Implementación en Ada 95 de servidores esporádicos en el nivel de aplicación

Por: M. González Harbour, J.J. Gutiérrez García, y J.C. Palencia Gutiérrez

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005 - Santander
SPAIN
email: {mgh, gutierjj, palencij}@ctr.unican.es
teléfono: 942 201483 - fax: 942 201402

Resumen[1,2]. El servidor esporádico es un algoritmo de planificación diseñado para planificar actividades aperiódicas en sistemas de tiempo real estricto, que se puede usar también en la planificación de actividades periódicas en sistemas distribuidos de tiempo real estricto. En los sistemas distribuidos, se puede incrementar sustancialmente la planificabilidad total del sistema, hasta un 50% más de utilización planificable, eliminando el retraso originado por el *jitter* mediante la utilización del servidor esporádico. La mayoría de los planificadores de tiempo real incluidos en sistemas operativos comerciales, o en el sistema de ejecución de Ada no suministran una implementación del algoritmo de planificación del servidor esporádico, y por lo tanto, para estos sistemas debe usarse una implementación en el nivel de aplicación. En este trabajo presentamos varias implementaciones de este algoritmo de planificación que se pueden realizar usando las características definidas en el lenguaje Ada 95. Estas implementaciones han sido diseñadas usando tipos de datos extensibles, para sacar partido de las características de herencia y polimorfismo del lenguaje.

Palabras clave: Planificación, Tiempo Real, Servidor Esporádico, Ada 95, Jitter.

1. Este trabajo es un borrador del artículo "*Implementing Application-Level Sporadic Server Schedulers in Ada 95*" publicado en Lecture Notes in Computer Science 1251, Reliable Software Technologies Ada-Europe'97, pp. 125-136
2. Este trabajo ha sido financiado en parte por la *Comisión Interministerial de Ciencia y Tecnología* dentro del proyecto TAP94-996

1. Introducción

La mayoría de las teorías de planificación de sistemas de tiempo real estricto se basan en métodos de análisis para tareas periódicas. Cuando se ejecutan actividades aperiódicas en el sistema, es necesario suministrar alguna forma de limitar los efectos de esas actividades sobre otras tareas de menor prioridad con requerimientos temporales estrictos, de forma que los resultados del análisis continúen siendo válidos. También es importante limitar los efectos de retraso causados por el *jitter*, que frecuentemente disminuyen la planificabilidad de los sistemas distribuidos. En estos sistemas muchas de las tareas se activan con la finalización de una tarea previa, que puede presentar un alto grado de variación temporal. Por consiguiente, aunque las tareas y mensajes continúan siendo básicamente periódicos, se activan en instantes irregulares; esto causa un efecto de retraso sobre tareas de menor prioridad, que pueden dañar seriamente la planificabilidad del sistema, reduciendo la utilización en muchos casos al 50 % [2].

La política de planificación del servidor esporádico fue diseñada para planificar la ejecución de actividades aperiódicas en sistemas de tiempo real estricto en un contexto de planificación por prioridades fijas. Es un algoritmo de reserva de ancho de banda que puede conseguir respuestas relativamente rápidas a eventos externos. El efecto de procesado de eventos aperiódicos utilizando un servidor esporádico nunca puede ser peor que el efecto de una tarea periódica equivalente. Gracias a esta propiedad, el servidor esporádico se puede utilizar para planificar actividades periódicas en

sistemas distribuidos, porque limita los retrasos en tareas de menor prioridad causados por el *jitter*, lo que permite alcanzar mayores niveles de utilización que los conseguidos si no se controlara el jitter adecuadamente.

La política de planificación del servidor esporádico tal como fue definida originalmente por Sprunt et al. [7], debería implementarse en el planificador de tareas, porque es necesario medir el tiempo de CPU consumido por las diferentes tareas. Sin embargo, dado que la mayoría de los sistemas operativos de tiempo real no suministran esta política de planificación, es posible implementarla (con ciertas restricciones) en el nivel de aplicación [1][4]. Otros algoritmos de planificación que han sido propuestos para planificar actividades aperiódicas en sistemas de tiempo real, y que suministran mejores resultados que el servidor esporádico [5], no se pueden implementar fácilmente en el nivel de aplicación con *overheads* pequeños, y ésta es la razón por la que hemos elegido el servidor esporádico.

