Increasing Schedulability in Distributed Hard Real-Time Systems

By: José Javier Gutiérrez García, and Michael González Harbour
Departamento de Electrónica, Universidad de Cantabria, SPAIN

Abstract

In this paper we present a study of the effects caused in distributed real-time systems by jitter in the activation of tasks and messages. We show that although jitter has usually a small impact in the schedulability of single-processor systems, in distributed architectures the worst-case response times are significantly delayed. Reducing or eliminating jitter in these systems can increase the schedulability of the system up to 50% more than when jitter is permitted. Jitter can be prevented by using a bandwidth-preserving scheduling algorithm such as the sporadic server. Since this kind of scheduling policy is not designed for communication networks, in this paper we describe how to adapt and implement the sporadic server algorithm for communication networks. Using the sporadic server both in the processors and networks, we can build distributed systems with up to 100% utilization of the CPUs and communication resources, while still guaranteeing that hard real-time requirements are met.

1. Introduction

Distributed real-time systems have an increasing importance in today's control systems, since low-cost networking facilities allow the interconnection of multiple devices and their controllers into a single large system. This architecture is very frequent in industrial environments in which the whole plant is controlled from an integrated system. A high degree of flexibility is achieved in the configuration of the plant, which becomes easier to maintain, upgrade, and adapt quickly to evolving manufacturing needs. Most of the distributed architectures used currently in industrial automation only have soft real-time requirements because of the technological limitations imposed by the interconnection networks and software technology. However, there is great interest in being able to handle hard real-time requirements across the entire system, because this would add more flexibility to the system and could be used to optimize the productivity even further.

One problem that is known to affect the schedulability (i.e., the maximum utilization at which a distributed system can run while guaranteeing that all hard real-time requirements are met) is deferred activation or jitter. The objectives of the work presented in this paper are to carry out a quantitative study of the effects of jitter in the schedulability of distributed real-time systems, and provide solutions to minimize these negative effects. In Section 2 we discuss the model of the distributed real-time system that we consider, and we point to the techniques used to analyze its timing behavior. Section 3 shows the results of the study of the effects of jitter. Section 4 suggests the use of the sporadic server to eliminate the negative effects of jitter. Section 5 provides details of the implementation of this scheduling policy in the communications subsystem. Finally, Section 6 gives our conclusions.

2. Distributed system model

The typical architecture in a distributed real-time control system consists of several processor nodes interconnected through one or more interconnection networks. The system's software is composed of concurrent tasks that are often statically allocated to processing nodes, because the effects of dynamic node

This work has been supported in part by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government, under Grant ROB91-0288

allocation are very difficult to predict for hard real-time systems. Each task may exchange messages with other tasks in the same node or in different nodes. Messages exchanged within tasks in different nodes are also statically allocated to a particular network. Tasks allocated in the same node may also share data or resources, through the usual synchronization mechanisms used in shared memory systems.

Each task in the system is activated by the arrival of a triggering event that may be generated by the environment, a timer, or by the arrival of a message from another task. Many of these events have an associated timing requirement, such as an end-to-end deadline, which is the maximum response time allowed from the arrival of an event to the final completion of the associated response. Events usually arrive in sequences, and events with hard deadlines are usually periodic or have a minimum interarrival time. During the execution of the response to an event, a sequence of several tasks and messages may, respectively, need to be executed in one or more nodes, or transmitted across the networks. Other timing requirements may also be imposed on individual actions within the response.

In order to make it possible to predict worst-case response times, we can schedule the activities in each processing node by using a priority-based approach that allows processing activities with shorter deadlines before (at a higher priority) activities with longer deadlines. One analysis technique used to assess the schedulability of the real-time system is called Rate Monotonic Analysis (RMA) and is well defined for single-processor systems [4]. The RMA techniques can also be applied to distributed systems [5][4][10] by modeling each network as if it were a processor, and each message as if it were a task. Priorities can be assigned to tasks and messages in these systems by using an optimization algorithm that tries to optimize on the ability of the different tasks and messages to meet their deadlines [8][2]. The analysis techniques take into account the effect of jitter in the activation of messages and tasks. The main question that we address in the following sections is how much does jitter influence the response time, and how can we prevent its negative effects.

