Technical Report
CMU/SEI-91-TR-026
ESD-91-TR-026

An Application-Level Implementation of the Sporadic Server

Michael González Harbour
Lui Sha

September 1991

Technical Report

CMU/SEI-91-TR-026
ESD-91-TR-026
September 1991

An Application-Level Implementation of the Sporadic Server

Michael González Harbour
Lui Sha

Rate Monotonic Analysis for Real-Time Systems Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.   Phone: (703) 274-7633.

Table of Contents

List of Figures

List of Tables

Acknowledgments

The stay of Michael González Harbour, first author, at the Software Engineering Institute has been funded in part by the university to which he belongs, the University of Cantabria, Spain, and by the “Dirección General de Investigación Científica y Técnica” of the Spanish government.

The authors would like to thank John Goodenough for his thorough review of this paper and his many insightful comments.

An Application-Level Implementation of the Sporadic Server

Abstract: The purpose of this paper is to introduce a sporadic server algorithm that can be implemented as an application-level task, and that can be used when no runtime or operating system level implementation of the sporadic server is available. The sporadic server is a simple mechanism that both limits and guarantees a certain amount of execution power dedicated to servicing aperiodic requests with soft or hard deadlines in a hard real-time system. The sporadic server is event-driven from an application viewpoint, but appears as a periodic task for the purpose of analysis and, consequently, allows the use of analysis methods such as rate monotonic analysis [1] to predict the behavior of the real-time system.

When the sporadic server is implemented at the application-level, without modification to the runtime executive or the operating system, some of its requirements cannot be met strictly and, therefore, some simplifications need to be assumed. We show that even with these simplifications, the application-level sporadic server proposed in this paper has the same worst-case performance as the full-featured runtime sporadic server algorithm, although the average case performance is slightly worse. The implementation requirements are a runtime prioritized preemptive scheduler and system calls to change a task's or thread's priority. Two implementations are introduced in this paper, one for Ada and the other for POSIX 1003.4a, Threads Extension to Portable Operating Systems.

1 Introduction

1.1 Background

Hard real-time systems have been defined as those systems in which system failure occurs if the timing constraints of the system are not met. Real-time systems are usually composed of a set of tasks that can be classified according to their criticalness and the nature of their timing requirements. Some typical timing requirements in real-time systems are periodic tasks with deadlines and aperiodic tasks with deadlines or with average response requirements.

The rate monotonic scheduling (RMS) algorithm was originally introduced by Liu and Layland [6] for scheduling independent periodic tasks with hard deadlines at the end of their period. Rate monotonic theory [1] has been extended to the design and analysis of task sets with synchronization [7], mode changes [8], input/output [9], aperiodic tasks [2] [3], and synchronization in multiprocessors [12]. Also, an exact schedulability analysis for a given set of tasks with deadlines at the end of their period has been developed [10], and this analysis has recently been extended to a schedulability analysis of tasks with deadlines before or after the end of the period [11]. RMS supports analysis for sets of tasks with critical and non-critical deadlines, allowing an analytical guarantee of the scheduling feasibility of the set of tasks considered critical, even in situations of transient overload. This, along with the simplicity of the analysis is one of the major issues that distinguishes RMS from the earliest deadline scheduling algorithm [6]. Typically, a system running under RMS can achieve 100% utilization of the processor by allocating 90% of the processor to tasks with hard deadlines (with an analytical guarantee of meeting each deadline), and the other 10% to background tasks (i.e., tasks with no deadlines).

The mechanism provided by rate monotonic theory for the scheduling and analysis of aperiodic tasks is the sporadic server; this mechanism preserves and limits a certain amount of high priority execution for the processing of aperiodic events, while guaranteeing the deadlines of all the other tasks in the system, even under burst conditions in the arrival of aperiodic processing requests (i.e., large number of requests in a short time interval). The sporadic server is easily incorporated into rate monotonic analysis, because aperiodic tasks can be analyzed as if they were periodic.

The key concept of the sporadic server is to provide a limited amount of *computation budget*, *C*, for processing aperiodic events at their assigned priority during a time interval called the *budget replenishment period*, *T*. Once the sporadic server is initialized with its budget and period, it reserves its execution time until an aperiodic request arrives. The request will be serviced (whenever there are no higher priority activities pending) as long as the execution budget has not been exhausted. If the request is completed within the available budget, the actual execution time used is subtracted from the budget, and a replenishment of this amount of execution time is scheduled for one replenishment period after the arrival of the aperiodic request. If the request is not completed before the execution budget is exhausted, the aperiodic task is assigned a background priority. When the replenishment period expires, the execution time is replenished and, if the sporadic server was executing at a background priority, its priority is elevated to its normal level. Since the amount of execution time is limited, a sporadic server can be designed so it does not cause lower priority tasks to miss their deadlines, even when a burst of requests arrive.

The sporadic server represents a better approach to scheduling aperiodic events than traditional methods. Two common approaches for servicing aperiodic requests are background processing (using the processor's idle time) and polling. Polling consists of creating a periodic task for servicing aperiodic requests. At regular intervals, the polling task is started and it services any pending aperiodic requests. However, if no aperiodic requests are pending, the polling task suspends itself until its next period. If a request arrives just when the polling task has suspended, it has to wait until the start of the next period. In the next example we show that the sporadic server can improve the worst-case response time to an aperiodic event by a factor of 100, compared to polling.

> **Example 1**. Consider a system with one periodic task with execution time of 99 units and period of 100 units. This system has to service an aperiodic request which arrives randomly once in a period of 100. Figure 1 shows the performance of the polling and sporadic server solutions.

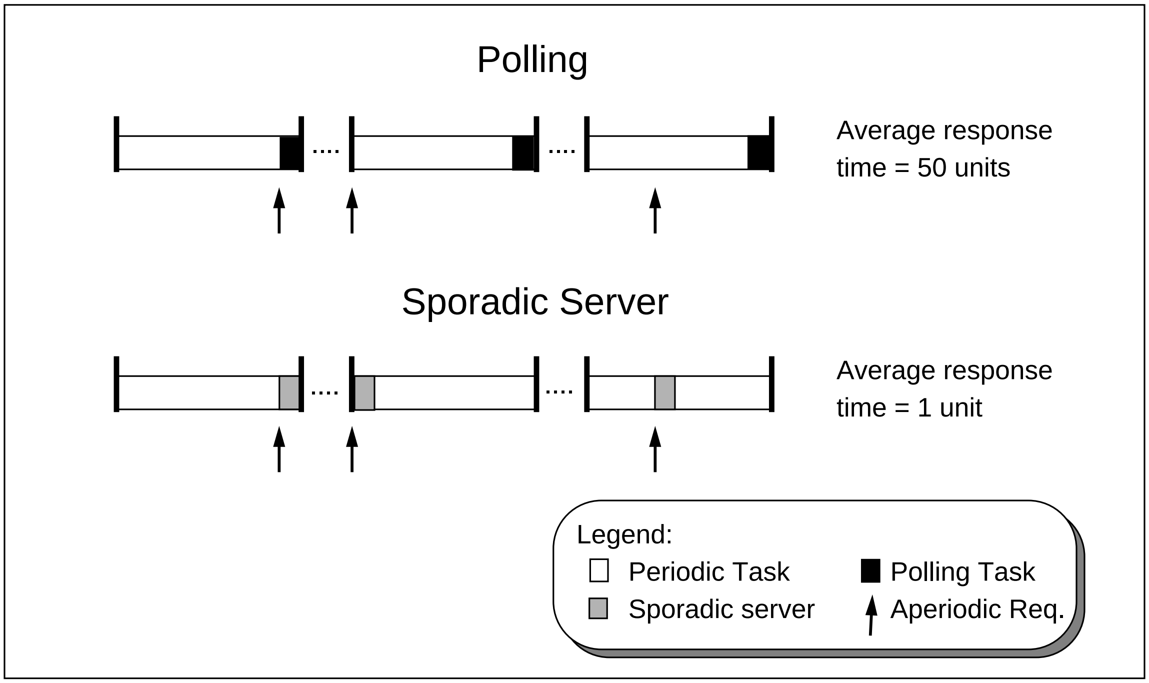


Figure 1 Comparison Between a Sporadic Server and a Polling Task

> In the polling solution, if the request arrives just when the polling task has suspended, the servicing start time will be 99 units, and therefore the worst-case response time will be 100 units. If a sporadic server is used with a priority higher than the periodic task, the worst-case response time is precisely one unit and the deadlines of the periodic task are still guaranteed. Moreover, if an unexpected burst of requests occurs, the sporadic server algorithm will limit the amount of high priority processing, thus ensuring that the timing requirements of all lower priority tasks are not endangered. It will also process the excess of work at a background priority level, giving it the opportunity to "catch up" with the following normal arrivals.

The sporadic server can be applied to scheduling aperiodic tasks with different kinds of timing requirements, such as individual deadlines, average response times, etc. It also has many other interesting applications [3], such as scheduling producer/consumer tasks, implementing the period transformation technique in a transparent way [1], or detecting and limiting the effects of faults on the estimation of task execution time requirements. Also, the sporadic server can be used to guarantee the periodic behavior of tasks which, because of deferred execution[1] or jitter, have a negative impact in the schedulability of lower priority tasks [12]. This feature can

[1] The deferred execution effect appears when instances of periodic tasks (jobs) are activated irregularly, with different intervals between activations. Under these circumstances if a particular job of a given task is activated late and the next job is activated early, two task activations occur during a time window equal to the task's period, and this may have a negative impact on the schedulability of lower priority tasks [12].

also be applied to scheduling of networks in distributed systems with periodic messages showing deferred activation.

1.2 The Application-Level Sporadic Server

In order to implement a sporadic server at the application-level, some of the requirements of the sporadic server algorithm must be relaxed. In particular, there are some functions in the sporadic server algorithm that can only be performed by the runtime executive, such as measuring the actual execution time of a task. Sprunt and Sha [3] developed an application-level sporadic server implemented with an Ada task, in which the full-featured sporadic server algorithm was subjected to some restrictions. The main restrictions were: no measurement of execution time (worst-case execution time is always assumed), no background execution, and a suboptimal replenishment policy. Although these restrictions may significantly decrease both the average and worst-case performance of the sporadic server, they provide the benefit of a full standard Ada compatibility.

In this paper, a new implementation of the sporadic server at the application-level is proposed, in which background execution is allowed and a better replenishment policy is used. Both factors contribute to a sporadic server that performs much better, both for average and worst-case. In fact, the worst-case performance is the same as in the full-featured runtime sporadic server (except for additional overhead) and the average case can be almost the same when the actual task execution time approaches the worst-case estimations.

To implement this enhanced application-level sporadic server in the environment of a multitask or multi-thread preemptive runtime executive or operating system (O.S.), the server must have access to a mechanism for dynamically changing the priorities of the tasks executing in the system. Most of the usual real-time kernels provide this functionality (threads extensions to POSIX [16], VRTX [17], Orkid [18], etc.). Therefore, the proposed sporadic server can be implemented under these kernels; in this paper we provide the guidelines for such implementation under the current Threads Extensions to the POSIX standard.

The Ada implementation has a particular difficulty. The language explicitly specifies a priority system that is static.[1] However, the Ada language reference manual [21] specifies that if all tasks in a program have no assigned priority, then the runtime system is free to use any convenient algorithm for deciding which eligible task to run. Sha and Goodenough [1] have shown that priorities may be changed dynamically at runtime when there are no Ada priorities at all. In fact, an increasing number of vendors are providing mechanisms in their runtime systems for dynamically changing the priorities of the tasks. The “Catalogue of Interface and Features and Options” (CIFO) [19] from the Ada Runtime Environment Working Group (ARTEWG), which is being implemented by most Ada vendors, includes this functionality. Furthermore, the Ada 9X Requirements Document [20] includes user-controlled scheduling under its require-

[1] Except for a limited change of priority during the rendezvous operation

ments and, although it does not explicitly mention dynamic priorities, this feature is very likely to appear, or at least not to be precluded, in the new standard.

Consequently, the dynamic priorities functionality will become very common in most Ada runtime systems, and is becoming more portable through the CIFO interface and, possibly, the new Ada 9X standard. Therefore the risk of using such a feature today is very low, and the enhanced application-level implementation of the sporadic server is a good choice when there is no runtime implementation available. In this paper, we also present the guidelines for implementing this algorithm in Ada.

2 The Sporadic Server Algorithm

2.1 The Runtime Sporadic Server

The sporadic server is a simple function providing control of event-driven processing. If a sequence of a particular type of events arrives faster than a program is ready to handle it, some mechanism must allow their processing to be controlled. This functionality can be achieved through the sporadic server, which simply limits the thread's maximum execution time in a given elapsed time window (i.e., the replenishment period). When this execution limit is exceeded, the thread's priority is temporarily lowered to a background level, allowing other time-critical tasks to be executed.

