Implementación de Arquitecturas de Gestión de Calidad de Servicios sobre Plataformas de Máquina Virtual

Alumno
Víctor José Fernández Rancaño

Tutora
Marisol García Valls

Índice

CAPITULO 1. INTRODUCCIÓN

1 Motivación

En los últimos años se ha producido un incremento en la capacidad de cómputo de los terminales empotrados debido a la mejora de los procesadores, lo que ha posibilitado el auge de las tecnologías móviles. Hasta hace muy pocos años existían funciones en estos terminales que estaban implementadas directamente en el hardware para hacerlas lo suficientemente eficientes. Con el aumento de estas capacidades se han podido implementar estas funciones, antes hardware, como hilos software gestionadas por los sistemas operativos.

Los sistemas operativos son, ante todo, los administradores de los recursos que el hardware ofrece. El principal recurso que administran es el procesador o procesadores, aunque también gestionan los medios de almacenamiento, los dispositivos de entrada/salida, los dispositivos de comunicación y los datos.

Existen varios tipos de sistemas operativos, entre los sistemas operativos que son usados en los terminales empotrados cabe destacar los sistemas operativos de tiempo real (SOTR o RTOS), que son aquellos que han sido desarrollados para aplicaciones de tiempo real. Como tal, se le exige corrección en sus respuestas bajo ciertas restricciones de tiempo. Si no las respeta, se dirá que el sistema ha fallado.

Sin las mencionadas restricciones hardware y el uso de los sistemas operativos de tiempo real en los dispositivos empotrados se abre un abanico de posibilidades para la ejecución de nuevos sistemas multimedia.

Los sistemas multimedia se caracterizan, desde el punto de vista de los recursos hardware requeridos, por ser avariciosos (*greedy*), es decir, siempre que pueda acceder al procesador lo harán y mejorarán la calidad mostrada al usuario, esto podría provocar una situación de inanición de uno de los medios del sistema multimedia. Esta característica especial de los sistemas multimedia hace que sea necesario la introducción de un sistema gestor de la calidad de servicio entre el sistema operativo de tiempo real y la aplicación de usuario para evitar que la avaricia de cada uno de los medios degrade la calidad de alguna de las aplicaciones que se ofrecen al usuario.

Por otro lado, la actual tendencia del mercado hacia el uso de procesadores multicore hace que para poder obtener toda la potencia que ofrecen estos nuevos procesadores sea necesario el uso de los gestores de calidad de servicio para mejorar en lo posible el rendimiento de los programas ejecutados.

En este trabajo se realiza una implementación alternativa de una arquitectura de gestión de la calidad de servicio sobre plataformas software que poseen máquina virtual, concretamente sobre tecnologías Java de tiempo real y sobre la implementación RTSJ. El hecho de portar esta arquitectura a esta plataforma impone numerosos obstáculos que se han resuelto y que han permitido evaluar la idoneidad de esta plataforma para estos propósitos.

2 Términos principales

Gestor de Calidad de Servicio

Sistema encargado de arbitrar la asignación y uso de recursos entre las distintas aplicaciones del sistema. Este arbitrio lo realiza atendiendo a los parámetros de calidad de servicio de las aplicaciones

Aplicación

Una entidad de programa que es capaz de procesar datos multimedia y ofrecer un resultado final al usuario.

Calidad de Servicio [1]

Es cada una de las distintas posibilidades que ofrece una aplicación en su resultado final y que el usuario es capaz de distinguir y seleccionar. Por ejemplo, para una aplicación de audio corresponderían a combinaciones de filtros como: reducción del ruido, sonido estéreo y modo surround.

Cluster

Entidad consistente en un grupo de tareas (a un nivel de abstracción menor que una aplicación). La agrupación de tareas permite asignar una única cuota de utilización de recursos. Ello permite compensar incumplimientos de utilizaciones de recursos y plazos de las tareas individuales.

Tarea

Es un hilo de ejecución o mínima unidad de ejecución del sistema. Una tarea pertenece a una aplicación y forma parte de un grupo determinado.

Sistema de Tiempo Real

Son aquellos sistemas en que las tareas que se ejecutan en él tienen un plazo de cumplimiento antes del cual la tarea debe haberse realizado. Existen dos tipos básicos de sistemas de tiempo real: *críticos*, en que el incumplimiento de un plazo de una de las tareas provoca errores irreparables y *acríticos*, en que el incumplimiento de un plazo de una tarea es un error reparable.

Maquina Virtual

Es un entorno de ejecución que provee una abstracción respecto al sistema sobre el que realmente se ejecuta una aplicación. Esto permite que un mismo programa se pueda ejecutar en distintas máquinas usando máquinas virtuales distintas.

Java

Es un lenguaje orientado a objetos que utiliza una máquina virtual para ejecutar el código objeto resultado de la compilación del código fuente Java. Se han implementado máquinas virtuales de Java para las principales plataformas, lo que hace que Java sea un lenguaje “independiente” de la plataforma.

RTSJ

Acrónimo de Real Time Specification for Java. Es una especificación de las extensiones que se deben añadir a la versión de Java estándar para cumplir los criterios de sistema de tiempo real.

Factibilidad

Es la característica de una tarea de tiempo real por la cual es posible determinar si los recursos que necesita esa tarea están disponibles en el sistema. La factibilidad depende de la carga que haya en el sistema, por lo que una tarea no tiene porque ser realizable siempre.

Planificador

Es la parte de sistema encargada de decidir, en función de su factibilidad, el orden de ejecución de los hilos en un sistema con las garantías suficientes de que se cumplirán sus restricciones de tiempo.

Elegibilidad

Es la característica por la que el planificador de hilos decide que hilo se puede ejecutar.

Despachador

Es la parte de sistema encargada de elegir, en función de su elegibilidad, qué hilo entrará en el procesador

Pila

La pila es el espacio de memoria en que se pasan los parámetros de las funciones y en que se guardan los ámbitos de ejecución (scope) de las mismas.

Recolector de Basura

Es un subsistema de la máquina virtual de Java que se encarga de liberar de la memoria aquellos datos que han dejado de usarse.

3 Vista Panorámica del Trabajo

El proyecto consiste en:

- la implementación y adaptación de una arquitectura de gestión de la calidad de servicio independiente de la plataforma
- su adecuación para ejecutar sobre entornos de máquina virtual
- la validación sobre una plataforma concreta como es la plataforma de Java de Tiempo Real (concretamente sobre TimeSys sobre Linux) de que la arquitectura creada es válida en un sistema operativo de tiempo real.

Una arquitectura de gestión de la calidad de servicio ofrece una interfaz común al usuario para las operaciones relacionadas con la ejecución de aplicaciones con calidad de servicio independientemente del sistema operativo que resida en la máquina en que se ejecute el gestor de la calidad de servicio.

Esto hace que el desarrollo sea, desde el punto de vista del usuario, el mismo para todas las plataformas y que tenga un comportamiento semejante.

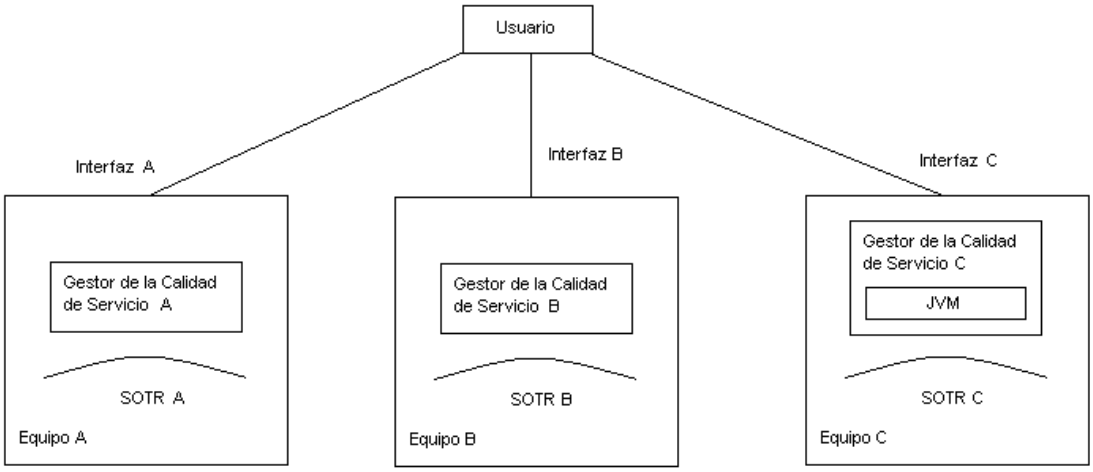

Figura 1.2.1 Interfaces Usuario-Gestor de Calidad sin arquitectura de gestión de la calidad

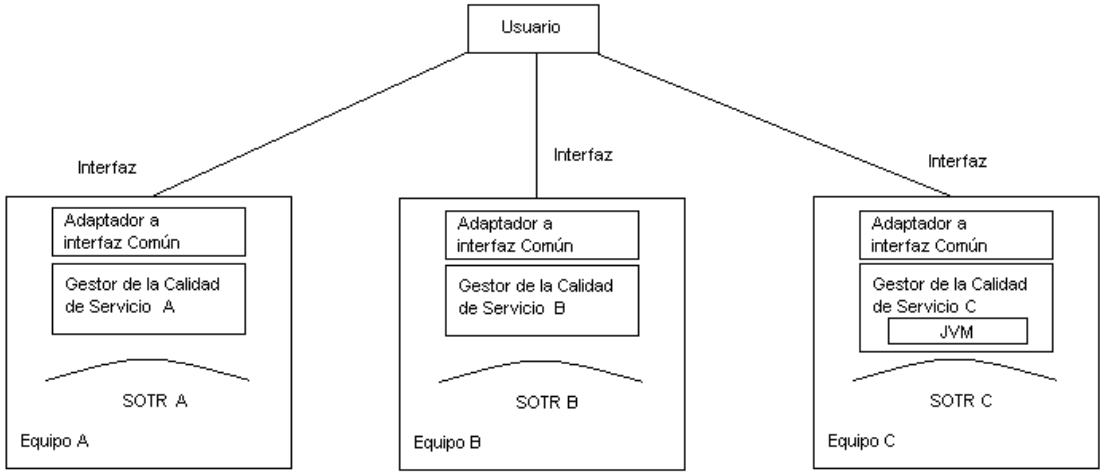

Figura 1.2.2 Interfaces Usuario-Gestor de Calidad con arquitectura de gestión de la calidad

Como se puede ver en las figuras anteriores si se quisiera soportar un nuevo tipo de equipo, por ejemplo el equipo D, en el primer caso (figura 1.2.1) habría que desarrollar el software del usuario desde cero usando la interfaz D para todas las aplicaciones del usuario.

Por el contrario, usando la interfaz común que ofrece la arquitectura de gestión de la calidad de servicio (figura 1.2.2) simplemente habría que desarrollar un adaptador de la interfaz D a la interfaz común y las aplicaciones de usuario quedarían intactas.

En las figuras anteriores también se pone de manifiesto que las implementaciones realizadas no tienen porqué ser implementadas de manera nativa en la máquina en la que se ejecuta, sino que se pueden usar máquinas virtuales, como es el caso del equipo C. Este es el caso de la implementación que se realizará en este trabajo.

Otra de las ventajas del uso de una arquitectura común es que las modificaciones del código de usuario sólo han de realizarse una vez, mientras que sin la interfaz común habría que repetirla para cada una de las interfaces, con lo que los costes de desarrollo se multiplicarían.

La funcionalidad ofrecida por una arquitectura de gestión de la calidad de servicio engloba la creación, modificación y eliminación de aplicaciones de tiempo real en el sistema, así como su inicio, parada y la reconfiguración del nivel de calidad de la aplicación de manera asíncrona.

Toda esta funcionalidad básica debe tener la suficiente precisión como para poder ajustar la calidad de una tarea en concreto y a la vez tener la suficiente generalidad como para poder ajustar la calidad de una o varias aplicaciones de usuario simultáneamente.

En este sentido, la unidad mínima que una arquitectura de gestión de la calidad maneja es una tarea, estando estas tareas agrupadas en clusters, y siendo estos últimos los que componen una aplicación.

Una vez definida esta jerarquía, la arquitectura de gestión de la calidad definida puede modificar la calidad a cualquiera de los tres niveles: tarea, cluster o aplicación. El objetivo de esta jerarquía es poder reconfigurar una parte de la aplicación o la aplicación entera en

función de la carga a la que esté sometido el sistema. Para explicar esto más claramente supongamos varios ejemplos distintos.

Supongamos un caso en que existen tres aplicaciones multimedia distintas que necesitan aproximadamente el mismo número de recursos. Estas aplicaciones multimedia suelen caracterizarse por ser “avariciosas”, es decir, consumen todos los recursos hardware que se le ofrezcan devolviendo un resultado de mayor calidad. Si las tres aplicaciones están ejecutándose simultáneamente y, eventualmente, una de ellas se elimina del sistema entonces las dos aplicaciones que todavía están ejecutándose tendrán más recursos a su disposición, por lo que las dos aplicaciones se podrán reconfigurar hacia un nivel de calidad superior y que, lógicamente, requiere más recursos para proporcionarlo.

Se puede suponer otro caso en que una aplicación recibe por la red un flujo de video, lo procesa y lo muestra al usuario. Esta aplicación podría estar compuesta por dos clusters distintos, uno que se dedicará a decodificar el flujo de video que recibe y otro que se dedicara a procesarlo y mostrarlo al usuario. En el caso de que hubiera un problema de red y la calidad del flujo de video que se recibe empeorara se podría hacer una reconfiguración interna de los clusters de la aplicación de manera que se le diera menor nivel de calidad al decodificador y mayor procesador. Esta reconfiguración lograría recuperar parte de la calidad que se ha perdido en la transmisión del flujo por la red y la calidad mostrada al usuario seguiría siendo la misma.

Con estos dos ejemplos queda claro que la reconfiguración de los niveles de calidad puede ser externa, es decir, provocada por el usuario o interna, es decir, el nivel de calidad cambia a niveles inferiores para conseguir mantener el mismo nivel de calidad a nivel superior, que es el observado por el usuario.

La mayoría de las funciones de reconfiguración a niveles inferiores deben ser realizadas por la arquitectura de gestión de la calidad de una manera automática y transparente para la aplicación de usuario. Esto requiere que la arquitectura monitorice las tareas (y por lo tanto los clusters) que hay en el sistema, de manera que, de forma automática, se pueda realizar los cambios de calidad a nivel inferior que garanticen el mantenimiento del nivel de calidad a nivel superior.

Sintetizando todos los aspectos que se han mencionado se podría definir una arquitectura de gestión de la calidad de servicio como una interfaz común para aplicaciones de usuario que requieren ciertos grados de calidad de servicio que permite que una misma aplicación de usuario pueda ejecutarse de manera semejante sobre distintos sistemas y que es capaz de reconfigurar su nivel de calidad dependiendo de las preferencias del usuario o de la carga existente en los sistemas.

CAPITULO 2. TECNOLOGÍAS RELACIONADAS

1 Arquitectura de gestión de calidad de servicio

Un sistema de gestión de la calidad de servicio es, en nuestro caso, una entidad central que se encarga de asignar recursos computacionales a las aplicaciones multimedia que se ejecutan. El sistema multimedia está formado por un conjunto de aplicaciones que son programas que se ejecutan en paralelo y que pueden ser lanzadas y paradas de forma dinámica y que ofrecen un cierto resultado de presentación de datos multimedia con un número discreto de posibilidades (llamados niveles de calidad) en cuanto a dicha presentación. El sistema de gestión de la calidad de servicio deberá asegurarse de que las aplicaciones reciben de forma efectiva los recursos que necesitan, mientras que las aplicaciones deben ofrecer el nivel de calidad acordado para dicha asignación de recursos. La ejecución de todo el sistema (tanto del gestor como de las aplicaciones) se apoya en un sistema operativo de tiempo real extendido con características para contabilizar la utilización de recursos por parte de las tareas.

El objetivo de este enfoque es gestionar la ejecución de un conjunto de aplicaciones multimedia en sistema multimedia empotrados de electrónica de consumo para maximizar la calidad global. Por este motivo el gestor es un software intermedio (localizado entre el sistema operativo y las aplicaciones) que lleva a cabo sus operaciones de control de forma centralizada para conseguir la mayor rapidez en la realización de sus protocolos de gestión, evitar interferencias entre las operaciones y poder tomar decisiones de optimización global en el sistema. El sistema de gestión de la calidad de servicio gestiona de forma dinámica los recursos del sistema y los asigna a las aplicaciones. Esta asignación se basa en un modelo de contrato entre estas dos entidades, es decir, el sistema de gestión de la calidad de servicio asigna un conjunto de recursos a las aplicaciones y éstas deben ofrecer su resultado de presentación con una cierta calidad. De igual modo, éste evita que otras aplicaciones utilicen más recursos de los que se les ha asignado, si ello pone en peligro la garantía de los recursos asignados.

Para ello es necesario que el sistema de gestión de la calidad de servicio interactúe con tres elementos externos. En primer lugar interactúa con el usuario, para lanzar y parar aplicaciones y para cambiar los niveles de calidad de éstas. También las aplicaciones multimedia se comunican con el sistema de gestión de la calidad de servicio. Por último el sistema operativo de tiempo real ofrece al sistema de gestión de la calidad de servicio servicios básicos para que éste utilice técnicas de tiempo real y pueda gestionar dinámicamente los recursos del sistema.

Además la operación fundamental del sistema de gestión de la calidad de servicio se basa en la siguiente funcionalidad:

1 *Interacción con el usuario*:
El sistema de gestión de la calidad de servicio ofrece al usuario una interfaz que le permite decidir lo que el sistema debe hacer y saber cómo éste se está comportando. Las aplicaciones pueden ofrecer interfaces específicas, en cuyo caso deberán comunicar al sistema de gestión de la calidad de servicio los comandos relevantes del

usuario, que serán lanzar y parar aplicaciones, cambiar el nivel de calidad y de importancia de las aplicaciones.

2 *Negociación con las aplicaciones*:
El contexto de este trabajo significa que ambas partes acuerdan el nivel de calidad que debe ser ofrecido por una aplicación para una cierta asignación de recursos que garantiza el sistema de gestión de la calidad de servicio.

3 *Establecimiento de las nuevas configuraciones del sistema*:
Una configuración del sistema define un escenario de ejecución para el mismo. Incluye información de las aplicaciones que van a ser ejecutadas, sus niveles de calidad, tareas que van a ser ejecutadas y las necesidades de recursos para cada tarea (y, por tanto, para cada aplicación) para ofrecer el nivel de calidad seleccionado. El sistema de gestión de la calidad de servicio debe decidir la viabilidad de una configuración seleccionada por la acción del usuario o propuesta en la optimización del sistema.

4 *Contabilidad y garantía del uso de recursos*:
El sistema de gestión de la calidad de servicio debe contabilizar la utilización que se hace de los recursos del sistema para poder garantizar la asignación de los mismos a las tareas y/o aplicaciones.

5 *Monitorización del comportamiento de las aplicaciones*:
A través de la monitorización de las aplicaciones del sistema el sistema de gestión de la calidad de servicio puede emprender de forma dinámica acciones de adaptación de los niveles de calidad de las aplicaciones, dependiendo de la disponibilidad actual de recursos y de la necesidad actual de las aplicaciones.

La arquitectura que se propone en este trabajo para el sistema de gestión de la calidad de servicio se ajusta al modelo funcional descrito en los siguientes requisitos:

1 *Flexibilidad*:
La adaptación de una arquitectura inicial para una sistema de gestión de la calidad de servicio con el fin de comprobar su viabilidad con distintos tipos de aplicaciones. Para el diseño inicial se consideraron aplicaciones de audio y video, por lo que se ha prestado especial atención al desarrollo de una solución flexible (en cuanto a la facilidad de adición de nueva funcionalidad) y abierta, donde la integración de distintos tipos de aplicaciones es posible.

2 *Separación de políticas y mecanismos*:
El diseño intenta separar estos dos conceptos. La arquitectura ofrece los mecanismos básicos. Las políticas no son parte de la arquitectura, pero pueden ser integradas fácilmente en el marco arquitectónico. De esta forma es fácil experimentar con diferentes aproximaciones a temas como cambios de modo, optimización y gestión de errores.

3 *Composición*:
La arquitectura ha sido diseñada de tal forma que pueda ser usada de forma parcial, dependiendo de las necesidades del sistema desarrollado o del nivel de abstracción que se desea tener. Por ejemplo, si existe un sistema con sólo una aplicación (por

ejemplo VTR, donde el audio y el video son gestionados de forma centralizada), puede ser conveniente usar las partes de la arquitectura que son más específicas del tratamiento de tareas y asignación básicas de recursos y evitar las partes de propósito general. De esta forma sería posible que el sistema utilice componentes sustitutos que son específicos para sus aplicaciones y, por lo tanto, ofrecen mejores resultados. En este trabajo, y como se muestra en la figura 2.1.1 las dos capas más bajas de la arquitectura podrían utilizarse para gestionar cuotas de uso de recursos.

Se identificaron tres entidades conceptuales que son fundamentales para el sistema: aplicación, nivel de calidad y cuota de uso de recursos. Estas entidades son susceptibles de ser gestionadas de forma dinámica y sobre ellas se pueden tomar decisiones en el proceso de monitorización. Cuando el sistema de gestión de la calidad de servicio debe tomar decisiones para mejorar el comportamiento del sistema, éste debe modificar la configuración del sistema. Ello puede ser realizado cambiando las decisiones respecto a cualquiera de las tres entidades. Por ejemplo, si una no cumple con el nivel de calidad que se ha acordado la acción que puede emprenderse es aumentar la asignación de cuota de uso de recursos para ella. Si esto no es suficiente puede reducirse el nivel de calidad de la aplicación y finalmente, si esta acción tampoco resuelve el problema, la decisión puede consistir en parar la aplicación. Este hecho aconseja la utilización de un diseño por niveles o capas como se muestras en la figura 2.1.1.

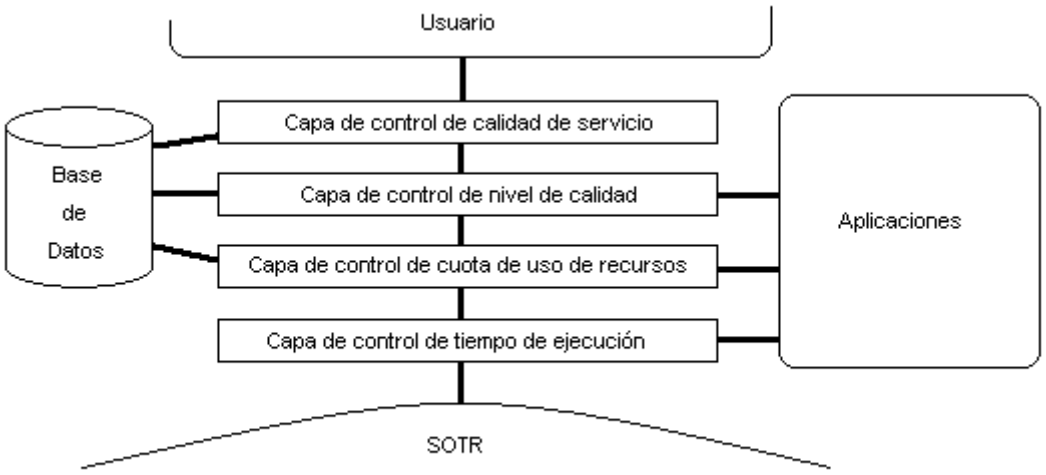

Figura 2.1.1 Arquitectura del sistema de gestión de la calidad de servicio

En la figura 2.1.1 se muestra la arquitectura del sistema de gestión de la calidad de servicio. Es una arquitectura de cuatro niveles, cada uno de los cuales maneja niveles de abstracción distintos dependiendo de la entidad que gestiona. Los tres niveles superiores (o de mayor nivel de abstracción) gestionan una entidad distinta. El nivel de gestión de la calidad de servicio realiza su operación sobre la entidad *aplicación*. El nivel de gestión de niveles de calidad realiza su operación sobre la entidad *nivel de calidad*. El nivel de gestión de cuotas de recursos realiza su operación sobre la entidad *cuota de recursos*. El nivel de control de ejecución se encarga de contabilizar la utilización de los recursos y de garantizar el uso de los mismos. Este nivel interactúa con el sistema operativo de tiempo real y el hardware para este fin. La base de datos contiene información del sistema, como las aplicaciones que se ejecutan en él, sus niveles de calidad y la configuración actual de cada entidad relevante.