3. Effects of jitter in the response time

A major problem that appears in distributed systems is the effect of deferred activation of tasks or messages, also called jitter. The activation time of messages generated by the execution of periodic tasks is not perfectly periodic, but depends on the completion time of the triggering task, which is variable. The same happens with tasks activated from messages. In general, jitter in one task causes delay in the worst-case response time of lower priority tasks. The analysis of a system with jitter is addressed by Tindell [10].

Figure 1 shows an example of the importance that jitter has in the schedulability of a hard real-time distributed system. The average of the maximum schedulable utilization of the CPUs and networks is plotted (with and without jitter) as a function of the ratio deadlines/periods (D/T) of the different responses to the events arriving to the system. The graphs are based on an example system with 50 periodic tasks distributed in 8 processors, and 43 periodic messages sent through three different networks. As an example, the maximum utilization for D/T=7 is around 50% if jitter is present, and almost 100% when jitter is avoided. The same kind of results is obtained with other examples. We can see that the difference between the two graphs is larger for longer end-to-end deadlines. Long deadlines are a very common situation in distributed systems, where a response that executes in different resources (CPUs and networks) has a local deadline of one period in each resource, and thus a global deadline equal to the period multiplied by the number of different resources used.

This negative effect on schedulability can be avoided by calculating the worst-case response time of a task that triggers either a message or another task, and creating a high priority agent task that releases the message or task with jitter only after the worst-case response time of the original task has elapsed. However, from a software engineering perspective, this technique is no better than the traditional cyclic executive approach in which the software design was constrained by timing requirements. In this paper we propose the use of the sporadic server scheduling policy [6] to eliminate the problem of jitter in the response time, but preserving the software engineering principle of separation of concerns between logical and timing requirements.

The sporadic server scheduling policy was designed to schedule the execution of aperiodic activities in a hard real-time system. The sporadic server reserves a certain

amount of execution time (the execution capacity) for processing aperiodic activities at the desired priority level. When an aperiodic event arrives, it activates the execution of an aperiodic task. The arrival time of the event is recorded for future reference. As the task is processed, the execution time spent is subtracted from the capacity, until the capacity gets exhausted. At this point, execution of the aperiodic task is suspended. Each portion of execution capacity that is consumed is replenished, i.e., added back to the available capacity, at a later time. This time is equal to the instant when the portion of consumed capacity became active (usually the instant when the triggering event arrived), plus a fixed time called the replenishment period. In some implementations, the aperiodic task can continue executing at a background priority level.

With the sporadic server policy, the effects of processing aperiodic activities using a sporadic server can be no worse than the equivalent effect of a periodic task with a period equal to the replenishment periodic and an execution time equal to the initial execution capacity of the sporadic server. Suppose a periodic task that suffers from deferred activation. If this task is scheduled using a sporadic server with a replenishment period equal to the task's period and an initial capacity equal to the task's worst-case execution time, then its effects on lower priority tasks cannot be worse than that of an equivalent periodic task, with no deferred execution. The worst-case completion time of the task itself is equal to the worst possible deferral in the activation, plus the worst-case completion time of the purely periodic task, which is easily obtained from the usual RMA schedulability analysis.

4. Optimized scheduling in communication networks

The sporadic server can be adapted to schedule message traffic in a network. Most of the currently available networks and network protocols do not use priorities, and many have unpredictable worst-case transmission times. However, some networks have been used in hard real-time communications, such as TDMA [10], token rings [7][1], the CAN bus [9], etc. Network protocols can be implemented with message priorities in order to increase the message schedulable utilization. For example, the CAN bus uses the message identifiers as a means to control access to the bus and also as the message priorities.

A network layer with priority preemptive messages can be implemented by dividing messages into fixed-size packets. Each message carries a network-wide priority, and all of its packets are assigned the same priority. The network scheduler has at each node a priority queue of message packets to be transmitted (Figure 2-a). An individual packet is non-preemptible (i.e., once its transmission starts no high priority messages can interrupt it), but higher priority packets are always inserted before lower priority packets at the scheduler's queue. In this way, high priority tasks do not get blocked by the transmission of a potentially very long message from a lower priority activity.