En este trabajo presentamos varias implementaciones en el nivel de aplicación del algoritmo del servidor esporádico, que representan diferentes compromisos entre su funcionamiento y su complejidad. Esas implementaciones aprovechan las nuevas características definidas en el lenguaje Ada 95, tales como los objetos protegidos, para realizar la sincronización de datos de una forma más eficiente, y las primitivas de programación orientada a objetos, que permiten un diseño cómodo de las diferentes variantes de la implementación, y que permiten al usuario escribir el código de su aplicación de manera independiente de la implementación. En los sistemas distribuidos, además de los servidores esporádicos para las tareas necesitamos usar el algoritmo del servidor esporádico para planificar los mensajes en las redes de comunicaciones si queremos eliminar el efecto del jitter. La implementación de los servidores esporádicos para las comunicaciones y su descripción se puede encontrar en [3].

El artículo está organizado como se indica a continuación. En el apartado 2 damos un breve repaso del algoritmo de planificación del servidor esporádico, y describimos las diferentes operaciones que se utilizarán en el código de la aplicación. En el apartado 3 discutimos los aspectos básicos de tres implementaciones del servidor esporádico sin tener en cuenta el mecanismo utilizado en la recuperación del tiempo de ejecución consumido; esas implementaciones corresponden a los diferentes tipos de operaciones de recepción de eventos aperiódicos que pueden estar disponibles para una aplicación en particular. El apartado 4 discute la implementación de las políticas de relleno, que son una parte fundamental del servidor esporádico; veremos que las diferentes implementaciones corresponden a diferentes niveles de servicio y *overhead*. El apartado 5 muestra los resultados obtenidos con esas implementaciones, incluyendo sus tiempos de ejecución. Finalmente, en el apartado 6 exponemos nuestras conclusiones.

2. El algoritmo del servidor esporádico

El servidor esporádico es un algoritmo definido para planificar actividades aperiódicas en sistemas de tiempo real estricto. Este algoritmo reserva una cierta cantidad de tiempo (la capacidad de ejecución) para procesar eventos aperiódicos a un nivel de prioridad dado. El algoritmo también limita el efecto de expulsión sobre tareas de menor prioridad permitiendo, por tanto, predecir sus tiempos de respuesta de peor caso incluso en presencia de eventos aperiódicos con ritmos de llegada no acotados (es decir, con un número de activaciones potencialmente elevado en un intervalo reducido de tiempo). La Fig.1 muestra una comparación del comportamiento temporal de una tarea de prioridad alta planificada con diferentes métodos. Como se puede observar, si el evento se planifica directamente a una prioridad alta (Fig. 1-a), el tiempo de respuesta es corto pero puede haber una expulsión excesiva sobre tareas de prioridad más baja cuando llega un cúmulo de eventos aperiódicos de golpe. Si se utiliza el método de muestreo periódico, es decir, una tarea que comprueba periódicamente la llegada de los eventos (Fig. 1-b), la expulsión por periodo de muestreo está acotada incluso en presencia de un cúmulo de eventos aperiódicos, aunque el tiempo de respuesta conseguido sea peor. Con el planificador por servidor esporádico (Fig. 1-c), el tiempo de respuesta es mejor y la expulsión sobre tareas de menor prioridad está también acotada, ya que cuando llegan eventos muy juntos, su ejecución es espaciada (por el período de relleno) para permitir la ejecución de tareas de menor prioridad.