El diseño interno de las tres capas superiores sigue un patrón común. Cada nivel gestiona una entidad y realiza operaciones similares, pero con un nivel de abstracción semejante.

2 RTSJ

RTSJ es el acrónimo correspondiente a la Especificación de Tiempo Real para Java (Real-Time Specification for Java). Esta especificación fue producida para extender la especificación del lenguaje Java y la especificación de la máquina virtual de Java y para proveer una Interfaz de Programación de Aplicación que permitiera la creación, verificación, análisis, ejecución y gestión de hilos Java cuyas condiciones de corrección incluyeran restricciones de tiempo (también conocida como hilos de tiempo real).

Esta especificación fue creada en base a siete principios guía que delimitaron el ámbito de trabajo e introdujeron los requerimientos de compatibilidad para la especificación de tiempo real para Java. Los siete principios guía fueron los siguientes:

- **Aplicabilidad a Entornos Java Particulares**
  La RTSJ no incluye especificaciones que restringen su uso en entorno Java particulares, tales como una versión particular del Kit de Desarrollo Java, un entorno de aplicación Java empotrado o la Java 2 Micro Edition.

- **Compatibilidad hacia atrás**
  La RTSJ no excluye la ejecución de programas previos Java sin características de tiempo real en las implementaciones de RTSJ.

- **"Escribe una vez, ejecuta en cualquier sitio"**
  La RTSJ reconoce la importancia de este lema de Java, pero también reconoce la dificultad de su consecución para programas de tiempo real y no intenta incrementar o mantener la portabilidad del binario a expensas de la predictibilidad.

- **Practicas actuales vs. características avanzadas**
  La RTSJ tiene en cuenta las prácticas actuales en materia de tiempo real y a la vez permite que implementaciones futuras incluyan características avanzadas.

- **Ejecución Predecible**
  La RTSJ soporta la ejecución predecible como una prioridad principal. Esto puede ser costoso en las medidas de prestaciones en la computación típica de propósito general.

- **Sin extensiones sintácticas**
  Para facilitar el trabajo de los desarrolladores de herramientas y, por lo tanto, para incrementar la probabilidad de implementaciones la RTSJ no introduce nuevas palabras reservadas ni hace otras extensiones sintácticas al lenguaje Java.

- **Permitir Variación en las decisiones de Implementación**
  La implementación de la RTSJ puede variar en un número de decisiones de implementación, tales como el uso de algoritmos eficientes o ineficientes o la

inclusión de algoritmos de planificación no requeridos en la implementación mínima. Por lo tanto la RTSJ no impone algoritmos o constantes de tiempo específicos pero requiere que la semántica de la implementación se cumpla.

Basándose en estos siete principios guía se hallaron siete áreas de mejora de la especificación del lenguaje Java para crear la RTSJ que se listan a continuación.

2.1 Planificación de Hilos

A la luz de la significante variedad de modelos de planificación y ante el reconocimiento de que cada modelo tiene una amplia aplicabilidad en los diversos sistemas de tiempo real de la industria la RTSJ permite que la especificación de la planificación que admita un mecanismo de planificación usado por defecto por los hilos de tiempo real Java pero que no implica ninguna ventaja respecto a otro u otros posibles mecanismos de planificación. La especificación fue construida para permitir que las implementaciones proveyeran otros algoritmos de planificación. Las implementaciones permiten, por lo tanto, la asignación programática de los parámetros apropiados para el mecanismo de planificación usado, así como los métodos necesarios para la creación, gestión, admisión y terminación de los hilos de tiempo real Java. También se espera, por ahora, que un mecanismo de planificación de hilos en particular esté ligado a una implementación. Sin embargo existe flexibilidad suficiente en la estructura de planificación de hilos para permitir que futuras versiones de la especificación permitan la carga dinámica de políticas de planificación.

Actualmente la RTSJ requiere un planificador base en todas sus implementaciones que está basado en prioridades y es expulsivo.

Una de las preocupaciones de la programación en tiempo real es asegurar la ejecución predecible de secuencias de instrucciones máquina. Una ejecución predecible de un hilo implica que un programador puede determinar, mediante análisis del programa o mediante pruebas, si hilos en particular completarán siempre su ejecución antes de que se cumpla una restricción temporal.

El término planificación se refiere a la producción de una secuencia (u ordenación) de la ejecución de un conjunto de hilos. Esta planificación intenta optimizar una métrica particular (una métrica que mide cómo de bien el sistema está cumpliendo las restricciones temporales). Un análisis de factibilidad (feasibility) determina si un hilo tiene un valor aceptable para la métrica.

Muchos sistemas usan la prioridad de los hilos como un intento para determinar la planificación. La prioridad es, típicamente, un entero asociado a un hilo, estos enteros comunican al sistema el orden en que los hilos deberían ejecutarse. El concepto generalizado de prioridad es la *elegibilidad de la ejecución*. El término expedición (dispatching) se refiere a la porción del sistema que selecciona un hilo con el nivel más alto de elegibilidad de ejecución del común de hilos que están preparados para ejecutarse.

En la práctica, en los sistemas de tiempo real actuales la asignación de prioridades está bajo el control del programador, en oposición con los sistemas en que la asignación de prioridades está bajo el control del sistema. La RTSJ deja la asignación de prioridades al

programador. Los algoritmos de factibilidad asumen que se ha usado un algoritmo de asignación de prioridades monotónico, aunque la RTSJ no requiere que las implementaciones verifiquen que tal asignación es correcta. Si, desde luego, la asignación de prioridades es incorrecta el análisis de factibilidad no tendrá sentido.

En resumen, la RTSJ se ha desarrollado para permitir a los implementadores flexibilidad para instalar algoritmos de planificación y de factibilidad arbitrarios en una implementación de la especificación. Con esto se puede hacer frente a la gran variabilidad en los requerimientos de la industria de los sistemas de tiempo real.

2.2 Gestión de la Memoria

La gestión automática de la memoria es una característica particularmente importante del entorno de programación de Java y se buscó una solución que permitiera, tanto como fuera posible, que el trabajo de la gestión de memoria fuera implementado automáticamente por el sistema y no interfiriera en las tareas de programación.

Existen muchos algoritmos de gestión de la memoria, también conocidos como recolección de basura, y muchos de ellos se aplican a ciertas clases de sistemas y estilos de programación en tiempo real. RTSJ ha buscado una manera de definir una gestión y liberación de la memoria que tuviera las siguientes características:

- Independencia del algoritmo de recolección de basura.
- Debe permitir caracterizar con precisión el efecto de un algoritmo de recolección de basura implementado sobre el tiempo de ejecución, sobre la preempción y sobre los hilos de tiempo real.
- Debe permitir la gestión y liberación de los objetos sin ninguna interferencia con ningún algoritmo de recolección de basura.

Aunque no es una materia de tiempo real, el acceso a la memoria física es deseable por muchas de las aplicaciones que podrían hacer un uso productivamente de una implementación de la RTSJ. Por lo tanto, hay definida una clase que permite a los programadores acceder a la memoria física a nivel de byte y otra que permite la creación de objetos en la memoria física.

Las pilas de memoria con recolección de memoria siempre han sido consideradas un obstáculo para la programación de sistemas de tiempo real debido a las latencias impredecibles introducidas por el recolector de basura. La RTSJ soluciona este tema proveyendo varias extensiones al modelo de memoria, que soporta la gestión de la memoria de una manera que no interfiere con la habilidad del código de tiempo real para proveer un comportamiento determinista. Esta meta se consigue permitiendo el alojamiento de objetos fuera de la pila de objetos con recolección de memoria tanto para objetos de corto tiempo de vida como para objetos de largo tiempo de vida.

La RTSJ presenta el concepto de *área de memoria*. Un área de memoria representa una región de la memoria que puede ser usada para alojar objetos. Algunas áreas de memoria existen fuera de la pila y sufren restricciones sobre lo que el sistema y el recolector de basura pueden hacer en ellas. Los objetos en algunas áreas de memoria nunca sufren la recolección de memoria, sin embargo, el recolector de basura debe ser capaz de escanear

esas áreas para obtener las referencias a los objetos que residen allí y mantener la integridad de la pila.

Existen cuatro tipos básicos de áreas de memoria:

- Memoria de Ámbito (Scoped Memory): provee un mecanismo para tratar las clases de objetos que tienen un tiempo de vida definido por el ámbito sintáctico.
- Memoria Física (Physical Memory): permite que se creen objetos dentro de regiones de memoria física específicas que tienen características particulares importantes, como por ejemplo, un acceso sustancialmente más rápido.
- Memoria Inmortal (Immortal Memory): es un área de memoria que contiene objetos que, una vez alojados, existen hasta el final de la aplicación, por ejemplo los objetos singleton.
- Memoria de Pila (Heap Memory): representa un área de memoria que es la pila. La RTSJ no cambia el tiempo de vida de los objetos en la pila.

2.3 Sincronización y Compartición de Recursos

Con frecuencia es necesario compartir recursos serializables. En estos entornos los sistemas de tiempo real introducen una complejidad adicional: la inversión de la prioridad.

Para evitar la inversión de la prioridad la RTSJ usa el protocolo de herencia de la prioridad. Este protocolo funciona de la siguiente manera. Si un hilo t_1 con una prioridad determinada intenta adquirir un *lock* que ya ha adquirido otro hilo t_2 con prioridad más baja entonces la prioridad de t_2 se eleva hasta la de t_1 sólo mientras t_2 tenga el control de *lock* (y recursivamente si el hilo con prioridad t_2 está esperando para adquirir el *lock* que posee otro hilo de prioridad aún menor).

RTSJ ha desarrollado una especificación que intenta ser lo menos intrusiva posible para permitir una sincronización de tiempo real segura. Para ello se ha redefinido la palabra clave *synchronized* para que incluya uno o varios algoritmos que eviten la inversión de prioridad entre los hilos de tiempo real que comparten los recursos serializados.[2]

En algunos casos la implementación requerida del algoritmo de inversión de prioridades no es suficiente para evitarla y simultáneamente permitir que un hilo tenga una elegibilidad lógica mayor que el recolector de basura por ello se han implementado ciertas clases de colas libres de esperas para estas situaciones.

2.4 Manejo de Eventos Asíncrono

Los sistemas de tiempo real típicamente interactúan muy estrechamente con el mundo real. El mundo real es asíncrono, por lo tanto RTSJ incluye mecanismos eficientes para disciplinas de programación que puedan acomodarse a esta asincronía inherente. RTSJ generaliza el mecanismo del lenguaje Java del manejo de eventos asíncrono. Las clases requeridas representan entidades que pueden ocurrir y lógica que se ejecuta cuando estas entidades ocurren. Una característica notable es que la ejecución de la lógica está planificada por el mismo planificador que maneja las tareas de usuario.

A veces el mundo real cambia tan drásticamente (y asíncronamente) que el punto actual de la lógica de ejecución debería ser inmediata y eficientemente transferido a otra localización. RTSJ incluye un mecanismo que extiende el manejo de excepciones Java que permite a las aplicaciones cambiar, programáticamente, la localización del control de otro hilo Java. Es importante notar que RTSJ restringe esta transferencia asíncrona de control a lógica escrita específicamente bajo la asunción de que esta localización del control puede cambiar asíncronamente.

Debido a estos cambios drásticos y asíncronos en el mundo real la lógica de la aplicación puede necesitar un realineamiento del hilo de tiempo real Java mediante una transferencia de su control expeditiva y segura hacia una terminación de una manera normal. Hay que notar que, al contrario que con el mecanismo tradicional, inseguro y descartado de Java para la parada de hilos, el mecanismo de RTSJ es seguro.

3 Maquina Virtual TimeSys

La máquina virtual de TimeSys fue la primera máquina virtual en implementar la especificación de tiempo real para Java (RTSJ). La TimeSys VM implementa la versión 1.0.2 de la RTSJ. Esta implementación está realizada en la distribución Red Hat 9.0 de Linux sobre un kernel 2.4.26 modificado. La versión de la máquina virtual de TimeSys que se ha usado durante todo este trabajo ha sido la versión 1.3.

La RTSJ define varios tipos de hilos de tiempo real, los cuales han sido implementados desarrollando una librería nativa basada en los hilos POSIX y que han llamado ***libpthreadrt***.

La implementación de la gestión de memoria también se ha realizado usando librerías nativas, pero existen varios tipos de memoria que no funcionan exactamente como se supone que deberían hacerlo según se especifica en la RTSJ.

La instalación de esta máquina virtual requiere la recompilación del kernel 2.4.26 con una serie de módulos que han sido modificados o directamente desarrollados para dar soporte a la funcionalidad adicional que requiere la RTSJ.

Los pasos necesarios para la instalación de la máquina virtual de TimeSys en un sistema Red Hat 9.0 previamente instalado son los siguientes:

- Compilación del kernel modificado 2.4.26 que se distribuye con la máquina virtual. Se recomienda activar todas las opciones en la subsección TimeSys en el paso de la compilación ***make config***, ***make menuconfig*** o ***make xconfig***.

- Una vez que se haya recompilado el kernel y el sistema se pueda arrancar correctamente con el nuevo kernel instalar la versión de la máquina virtual TimeSys. Por defecto la instalará en el directorio ***/opt/timesys/rtsj_ri_1.3***

- Hacer un enlace simbólico de la librería libpthreadrt.so.3.1 que se halla en el directorio *lib* del directorio de instalación a la carpeta ***/lib***

- La máquina virtual se encuentra en la carpeta bin del directorio de instalación. El binario se llama ***tjvm***.

Para ejecutar correctamente el binario se debe establecer antes la variable de entorno ***LD_ASUME_KERNEL*** y en la invocación se deben definir las variables de inicialización del tamaño de las regiones de memoria. En el Anexo I se muestra el código del script que se ha usado para ejecutar esta máquina virtual en las pruebas.

4 Interfaz de Gestión de Recursos para Java

La interfaz de gestión de recursos para Java (resource management interface for Java – RM API) se desarrolló con el objetivo de evitar ciertas lagunas existentes en la implementación de la máquina virtual de Java que impedían la monitorización de recursos y ciertas estrategias de balance de carga.

La RM API provee funcionalidad para gestionar entidades aisladas. Cada una de estas entidades aisladas es una aplicación Java que no comparte ningún estado con el resto de las entidades aisladas del sistema.

La RM API modela cada recurso como un conjunto de atributos de recurso. Éstos definen las propiedades del recurso y su implementación. El puente entre la interfaz de gestión de recursos y la implementación del recurso es el *dispensador*. El dispensador monitoriza la cantidad disponible de recurso para procesamiento.

Cada recurso tiene definida una política de consumo del recurso, que determina en que condiciones y en qué momento se ha consumido la cuota del recurso solicitada por cada una de las aplicaciones. Una vez que la cuota de un recurso solicitada por una aplicación se ha consumido el recurso informa al dispensador de ello y el dispensador informa a su vez a la aplicación de que su cuota ha expirado.

Como se ha mencionado anteriormente la RM API sólo gestiona recursos de entidades aisladas, de manera que no es factible mediante ella gestionar varias aplicaciones interrelacionadas. En el caso de los sistemas multimedia esta condición no se cumple ya que una aplicación puede influir en el estado de calidad de otra de las aplicaciones si el número de recursos del sistema disponible no es suficiente para proveer el nivel de calidad deseado para ambas aplicaciones.

Esta interfaz es sólo una API a nivel del lenguaje Java, sin embargo, el propósito de este trabajo es la implementación de una API sobre el sistema operativo y la máquina virtual, en este caso situada al mismo nivel que la máquina virtual de Java.

Las dos últimas limitaciones mencionadas hacen que la RM API no sea adecuada para la implementación de un sistema de gestión de la calidad de servicio sobre un sistema operativo de tiempo real.

5 Máquina Virtual Jamaica

La Máquina Virtual Jamaica es una implementación de la especificación de la Máquina Virtual de Java. Esta implementación provee un entorno de ejecución de tiempo real para aplicaciones escritas en el entorno Java 2. Ha sido diseñado para sistemas embebidos y de tiempo real y ofrece el siguiente soporte de estas funciones:

- Garantiza la ejecución en *hard realtime*.
- Soporta la especificación de tiempo real para Java (RTSJ)
- Soporta código nativo
- Portabilidad

Jamaica es la única implementación que provee hard realtime en todas las características de Java junto con unas altas prestaciones en tiempo de ejecución.

Todos los hilos ejecutados por la máquina virtual Jamaica son hilos de tiempo real, de manera que no hay necesidad de distinguir hilos de tiempo real e hilos estándar. El sistema soporta una implementación expulsiva por lo que un hilo de alta prioridad es capaz de apropiarse del procesador antes que un hilo de baja prioridad o incluso expulsarlo del procesador para acceder a él. No hay restricciones en el uso del lenguaje Java, por lo que no se requiere una especial precaución a la hora de codificar los programas Java.

La máquina Virtual Jamaica soporta la versión 1.0.2 de la especificación RTSJ y la versión 1.2 de la interfaz nativa de Java (JNI). El soporte de la RTSJ permite la compatibilidad del código Java escrito para otros sistemas de tiempo real mientras que soporte de JNI provee la capacidad de incluir código nativo en las aplicaciones Java que permite el acceso más sencillo a los dispositivos hardware y codificar de forma nativa las partes de código más críticas por su eficiencia.

Durante el desarrollo de código para Jamaica hay que tener especial cuidado para reducir los esfuerzo de portabilidad de al mínimo. Jamaica ha sido implementado en C usando un compilador de C GNU por lo que cualquier sistema que soporte este tipo de compatibilidad es susceptible de soportar Jamaica. Los hilos están basados en los hilos nativos del sistema operativo, que en la gran mayoría de los sistemas son hilos POSIX.

Actualmente Jamaica está disponible para las siguientes plataformas de desarrollo:

- Linux
- SunOs/Solaris
- Windows

Y genera ejecutables para los siguientes sistemas operativos de tiempo real:

- eCos
- embCos
- INTEGRITY
- Linux/RT
- NetOS
- OS-9
- PikeOS
- QNX
- RTEMS
- ThreadX
- WinCE
- VxWorks

Cada sistema operativo soportado se define como un *target*. Es conveniente leer en la documentación las funcionalidades y/o peculiaridades de cada target, ya que hay ciertas funciones que ofrecen las API de Jamaica que funcionan de forma distintas según el target elegido. Un ejemplo de esto es la implementación del target VxWorks. En la documentación de este target se puede encontrar una limitación muy importante:

java.lang.Runtime.exec() is not implemented

El funcionamiento de Jamaica a la hora de generar el código para cada uno de los targets se muestra en la figura 2.5.1. En esta figura se muestra cómo el código Java, mediante la herramienta Jamaica Builder, se transforma en código C nativo que posteriormente se compila y enlaza generando código nativo en el target seleccionado[3].

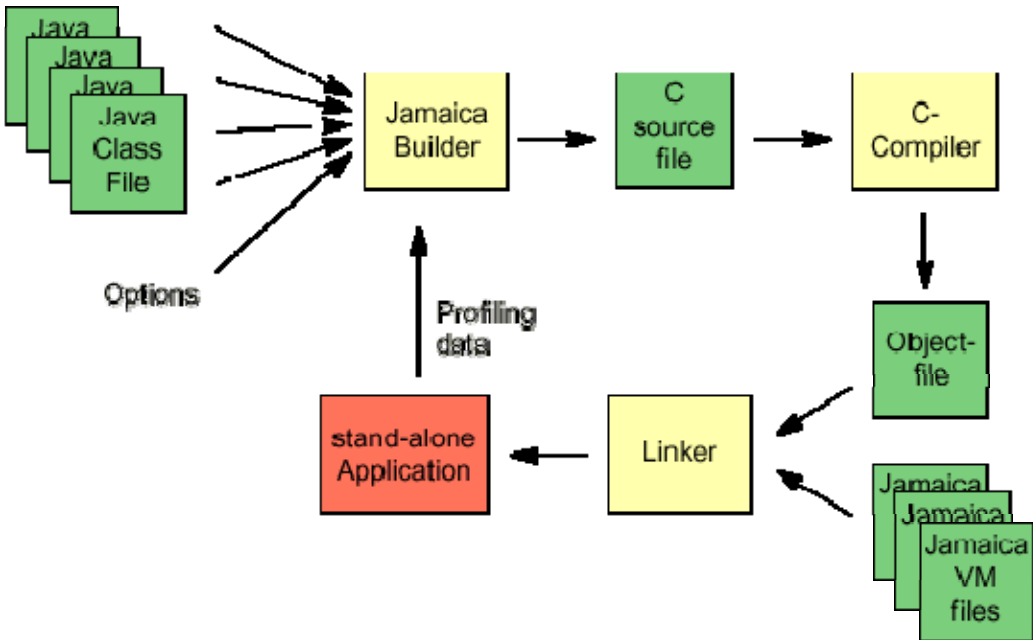

Figure 2.5.1 Proceso de generación de código objeto en Jamaica

Hay que destacar algo muy importante en este proceso de generación. Las clases Java se transforman en código nativo, por lo que la carga dinámica de clases codificada en una clase Java estándar deja de funcionar. Es decir, si se ejecuta la siguiente linea de código en una de las clases java de entrada se lanzará la excepcion ClassNotFoundException.

Class.forName(“Example”);

Los cálculos de tiempo de CPU de cada uno de los hilos del sistema se puede realizar de forma muy sencilla con Jamaica usando la API específica que ofrece. Existen dos funciones con las que se puede saber el número de ciclos de CPU que un hilo ha consumido durante su ejecución. Las funciones son:

```
com.aicas.jamaica.lang.CpuTime.getCpuTime(this)
com.aicas.jamaica.lang.CpuTime.getTotalCpuTime(this)
```

La primera función devuelve el número de ciclos de CPU de código no nativo ejecutado por el hilo que se pasa como parámetro. El segundo devuelve el número de ciclos de CPU de código nativo y no nativo ejecutado por el hilo que se pasa como parámetro.

En este proyecto se utiliza Jamaica generando código nativo para el sistema operativo de tiempo real QNX Neutrino RTOS, cuyas características principales se mencionan en el siguiente apartado.

6 Sistema Operativo QNX

QNX es un sistema operativo de tiempo real basado en Unix que cumple con la norma POSIX. Está desarrollado principalmente para su uso en dispositivos empotrados. Está disponible para las siguientes arquitecturas:

- x86
- MIPS
- PowerPC
- SH4
- ARM
- StrongARM
- xScale

QNX está basado en una estructura de microkernel y se puede definir como un sistema operativo de tiempo real, expulsivo, priorizado, distribuido, multitarea, multiusuario y tolerante a fallos.

El microkernel de QNX está basado en la idea de ejecutar la mayor parte del sistema operativo en forma de un número de pequeñas tareas, conocidas como servidores. Esto difiere de la forma tradicional de kernel (kernel monolítico), en la que el sistema operativo es un único y gran programa compuesto de una enorme cantidad de módulos con capacidades especiales. En el caso de QNX el uso del microkernel permite al usuario desactivar cualquier funcionalidad del sistema operativo que no requerida ser modificada deteniendo la ejecución de determinados servidores.

El kernel QNX contiene únicamente la planificación de tareas, la comunicación entre procesos, las interrupciones y los temporizadores. Todo se ejecuta como procesos de usuario, incluido un proceso especial llamado *proc* que realiza la creación de procesos y la gestión de memoria en conjunción con el microkernel.