The sporadic server algorithm controls the scheduling of one or more tasks that service aperiodic events at a particular priority level; it has two important attributes that determine its behavior: the maximum *execution capacity* or *execution budget*, C_i, and the *replenishment period* or time interval used for replenishing the consumed execution capacity, T_i. The sporadic server algorithm preserves its execution budget complete until an aperiodic request occurs; at this point, an aperiodic task will be scheduled to run at its assigned priority level, consuming part or all of the available execution capacity. When all the available execution time has been consumed, the aperiodic task is not permitted to continue running at this priority level, although it can be assigned a lower background priority. The sporadic server replenishes each portion of consumed server execution capacity at some time RT_i, after it has been consumed. At time RT_i that portion of execution time becomes available again for consumption. Each portion of execution capacity that is replenished must be tagged with its replenishment time because it is not allowed to be replenished until at least one period later. The replenishment of the execution capacity consumed by the aperiodic task execution is determined according to the sporadic server replenishment policy, described next.

The following terms are used to explain the method of replenishing server execution time in the sporadic server algorithm:

Active	A priority level, P_i, is considered to be *active* if the priority at which the system is currently executing is equal to or greater than the priority P_i.
Idle	A priority level P_i is considered *idle* if the priority at which the system is currently executing is less than the priority of P_i.

Recall that the sporadic server's capacity is comprised of a number of replenished portions of execution capacity tagged with their availability times (initially, the sporadic server has all its execution capacity available). We sometimes refer to the available execution capacity as available *tickets*, in the remainder of this document. Assume a sporadic server that controls the execution of one or more aperiodic tasks with priority level P_i. When an aperiodic request arrives and one of the aperiodic tasks becomes eligible to run, it is permitted to consume the sporadic server's capacity from the oldest available portion until it completes execution or until that portion of execution capacity is exhausted. At that time, the amount of consumed execution capacity has to be scheduled to be replenished at some instant RT_i later; RT_i is set to the maximum of the time at which priority level P_i became active and the time at which that portion of execution capacity became available, plus the replenishment period of the sporadic server (see Figure 2). If the aperiodic task has not yet completed execution and there is more execution capacity currently available, it is permitted to consume more capacity in the same way. Each portion of consumed execution capacity generates a replenishment operation, but those replenishments scheduled to occur at the same time can be grouped together. Also, consumption and replenishment of very small portions of execution capacity can be grouped into the next available portion, because the overhead of considering such small portions can be greater than the actual execution capacity they represent.

Example 2. In the next example (Figure 2), we want to allocate 10 msec. with a replenishment period of 18 msec. for handling a particular type of events:

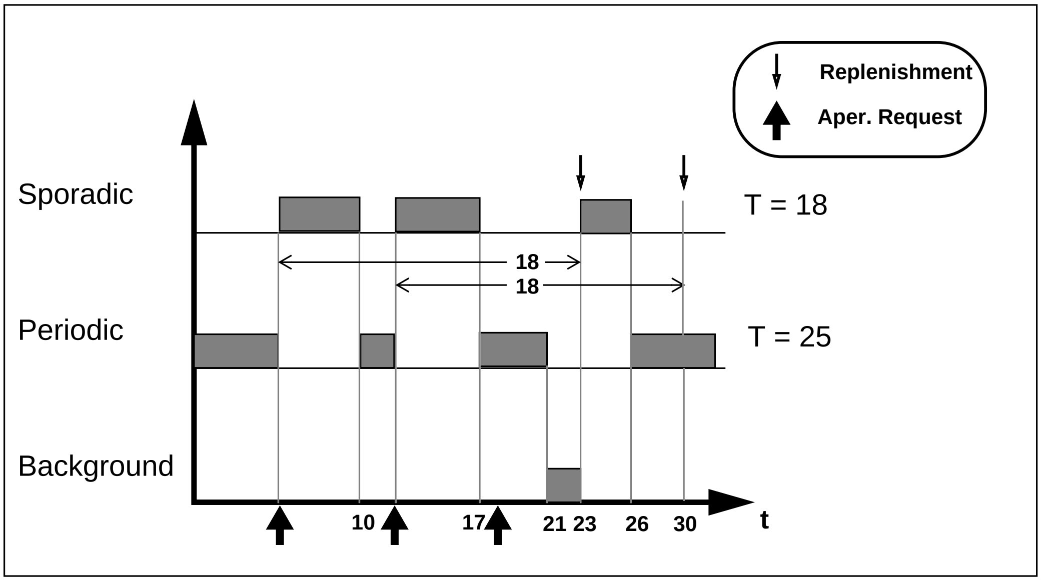


Figure 2 Example of a Sporadic Server-Controlled Task

In this example, each event takes 5 msec. to be serviced. The first two aperiodic requests arrive at times 5 and 12 msec. and are serviced immediately. After this, the execution budget is exhausted and when the next aperiodic request arrives at t=18 msec., the sporadic server is assigned a background priority. In this way, the periodic task is able to complete its execution on time. Additional execution budget for 5 msec. is replenished at times t=23 and t=30 msec. (one replenishment period after each request) respectively for the first two requests.

2.2 The Application-Level Sporadic Server

To make possible an application-level implementation of the sporadic server, we have to take a look at the main reasons that make it necessary to implement the full-featured sporadic server in the runtime system:

1. *Measuring the actual execution time*. The sporadic server algorithm has to measure the actual execution time of an associated aperiodic task, in order to determine how much execution time has been consumed and also to defer further service when the execution budget is exhausted. This function cannot be implemented at the application-level, since a task cannot determine in advance when it will be preempted by higher priority tasks and for how long.

2. *Tracking of Priority Level.* The replenishment policy of the sporadic server makes it possible to establish the replenishment origin (the time instant which, when added to the replenishment period, gives the replenishment time) before the instant at which the aperiodic event arrives. The algorithm specifies the replenishment origin to be set to the maximum of the instant at which the priority level of the sporadic server became active, and the instant at which the portion of consumed execution capacity became available. Tracking of the priority level can only be done at the runtime system level, where there is knowledge of the time instants at which each task is ready to execute (recall that the activation of priority level P_i is defined as the earliest instant from which activity of priority greater than or equal to P_i is being continuously executed).

   Figure 3 shows the replenishment algorithm applied to a set of tasks. The highest priority task is a periodic task, τ_1, with period T=6 and execution time C=3. The next priority level corresponds to an aperiodic task under the control of a sporadic server with replenishment period T=8 and execution budget C=2. Lower priority tasks execute whenever the sporadic server priority level is idle. There are three aperiodic requests arriving, each with a computation requirement of 2 units of time; their arrival times and replenishment origins are:

   - Arrival time = 1; the replenishment origin is set to t=0, when the sporadic server priority level became active. The replenishment time will be t=8.

   - Arrival time = 8.5; this time the replenishment origin is set to t=8. Although the priority level was activated at t=6, the execution capacity became available from the previous replenishment only at t=8. The replenishment time will be t=16.

   - Arrival time = 13; the replenishment origin is set to t=16, because until that time there is no execution capacity available. The replenishment time will be t=24.

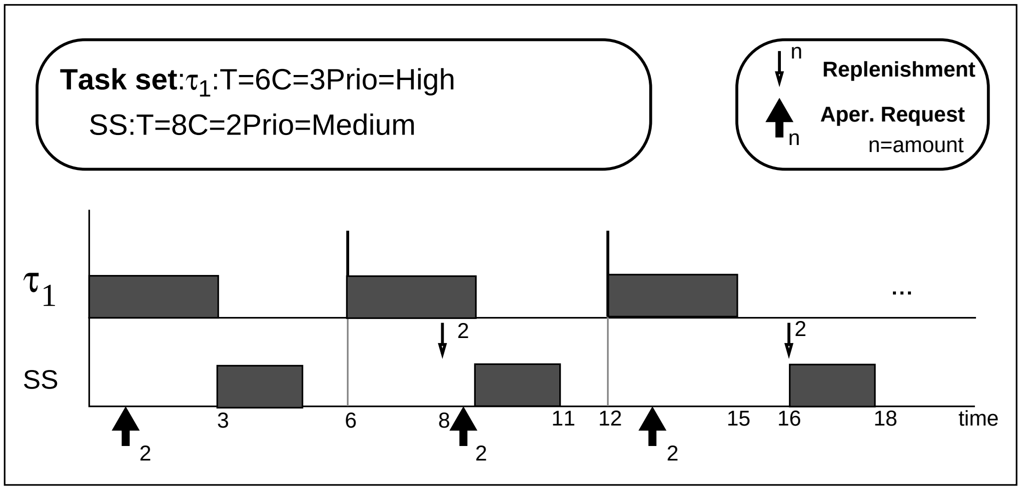


Figure 3 Priority Level Replenishment Policy

Intuitively, the reason for this early replenishment is that if the request had arrived earlier, for example at the activation of the priority level in the first request, the execution pattern would have been the same, provided that there was available execution capacity at this point. Notice that with this earlier request the aperiodic task could not have started to execute before, due to preemption by higher priority tasks. We can take advantage of this fact and make the replenishment earlier because in this way we will have execution capacity available earlier for the next aperiodic requests. In Figure 4 we can see that the execution patterns for aperiodic requests which arrive exactly at the replenishment origins of the first two requests from Figure 3, are exactly the same (the third request was before its replenishment origin, so there is no advantage to moving it back). Moreover, the behavior is also the same as that of an equivalent periodic task with period equal to the replenishment time and execution time equal to the execution requirement of the aperiodic requests.

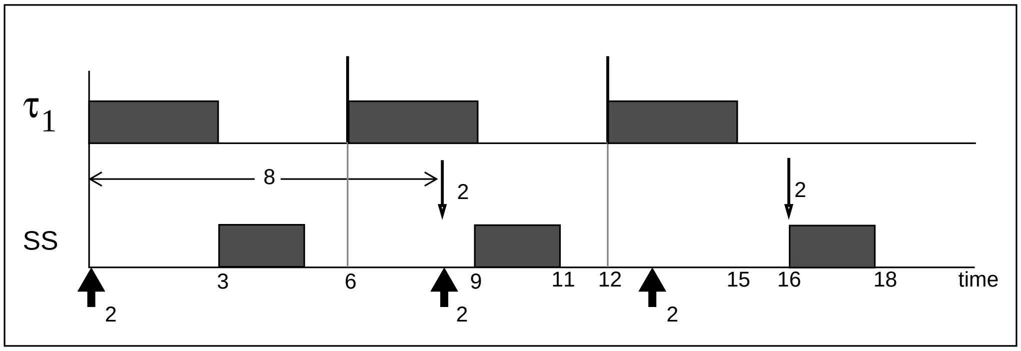


Figure 4 Replenishments of Earlier Requests

3. *Background processing*. The sporadic server task alternates between normal and background priority, depending one the requests and the availability of execution time. The runtime system must be called to change the task priorities and reschedule the CPU. Some systems provide such calls, so they can be issued from an application-level task, but other systems, like standard Ada, do not provide this mechanism.

Sprunt and Sha [4] developed an application-level sporadic server implemented with an Ada task, in which each of these difficulties was solved by making the following restrictions:

1. *Measurement of execution time*. Instead of the actual execution time, the worst-case execution time of each request is used for budget consumption and replenishment. In this way, there is no need to measure the execution time. With this restriction there is no loss of worst-case performance; only the average case performance is decreased, since it is not possible to take advantage of an actual execution time smaller than the worst-case. Under this restriction, a sporadic server is not allowed to service an aperiodic request unless it has available execution time greater than or equal to the worst-case execution time of that request. This restriction is imposed because there is no measurement of the execution time, so there is no way to stop a task at the instant at which its budget becomes exhausted.

2. *Replenishment Policy*. Since it is not possible to keep track of the active/idle status of the priority levels in the system (which can only be done by the scheduler), a simpler replenishment policy is used: the consumed execution time is replenished one replenishment period after the sporadic service is initiated. This decreases the performance of the sporadic server because it may delay the time at which the sporadic service is initiated from the request arrival, and certainly from the instant at which the priority level was active. Figure 5 shows the behavior of this replenishment policy for the same example that appeared in Figure 3. It can be seen how this replenishment policy introduces more latency into the processing of aperiodic events.

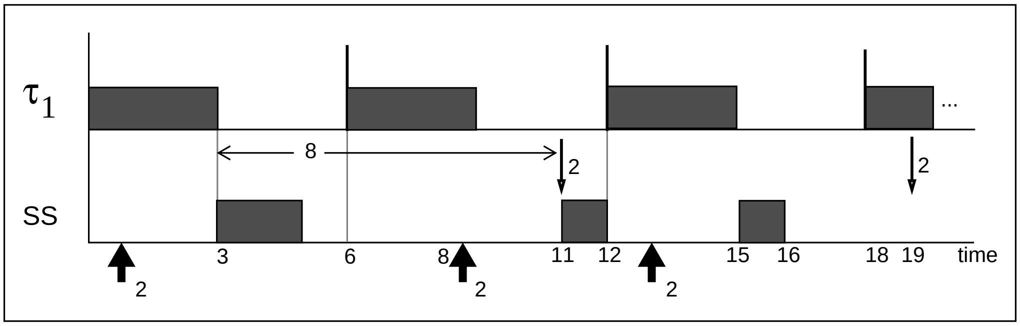


Figure 5 Service Initiation Replenishment Policy

3. *Background processing*. Since standard Ada does not provide any means of dynamically changing the priority of a task (except in a limited manner during a rendezvous operation), the task must always operate at its assigned priority; therefore, it is not possible to take advantage of background execution.