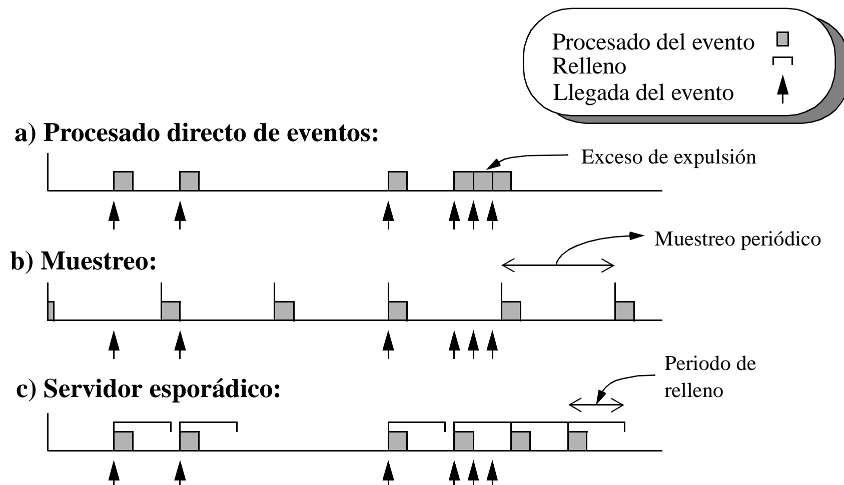

Fig. 1. Comportamiento temporal de varios planificadores aperiódicos

El algoritmo del servidor esporádico tiene los dos atributos básicos mencionados anteriormente: la capacidad de ejecución y el periodo de relleno. La capacidad de ejecución es el tiempo de ejecución reservado para el procesado de la tarea de aplicación que se está planificando al nivel de prioridad deseado. Cuando llega un evento aperiódico (o periódico), se activa la ejecución de su tarea de aplicación, y el instante de llegada del evento se registra para futuras referencias. Durante la ejecución de la tarea, el tiempo de ejecución gastado será restado de la capacidad, hasta que ésta se agote. En este caso, la ejecución de la tarea aperiódica se suspende. Cada porción de la capacidad de ejecución que se haya consumido se rellenará (es decir, se volverá a sumar a la capacidad disponible) en un instante posterior. Este instante es igual al instante en el cuál se activó la porción de capacidad consumida (normalmente, la llegada del evento de disparo) más un tiempo fijo llamado el periodo de relleno. En algunas implementaciones, la tarea de aplicación que no tenga capacidad de ejecución disponible puede seguir ejecutando con una prioridad de *background*, inferior a la prioridad de cualquier otra tarea con requerimientos de tiempo real estricto. En otras implementaciones, la tarea con capacidad agotada es sencillamente suspendida hasta que una operación de relleno incremente su capacidad de ejecución.

Una propiedad interesante del servidor esporádico es que sus efectos sobre la planificabilidad de tareas de menor prioridad no puede ser peor que los efectos de una tarea equivalente con un periodo igual al periodo de relleno y un tiempo de ejecución igual a la capacidad de ejecución inicial. Supongamos una tarea periódica que sufre activación retrasada (jitter). Si esta tarea se planifica utilizando un servidor esporádico con un periodo de relleno igual al periodo de la tarea, y una capacidad inicial igual al tiempo de ejecución de peor caso de la tarea, entonces su efecto sobre tareas de menor prioridad no puede ser peor que el de su tarea periódica equivalente sin jitter. El tiempo de respuesta de peor caso de la propia tarea es igual al mayor retraso posible en su activación, más el tiempo de finalización de peor caso de la tarea periódica equivalente, que se puede obtener fácilmente mediante las técnicas RMA usuales [4].

Cuando se implementa el servidor esporádico en el nivel de aplicación, no es posible conocer el tiempo de ejecución consumido, y por lo tanto, debemos suponer que en el peor caso se ha consumido un tiempo de ejecución igual al tiempo de ejecución de peor caso de cada uno de los eventos. Esta suposición pesimista es necesaria para garantizar los resultados del análisis de tiempo real de peor caso.

Desde el punto de vista de la implementación, el planificador por servidor esporádico es un objeto que define varias operaciones que deben ser invocadas por la tarea de aplicación que desee ejecutar bajo su control. El objeto tiene tres atributos públicos: capacidad inicial (*Initial_Capacity*), periodo de relleno (*Replenishment_Period*) y tiempo de ejecución de peor caso (*Worst_Case_Execution_Time*); el último es necesario para la implementación porque si la

capacidad de ejecución disponible es menor que el tiempo de ejecución de peor caso, entonces no se permite la ejecución de la tarea a su nivel normal de prioridad. Las operaciones definidas en el objeto servidor esporádico son:

- *Initialize*: Establece los valores iniciales de los atributos e inicializa el manejador de rellenos.
- *Prepare_To_Wait*: Realiza las operaciones necesarias antes de que la tarea se suspenda en espera del evento de disparo.
- *Prepare_To_Execute*: Realiza las operaciones requeridas después de la llegada del evento y antes de que se de comienzo a su procesado.
- *Schedule_Next*: Realiza las operaciones de planificación necesarias antes de que la tarea se prepare para esperar al próximo evento.

Tabla 1. Tarea de aplicación que utiliza un servidor esporádico

```
task body Application_Task is
   SS : Sporadic_Server.Scheduler;
begin
   Initialize(SS);
   loop
      Prepare_To_Wait(SS);
      Wait_For_Event;
      Prepare_To_Execute(SS);
      Do_Task's_Work;
      Schedule_Next(SS);
   end loop;
end Application_Task
```

La Tabla 1 muestra el pseudocódigo de una tarea de aplicación que ejecuta bajo el control del servidor esporádico en el nivel de aplicación. El planificador por servidor esporádico se diseña como una clase abstracta de objetos con las cuatro operaciones descritas anteriormente, y a partir de ella se derivan todas las diferentes implementaciones. En las siguientes secciones describimos estas implementaciones.

