Real Time Systems: A Tutorial [*]

Fabio Panzieri and Renzo Davoli

Dipartimento di Matematica
Università di Bologna
Piazza di Porta S. Donato 5
40127 Bologna (Italy)

Abstract. In this tutorial paper, we introduce a number of issues that arise in the design of distributed real-time systems in general, and hard real-time systems in particular. These issues include time management, process scheduling, and interprocess communications within both local and wide area networks. In addition, we discuss an evaluation, based on a simulation model, of a variety of scheduling policies used in real-time systems. Finally, we examine some relevant examples of existing distributed real-time systems, describe their structuring and implementation, and compare their principal features.

1 Introduction

The principal responsibility of a real-time (RT) system can be summarized as that of producing correct results while meeting predefined deadlines in doing so. Hence, the computational correctness of the system depends on both the logical correctness of the results it produces, *and* the timing correctness, i.e. the ability to meet deadlines, of its computations.

Hard real-time (HRT) systems can be thought of as a particular subclass of RT systems in which lack of adherence to the above mentioned deadlines may result in a catastrophic system failure. In the following we shall use the phrase "soft real-time (SRT) systems" to indicate to those RT systems in which the ability to meet deadlines is indeed required; however, failure to do so does not cause a system failure.

The design complexity of HRT and SRT systems can be dominated by such issues as the application timing and resource requirements, and the system resource availability. In particular, in the design of a HRT system that support critical applications (e.g. flight control systems, nuclear power station control systems, railway control systems), that complexity can be exacerbated by such possibly conflicting application requirements as the demand for highly reliable and highly available services, under specified system load and failure hypotheses, and the need to provide those services while satisfying stringent timing constraints.

[*] Partial support for this work was provided by the Italian National Research Council (CNR) under contract N. 92.00069.CT12.115.25585.

In particular, as a HRT system has to provide services that be both timely and highly available, the design of any such system requires that appropriate fault tolerance techniques, capable of meeting hard real-time requirements, be deployed within that system.

Current technology allows the HRT system designer to implement cost-effective fault tolerance techniques, based on the use of redundant system components. However, the development of redundancy management policies, that meet real-time requirements, can introduce further complexity in the system design (and validation) process. Thus, in essence, the design of a HRT system requires that a number of performance/reliability trade-off issues be carefully evaluated.

Both HRT and SRT systems may well be constructed out of geographically dispersed resources interconnected by some communication network, so as to form a distributed RT system. (Conforming to the definition proposed in [8, 29, 36], distributed HRT systems can be classified as *responsive* systems, i.e. distributed, fault tolerant, real-time systems.)

In this tutorial paper, we shall focus on issues of design and implementation of distributed RT systems, and describe five operational examples of those systems, namely [52, 17, 33, 56, 47]. In particular, we shall discuss the key paradigms for the design of timely and available RT system services, and examine techniques for process scheduling, time management, and interprocess communications over local and wide area networks.

This paper is structured as follows. In the next Section, we discuss the principal issues arising in the design of RT systems. In Section 3, we examine a number of scheduling policies that are usually deployed in those systems. In addition, in that Section we introduce an evaluation of those policies, based on a simulation study, that allows one to asses the adequacy of those policies with respect to different parameters that can characterize the system load and its communication costs. Section 4 introduces the distributed RTOSs mentioned above. Finally, Section 5 proposes some concluding remarks.

2 Design Issues

A generic (i.e. hard or soft) real-time system can be described as consisting of three principal subsystems [23], as depicted in Figure 1 below.

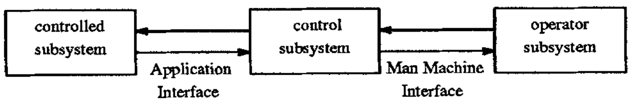

Fig. 1. Example of Real-Time System Organization

In Figure 1, the *controlled subsystem* represents the application, or environment (e.g. an industrial plant, a computer controlled vehicle), which dictates the real-time requirements; the *control subsystem* controls some computing and communication equipment for use from the controlled subsystem; the *operator subsystem* initiates and monitors the entire system activity. The interface between the controlled and the control subsystems consists of such devices as sensors and actuators. The interface between the control subsystem and the operator consists of a man-machine interface.

The controlled subsystem is implemented by tasks (termed *application tasks*, in the following) that execute using the equipment governed by the control subsystem. This latter subsystem can be constructed out of a possibly very large number of processors, equipped with such local resources as memory and mass storage devices, and interconnected by a real-time local area network (i.e. a local network that provides bounded maximum delay of a message exchange - see Subsection 2.4). Those processors and resources are governed by a software system that we term the Real-time Operating System (RTOS).

The deployment of RTOSs in safety critical environments (e.g. guidance and navigation systems) imposes severe reliability requirements on the design and implementation of those RTOSs [10]. As discussed in [26], these requirements can be defined in terms of maximum acceptable probability of system failure. Thus, for example, flight control systems, such as that used in the Airbus A-320, require 10^{-10} probability of failure per flight hour. Vehicle control systems in which the cost of a failure can be quantified in terms of an economic penalty, rather than loss of human lifes (e.g. systems for satellite guidance, unmanned underwater navigation systems), require 10^{-6} to 10^{-7} probabilities of failure per hour.

Fault tolerance techniques, based on the management of redundant hardware and software system components, are commonly used in order to meet these reliability requirements. However, it is worth pointing out that the implementation of these techniques, that indeed determine the system reliability, require that some of the system performance be traded for reliability. Methodological approaches that allow one to assess these trade-off issues are discussed in [1, 38, 57, 39].

The principal issues concerning the design of a RTOS are introduced below, in isolation. In particular, in the following we shall discuss (i) relevant characteristics of the RT applications that may use a RTOS, (ii) two general paradigms that can be applied to the design of a RTOS, (iii) time management, and (iv) interprocess communication issues in distributed RT systems.

2.1 RT Applications

A RT application can be modelled as a set of cooperating tasks. These tasks can be classified, according to their timing requirements, as *hard real time* (HRT), *soft real time* (SRT), and *not real time* (NRT) tasks. A **HRT task** is a task whose timely (and logically correct) execution is deemed as critical for the operation of the entire system. The deadline associated to a HRT task is conventionally termed *hard deadline*, owing to the critical nature of that task. As a consequence, it is assumed that missing a hard deadline can result in a catastrophic system failure. A **SRT task**, instead, is characterized by an execution deadline whose adherence is indeed desirable, although not critical, for the functioning of the system (hence, the SRT task deadline is usually termed *soft deadline*). **NRT tasks** are those tasks which exhibit no real-time requirements (e.g. system maintenance tasks that can run occasionally in the background).

Application tasks can be further classified as *periodic*, *aperiodic* (or *asynchronous* [60]), and *sporadic* tasks. **Periodic tasks** are those tasks that enter their execution state at regular intervals of time, i.e. every T time units. These tasks, generally used in such applications as signal processing and control, are typically characterized by hard deadlines [34]. **Aperiodic tasks** are those tasks whose execution time cannot be anticipated, as their execution is determined by the occurrence of some internal or external event (e.g. a task responding to a request from the operator). These tasks are usually characterized by soft deadlines. Finally, aperiodic tasks characterized by hard deadlines are termed **sporadic tasks** [30] (e.g. tasks dealing with the occurrence of system failures, or with emergency requests from the operator).

In view of the above classifications, one can observe that the principal responsibility of a RTOS is to guarantee that each individual execution of each application task meet the timing requirements of that task. However, it is worth noting that, in order to fulfil that responsibility, the objective of a RTOS cannot be stated just as that of minimizing the average response time of each application task; rather, as pointed out in [58, 60], the fundamental concern of a RTOS is that of being *predictable*, i.e. the functional and timing behaviour of a RTOS should be as deterministic as necessary to meet that RTOS specification. Thus, fast hardware and efficient algorithms are indeed useful, in order to construct a RTOS that meet real-time requirements; however, they are not sufficient to guarantee the predictable behaviour required from that system. .

2.2 RTOS Design Paradigms

Two general paradigms for the design of predictable RTOSs can be found in the literature. These paradigms have led to the development of two notably different RTOS architectures, termed Event-Triggered (ET) and Time-Triggered (TT) architectures [24], respectively. In essence, in ET RTOSs (e.g. [55]), any system activity is initiated in response to the occurrence of a particular event, caused by the system environment. Instead, in TT RTOSs (e.g. [21]), system activities are initiated as predefined instants of the *globally synchronized* time (see next Subsection) recur.

In both architectures, the RTOS predictability is achieved by using (different) strategies to assess, prior to the execution of each application task, the resource needs of that task, and the resource availability to satisfy those needs. However, in ET architectures, these resource needs and availability may vary at run-time, and are to be assessed dynamically. Thus, resource need assessment in ET architectures is usually based on parametric models [40]. Instead, in TT architectures these needs can be computed off-line, based on a pre-run time analysis of the specific application that requires the use of the TT architecture; if these needs cannot be anticipated, worst-case estimates are used.