This may drastically reduce the average case performance of the sporadic server, but not its worst case.

In this paper, a new application-level sporadic server algorithm will be introduced, which takes advantage of the dynamic priorities mechanism available in most real-time kernels and in many Ada runtime systems. The restrictions that apply under this new sporadic server are:

1. *Measurement of execution time*. The execution time cannot be measured, so the approach taken in [4] is also applied. This means that the worst-case execution time is always assumed, and the aperiodic task cannot be initiated unless there is enough available execution time to guarantee its completion under the worst case.

2. *Replenishment policy*. An intermediate replenishment policy is used, in which it is not necessary to keep track of the priority levels; it has the same worst-case performance as the best replenishment policy, used under the runtime sporadic server. In the new replenishment policy, if an aperiodic event arrives at time *t* and an execution budget quantum of *Q* is available to service the aperiodic task at the sporadic server's priority, this budget quantum has to be replenished at time *t+T*, where *T* is the replenishment period. Figure 6 shows the behavior of this replenishment policy when applied to the example of Figure 3, where the priority level policy was used. The replenishment time in the runtime sporadic server can be set earlier than *t+T* [4], depending on the priority-level activation instant, but the worst-case performance of the sporadic server is the same under both policies. On the other hand, under the replenishment policy used in the application-level sporadic server in [4], it is necessary to wait until the sporadic server service is initiated and then set the replenishment time a period later; this may increase the worst-case response time of the sporadic server by as much as 100%.

3. *Background processing*. The dynamic priorities mechanism is used, and, therefore, this application-level sporadic server can change the aperiodic task's priority from normal to background, and back, whenever necessary. This improves the average case performance of the sporadic server over the previous application-level implementation.

In order to implement the replenishment policy, the sporadic server is able to compute the replenishment origin only if the server is activated *precisely* at the instant of the arrival of the aperiodic request. This is accomplished by the use of the dynamic priorities mechanism. The aperiodic task is forced to wait for the aperiodic event at the highest priority in the system. Since the task is waiting, this imposes no burden on other tasks (except for a slight overhead that may be computed as blocking time for the rest of the tasks). As soon as the aperiodic request arrives, the aperiodic task is activated, the replenishment is scheduled, and the task's priority

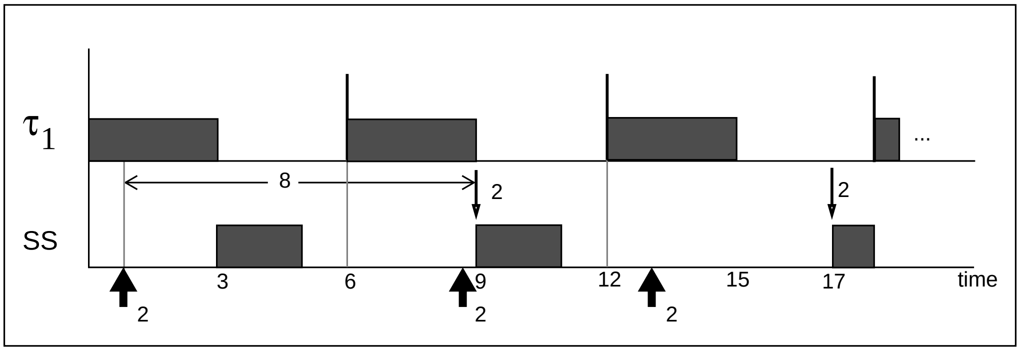


Figure 6   Request Arrival Replenishment Policy

is immediately lowered to its normal priority level if sufficient execution capacity is available; otherwise, it is lowered to background priority.

The restrictions have an associated cost:

- *Decreased average case performance*. The average case performance is decreased over the runtime sporadic server because the replenishment policy is less favorable and because the worst-case execution time is always assumed, even when the actual execution time may be smaller.
- *Some sporadic server applications are not supported*. Since the actual execution time is not measured, applications of the sporadic server that rely on this feature, such as implementing the period transformation technique, or debugging the timing requirements [4], are not supported. In any case, the rest of the applications, such as preserving soft and hard deadlines, scheduling producer/consumer tasks, eliminating the effects of deferred execution or message transmission, etc., are fully supported.
- *There is more overhead for sporadic server tasks*. The application-level implementation requires more context switches and, therefore, incurs more overhead for sporadic server tasks.

Despite this associated cost, this implementation provides a highly efficient version of the sporadic server, with worst-case performance similar to that of the runtime sporadic server. Its main advantages are:

- *Availability and simplicity.* Usually the application developer cannot modify the runtime kernel.The application-level sporadic server can be used right now if no sporadic server implementation is available. It is also very simple to implement.
- *Worst-case performance*. Except for the effects of additional context switches, the worst-case response time is the same as for the runtime sporadic server.

- *Compatibility*. Runtime and application-level sporadic servers may be implemented using exactly the same interface. In this way, the application code will not need to be changed, even if the actual implementation chosen supports the sporadic server in the runtime.
- *Lack of overhead when no sporadic server is used*. The runtime sporadic server introduces a small overhead for every task, even if no sporadic servers are used. At each scheduling point, the runtime system must at least check if sporadic servers are being used. In the application-level sporadic server, if a particular application does not want to use sporadic servers, this overhead disappears.

2.3 Schedulability of the Application-Level Sporadic Server

Sprunt, Sha, and Lehoczky proved in [3] that a sporadic server can be treated as a standard periodic task with period and execution time equal to the sporadic server replenishment period and execution budget, respectively. In their proof, they use the sporadic server algorithm that corresponds to the runtime sporadic server. This algorithm does the replenishments one period after the activation of the sporadic server's priority level. We will call this replenishment policy the PL (Priority Level) Policy.

The same result applies to the proposed application-level sporadic server, which replenishes one period after the arrival of the aperiodic request (AR Policy). Under this policy the replenishments are always done at the same time or after the replenishments under the PL policy. This means that under the AR policy the execution of the next request is never started before it would have been started under the PL replenishment policy. This execution pattern can only introduce a smaller amount of preemption for lower priority tasks and, therefore, has no detrimental schedulability effect. Consequently, the new replenishment policy has no effects on the schedulability of the rest of the tasks.

The new replenishment policy shows the same worst-case response as the runtime sporadic server, for aperiodic tasks with hard deadlines having a minimum interarrival time**.** If an aperiodic request arrives when the execution budget is fully available, there will be enough execution time for the request to be fully serviced; the replenishment policy will not influence the execution pattern in this case. Under the AR policy, the consumed execution time will be replenished one replenishment period after the arrival of the aperiodic request. If the PL policy is being used, the worst-case replenishment interval will occur when the priority level becomes active, just when the request arrives. Consequently, the worst-case replenishment interval will be the same for both policies. The next request will not arrive before the specified minimum interarrival time has elapsed, that is, when all the consumed execution time has been replenished and the execution budget is again complete.

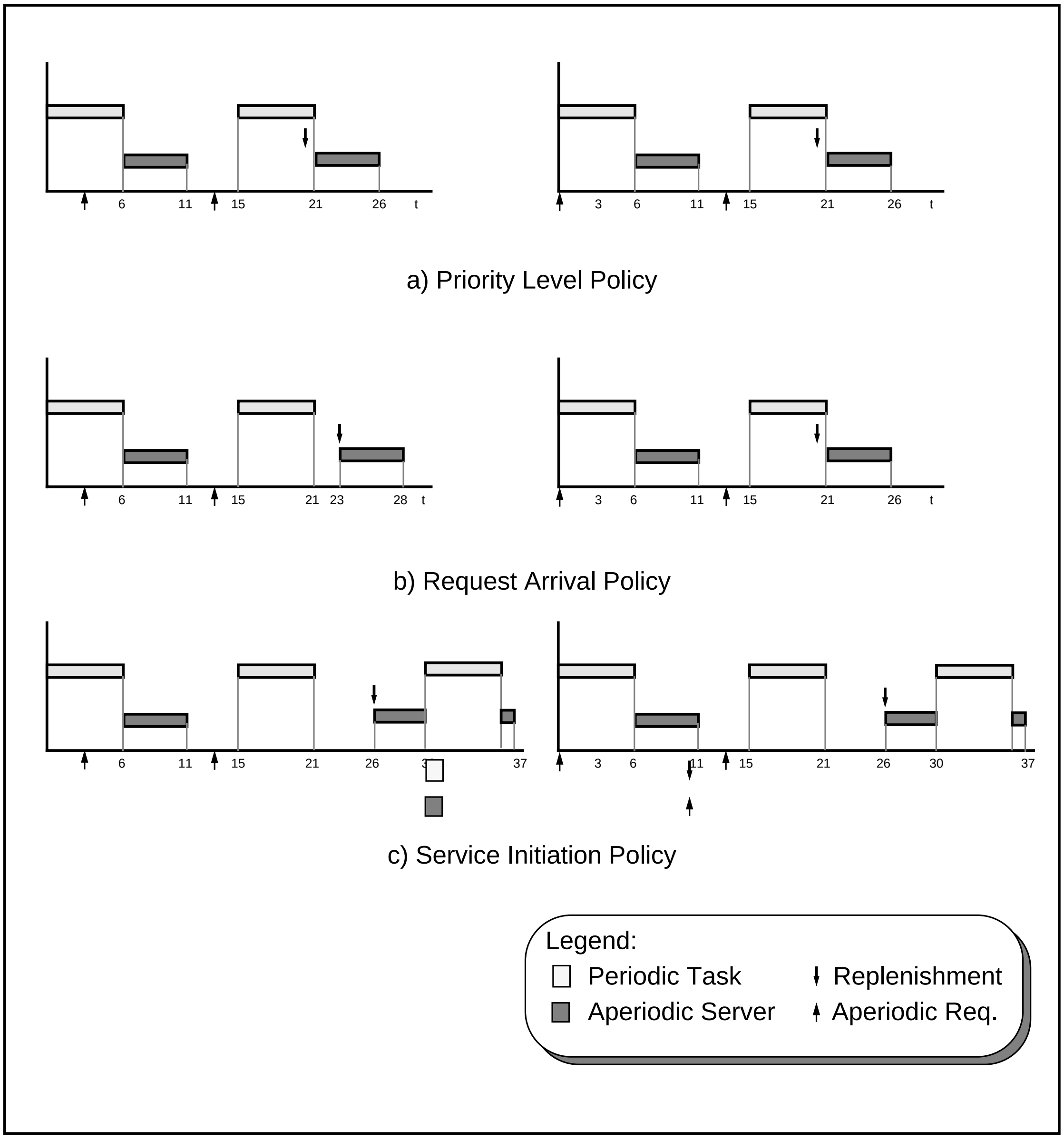
Consequently, the application-level sporadic server has no detrimental effects on the schedulability of lower priority tasks (i.e., on their ability to meet their deadlines) and, although the server has an average case response time that is worse than for the runtime sporadic server, the worst-case response for hard deadline aperiodic tasks is the same for both. The next example shows a comparison between the three sporadic server implementations that have been discussed.

Example 3. In the next example (Figure 7), a comparison is made between the three replenishment policies that have been considered. In all cases there is a sporadic server with period T=20 and budget C=5. There is also a higher priority periodic task, with period T=15 and execution time C=6. No background processing is available (it is supposed to be consumed by other low priority tasks). All examples are shown for the same set of two different request patterns.

Figure 7(a) shows the behavior when the priority level policy is used. The first request arrives at t=3 and t=0 respectively, for both cases shown. In both cases the replenishment of the consumed tickets is scheduled to occur at t=20, because the priority level was active since t=0. The second request is finished at t=26.

When the request arrival policy is used (Figure 7[b]) the replenishment time for the case where the first request arrives at t=0 is still t=20. However, when the first request arrives at t=3, the associated replenishment occurs at t=23, thus increasing the latency for the second request.