3. Las implementaciones del servidor esporádico

Dependiendo de las operaciones que estén disponibles para esperar al evento que dispara a la tarea de aplicación, tenemos tres implementaciones básicas del servidor esporádico:

- *Simple*: Si el evento de disparo está marcado con su instante de llegada (*timestamp*), entonces el cálculo del instante de activación del evento es sencillo: es el máximo entre el tiempo marcado en el evento y el último instante en que la tarea no tuvo capacidad de ejecución disponible. Esto da idea de la sencillez de la implementación del servidor esporádico, por lo que le llamamos *Simple*. A efectos de usar el tiempo marcado en el evento, el pseudocódigo del bucle principal de la tarea de aplicación debe ser como el mostrado en la Tabla 2. Las acciones realizadas por las diferentes operaciones del servidor esporádico se muestran en la Tabla 3. El parámetro *S* en esas operaciones es el planificador.

Tabla 2. Tarea de aplicación con marcado temporal de eventos

```
loop
   Prepare_To_Wait(SS);
   Wait_For_Event (Timestamp);
   Prepare_To_Execute
               (SS,Timestamp);
   Do_Task's_Work;
   Schedule_Next(SS);
end loop;
```

Tabla 3. Operaciones de la implementación

```
Prepare_To_Wait (S) is null;

Prepare_To_Execute
     (S,Timestamp) is
begin
   S.Activation_Time:=
      Max(Timestamp,S.Next_Start);
end;

Schedule_Next (S) is
begin
   S.Next_Start:=
      S.Activation_Time+
      S.Replenishment_Period;
   delay until Next_Start;
end;
```

- *High_Priority*: Si no se puede disponer del instante de llegada marcado en el evento, el instante de activación de la tarea se puede registrar después de la llegada del evento, leyendo el reloj. Sin embargo, la tarea podría ser expulsada entre la llegada del evento y el

registro, y por tanto, el instante de activación anotado podría ser posterior al debido, causando que el servidor esporádico planificara el siguiente evento demasiado tarde. Este problema se puede resolver parcialmente si el marcado del instante de activación se realizara a un nivel de prioridad mayor. En la implementación *High_Priority*, la operación *Prepare_To_Wait* incrementa la prioridad de la tarea al máximo nivel y *Prepare_To_Execute* la decrementa a su nivel normal después de haber registrado el instante de activación de la tarea. Además, como la tarea puede ser expulsada antes de que se eleve su prioridad, lo que produce un marcado erróneo del instante de activación de los eventos que podrían haber llegado durante la ejecución de la instrucción *delay* en la operación *Schedule_Next*, este *delay* se mueve a la operación *Prepare_To_Execute* (esto es posible porque el requerimiento de esta instrucción es que ocurra antes de que se procese el evento). La Tabla 4 describe las operaciones para esta implementación. En esta implementación, la tarea de aplicación tiene el mismo pseudocódigo que se muestra en la Tabla 1.

Tabla 4. Operaciones de la implementación

```
Prepare_To_Wait (S) is
begin
   S.Current:=Get_Priority;
   Set_Priority(Highest);
end;

Schedule_Next (S) is
begin
   S.Next_Start:=
      S.Activation_Time+
      S.Replenishment_Period;
end;

Prepare_To_Execute (S) is
   Timestamp:=Clock;
begin
   Set_Priority (S.Current);
   delay until S.Next_Start
   S.Activation_Time:=
      Max(Timestamp, S.Next_Start);
end;
```

Con la aproximación *High_Priority*, se minimiza la diferencia entre la llegada actual del evento y el registro del instante de activación. La creación de una sección de prioridad alta introduce un retraso extra en las otras tareas que es pequeño y acotado, básicamente igual al tiempo que lleva el manejo del evento, el registro del instante de llegada, y la disminución de la prioridad. Aunque el instante de activación se registra a prioridad alta, inmediatamente después de que evento llegue, existe una pequeña diferencia entre la llegada del evento y el registro del instante de activación causada por el *overhead* en la espera y el registro del instante. Esto implica que el siguiente evento se puede planificar ligeramente tarde, a menos que el periodo de relleno se reduzca para tener en cuenta el efecto del *overhead*.

- *High_Priority_Polled*: Existe una solución que se puede utilizar para resolver el hecho de que la implementación *High_Priority* no sea óptima cuando el evento no se puede marcar con su instante de llegada. Esta solución es aplicable cuando se puede averiguar si el evento ha llegado o no (muestreo). En este caso, el servidor esporádico requiere otra nueva operación, *Execute_Without_Waiting*, que se invoca si el evento ya había llegado cuando se preguntó por su llegada. Esta operación calcula el instante de activación del evento como el instante de activación del evento anterior más el periodo de relleno. Para esta implementación, el pseudocódigo del lazo principal en la tarea de la aplicación debe ser como el mostrado en la Tabla 5.