La comunicación entre procesos de QNX consiste en el envío de mensajes síncrono. Esta se realiza en un única operación llamada *MsgSend*. El mensaje es copiado por el kernel, desde el espacio de direcciones del proceso que lo envía al espacio de direcciones del proceso que lo recibe. Si el proceso receptor está esperando el mensaje se transfiere el control del procesador en ese momento sin pasar por el planificador de tareas. Esto se traduce en que el envío de un mensaje a un proceso que está esperando no da como resultado la “pérdida de turno” para el procesador. Esta integración entre paso de mensaje y planificación del procesador es uno de los mecanismos claves que hacen que el paso de mensajes en QNX sea ampliamente usado. Todas las operaciones de entrada/salida, con ficheros o de red funcionan mediante este mecanismo y los datos transferidos son copiados mediante el paso de mensajes.

QNX soporta *symmetric multiprocessing* y *bound multiprocessing*, que hace posible restringir la ejecución de un proceso a un procesador. Esto mejora la latencia de la caché y facilita la migración de aplicaciones no-SMP a sistemas multiprocesador. QNX soporta la planificación preemtiva de prioridades estricta y la partición de planificación adaptiva (APS).

CAPITULO 3. GESTOR DE QoS INDEPENDIENTE DE LA PLATAFORMA

1 Arquitectura del Gestor

La arquitectura de un gestor de la calidad de servicio está determinada, principalmente, por las características de los sistemas que debe gestionar, de manera que debe ser capaz de tener la misma capacidad para gestionar todos los tipos de sistema soportados.

La arquitectura del gestor de la calidad de servicio[3] está diseñada para soportar distintos sistemas y las aplicaciones que puedan ejecutarse en ellas. Esta arquitectura, por lo tanto, no tiene que ser implementada completamente si va a usarse solamente en uno de los tipos de sistemas que están soportados.

El gestor de calidad de servicio se pensó para distintos tipos de sistemas, como pueden ser los sistemas multimedia, los sistemas de tiempo real y los sistemas de electrónica de consumo.

La característica más destacada de los sistemas multimedia es la capacidad de procesar datos de distinta naturaleza: video, audio, gráficos, etc El procesamiento suele realizarse de forma continua, es decir, la entrada que reciben es un flujo continuo de datos que deben procesar de forma constante. La salida de estos sistemas presenta restricciones temporales de forma que existe una frecuencia preestablecida para la salida que éstos ofrecen y que es fijada por los estándares de la industria (cuyo origen radica en estudios de la tolerancia del usuario final a la frecuencia de presentación de datos multimedia).

Los sistemas multimedia de propósito general suelen centrarse en los niveles más bajos de la problemática de ejecución de aplicaciones, es decir, tratan únicamente temas de planificación y de nivel de sistema operativo. La utilización eficiente de los recursos y la limitación en el número de los mismos no es un problema, ya que trabajan con estaciones de trabajo potentes.

Los *sistemas de tiempo real* presentan restricciones temporales que deben cumplirse para su correcto funcionamiento. Pueden críticos o acríticos según sus plazos sean o no de cumplimiento obligatorio. Los sistemas multimedia son de naturaleza acrítica.

Los sistemas de electrónica de consumo suelen ser sistemas embarcados con recursos limitados que deben ser robustos si quieren tener éxito en el mercado ya que son productos comerciales. Esto implica que deben mantener un funcionamiento adecuado durante todo el tiempo de funcionamiento del mismo. Además, dado que son productos fabricados en masa la economía de los recursos es un problema decisivo, de forma que en ellos es necesario incluir la cantidad de recursos que sea estrictamente necesaria. Esto implica que los recursos tendrán un uso intensivo por parte de las aplicaciones y deberían ser utilizados de forma rentable.

1.1 Estructura del sistema de gestión de la calidad de servicio

La estructura del sistema de gestión de la calidad de servicio define tres tipos distintos de tareas, las tareas periódicas, continuas e imprecisas. Las tareas continuas son aquellas que se activan por datos de entrada y que, por lo tanto, se ejecutan continuamente mientras reciban una entrada que procesar. Las tareas periódicas son aquellas cuya activación se realiza con una periodicidad fija. Tienen un plazo de tiempo en el que deben ejecutarse cada vez que se activan. Las tareas imprecisas son aquellas cuya salida mejora con la cantidad de recursos que se les asigna. Éstas suelen tener dos partes, la obligatoria y la opcional. Si consiguen ejecutar la parte obligatoria se considera que el resultado el aceptable y si, adicionalmente, se consigue ejecutar la parta opcional se considera que el resultado es preciso.

Las tareas típicas que forman parte de una cadena de procesamiento en aplicaciones multimedia generalmente son una combinación entre tareas periódicas e imprecisas. Son tareas que se ejecutan siempre que tienen datos de entrada listos y en las que la calidad del resultado que ofrecen depende de los recursos asignados. Esto es debido a que no es posible determinar cuál sería la parte obligatoria y cuál la opcional para una tarea semejante, ya que los algoritmos de procesamiento de datos multimedia pueden aplicar innumerables iteraciones sobre los datos.

Las tareas típicas de procesamiento de datos multimedia tienen necesidades de recursos variables que dependen fuertemente de las características de los datos de entrada en cada momento. Sin embargo, es posible obtener estimaciones de sus necesidades de recursos a través de simulaciones y del conocimiento de los expertos de aplicación. Por lo tanto las tareas reciben una asignación de cuota de recursos que puede corresponder a un valor algo mayor que la estimación del tiempo medio. Esta cuota se asigna a una tarea para cada período de tiempo *T* en que dicha cuota es restablecida.

Dado que el plazo de tiempo de una tarea individual viene marcado por el plazo de tiempo de la aplicación completa (plazo extremo a extremo) el plazo de una tarea puede ser flexible. Es decir, el incumplimiento del plazo de una tarea puede ser compensado por el cumplimiento del plazo de la tarea siguiente en la cadena de procesamiento. Además, por este motivo a varias tareas se les puede asignar un mismo valor de cuota de utilización de recursos. Esta es la base de los clusters o agrupaciones de tareas. De esta forma las aplicaciones individuales pueden tener un plazo igual o superior al período. Por tanto los grupos de tareas individuales permiten asignar cuotas de uso a varias tareas y no sólo a una tarea individual y permite también la compensación de fallos de plazo de tareas individuales y también compensación de uso de cuotas de recursos.

Las prioridades que se asignan a las tareas también influyen en el nivel de servicio que una tarea puede alcanzar. Todos los sistemas operativos manejan distintos niveles de prioridad en la ejecución de procesos e hilos y tenerse en cuenta en el diseño de un sistema que operará en tiempo real.

En las aplicaciones multimedia la prioridad de una tarea depende principalmente de dos factores: la importancia de la aplicación a la que pertenece y la importancia de la tarea dentro de la aplicación. La importancia de una tarea dentro de la aplicación se relaciona con su período y su posición dentro de la cadena de procesamiento de la aplicación.

1.2 Modelo Estructural

Como se ha comentado antes existe una clara separación de las responsabilidades que competen tanto al sistema de gestión de la calidad de servicio como a las aplicaciones. Por tanto no es necesario que el sistema conozca la actividad específica de las aplicaciones ni su estructura interna detalladamente, es decir, no incorpora conocimiento sobre la semántica de las aplicaciones. Sin embargo, sí debe tener conocimiento de ciertos datos de las mismas que necesita saber para poder llevar a cabo su gestión de la ejecución de las mismas en el sistema. Estos datos del sistema son:

- Los distintos niveles de calidad que puede ofrecer una aplicación.

- La estructura de tareas de las aplicaciones y sus necesidades de recursos para los distintos niveles de calidad.

- La importancia relativa de las aplicaciones multimedia que se ejecutan en el sistema.

Toda esta información constituye el lenguaje que comparten el sistema de gestión de la calidad de servicio y las aplicaciones y que deben utilizar para intercambiar información en su interacción.

La información necesaria para obtener el modelo de una aplicación se basa en la definición del conjunto de entidades que se detallan a continuación (figura 3.1.2.1) y de las relaciones entre ellas (figura 3.1.2.2) que constituye el modelo estructural del sistema.

- **Aplicación**: Una entidad de programa que es capaz de procesar datos multimedia y ofrecer un resultado final al usuario.

- **Nivel de Calidad**: Es cada una de las distintas posibilidades que ofrece una aplicación en su resultado final y que el usuario es capaz de distinguir y seleccionar. Por ejemplo, para una aplicación de audio corresponderían a combinaciones de filtros como: reducción del ruido, sonido estéreo y modo surround.

- **Configuración de Nivel de Calidad**: Cada una de las formas de implementar físicamente un nivel de calidad. Es decir, un mismo nivel de calidad puede ser implementado con la aplicación de distintas combinaciones de filtros o mediante la utilización de distintas combinaciones de recursos del sistema.

- **Tarea**: Es un hilo de ejecución o mínima unidad de ejecución del sistema. Una tarea pertenece a una aplicación y forma parte de un grupo determinado. Las tareas de una aplicación multimedia operan en el procesamiento de los datos de dicha aplicación para ofrecer un resultado final de la misma

- **Recurso**: Es uno de los dispositivos del hardware por el que compiten las tareas para poder ejecutarse. Ejemplos de recursos son el procesador, la memoria, el bus y los coprocesadores.

- **Cuota de Uso de Recurso**: Es un valor que indica cuál es la utilización permitida sobre un recurso determinado o cantidad de recurso que puede ser consumida. Dicha

cuota es asignada a un grupo de tareas o a tarea individual. En el modelo inicial del sistema éste se presenta de tal forma que se asignan cuotas de uso sólo para el procesador aunque pueden ser asignadas para la utilización de cualquier otro recurso.

- **Configuración de Tarea**: Es la asignación de cuotas de recursos a una tarea. Las cuotas que se asignan son una por recurso al que accede la tarea. En este sentido las cuotas asignadas a tareas pueden usarse como cuotas de referencia para controlar el acceso a los recursos de las distintas tareas que componen un grupo de tareas.

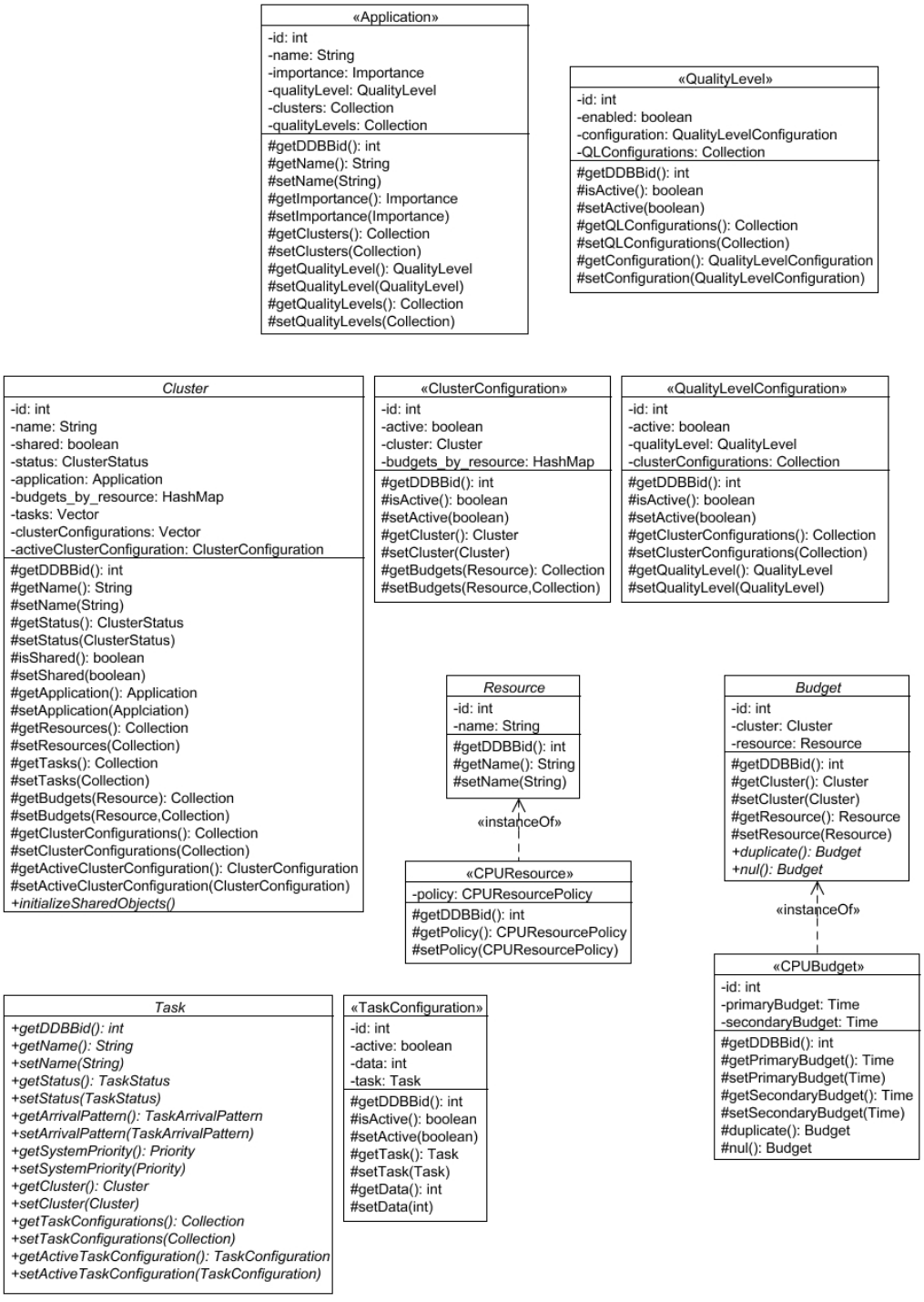

Figura 3.1.2.2 Diagrama UML de las entidades

- **Grupo de Tareas**: Es una agrupación de tareas que presentan alguna relación en la cadena de flujo de datos de la aplicación a la que pertenece.

- **Configuración de Grupo de Tareas**: Es la asignación de cuotas de uso de recursos a un grupo de tareas. Las cuotas que se asignan son una por recurso al que accede dicho grupo.

Las relaciones entre las entidades que se acaban de mencionar se muestra en el siguiente diagrama:

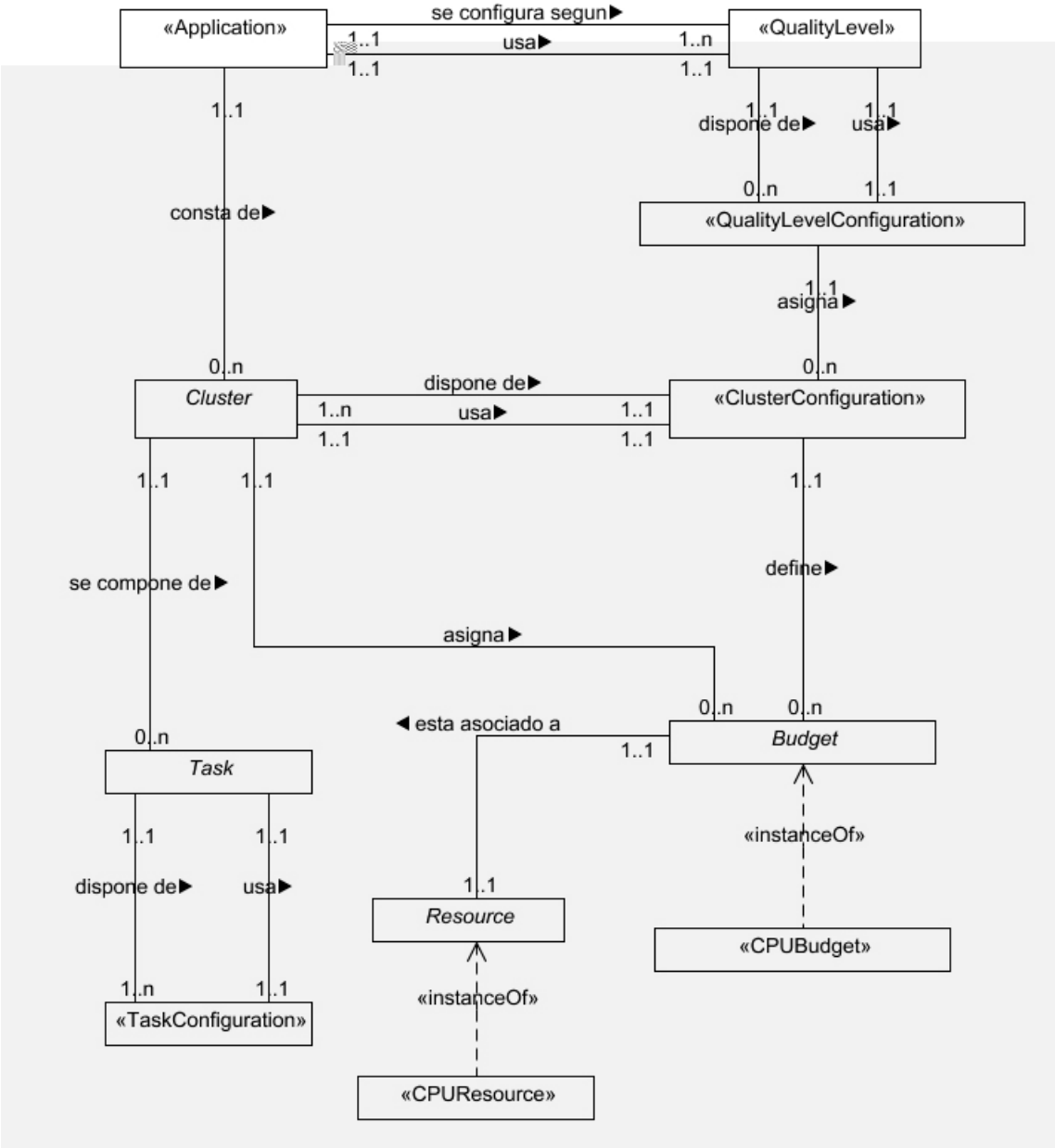

Figura 3.1.2.2 Diagrama UML de las relaciones entre entidades

Conforme a estos requisitos básicos que debe cumplir el sistema de gestión de la calidad de servicio se ha creado una arquitectura en cuatro niveles para el este sistema que se muestra a continuación.

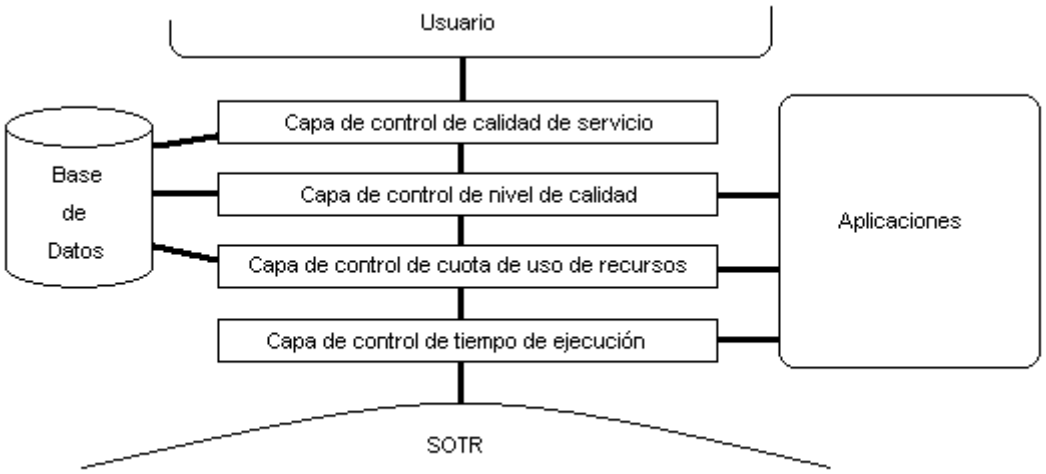

Figura 3.1.2.3 Arquitectura del sistema de gestión de la calidad de servicio

Como se puede ver en la figura 3.1.2.3 la base de datos contiene el modelo de todo el sistema, es decir, la descripción de todas las aplicaciones y sus parámetros necesarios, como son los niveles de calidad a los que pueden funcionar una aplicación, etc. Los niveles de la arquitectura, a excepción del nivel de control de tiempo de ejecución, que se mantiene independiente, acceden a la base de datos (al modelo completo del sistema) para llevar a cabo su operación.

La interacción de la arquitectura con el usuario y las aplicaciones se realiza de forma distinta. Las aplicaciones interactúan con el sistema de gestión de la calidad de servicio para informarle de modificaciones de los parámetros, como los niveles de calidad válidos en un determinado momento, de forma que éste pueda actualizar información convenientemente al modelo que posee en la base de datos. De igual forma el sistema de gestión de la calidad de servicio envía órdenes a las aplicaciones que éstas deben ejecutar. Un ejemplo de ello es la indicación a una aplicación de que le ha sido cambiado el nivel de calidad, la aplicación debe reconfigurarse o cambiar ciertos parámetros de ejecución si es necesario. El usuario, por el contrario, interactúa con el usuario a través de las órdenes que éste da al sistema. La información que la arquitectura devuelve al usuario está relacionada con decisiones significativas que el sistema toma fundamentalmente por motivos de adaptación a la carga del sistema en un momento determinado.

El nivel de control de tiempo de ejecución supone una extensión a la funcionalidad básica que ofrecen las primitivas de un sistema operativo de tiempo real, tales como primitivas de concurrencia, sincronización, etc. Dichas abstracciones ofrecen un nivel de abstracción más alto que es más apropiado para las necesidades de las aplicaciones multimedia e incluyen la posibilidad de contabilizar el uso de recursos de las aplicaciones y obtención de información de monitorización

Como se puede ver en la figura 3.1.2.4 la funcionalidad relacionada con la monitorización y configuración del sistema conforma la estructura común del sistema y el resto de la funcionalidad forma la estructura específica de cada capa.

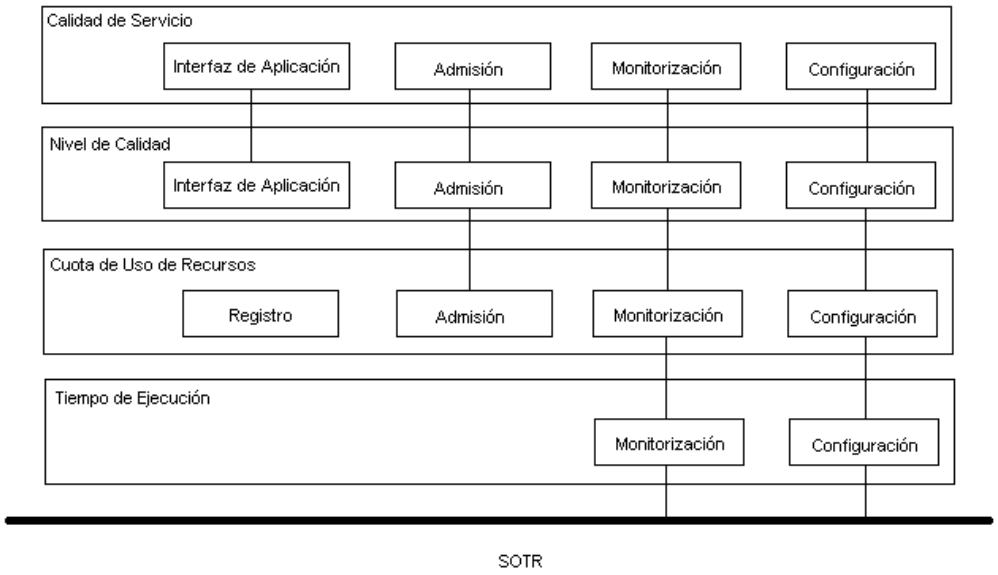

Figura 3.1.2.4 Estructura común y específica de las capas de sistema de gestión

1.2.1 Nivel de Control de la Calidad de Servicio

Este nivel es responsable de la calidad global de sistema. Decide qué aplicaciones se ejecutan en el sistema y cuál es el criterio para seleccionar sus respectivos niveles de calidad. Por este motivo este nivel necesita interactuar con el usuario. Ante una petición del usuario para que una aplicación se ejecute se realizará una prueba de admisión que decida si la aplicación puede o no ejecutarse.

El usuario es quién finalmente decide las aplicaciones y sus respectivos niveles de calidad. Esta capa puede modificar los parámetros que el usuario no establece para optimizar la calidad global del sistema.