Finally, Figure 7(c) shows the behavior for the service initiation policy. Under this policy service for the first request starts at t=6 in both cases; therefore, the replenishment occurs at t=26 and this extends the finalization time for the second request to t=37. This policy leads to the largest latencies for aperiodic tasks.



a) Priority Level Policy

b) Request Arrival Policy

c) Service Initiation Policy

Figure 7 Comparison Between the Different Replenishment Policies

3 Ada Task Implementation

3.1 Interface and Use of the Sporadic Server Package

The Ada implementation of the proposed sporadic server is contained in a package with the following specification:

Table 1 Sporadic Server Specification

```
with TASK_IDs;             -- Used in private part
with DYNAMIC_PRIORITIES;

package Sporadic_Server is

      type parameters is limited private;

      procedure initialize
               (This_Sporadic_Server : out parameters;
                Period, Budget       : in DURATION;
                Normal_Priority,
                Background_Priority  : in DYNAMIC_PRIORITIES.PRIORITY);

      procedure detach (This_Sporadic_Server: in out parameters);

      procedure arm ( This_Sporadic_Server : in parameters;
                      Using_High_Priority  : in BOOLEAN := TRUE);

      procedure request (This_Sporadic_Server  : in parameters;
                         With_Request_Duration : in DURATION;
                         With_Request_Time     : in CALENDAR.TIME
                                                   := CALENDAR.CLOCK);

      procedure finish;

      Request_Duration_Error  : exception;
      Not_Enough_SS_Resources : exception;
      Invalid_Argument        : exception;

private

      MAX_NUM_OF_REPLENISHMENTS : constant INTEGER
                                  :=implementation_defined;
      type parameters is implementation defined;
      ...

end Sporadic_Server;
```

This specification provides the user with a standard interface to the sporadic server mechanism, which can be used independently of the kind of implementation (runtime or application-level) used.

The functionality of the interface procedures is the following:

1. *INITIALIZE*. This procedure allows the user to declare a spore1fadic server with the desired attributes, such as period, budget, normal (assigned) priority level, and background priority level. The procedure will return a variable with the sporadic server's parameters (the sporadic server control block), which must be provided in the other calls to the sporadic server procedures. Usually, this procedure is called once by each aperiodic task that wishes to execute under the control of a sporadic server.

2. *DETACH*. This call is made when an aperiodic task no longer needs to continue under the control of a sporadic server (for example, when it finishes). It deallocates the storage and pending operations of the sporadic server specified. This call can be used followed by one to *initialize* to change the attributes of a particular sporadic server.

3. *ARM*. This call must be issued just before waiting for an aperiodic request, in order to prepare the sporadic server specified by *This_Sporadic_Server* to receive this request. The argument *Using_High_Priority* is a boolean input argument that, when set true, forces the sporadic server to wait at a high priority level, to ensure that when the aperiodic request arrives, the aperiodic task is activated immediately, and so, the replenishment is scheduled at the right time. When the aperiodic request is processed by an Interrupt Service Routine (ISR) instead, this high priority waiting is not necessary and this argument should be set to false.

4. *REQUEST*. This call must be issued immediately after the arrival of the aperiodic request. The user must provide the request duration and request time arguments:

   > *With_Request_Duration*: This is the worst-case execution time of the aperiodic task for the kind of request that has arrived. Different request durations are allowed and can be determined by the application according to the type of aperiodic request that has arrived.
   >
   > *With_Request_Time*: The arrival time of the request.

   Depending on the available sporadic server execution time, the call will determine the availability of execution capacity, schedule the replenishment, and drop the aperiodic task's priority to its normal or background priority.

5. *FINISH*. This call tells the sporadic server manager that no more sporadic server functions will be requested from now on, thus enabling it to deallocate any storage space held, and terminate any internal task.

The typical application code for an aperiodic task under the control of a sporadic server is shown in Table 2.

Another possibility for the application task is that the aperiodic request may be serviced by an interrupt service routine, which in turn signals the aperiodic task through a semaphore or an asynchronous message (mailbox mechanism). In this case, the high-priority processing of the aperiodic request is guaranteed by the priority of the hardware interrupt (higher than all other software priorities, in Ada). The user code, then, will be slightly different (Table 3). In the call to *arm* it is possible to set the argument *Using_High_Priority* to false, provided that the ISR reads and stores the time at which the request was made. In this way, the sporadic server's overhead is reduced, because some context switches are avoided.

Table 2 Application Aperiodic Task

```
task body Aperiodic is

       ...
       This_Sporadic_Server : Sporadic_Server.parameters;

begin
       User initialization;
       Sporadic_Server.initialize(This_Sporadic_Server, Period,
                           Budget, Normal_Priority, Backgr_Priority);

       loop
              Sporadic_Server.arm(This_Sporadic_Server);
              Wait for the aperiodic event;
              Sporadic_Server.request
                           (This_Sporadic_Server, Request_Duration);
              Process aperiodic event;
       end loop;

end Aperiodic;
```

3.2 Implementation of the Sporadic Server

3.2.1 External Requirements

To implement the proposed application-level sporadic server, it is necessary to have access to a mechanism for dynamically changing the task priorities. To change the task priorities, it is also necessary to have a mechanism for identifying tasks. Both mechanisms appear in the CIFO Release 2.0 [19], and their interface is shown in Table 4. If the available dynamic priorities mechanism does not conform to the specification of these packages, it is still possible to map them through this interface. The sporadic server's code need not be changed.

Also, an error handler (Table 5) must be supplied for handling unexpected fatal errors that may occur inside the sporadic server implementation (similar to a panic error that would be generated by a runtime implementation). This handler and the dynamic priorities package make the sporadic server code completely independent from the environment in which it is executed.

Table 3 Application Aperiodic Task With ISR

```
procedure ISR is    -- Called by hardware interrupt.

begin
       Read clock into Request_Time, and process interrupt;
       Signal semaphore or mailbox, storing Request_Time;
end ISR;

task body Aperiodic is

       ...
This_Sporadic_Server : Sporadic_Server.parameters;

begin
       User initialization;
       Sporadic_Server.initialize( This_Sporadic_Server, Period, Budget,
                                   Normal_Priority, Backgr_Priority);
       loop
             Sporadic_Server.arm( This_Sporadic_Server,
                                   Using_high_Priority=>FALSE);
             wait for semaphore or mailbox and read Request_Time;
             Sporadic_Server.request( This_Sporadic_Server,
                                      Request_Duration, Request_Time);
             Process aperiodic event;
       end loop;
end Aperiodic;
```

Table 4 Dynamic Priority Utilities

```
package TASK_IDs is

       type TASK_ID is private;
       function SELF return TASK_ID;
private
       ...
end TASK_IDs;

with TASK_IDs;
package DYNAMIC_PRIORITIES is

       type PRIORITY is range implementation-defined;

       procedure SET_PRIORITY ( OF_TASK : in TASK_IDS.TASK_ID;
                                TO      : in PRIORITY);
       function PRIORITY_OF (THE_TASK : in TASK_IDS.TASK_ID)
                                return PRIORITY;
end DYNAMIC_PRIORITIES;
```

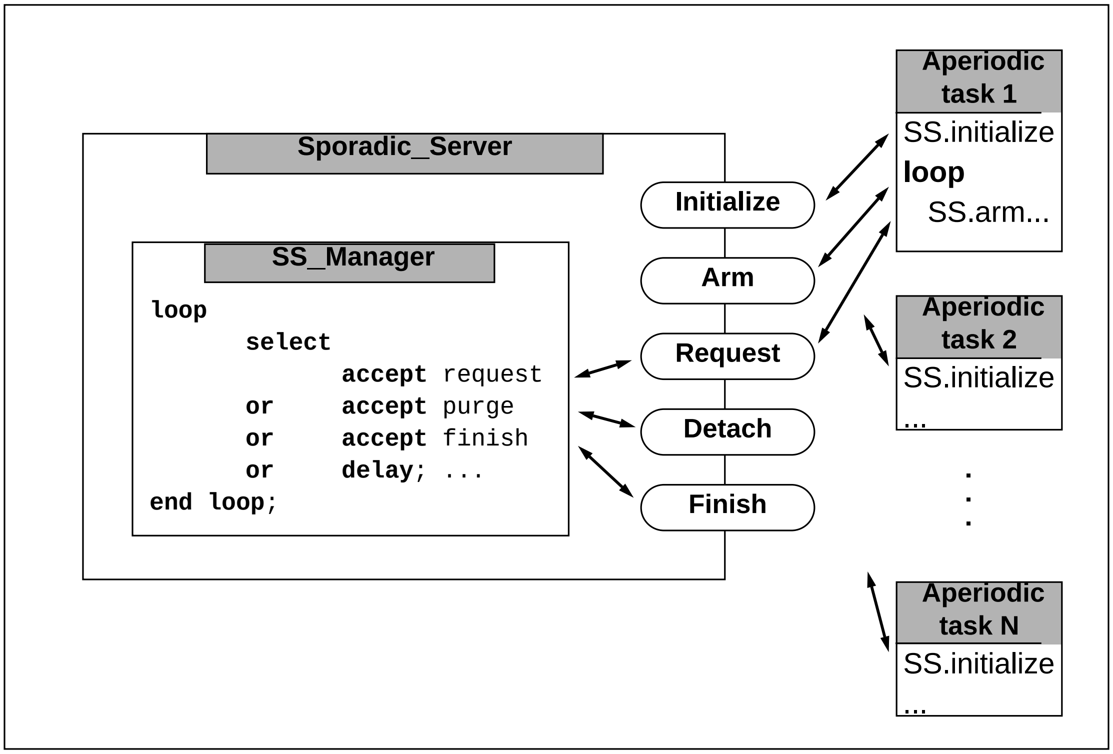
Table 5 Error Handler Specification

```
package Handler is

        procedure error (s : in STRING);

end Handler;
```

3.2.2 General Structure

The body of the sporadic server package includes the code for each of the procedures of the interface, which are very simple. The body also includes the sporadic server manager task, in which the sporadic server algorithm is implemented. The general structure of this package can be seen in Figure 8.

The sporadic server manager task is the only task allowed to modify the contents of the sporadic server control blocks; it keeps track of all the available execution capacity and replenishments of all the aperiodic tasks controlled by sporadic servers that are present in the system. Only one sporadic server manager task is used for all the sporadic servers in the system. The



```
loop
        select
                accept request
        or      accept purge
        or      accept finish
        or      delay; ...
end loop;
```

Figure 8 General Structure of the Sporadic Server Package

main data structures of the sporadic server manager are the sporadic server parameters or control blocks, and the replenishment queue:

- There is one sporadic server control block per sporadic server, and it is created during the initialization (through *initialize*). It stores all the attributes of the sporadic server, such as period, budget, available tickets, priority levels, etc.
- The replenishment queue is a single priority queue; the priority is the replenishment time and the order of extraction is the earliest replenishment time. The replenishment queue stores all the replenishment operations that are pending at each instant.

The basic structure of the sporadic server manager is an endless loop with a timed wait (selective wait with delay alternative). When an aperiodic request arrives, the task's entries are called by the aperiodic tasks through the interface procedures *request*, *detach*, and *finish*. The *delay* alternative is set to the earliest replenishment to be performed (except when the replenishment queue is empty). The sporadic server manager task will be described with more detail in Section 3.2.4

3.2.3 Interface Procedures

The interface procedures provide an implementation-independent way for the application to make use of all the sporadic server functions. Implementing these procedures is very simple, because the main part of the sporadic server algorithm is performed by the sporadic server manager task.

1. *INITIALIZE.* This procedure creates a sporadic server parameter variable, or control block, and initializes it with the attributes provided through the procedure arguments. The sporadic server is initialized with its complete execution budget. An identification of the calling task is stored in the control block, so its priority can be changed later on. Also, its priority is set to its normal level. The procedure returns an implementation-dependent pointer to the sporadic server control block (access type for dynamic storage, index to an array element for static storage, etc.). This pointer will be used further by the aperiodic tasks when calling other sporadic server procedures.
2. *DETACH*. In order to eliminate all references to the specified control block stored in the replenishment queue, this procedure makes a call to the sporadic server manager task. After ensuring that this sporadic server control block will no longer be used by the manager task, it deallocates its storage space.
3. *ARM*. This procedure changes the priority of the calling task, to prepare it for receiving an aperiodic request. If the task itself is receiving the aperiodic request it must wait at the highest priority in the system, in order to correctly implement the desired replenishment policy (a period after the arrival of the request). Otherwise, if the aperiodic request is handled by a high-priority

Table 6 Interface Procedures

```
package body Sporadic_Server is

       task Sporadic_Server_Manager is
             entry request ( ss_cb           : in parameters;
                             request_duration: in DURATION;
                             request_time    : in CALENDAR.TIME);
             entry purge (ss_cb : in parameters);
             entry finish;
       end Sporadic_Server_Manager;

       procedure initialize (This_Sporadic_Server, Period, Budget,
                           Normal_Priority, Background_Priority) is
       begin
             Create and initialize This_Sporadic_Server;
       end initialize;

       procedure detach (This_Sporadic_Server) is
       begin
             Sporadic_Server_Manager.purge (This_Sporadic_Server)
             Deallocate This_Sporadic_Server;
       end detach;

       procedure arm (This_Sporadic_Server,Using_High_Priority) is

       begin
             Set task’s priority to high or normal,
             according to Using_High_Priority;
       end arm;

       procedure request ( This_Sporadic_Server, With_Request_Duration,
                            With_Request_Time) is

       begin
             Sporadic_Server_Manager.request
                 (This_Sporadic_Server, Request_Duration, Request_Time);
       end request;

       procedure finish is

       begin
             Sporadic_Server_Manager.finish;
       end finish;
       ...
```

interrupt service routine, the argument *Using_High_Priority* can be set to false; in this case, the calling task will be set to its assigned priority level.