Tabla 5. Tarea de aplicación con muestreo de eventos

```
loop
   Poll_For_Event_Arrival
   if available then
      Execute_Without_Waiting;
   else
      Prepare_To_Wait(SS);
      Wait_For_Event;
      Prepare_To_Execute(SS);
   end if;
   Do_Task's_Work;
   Schedule_Next(SS);
end loop;
```

Las operaciones en esta implementación son similares a las de la implementación *High_Priority*, excepto que la instrucción *delay* que se utiliza para cumplir con el periodo de relleno se mueve a la operación *Schedule_Next*. La razón para esto es invocar la operación de

muestreo lo más tarde posible, de manera que aumente la probabilidad de que encuentre que el evento ya ha llegado. La Tabla 6 muestra la implementación de las operaciones para este caso.

Tabla 6. Operaciones en la implementación *High_Priority_Polled*

```
Prepare_To_Wait (S) is null;

Schedule_Next (S) is
begin
   S.Next_Start:= S.Activation_Time+
      S.Replenishment_Period;
   delay until S.Next_Start
   S.Current:=Get_Priority;
   Set_Priority(Highest);
end;

Execute_Without_Waiting(S) is
begin
   S.Activation_Time:= S.Next_Start;
   Set_Priority (S.Current);
end;

Prepare_To_Execute (S) is
   Timestamp:=Clock;
begin
   S.Activation_Time:=
      Max(Timestamp, S.Next_Start);
   Set_Priority (S.Current);
end;
```

Además de los tres tipos de implementaciones, la política utilizada para rellenar la capacidad de ejecución consumida puede tener diferentes grados de complejidad y niveles de servicio. Se han definido cuatro tipos de política de relleno, que se discutirán en el siguiente apartado. La combinación de las tres implementaciones básicas y de las cuatro políticas de relleno generan doce implementaciones diferentes para el servidor esporádico. La Fig. 2 muestra el diagrama de las diferentes clases con sus atributos y operaciones, de acuerdo con la notación OMT de Rumbaugh [6]. Debido a que Ada 95 no soporta directamente herencia múltiple, cada objeto servidor esporádico tiene un atributo puntero que apunta al manejador de rellenos deseado (si se necesita uno). Podríamos haber utilizado aquí una aproximación de herencia mixta definiendo cada uno de los planificadores de relleno como un paquete genérico que extendiera el planificador del servidor esporádico con el planificador de rellenos adecuado. Esta aproximación nos hubiera permitido evitar la repetición del código de varias operaciones, pero podría imposibilitarnos el uso de la herencia o el polimorfismo para los planificadores de relleno, ya que no podrían seguir siendo objetos etiquetados (*tagged*). Esta es la razón de que no usemos la aproximación mixta.

4. Las políticas de relleno

Hemos definido cuatro políticas de relleno para nuestras implementaciones del servidor esporádico. La más simple es la política *Single* en la que la capacidad inicial del servidor esporádico es igual al tiempo de ejecución de peor caso de la tarea activada por el evento. En este caso la política de relleno es extremadamente simple, porque la capacidad se consume completamente cuando se procesa el evento, y se rellena completamente un periodo de relleno después de la llegada del evento. Esta política se utiliza en las implementaciones llamadas *Simple_1*, *High_Priority_1* y *High_Priority_Polled_1* (ver Fig. 2), y corresponde a las operaciones del servidor esporádico mostradas en la Tabla 3, en la Tabla 4 y en la Tabla 6.

Las otras tres políticas de relleno corresponden al caso en que la capacidad inicial permite la ejecución de múltiples eventos antes de que se agote. Hemos definido esas tres políticas como tres objetos diferentes para el manejo de rellenos, derivados de la misma clase, cada una con tres operaciones (ver Fig. 2): inicializa (*Initialize*), planifica un relleno (*Schedule_Replenishment*) y solicita ejecución (*Request_Execution*). Los tres diferentes manejadores de rellenos implementados son:

- *Non_Queued*: Este manejador no tiene en cuenta los instantes de activación de las diferentes porciones de tiempo de ejecución consumido. Cuando no hay suficiente capacidad de ejecución para servir un evento, la operación *Request_Execution* suspende la tarea de la aplicación hasta un periodo de relleno después del instante de activación del último evento procesado. Después de la suspensión se rellena totalmente la capacidad de ejecución. *Schedule_Replenishment* resta el tiempo de ejecución consumido de la capacidad disponible.