Las responsabilidades de esta capa son las que están más próximas al nivel de usuario. Este nivel inicia el control de admisión para las aplicaciones, lo que tiene lugar cuando el usuario pide un cambio de un nivel de calidad para una aplicación o cuando una aplicación se lanza. También decide los valores de parámetros de aplicación y de sistema, que son los que determinan la política a aplicar.

1.2.2 Nivel de Control de Nivel de Calidad

Este nivel es responsable de mantener la calidad de una aplicación. Este nivel hace corresponder a un nivel de calidad un conjunto de cuotas para él, es decir, una configuración de nivel de calidad. El nivel de calidad de servicio le indica al nivel de control de nivel de calidad un nivel de calidad y el nivel de calidad debe mantener este nivel sin decrementarlo. Si en el sistema existen suficientes recursos libres se realiza una adaptación temporal de las

cuotas para aumentar los recursos destinados a una aplicación concreta. Sin embargo, si no es posible mantener el nivel de calidad requerido no se le permite que lo baje, sino que debe pasar la indicación de ello al nivel de calidad de servicio, que tomará la decisión apropiada.

Este nivel ejecuta los mecanismos básicos de cambio de modo. Los cambios de modo ocurren cuando se lanza, para o cambia de nivel una aplicación.

Un nivel de calidad de una aplicación se corresponde a un conjunto de cuotas de uso de recursos para las tareas de la misma. Esta correspondencia se traza en este nivel basándose en el modelo del sistema que se almacena en la base de datos y que contiene todas las posibles correspondencias existentes. Este nivel está encargado de escoger la mejor en cada momento de forma que se optimice la calidad global del sistema.

1.2.3 Nivel de Cuota de Uso de Recursos

Este nivel es responsable, principalmente, de gestionar las cuotas de uso de recursos y de realizar una prueba de admisión para aceptar aplicaciones en el sistema. Entre su funcionalidad avanzada se encuentra la posibilidad de mejorar la calidad de un nivel de calidad, es decir, puede decidir, de forma temporal, mejorar la configuración de un nivel de calidad de manera que dé lugar a un mejor cumplimiento de los requisitos del nivel.

Las cuotas que pasan las pruebas de planificabilidad son impuestas al nivel de control de tiempo de ejecución. Esto implica que las cuotas son comprobadas y sólo son aceptadas si pueden ser garantizadas. Este nivel también puede instruir al nivel de control de tiempo de ejecución para mantener y hacer cumplir las cuotas que son garantizadas.

1.2.4 Nivel de Control de Tiempo de Ejecución

Este nivel interactúa directamente con el sistema operativo y su misión es ofrecer extensiones a las primitivas básicas de éste. La funcionalidad adicional que proporciona es de un mayor nivel de abstracción y se ajusta mejor a las necesidades de las aplicaciones multimedia. Aunque la abstracción que proporciona este nivel es mayor que la del sistema operativo la interfaz que presenta es de bajo nivel y se ha diseñado para ser flexible. De esta forma es factible adaptarlo para experimentar con distintas interfaces de más alto nivel.

El nivel de Control de Tiempo de Ejecución ofrece un entorno para las aplicaciones multimedia. Su responsabilidad principal es mantener las garantías de cuotas de asignación de recursos para las tareas o grupos de tareas. Asigna recursos a las mismas de f horma que éstas puedan satisfacer sus requisitos de ejecución. Esta capa es responsable de prevenir que cada aplicación interfiera con las cuotas de las restantes aplicaciones.

En caso de que existan suficientes recursos libres en el sistema este nivel puede permitir a algunas tareas o clusters que en una activación determinada sobrepasen su asignación de uso de recursos consumiendo cuotas libres de otras tareas. Este nivel no puede reasignar cuotas; esto se hace en un nivel superior.

2 Independencia de la Plataforma de Ejecución

La principal ventaja de la implementación de un sistema de gestión de la calidad de servicio independiente de la plataforma es que cualquier aplicación de usuario funcionará igual independientemente de la plataforma (sistema operativo y hardware específicos) sobre la que se ejecute.

Naturalmente esto implica por un lado perder prestaciones ya que se están ignorando las particularidades de cada uno de los sistemas, pero por otro lado implica un gran ahorro en mantenimiento ya que una vez que esté hecha la implementación del sistema para un sistema operativo en concreto el coste de portabilidad de las aplicaciones de usuario que ya estaban desarrolladas será mucho menor.

La pérdida de prestaciones de las aplicaciones de usuario respecto a un sistema implementado de forma nativa dependerá de la calidad de la implementación que se haga ya que el modelo anteriormente presentado del sistema de gestión de la calidad de servicio es lo suficientemente amplio para soportar la funcionalidad de la mayoría de los sistemas operativos de tiempo real actuales.

De las cuatro capas que componen la estructura del sistema de gestión de la calidad de servicio sólo la capa de Control de Tiempo de Ejecución es dependiente del sistema operativo de tiempo real. De esta manera la implementación del sistema de gestión para un sistema operativo nuevo se reduciría a reimplementar sólo esta capa. Esta implementación supondría la implementación de un adaptador de las funciones primitivas del sistema operativo a las funciones propias del sistema de gestión de la calidad de servicio.

Para conseguir que sólo haya que reimplementar uno de los cuatro niveles la implementación del sistema independiente de la plataforma se ha realizado en Java, de manera que cada uno de las capas del sistema está representada por un conjunto de interfaces, como se muestra en la figura 3.2.1. La implementación en Java provee de por sí la independencia de la plataforma para las tres capas no dependientes del sistema operativo subyacente. Para la implementación de la capa dependiente del sistema operativo se puede usar, si es necesario, la interfaz nativa de Java.

Para probar la validez del modelo mostrado en el apartado 3.1 se ha hecho una implementación de las capas de Control de Tiempo de Ejecución y de Cuota de Uso de Recursos. A continuación se describirán los pasos que se han seguido para realizar esta implementación.

2.1 Implementación del sistema independiente de la plataforma

El sistema de gestión de la calidad de servicio se ha implementado en Java, concretamente en RTSJ sobre máquina virtual. RTSJ es un estándar para tiempo real del lenguaje Java y, por tanto, ello y el tener máquina virtual asegura su portabilidad entre plataformas. Se ha decidido realizar esta implementación porque la RTSJ es el modelo más general y por lo tanto supondrá la implementación de la mayor parte de los sistemas de tiempo real. En la mayoría de los casos la implementación de otros sistemas de tiempo real supondrá implementar sólo un subconjunto de las funciones que se soportan con RTSJ.

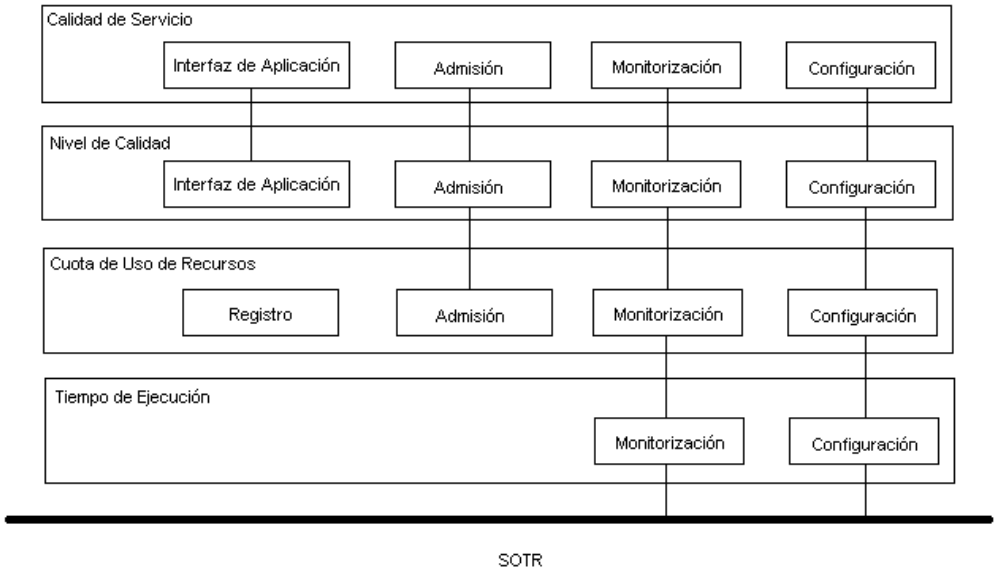

Figura 3.2.1 Interfaces que componen cada una de las capas del sistema

La versión de la RTSJ utilizada es la 1.0.2, que está desarrollada basándose en la versión 1.3 del JDK de Java. Esto hace que la implementación sea más sencilla y no tenga que recurrirse al uso de código nativo, ya que esta labor la realiza la RTSJ internamente.

La implementación de la Capa de Control de Tiempo de Ejecución implica el uso de cuatro clases fundamentales de la API que exporta la RTSJ: `RealTimeThread`, `Scheduler`, `Clock` y `Event`. Cada una de estas clases, y su función en la implementación se explicaran en detalle a continuación.

A continuación se muestra un cuadro resumen de las principales características de estas clases:

Clase	Paquete	Principales Métodos

`Scheduler`	`javax.realtime`	`addToFeasibility()`
		`removeFromFeasibility()`
		`setIfFeasible()`
		`getDefaultScheduler()`
		`setDefaultScheduler()`
		`isFeasible()`
		`fireSchedulable()`

`RealtimeThread`	`javax.realtime`	`addIfFeasible()`
		`addToFeasibility()`
		`removeFromFeasibility()`
		`setScheduler()`
		`getScheduler()`
		`getProcessingParameters()`
		`setProcessingParameters()`
		`setReleaseParameters()`
		`setReleaseParameters()`

`Clock`	`Javax.realtime`	`getRealtimeClock()` `getTime()`

2.1.1 RealTimeThead

El primer paso, y el más importante a la hora de realizar la implementación es realizar la implementación de la interfaz de la tarea.

En el caso de la RTSJ se definen dos clases que son susceptibles de implementar la interfaz tarea. Estas clases son `RealtimeThread` y `NoHeapRealtimeThread`. La clase `NoHeapRealtimeThread` extiende de `RealtimeThread`, por lo que su comportamiento es similar excepto que sus datos asociados no se guardan en la pila. Estas clases representan los hilos ejecución de tiempo real en la RTSJ y son los que se usaron para crear las clases **`RealtimeTask`** y **`NoHeapRealtimeTask`** que extendían, respectivamente, de `RealtimeThread` y `NoHeapRealtimeThread` y que implementaban la interfaz **`Task`**.

Los `RealtimeThread` y `NoHeapRealtimeThread` poseen cuatro variables que han ser configuradas antes de que el hilo se lance: coste, plazo de ejecución, período y tiempo entre invocaciones. Dependiendo de los valores que se configuren los hilos pueden tener tres tipos de comportamiento: periódico, aperiódico y esporádico. La correspondencia entre las variables que se establecen y el comportamiento es el siguiente:

- Periódico: coste, período y plazo de ejecución
- Aperiódico: coste y plazo de ejecución
- Esporádico: coste, tiempo entre invocaciones y plazo de ejecución.

Los hilos periódicos corresponderían a las tareas modeladas en el sistema de gestión como periódicas, los hilos aperiódicos corresponderían a las tareas continuas y los hilos esporádicos corresponderían a las tareas mixtas, activadas por datos de entrada.

La funcionalidad de estos hilos es similar a la de los hilos estándar de Java (que a su vez tienen el comportamiento de los hilos estándar de POSIX, ya que invocan a las funciones POSIX de forma nativa), excepto que se especifican los parámetros de comportamiento en el momento de la instanciación de los objetos. Según el comportamiento del hilo la RTSJ establece unos temporizadores que lanzan eventos si se vence el plazo de ejecución y el hilo no ha podido acceder al procesador el tiempo necesario o si el hilo ha accedido al procesador más tiempo del que tiene asignado dado que hay más recursos de los necesarios para cumplir los criterios de calidad.

En este sentido se pueden lanzar dos tipos de eventos: eventos de sobre-ejecución y de pérdida de plazo. Los eventos de sobre-ejecución se lanzan si el hilo ha accedido al procesador más tiempo del que tiene asignado, mientras que los eventos de pérdida de plazo se lanzan si el hilo no ha accedido al procesador el tiempo mínimo necesario para realizar sus funciones.

Al realizar la implementación de la interfaz **`Task`** hay que tener en cuenta cada evento generado por sobre-ejecución y pérdida de plazo se procesa generando un nuevo hilo que se planifica y se añade al sistema, con lo que hay que tenerlos en cuenta a la hora de calcular la carga total en el sistema.

Tal y como está definida la especificación de la RTSJ los eventos lanzados en caso de sobre-ejecución o de pérdida de plazo no son recibidos directamente por el hilo sobre el que se ha

producido la sobre-ejecución o la pérdida de plazo, sino que al lanzarse el evento se invoca una función de callback que se puede especificar en el momento de creación del hilo, o posteriormente. Esta función de callback permite que el hilo no tenga que implementar lógica adicional en caso de que no se pueda garantizar un nivel mínimo de recursos en el sistema y que sea otro elemento, codificado para tal fin, el que se encargue de esa labor.

En la implementación realizada de HOLAQoS sobre RTSJ-TimeSys estas funciones de callback son manejadas por el planificador de tareas, de manera que el planificador es el elemento que realiza la actualización de los valores de cuotas de recursos consumidas y restantes en el sistema.

Teniendo en cuenta estos aspectos del funcionamiento de la RTSJ y viendo la definición de la interfaz **`Task`**, que se describe a continuación la implementación de la clase **`RealTimeTask`** fue bastante sencilla.

```
public interface Task extends Runnable{
	public int getDDBBid();
	public String getName();
	public void setName(String name);
	public TaskStatus getStatus();
	public void setStatus(TaskStatus status);
	public TaskArrivalPattern getArrivalPattern();
	public Priority getSystemPriority();
	public boolean setSystemPriority(Priority priority);
	public Cluster getCluster();
	public void setCluster(Cluster cluster);
	public void setTaskConfigurations(Collection taskConfigurations);
	public void addTaskConfigurations(Collection taskConfigurations);
	public boolean setActiveTaskConfiguration(TaskConfiguration taskConf);
	public TaskConfiguration getActiveTaskConfiguration();
	public Collection getTaskConfigurations();
	public String toString();
}
```

Como se puede ver los métodos que la interfaz **`Task`** obliga a implementar son de dos tipos principalmente:

- Getters y Setters de distintas propiedades
- Run. Este método lo impone la interfaz Runnable de la que extiende

Ante esto se optó por realizar la siguiente implementación de la clase **`RealTimeTask`**. Se asoció una propiedad por cada pareja de métodos **`get-set`** y se declaró la clase como abstracta, para que las clases hijas implementaran la lógica del hilo definiendo el método run.

La implementación de la clase `NoHeapRealTimeTask` fue análoga, cambiando sólo la superclase y los parámetros de los constructores.

```
public abstract class RealTimeTask extends RealtimeThread implements Task  {
	private int		DDBBid;
	private TaskStatus	status;
```

```
        private TaskArrivalPattern arrivalPattern;
        private Priority           priority;
        private Cluster            cluster;
        private Vector             taskConfigurations;
        private TaskConfiguration  activeTaskConfiguration;

        public RealTimeTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
                            HighResolutionTime deadline,  HighResolutionTime start,
                            HighResolutionTime period) ;

        public RealTimeTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
                     HighResolutionTime deadline) ;

        public RealTimeTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
                    HighResolutionTime deadline, HighResolutionTime minimumInterarrival) ;

        public TaskArrivalPattern getArrivalPattern;
        public Cluster getCluster() ;
        public TaskStatus getStatus;
        public Priority getSystemPriority() ;
        public void setCluster(Cluster cluster) ;
        public void setStatus(TaskStatus status) ;
        public boolean setSystemPriority(Priority priority;
        public void addTaskConfigurations(Collection taskConfigurations;
        public Collection getTaskConfigurations;
        public void setTaskConfigurations(Collection taskConfigurations;
        public int getDDBBid() ;
        public TaskConfiguration getActiveTaskConfiguration;
        public boolean setActiveTaskConfiguration(TaskConfiguration taskConf;
        public abstract void run();
}
```

La implementación de los métodos fue la siguiente:

```
public RealTimeTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
                    HighResolutionTime deadline,  HighResolutionTime start,
                    HighResolutionTime period) {
        super(new PriorityParameters(priority.getPriority()),
                new PeriodicParameters(
                  new RelativeTime(start.getMillis(), start.getNanos()),
                  new RelativeTime(period.getMillis(), period.getNanos()),
                  new RelativeTime(cost.getMillis(), cost.getNanos()),
                  new RelativeTime(deadline.getMillis(), deadline.getNanos()),
                  null,
                  null
                 )
                );
        this.DDBBid           = DDBBid.intValue();
        this.cluster              = null;
        this.arrivalPattern   = TaskArrivalPattern.PERIODIC;
        this.status               = TaskStatus.NON_ACTIVE;
```

```
        this.priority             = priority;
        this.taskConfigurations = new Vector(1,1);
}

public RealTimeTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
                    HighResolutionTime deadline) {
        super(new PriorityParameters(priority.getPriority()),
                new AperiodicParameters(
                  new RelativeTime(cost.getMillis(), cost.getNanos()),
                  new RelativeTime(deadline.getMillis(), deadline.getNanos()),
                  null,
                  null
                )
          );
        this.DDBBid          = DDBBid.intValue();
        this.cluster             = null;
        this.arrivalPattern  = TaskArrivalPattern.CONTINOUS;
        this.status               = TaskStatus.NON_ACTIVE;
        this.priority            = priority;
        this.taskConfigurations = new Vector(1,1);
}

public RealTimeTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
                    HighResolutionTime deadline, HighResolutionTime minimumInterarrival) {
        super(new PriorityParameters(priority.getPriority()),
                new SporadicParameters(
                 new RelativeTime(minimumInterarrival.getMillis(),
                                  minimumInterarrival.getNanos()
                                  ),
                 new RelativeTime(cost.getMillis(), cost.getNanos()),
                 new RelativeTime(deadline.getMillis(), deadline.getNanos()),
                 null,
                 null
                )
          );
        this.DDBBid             = DDBBid.intValue();
        this.cluster              = null;
        this.arrivalPattern  = TaskArrivalPattern.SPORADIC;
        this.status               = TaskStatus.NON_ACTIVE;
        this.priority             = priority;
        this.taskConfigurations = new Vector(1,1);
}

public TaskArrivalPattern getArrivalPattern() {
        return this.arrivalPattern;
}
public Cluster getCluster() {
        return this.cluster;
}
public TaskStatus getStatus() {
        return this.status;
```

```
}
public Priority getSystemPriority() {
        return this.priority;
}
public void setCluster(Cluster cluster) {
        this.cluster = cluster;
}
public void setStatus(TaskStatus status) {
        this.status = status;
}
public boolean setSystemPriority(Priority priority) {
        if(!this.setSchedulingParametersIfFeasible(
             new PriorityParameters(priority.getPriority())))
                return false;
        this.priority = priority;
        return true;
}
public void addTaskConfigurations(Collection taskConfigurations) {
        this.taskConfigurations.addAll(taskConfigurations);
}
public Collection getTaskConfigurations() {
        return this.taskConfigurations;
}
public void setTaskConfigurations(Collection taskConfigurations) {
        this.taskConfigurations.clear();
        this.addTaskConfigurations(taskConfigurations);
}
public int getDDBBid() {
        return this.DDBBid;
}
public TaskConfiguration getActiveTaskConfiguration(){
        return this.activeTaskConfiguration;
}
public boolean setActiveTaskConfiguration(TaskConfiguration taskConf){
        if(!this.taskConfigurations.contains(taskConf)){
                return false;
        }
        if(this.activeTaskConfiguration != null)
                this.activeTaskConfiguration.setActive(false);
        this.activeTaskConfiguration = taskConf;
        this.activeTaskConfiguration.setActive(true);
        return true;
}
```

2.1.2 Planificador basado en prioridades

Una vez se ha realizado la adaptación de las clases `RealtimeThread` y `NoHeapRealtimeThread` se presenta a continuación la implementación de las dos interfaces que componen la *Capa de Control de Tiempo de Ejecución: Monitorización y Configuración*.

El núcleo de ambas implementaciones es el planificador basado en prioridades que ofrece la RTSJ. Este planificador está representado por el objeto **`PriorityScheduler`** que es el planificador de hilos la RTSJ. Este planificador está implementado siguiendo el patrón **Singleton**, por lo que es accesible desde cualquier parte del código, esto hace que se pueda separar la implementación de la interfaz de monitorización de la implementación de la interfaz de configuración.

El planificador de la RTSJ se encarga de planificar los hilos y de hacer los cálculos necesarios para saber si un RealtimeThread o NoHeapRealtimeThread puede ser añadido al sistema. De esta manera, invocando adecuadamente las funciones que ofrece el planificador de la RTSJ y tratando adecuadamente los códigos de error que devuelven esas invocaciones (tarea de una cierta complejidad) se ha obtenido la implementación de la capa de Control de Tiempo de Ejecución.

El planificador basado en prioridades, como su propio nombre indica, implementa una política de planificación de tareas basada en su parámetro de prioridad, de forma que garantiza que la tarea en ejecución es siempre aquella tarea no bloqueada de mayor prioridad. Este planificador recibe objetos de tipo `Schedulable`, que son los objetos que contienen la lógica a ejecutar, y, en función de los parámetros de planificación de esos objetos, decide si el objeto se puede añadir al sistema con garantías de que los recursos necesarios para el objeto están disponibles en el sistema. Una vez añadido un objeto Schedulable al planificador ese objeto accederá al procesador dependiendo de la prioridad que tenga definida. Esto se puede ver más claramente en la figura 3.2.2

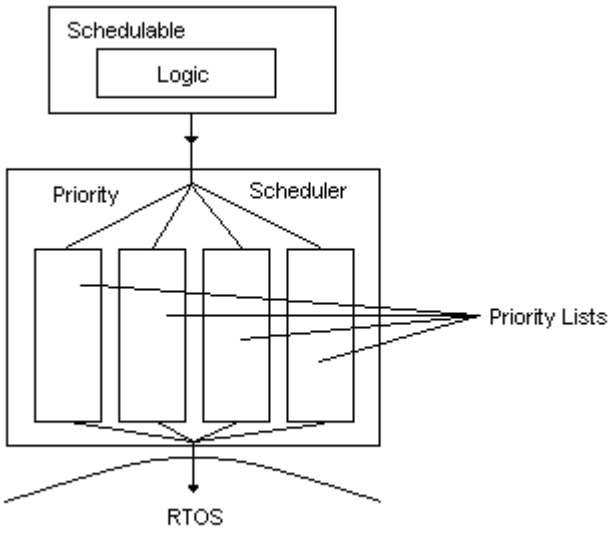

Figura 3.2.2 Funcionamiento del planificador basado en prioridades

Las clases `RealtimeThread` y `AsyncEventHandler` implementan la interfaz `Schedulable`. Esto significa que tanto los hilos en ejecución en el sistema como los eventos generados durante la monitorización de éstos. Los parámetros que gestiona el objeto `Schedulable` determinan el comportamiento de la entidad a la hora de planificar su acceso a los recursos del sistema. Los parámetros del objeto `Schedulable` son de cuatro tipos distintos:

- `Scheduling Parameters`: Son parámetros que determinan la prioridad de acceso de la entidad al procesador del sistema.
- `Memory Parameters`: Son parámetros que determinan el uso que la entidad hace de la memoria del sistema.
- `Release Parameters`: Son parámetros que determinan el uso que la entidad hace del procesador del sistema.
- `Processing Group Parameters`: Son parámetros que determinan el uso que un grupo de entidades hacen del procesador del sistema.

A continuación se muestran los métodos más importantes de la interfaz `Schedulable`.