4. *REQUEST*. This procedure is a just a gateway to the entry call *request* at the sporadic server manager task.
5. *FINISH*. This procedure makes a call to the sporadic server manager task, which then terminates itself. Another implementation of this procedure could be an abort statement for the sporadic server manager task. However, this is not recommended, because some compilers can optimize the tasking operations if there are no abort statements in the program.

The pseudocode of these procedures is shown in Table 6.

3.2.4 Sporadic Server Manager Task

The purpose of this task is to manage all the ticket consumption and replenishment operations of all sporadic servers in the system. Basically, its code is an endless loop in which a timed wait is performed until either an aperiodic request requires processing or a replenishment action is due. Each time the delay expires, the replenishment queue is checked to see it there are any replenishment actions pending.

The basic data structures managed by this task are the sporadic server parameter variables, or control blocks, and the replenishment queue:

1. There is a *sporadic server parameter variable,* or control block, for each aperiodic task that has issued a call to *initialize*. This control block is a record containing the attributes of the sporadic server. Its description is given in Table 7:

Table 7 Sporadic Server Parameters

```
type parameters is record
       period, budget,
               available_tickets,
               last_request_size       : DURATION;
       normal_pri, background_pri      : Dynamic_priorities.priority;
       my_task                         : Dynamic_priorities.task_id;
       last_replenishment              : CALENDAR.TIME;
end record;
```

- The *period, budget, normal_pri,* and *background_pri* are initial attributes set by the application when the parameter variable is created.
- *Available_tickets* is the amount of sporadic server execution time left, and is changed with each consumption or replenishment operation.
- *Last_request_size* allows the worst-case execution size of an aperiodic request to be stored. This size will then be used when a sporadic server task is awakened from the background state to the assigned priority state and a ticket consumption must be made.

- *My_task* is the identification of the aperiodic task used by the sporadic server manager task to change the task's priority.
- *Last_replenishment* stores the last time at which the sporadic server was replenished. This time is used to make sure that no replenishment is scheduled to occur before one period after the last replenishment has been carried out. This prevents too early replenishments that could arise when a request time less than the last replenishment is specified as an argument to *request*.

2. The *replenishment queue* is a priority queue in which the highest priority element is the one with the earliest replenishment time. It can be implemented with a standard generic package which must include at least the following functionality:

   Empty function, to determine if the queue is empty.
   Insert in priority order.
   Extract the highest priority element.
   Read the highest priority element.
   Delete an element from any position in the queue.

   Such a queue can be implemented through a heapform heap data structure [13], which is stored in a fixed size array (and hence does not need dynamic memory), and has the following characteristics, related to *n*, the number of elements present in the queue:

   Insertion: O(log n)
   Highest priority extraction: O(log n)
   Read highest priority: O(1)
   Deletion: O(n) (Change is Deletion + Insertion = O(n)).

   Under normal operation of the sporadic server, all operations on the queue are either O(1) or O(log n), thus very efficient. Only the detach operation uses *deletion* (O(n)), but this is a rather infrequent operation. Although this implementation of a priority queue does not preserve the order of arrival for elements of the same priority, this feature is not needed for the replenishment queue.

The sporadic server manager task has three entries. The most important entry is named *request*, and is called by aperiodic tasks (through the *request* procedure) each time an aperiodic request has to be processed. The second entry, named *purge* is only used when an aperiodic task no longer wants to operate under the control of its sporadic server, and issues a call to *detach*; in this case, all references to the corresponding sporadic server control block, which are in the form of pending replenishment operations, must be eliminated from the replenishment queue to avoid erroneous behavior when the replenishment time is due. The third entry, *finish,* is used to make the sporadic server manager task complete its execution.

The sporadic server manager task waits for these entries inside a *select* statement which also has a *delay* alternative. The *delay* alternative is set to the first replenishment time due, which is obtained by reading the highest priority element from the replenishment queue (unless it is empty, in which case the manager task only waits for one of the entries to be called). When this delay has expired, a call to the *replenish* procedure is issued to replenish all due tickets. The *select* statement is executed inside an endless loop. The basic structure of the sporadic server manager task is shown in Table 8. The *request* process and replenishment actions will be discussed with more detail in the next sections.

Table 8 Sporadic Server Manager Task

```
task body Sporadic_Server_Manager is

       replenishment_queue : Priority_queue; -- Priority is time
       procedure Replenish;
       procedure Consume_Tickets (...);

begin
    loop
       select
             accept request (...) do
                    if tickets available then
                           obtain Replenishment_Origin;
                           Consume_Tickets(Replenishment_Origin, ...);
                           set to normal priority;
                    else
                           set to background priority;
                    end if;
             end request;
       or
             accept purge (This_SS) do
                    eliminate This_SS from replenishment_queue;
             end purge;
       or
             accept finish;
             exit;
       or
             when replenishment_queue not empty =>
                    delay until next replenishment time;
                    Replenish;
       end select;
    end loop;
end Sporadic_Server_Manager;
```

3.2.5 Processing of an Aperiodic Request

Each time an aperiodic request needs to be processed, the aperiodic task makes a call to *request*; this call leads to the execution of the r*equest* accept statement. If enough execution tickets are available, the replenishment origin is obtained as the maximum of the *request_time* and the time at which this sporadic server was last replenished; then, the appropriate amount of execution capacity is consumed by calling the procedure *Consume_Tickets*, which performs the actions shown in Table 9. When tickets are consumed, a replenishment must be inserted in the replenishment queue for some time later. The replenishment time, according to the policy being used, is set one period after the supplied replenishment origin.

Table 9 Consumption of Execution Time

```
procedure Consume_tickets (This_ss, amount, Replenishment_Origin) is

begin
      decrease This_ss.available_tickets by amount;
      schedule replenishment at Replenishment_Origin + This_ss.period;
end Consume;
```

3.2.6 Replenishment Procedure

This procedure checks which of the replenishments found in the replenishment queue are due at the present time and performs a replenishment action for each of them. The replenishment action (Table 10) consists of extracting the first element from the replenishment queue and increasing the available sporadic server execution time by the amount that was consumed when the replenishment was scheduled. Once the *available_tickets* variable has been updated, it is necessary to check if the sporadic server is in the background state, with an aperiodic request pending or incomplete. If so, and if there are enough tickets, the sporadic server state must be switched back to normal priority so that the aperiodic task can proceed at its assigned priority level; also, the corresponding number of tickets must be consumed and a replenishment must be scheduled for the consumed tickets.

Table 10 Replenishment Action

```
procedure Replenish is

begin
      while tickets to replenish loop
            dequeue from replenishment_queue;
            increase available tickets;
            if task’s priority = background and then
             enough tickets are available then
                  Consume_tickets (...);
                  increase task’s priority to normal level;
            end if;
      end loop;
end Replenish;
```

3.3 Optimization of the Sporadic Server

The only external requirement for the sporadic server implementation previously presented is that it must have access to the mechanism for dynamically changing the task priorities. However, there are Ada implementations that provide optimized mechanisms for task synchronization. These implementations could be used to implement the sporadic server, significantly improving the performance by avoiding unnecessary context switches among tasks. Two of these optimizations are passive tasks and semaphores.

3.3.1 Passive Task Optimization

The passive task is a compiler optimization provided by some Ada vendors. A passive task behaves exactly like a normal Ada task but it may dramatically improve performance for all tasking operations. The basic idea of the passive task is that the compiler will not generate a "real" task for it, but rather a set of procedures whose execution is protected by a semaphore. In this way, a rendezvous with such a task is in practice implemented through a pair of semaphore lock and unlock operations and a procedure call. Semaphores can be implemented efficiently, for example by only requiring a kernel call when they are locked. There will be a context switch cost only when a calling task fails to lock the semaphore, and in most cases this will not be the case; therefore, the semaphore mechanism allows a very fast operation.

Of course, this optimization can only be applied to tasks exhibiting a special structure; in particular, no operations are allowed that require the existence of a real task, such as *delay* statements, entry calls, etc. The most suitable structure for this optimization is a task that is built for protecting access to a shared resource and executes only while rendezvoused with another task. This task is usually written as a *select* statement inside an endless loop, with different *accept entries*.

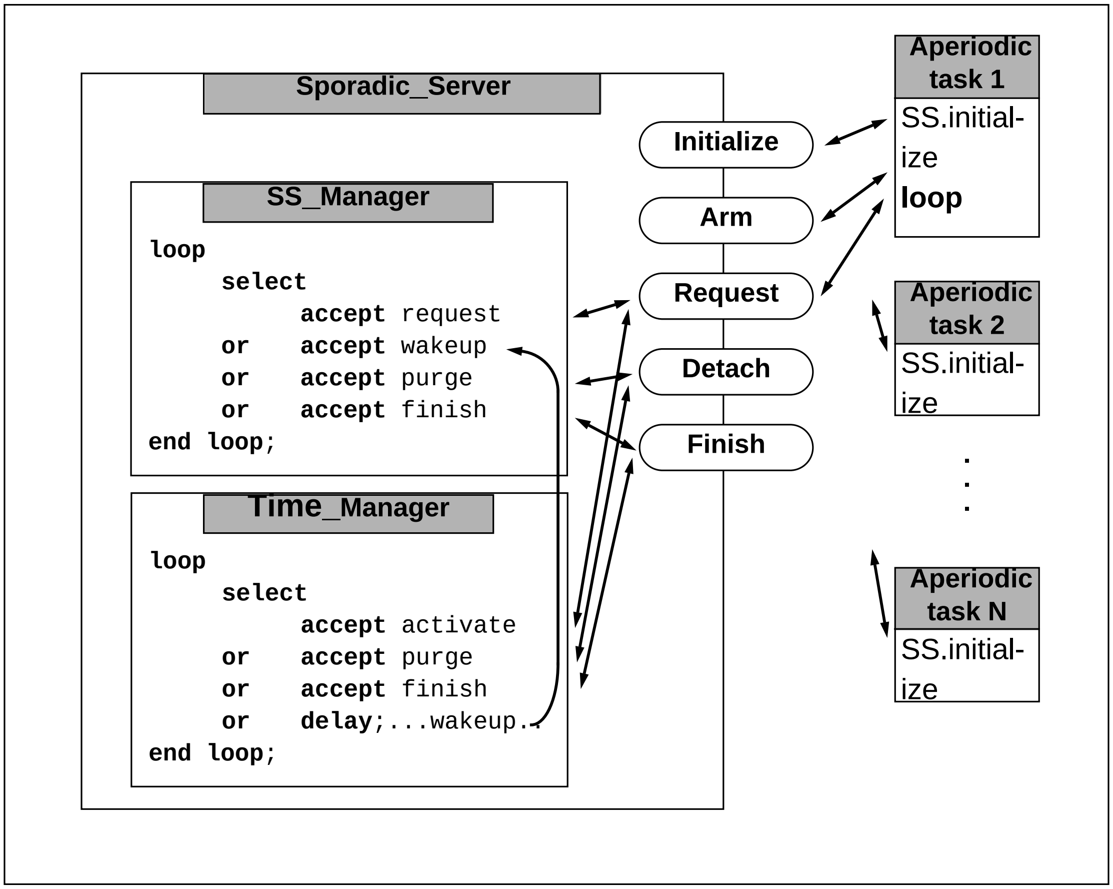
Unfortunately, the sporadic server manager structure that was presented in the previous section does not conform to this restriction. It does not always execute inside a rendezvous, as there is a *delay* alternative in its *select* statement that enables the task to execute on its own. Therefore, the passive task optimization cannot be done directly on this task, and each call to it must be a full Ada rendezvous, which is generally a high-cost synchronization primitive.

The only independent operation that this task has to perform is to wake up a sporadic server task that is executing at background priority, when new tickets become available. All the rest of the operations could be done inside the *accept* statements, while in rendezvous with other tasks (i.e., the aperiodic tasks calling the sporadic server utilities). In fact, if aperiodic events arrive at a low rate and there are tickets available almost always, the number of times that this wake-up operation must be performed, and consequently the number of unavoidable full context switches, is very small. Furthermore, for sporadic tasks with a minimum interarrival time and under the control of the appropriate sporadic server, no wake-up operations are needed: if the replenishment period is set to be less than or equal to the minimum interarrival time, there is guarantee that when a request arrives the execution capacity as already been replenished.