TT architecture advocates criticize the ET architectural approach as the ET architectures, owing to their very nature, can be characterized by an excessive number of possible behaviors that must be carefully analyzed in order to establish their predictability [24]. In contrast, ET architecture advocates claim that these architectures are more flexible than TT architectures, and ideal for a large class of applications that do not allow to predetermine their resource requirements. In particular, they argue that TT architectures, owing to the worst case estimate approach mentioned above, are prone to waste resources in order to provide predictable behavior.

In both ET and TT architectures the resource need and availability assessment is to be carried out while taking into account the timing requirements of the applications. Hence, issues of time management, that characterize the system's temporal behaviour, are of crucial importance in the design of any RT system.

2.3 Time Management

One of the principal concerns, in the field of time management in RT systems, consists of providing adequate mechanisms for measuring (i) the time instants at which particular events must occur, and (ii) the duration of the time intervals between events. In a distributed RT system, these concerns become particularly critical, as the occurrence of the same event can be observed from such inherently asynchronous devices as a number of different processors.

However, this problem can be adequately dealt with by providing the RT applications with a common time reference of specified accuracy. This time reference can be constructed by synchronizing the values of the local real-time clocks, incorporated in each processor of the system, so as to obtain a global notion of time within that system.

A large variety of clock synchronization algorithms can be found in the literature, e.g. [28, 42, 27, 5, 50], based on the exchange of clock synchronization messages among the system nodes. We shall not describe these algorithms here, as they are discussed in detail in the already cited references. However, we wish to mention that, as pointed out in [41], any such algorithm has to meet the following four requirements:

1. the clock synchronization algorithm is to be capable of bounding, by a known constant, the maximum difference of the time values between the observation of the same event from any two different nodes of the system (measured according to the value of the local clock of each of these two nodes);
2. the notion of global time constructed by the synchronization algorithm is to be sufficiently accurate to allow one to measure small time intervals at any point in time;
3. the clock synchronization algorithm is to be capable of tolerating the possible fault of a local RT clock, or the loss of a clock synchronization message;
4. the overall system performance is not to be degraded by the execution of the clock synchronization algorithm.

In order to meet these requirements, either centralized or decentralized clock synchronization algorithms can be deployed. A centralized approach can be implemented by means of a central synchronization unit, e.g. a "time server" node responsible for periodically distributing time synchronization messages to the other nodes in the system; some such an approach can typically be very vulnerable to failures of the synchronization unit itself. Instead, a decentralized approach, owing to the redundancy inherent in the distributed infrastructure that can be used for its implementation, can offer better guarantees as to fault tolerance (provided that implementation be based on a realistic fault model).

As already mentioned, the clock synchronization algorithms in distributed RT systems can be implemented by message exchanges. However, it is worth pointing out that these implementations may introduce overheads that can affect the overall system performance, thus violating the requirement 4 above. In order to overcome this problem, a practical and effective solution has been proposed in [41] (and developed within the context of the MARS project [21]). This solution is based on the implementation of an accurate clock synchronization algorithm in a special-purpose VLSI chip; this chip can be incorporated in a subset of nodes of the system, and used by those nodes to exchange clock synchronization messages. The rest of the system nodes can maintain their clocks synchronized by monitoring the synchronization message traffic. This implementation notably reduces (to less than 1%, it is claimed in [41]) the CPU load and the network traffic caused by the clock synchronization algorithm.

2.4 Interprocess Communications

In view of the predictability requirement mentioned earlier, distributed RT systems require primarily that the communication support they use provide them with deterministic behaviour of the communication infrastructure. This behaviour can be achieved by constructing a communication protocol architecture characterized by such deterministic properties as *bounded channel access delay*, and *bounded message delay*.

The channel access delay is defined as the interval of time between the instant in which a task issues a request for sending a message, and the instant in which the communication interface, local to the node where that task is running, actually transmits that message on the communication channel. The message delay, instead, is defined as the interval of time between the instant in which a task requests the transmission of a message, and the instant in which that message is successfully delivered to its destination; hence, the message delay includes the channel access delay. If a message is delivered with a message delay that exceeds a target (e.g. application dependent) value, that message is considered lost.

It as been pointed out in [14] that, in such distributed RT applications as those based on non-interactive audio and video communications, an additional property that RT protocols are required to possess consists of the provision of bounded message delay *jitter*; this jitter is the absolute value of the difference between the actual message delay of a transmitted message, and the target message delay. Issues of delay jitter control in packet switching networks are discussed in [13]; protocols characterized by the bounded delay jitter property, for use for communications over those networks, are described in [14, 12, 11].

Further general properties that can be required from a RT protocol include stability, and fault tolerance. The former property refers to the ability of the protocol to continue to operate effectively in the presence of network traffic variations and temporary network overloading. The latter property refers to the protocol ability to survive communication channel failures (e.g. omission failures [6], such as those that can be caused by a noisy channel).

A survey of basic techniques for the design of protocols for distributed RT systems is discussed in [25]. In this paper, the authors examine time constrained protocols that can be deployed in distributed RT systems based on broadcast (both local and wide area) networks. In particular, they classify these protocols in *controlled access* and *contention based* protocols. The former class includes Time Division Multiple Access Protocols; the latter, instead, includes token based schemes. In addition, this paper points out a number of performance/reliability trade-off issues that arise in the design of these protocols. These issues include the relations among the message loss percentage, the message transmission rate, and the timing constraints associated to the messages.

Further work on RT communications, emphasizing HRT communication issues, can be found in [49, 61]. In [49], the author proposes a protocol for HRT communication in local area networks that provides bounded channel access delay. In [61], the authors evaluate the performance of four protocols for HRT communications, termed Virtual Time CSMA protocols. The performance metrics they use for this evaluation are based on the percentage of messages that miss their deadlines, and the effective channel utilization.

Finally, an interesting protocol for communications in distributed HRT systems has been recently proposed in [24]. This protocol, designed for the support of distributed TT architectures, provides principally (i) predictable message delay, (ii) group communications and membership service [7], (iii) redundancy management, and (iv) accurate clock synchronization. A further attractive (and unconventional) property of this protocol is that it is designed so as to be highly scalable, i.e. capable of operating efficiently on different communication media (e.g. twisted pairs as well as optical fibers).

3 Scheduling

In a RT system, the responsibility of the scheduling algorithm is to determine an order of execution of the RT tasks that be *feasible*, i.e. that meet the resource and timing requirements of those tasks. In the design of a RT system, the choice of an appropriate scheduling algorithm (or policy) may depend on several issues, e.g. the number of processors available in the system, their homogeneity or heterogeneity, the precedence relations among the application tasks, the task synchronization methods. In addition, application dependent characteristics of the RT tasks may contribute to determine the choice of the scheduling algorithm. For example, RT application tasks can be *preemptable*, or *non-preemptable*. A preemptable task is one whose execution can be suspended by other tasks, and resumed later; a non-preemptable task must run until it completes, without interruption. Thus, both preemptive and non-preemptive algorithms have been proposed. (However, for the purposes of this tutorial paper, non-preemptive scheduling will not be discussed as a large number of non-preemptive scheduling problems has been shown to be NP-hard [4].)

RT scheduling algorithms can be classified as either *static* or *dynamic* algorithms. A static scheduling algorithm is one in which a feasible schedule is computed off-line; one such algorithm typically requires a priori knowledge of the tasks' characteristics. In contrast, a dynamic scheduling algorithm determines a feasible schedule at run time. Thus, static scheduling is characterized by low run-time costs; however, it is rather inflexible, and requires complete predictability of the RT environment in which it is deployed. Instead, dynamic scheduling entails higher run-time costs; however, it can adapt to changes in the environment.

The literature on task scheduling algorithms is very vast (e.g. see [16, 4, 60]); a complete taxonomy of these algorithms and their properties is beyond the scope of this paper. Rather, we shall confine our discussion below to summarizing the most common scheduling algorithms that are used in the implementation of RT systems, and introduce the results obtained from a recent simulation study of these algorithms, that we have carried out.

3.1 Scheduling Algorithms

The scheduling of periodic tasks on a single processor is one of the most classical scheduling problems in RT systems [34]. Two alternative approaches have been proposed to solve this problem, based on the assignment of either a fixed or, alternatively, a dynamic priority value to each task. In the fixed priority approach, the task priority value is computed once, assigned to each task, and maintained unaltered during the entire task life time. In the dynamic priority approach (also termed deadline driven), a priority value is dynamically computed and assigned to each task, and can be changed at run-time. These approaches have led to the development of a variety of preemptive scheduling policies (preemption, in priority driven scheduling policies, means that the processing of a task can be interrupted by a request for execution originated from a higher priority task). These include the ***Rate Monotonic*** (RM), the ***Earliest Deadline First*** (EDF), and the ***Least Slack Time First*** (LSTF) policies, introduced below.

The RM policy assigns a fixed priority value to each task, according to the following principle: the shorter the task period, the higher the task priority. It has been shown in [34] that this policy is *optimal* among fixed priority policies (i.e. given a set of tasks, it always produces a feasible schedule of that set of tasks, if any other algorithm can do so).