`Schedulable`	javax.realtime	addToFeasibility() removeFromFeasibility() getMemoryParameters() setMemoryParameters() setMemoryParametersIfFeasible() getProcessingGroupParameters() setProcessingGroupParameters() setProcessingGroupParametersIfFeasible() getReleaseParameters() setReleaseParameters() setReleaseParametersIfFeasible() getSchedulingParameters() setSchedulingParameters() setSchedulingParametersIfFeasible() setSheduler() getScheduler()

El criterio para decidir si un objeto `Schedulable` puede ser añadido al sistema depende de la carga del sistema. Tras la descompilación y el análisis del código de la clase `PriorityScheduler` se ha averiguado que si la carga del sistema después de añadir el nuevo objeto `Schedulable` es mayor del 80% el objeto no puede ser añadido. Éste es un margen de seguridad con el trabaja la máquina virtual de TimeSys.

Como es lógico pensar, las clases `RealTimeThread` y `NoHeapRealtimeThread` son objetos Schedulable. Sus parámetros de planificación son coste, plazo de ejecución, período y tiempo entre invocaciones, es decir, los mismos parámetros que se invocan en su creación.

En la implementación realizada en el momento de añadir un objeto Schedulable al planificador también se establecen los temporizadores y las funciones de callback en caso de que se produzcan los eventos de sobre-ejecución y de pérdida de plazo. El procedimiento para establecer estos temporizadores se explica en detalle en el apartado 2.1.4 de este capítulo.

Teniendo en cuenta todo esto en cuenta, es fácil darse cuenta de que el planificador sólo se usa en las funciones `createTask` y `deleteTask` de la implementación de la capa de Control de Tiempo de Ejecución. El código de estas funciones es el siguiente:

```
public RTStatus createTask(Task task) {
```

```
        if (!(task instanceof Schedulable)) {
                return RTStatus.TASK_TYPE_UNKNOWN_ERROR;
        }

        if(task.getStatus() != TaskStatus.NON_ACTIVE){
                return RTStatus.TASK_ALREADY_CREATED;
        }

        ((Schedulable) task).setScheduler(this.scheduler);

        if (!((Schedulable) task).addTsoFeasibility()){
                ((Schedulable) task).removeFromFeasibility();
                return RTStatus.NOT_FEASIBLE_ERROR;
        }

        ProcessingGroupParameters params = task.getProcessingGroupParameters();
        Params.setCostOverrunHandler(new OverrunHandler());
        Params.setDeadlineMissHandler(new MissHandler());

        task.setStatus(TaskStatus.ACTIVE);
        return RTStatus.OK;
}

public RTStatus deleteTask(Task task) {
        if (!(task instanceof Schedulable) || !(task instanceof Thread) ||
                        ((Schedulable)task).getScheduler() != this.scheduler) {
                return RTStatus.TASK_TYPE_UNKNOWN_ERROR;
        }

        if(task.getStatus() == TaskStatus.NON_ACTIVE){
                return RTStatus.TASK_NOT_SCHEDULED;
        }

        if(!((Thread)task).isAlive()){
                task.setStatus(TaskStatus.ACTIVE);
        }

        if(task.getStatus() != TaskStatus.ACTIVE){
                Log.RTwarning("Task "+ task.getDDBBid() +" active. Stopped!");
                //return RTStatus.TASK_STILL_RUNNING;
        }

        if (!((Schedulable) task).removeFromFeasibility())
        Log.RTwarning("Task "+ task.getDDBBid() +" removed from feasibility with warnings");

        task.setStatus(TaskStatus.NON_ACTIVE);

        return RTStatus.OK;
}
```

El resto de funciones usadas en la implementación de la capa de Control de Tiempo de Ejecución son las funciones que exporta la API de la clase Thread de Java, es decir, las funciones de la API POSIX para los pthreads.

2.1.3 Clock y RealtimeClock

Para poder controlar el tiempo que cada una de las tareas accede al procesador no se pueden usar las funciones que provee por defecto la API de Java. Esto se debe a dos motivos, el primero es que la máquina virtual de TimeSys, al igual que cualquier máquina virtual de Java implementada, es una aplicación más en el sistema y no accede directamente al reloj del sistema sino que accede al reloj del sistema a través de la API que el sistema operativo ofrece a las aplicaciones de usuario. El segundo motivo es que la API que la máquina virtual de Java ofrece a su vez a las aplicaciones que están ejecutándose en ella no ofrece la precisión que la API de POSIX es capaz de proveer. Esto ha hecho que la implementación de la RTSJ implemente de forma nativa el acceso al reloj del sistema.

Para hacer los cálculos de tiempo se ha usado la clase `Clock` que provee la especificación de la RTSJ. Esta clase permite obtener el tiempo actual del reloj del procesador usando funciones nativas del sistema.

La clase `Clock` es una clase abstracta, de manera que realmente la clase que implementa la lógica necesaria es la clase `RealtimeClock`. Esta clase `RealtimeClock`, al igual que la clase `PriorityScheduler` está implementada usando un patrón Singleton, por lo que puede ser accedida en todo momento.

El patrón **Singleton** es un patrón de diseño software orientado a objetos que se caracteriza por que una determinada clase de objeto sólo puede tener una única instancia en todo el sistema. En este sentido es semejante a una variable global en la programación funcional. En este caso está claro que el reloj es único para el sistema, por lo que es susceptible de ser definido usando este patrón de diseño software.

Siguiendo la especificación del patrón Singleton tanto la instanciación como el acceso al reloj se realizarán de la misma manera:

```
RealtimeClock c= Clock.getRealtimeClock();
```

La invocación al método para obtener el tiempo del procesador es la siguiente:

```
AbsoluteTime time = Clock.getRealtimeClock().getTime();
```

`AbsoluteTime` es una clase que internamente almacena tiempos con una resolución de nanosegundos. En las pruebas realizadas la máxima precisión que se ha necesitado es de milisegundos, por lo que ésta es más que suficiente en este caso.

2.1.4 AsyncEvent y Timer

La invocación de los eventos de sobre-ejecución o de pérdida de plazo se realiza usando la clase `AsyncEvent` y su subclase `Timer`. La clase `AsyncEvent` invoca de código definido por el usuario cuando se produce un determinado evento. Este código que se ejecuta como tratamiento del evento tiene la particularidad de que sigue unas determinadas restricciones de coste de ejecución y plazo de ejecución, al igual que una tarea más del sistema. Esto permite que se pueda controlar la carga que impone en el sistema la gestión de un fallo provocado por sobrecarga sin sobrecargar aún más el procesador por ese código extra que se debe ejecutar.

Por cada RealtimeThread que se instancia la implementación de la RTSJ instancia también, al menos, un `Timer`. Un `Timer`, como su propio nombre indica, no es más que un temporizador, que una vez vencido comprueba una condición. Si esa condición se cumple se ejecuta el código asociado.

En la implementación realizada el temporizador creado se usaba para detectar si se había producido alguna condición de sobre-ejecución. El código usado para realizar esto es el siguiente:

```
public class BudgetScheduler extends PriorityScheduler {
     private static BudgetScheduler defaultBudgetScheduler = getInstance();

     protected BudgetScheduler() {
        clusters = new Vector();
        tasks    = new HashMap();
     }

     public static BudgetScheduler getInstance(){
        if(defaultBudgetScheduler == null)
               defaultBudgetScheduler = new BudgetScheduler();
        return defaultBudgetScheduler;
     }

     protected boolean addToFeasibility(Schedulable schedulable){
        if(!(schedulable instanceof Task)) {
               return false;
        }

        boolean result = super.addToFeasibility(schedulable);

         if(result){
         addMonitorizationInfo(schedulable);
         AsyncEventHandler handler = new OverrunHandler((Task)schedulable);
         handler.setScheduler(this);
         if(!handler.addIfFeasible()){
               Log.RTerror("Handler could not be added");
         }
         schedulable.getReleaseParameters().setCostOverrunHandler(
                                                    handler
                                                   );

         Log.RTdebug(handler.toString());
     }

     return result;
     }
}
```

En este ejemplo la clase **OverrunHandler** es la clase que implementa la lógica necesaria en caso de que se produzca sobre-ejecución. En el siguiente código de ejemplo simplemente se muestra un mensaje en el momento en que se produce un evento de sobre-ejecución:

```
public class OverrunHandler extends AsyncEventHandler{
     Task task;

     public OverrunHandler(Task task) {
        super();
        this.task = task;
     }

     // run method for this class
     public void handleAsyncEvent(){
        HighResolutionTime now = Clock.getRealtimeClock().getTime();

        long time = now.getMilliseconds() * 100000 + now.getNanoseconds();

        Log.RTerror("Task "+ task.getDDBBid() + " has costoverrun @ " + time + " " +
getPendingFireCount());
     }
}
```

Esta implementación es muy potente, pero, lamentablemente, no funciona correctamente, ya que no se lanzan los eventos de sobre-ejecución ni de pérdida de plazo cuando se deberían lanzar, por lo que la potencia de la RTSJ queda considerablemente mermada sin esta característica. Según se puede leer en http://www.rtsj.org/docs/OSPlatforms.html para el correcto funcionamiento de esta característica es necesario un soporte especial del sistema

operativo que permita la flexibilidad suficiente para poder conocer el tiempo consumido de CPU de cada uno de los hilos en ejecución en el sistema.

Este es un problema de la máquina virtual de TimeSys, cuyo importe para una licencia académica es demasiado elevado. Por el contrario la máquina virtual Jamaica sí tiene soporte para esta característica y no se requiere ningún soporte especial del sistema operativo, por lo que es mucho más adecuada para su uso con fines académicos.

3 Interfaces

La lógica de cada una de las capas es semejante, pero la diferencia entre ellas radica en el nivel de abstracción de las entidades sobre la que actúan. De esta manera los niveles inferiores del sistema de gestión actúan sobre entidades con un nivel de abstracción menor y, por lo tanto, mayor dependencia del entorno hardware y software de la máquina en la que ejecutan. Por el contrario los niveles superiores del sistema tienen una dependencia muy pequeña o nula respecto al sistema en que se ejecutan. Al tener una lógica semejante las interfaces de cada una de las capas son análogas a excepción del significado de la entidad que modifican.

Las interfaces de las distintas capas del sistema de gestión de la calidad de servicio están implementadas en varios componentes, de manera que no sea necesaria la implementación de la funcionalidad completa de la capa correspondiente. La subdivisión en componentes para cada capa se hace de acuerdo a la figura 3.1.5.3.

3.1 Capa de Control de Tiempo de Ejecución

La capa de control de tiempo de ejecución es la capa encargada de la creación y eliminación de tareas del sistema y de la recolección de la información necesaria para las capas superiores.

La capa de control de tiempo de ejecución está divido en dos componentes que pueden ser implementadas independientemente:

3.1.1 Monitorization Component

La componente de monitorización de la capa de control de tiempo de ejecución es la encargada de la recolección de la información sobre el uso de recursos de las tareas y clústeres activos en el sistema.

- `RTStatus setSystemTasksMonitorizationOn(Resource resource);`

  `Activa la monitorización de todas las tareas creadas que usan el recurso pasado como parámetro.`

  Devuelve RTStatus.OK.

- `RTStatus setSystemTasksMonitorizationOff(Resource resource);`

  Desactiva la monitorización de todas las tareas creadas que usan el recurso pasado como parámetro.

  `Devuelve RTStatus.OK.`

- `RTStatus setSystemTaskMonitorizationOn(Task task, Resource resource);`

  Activa la monitorización del recurso especificado sobre la tarea pasada como parámetro.

  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si la tarea no ha sido creada o RTStatus.OK en caso contrario.

- `RTStatus setSystemTaskMonitorizationOff(Task task, Resource resource);`

  Desactiva la monitorización del recurso especificado sobre la tarea pasada como parámetro.

  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si la tarea no ha sido creada o RTStatus.OK en caso contrario.

- `RTStatus resetTaskMonitorization(Task task, Resource resource);`

  Establece el valor de la cuota consumida en el recurso especificado por la tarea al valor inicial.

  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si la tarea no ha sido creada o RTStatus.OK en caso contrario.

- `Collection getTasksMonitorizationInfo(Collection tasks);`

  Devuelve una colección de objetos MonitorizationInfo del mismo tamaño que la colección que se le pasa como parámetro.
  Si no se encuentra la información de alguna de las tareas que se le pasan como parámetro en la posición correspondiente de la colección de salida se inserta un null.

- `Budget getRemainingTaskBudget(Task task, Resource resource);`

  Devuelve la cuota restante del recurso especificado asociado a la tarea especificada. Si no se encuentra la información de monitorización de la tarea se devuelve una cuota nula(el resultado de ejecutar Budget.nul()).

- `Budget getExecutedTaskBudget(Task task, Resource resource);`

  Devuelve la cuota ejecutada del recurso especificado asociado a la tarea especificada. Si no se encuentra la información de monitorización de la tarea se devuelve una cuota nula(el resultado de ejecutar Budget.nul()).

- `RTStatus setSystemClustersMonitorizationOn(Resource resource);`

  Activa la monitorización de todos los clusters creados que usan el recurso pasado como parámetro.
  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si alguna tarea de un cluster no ha sido creada o RTStatus.OK en caso contrario.

- `RTStatus setSystemClustersMonitorizationOff(Resource resource);`

  Desactiva la monitorización de todos los clusters creados que usan el recurso pasado como parámetro.

Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si alguna tarea de un cluster no ha sido creada o RTStatus.OK en caso contrario.

- `RTStatus setSystemClusterMonitorizationOn(Cluster cluster, Resource resource);`

  Activa la monitorización de todas las tareas creadas del cluster especificado que usan el recurso pasado como parámetro.
  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si alguna tarea de un cluster no ha sido creada o RTStatus.OK en caso contrario.

- `RTStatus setSystemClusterMonitorizationOff(Cluster cluster, Resource resource);`

  Desactiva la monitorización de todas las tareas creadas del cluster especificado que usan el recurso pasado como parámetro.
  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si alguna tarea de un cluster no ha sido creada o RTStatus.OK en caso contrario.

- `RTStatus resetClusterMonitorization(Cluster cluster, Resource resource);`

  Establece el valor de la cuota consumida en el recurso especificado por las tareas creadas del cluster pasado como parámetro al valor inicial.
  Devuelve RTStatus.TASK_INFORMATION_NOT_FOUND si alguna tarea no ha sido creada o RTStatus.OK en caso contrario.

- `Collection getClustersMonitorizationInfo(Collection cluster);`

  Devuelve una colección con los resultados de la invocación a getTasksMonitorizationInfo() sobre las tareas de cada uno de los clusters.

3.1.2 Settings Component

La componente de configuración de la capa de control de tiempo de ejecución es la encargada de la creación, inicio, parada y eliminación de las tareas y clústeres del sistema.

- `RTStatus createTask(Task task);`

  Añade la tarea que se le pasa como parámetro al planificador.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR, RTStatus.TASK_ALREADY_CREATED o RTStatus.NOT_FEASIBLE_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- `RTStatus startTask(Task task);`

  Comienza la ejecución de la tarea que se le pasa como parámetro.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR o RTStatus.TASK_NOT_SCHEDULED si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus restartTask(Task task);

  Interrumpe la ejecución de la tarea que se le pasa como parámetro y la vuelve a arrancar.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus deleteTask(Task task);

  Elimina la tarea que se le pasa como parámetro del planificador. Si está activa se interrumpe su ejecución.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR o TRStatus.TASK_NOT_SCHEDULED si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus changeTaskPriority(Task task,Priority priority);

  Cambia la prioridad de la tarea que se pasa como parámetro.
  Devuelve RTStatus.NOT_FEASIBLE_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus sleepTask(Task task);

  Duerme la ejecución de la tarea que se le pasa como parámetro hasta que se invoque la función resumeTask.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus suspendTask(Task task);

  Suspende la ejecución de la tarea que se le pasa como parámetro hasta que se invoque la función resumeTask.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus resumeTask(Task task);

  Despierta la tarea que se le pasa como parámetro si previamente se había invocado a la función sleepTask.
  Devuelve RTStatus.TASK_TYPE_UNKNOWN_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus refillTaskBudget(Task task, Resource resource);

  Devuelve a su valor inicial el valor actual de la cuota del recurso asignada a la tarea que se pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus initTaskBudget(Task task, Resource resource);

  Inicializa el valor de la cuota del recurso asignada a la tarea que se pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus setTaskBudget(Task task, Resource resource, Budget budget);

  Establece el valor actual de la cuota del recurso asignada a la tarea que se pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus setTaskAccountingOn(Task task, Resource resource);

  Comienza la monitorización del uso que hace la tarea que se pasa como parámetro sobre el recurso especificado.
  Devuelve RTStatus.OK.

- RTStatus setTaskAccountingOff(Task task, Resource resource);

  Finaliza la monitorización del uso que hace la tarea que se pasa como parámetro sobre el recurso especificado.
  Devuelve RTStatus.OK.

- RTStatus createCluster(Cluster cluster);

  Añade las tareas del cluster que se le pasa como parámetro al planificador. Si no se puede añadir una de las tareas no se añade ninguna tarea al planificador.
  Devuelve RTStatus.CLUSTER_ALREADY_CREATED, RTStatus.TASK_ALREADY_CREATED o RTStatus.NOT_FEASIBLE_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus startCluster(Cluster cluster);

  Comienza la ejecución de las tareas del cluster que se le pasa como parámetro. Si no se puede comenzar una de las tareas no se añade ninguna tarea al planificador.
  Devuelve RTStatus.TASK_NOT_FEASIBLE si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus restartCluster(Cluster cluster);

  Invoca la función restartTask sobre todas las tareas del cluster que se le pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus deleteCluster(Cluster cluster);

  Elimina las tareas del cluster que se le pasa como parámetro del planificador. Si está activa se interrumpe su ejecución.
  Devuelve TRStatus.CLUSTER_NOT_SCHEDULED si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus changeClusterPriority(Cluster cluster,Collection priorities);

  Cambia las prioridades de las tareas que componen el cluster con las que se le pasan.
  Devuelve RTStatus.SIZE_COLLECTION_INCORRECT_ERROR o RTStatus.NOT_FEASIBLE_ERROR si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus suspendCluster(Cluster cluster);

  Duerme la ejecución de las tareas del cluster que se le pasa como parámetro hasta que se invoque la función resumeCluster.
  Devuelve RTStatus.CLUSTER_NOT_SCHEDULED si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus resumeCluster(Cluster cluster);

Despierta las tareas del cluster que se le pasa como parámetro si previamente se había invocado a la función suspendCluster. Devuelve RTStatus. CLUSTER_NOT_SCHEDULED si se produce algún error o RTStatus.OK en caso contrario.

- RTStatus refillClusterBudget(Cluster cluster, Resource resource);

  Devuelve a su valor inicial el valor actual de las cuotas del recurso asignadas a las tareas del cluster que se pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus initClusterBudget(Cluster cluster, Resource resource);

  Inicializa el valor de las cuotas del recurso asignadas a las tareas del cluster que se pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus setClusterBudget(Cluster cluster, Resource resource, Budget budget);

  Inicializa el valor de las cuotas del recurso asignadas a las tareas del cluster que se pasa como parámetro.
  Devuelve RTStatus.OK.

- RTStatus setClusterAccountingOn(Cluster cluster, Resource resource);

  Comienza la monitorización del uso que hacen las tareas del cluster que se pasa como parámetro sobre el recurso especificado.
  Devuelve RTStatus.OK.

- RTStatus setClusterAccountingOff(Cluster cluster, Resource resource);

  Detiene la monitorización del uso que hacen las tareas del cluster que se pasa como parámetro sobre el recurso especificado.
  Devuelve RTStatus.OK.

- Queue createQueue(String name, **int** count, QueueFlags flags);

  Devuelve una nueva instancia de la clase hola.concurrent.queue.Queue.

- RTStatus deleteQueue(Queue queue);

  Libera los recursos asociados a la Queue que se le pasa como parámetro.
  Devuelve RTStatus.OK.

- Semaphore createSemaphore(String name, **int** count, SemaphoreFlags flags);

  Devuelve una nueva instancia de la clase hola.concurrent.semaphore.Semaphore.

- RTStatus deleteSemaphore(Semaphore semaphore);

```
Libera los recursos asociados al Semaphore que se le pasa como
parámetro.
Devuelve RTStatus.OK.
```

- `RTStatus sendEvent(Task task, EventType event);`

```
Envía un evento a la tarea que se le pasa como parámetro.
Devuelve RTStatus.OK.
```

- `RTStatus receiveEvent(EventType event, EventFlags flags, HighResolutionTime time);`

```
Recibe un evento del sistema.
Devuelve RTStatus.OK.
```

3.2 Capa de Control de Cuota de Uso de Recursos

La capa de control de uso de recursos establece y calculas las cuotas de uso de recursos de cada una de las tareas del sistema, es decir, es la encargada de garantizar que las todas tareas acceden a los recursos que han sido puestos a su disposición cuando fueron creadas en el sistema.

La capa de control de uso de recursos está divido en cuatro componentes que pueden ser implementadas independientemente:

3.2.1 Admission Component

La componente de admisión de la capa de control de cuotas de usos de recurso es la encargada de verificar si la modificación de las cuotas de uso de recursos actuales son factibles.

- `BdcStatus testClusterConfiguration(Collection clustersConfiguration);`

```
Prueba si la configuración de los clusters que le pasa como
parámetro es realizable.
Devuelve BdcStatus.OK o
BdcStatus.CLUSTER_CONFIGURATION_NOT_FEASIBLE;
```

- `BdcStatus testTaskConfiguration(Collection tasks);`

```
Prueba si la configuración de las tareas que le pasa como
parámetro es realizable.
Devuelve BdcStatus.OK o
BdcStatus.TASK_CONFIGURATION_NOT_FEASIBLE;
```

3.2.2 Monitorization Component

La componente de monitorización de la capa de control de cuotas de usos de recurso es la encargada de la recolección de la información sobre el uso de las cuotas consumidas y disponibles de las tareas y clústeres activos en el sistema.

- `BdcStatus setSystemClustersMonitorizationOn(Resource resource);`

Invoca la función setSystemClustersMonitorizationOn de nivel de control de tiempo de ejecución.
Devuelve BdcStatus.OK.

- BdcStatus setSystemClustersMonitorizationOff(Resource resource);

  Invoca la función setSystemClustersMonitorizationOff de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK.

- BdcStatus setSystemTasksMonitorizationOn(Resource resource);

  Invoca la función setSystemTasksMonitorizationOn de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK.

- BdcStatus setSystemTasksMonitorizationOff(Resource resource);

  Invoca la función setSystemTasksMonitorizationOFF de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK.

- BdcStatus setTaskMonitorizationOn(Task task,Resource resource);

  Invoca la función setSystemTaskMonitorizationOn de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.TASK_INFORMATION_NOT_FOUND si se produce algún error o BdcStatus.OK en caso contrario.

- BdcStatus setTaskMonitorizationOff(Task task,Resource resource);

  Invoca la función setSystemTaskMonitorizationOff de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.TASK_INFORMATION_NOT_FOUND si se produce algún error o BdcStatus.OK en caso contrario.

- BdcStatus setClusterMonitorizationOn(Cluster cluster,Resource resource);

  Invoca la función setSystemClusterMonitorizationOn de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.CLUSTER_INFORMATION_NOT_FOUND si se produce algún error o BdcStatus.OK en caso contrario.

- BdcStatus setClusterMonitorizationOff(Cluster cluster,Resource resource);

  Invoca la función setSystemClusterMonitorizationOff de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.CLUSTER_INFORMATION_NOT_FOUND si se produce algún error o BdcStatus.OK en caso contrario.

3.2.3 Registry Component

La componente de registro de la capa de control de cuotas de usos de recurso es la encargada de mantener una información de las tareas que se han creado y/o eliminado del sistema.

- `BdcStatus registerTaskCreation(Task task);`

  Registra la creación de la tarea que se pasa como parámetro.
  Devuelve BdcStatus.TASK_ALREADY_REGISTERED si se produce algún error o BdcStatus.OK en caso contrario.

- `BdcStatus registerTaskDeletion(Task task);`

  Registra la eliminación de la tarea que se pasa como parámetro.
  Devuelve BdcStatus.TASK_NOT_REGISTERED si se produce algún error o BdcStatus.OK en caso contrario.