Consequently, the sporadic server manager task can be written as two separate tasks. The first task manages the ticket consumption and replenishment and is only executed while in rendezvous with other tasks, and therefore can be optimized or made passive. The second task is a non-optimizable task that keeps track of the time instants at which sporadic server tasks must be awakened, to be switched from background to assigned priority. Notice that this non-optimizable task needs to carry out only those replenishments that are capable of waking up a task. The rest of the replenishments are performed by the optimized task, not at the time when they are due, but at the time when they are needed: the instant at which the *request* entry is called by each sporadic server task.

The basic structure of the optimized sporadic server package is shown in Figure 9. There is a new task, called the *Time_Manager*, which is non-optimizable and keeps track of all the wake-up operations that need to be done. The sporadic server manager task has now one more entry, *wakeup*, which is called by the time manager task when a wake-up operation is needed.

The time manager task must have the highest application priority in the system. The basic data structure that it handles is the *wakeup_queue*, which stores the sporadic server tasks that must be awakened and the wake-up times. Just like the replenishment queue in the non-optimized sporadic server package, it is a priority queue.



```
loop
     select
          accept request
     or   accept wakeup
     or   accept purge
     or   accept finish
end loop;
```

```
loop
     select
          accept activate
     or   accept purge
     or   accept finish
     or   delay;...wakeup..
end loop;
```

Figure 9 Structure of the Optimized Sporadic Server Package

The code of the time manager task is basically a *select* statement inside an endless loop. The *select* statement waits for three entries, *activate_wakeup*, *purge,* and *finish,* and has a *delay* alternative. The delay expiration is set to the time at which the next *wakeup* operation must be performed, which is obtained by reading the top of the wake-up queue (if it is not empty). The *activate_wakeup* accept statement inserts the wake-up operation into the queue. The *purge* accept statement has to delete any references to the specified sporadic server control block. The entry call named *finish* is used to terminate the task. The pseudocode of the time manager task is given in Table 11.

Table 11 Time Manager Task

```
task body Time_Manager is

    wakeup_queue : priority queue; -- Priority is time;

begin
    set highest priority;
    loop
        select
            accept activate_wakeup (time_due, This_SS) do
                enqueue (time_due, This_SS) into wakeup_queue;
            end activate_wakeup;
        or
            accept purge(This_SS) ...
        or
            accept finish;
            exit;
        or
            when wakeup_queue not empty =>
                delay until next wake up time;
                dequeue from wakeup_queue;
                Sporadic_Server_Manager.wakeup;
        end select;
    end loop;
end Time_Manager;
```

The time manager is called by the *request* interface procedure (Table 12), which executes under the environment of the aperiodic task issuing the call. It is only called when the sporadic server manager has found that there are not enough tickets to process the request. The wake-up time is supplied by the sporadic server manager. The rest of the interface procedures are basically the same as in the non-optimized sporadic server.

The sporadic server manager task (Table 13) shows a different structure than the non-optimized version. Instead of using a general replenishment queue, a local replenishment queue is stored in each sporadic server control block to keep track of the replenishments of that particular sporadic server. The reason for this is that now tickets are only replenished when they

Table 12 Request Procedure in the Optimized Sporadic Server

```
procedure request(This_Sporadic_Server,Request_Duration,Request_Time) is

       new_priority  : DYNAMIC_PRIORITIES.PRIORITY;
       wakeup_time   : CALENDAR.TIME;

begin
       Sporadic_Server_Manager.request ( This_Sporadic_server,
                                         request_duration, request_time,
                                         new_priority, wakeup_time);
       if new_priority = background then
              Time_Manager.activate_wakeup ( wakeup_time,
                                             This_Sporadic_Server);
       end if;
       Drop task’s priority to new_priority;
end request;
```

are needed, i.e., when the aperiodic task makes a call to *request* or when a *wakeup* operation must be performed. In these two circumstances, the particular sporadic server which has to be replenished is supplied in the call, so only those due tickets that are needed by the particular sporadic server making the request or involved in the *wakeup* operation must be replenished. Because the replenishment period of the sporadic server does not change and all the replenishments are scheduled in order, the local replenishment queues need not be priority queues, just normal FIFO queues.

When the *request* accept statement is executed in a particular sporadic server, if there are no available tickets and the sporadic server is switched to background priority, the wake-up time must be calculated and returned to the *request* procedure, which in turn will call the *Time_Manager* to schedule the wake-up operation. The wake-up time is obtained from the local replenishment queue, by reading its elements, starting from the earliest replenishments, and checking when there will be enough tickets available.

The replenishment procedure is almost the same as in the non-optimized sporadic server, except that now it only applies to a particular sporadic server.

3.3.2 Semaphore Optimization

There are some Ada compilers that do not offer a mechanism like the passive task optimization, but do offer other high-speed synchronization primitives such as semaphores. In this case, the optimized sporadic server can be ported to this paradigm by just changing the optimized sporadic server manager task to a set of procedures guarded by a semaphore.

Each of the *accept* statements in the optimized sporadic server manager task must be converted to a procedure, and instructions for locking the sporadic server semaphore at the be-

Table 13 Optimizable Sporadic Server Manager

```
task body Sporadic_Server_Manager is

       local_replenishment_queue : normal_queue in each ss ctrl block;

begin
    loop
       select
             accept request ( ss_cb, request_duration, request_time,
                              new_priority, wakeup_time) do
                    replenish (ss_cb);
                    process the request and obtain
                           new_priority and wakeup_time;
             end request;
       or
             accept wakeup (ss_cb) do
                    replenish (ss_cb);
             end wakeup;
       or
             accept purge (...) ...

       end select;
    end loop;
end Sporadic_Server_Manager;
```

ginning of each procedure and unlocking it at the end, must be inserted. In this way, a common synchronization paradigm such as the semaphore can be used to achieve the same optimized behavior of the passive task.

3.3.3 Static Memory Allocation

In the previous implementations, the sporadic server control blocks have been allocated dynamically (with the Ada construction *new*), and a pointer of *access* type has been used as the sporadic server parameter pointer. In many real-time programs, the use of dynamic memory is not recommended, because it shows unbounded time behavior and creates the possibility of a critical task running out of memory. For this reason, the sporadic server package has also been implemented using only statically allocated memory. As current Ada does not allow pointers to static data structures, the static memory sporadic server has been implemented using an array of sporadic server control blocks, or parameter variables. This array is initially empty, and when each sporadic server is created one of its elements is allocated. When a sporadic server is deallocated, the array element is marked as empty, so that it can be used again by another sporadic server. The pointer to the sporadic server, which is a private type variable returned to the aperiodic task by the *initialize* procedure, is now an index to the array element.

The structure of the sporadic server package is basically the same as before, except for the different references to the sporadic server control blocks and for a new entry that is added to the sporadic server manager task (Table 14) to manage the creation and deallocation of sporadic servers in a coherent way.

Table 14 Static Storage Sporadic Server

```
package Sporadic_Server is

       ...

private
       type parameters is an index to an array of ss control blocks;
       ...

end Sporadic_Server;

package body Sporadic_Server is

       SS : array (parameters) of SS_control_block;

       task Sporadic_Server_Manager is
             entry init (ss_cb : out parameters);
             ...
       end Sporadic_Server_Manager;

       task body Sporadic_Server_Manager is

       begin
           loop
             select
                    accept init (ss_cb) do
                           ss_cb := index to free slot in SS array;
                    end init;
             or
                    ...
             end select;
           end loop;
       end Sporadic_Server_Manager;

end Sporadic_Server;
```

3.4 Sporadic Server Code

The code for the Ada task sporadic server, in each of its implementations, can be obtained from the Rate Monotonic Analysis for Real-Time Systems Project (RMARTS) at the Software Engineering Institute.

4 Library-Level Sporadic Server in POSIX

4.1 Real-Time and Threads Extensions to POSIX

The purpose of the IEEE Standard Portable Operating System Interface for Computer Environments (POSIX) [14] is to define a standard operating system interface and environment based on the UNIX[1] Operating System to support application portability at the source level. In this way, systems implementors and application software developers can work independently, and portability across a wide range of implementations can be accomplished for both the operating system and the applications.

Since the POSIX interface does not support applications with real-time requirements, extensions to the 1003.1 standard have been proposed and are currently in the standardization process [15]. The IEEE Standard P1003.4, Realtime Extensions to POSIX, defines in its current draft a minimum set of the interfaces required to make POSIX usable to real-time applications on single processor systems. The specific functions covered in this draft of the standard include:

Priority Scheduling: Several possibilities, among them a preemptive scheduler with no fewer than 32 priority levels.

Synchronization: Binary semaphores.

Memory Locking: Processes can make their address space resident in memory.

Shared Memory: Processes have independent address spaces, but there are mechanisms defined in the interface to share memory.

Clocks and Timers: Time-related functions that provide enough resolution for real-time applications.

Message Passing: An interprocess communications (IPC) interface, allowing asynchronous communication between processes.

Input/Output: Synchronous and asynchronous I/O.

Real-Time Signals: An extension to POSIX signals to improve determinism.

Another interesting extension to POSIX from the real-time point of view, which is also in the standardization process, is the Threads Extension for Portable Operating Systems (IEEE Std. P1003.4a) [16]. This standard provides interfaces to support multiple threads of control in each POSIX process; all threads share the same address space. The scope of this interface is single-processor or shared memory multiprocessor systems. Threads extension provides an interface to light-weight mechanisms to support very efficient implementations of logical or phys-

[1] UNIX is a registered trademark of AT&T.

ical concurrency for real-time systems. In particular, the threads extension is very important for small embedded systems; an application environment profile (AEP) for such systems is being designed; the AEP will define which options are needed and not needed.

The interest in the threads extension does not come only from the real-time community. Time-sharing multiprocessor systems are also interested in this extension because of its scope. Some data-base management applications or window environments, too, could benefit from this kind of lightweight threads of control, as opposed to the heavy-weight POSIX (or UNIX) process model.

Some of the most important functions covered in the current draft of 1003.4a are:

Thread Management: Creation, termination, and cancellation of threads within a single process.

Priority Scheduling: Preemptive scheduler with several options.

Synchronization: The paradigms defined by the interface are the mutex, for mutually exclusive access to shared resources, and condition variables (conditional critical sections), for waiting and signaling between threads. Priority inheritance protocols have been added as options to the interface.

1003.4 Mechanisms: Also, all the mechanisms available for processes under the real-time extensions, are available for threads (IPC message passing, binary semaphores, POSIX signals, etc.).

4.1.1 POSIX and RMS

Some interfaces necessary for the RMS scheduling algorithms already appear in the Standard 1003.4a. In particular, priority scheduling, the processor allocation scope, and some appropriate synchronization protocols already appear in Draft 5 of the standard [16]. The priority inheritance protocol and priority ceiling protocol are included as options in the mutually exclusive synchronization paradigm that has been defined for the POSIX threads.

Unlike the synchronization protocols, the sporadic server is not yet included in the 1003.4a standard. In order to facilitate its inclusion in this standard, a proposal has been made to define an interface that is compatible with both a kernel-level implementation and a library-level implementation. The current threads extension defines interfaces for a priority preemptive scheduler, and also provides an interface for dynamically changing the threads priorities. Therefore, the application-level sporadic server that has been introduced in this paper can be directly implemented at the library level. If the interface that is presented in this paper is approved, it will allow the implementor to decide at which level he or she wants to implement the sporadic server support, while the application developer can still write portable code that is functionally independent of the actual implementation. On the other hand, if the sporadic server interface is not included in the standard, the application-level sporadic server presented in this paper can still be implemented by the application developer.

The library implementation preserves all the important characteristics of the sporadic server which make it so useful for real-time systems. Its major advantages are its simplicity, and the fact that there is no penalty for applications that do not need to use the sporadic server; the main disadvantages are more overhead for the aperiodic thread and a worse average case performance. For those hard real-time systems with very rigid timing constraints, an implementation may choose the kernel level, which will provide a more efficient sporadic server mechanism, at the cost of a more complex kernel and a slight penalty in the overhead of the rest of the threads.

The library-level sporadic server that is proposed in this paper is quite similar to the Ada implementation. It provides a full version of the sporadic server, with the same worst-case performance as the normal sporadic server, at the affordable cost of losing some average case performance. Therefore, as we mentioned in Section 2.3, the simplified sporadic server can manage soft deadlines as well as guarantee the response to aperiodic events with hard deadlines.

In the remainder of this section, we briefly present the proposed sporadic server C-language interface, and the general structure of an aperiodic thread using this interface. We also present the pseudocode of a library-level implementation of the sporadic server, using the current 1003.4a interface as defined in Draft 5 of the standard [16]. Finally, in Appendix A we present the proposed POSIX interface for the sporadic server, using the terminology defined in this standard.

4.2 Using the Library-Level Sporadic Server

4.2.1 Data Structures

Each aperiodic thread that executes under the control of a sporadic server must initialize it and use it through the function calls that appear next. In all these calls, the user must supply a pointer to a data structure called the *sporadic server control block*. This pointer is initialized during the initialization call and updated in subsequent calls to the sporadic server functions. The pointer, a variable of the type named *pthread_ss_t*, is a record (*struc*) whose fields are implementation dependent, and store the basic sporadic server parameters: period, budget, priority levels, execution capacity, etc.