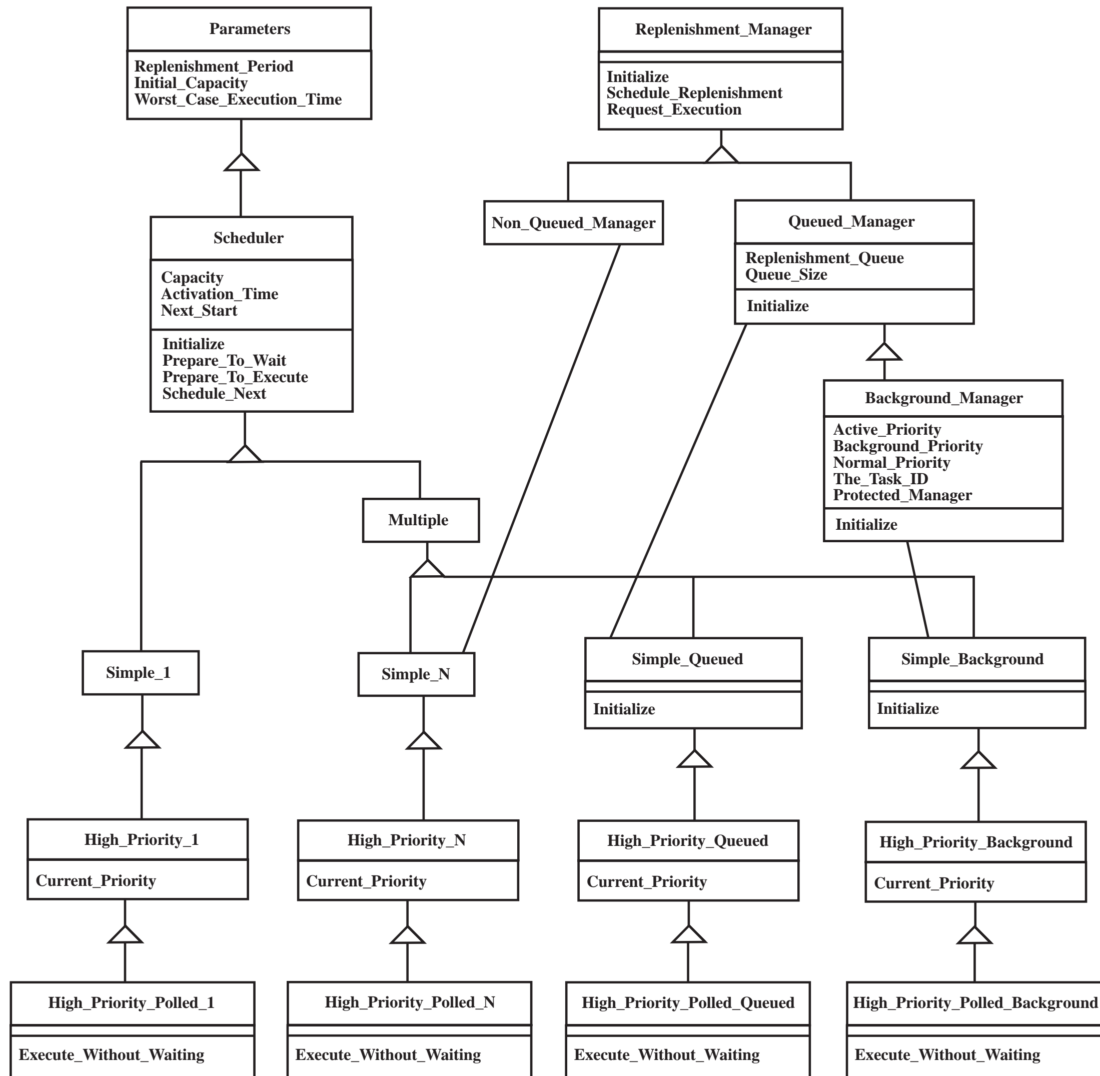

Fig. 2. Diagrama de las clases de servidor esporádico

- *Queued*: En este manejador existe una cola para las operaciones de relleno. Cada vez que se procesa un evento, *Schedule_Replenishment* resta el tiempo de ejecución consumido de la capacidad disponible e inserta una operación de relleno en la cola, planificándolo para que ocurra un periodo de relleno después del instante de activación del evento. La operación *Request_Execution* procesa los rellenos pendientes, y si no hay suficiente capacidad de ejecución, suspende la tarea de la aplicación hasta un instante en que una operación de relleno pueda causar que la capacidad disponible sea suficiente para procesar un evento. Una operación de relleno se procesa desencolándola y añadiendo la capacidad consumida a la capacidad disponible. La cola de rellenos está organizada como una cola con prioridad, en la que la prioridad es el instante en el que se debe efectuar la operación de relleno.
- *Background*: Esta es una extensión del manejador encolado en la que se permite que la tarea de la aplicación ejecute con una prioridad de *background* cuando haya agotado su capacidad de ejecución. Una tarea especial (*Rooster*) se encarga de despertar (es decir, elevar su prioridad) a la tarea de la aplicación cuando una operación de relleno incrementa la capacidad disponible. Las comunicaciones entre las tareas de la aplicación y la tarea *Rooster* se hacen a través de un objeto protegido llamado *Wakeup_Manager*, para minimizar el número de cambios de contexto, y

por tanto el *overhead*. Además, cada planificador del servidor esporádico tiene su propio objeto protegido, llamado *Protected_Manager*, para tener acceso a la cola de rellenos y a los parámetros del servidor esporádico. La Fig. 3 muestra la arquitectura básica de esta implementación.

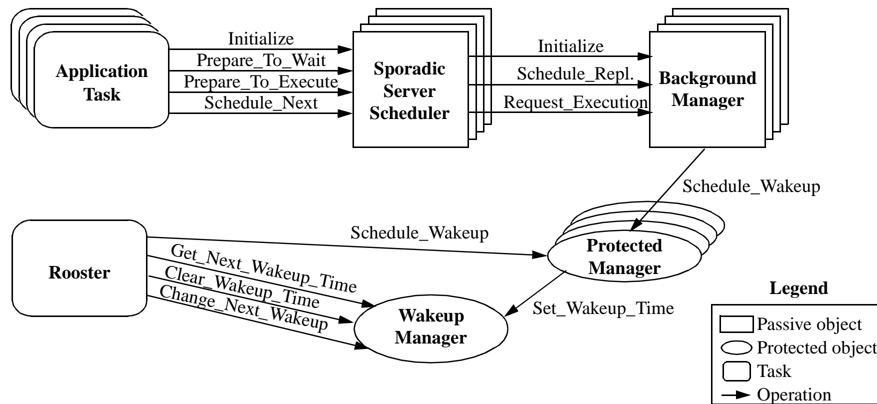

Fig. 3. Arquitectura básica de las implementaciones que utilizan el *Background Manager*