- `BdcStatus registerClusterCreation(Cluster cluster);`

  Registra la creación del cluster que se le pasa como parámetro, así como de sus tareas asocidadas.
  Devuelve BdcStatus.CLUSTER_ALREADY_REGISTERED, BdcStatus.TASK_ALREADY_REGISTERED si se produce algún error o BdcStatus.OK en caso contrario.

- `BdcStatus registerClusterDeletion(Cluster cluster);`

  Registra la eliminación del cluster que se le pasa como parámetro, así como de sus tareas asocidadas.
  Devuelve BdcStatus.CLUSTER_NOT_REGISTERED si se produce algún error o BdcStatus.OK en caso contrario.

3.2.4 Settings Component

La componente de configuración de la capa de control de cuotas de usos de recurso es la encargada de crear, iniciar, parar y eliminar tareas y clústeres en el sistema.

- `BdcStatus setClusterConfig(Cluster cluster,ClusterConfiguration clusterConfiguration);`

  Establece la configuración del cluster como activa para el cluster pasado como parámetro.
  Devuelve BdsStatus.CLUSTER_CONFIGURATION_NOT_FOUND si la configuración del cluster no está entre las posibles para ese cluster o BdcStatus.OK en caso contrario.

- `BdcStatus setTaskConfig(Task task,TaskConfiguration taskConfiguration);`

  Establece la configuración de la tarea como activa para la tarea pasada como parámetro.
  Devuelve BdsStatus.TASK_CONFIGURATION_NOT_FOUND si la configuración de la tarea no está entre las posibles para esa tarea o BdcStatus.OK en caso contrario.

- `BdcStatus refillClusterBudget(Cluster cluster);`

Invoca la función refillClusterBudget de nivel de control de tiempo de ejecución.
Devuelve BdcStatus.OK.

- BdcStatus refillTaskBugdet(Task task);

  Invoca la función refillTaskBudget de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK.

- BdcStatus setClusterPriorities(Cluster cluster, Collection priorities);

  Invoca la función changeClusterPriorities de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.SIZE_COLLECTION_ERROR o BdcStatus.NOT_FEASIBLE_ERROR si se produce algún error o BdcStatus.OK en caso contrario.

- BdcStatus setTaskPriority(Task task,Priority priority);

  Invoca la función changeTaskPriority de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.NOT_FEASIBLE_ERROR si se produce algún error o BdcStatus.OK en caso contrario.

- BdcStatus suspendCluster(Cluster cluster);

  Invoca la función suspendCluster de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK en caso contrario.

- BdcStatus startCluster(Cluster cluster);

  Invoca la función startCluster de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.NOT_FEASIBLE_ERROR si se produce algún error o BdcStatus.OK en caso contrario.

- BdcStatus restartCluster(Cluster cluster);

  Invoca la función restartCluster de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK en caso contrario.

- BdcStatus deleteCluster(Cluster cluster);

  Invoca la función deleteCluster de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.CLUSTER_NOT_SCHEDULED si se produce un error o BdcStatus.OK en caso contrario.

- BdcStatus createTask(Task task);

  Invoca la función createTask de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.TASK_ALREADY_CREATED o BdcStatus.NOT_FEASIBLE_ERROR si se produce un error o BdcStatus.OK en caso contrario.

- `BdcStatus deleteTask(Task task);`

  Invoca la función deleteTask de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.TASK_NOT_SCHEDULED si se produce un error o BdcStatus.OK en caso contrario.

- `BdcStatus suspendTask(Task task);`

  Invoca la función suspendTask de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK en caso contrario.

- `BdcStatus restartTask(Task task);`

  Invoca la función restartTask de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.OK en caso contrario.

- `BdcStatus startTask(Task task);`

  Invoca la función startTask de nivel de control de tiempo de ejecución.
  Devuelve BdcStatus.NOT_FEASIBLE_ERROR si se produce algún error o BdcStatus.OK en caso contrario.

3.3 Capa de Control de Nivel de Calidad

La capa de control de nivel de calidad establece y modifica las cuotas de uso de recurso de cada una de las tareas y clústeres del sistema para mantener que la calidad de las aplicaciones se mantenga independientemente de las variaciones en la disponibilidad de los recursos del sistema.

La capa de control de nivel de calidad está divido en cuatro componentes que pueden ser implementadas independientemente:

3.3.1 Admission Component

La componente de admisión de la capa de control de nivel de calidad es la encargada de verificar si la modificación de los niveles de calidad actuales son factibles.

- `QLStatus checkSystemConfiguration(Collection applications, Collection qualityLevels);`

  Prueba si la configuración de las aplicaciones que le pasa como parámetro es realizable.
  Devuelve QLStatus.OK o QLStatus.QUALITY_CONFIGURATION_NOT_FEASIBLE;

- `QLStatus checkApplicationConfiguration(Application application, QualityLevel level);`

  Prueba si es realizable la ejecución de la aplicación con el nivel de calidad pasado como parámetro.

```
Devuelve QLStatus.OK o
QLStatus.APPLICATION_CONFIGURATION_NOT_FEASIBLE;
```

- `QLStatus selectCriteria(SelectCriteria criteria);`

```
Establece un criterio de selección.
Devuelve QLStatus.OK.
```

3.3.2 Monitorization Component

La componente de monitorización de la capa de control de nivel de calidad es la encargada de iniciar la monitorización de los niveles de calidad de las aplicaciones.

- `QLStatus initSystemMonitorizationInfo();`

```
Inicializa la información de monitorización del sistema.
Devuelve QLStatus.OK.
```

- `SystemLoad updateSystemMonitorizationInfo();`

```
Devuelve la carga total actual del sistema.
```

- `QLStatus balanceSystemLoad(SystemLoad load);`

```
Realiza un balance de carga para intentar conseguir una carga
del sistema como la que se le pasa como parámetro.
Devuelve QLStatus.OK o QLStatus.SYSTEM_LOAD_UNREACHABLE.
```

- `QLStatus setMonitoringPeriod(Time period);`

```
Establece el período de monitorización.
Devuelve QLStatus.OK.
```

3.3.3 Application Interface Component

La componente de interfaz de aplicación de la capa de control de nivel de calidad es la encargada de comunicarse con las aplicaciones para que éstas aumenten su nivel de calidad o lo disminuyan en función de la disponibilidad de recursos y, por lo tanto, de la calidad que esté en condiciones de ofrecer el sistema.

- `QLStatus sendCommandToApplication(Application application, QLCommand command);`

```
Envía un comando de calidad de servicio a una aplicación.
Devuelve QLStatus.OK o QLStatus.COMMAND_NOT_EXECUTED.
```

- `QLStatus receiveCommandFromApplication(QLCommand command);`

```
Recibe un comando de una de las aplicaciones.
Devuelve QLStatus.OK o QLStatus.COMMAND_NOT_EXECUTED.
```

- `QLStatus sendCommandToSystem(QLCommand command);`

```
Envía un comando al nivel de control de cuotas de uso de
recurso.
Devuelve QLStatus.OK o QLStatus.COMMAND_NOT_SENT;
```

- `QLStatus receiveCommandFromSystem(Application application, QLCommand command);`

  ```
  Recibe un comando del nivel de control de cuotas de uso de
  recursos para una aplicación en particular.
  Devuelve QLStatus.OK o QLStatus.COMMAND_NOT_RECEIVED;
  ```

3.3.4 Settings Component

La componente de interfaz de aplicación de la capa de control de nivel de calidad es la encargada de lanzar y parar aplicaciones en el sistema.

- `QLStatus setSystemConfiguration();`

  ```
  Establece la configuración del sistema que previamente se ha
  probado.
  Devuelve QLStatus.OK o QLStatus.CONFIGURATION_NOT_FEASIBLE.
  ```

- `QLStatus launchApplication(Application application, QualityLevel level);`

  ```
  Lanza una aplicación el nivel de calidad especificado.
  Devuelve QLStatus.OK o QLStatus.APPLICATION_NOT_LAUNCHED.
  ```

- `QLStatus stopApplication(Application application);`

  ```
  Detiene la ejecución de la aplicación pasada como parámetro.
  Devuelve QLStatus.OK.
  ```

3.4 Capa de Control de Calidad de Servicio

La capa de control de calidad de servicio es la encargada de modificar la calidad ofrecida por las aplicaciones en función de los deseos del operador del sistema.

La capa de control de calidad de servicio está divido en cuatro componentes que pueden ser implementadas independientemente:

3.4.1 Admission Component

La componente de admisión de la capa de control de calidad de servicio es la encargada de verificar si el nivel de calidad de servicio seleccionado es factible en el sistema.

- `QosStatus checkSystemConfiguration(Collection applications, Collection qualityLevels);`

  ```
  Prueba si la configuración de las aplicaciones que le pasa
  como parámetro es realizable.
  Devuelve QosStatus.OK o
  QosStatus.QUALITY_CONFIGURATION_NOT_FEASIBLE;
  ```

- `QosStatus checkApplicationConfiguration(Application application, QualityLevel level);`

  Prueba si es realizable la ejecución de la aplicación con el nivel de calidad pasado como parámetro.
  Devuelve QosStatus.OK o
  QosStatus.APPLICATION_CONFIGURATION_NOT_FEASIBLE;

- `QosStatus checkSystem (Collection applications);`

  Prueba si es realizable la ejecución de las aplicaciones que le pasan como parámetro.
  Devuelve QosStatus.OK o QosStatus. CONFIGURATION_NOT_FEASIBLE;

- `QosStatus checkApplication (Application application);`

  Prueba si es realizable la ejecución de la aplicación.
  Devuelve QosStatus.OK o QosStatus.APPLICATION_NOT_FEASIBLE;

3.4.2 Monitorization Component

La componente de monitorización de la capa de control de calidad de servicio es la encargada de verificar la carga total del sistema.

- `SystemLoad updateSystemMonitorizationInfo();`

  Devuelve la carga total actual del sistema.

- `QosStatus balanceSystemConfiguration(SystemLoad load);`

  Realiza un balance de carga para intentar conseguir una carga del sistema como la que se le pasa como parámetro.
  Devuelve QosStatus.OK o QosStatus.SYSTEM_LOAD_UNREACHABLE.

3.4.3 User Interface Component

La componente de interfaz de usuario de la capa de control de calidad de servicio es la encargada de comunicarse con el operador del sistema.

- `QosStatus getUserRequest(Collection application, Collection qualityLevels, QLCommands command);`

  Recibe una lista de comandos a ejecutar por parte del usuario.
  Devuelve QosStatus.OK o QosStatus.COMMAND_NOT_EXECUTED.

- `QosStatus notifyUser(Notification notification);`

  Envía una notificación al usuario.
  Devuelve QosStatus.OK o QosStatus.NOTIFICATION_NOT_DELIVERED.

3.2.4 Settings Component

La componente de configuración de la capa de control de calidad de servicio es la encargada de lanzar y parar aplicaciones en el sistema.

- `QosStatus setSystemConfiguration();`

Establece la configuración del sistema que previamente se ha probado.
Devuelve QosStatus.OK o QosStatus.CONFIGURATION_NOT_FEASIBLE.

- QosStatus launchApplication(Application application, QualityLevel level);

  Lanza una aplicación el nivel de calidad especificado.
  Devuelve QosStatus.OK o QosStatus.APPLICATION_NOT_LeUNCHED.

- QosStatus stopApplication(Application application);

  Detiene la ejecución de la aplicación pasada como parámetro.
  Devuelve QosStatus.OK

CAPITULO 4. PRUEBAS

Las pruebas se han realizado sobre un prototipo implementado usando la implementación de la RTSJ que provee la máquina virtual de TimeSys y sobre un prototipo de gestor de calidad de servicio implementado sobre una máquina virtual de Java estándar, en este caso una máquina basada en el JRE 1.3, que es la versión de Java estándar en la que se ha basado TimeSys para crear su máquina virtual de tiempo real.

Las pruebas se han orientado a la comparación del comportamiento y las prestaciones de estas dos implementaciones. Las prestaciones que se han tenido más en cuenta a la hora del diseño de las pruebas han sido la ejecución coherente con las prioridades especificadas para los hilos, el tiempo de creación de éstos, el tiempo de creación de distintos tipos de objetos (tanto del sistema como definidos por el usuario) y de tipos primitivos. Asimismo estas pruebas se han repetido para distintas cargas del procesador para poder ver cómo los algoritmos que se usan escalan conforme la cantidad de recursos del sistema va disminuyendo y conforme va aumentando el número de entidades en el sistema.

El principal objetivo de estas pruebas es validar empíricamente la arquitectura propuesta para entornos con máquina virtual. Independientemente de la plataforma de ejecución utilizada (máquina virtual de Java directamente sobre sistema operativo o máquina virtual de tiempo real) el objetivo es determinar que este tipo de arquitectura de gestión supone una sobrecarga comparable y mínima en comparación con los beneficios que una gestión organizada de la calidad de servicio aporta a la ejecución de un sistema.

1 Entorno de Pruebas

Para la realización de las pruebas se utilizaron dos entornos de pruebas distintos. La razón de usar estos dos entornos fue la limitación que existe cuando se usa la maquina virtual de Java sobre el sistema operativo que impide tener el control directo y real sobre los recursos (por ejemplo, saber en qué momento un hilo accede al procesador).

Para poder tener acceso a esta información se usó como entorno de pruebas primeramente una implementación de la RTSJ que provee la máquina virtual Jamaica y que permite saber el número exactos de ciclos de de CPU que un hilo ha ejecutado. Este cálculo se puede obtener tanto teniendo en cuenta el código que se ejecuta de forma nativa como ignorando esos ciclos.

El segundo entorno de prueba fue la implementación estándar de la RTSJ. Este entorno permite conocer las prestaciones del sistema en su modo normal de funcionamiento, en el que hay una máquina virtual entre el sistema operativo y la aplicación de cliente. Esta máquina virtual puede imponer ciertos retardos en la ejecución de las tareas del sistema y es importante realizar las pruebas para ver el grado en que estos retardos pueden afectar a las prestaciones del sistema.

La máquina virtual Jamaica genera código nativo para el sistema operativo de tiempo real QNX Neutrino por lo que fue necesario instalar ese sistema operativo en la máquina de pruebas. Para la realización de las pruebas se instalaron en la misma máquina los sistemas operativos QNX Neutrino y Red Hat 9.0 de manera que se pudieran probar las prestaciones

del código generado tanto de manera nativa como usando una máquina virtual entre el sistema operativo de tiempo real y la aplicación de usuario.

La máquina en la que se instalaron los entornos de pruebas fue una máquina poco potente para que fuera mucho más fácil saturar sus recursos y así probar el comportamiento del sistema en su límite de funcionamiento.

Características Hardware de la máquina:

Procesador:	AMD k-7 800 MHz
Memoria RAM:	256 SDRAM

Características Software de la máquina:

Sistema Operativo:	Red Hat 9.0 con TimeSys
Versión RTSJ:	1.0.2

Sistema Operativo:	Neutrino RTOS 6.3.2
Versión RTSJ:	1.0.2

2 Pruebas de prioridad de hilos

Usando el planificador de hilos creado se realizaron varias pruebas con el objetivo de determinar el comportamiento de la implementación basada en la RTSJ respecto a los recursos asignados a cada uno de los hilos en ejecución. Estos recursos dependen de la prioridad de cada uno de los hilos.

Las pruebas realizadas fueron comparativas, es decir, ser realizaron pruebas para validar el funcionamiento de la implementación del sistema de gestión de la calidad de servicio basada en la RTSJ y para la implementación basada en el JDK 1.3 estándar de Java y posteriormente se compararon los resultados.

Las pruebas que se realizaron usando una tarea sumamente avariciosa en lo que respecta al tiempo consumido de procesador. Esta tarea era un simple bucle de autoincremento. El objetivo del uso de esta tarea es ver el comportamiento del sistema en escenario de gran carga del procesador y saber si el planificador realmente concede más recursos a los hilos que tienen una mayor prioridad.

Una vez terminadas todas las tareas se realizaban cálculos del tiempo total de procesador de que había dispuesto cada una de las tareas, con lo que se estaba en disposición de determinar cuál había sido el funcionamiento del sistema para el escenario en que se había realizado la prueba.

Para determinar el comportamiento del sistema se llevaron a cabo varias pruebas distintas con diferente número de hilos cada una tanto para la implementación basada en RTSJ como para la implementación basada en el JDK 1.3. Las pruebas y sus resultados se explican en los siguientes apartados.

2.1 Prueba con hilos de la misma prioridad

La primera prueba realizada fue una prueba en que todas las tareas que se ejecutaban en el sistema de gestión tenían la misma prioridad que tenía la prioridad máxima posible.

Se configuró la tarea a ejecutar de manera que si dispusiera del procesador a tiempo completo tardaría 2 minutos y 22 segundos en terminar de ejecutar el bucle de que constaba la tarea. El tiempo fue el mismo usando hilos de tiempo real que usando hilos estándar del JDK 1.3.

Con esta tarea se realizó una primera prueba usando cuatro tareas. Dos de las tareas tenían mayor prioridades que las dos restantes.

Los resultados de los cuatro hilos para la implementación basada en la RTSJ fueron:

Prioridad del Hilo	Tiempo CPU Tarea (flops)	Tiempo Total Tarea (flops)
20	116283073613	116283090386
20	116283073613	116283090386
11	116283073613	116283090386
11	116283073613	116283090386

Los resultados de los cuatro hilos para la implementación basada en JDK 1.3 fueron:

Prioridad del Hilo	Tiempo CPU Tarea (flops)	Tiempo Total Tarea (flops)
4	116569015974	116569043232
4	116282844044	116282863537
2	117667404602	117667423018
2	117044308035	117044326032

Como se puede ver en los resultados la implementación basada en RTSJ reparte los ciclos de CPU de una forma ecuánime ya que todos los hilos pueden acceder a los mismos ciclos de reloj. No obstante, a pesar de que accedan al mismo número de ciclos de reloj, los hilos de menor prioridad deben esperar a que los hilos de mayor prioridad liberen la CPU para acceder a ella.

Por el contrario se puede observar fácilmente que los hilos estándar de Java tienen un comportamiento parecido pero no pueden garantizar esa ecuanimidad en el acceso a la CPU. A pesar de que no exista esa ecuanimidad sí se respetan las prioridades y los hilos de menor prioridad no pueden acceder a la CPU hasta que los hilos de mayor prioridad la liberan.

2.2 Prueba con hilos de la distinta prioridad

Esta prueba se repitió usando nueve hilos en vez de cuatro, de manera que cada hilo tuviera una prioridad distinta. En este caso los resultados fueron los semejantes. Los resultados se muestran a continuación:

Prioridad del Hilo	Tiempo CPU Tarea (flops)	Tiempo Total Tarea (flops)
35	116384253014	116284271197
32	116384253014	116284271197
29	116384253014	116284271197
26	116384253014	116284271197
23	116384253014	116284271197
20	116384253014	116284271197
17	116384253014	116284271197
14	116384253014	116284271197
11	116384253014	116284271197

Los resultados de los diez hilos para la implementación basada en JDK 1.3 fueron:

Prioridad del Hilo	Tiempo CPU Tarea (flops)	Tiempo Total Tarea (flops)
10	116295885198	11687928307
9	116287747068	11687928307
8	116288226912	11687928307
7	116291656881	11687928307
6	116292033815	11687928307
5	116287172521	11687928307
4	116298578468	11687928307
3	116287904260	11687928307
2	117571107598	117571132614

Como se ve en los resultados el comportamiento de la implementación basada en la RTSJ se mantiene invariante, ya que el planificador de la RTSJ concede exactamente la misma cantidad de ciclos de reloj a todas las tareas. Por el contrario, se puede ver en el caso de la implementación estándar de Java que el tiempo de CPU de la tarea (tiempo de la ejecución del código no nativo) varía pero, no obstante, el tiempo total de la tarea (suma de los tiempos de ejecución del código nativo y no nativo) es constante.

La variación del valor del hilo de prioridad 2 es debida a que los procesos de interfaz de usuario tienen esa misma prioridad, por lo que ese hilo no tiene la CPU totalmente a su disposición como en el caso de los hilos de mayor prioridad.

Este comportamiento de los hilos de la implementación estándar de Java es importante, porque, aunque parezca que los hilos tienen un igual acceso a la CPU realmente no lo tienen, ya que los ciclos de CPU ejecutados de código no nativo, es decir, dentro de la máquina virtual, no son los mismos.

3 Pruebas de carga del sistema

El resto de pruebas realizadas en el sistema se realizaron para verificar el comportamiento del sistema en caso de que el sistema estuviera muy cargado de tareas, aunque estas tareas no saturaran los recursos del sistema.

Para realizar estas pruebas se fueron añadiendo progresivamente tareas al sistema para comprobar la escalabilidad de los algoritmos de creación de objetos y de utilización de memoria conforme aumentaba el volumen de datos que el sistema debía tratar.

Las pruebas realizadas se centraron en ver cómo variaban los tiempos de creación diferentes tipos de objetos. Para ello se usaron dos escenarios, un escenario en que las tareas que se creaban que consumían recursos de procesador y otro en que las tareas, una vez instanciados los objetos se dormían y, por lo tanto, no cargaban el sistema. Al igual que antes se realizaron las pruebas para la implementación basada en la RTSJ y para la implementación basada en el JDK 1.3.

3.1 Pruebas de carga con uso del procesador

Una característica importante en los sistemas de tiempo real es el tiempo de instanciación de un hilo, en este caso se midió la escalabilidad del algoritmo de creación de hilos. En la figura 4.3.1.1 se muestra una gráfica comparativa del tiempo de creación de hilos en milisegundos.

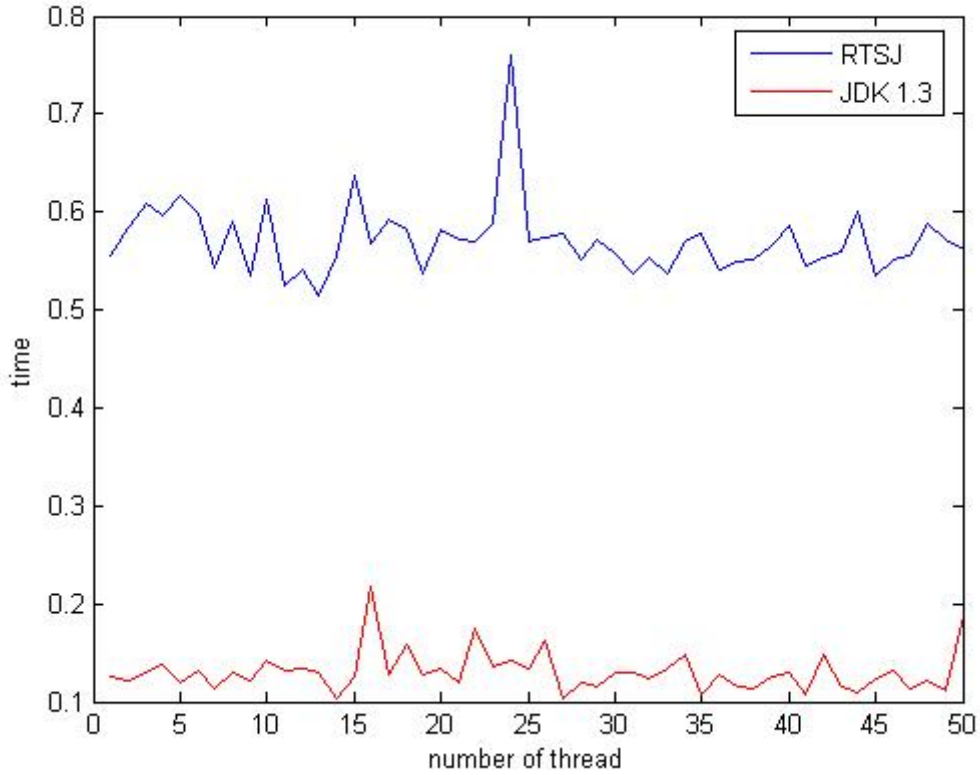

Figura 4.3.1.1 Tiempo de creación de hilos

Como se puede ver en la figura el tiempo de creación de hilos en la implementación basada en la RTSJ es mucho mayor que en la implementación basada en el JDK 1.3. Esto es debido a que los RealtimeThread instancian una serie temporizadores necesarios para el funcionamiento de las tareas periódicas o esporádicas.