4.2.2 Initializing and Detaching Sporadic Servers

The sporadic server is initialized through a call to *pthread_ss_init*(), in which its parameters, the replenishment period, execution budget, normal priority, and background priority, are set. The arguments of this function are:

```
int pthread_ss_init(pthread_ss_t*ss,
                    struc timespec*period,
                    struc timespec*budget,
                    int normal_priority,
                    int background_priority)
```

where *timespec** is a pointer to the POSIX data structure that describes a time value.

When the initialized sporadic server is no longer necessary, a call may be issued to *pthread_ss_detach*(), to eliminate any internal references to the corresponding sporadic server and to deallocate its storage space. The function header is:

```
int pthread_ss_detach(pthread_ss_t*ss)
```

4.2.3 Arming and Requesting the Sporadic Server

While the interface described in the previous section would be enough for a kernel-level implementation of the sporadic server, two other functions are needed to make the library-level implementation possible. These functions can be chosen to be null in a full kernel-level implementation.

The first function, *pthread_ss_arm*(), is necessary to initiate the wait for the aperiodic event. As the replenishment time must be set to one period after the arrival of the aperiodic request, the aperiodic thread must wait for this event with a priority level higher than the priorities of any other threads in a library-level implementation. As the sporadic server thread is in a wait state, although its priority is high, the blocking effect on the rest of the threads is very low because as soon as the aperiodic request arrives, the thread's priority is lowered to the appropriate value. The function header is:

```
int pthread_ss_arm(pthread_ss_t*ss)
```

The second function needed in the library-level implementation is the start of the aperiodic processing, *pthread_ss_request*(). As soon as the aperiodic request arrives, this call is issued to determine the aperiodic thread's priority, normal or background, according to the available tickets, and to manage the replenishment policy.

In this call, the worst-case execution time of the aperiodic processing must be supplied. This time can be different from one execution to the other, depending on the kind of event being processed. A kernel-level implementation may choose to use this worst-case execution time or the actual execution time as the corresponding amount of consumed execution time. In the latter case, it may also choose to signal an error when the actual execution time exceeds the given worst-case bound. The arguments of this function are:

```
int pthread_ss_request  (pthread_ss_t*ss,
                         struc timespec*request_size)
```

4.2.4 User Code for an Aperiodic Thread

Given these functions, the user code for the aperiodic thread should have the structure shown in Table 15. The piece of code named “wait *for aperiodic event”* must be kept very small, ideally just a few instructions, because in a library-level implementation it will execute at a high priority and, therefore, its duration will be blocking time for the rest of the threads.

Table 15 Pseudocode of an Aperiodic Thread

```
/* Begin aperiodic thread */

       #include <sys/timers.h>
       #include <sporadic_server.h>]

       declare sporadic server vars. (sserver,period,budget,request_size);
{
       perform user initialization;
       pthread_ss_init (&sserver,period,budget,normal_pri,background_pri);
       while (1) {
              pthread_ss_arm(&sserver);
              wait for aperiodic event;
              pthread_ss_request(&sserver,request_size);
              process the event;
       } /* end while */;
} /* End aperiodic thread. */
```

4.3 Library-Level Implementation of the Sporadic Server

In this section, we present the pseudocode of a library-level implementation of the sporadic server as an example of the simplicity of such an implementation. This is not intended as either a guideline or a requirement for an actual implementation.

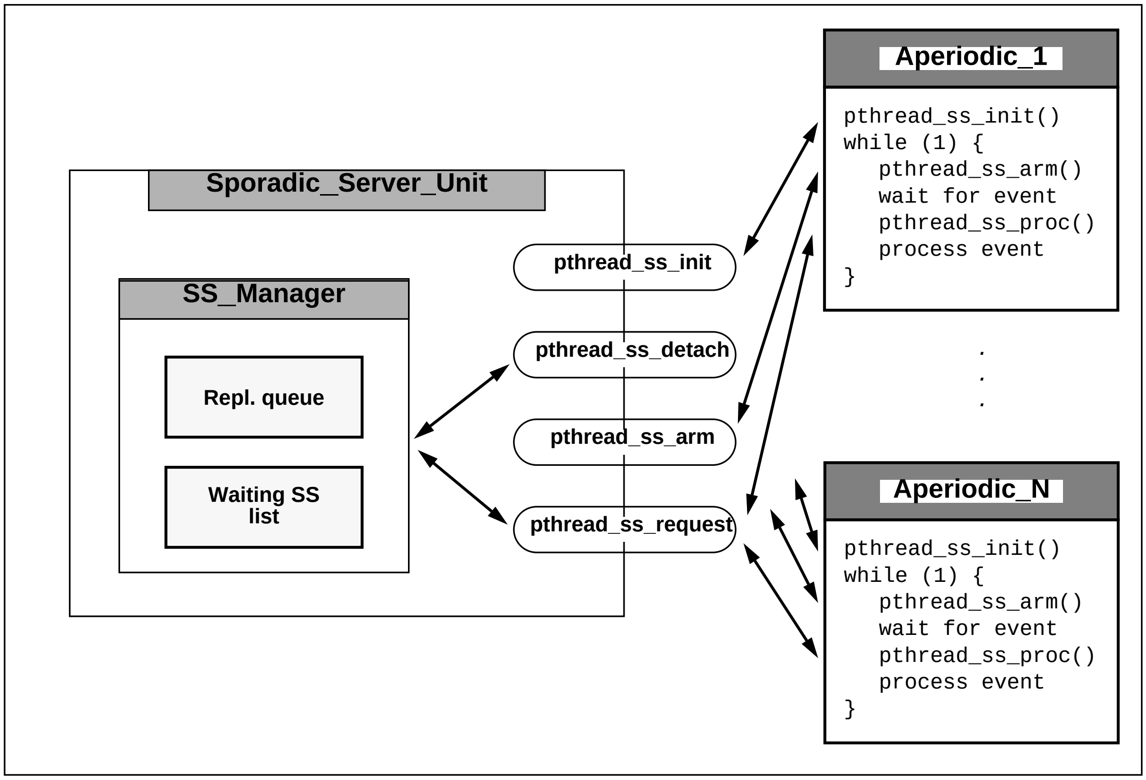
In this library-level implementation, a thread with the highest priority in the system manages all the ticket consumption and replenishment operations of all the sporadic server threads in the system. This thread is called the *sporadic_server_manager*.

Each sporadic server thread executes the initialization function and then issues a call to *pthread_ss_arm*() each time it has to wait for an aperiodic event. This function assigns the thread the highest priority in the system, preparing the sporadic server thread to wait for the event. As a result, the thread waits at a very high priority, but without consuming CPU resources. Ideally, no other threads should use this priority level.

When the event arrives, the sporadic server thread preempts the current work (because of its high priority) and issues a call to *pthread_ss_request*(). This function activates the *sporadic_server_manager* thread, which takes care of the ticket consumption and replenishment actions, determines and sets the appropriate priority of the thread (normal or background) and then returns the call. In this way, the sporadic server thread waits at the high priority and, when the aperiodic request arrives, it sets the replenishment time (if necessary) and immediately drops the aperiodic thread's priority to the appropriate level, so that the total time spent at the high priority is very small.

The *sporadic_server_manager* (Figure 10) maintains a queue of replenishment actions and a list of sporadic server requests waiting to execute. Basically, its structure is a loop in which the thread waits until a sporadic server requests to execute or a replenishment time has expired. When it wakes up it determines if there are replenishments to perform and if there are sporadic server threads waiting. The waiting sporadic server list is protected through a mutex, and each sporadic server thread is synchronized with the *sporadic_server_manager* through condition signaling. A mutex per sporadic server thread is also used to check and set the thread's priority in an indivisible way. The pseudocode of the library-level sporadic server appears in Table 16. Although this implementation will work on a multiprocessor, it may be better to have one sporadic server manager per processor to minimize the contention on such systems.

The optimization presented in the Ada task implementation of the sporadic server can also be applied here to reduce the number of context switches. In this case, each piece of consumed execution capacity is replenished not when it expires, but only when it is needed, that is, when the call to *pthread_ss_request()* is issued. Each sporadic server control block will include its own replenishment queue, which can be a simple FIFO queue. There will also be a wake-up priority queue, to store all the replenishment operations that imply switching an aperiodic task from background to assigned priority.



```
pthread_ss_init()
while (1) {
   pthread_ss_arm()
   wait for event
   pthread_ss_proc()
   process event
}
```

```
pthread_ss_init()
while (1) {
   pthread_ss_arm()
   wait for event
   pthread_ss_proc()
   process event
}
```

Figure 10 Structure of the Library-Level Sporadic Server

Table 16 Pseudocode of the Library-Level Sporadic Server (Global Variables)

```
/* Sporadic_Server_Manager’s Global Variables */
/*************************************************/

      declare the replenishment_queue;

      declare the ss_list; /* List of pending requests */
                          /* each with:request_size,
                                            waiting_flag,
                                            ss_control_block pointer,
                                            calling thread id
                                            request type (process or detach)
                           */

      pthread_mutex_t*ss_list_mutex;/*mutex for the ss_list*/

      pthread_cond_t*ss_list_cond;  /*condition var. for making requests*/
```

Table 16 (continued) Initialization Functions

```
/* Function pthread_ss_init */
/****************************/
      int pthread_ss_init (pthread_ss_t *ss,
                    struc timespec *period,
                    struc timespec *budget,
                    int normal_priority,
                    int background_priority)

      static pthread_once_t ss_manager=pthread_once_init;
      pthread_mutex_t *ss_thread_mutex;
      pthread_cond_t *ss_thread_cond;

      {
             /* Create the mutex and cond objects to manage the thread*/

             pthread_mutex_init(ss_thread_mutex,pthread_mutexattr_default);
             pthread_cond_init(ss_thread_cond, pthread_condattr_default);

             /* Start the sporadic_server_manager (once) */

             pthread_once(&ss_manager,&sporadic_server_init);

             initialize private attributes of sporadic server control block
                 (period, budget, priorities, mutex, condition, owner id ...);

      } /* End pthread_ss_init */

/* Function spordic_server_init */
/********************************/

      void sporadic_server_init ()

      pthread_t *thread;

      {
             initialize replenishment queue;
             initialize the list of waiting sporadic servers (SS_list);

             /* Create the mutex and cond objects for the ss waiting list*/

             pthread_mutex_init(ss_list_mutex,pthread_mutexattr_default);
             pthread_cond_create (ss_list_cond,pthread_condattr_default);

             /* Start the sporadic_server_manager thread */

             pthread_create(thread,pthread_attr_default,
                           &sporadic_server_manager);
             pthread_setprio(thread,PRIO_MAX);
      } /* End sporadic_server_init */
```

Table 16 (continued) Sporadic Server Manager Thread

```
void sporadic_server_manager ()
       {
          while (1) {
             get next replenishment from queue;

             /* Wait until replenishment time or ss request */

             pthread_cond_timedwait(ss_list_cond,ss_list_mutex,
                                        replenishment_time);

             /* Replenish due tickets */
             while (there are replenishments due) {
                    replenish tickets;

                    /* Replenishment Action: Indivisibly
                    test & increase priority */
                    pthread_mutex_lock (ss_thread_mutex);
                    if (pthread_getprio(ss_thread)=background p.) {
                       if (available_tickets>=request_size) {
                           decrement available_tickets by request_size;
                           set replenishment time into queue;
                           pthread_setprio(normal_priority,ss_thread);
                        } /* end if */
                    } /* end if */
                    pthread_mutex_unlock (ss_thread_mutex);
             } /* end while */

             /* Signal continuation to waiting sporadic servers */
             while (there are ss_threads waiting in ss_list) {

                if (request_type=process) {
                    /* Determine priority and replenishment actions */
                    if (available_tickets>=request_size) {
                           decrement available_tickets by request_size;
                           set replenishment time into queue;
                           prio:=normal_priority;
                    } else {
                           prio:=background_priority;
                    } /* end if */
                } else {                /*request_type=detach*/
                    detach ss from replenishment_queue;
                } /*end if*/

                    /* Lock ss thread mutex, set priority and signal */

                    pthread_mutex_lock (ss_thread_mutex);
                           pthread_setprio(prio,ss_thread);
                           clear waiting_flag;
                    pthread_mutex_unlock(ss_thread_mutex);
                    pthread_cond_signal(ss_thread_cond);
             } /* end while */
          } /* end while */
       } /* end sporadic_server_manager */
```

Table 16 (continued) Sporadic Server Detach Function

```
/* Function pthread_ss_detach */
/*******************************/

      int pthread_ss_detach (pthread_ss_t *ss)

      {

             /* Request detach to Sporadic_server_manager */

             pthread_mutex_lock(ss_list_mutex);
                   set request type (=detach) and waiting_flag in ss_list;
             pthread_mutex_unlock(ss_list_mutex);
             pthread_cond_signal(ss_list_cond);

             /* Wait for manager continuation signal */

             pthread_mutex_lock (ss_thread.mutex);
                   if (waiting_flag set) {
                          pthread_cond_wait(ss_thread.cond,
                                       ss_thread.mutex);
                   } /* end if */
             pthread_mutex_unlock (ss_thread.mutex);
      } /* end pthread_ss_request */
```

Table 16 (continued) Sporadic Server Request Function

```
/* Function pthread_ss_request */
/****************************/

       int pthread_ss_request (pthread_ss_t *ss_thread
                     struc timespec *request_size)

       {

              /* Request processing by Sporadic_server_manager */

              pthread_mutex_lock(ss_list_mutex);
                     set request_size and waiting_flag in ss_list;
              pthread_mutex_unlock(ss_list_mutex);
              pthread_cond_signal(ss_list_cond);

              /* Wait for manager continuation signal */

              pthread_mutex_lock (ss_thread.mutex);
                     if (waiting_flag set) {
                            pthread_cond_wait(ss_thread.cond,
                                          ss_thread.mutex);
                     } /* end if */
              pthread_mutex_unlock (ss_thread.mutex);
       } /* end pthread_ss_request */
```

Table 16 (continued) Sporadic Server Arm Function

```
/* Function pthread_ss_arm */
/***************************/

       int pthread_ss_arm (pthread_ss_t *ss_thread)

       {

              /* Set priority to high value */

              pthread_mutex_lock (ss_thread.mutex);
                     pthread_setprio(ss_thread,PRIO_MAX);
              pthread_mutex_unlock (ss_thread.mutex);
       } /* end pthread_ss_arm */
```

References

[1] L. Sha and J.B. Goodenough. “Real-Time Scheduling Theory and Ada,” *IEEE Computer*, April 1990.