Es necesario modificar las operaciones del servidor esporádico con respecto a las operaciones descritas en las Tablas 3, 4 y 6 para la política de planificación *Single* del siguiente modo: la operación *Schedule_Next* invoca a la operación *Schedule_Replenishment* en lugar de calcular *Next_Start*; y la instrucción *delay* se sustituye por una llamada a la operación *Request_Execution*.

5. Funcionamiento de las diferentes implementaciones

Al evaluar los resultados de las diferentes implementaciones del servidor esporádico debemos tener en cuenta los diferentes requerimientos que pueden tener las tareas de la aplicación. Por ejemplo, podríamos tener tareas aperiódicas esporádicas que tuvieran un tiempo mínimo garantizado entre llegadas de eventos. En este caso la tarea podría tener impuestos requerimientos temporales estrictos, y la forma de garantizar que esos requerimientos se van a cumplir sería crear un servidor esporádico capaz de procesar todos los eventos que llegan al sistema, aunque lleguen a su máximo ritmo permitido. El servidor esporádico se crearía con capacidad para procesar un evento, y con un periodo de relleno igual al intervalo mínimo entre llegadas. La política de rellenos podría ser en este caso la más simple (la política *Single*), ya que sólo se procesaría un evento por periodo. Deberíamos elegir la implementación *Simple_1* si los eventos vienen marcados con su instante de llegada, o la implementación *High_Priority_Polled_1* si podemos comprobar si el evento ha llegado o no. Si ninguna de ellas está disponible, podemos elegir la implementación *High_Priority_1*, en la cual debemos reducir el periodo de relleno en una cantidad igual al retraso de peor caso entre la llegada del evento y el registro de su instante de llegada. Este retraso se puede calcular utilizando la teoría RMA [4]. Por supuesto, la reducción del periodo de relleno tiene su pequeño impacto negativo en la planificabilidad de las tareas de menor prioridad.

En el caso de tareas periódicas con jitter podemos requerir también que el servidor esporádico sea capaz de procesar un evento por periodo, ya que de otra forma podríamos acumular eventos e incrementar fuertemente el tiempo de repuesta. En este caso, los requerimientos y soluciones son los mismos que para tareas esporádicas con requerimientos estrictos mencionado anteriormente.