No obstante, a pesar de esta diferencia los tiempos de creación de hilos se mantienen constantes independientemente del número de hilos que haya en el sistema, esta característica no se mantiene constante respecto a la memoria total que hay en el sistema, ya que conforme va aumentando la memoria en el sistema se tarda más en instanciar un RealtimeThread, como se puede ver en la figura 4.3.1.2, en la que se muestra el tiempo de creación de hilos en milisegundos dependiendo de la memoria usada por el sistema.

Como se puede ver en esta figura el tiempo de creación de un RealtimeThread aumenta linealmente conforme aumenta la memoria usada por la máquina virtual. Lo más curioso de la gráfica es que sólo ocurre en ciertos hilos, no en todos, aunque claramente se ve una relación lineal en el incremento del tiempo de creación de los hilos.

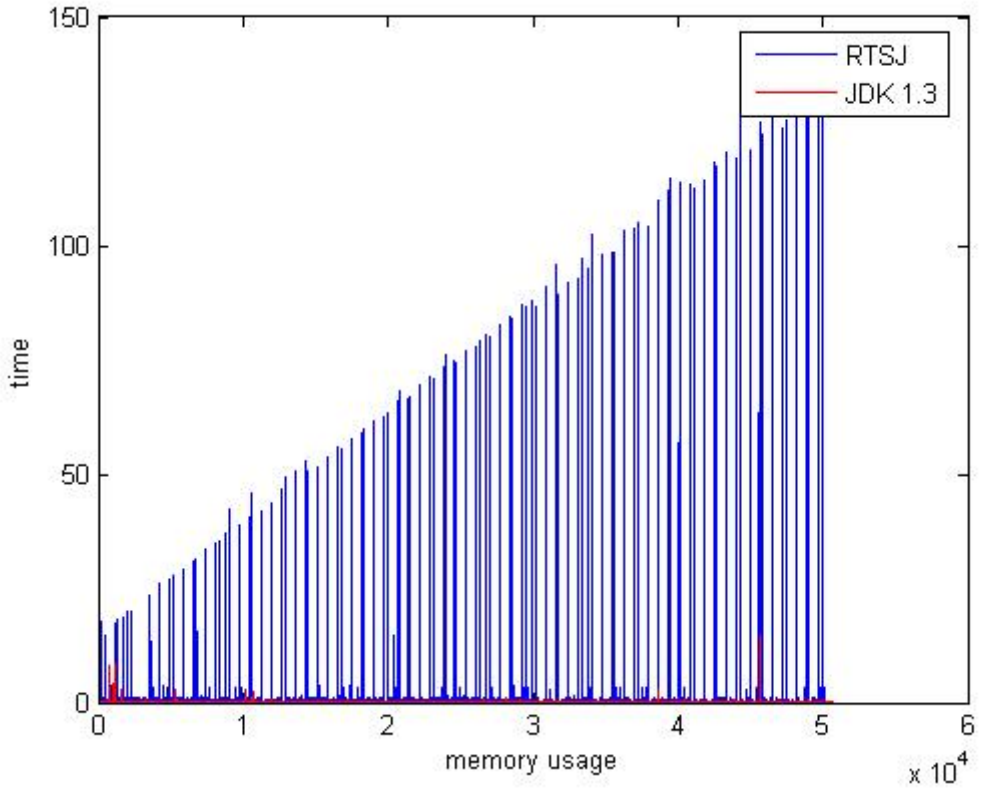

Figura 4.3.1.2 Tiempo de creación de hilos

La siguiente característica que se midió fue el tiempo de creación de tipos básicos del lenguaje, en este caso se midió el tiempo de creación de un entero, de un string y de una clase definida por el usuario. En las siguientes imágenes se muestran gráficas comparativas de los tiempos empleados para realizarlo.

En las gráficas 4.3.1.3 a-d se puede ver que el tiempo de creación en milisegundos de tipos básicos del lenguaje es semejante en ambas implementaciones, como cabía esperar, ya que ambas implementaciones se han realizado en el mismo lenguaje y están basadas en la misma versión de la máquina virtual de Java.

En esas mismas gráficas se puede ver que el tiempo de creación de objetos de usuario también es semejante en ambas implementaciones. Lo único distinto es el tiempo de creación del primer objeto, que es considerablemente más costoso en la implementación basada en la RTSJ. El caso del primer objeto instanciado es distinto del resto ya que en este caso la máquina virtual lee el fichero .class en donde está almacenado el bytecode de la clase, sin embargo en el resto de las instanciaciones el bytecode ya está en memoria y no es necesario leer el fichero.

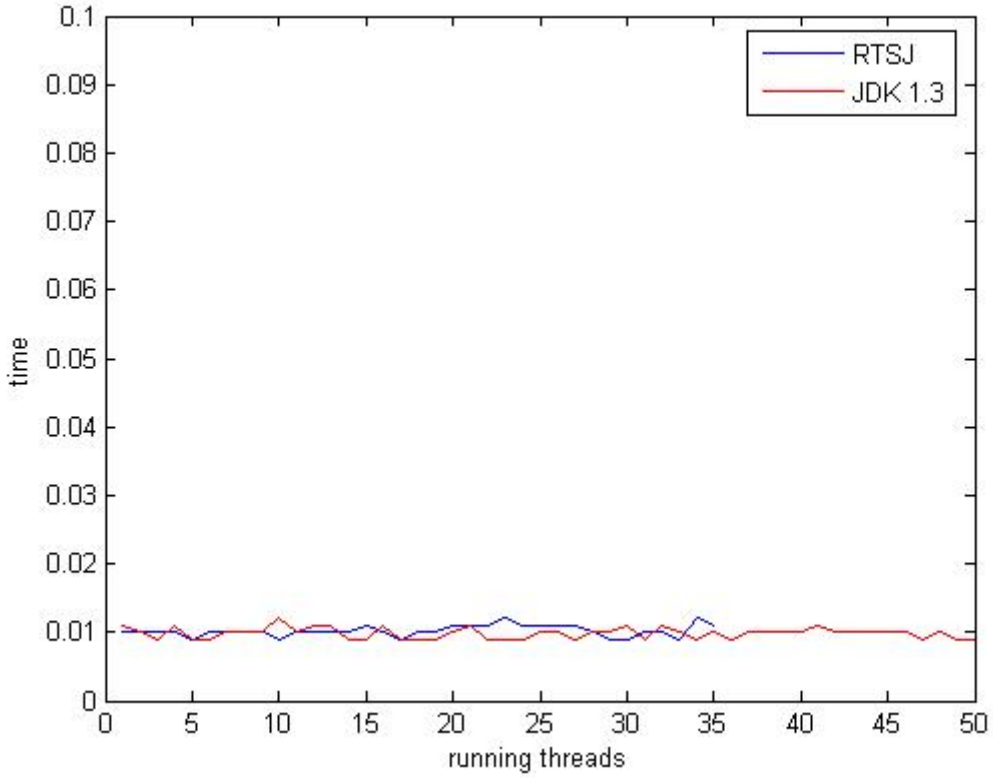

Figura 4.3.1.3 (a) Tiempo de creación de un entero

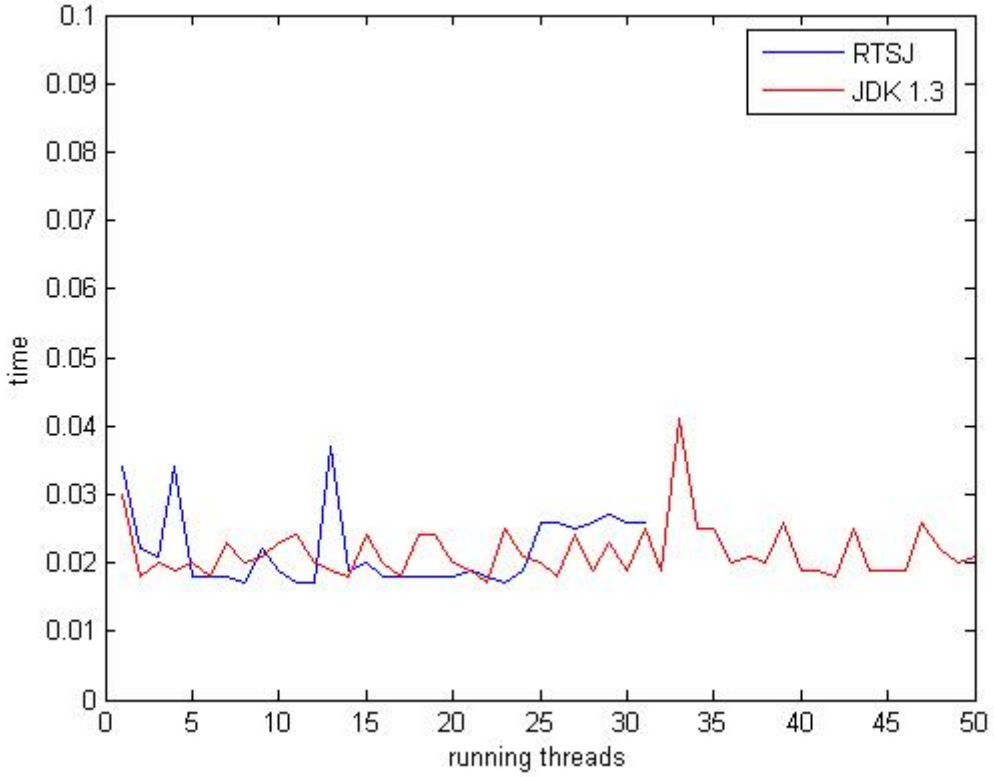

Figura 4.3.1.3 (b) Tiempo de creación de un string

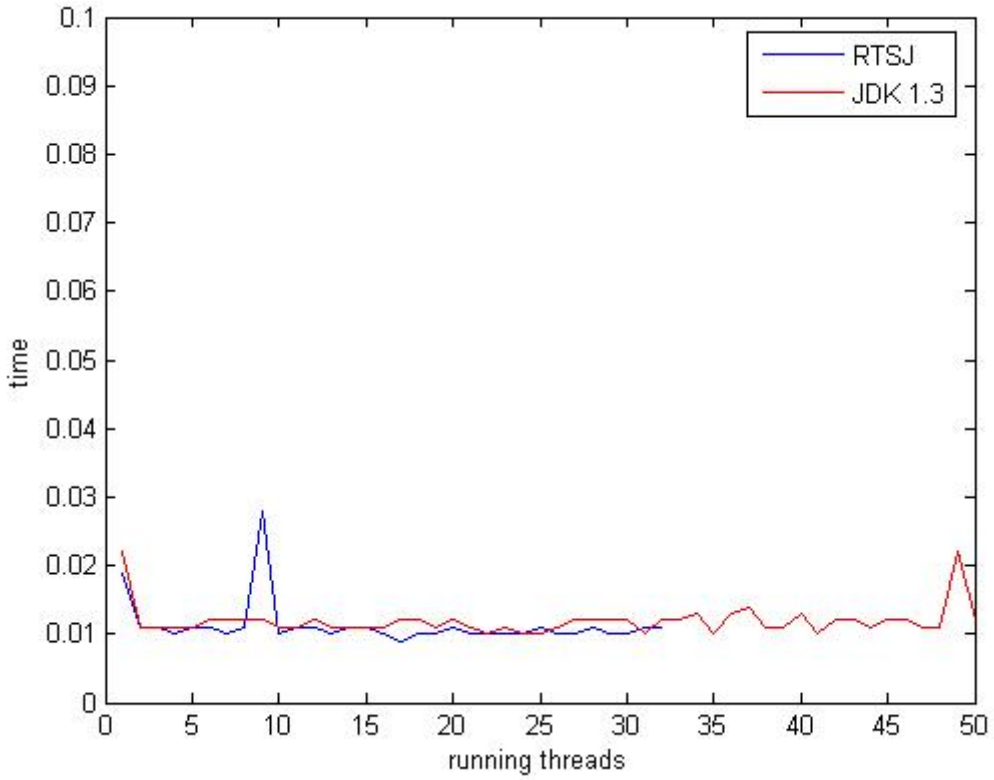

Figura 4.3.1.3 (c) Tiempo de creación de un objeto

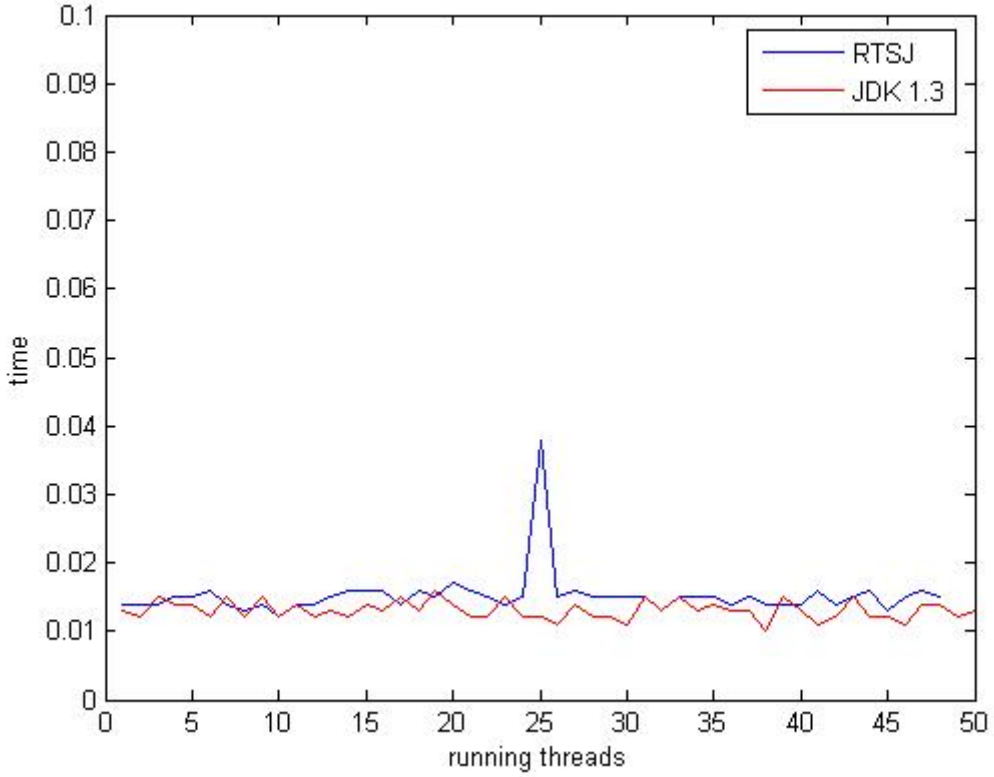

Figura 4.3.1.3 (d) Tiempo de creación de una clase definida por el usuario

Se realizaron pruebas también de tiempo empleado en la instanciación de objeto en los diferentes tipos de memoria que implementa la RTSJ. Estos tipos de memoria también se pueden emplear usando el JDK 1.3, por lo que se procedió a realizar las comparaciones pertinentes. A continuación se muestran las gráficas comparativas de los tiempos de creación de objetos en milisegundos en los diferentes tipos de memoria.

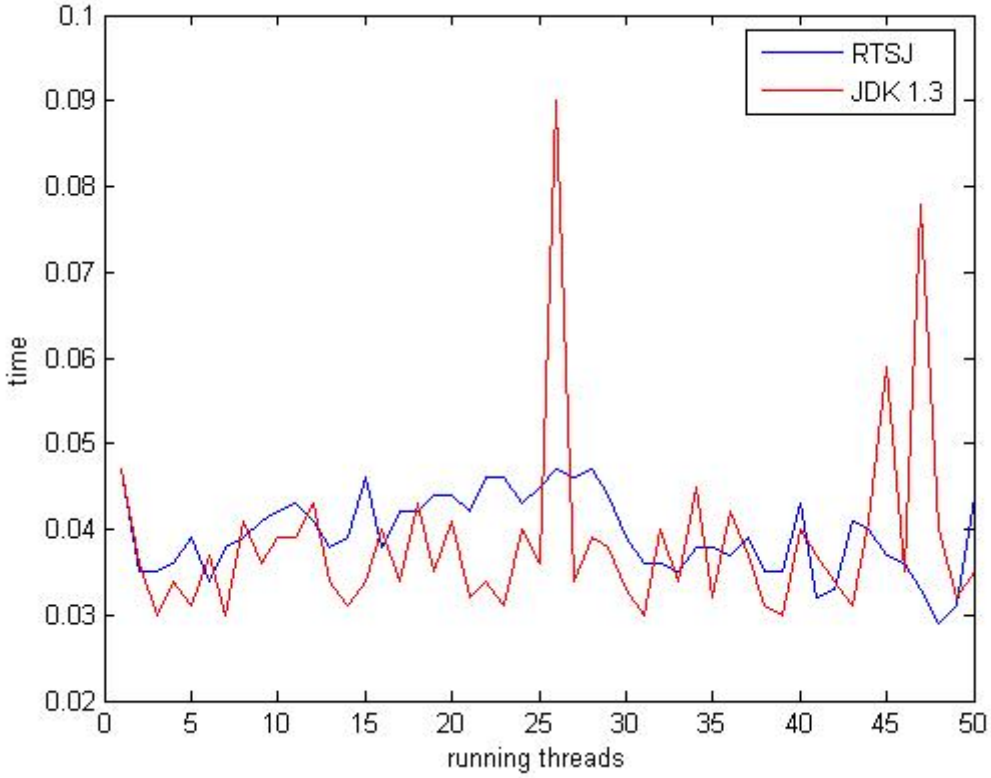

Figura 4.3.1.4 Tiempo de creación de un objeto en memoria heap

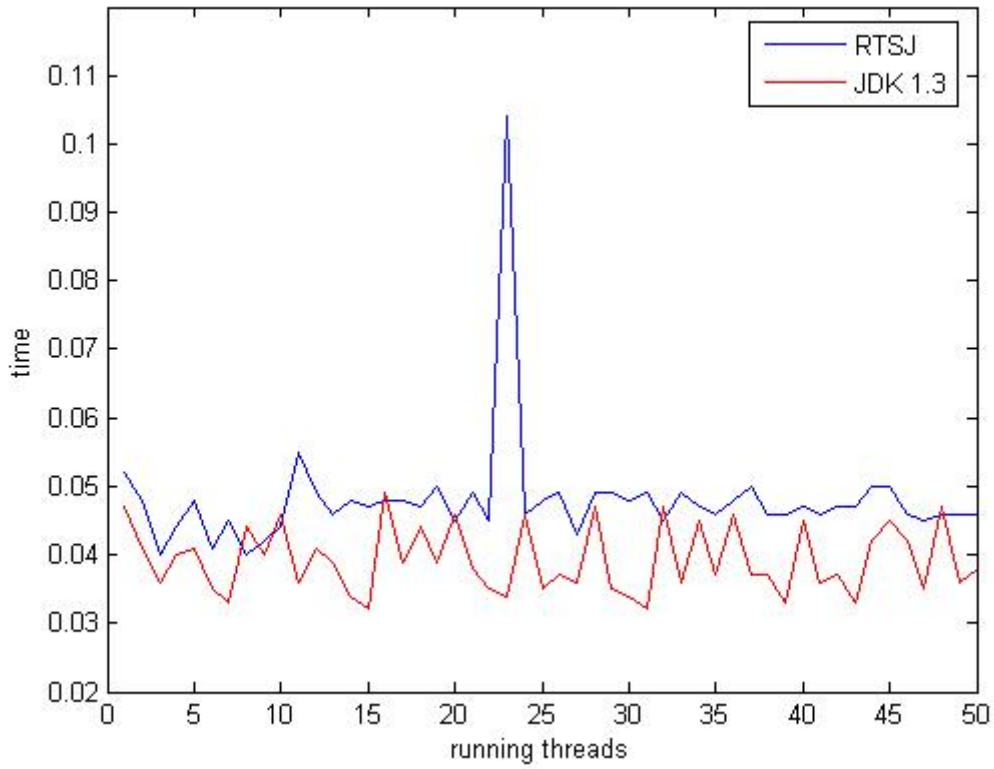

Figura 4.3.1.5 Tiempo de creación de objetos en memoria inmortal

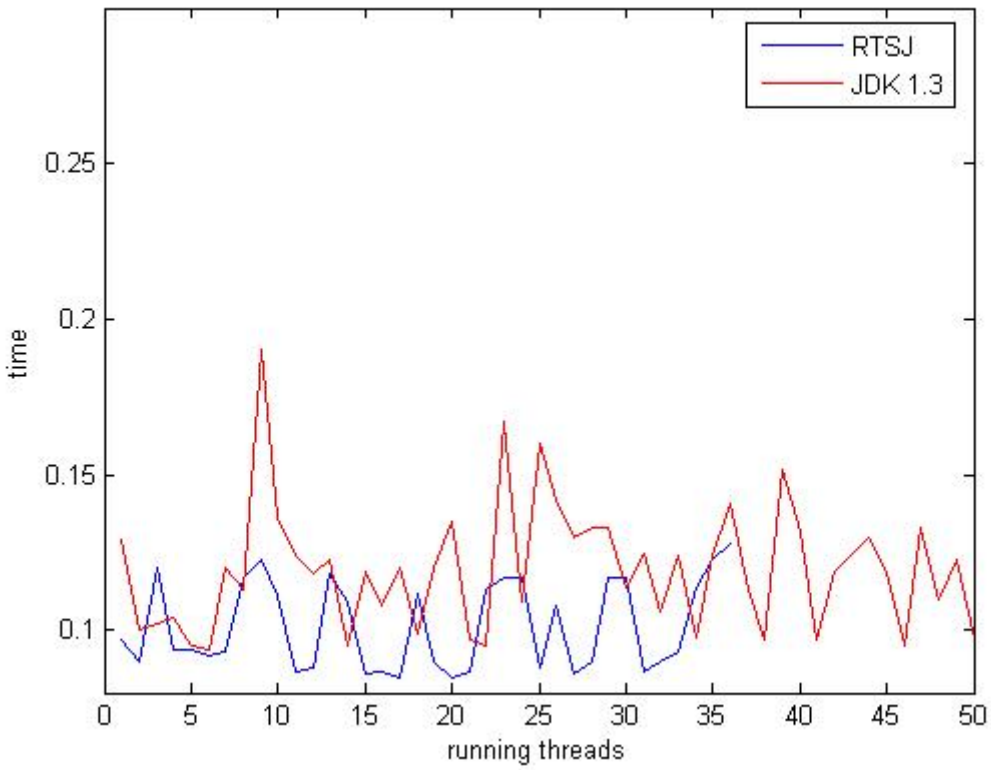

Figura 4.3.1.6 Tiempo de creación de objetos en memoria LT

Como se puede ver en las gráficas la implementación basada en la RTSJ es ligeramente más eficiente en el uso de la memoria que la implementación basada en el JDK 1.3. La excepción está en el uso de la memoria LT cuyas prestaciones son semejantes en ambos casos o ligeramente mejores en el caso de la implementación basada en JDK 1.3.