The EDF and LSTF policies implement dynamic priorities. With the EDF policy, the earlier the deadline of a task, the higher the priority assigned to that task. Instead, with the LSTF policy, the smaller the ***slack time*** (see below) of a task, the higher the priority value assigned to that task. The task slack time is defined as the difference between the amount of time from the current time value to the deadline of a task, and the amount of time that task requires to perform its computation.

In order to deal with the scheduling of aperiodic tasks, the following five different policies have been proposed [30]. The first policy consists of scheduling the aperiodic tasks as background tasks, i.e. aperiodic tasks are allowed to make their computations only when no periodic tasks are active. The second policy, termed ***Polling***, consists of creating a periodic process, characterized by a fixed priority, that serves the aperiodic task requests (if any). The main problem with this policy is the incompatibility between the cyclic nature of this policy, and the bursty nature of the aperiodic tasks.

The third and fourth policies are the ***Priority Exchange*** (PE) and the ***Deferrable Server*** (DS) policies. Both these policies aim to maximizing the responsiveness of aperiodic tasks by using a high priority periodic server that handles the aperiodic task requests. In both the PE and the DS policies, the server preserves the execution time allocated to it, if no aperiodic task requests are pending. (In fact, these policies are also termed ***bandwidth preserving***, as they provide a mechanism for preserving the resource bandwidth allocated for aperiodic services if, when this bandwidth becomes available, it is not needed.)

The difference between these two policies is in the way they manage the high priority of their periodic servers. In the DS policy, the server maintains its priority for the duration of its entire period; thus, aperiodic task requests can be serviced at the server's high priority, provided that the server's execution time for the current period has not been exhausted. In contrast, in the PE policy, the server exchanges its priority with that of the pending, highest priority, periodic task, if no aperiodic task requests occur at the beginning of the server period.

The DS and PE policies have been developed in order to deal with sporadic tasks (i.e. aperiodic HRT tasks, as defined in Subsection 2.1 of this tutorial paper). The fifth policy that we consider, i.e. the ***Sporadic Server*** (SS) policy, has been designed to deal with the scheduling of aperiodic (SRT) tasks. This policy, yet again based on the creation of a periodic server of aperiodic requests, is characterized by a response time performance comparable to that of the DS and PE policies, and a lower implementation complexity than these two policies. The SS policy is discussed in detail in [30].

Task scheduling in tightly coupled distributed systems, such as a shared memory multiprocessor, can be governed by a single scheduler responsible for allocating the processing elements to the application tasks. McNaughton, in [37], has proposed an optimal, preemptive scheduling algorithm for independent tasks. This algorithm has been extended to deal with such different issues as tasks having DAG precedence graphs, and periodic executions (see [16] for a complete survey).

In loosely coupled distributed RT systems, owing to the high cost of process migration between processors, and to the loss of predictability that operation may entail, tasks can be statically assigned to the system processors. In these systems, the scheduler is usually structured in two separate components; namely, an *allocator*, and a *local scheduler*. The allocator is responsible for assigning tasks to the distributed system processors; the local scheduler (one for each processor) implements a single processor scheduling policy, such as those introduced earlier, to dispatch the (local) execution requests. It is worth mentioning that the allocation algorithms are usually based on some heuristic approach, as the problem of allocating tasks to processors can be very complex. (For example, it has been shown [3] that finding an optimal assignment of tasks, characterized by an arbitrary communication graph, to four or more processors with different speeds is an NP-hard problem.)

The I/O subsystem of a real-time system may require its own scheduler. The simplest way to access an I/O resource is by using a non-preemptive FIFO policy. However, the preemptive scheduling techniques introduced above for processor scheduling (i.e. RM, EDF, LSTF) can be implemented to schedule I/O requests.

Relevant figures of merit that can be used to assess the effectiveness of a scheduling policy include the *Breakdown Utilization* (BU), the *Normalized Mean Response Time* (NMRT), and the *Guaranteed Ratio* (GR), introduced below.

The BU, as defined in [30], is the degree of resource utilization at or below which the RTOS can guarantee that all the task deadlines will be met. This figure provides a metric for the assessment of the effectiveness of a scheduling policy, as the larger the breakdown utilization, the larger the cpu time devoted to task execution.

The NMRT is the ratio between the time interval in which a task becomes ready for execution and terminates, and the actual cpu time consumed for the execution of that task. Yet again, this figure provides a metric of the effectiveness of the selected scheduling policy as, the larger the NMRT, the larger the task idle time.

Finally, for dynamic algorithms, a relevant performance metric is the GR, i.e. the number of tasks whose execution can be guaranteed versus the total number of tasks that request execution.

3.2 Simulation Study

In order to evaluate the effectiveness of the algorithms introduced above, we have developed a distributed RT system simulation model that incorporates the majority of those algorithms, suitable for the scheduling of periodic, aperiodic, and sporadic tasks [43].

In particular, our model implements the RM, the EDF, and the LSTF algorithms, for the scheduling of periodic tasks.

Aperiodic task scheduling can be supported, in our model, by means of the background (BG), the Polling (PL), the DS, and the SS algorithms. The BG scheduling algorithm is implemented by executing aperiodic tasks in those time intervals in which no periodic tasks are active. The PL, DS, and SS algorithms are implemented by periodic servers that schedule aperiodic tasks at regular intervals of time, provided that no periodic task be in execution.

The scheduling of the sporadic tasks is simulated by implementing a periodic server, fully dedicated to the scheduling of those tasks, that is enabled sufficiently

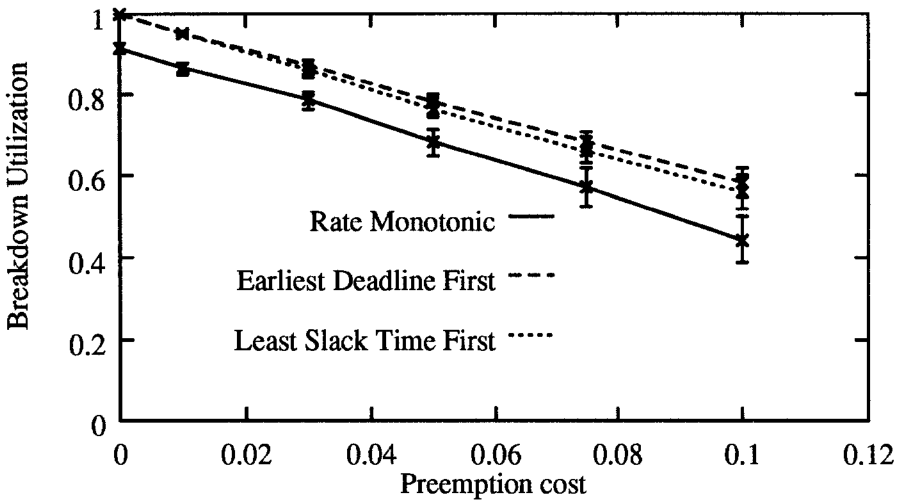

Fig. 2. RM, EDF, LSTF Performance

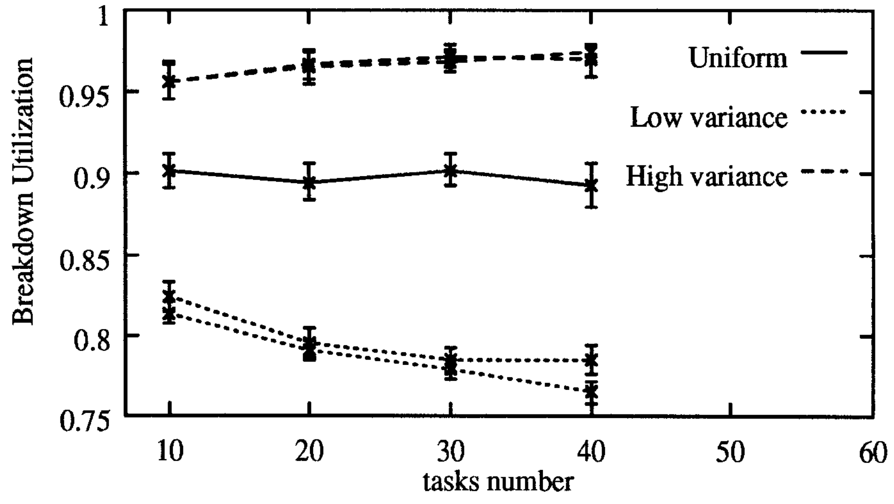

Fig. 3. RM Performance under different task period distributions

frequently to guarantee not to miss the sporadic task hard deadlines.

Moreover, in our model, the scheduling of tasks accessing I/O resources can be governed by one of the preemptive scheduling algorithms mentioned above (i.e. the RM, the EDF, and the LSTF algorithms). In addition, our model allows its user to choose a FIFO discipline for I/O resource management, and to specify arbitrary network delays.

Finally, our model embodies a number of task synchronization protocols that implement concurrency control mechanisms, and solve (or prevent [2]) the *priority inversion* problem [46].