In this priority scheduling context, it is easy to adapt the sporadic server concept (Figure 2-b). For a given sequence of messages, the sporadic server parameters are the initial capacity, or number of packets that may be transmitted at the normal priority, and the replenishment period. The network scheduler can transmit as many as the reserved number of packets at the normal priority. For each of these packets transmitted, the capacity to transmit another packet is replenished one replenishment period after the packet was queued.

In the sporadic server scheduler for tasks, we associate one sporadic server to each task. However, for messages there is no such concept as a task, which wraps multiple responses associated to a particular event sequence; messages arrive at the communications subsystem with no apparent relation to other previous messages. As a consequence, we have defined each sporadic server scheduler to work over any message of a particular priority level. This means that a sufficient number of priorities is needed to provide each message sequence with a different priority level.

Figure 2-b shows the general structure of a conceptual sporadic server message scheduler. The sporadic server has the same interface as the conventional priority message scheduler (Figure 2-a) with the addition of an operation to set the attributes of each sporadic server. These attributes are:

- *Initial capacity*: number of message packets that may be transmitted at the normal priority without restriction
- *Replenishment period*. Time that has to elapse from the instant when a message packet was queued until the server regains the capacity consumed by the

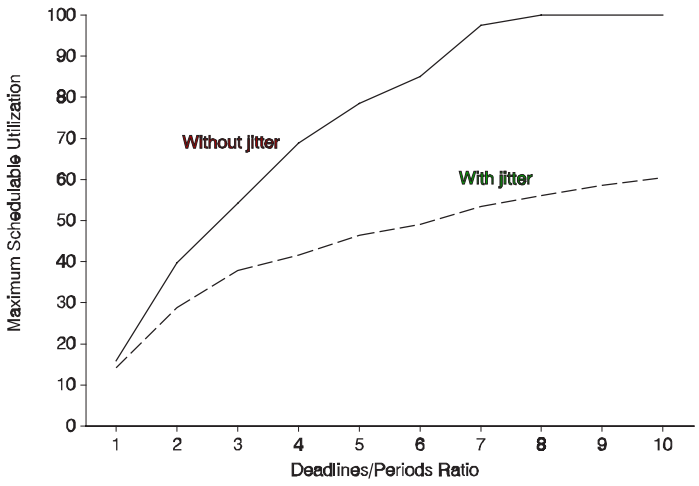

Figure 1. Effects of jitter on the maximum schedulable utilization

transmission of that packet.

For each priority level, and thus for each sporadic server, there is a per-priority state that is defined by the following parameters:

- *Actual Capacity*. At any instant, it is the number of message packets that may be transmitted at the associated priority. Initially, the actual capacity is equal to the initial capacity.
- *Activation time*. This parameter is set equal to the current time at two situations: when message packets of the associated priority are queued to a sporadic server with an actual capacity greater than zero and there are no other messages queued; and also when a replenishment is executed (see description below).
- *Used Capacity*. Number of message packets consumed (i.e. set at the associated priority) since the last recorded activation time.

According to the state of the queue and the value of the actual capacity, a sporadic server may be in three different states:

- *Idle*: No packets are queued for that priority
- *Normal*: There are packets queued for that priority and the capacity is greater than zero.
- *Background*: There are packets queued for that priority and the capacity is zero.

There is a special object that is responsible for managing the replenishment operations for all the sporadic servers, which is called the replenishment manager. It has a queue of pending replenishments, which are sorted according to their replenishment time. Each replenishment operation stored in this queue has the following information:

- *Replenishment time*. It is the instant at which a replenishment expires, i.e. when a portion of consumed capacity can be added back to the actual capacity.
- *Replenished capacity*. It is the capacity to be replenished, in number of packets.
- *Priority*: It is the priority level that has to be replenished.

The operations that may be invoked on the replenishment manager are:

- *Schedule a replenishment*. This operation consists of adding a replenishment to the replenishment queue. The replenishment time, capacity and priority must be supplied by the caller.