[2] B. Sprunt, L. Sha. and J. Lehoczky. “Aperiodic Task Scheduling for Hard Real-Time Systems.” *The Journal of Real-Time Systems*, Vol. 1, 1989, pages 27-60.

[3] B. Sprunt, L. Sha.and J. Lehoczky. “Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.” Technical Report CMU/SEI-89-TR-11, DTIC: Ada211344 1989.

[4] B. Sprunt and L. Sha. “Implementing Sporadic Servers in ADA.” Technical Report CMU/SEI-90-TR-6, DTIC: ADA226723, 1990.

[5] B. Sprunt. “Aperiodic Task Scheduling for Real-Time Systems.” Ph.D. Thesis, Carnegie-Mellon University, August 1990.

[6] C.L. Liu and J.W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.” *Journal of the ACM 20 (1)*, January 1973, pages 46-61.

[7] L. Sha, R. Rajkumar and J.P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” *IEEE Trans. on Computers*, Vol. 39, 1990, pages 1175-1185.

[8] L. Sha. “Mode Change Protocols for Priority-Driven Preemptive Scheduling.” *Journal of Real-Time Systems*, Vol. 1, 1989, pages 243-264.

[9] M.H. Klein and T. Ralya. “An Analysis of Input/Output Paradigms for Real-Time Systems.” Technical report CMU/SEI-90-TR-19, ADA226724, Carnegie Mellon University, July 1990.

[10] J.P. Lehoczky, L. Sha and Y. Ding. “The Rate-Monotonic Scheduling Algorithm - Exact Characterization and Average Case Behavior.” *Proc. IEEE Real-Time Systems Symp.*, CS Press, Los Alamitos, CA, 1986.

[11] J.P. Lehoczky. “Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines.” *Proc. IEEE Real-Time Systems Symp.*, 1990.

[12] R. Rajkumar, L. Sha and J.P. Lehoczky. “Real-Time Synchronization Protocols for Multiprocessors.” *IEEE Real-Time Systems Symp.*, CS Press, Los Alamitos, CA, 1988.

[13] W. Amsbury. “Data Structures. From Arrays to Priority Queues.” Wadsworth, 1985.

[14] POSIX 1003.1. “Portable Operating System interface for Computer Environments.” *IEEE Standard* 1003.1, 1988.

[15] POSIX 1003.4. “Realtime Extension for Portable Operating Systems.” *IEEE Standard* P1003.4, Draft 9, Dec. 1989.

[16] POSIX 1003.4a. “Threads Extension for Portable Operating Systems.” *IEEE Standard* P1003.4a, Draft 5, Dec. 1990.

[17] J.F. Ready. “VRTX: A Real-Time Operating System for Embedded Microprocessor Applications.” *IEEE Micro*, Aug 1986, pages 8-17.

[18] ORKID Working Group,. “ORKID Open Real-Time Kernel Interface definition.” VITA, Draft 2.1, Aug. 1990.

[19] Ada Runtime Environment Working Group. “A Catalog of Interface Features and Options for the Ada Runtime Environment.” ACM-SIGAda, Release 2.0, Dec. 1987.

[20] “Ada 9X Requirements.” Ada 9X Project Team, Dec. 1990.

[21] “Reference manual for the Ada Programming Language.” ANSI/MIL-Std. 1815A, 1983.

Appendix A POSIX Sporadic Server Interface

The following functions are defined if symbol _POSIX_THREADS_SPORADIC_SERVER is defined. They provide the interface to the sporadic server mechanism to be used by any thread of a process:

pthread_ss_init(): Initialize a sporadic server to control the calling thread.

pthread_ss_detach(): Detach all references to the given sporadic server, which is no longer going to be used.

pthread_ss_arm(): Arm the sporadic server to wait for an aperiodic request.

*pthread_ss*_request(): Initiate the processing of an aperiodic request.

A.1. Sporadic Server Initialization and Detaching

A.1.1. Synopsis

Functions: *pthread_ss_init*(), pthread_ss_detach()

```
    #include <sys/timers.h>
    #include <pthread.h>

                    int pthread_ss_init (pthread_ss_t *ss,
                    struc timespec *period,
                    struc timespec *budget,
                    int normal_priority,
                    int background_priority);

                    int pthread_ss_detach(pthread_ss_t*ss);
```

A.1.2. Description

The *pthread_ss_init*() function initializes a sporadic server with the given budget and period to control the execution of the calling thread. The calling thread's execution must be limited to a certain budget during a window determined by the replenishment period. This thread calls the initialization function at its initialization section. When the execution limit has not been exceeded in a current window, this function, used along with *pthread_ss_arm*() and *pthread_ss_request*(), will cause the calling task to be assigned the given normal priority. Otherwise, it will be assigned the given background priority until new execution time is available for it.

When the calling thread is no longer going to execute under the control of the sporadic server it can call *pthread_ss_detach*() to detach the sporadic server. This call will release all memory allocated to that sporadic server, eliminate any pending operations with it and then return control to the caller. An implementation may cause *pthread_ss_detach*() to set *ss* to an implementation-defined illegal value.

A.1.3. Returns

Upon successful completion, *pthread_ss_init*() and *pthread_ss_detach*() will return 0. Otherwise a value of -1 is returned, the sporadic server is not initialized, and *errno* is set to indicate the error.

A.1.4. Errors

If any of the following conditions occur, the *pthread_ss_init*() function shall return -1 and set *errno* to the corresponding value:

EINVAL The value specified by *ss* is invalid, or one of the specified parameters is invalid. Two of these invalid conditions are: the requested priorities are outside the range of priorities available under the thread's current scheduling algorithm; the relationship 0<Budget<Period is not true.

EAGAIN The maximum amount of resources needed to manage sporadic servers has been exceeded.

ENOSYS The implementation does not support the sporadic server.

If any of the following conditions occur, the *pthread_ss_detach*() function shall return -1 and set *errno* to the corresponding value:

EINVAL The value specified by *ss* is invalid.

ENOSYS The implementation does not support the sporadic server.

A.1.5. References

pthread_ss_arm(), *pthread_ss_request*()

A.2. Arm Sporadic Server

Function: *pthread_ss_arm*()

A.2.1. Synopsis

#include <pthread.h>

**int pthread_ss_arm (pthread_ss_t ***ss);

A.2.2. Description

This function arms the sporadic server controlling the calling thread to become ready to accept an aperiodic request. It returns control to the calling thread, which must then execute the appropriate instructions to suspend the thread waiting until the aperiodic request arrives. This piece of code, immediately after the call to *pthread_ss_arm*() and before the call to *pthread_ss_request*, should be kept very short, ideally just a few instructions, because an implementation may choose to execute it at the highest priority level and, thereby introducing blocking time into all the other threads.

A kernel-level implementation of the sporadic server may choose to leave this function null. If a thread calls this function without a call to *pthread_ss_init* having been issued once by this thread, the behavior is undefined.

A.2.3. Returns

Upon successful completion, *pthread_ss_arm*() will return 0. Otherwise a value of -1 is returned, the sporadic server is not armed, and *errno* is set to indicate the error.

A.2.4. Errors

If any of the following conditions occur, the *pthread_ss_arm*() function shall return -1 and set *errno* to the corresponding value:

EINVAL	The specified sporadic server has an invalid value.
ENOSYS	The implementation does not support the sporadic server.

A.2.5. References

pthread_ss_init(), *pthread_ss_request*()

A.3. Sporadic Server Processing

Function: *pthread_ss_request*()

A.3.1. Synopsis

```
#include <sys/timers.h>
#include <pthread.h>

int pthread_ss_request (pthread_ss_t *ss,
struc timespec *request_size);
```

A.3.2. Description

This function tells the sporadic server controlling the calling thread that an aperiodic request has arrived, and also provides the worst-case execution time that the processing of the aperiodic event may consume, through the argument *request_size*. With regard to the ticket consumption and replenishment mechanisms, an implementation may choose to use this value or the actual execution time. Also, an implementation may choose to check that the actual execution time is less than or equal to the given worst-case bound.

A kernel-level implementation of the sporadic server may choose to leave this function null. If a thread calls this function without a call to *pthread_ss_init* having been issued once by this thread, the behavior is undefined.

A.3.3. Returns

Upon successful completion, *pthread_ss_request*() will return 0. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

A.3.4. Errors

EINVAL — The specified sporadic server has an invalid value or the *request_size* argument has an invalid value or is greater than the sporadic server's budget.

ERSIZE — The execution time of the processing of the last event by this sporadic server exceeded the corresponding request-size argument.

ENOSYS — The implementation does not support the sporadic server.

A.3.5. References

pthread_ss_init(), *pthread_ss_arm*()

A.4. Performance Metrics

Sporadic Server Arm — This is the time interval needed to execute *pthread_ss_arm*(). If this function increases the thread's priority, the new priority must be specified, along with the times spent by the function at each priority level.

Sporadic server Process — This is the time interval needed to execute *pthread_ss_request*(), when no other threads of priority higher than the background priority of the aperiodic thread are ready to execute on its processor.

Replenishment Overhead — This is the amount of time during which a thread is suspended, counting from the instant at which a single sporadic server with lower assigned priority initiates a replenishment action, to the instant it finishes. Metrics shall be provided for both the cases in which the sporadic server was or was not active at its background priority.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S
CMU/SEI-91-TR-26

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-91-TR-26

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (City, State and ZIP Code)
ESD/AVS
Hanscom Air Force Base, MA 01731

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESD/AVS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
An Application-Level Implementation of the Sporadic Server

12. PERSONAL AUTHOR(S)
Michael González Harbour and Lui Sha

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM TO

14. DATE OF REPORT (Yr., Mo., Day)
September 1991

15. PAGE COUNT
64

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
Sporadic server, rate monotonic scheduling, real-time scheduling

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this paper is to introduce a sporadic server algorithm that can be implemented as an application-level task, and that can be used when no runtime or operating system level implementation of the sporadic server is available. The sporadic server is a simple mechanism that both limits and guarantees a certain amount of execution power dedicated to servicing aperiodic requests with soft or hard deadlines in a hard real-time system. The sporadic server is event-driven from an application viewpoint, but appears as a periodic task for the purpose of analysis and, consequently, allows the use of analysis methods such as rate monotonic analysis [1] to predict the behavior of the real-

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Charles J. Ryan, Major, USAF

22b. TELEPHONE NUMBER (Include Area Code)
(412) 268-7631

22c. OFFICE SYMBOL
ESD/AVS (SEI

ABSTRACT —continued from page one, block 19

time system.

When the sporadic server is implemented at the application-level, without modification to the runtime executive or the operating system, some of its requirements cannot be met strictly and, therefore, some simplifications need to be assumed. We show that even with these simplifications, the application-level sporadic server proposed in this paper has the same worst-case performance as the full-featured runtime sporadic server algorithm, although the average case performance is slightly worse. The implementation requirements are a runtime prioritized preemptive scheduler and system calls to change a task's or thread's priority. Two implementations are introduced in this paper, one for Ada and the other for POSIX 1003.4a, Threads Extension to Portable Operating Systems.