, July 1992, pp. 86–93.

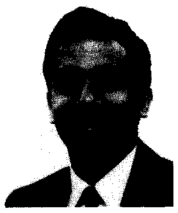
[111] ——, "On the ability of establishing real-time channels in point-to-point packet-switched networks," *IEEE Trans. Commun.* (in press), 1993.

[112] ——, "Synchronous bandwidth allocation in {FDDI} networks," in *Proc. ACM Multimedia'93* 1993, pp. 31–38.

Kang G. Shin (Fellow, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1970, and both M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaka, NY, in 1976 and 1978, respectively.

From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, NY. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California at Berkeley, and the International Computer Science Institute, Berkeley, CA. He is Professor and Associate Chair of Electrical Engineering and Computer Science for the Computer Science and Engineering Division, The University of Michigan, Ann Arbor. In 1985, he funded the Real Time Computing Laboratory, where he and his colleagues are currently building a 19–node hexagonal mesh multicomputer, called HARTS, to validate various architectures and analytic results in the area of distributed real-time computing. He has also been applying the basic research results of real-time computing to manufacturing-related applications, ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes. Recently, he has initiated research on the open-architecture Information Base for machine tool controllers. He has authored/coauthored over 200 technical papers (about 110 of these in archival journals) and several book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing.. In 1987, he received the Outstanding IEEE TRANSACTIONS ON AUTOMATIC CONTROL Paper Award for a paper on robot trajectory planning. In 1989, he received the Research Excellence Award from the University of Michigan.

Dr. Shin was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, the Guest Editor of the August 1987 Special Innsue on Real-Time Systems for the IEEE TRANSACTIONS ON COMPUTERS, a program Co-Chair for the 1992 International Conference on Parallel Processing, and served on numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991–1993, is a Distinguished Visitor of the Computer Society of the IEEE, an Associate Editor of IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS, and an Area Editor of *International Journal of Time-Critical Computing Systems.*



Parameswaran Ramanathan (Member, IEEE) received the B.Tech. degree from the Indian Institute of Technology, Bombay, India, in 1984, and the M.S.E. and Ph.D. degrees from the University of Michigan, Ann Arbor, in 1986 and 1989, respectively.

From 1984 to 1989 he was a Research Associate in the Department of Electrical Engineering and Computer Science at the university of Michigan. At present, he is an Assistant Professor in the Department of Electrical and Computer Engineering and in the Department of Computer Sciences at the University of Wisconsin–Madison. His research interests include the areas of real-time systems, fault-tolerant computing, distributed systems, and parallel algorithms.