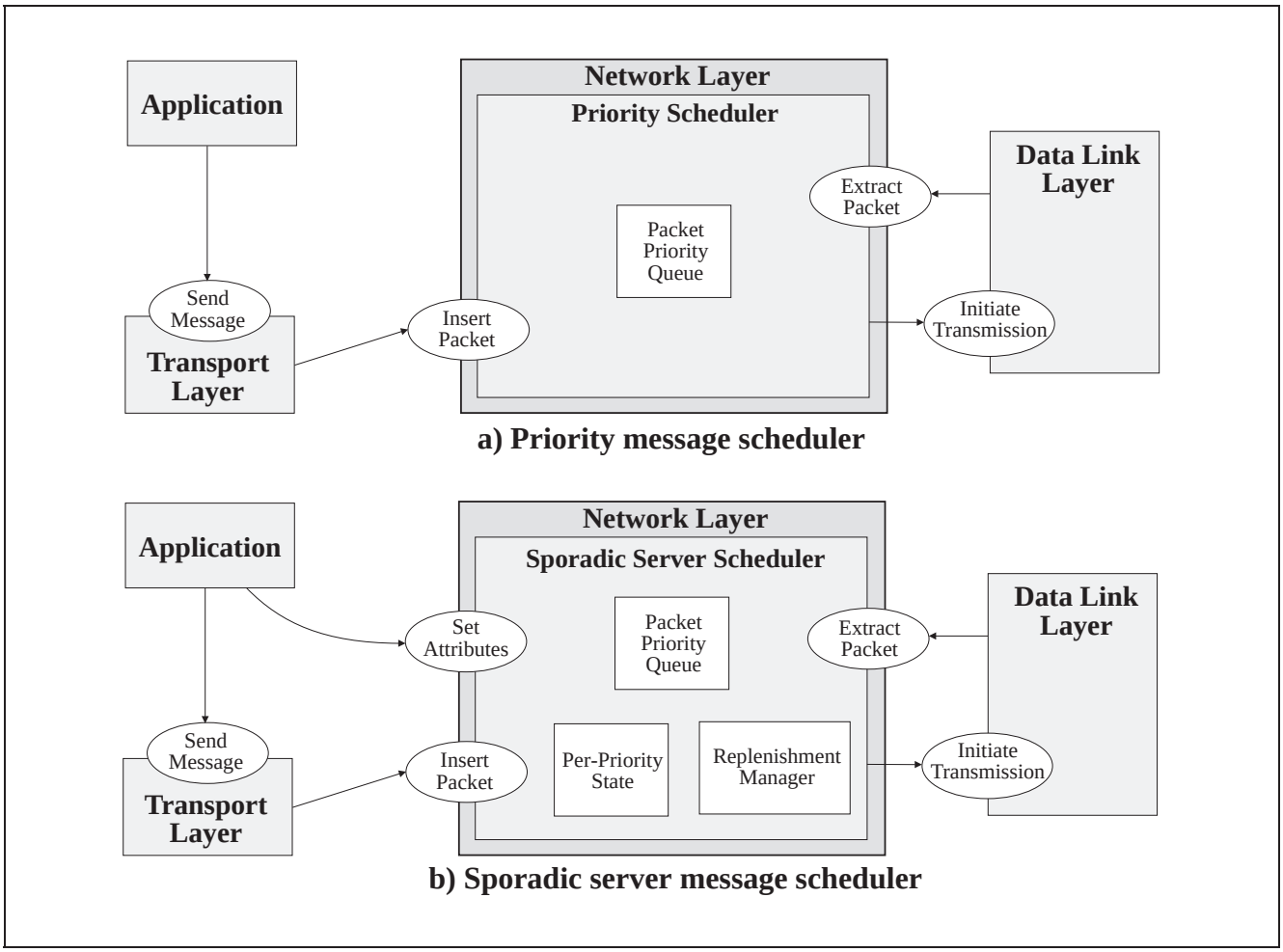

Figure 2. Structure of the Network Message Scheduler

- *Execute pending replenishments*. This operation consists of extracting from the replenishment queue all the replenishment operations that have expired (i.e., whose replenishment time is before or at the current time). For each replenishment operation, the replenished capacity is added to the actual capacity of the associated priority, and the activation time is recorded. If that actual capacity was zero and the state of the associated sporadic server was background, it is switched to normal.