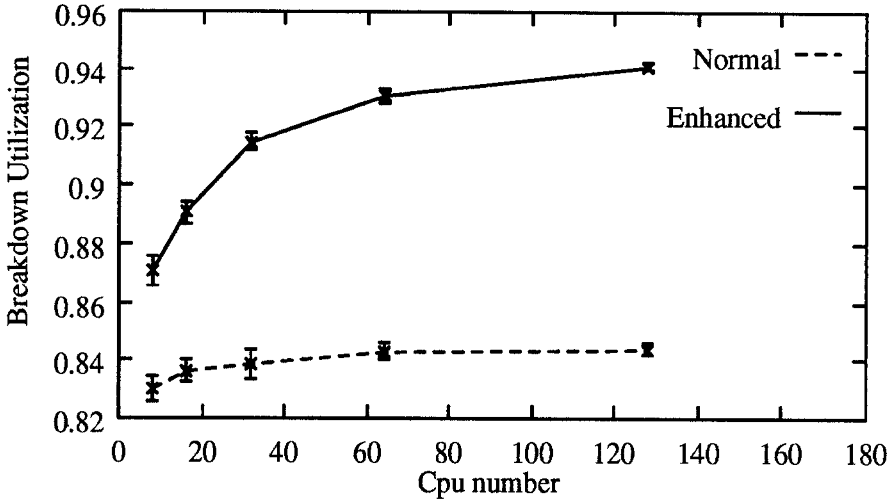

Fig. 4. Allocation algorithms performance

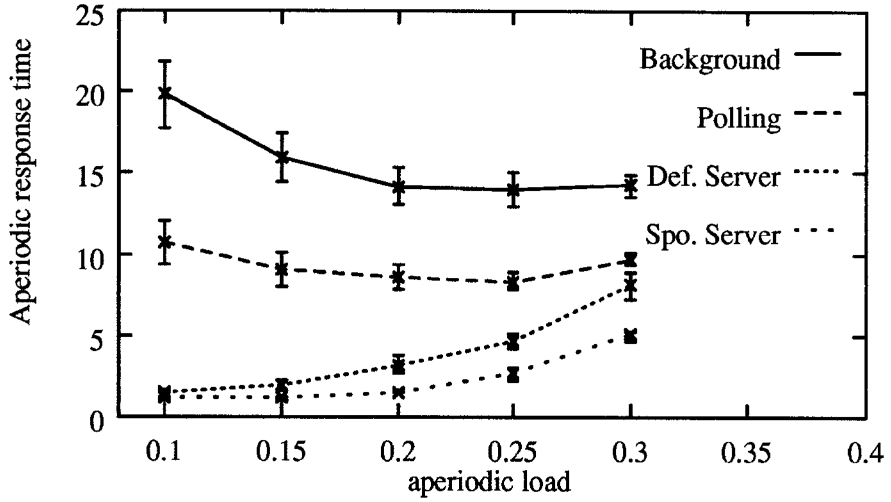

Fig. 5. Background, polling, DS, SS performance

The phrase 'priority inversion' is used to indicate the situation in which the execution of a higher priority task is delayed by lower priority tasks [9]. With priority driven RT schedulers, this problem can occur when there is contention for shared resources among tasks with different priorities. In order to simulate the mastering and control of that problem, our model implements the Basic Priority Inheritance (BPI), the Priority Ceiling (PC), the Priority Limit (PL), and the Semaphore Control (SC) protocols [51]. The principal scope of each of these four protocols is to minimize the so-called Worst Case Blocking Time, i.e. the time interval in which the execution of a higher priority task can be delayed

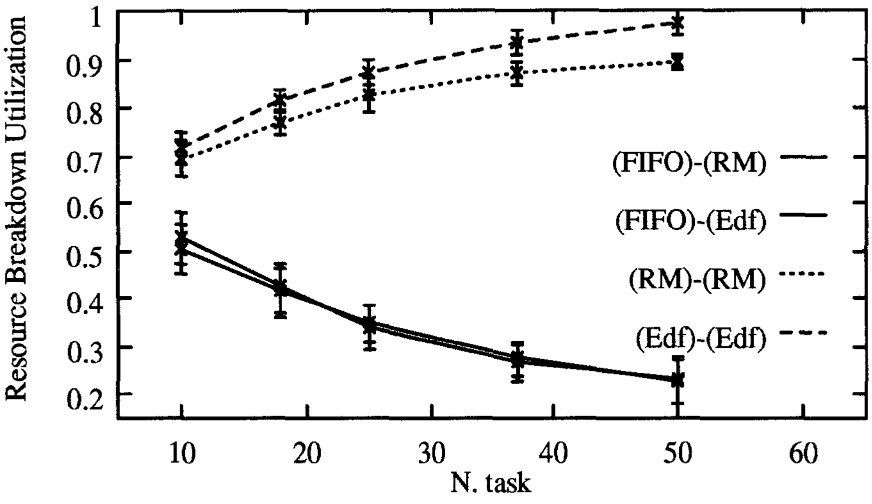

Fig. 6. I/O scheduling performance

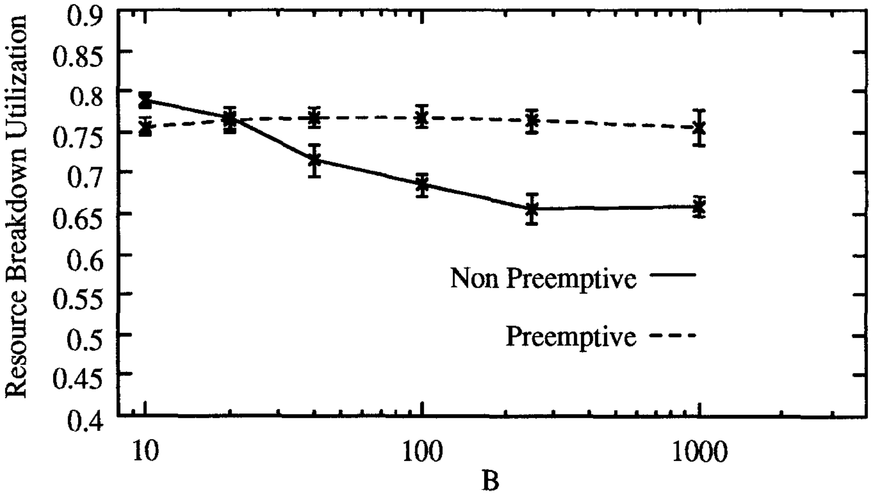

Fig. 7. Preemptive and non preemptive controller (uniform distribution case)

by lower priority tasks.

An alternative approach to the solution of the priority inversion problem has been proposed in [2], and is based on preventing the occurrence of that problem. In order to assess the effectiveness of that approach, our model incorporates a particular priority prevention protocol described in [2].

Our simulation model has been implemented, using the C programming language, so as to accept in input a description of the distributed RT system to simulate, and to produce, as output, statistical results of the simulation experiments.

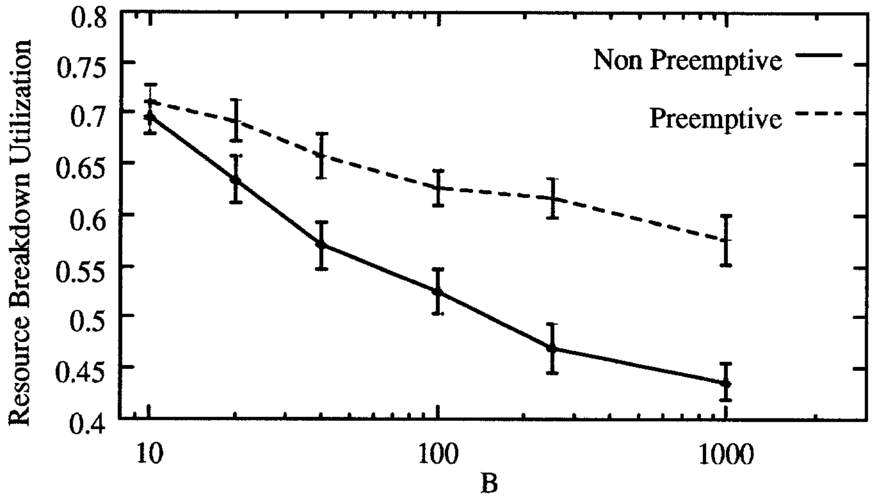

Fig. 8. Preemptive and non preemptive controller (high variance case)

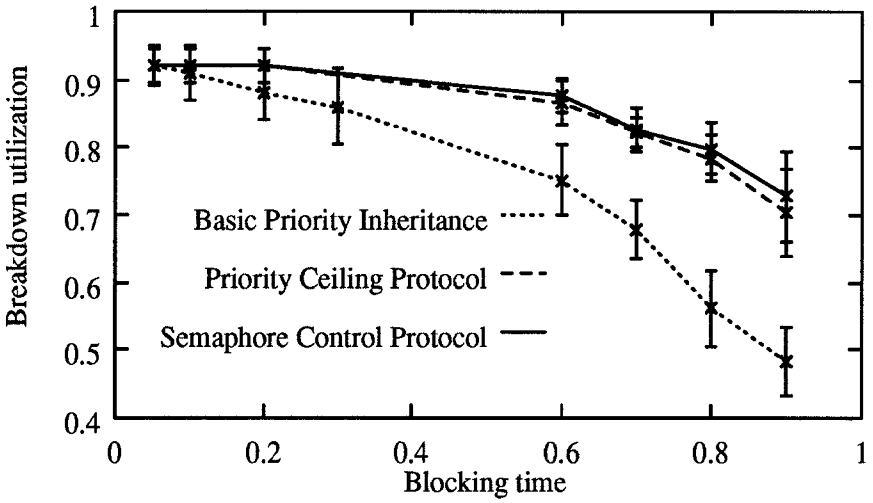

Fig. 9. Priority control protocols performance

The input DRTS description consists of the specification of both system load, and operating system parameters. The system load parameters include the following random variables: number of periodic (PT) and aperiodic tasks (AT) that may request execution, the task period (P), the CPU request (CR) and the deadline (D) of each task, and their probability distribution. The operating system parameters include the scheduling and task synchronization policies the operating system is to use, and the two random variables: operating system preemption cost (PrC), and network overhead (NO).