Para aquellas tareas periódicas o aperiódicas con requerimientos no estrictos de tiempo real, podemos usar un servidor esporádico para mejorar los tiempos de respuesta promedio, mientras se mantienen acotados los tiempos de respuesta de las tareas de menor prioridad. En este caso probablemente queramos un servidor capaz de procesar varios eventos en cada periodo de relleno, y por lo tanto elegiremos un manejador de rellenos *Multiple*. Si el sistema tiene un nivel bajo de utilización y tenemos suficiente margen temporal entre los tiempos de respuesta y los plazos para las tareas con requerimientos estrictos, podemos

utilizar un manejador que permita la ejecución en *background*. La implementación del servidor esporádico (*Simple*, *High_Priority* o *High_Priority_Polled*) se elige de acuerdo con las operaciones de llegada de eventos disponibles en el sistema.

La Tabla 7 muestra los *overheads* asociados con cada una de las operaciones. Los tiempos de ejecución están medidos en microsegundos y la plataforma utilizada para medirlos fue GNAT 3.05 bajo Linux 1.2.13 en una CPU 486 a 100 MHz. Estos números pueden ayudar a seleccionar la implementación más apropiada para una aplicación particular. Una descripción más detallada de estas medidas, incluyendo todos los datos necesarios para la aplicación de las técnicas RMA [4][8] que determinan si se verifican los requerimientos temporales, están disponibles en la siguiente dirección, junto con el código fuente de las implementaciones: *ftp://ftp.unican.es/pub/misc/mgh/*

Tabla 7. Medidas del funcionamiento de las operaciones del servidor esporádico (μs)

Implementación	Prepare_To_Wait	Prepare_To_Execute	Schedule_Next	Execute_Without_Waiting
Simple_1	10	51	62	-
Simple_N	3	32	123	-
Simple_Queued	9	43	328	-
Simple_Background	7	281	734	-
High_Priority_1	188	228	46	-
High_Priority_N	169	181	70	-
High_Priority_Queued	134	297	112	-
High_Priority_Background	139	723	131	-
High_Priority_Polled_1	0	117	197	60
High_Priority_Polled_N	0	138	274	71
High_Priority_Polled_Queued	0	119	433	69
High_Priority_Polled_Background	0	335	795	199

6. Conclusiones

El algoritmo de planificación del servidor esporádico es muy adecuado para el procesado de eventos aperiódicos con un tiempo de respuesta relativamente pequeño, mientras se garantizan los requerimientos temporales de todas las tareas de menor prioridad. El servidor esporádico puede eliminar además el efecto del jitter en tareas periódicas. Esto es especialmente importante en sistemas distribuidos en los que el efecto del jitter puede reducir la planificabilidad del sistema hasta la mitad. Aunque el servidor esporádico está pensado para implementarse en el nivel del planificador de las tareas, muy pocos sistemas comerciales ofrecen esta política de planificación. Para resolver este problema, en este trabajo hemos presentado diferentes estrategias que se pueden utilizar para la implementación del algoritmo del servidor esporádico en el nivel de la aplicación.

Se muestran diferentes implementaciones con diferentes compromisos entre resultados y nivel de servicio. La implementación básica del servidor esporádico se puede elegir de acuerdo con las operaciones que sea capaz de realizar el sistema en la recepción de los eventos externos. La política de manejo de rellenos se elige de acuerdo con los requerimientos temporales de la aplicación. También se suministran las medidas para ayudar al programador a elegir la mejor implementación para su sistema.

7. Bibliografía

[1] M. González Harbour and L. Sha: “An Application-level Implementation of the Sporadic Server” Technical Report CMU/SEI-TR-26-91, September 1991.

[2] J.J. Gutiérrez García, and M. González Harbour, “Increasing Schedulability in Distributed Hard Real-Time Systems”. Proceedings of the 7th Euromicro Workshop on Real-Time Systems, Odense, Denmark, June 1995, pp. 99-106.

[3] J.J. Gutiérrez García, and M. González Harbour, “Minimizing the Effects of Jitter in Distributed Hard Real-Time systems”. Journal of Systems Architecture, 42, Num. 6&7, December 1996.

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

[5] S.Ramos-Thuel and J.P. Lehoczky, "On-line scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems", Proceedings of the Real-Time Systems Symposium, December 1993, pp. 160-171.

[6] J. Rumbaugh, "Object-Oriented Modeling and Design", Prentice-Hall, 1991.

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

[8] K. Tindell, and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems". *Microprocessing & Microprogramming*, Vol. 50, Nos.2-3, April 1994, pp. 117-134.