The basic operations of the sporadic server scheduler are the following:

- *Set attributes*. The initial capacity and replenishment period are stored for the requested priority
- *Insert*. Introduce a message packet into the queue of the specified priority, in FIFO order. If the state of the associated sporadic server was idle, it is switched to normal or background (respectively for capacity>0 or capacity=0). If the state is switched to normal, the activation time is recorded.
- *Extract*. Executes the pending replenishments, and then extracts a message packet from the queue. This operation is usually invoked from the data link driver, to select a packet to be transmitted. The queue is searched for the first highest priority message packet whose associated server is in the normal state. If there are no packets in the normal state, the highest priority packet in the background state is extracted. Extracting a packet that is in the normal state implies consuming one unit of actual capacity and, if the actual capacity becomes zero, or if the sporadic server becomes idle (because there are no more packets queued at that priority), scheduling a replenishment operation. The replenished amount is equal to the used capacity, and the replenishment

time is set to the recorded activation time plus the replenishment period. Extracting a packet that is in the background state does not consume any capacity (recall that in this case it is already zero), and can only be done if no packet is queued in a server in the normal state.

5. Implementation of hard real-time communications with optimized scheduling

5.1. Message queues

We have implemented a hard real-time communication system that uses the sporadic server scheduling policy, in order to prove its implementability using current network technology, and also as an experimental tool with which we can do further research in this area.

The implementation has been developed in Ada, using for the transport layer the interprocess communication interface specified in the POSIX.1b standard [3]. This interface defines message queue operations such as send or receive a message, and open or close a message queue. Each message has an associated priority that is used to queue the message in priority order. Although the interface was designed for a single node, we have implemented it for a distributed system, in a network-transparent way. This introduces a high degree of flexibility: Ada tasks communicate with each other without knowing if the communication is local, or across the network; this aspect is determined at configuration time, when an optimization technique may be used to allocate tasks to nodes and messages to networks. Our implementation only uses a common extension to the Ada 83 language, which is the CIFO 3.0 suspend/resume facility (or alternatively dynamic priorities), and it will be completely portable using the Ada 95 features.

The message priorities are used to implement a priority-preemptive message scheduler based on the transmission of bounded-length packets. This hard real-time message scheduler has been implemented for two kinds of interconnection networks: A VME bus, in which shared memory is used to implement the transmission of messages to and from message queues, and point-to-point networks such as serial lines (RS-232 and RS-485). Two or more of these networks can be used in parallel in a heterogeneous configuration.

5.2. Operations on remote message queues

When the message queue whose use is requested is in another node, a message is sent to the remote node with