The output produced by our implementation is intended to allow one to

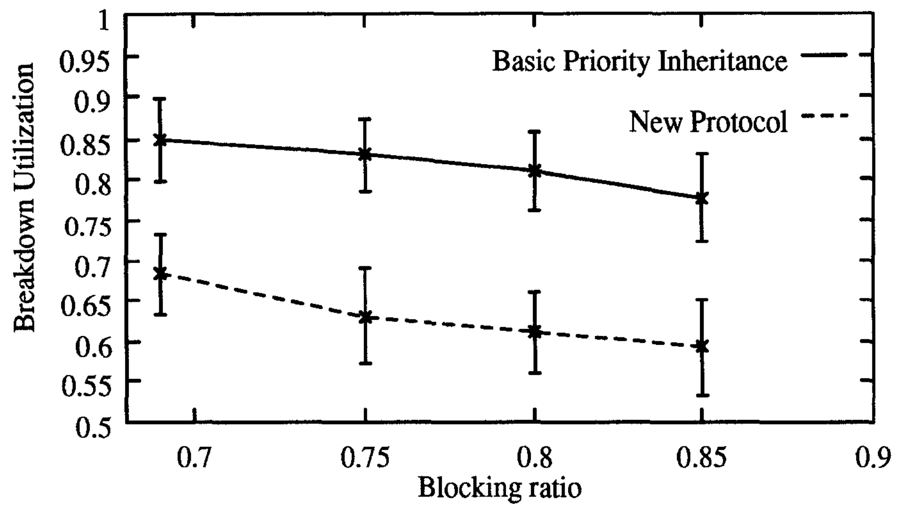

Fig. 10. Comparison between BPI and a priority prevention protocol

evaluate the performance of the various algorithms mentioned above. Thus, our model provides its users with the BU and NMRT figures of merit, introduced earlier.

3.3 Simulation Study

In our simulation study, we have examined the performance of the algorithms introduced previoulsy, under a variety of different system load conditions, and operating system characteristics. The results we have obtained are summarized below.

To begin with, the BU obtained with the RM, EDF and LSTF algorithms, for the scheduling of **periodic tasks**, have been examined as a function of the operating system preemption cost. In the following, we shall assume that the task deadline coincide with the task period, and that the cpu request of a generic task i is 'generated' from the uniform distribution in the interval $[0,p_i]$, where p_i denotes the task i period. The simulation results discussed in this Subsection have been obtained by using the method of independent replications (300 independent runs for each experiment), and 95% confidence intervals have been constructed for the performance indices.

Assuming that:

1. the PrC is the same for each one of these three algorithms,
2. PT is a constant, equal to 10,
3. P is uniformly distributed in the interval [1, 100],

our results show that the EDF and LSTF dynamic algorithms perform better than the RM static algorithm, as illustrated in Figure 2. However, in practice, the above assumption 1 can be unrealistic, as the dynamic algorithms must compute and assign the task priorities at run time, thus introducing additional overheads to the preemption cost; hence, the use of the RM algorithm can be favored to that of the dynamic algorithms, as its implementation is simpler, and the preemption cost it entails is lower.

This observation has led us to concentrate our investigation on the RM algorithm, as far as periodic task scheduling is concerned. Thus, we have examined its behavior as the number of tasks in execution grows. In addition, we have considered the following four different probability distributions of the random variable P:

1. Uniform distribution in [1, 100] (variance = 816.8),
2. Beta distribution with parameters a = 15 and b = 15 (variance = 79.4), and parameters a = 0.5 and b = 0.5 (variance = 1862.2),
3. Normal distribution with parameters mean = 50.5 and variance = 78.4,
4. Exponential distribution with parameter a = 0.5,

The Beta, Normal and Exponential distributions are scaled in the interval [1,100]. The results produced by our simulation model are illustrated in Figure 3.

This Figure shows that the RM algorithm is extremely sensitive to the variance of the random variable P. In particular, low variance of P can notably degrade the RM scheduling performance. In essence, this can be explained as follows. The RM algorithm assigns higher priority to tasks with shorter periods. Thus, if P has low variance, the different task periods are characterized by short time intervals between the periods' terminations. Owing to this observation, we have developed an algorithm that allocates independent tasks to the distributed RT system CPUs, so as to provide a high variance for P on each of these CPUs.

Figure 4 depicts the result produced by our model as a function of the number of CPUs. This Figure illustrates that a conventional task allocation algorithm (indicated as Normal in Figure 4), that ignores the task distribution issue by, for example, polling each CPU in the system until it finds one available for task execution, produces very low BU values compared to our allocation algorithm (indicated as Enhanced in Figure 4).

As to **aperiodic tasks**, the NMRT is the most relevant figure of merit when these tasks are introduced in a distributed RT system, and coexist with the periodic tasks. The experiment we have carried out consisted of simulating the presence (on the same CPU) of both periodic and aperiodic tasks. We assume that :

- the scheduling algorithm used is the RM algorithm,
- the periodic task load is about 69%, and the number of periodic tasks is 10, with period uniformly distributed in the interval [1,100],
- the number of aperiodic tasks is 10,
- the time between consecutive activations of each aperiodic task is exponentially distributed with mean equal to 20,
- the aperiodic task server is the task with highest priority.

The NMRT simulation results we have obtained, as a function of the aperiodic task load, show that the bandwidth preserving algorithms (i.e the DS, SS, IS algorithms) perform better than such traditional algorithms as polling and background, as depicted in Figure 5.

Essentially, this is because the aperiodic task execution can start any time during the server period. Thus complex algorithms, such as DS, SS, and IS, allow the scheduler to start rapidly the execution of the aperiodic tasks. Compared with easier methods, such as polling, these algorithms meet effectively the execution requirements of those aperiodic tasks that request short execution time (even if these requests are very frequent). However, we have observed that, when an aperiodic task requires an amount of CPU execution time close to that of the most time consuming task of the system, the differences among the various methods tend to disappear.

As pointed out in [45], **I/O requests** are scheduled, in general, according to a FIFO discipline; this can lead to a low resource utilization, as illustrated in Figure 6. The results shown in this Figure have been obtained by simulating a system characterized as follows :

1. a variable number of periodic tasks, with period P uniformly distributed in the interval [1,100], are concurrently running in the system,
2. every task is divided in three parts: input, processing, and output. We assume that the time spent during the I/O phase is the same consumed for processing data.

We have considered both non-preemptive and preemptive I/O controllers. A non preemptive controller is one that cannot interrupt an I/O operation once this has been started. With a preemptive controller, instead, a high priority I/O operation can preempt a lower priority one. Consequently, the use of a preemptive controller may appear to be more appropriate in a Real Time system. However, if the RM algorithm is implemented in order to assign priorities to the tasks (and hence to the task I/O requests) the following non obvious results can be observed.

We have carried out a number of simulations that show the performance differences (in terms BU) between the preemptive and the non preemptive controllers.

We have examined the behavior of these two controllers when the task periods are generated with a variety of different distributions. Figure 7 shows the BU values obtained when the task periods are uniformly distributed in the interval [1,B]. It can be seen that the preemptive controller can lead to a greater resource BU for a limited number of values of B, only. Using a low variance distribution (i.e. the normal distribution with variance equal to 78.4) for the period random variable P, we have obtained that, for all values of B, the BU achieved by the non preemptive controller is always greater than that achieved by the preemptive controller. In contrast, using a high variance distribution (i.e. a beta distribution with variance equal to 1862.2) the preemptive controller exhibits its superiority, as illustrated in Figure 8. Moreover, we have noted that the difference in terms of performance between the two kinds of controllers tend to disappear as the number of tasks grows. Thus, the benefits that can be obtained using a preemptive controller cannot be considered as absolute, as these benefits depend upon the system load.