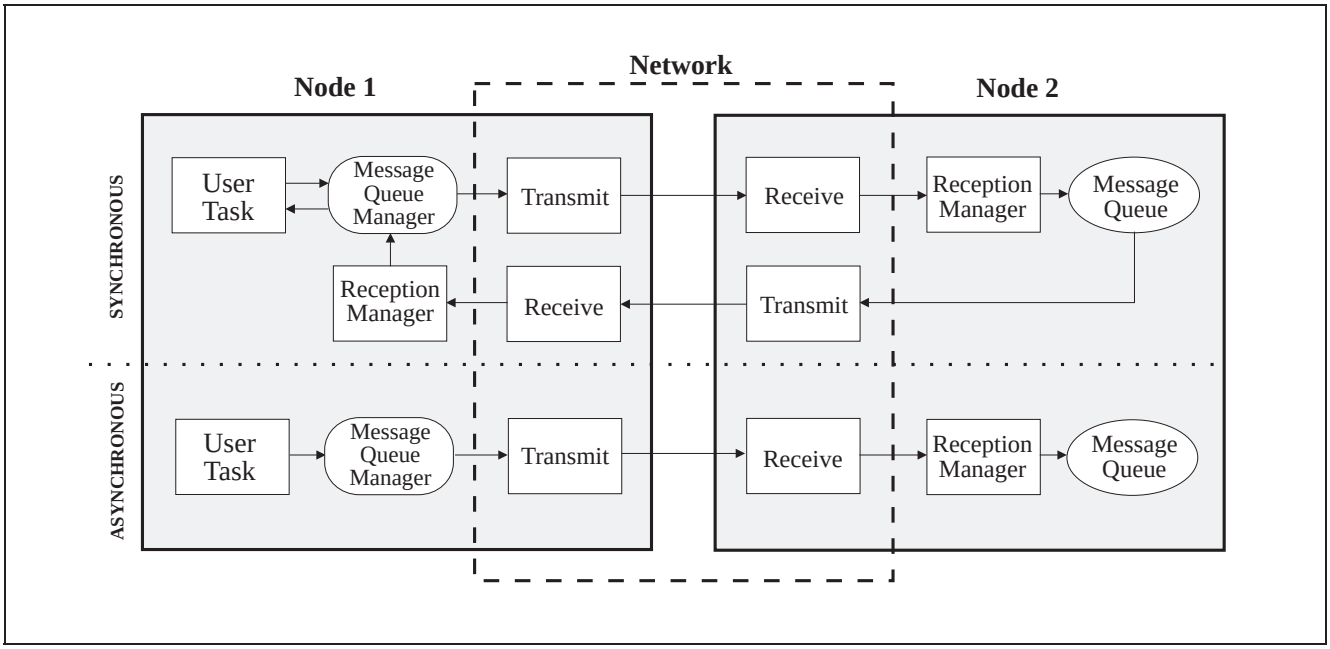

Figure 3. Models of operations on remote message queues

the information that is required to perform the operation. In some cases, a response message with additional information is required. Figure 3 shows a high-level model of operations affecting message queues in remote nodes. There are two different kinds of operations:

- *Synchronous*. The task that invokes the operation is suspended waiting for the response from the remote node. Once the response arrives, the task may continue executing. An example of a synchronous remote operation is receiving a message from a remote queue.
- *Asynchronous*. The task invoking the operation does not suspend, and may continue its execution after the call. An example is sending a message to a remote queue.

The communication network is used both for transmitting application messages, as well as messages required by the implementation of message queues. For example, to receive an application message from a remote queue we must first send a message requesting transmission. A protocol has been defined to handle all the different kinds of application and implementation messages. All these messages are divided into packets for transmission across a specific network. Several networks may be used in parallel, but the allocation of message queues to networks is static, and is determined at configuration time.

5.3 Implementation of the data link layer

We have implemented a data link layer for two kinds of networks: point-to-point serial lines, and a VME bus. Point-to-point line drivers are very easy to implement, because there is no priority arbiter needed. The line can only be operated from one single node, and there is no contention with other nodes. In this case, a conventional driver (i.e., not based on priorities) may be used.

The implementation of the data link layer is more complex in networks in which several nodes are able to access the network at the same time. In these networks, the contention must be solved based on the priorities of the messages to be transmitted. For example, in the CAN bus, the message identifier is used to resolve contention and effectively acts as a message priority.

We have implemented the data link layer for real-time communications across a VME bus, which is a very common bus in industrial applications. In our implementation, we use shared memory to exchange messages between the different nodes. Transmission is achieved by copying the packet into a buffer located in shared memory. A global data structure is created also in shared memory to store the information that is required to implement the priority-based arbitration. Although this structure is global it does not represent a major contention problem, because the VME bus itself is already a global resource shared by all the nodes in the system. Access to the global control structure is protected through a bus semaphore. The information stored in this data structure is the maximum priority of the packets that are ready for communication at each node, and two boolean parameters that, respectively, reflect whether there are packets to transmit, and whether an interrupted node is requested to transmit or receive a packet. The operation of the bus driver is as follows:

a) *Packet queuing*. When a packet is queued for transmission and the bus is idle, the packet is transmitted (see b).
b) *Packet transmission*. The packet is copied to shared memory, and the receiver node is interrupted to receive the packet (see c).
c) *Reception Interrupt*. The transmitter of packet interrupts the receiver node when the packet is ready to be retrieved. The receiver node copies the packet from shared memory, and then checks if more packets need to be transmitted (see d)
d) *Checking for transmission*. If there are packets to transmit, the control data structure is checked to determine which is the node with the maximum priority packet. If it is the node executing the operation, the packet is transmitted (see b); otherwise, the corresponding node is interrupted for transmission (see e).
e) *Transmission interrupt*. When a node is interrupted for transmission it re-checks for transmission (see d).

In order to execute the above operations, the sporadic server scheduler has to include an operation to query the priority of the highest priority packet to be transmitted. In addition, when a replenishment operation switches the state of a sporadic server from background to normal, and the maximum priority of the packets pending for transmission in that node changes, the global control structure must be updated accordingly. Finally, the replenishment manager in the sporadic server scheduler must be an active object, because replenishment operations

must be carried out at the time when they expire, to maintain the consistency of the global control structure. In the sporadic server implementation for point-to-point networks the replenishment manager was a passive object (executed in the environment of the task that extracts a packet from the server) because replenishments only needed to be executed when a new packet was going to be transmitted.

5.4. Modeling the real-time performance of message queues

The performance of this implementation has been measured to determine the overheads introduced in the communication by the sporadic server scheduling algorithm. This overhead is modeled in the real-time analysis process as extra transmission time, as well as extra messages. Access to shared resources such as the message queues or the packet priority queues is protected with synchronization primitives, and thus is modeled as blocking time, in the usual way synchronization is treated in the real-time analysis. The interrupt handlers and the task that manages the reception of messages have to be included in the analysis as additional tasks. The sporadic server for point-to-point communications does not need any special treatment; we just have to account for a slight increase in the execution times of the message scheduler operations (about a 50% increment compared to the normal priority scheduler). In the VME bus implementation, the sporadic server introduces a new task, the replenishment manager, that has to be taken into account in the analysis. The model of the communications system that has been developed together with the measurements of the execution times of each of its actions can be used to predict the worst-case response time of any hard real-time distributed application using this application.

6. Conclusions

In this paper we have shown that the schedulability penalty introduced by the effects of deferred activation in hard real-time distributed system is very large. By simply eliminating jitter we can increase schedulability, up to a surprising 50% increment. Jitter can be eliminated by using the sporadic server scheduling mechanism both to schedule tasks in each of the processing nodes, as well as messages in each of the networks.

Although the sporadic server was designed as a scheduling algorithm for processors, in this paper we have shown that the concept can be easily adapted to schedule messages in a real-time communication network. We have implemented this algorithm in Ada using the POSIX.1b message queue interface which provides for message priorities. Our implementation allows us to design, implement, and analyze hard real-time distributed systems with a high degree of utilization and guaranteeing that all hard real-time requirements are met.

References

[1] G. Agrawal, B. Chen, W. Zhao, and S. Davari. "Architecture Impact of FDDI Network on Scheduling Hard Real-Time Traffic". Workshop on Architectural Aspects of Real-Time Systems, December 1991.

[2] J.J. Gutiérrez García, and M. González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems". 3rd Workshop on Parallel and Distributed Real-Time Systems, Santa Barbara, CA, 1995.

[3] IEEE Standard 1003.1b-1993, "Draft Standard for Information Technology —Portable Operating System Interface (POSIX)— Part 1: System Application Program Interface (API) — Amendment 1: Realtime Extension [C Language]". The Institute of Electrical and Electronics Engineers, 1993.

[4] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour, "A Practitioner's Handbook for Real-Time Systems Analysis". Kluwer Academic Pub., 1993.

[5] L. Sha, and S.S. Sathaye, "A Systematic Approach to Designing Distributed Real-Time Systems". IEEE Computer, September 1993, pp. 68-78

[6] B. Sprunt, L. Sha, and J.P. Lehoczky. "Aperiodic Task Scheduling for Hard Real-Time Systems". The Journal of Real-Time Systems, Vol. 1, 1989, pp. 27-60

[7] J.K. Strosnider, T. Marchok, J.P. Lehoczky, "Advanced real-time scheduling using the IEEE 802.5 Token Ring", Proceedings of the IEEE Real-Time Systems Symposium, Huntsville, Alabama, USA, 1988, pp. 42-52.

[8] K. Tindell, A. Burns, and A.J. Wellings. "Allocating Real-Time Tasks. An NP-Hard Problem Made Easy". Real-Time Systems Journal, Vol. 4, No. 2, May 1992. pp. 145-166.

[9] K. Tindell, A. Burns, and A.J. Wellings, "Calculating Controller area Network (CAN) Message Response Times". Proceedings of the 1994 IFAC Workshop on Distributed Computer Control Systems (DCCS), Toledo, Spain, September 1994.

[10] K. Tindell, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems". Department of Computer Science Report, University of York, 1993.