The **priority inversion** problem can typically occur when RT tasks share data. Concurrent accesses to those data can be handled by means of concurrency control mechanisms such as semaphores. However, if a low priority task locks a semaphore, higher priority tasks which require that semaphore are forced to wait its release, thus incurring in a so-called blocking time overhead. Priority control protocols that limit this overhead in a RT system have been developed in order to guarantee the tasks deadlines (i.e. the BPI, SC, PL, and PC protocols already mentioned). As illustrated in Figure 9, these protocols exhibit different performance; in particular, as the blocking time grows, the BPI degrades notably. Instead, the performance of the PC, the SC, and the PL protocols maintain values which are very close to each other (the PL protocol performance results are omitted from Figure 9). However, the SC protocol is an optimal but hard to implement protocol; hence, a number of recent RT system implementations (e.g. Real Time MACH [59]) favor the use of the PC protocol.

Finally, our simulation model implements a recently proposed priority prevention protocol [2]. This protocol differs from the priority control protocols previously examined as it is capable of eliminating the priority inversion problem. Using this protocol, the analysis of a RT system is indeed easier, as less effort is required to construct a feasible schedule for that system. However, the performance of this priority prevention protocol turns out to be lower than that obtained with the priority control protocols discussed above, as illustrated in Figure 10.

4 Case Studies

In this Section we introduce five relevant examples of distributed RT systems; namely, the SPRING kernel, HARTS, MARS, MARUTI, and CHAOS. These systems are discussed below, in isolation.

4.1 SPRING

SPRING is a distributed RTOS kernel developed at the University of Massachusetts. The SPRING designers claim that the development of conventional RT systems has been often affected by a number of misconceptions and implementation deficencies, as discussed at length in [58]. The SPRING kernel [48, 47] aims to overcoming those misconceptions and deficiencies. In particular, the key issues addressed in the SPRING design approach include *flexibility* and *predictability*, within the context of an ET distributed RT system.

In SPRING, tasks are classified as follows, on the basis of their relative costs of a deadline miss:

- Critical tasks (or HRT tasks) are those which must meet their deadlines, otherwise catastrophic failures may occur;
- Essential tasks are those which are indeed relevant to the operation of the system; however, in case of fault, they cannot cause dangerous situations to occur.
- Non-essential tasks may or may not have RT constraints. However, a non-essential task missing a deadline may cause only a poorer quality of service, as its timing constraints are very loose (i.e. those constraints specify a preferred answer time, only).

Each essential and non-essential task is characterized by a "criticalness" parameter associated with it. This parameter is used to quantify the relevance of a task, relative to the specific application it implements. The SPRING kernel executes essential and non-essential tasks by maximizing the value that can be obtained as the sum of the criticalness parameters of those tasks. (Critical tasks are outside the scope of this maximization process, as they are executed with the highest priority.)

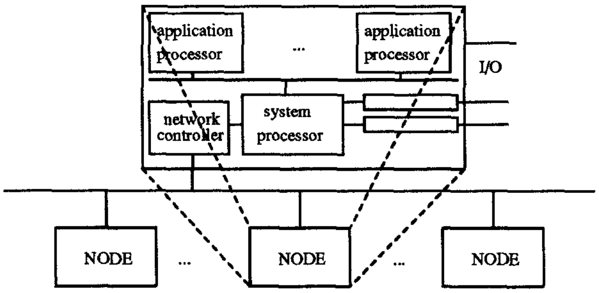

Fig. 11. A schematic view of the Springnet system

The hardware model for the SPRING kernel is a multiprocessor distributed system. Each node is composed by one (or more) application processors, one (or more) system processors and an I/O subsystem (see Fig. 11). Application processors execute critical and essential tasks; system processors run most of the operating system, as well as specific tasks which do not have deadlines. The I/O subsystem handles non-critical I/O, slow I/O devices and fast sensors.

The SPRING kernel is able to schedule task groups. A task group is a collection of tasks having precedence constraints among themselves but sharing a single deadline. Moreover, SPRING supports incremental tasks, i.e. tasks that compute an answer as soon as possible, and continue to refine the return value for the rest of their requested computation time. (A complete discussion on incremental tasks can be found in [35]).

In a loosely coupled processors environment, such as that used in SPRING, an optimal scheduling algorithm, in the worst case, may perform an exhaustive search on all the possible task partitions; this is a computationally intractable problem. SPRING addresses the scheduling problem by using a heuristic approach. In essence, whenever the execution of a task is requested, SPRING attempts to guarantee that execution locally, i.e. it tries to construct a feasible schedule that include that task in the same node where its execution request occurred. If that attempt fails, SPRING allocates a different node for that request. Finally, it is worth mentioning that fault tolerance issues have received little attention in the design of the SPRING Kernel.

4.2 HARTS

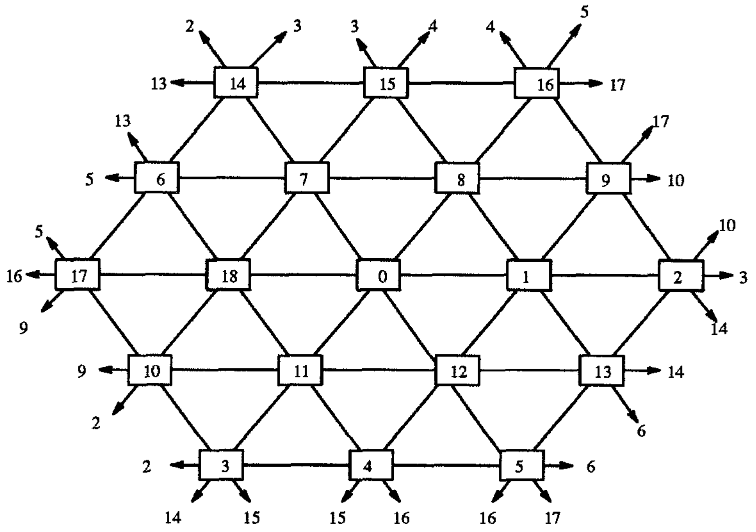

Fig. 12. An example of network mesh for HARTS: the hexagonal mesh of size 3

The distributed RT architecture of the HARTS project (Hexagonal Architecture for Real Time Systems [56]) is based on shared memory multiprocessor nodes interconnected by a wrapped hexagonal network (Fig.12 shows an example of the HARTS hexagonal network). This architecture aims to providing RT applications with high performance, reliability, and predictability.

The HARTS network can be conveniently implemented in hardware (e.g. by means of a VLSI chip), as it is planar, and characterized by a fixed number of connections. This network is scalable, and provides fault tolerance support.

One of the main research interests of the HARTS project is to focus onto low-level architectural issues, such as message routing and buffering, scheduling, and instruction set design. Moreover, as HARTS embodies both distributed system and multiprocessor architectural features, it allows one to evaluate the behavior of RT programs in both distributed and multiprocessor environments.

HARTOS [19] is the operating system developed for HARTS. The HARTOS kernel aims to providing a uniform interface for real-time communications between processes, regardless of their physical location. In particular, the HARTOS link-level protocol supports the co-existence, in the communication network, of a mix of normal and real time traffic. The application interface provides system calls for RT communications, as well as remote procedure calls, naming, and datagram delivery.

In addition, HARTOS supports fault tolerance, queued message passing, non queued event signals, and shared memory (between processors in the same node). Fault-tolerance is implemented at two separate levels of abstraction; namely, the task and network levels. At the task level, HARTOS provides replication features, and multicast communications; instead, at the network level, HARTOS implements a message rerouting scheme that deals with faulty nodes or links. An early implementation of HARTOS was based on pSOS kernel, an operating system for uniprocessor Real-Time systems; the current implementation is based on *x*-kernel [18].

4.3 MARS

MARS (MAintainable Real-time System) [52] is a fault-tolerant distributed real-time system architecture for process control, developed at Technishe Universität Wien. It is intended for use from industrial applications that impose hard-real-time constraints. MARS is designed so as to maintain a completely deterministic behavior even under peak-load conditions, i.e. when all possible stimuli occur at their maximum allowed frequency. MARS guarantees this behavior as it is strictly time driven and periodic. In MARS, all the activities are synchronous, and based on a globally synchronized clock (in particular, the only interrupt present in the system is the clock interrupt, which marks both CPU and bus cycles); this feature favors the system predictability.

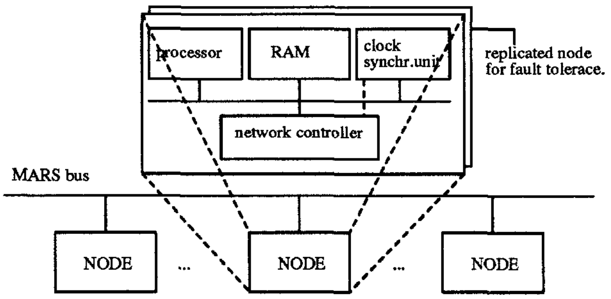

Fig. 13. A schematic view of a MARS cluster

The current implementation runs on a cluster of single-board mono-processor nodes (see Fig.13 for a schematic view of a MARS cluster). From a physical point of view the network hardware of the cluster is a standard ethernet; instead, the channel access protocol is based on a TDMA discipline (as the standard ethernet CSMA-CD (IEEE 802.3) protocol cannot guarantee peak-load timing correctness). As already discussed, a custom chip implements the clock synchronization protocol.

Key issues of this project include:

- fault tolerance: MARS can deal effectively with fail silent nodes and omission failures by using replicated hardware and messages;
- static scheduling: MARS implements static, pre-run time scheduling of the application tasks;
- repearability: in MARS, redundant components may be removed from a running cluster (e.g. for repair), and reintegrated later without affecting the system behavior,
- management of redundant networks (still under development).

Finally, an important development of this project, that is worth mentioning, is the real-time programming environment of MARS [44], a graphical based CASE for RT software development.

4.4 MARUTI

MARUTI [33, 32] is a hard-real-time, fault tolerant, distributed operating system developed at the Department of Computer Science of the University of Maryland. MARUTI is built as a modular system, using an object oriented approach; its architecture emphasizes the independence between the system elements. This system is driven by a time constrained model which imposes restrictions on both execution beginning and ending of the application tasks. In MARUTI, interrupt driven tasks can co-exist with conventional ones. A basic concept that MARUTI implements is that of the *calendar*; this is a data structure which is used to allow the verification of the schedulability of the application tasks, the reservation of guaranteed services, and the synchronization among tasks.

Jobs in MARUTI are invocations of executable objects. MARUTI accepts new jobs during the execution of already accepted jobs; it implements the following two different scheduling disciplines: the off-line and on-line disciplines. Tasks having non deterministic execution, as well as tasks which do not have timing constraints, are scheduled using the off-line discipline; instead, HRT tasks are executed in on-line mode. On-line tasks can preempt, if necessary, off-line ones.

The system provides the following two classes of objects: Kernel and Application level objects. At the Kernel level, MARUTI provides an Interrupt handler object, used to define a service object for any kind of interrupt in the system, a time service object, used for synchronization and ordering of events, and a scheduler. At the Application level, this system includes the allocator, which maps tasks (running in off-line mode) to processors, a file service, and a name service. Objects can communicate by either using shared buffers (if they are running at the same site), or message passing.

4.5 Chaos

CHAOS [17, 53] (Concurrent Hierarchical Adaptable Object System), is a complete programming and operating system for Real-Time applications.

Similar to MARUTI, CHAOS is object-based. The CHOS run-time system offers kernel level primitives which support the development of real-time software structured as a collection of interacting objects. CHAOS is particularly suitable for the development of large, complex real-time applications characterized by stringent timing constraints. Its goal is to support the programming of adaptable, efficient, predictable, and accountable applications.

Efficiency is important since RT applications are time-constrained; hence, system overload should be kept to a minimum so that these constraints can be satisfied. however, the system is to provide efficiency of execution, without compromising accountability, predictability, or reliability.

Accountability means that the kernel must either honor its critical commitments, or it has to report detected failure to the higher level software, before the system reach an unsafe state. (For example, in CHAOS an unanticipated change in the system environment, noted by the kernel, might cause that an invocation miss its hard deadline, and the kernel raise an appropriate exception.) Thus, one of the goals of accountability is to allow the application programmer to develop application specific methods for recovering from failures.

RT applications using CHAOS can be constructed using system provided primitives, customizing these primitives to the applications needs, or by defining new, application-specific primitive operations. CHAOS ensures the predictable behavior of all those synthesized primitives. In addition, it provides mechanisms by means of which the application programmer can monitor the application software, and adapt it to achieve the required performance. CHAOS is particularly suitable for supporting the implementation of robotics applications (the major test bed for CHAOS implementations is a 6-legged walking machine).

As to scheduling, CHAOS uses a two-level scheduling model [54, 15]. The higher level consists of the *object scheduler*; this scheduler receives the invocations and assigns them to specific threads of execution, on the basis of their attributes and processors' load. This scheduler (or allocator) is based on a heuristic greedy algorithm. At the processor level, scheduling is carried out by using an Earliest Deadline First dispatcher. Finally, CHAOS inherits, from the distributed database theory, the concept of atomicity. Transactional methods can be used to drive the invocation of recovery actions that avoid that partial execution results be permanently stored. Atomicity in Chaos refers to RT correctness, as recovery actions may be caused by timing faults.

To conclude this Section, we wish to point out that the five different systems we have introduced have been chosen as they are sufficiently representative of a rather wide spectrum of design choices that can be made in the design of a RT system.

For example, SPRING, although designed to provide ET applications with a predictable RT infrastructure, essentially neglects issues of fault tolerance; rather, its designers have favored the development of a flexible architecture that can accommodate application tasks characterized by different requirements (i.e. critical, essential, and non-essential tasks).

The design of HARTS emphasizes low level architectural issues, and investigates the use of a special-purpose network for distributed RT communications.

MARS is a predictable, time-triggered, distributed architecture; its design exploits the properties of the synchronous systems in order to provide its users with a dependable, real-time, distributed infrastructure for HRT applications.

Finally, MARUTI and CHAOS explore the use of an object oriented approach to the design of fault tolerant RTOSs. However, the design of MARUTI emphasizes issues of integration of conventional, and interrupt-driver RT tasks; instead, CHAOS emphasizes issues of accountability, and support for the development of application dependent fault tolerance techniques.

5 Concluding Remarks

In this tutorial paper we have discussed a number of RT system design issues, and described briefly five examples of distributed RT systems, that have been recently developed. To conclude this paper, we summarize below the principal criteria and metrics that can be used to evaluate RT systems in general, and distributed RT systems in particular.

To begin with, we have pointed out that "timeliness" is indeed a crucial requirement to be met in the design of a RT system; however, this requirement is not sufficient to guarantee the effectiveness of any such system, as a RT system is to be designed so as to be "predictable", primarily.

We have examined and contrasted two principal architectural paradigms for the design of predictable RT systems; namely, the Time Triggered and the Event Triggered paradigms. These two paradigms aim to meeting the predictability requirement mentioned above by implementing static or dynamic strategies, respectively, for the assessment of the resource and timing requirements of the RT application tasks.

Issues of clock synchronization in distributed RT systems have been introduced next. In this context, we have observed that the overhead introduced by the exchange of the clock synchronization messages is a relevant metric to assess the effectiveness of the clock synchronization algorithms that can be used in those systems.

We have then discussed interprocess communication design issues in RT systems. The principal requirements to be met by the communication infrastructure, in order to support RT applications, have been introduced (namely, bounded channel access delay, bounded message delay, and bounded delay jitter). Relevant figures of merit for the evaluation of RT communication mechanisms, that have emerged from our discussion, include: the message loss percentage, the message transmission rate, the deadline miss percentage, the effective channel utilization, and the scalability of the mechanism.

Finally, we have examined issues of scheduling in RT systems, and discussed the results of a simulation study that we have carried out in order to assess a number of scheduling policies. The figures of merit that we have proposed for the assessment of the those policies include: the resource breakdown utilization, the normalized mean response time, and, for dynamic scheduling policies, the guaranteed ratio.

References

1. Anderson T., Lee P. A.: Fault Tolerance - Principles and Practice. London: Prentice-Hall International, 1981
2. Babaoglu O., Marzullo K., Schneider F. B.: A Formalization of Priority Inversion, Technical Report UBLCS-93-4 University of Bologna, March 1993.
3. Bokhari S. H., Shahid H. A Shortest Tree Algorithm for Optimal Assignements across Space and Time in a Distributed Processor System. IEEE Trans. on Software Engineering, SE-7(6), 1981.
4. Cheng S., Stankovic J. A.: Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey. In Hard Real Time Systems, J. A. Stankovic and K. Ramamritham (Eds.), IEEE Computer Society Press, 1988, 150-173.
5. Cristian F., Aghili H., Strong R.: Clock Synchronization in the Presence of Omission and Performance Faults, and Processor Joins. In Proc. FTCS-16, Vienna, Austria, July 1986, 218-223.
6. Cristian F.: Understanding Fault Tolerant Distributed Systems. Comm. of the ACM, (34)2: February 1991, 56–78.
7. Cristian F.: Reaching Agreement on Processor Group Membership in Synchronous distributed Systems. Distributed Computing, 4: 1991, 175–187.
8. Cristian F.: Contribution to the panel: What are the Key Paradigms in the Integration of Timeliness and Availability? (position paper). In Proc. 2nd International Workshop on Responsive Computer Systems, Saitama, Japan, October 1-2 1992.
9. Davari S., Sha L.: Sources of Unbounded Priority Inversions in Real-time Systems and a Comparative Study of Possible Solutions. ACM Operating Systems Review, Vol. 26, N. 2, April 1992, 110-120.
10. Falcone M., Panzieri F., Sabina S., Vardanega T.: Issues in the design of a Real-time Executive for On-board Applications. in Proc. 6th IEEE Symp. on Real-time Operating System and Software, Pittsburgh, PA, May 1989.
11. Ferrari D., Verma D.: A Continuous Media Communication Service and its Implementation. Proc. GLOBECOM '92, Orlando, Florida, December 1992.
12. Ferrari D., Verma D.: A Scheme for Real-time Channel Establishment in Wide-area Networks. IEEE JSAC, (8)3: April 1990, 368–379.
13. Ferrari D.: Design and Applications of a Delay Jitter Control Scheme for Packet-switching Internetworks. In Network and Operating System Support for Digital Audio and Video. R.G. Herrtwich (Ed.), LNCS 614, Springer-Verlag, Berlin Heidelberg, 1992, 72–83.
14. Ferrari D.: Real-time Communication in Packet Switching Wide-Area Networks. Tech. Rep., International Computer Science Institute, Berkeley (CA), 1989.
15. Gheith A., Schwan K.: Chaos^{a}τc: Kernel Support for Atomic Transactions in Real-Time Applications. In Proc. of Fault-Tolerant Computing Systems (FTCS), June 1989.
16. Gonzales, M. J. Jr.: Deterministic Processor Scheduling ACM Computing Surveys, 9(3): September 1977, 173–204.

17. Gopinath P., Schwan K.: Chaos: Why one cannot have only an Operating System for Real-Time Applications. ACM Operating System Review, 23(3): July 1989, 106–140.
18. Hutchinson N., Peterson L.: The x-kernel: An Architecture for Implementing Network Protocols. IEEE Trans. on Software Engineering, January 1991, 1–13.
19. Kandlur D. D., Kiskis D. L., Shin K. G.: Hartos: A Distributed Real-Time Operating System. ACM Operating System Review, 23(3): July 1989, 72–89.
20. Kopetz H. et al.: Real-time System Development: The Programming Model of MARS. Research Report N. 11/92, Institut für Informatik, Technische Universität Wien, Wien (Austria), 1992.
21. Kopetz H., Damm A., Koza C., Mulazzani M., Schwabl W., Senft C., Zainlinger R.: Distributed Fault Tolerant Real-Time Systems: The MARS Approach. IEEE Micro: February 1989, 25–40.
22. Kopetz H., G. Grünsteidl: TTP - A Time-triggered Protocol for Fault Tolerant Real-Time Systems. Research Report N. 12/92/2, Institut für Informatik, Technische Universität Wien, Wien (Austria), 1992.
23. Kopetz H., Kim K. H.. Temporal Uncertainties among Real-Time Objects. In Proc. IEEE Comp. Soc. 9th Symp. on Reliable Distributed Systems, Huntsville (AL), October 1990.
24. Kopetz H.: Six Difficult Problems in the Design of Responsive Systems. In Proc. 2nd International Workshop on Responsive Computer Systems, 2–7, Saitama, Japan, October 1-2 1992.
25. Kurose J. F., Schwartz M., Yemini Y.: Multiple Access Protocols and Time-constrained Communication. ACM Computing Surveys, 16(1), March 1984, 43–70.
26. Lala J., Harper R. E., Alger L. S.: A Design Approach for Ultrareliable Real-Time Systems. IEEE Computer, 24(5): May 1991, 12–22.
27. Lamport L., Melliar Smith L. M.: Synchronizing Clocks in the Presence of Faults. Journal of the ACM, 32: January 1985, 52-78.
28. Lamport L.: Time, Clocks and the Ordering of Events in a Distributed System. Comm. of the ACM, 21: July 1978, 558-565.
29. Le Lann G.: Contribution to the panel: What are the Key Paradigms in the Integration of Timeliness and Availability? (position paper). In Proc. 2nd International Workshop on Responsive Computer Systems, Saitama, Japan, October 1-2 1992.
30. Lehoczky J., Sprunt B., Sha L.: Aperiodic Task Scheduling for Hard Real-Time Systems. In Proc. IEEE Real Time Systems Symposium, 1988.
31. Lehocsky J.P., Sha L., Strosnider J.K.: Enhanced Aperiodic Resposiveness in Hard Real-Time Environments. In Proc. of 8th Real-time System Symposium, Dec.1987
32. Levi S. T., Agrawala A. K.: Real Time System Design. McGraw-Hill, 1990
33. Levi S. T., Tripathi S. K., Carson S. D., Agrawala A. K., The MARUTI Hard-Real-Time Operating System. ACM Operating System Review, 23(3): July 1989, 90–105.
34. Liu C. L., Layland J. W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, 20(1): January 1973, 46–61.
35. Liu J. W., Lin K. J., Shin W. K., Shi Yu A. C., Chung J. Y., Zhao W..: Algorithms for Scheduling Imprecise Computations. IEEE Computer: May 1991, 58–68.
36. Malek M.: Responsive Systems: A Challenge for the Nineties. In Proc. Euromicro 90, 16th Symp. on Microprocessing and Microprogramming. North Holland, August 1990.
37. McNaughton, R.: Scheduling with Deadline and loss Functions. Management Science 6(1), October 1969, 1–12.

38. Meyer J. F.: Closed Form Solutions of Performability. IEEE Trans. on Computers, C-31(7): July 1982, 648–657.
39. Muppala J. K. et al.: Real-time Systems Performance in the Presence of Failures. IEEE Computer, 24(5): May 1991, 37–47.
40. Natarajan S., Zhao W., Issues in Building Dynamic Real-Time Systems. IEEE Software, 9(5): September 1992, 16–21.
41. Ochsenreiter O., Kopetz H.: Clock Synchronization in Distributed Real-Time Systems. IEEE Transactions on Computers, C-36(8): August 1987, 933–940.
42. Owicki S., Marzullo K.: Maintaining Time in a Distributed System. In Proc. 2nd ACM Symp. on Principles of Distributed Computing: August 1983, 295–305.
43. Panzieri F., Donatiello L., Poretti L.: Scheduling Real Time Tasks: A Performance Study. In Proc. Int. Conf. Modelling and Simulation, Pittsburgh (PA), May 1993.
44. Posposchil G., Puschner P., Vrchotichy A., Zainlinger R.: Developing Real-Time Tasks with Predictable Timing. IEEE Software: September 1992, 35–44.
45. Rajkumar R., Sha L., Lehoczky J. P.: On Countering the Effects of Cycle-Stealing in Hard Real Time Environment. In Proc. IEEE Real Time Systems Symposium, 1987.
46. Rajkumar R., Sha L., Lehoczky J. P.: An Optimal Priority Inheritance Protocol for Real-Time Synchronization. ACM TOCS, 17 October 1988.
47. Ramamritham K., Stankovic J. A.: The Spring Kernel: a New Paradigm for Real-Time Systems. ACM Operating System Review, 23(3): July 1989, 54–71.
48. Ramamritham K., Stankovic J. A.: The Spring Kernel: a New Paradigm for Real-Time Systems. IEEE Software: May 1991, 62–72.
49. Ramamritham K.: Channel Characteristics in Local Area Hard Real-time Systems. Computer Networks and ISDN Systems, North-Holland, September 1987, 3-13.
50. Rangarajan S., Tripathi S. K.: Efficient Synchronization of Clocks in a Distributed System. in Proc. Real-time Systems Symposium, San Antonio, Texas, December 4-6, 1991, pp. 22-31.
51. Sha L., Lehoczky J. P., Rajkumar R.: Solution for Some Practical Problem in Prioritized Preemptive Scheduling. In Proc. IEEE Real-Time Systems Symposium, New Orleans, Luisiana, December 1986.
52. Schwabl W., Kopetz H., Damm A., Reisinger J.: The Real-Time Operating System of MARS. ACM Operating System Review, 23(3): July 1989, 141–157.
53. Schwan K., Gopinath P., Bo W.: Chaos: Kernel Support for Objects in the Real-Time Domain. IEEE Transactions on Computers, C-36(8): August 1987, 904–916.
54. Schwan K., Zhou H., Gheith A.: Multiprocessor Real-Time Thread. ACM Operating System Review, 26(1): January 1992, 54–65.
55. Seaton S., Verissimo P., Waeselnyk F., Powell D., Bonn G.: The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In Proc. FTCS-18, 1988, 246–251.
56. Shin K. G.: Harts: A Distributed Real-Time Architecture. IEEE Computer: May 1991, 25–35.
57. Smith R. M., Trivedi K. S. , Ramesh A. V.: Performability Analysis: Measures, an Algorithm, and a Case Study. IEEE Trans. on Computers, C-37(4): April 1988, 406–417.
58. Stankovic J. A.: Misconceptions About Real-Time Computing: A Serious Problem for next-generation Systems. IEEE Computer, October 1988, 10–19.
59. Tokuda H., Nakajima T.: Evaluation of Real-Time Synchronization in Real-Time Mach. In Proc. Mach Symposium 1990, Monterey, CA.

60. Xu J., Parnas D. L.: On Satisfying Timing Constraints in Hard Real-Time Systems. IEEE Transactions on Software Engineering, 19(1): January 1993, 70–84.
61. Zhao W., Ramamritham K.: Virtual Time CSMA Protocols for Hard Real-time Communication. IEEE Trans. on Software Engineering, SE-13(8), August 1987, 938–952.