Por último se hizo un estudio de la memoria total consumida por cada una de las implementaciones en función del número de hilos existentes en el sistema. En la figura 4.3.1.8 se muestra claramente como, para una misma cantidad de datos usados por las tareas el sistema basado en la RTSJ usa una cantidad de memoria mucho mayor, aproximadamente un 15% más, que en el sistema basado en el JDK 1.3

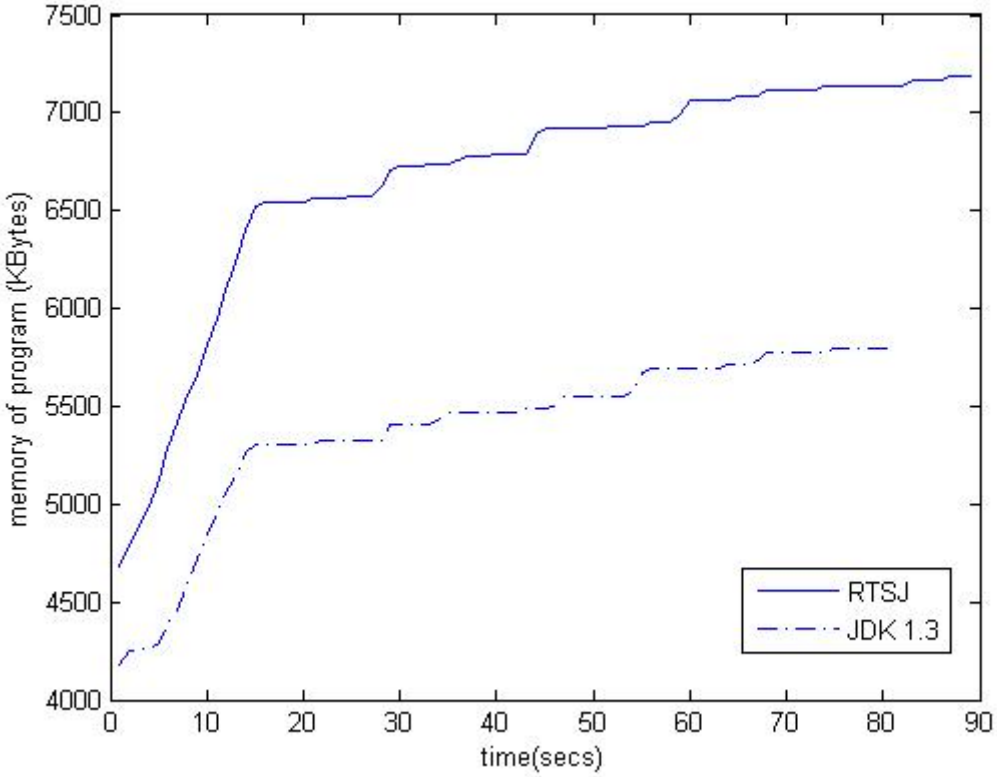

Figura 4.3.1.8 Memoria usada por ambas implementaciones

4 Pruebas de carga sin uso del procesador

En ausencia de uso de procesador el sistema de capaz de soportar una cantidad mucho mayor de hilos. No obstante esta cantidad de hilos es finita y está delimitada por ciertos límites. En el caso de la implementación basada en la RTSJ el número máximo de hilos es 256 y está limitado por el número máximo disponible de temporizadores en el sistema. En el caso de la implementación basada en el JDK 1.3 el número máximo de hilos en el sistema es 1024 y está limitado por la memoria máxima disponible por la librería nativa de hilos POSIX.

En la figura 4.4.1 se puede ver que el tiempo de creación de hilos en milisegundos en ambas implementaciones sigue siendo aproximadamente constante y no depende del número hilos en el sistema.

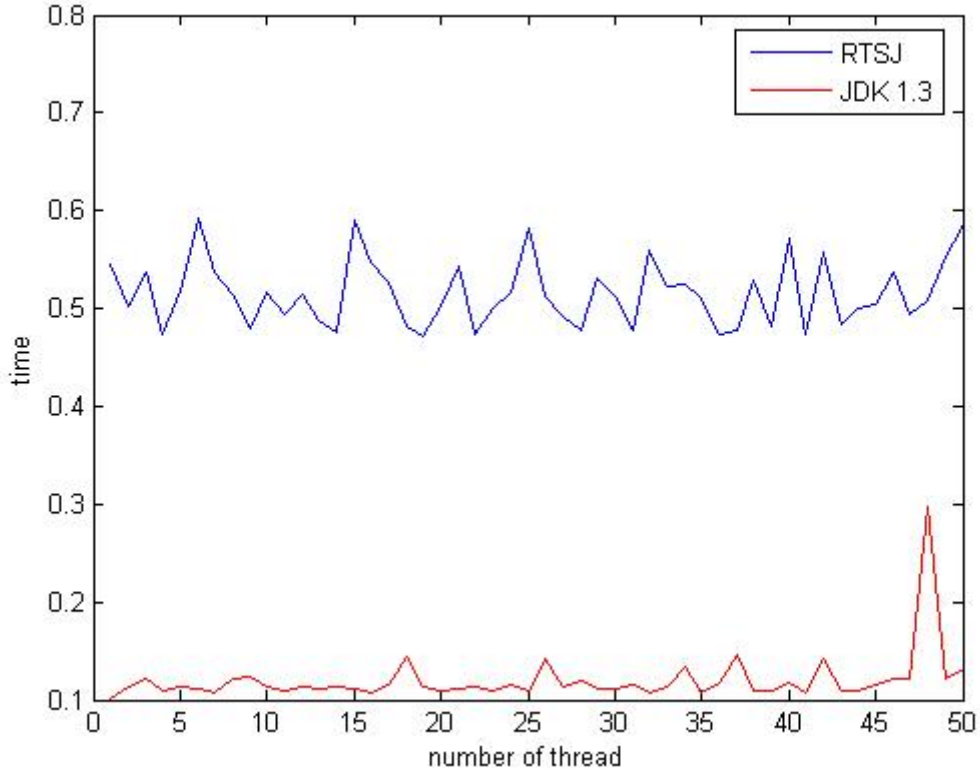

Figura 4.4.1 Tiempo de creación de hilos sin carga en el procesador

5 Protocolos de cambio de modo

La acción más costosa en tiempo y recursos que se puede invocar sobre un sistema de gestión de la calidad de servicio es un protocolo de cambio de modo, por lo tanto es necesario saber cuales son las prestaciones de los sistemas implementados ante las peticiones de cambio de modo. Para ello se realizaron pruebas para calcular el tiempo que se tardaban en realizar los cambios de modo usando un protocolo de cambio de modo inmediato y un protocolo de cambio de modo progresivo.

Ambos protocolos de cambio de modo cambian un nivel de calidad una aplicación por otro y arrancan y/o paran las tareas para que sólo se ejecuten las tareas que correspondan en cada nivel de calidad. La diferencia entre un protocolo y otro es que el protocolo inmediato realiza el procedimiento por acciones (parada de todos los hilos, reconfiguración de todos los hilos y rearranque de todos los hilos) mientras que el protocolo progresivo realiza el procedimiento por hilos (para cada hilo lo parada, reconfiguración y rearranque).

En las simulaciones realizadas se probó el caso peor, es decir, el caso en que hubiera que para todos los hilos y volver a rearrancarlos ya que es en este caso en el que el sistema tarda más tiempo en ejecutarse. A continuación se muestra el código desarrollado para realizar estas pruebas.

El protocolo de cambio de modo *inmediato* se corresponde con el siguiente código:

```
public QLStatus checkApplicationConfiguration(Application app, QualityLevel ql) {
        Vector v = new Vector(app.getClusters());

        for(int i = 0; i < v.size(); i++){
          Cluster c = (Cluster)v.elementAt(i);

          this.budgetLayer.settings.deleteCluster(c);
          this.budgetLayer.settings.createCluster(c);
          c.setClusterConfigurations(
                   ql.getConfiguration().getClusterConfigurations()
                   );
          this.budgetLayer.settings.startCluster(c);
        }
}
```

Por el contrario el protocolo de cambio de modo *progresivo* se corresponde con el siguiente código

```
public QLStatus checkApplicationConfiguration(Application app, QualityLevel ql) {
        Vector v = new Vector(app.getClusters());

        for(int i = 0; i < v.size(); i++){
          Cluster c     = (Cluster)v.elementAt(i);
          Vector  confs = ql.getConfiguration().getClusterConfigurations();
          Vector  tasks = c.getTasks();

          for(int j = 0; j < tasks.size(); j++) {
             Task t = (Task)tasks.elementAt(j);
             this.budgetLayer.settings.deleteTask(t);
             this.budgetLayer.settings.createTask(t);
             t.setTaskConfiguration ((TaskConfiguration)confs.elementAt(j));
             this.budgetLayer.settings.startTask(t);
          }
        }
}
```

En la figura 4.5.1 se pueden ver los tiempos en milisegundos empleados en los cambios de modo de un cluster compuesto de dos, tres, cuatro y cinco tareas. Como se puede ver en las gráficas el tiempo medio empleado en los cambios de modo es semejante con los dos protocolos de cambio de modo, teniendo valores medios de 39, 59, 79 y 99 milisegundos respectivamente. Como se puede inferir rápidamente de estos datos, en media, en realizar el cambio de modo de cada tarea se tarda 20 milisegundos. Estos veinte milisegundos se emplean en detener el hilo de la tarea, en arrancarlo nuevamente y en la reconfiguración de los parámetros de la tarea.

En la implementación realizada la reconfiguración de los parámetros de la tarea se reduce a la asignación de determinados valor a tres variables. Esto es lo que hace que ambos protocolos de cambio de modo tarden, en media, el mismo tiempo en ejecutarse. La mayor parte del tiempo empleado en el cambio del modo se realiza en la parada y posterior rearranque del hilo, lo que hace que el tiempo de reasignación sea despreciable.

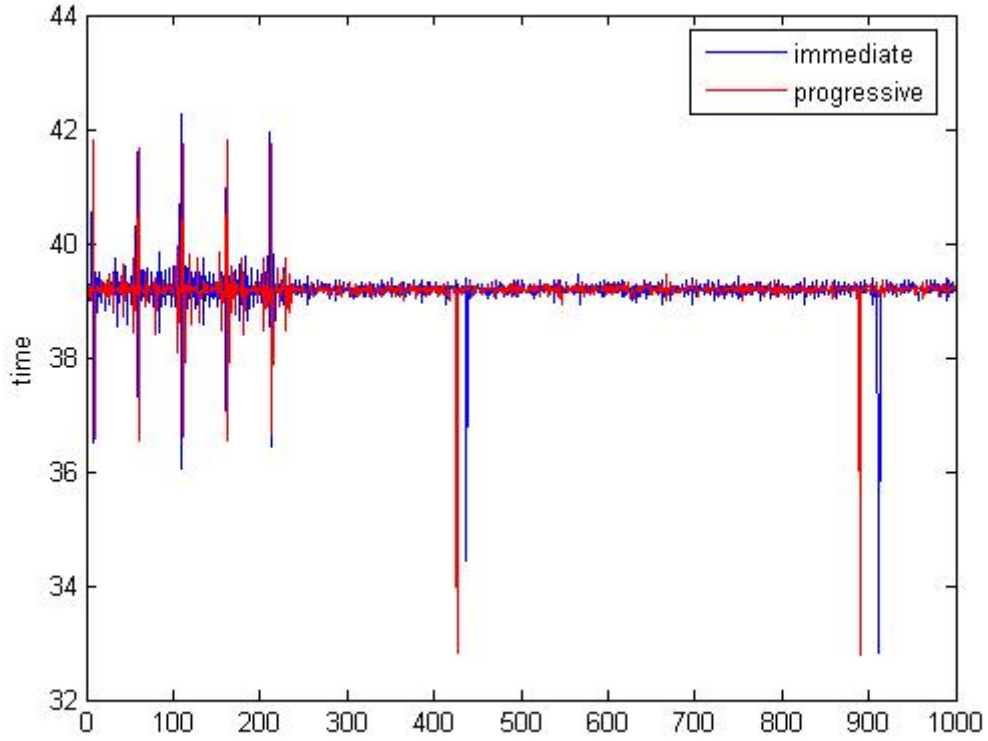

Figura 4.5.1 Tiempo empleado en los cambios de modo con 2 tareas

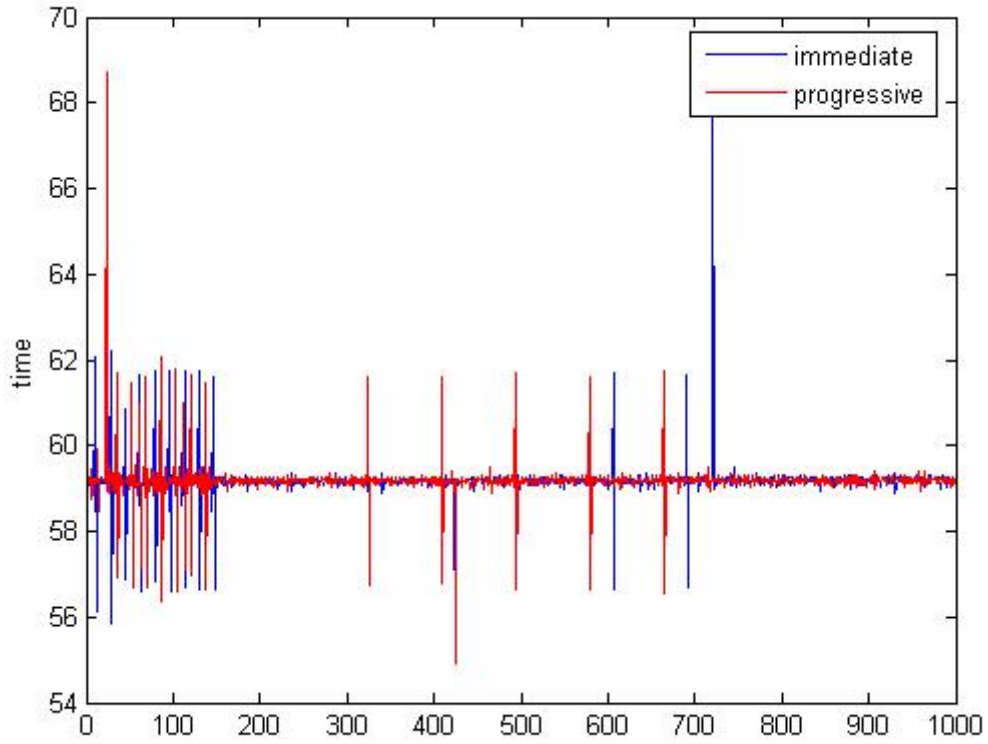

Figura 4.5.1 Tiempo empleado en los cambios de modo con 3 tareas

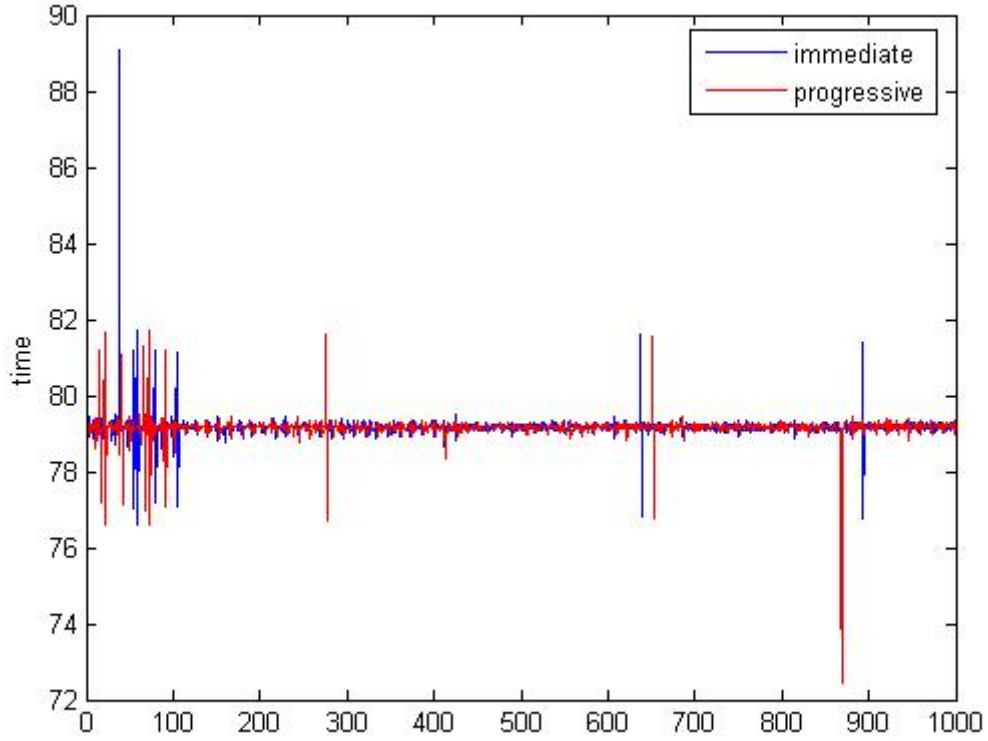

Figura 4.5.1 Tiempo empleado en los cambios de modo con 4 tareas

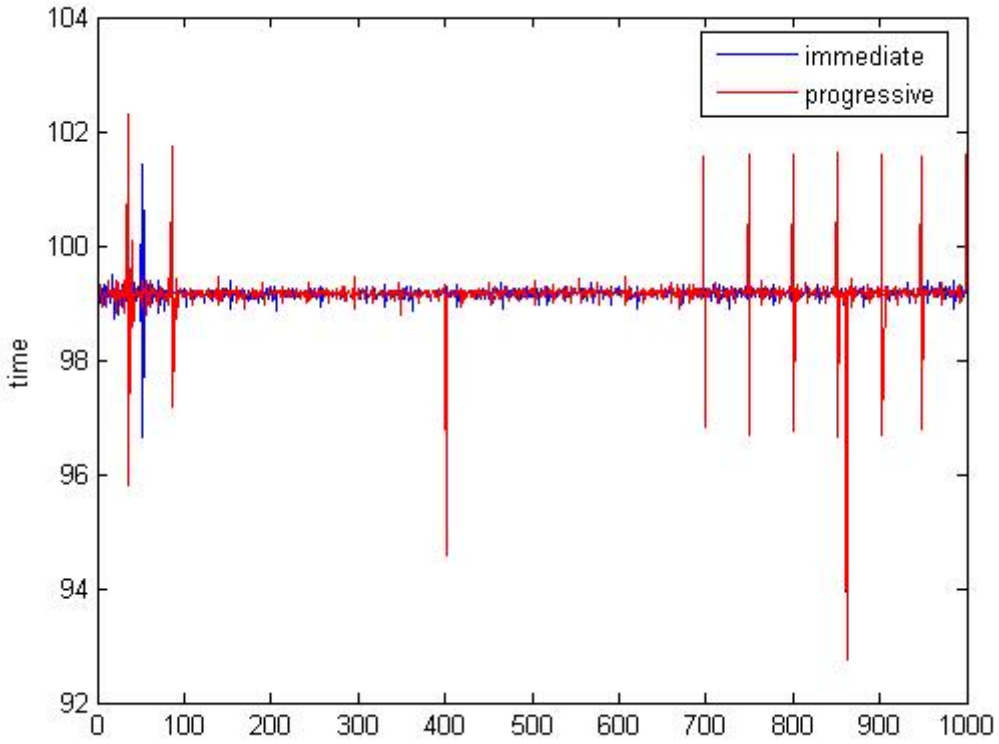

Figura 4.5.1 Tiempo empleado en los cambios de modo con 5 tareas

Con los datos obtenidos en las simulaciones se pueden realizar una comparación de las medias del tiempo de cambio de modo de cada uno de los protocolos. Esta comparación está representada en la figura 4.5.2 y, como se puede ver, las medias coinciden exactamente para las cuatro pruebas realizadas. Otro aspecto que cabe destacar en la figura es que los protocolos de cambio de modo tienen claramente una complejidad Θ(n).

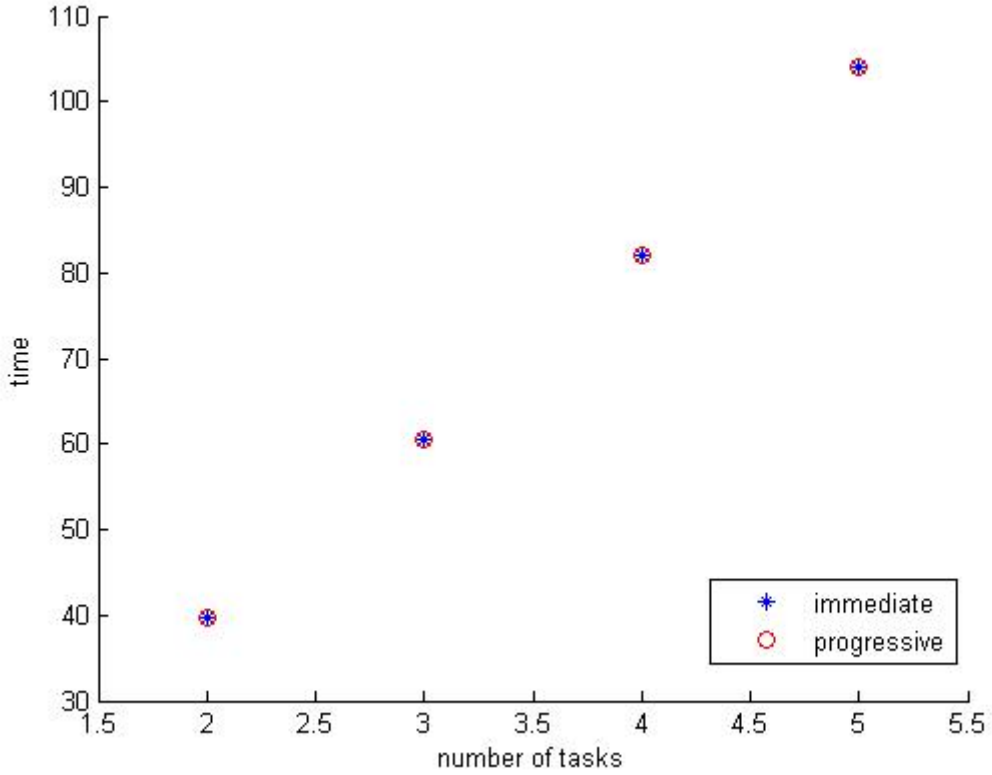

Figura 4.5.2 Escalabilidad de los protocolos de cambio de modo

CAPITULO 5. CONCLUSIONES

Una vez realizada y probada la implementación de la arquitectura del sistema de gestión de la calidad de servicio se pueden empezar valorar los resultados obtenidos. El objetivo principal de esta implementación era demostrar que la arquitectura era válida para ser usada sobre plataformas software basadas en una máquina virtual. La implementación ha demostrado que la arquitectura es válida y que funciona tanto en sistemas diseñados específicamente para tiempo real (RTSJ) como para los sistemas no diseñados específicamente para tiempo real (JDK estándar) aunque esto último no tenga utilidad práctica y sólo se haya implementado para tener una base sobre la que comparar.

Como era previsible, las plataformas software basadas en una máquina virtual tienen unas prestaciones ostensiblemente inferiores a las que tendría un sistema implementado de forma nativa ya que la introducción de la máquina virtual entre el sistema operativo y el sistema de gestión de la calidad de servicio supone una pérdida de velocidad de procesado (aunque también supone un aumento de la fiabilidad del sistema).

A pesar de esto y, como se puso de manifiesto en el capítulo introductorio, estas implementaciones siguen siendo una opción atractiva ya que el mismo código puede ser ejecutado sobre distintos sistemas operativos de tiempo real sin sufrir ninguna modificación. Esta característica por sí sola hace que estas implementaciones basadas en máquina virtual se deban tener en cuenta tanto desde el punto de vista de la relación calidad / coste del producto como desde el punto de vista de la escalabilidad y la evolución de los posibles sistemas implementados.

Conforme a las pruebas comparativas realizadas entre la implementación basada en la RTSJ y la implementación basada en el JDK estándar se puede concluir que los únicos aspectos en que la implementación basada en la RTSJ es manifiestamente inferior a la implementación basada en el JDK estándar es en el tiempo de instanciación de las tareas y en el uso de memoria del sistema. En el resto de prestaciones (véase tiempo de acceso a memoria y tiempo de instanciación de objetos en memoria) las prestaciones son similares en ambos casos. Con esto se puede concluir que la implementación de tiempo real en este caso es sólo ligeramente menos eficiente que la implementación convencional y que esta pérdida de prestaciones es asumible.

No obstante, aunque las prestaciones de la implementación de tiempo real sean comparables con las de la implementación convencional esto no significa que sean suficientes para un sistema comercial de tiempo real, ya que muchos de los sistemas comerciales de tiempo real requieren requisitos que actualmente no es capaz de cumplir ninguna de las dos implementaciones realizadas.

Las limitaciones actuales de la versión usada de la RTSJ (imposibilidad de detección de las pérdidas de plazo y de las sobre-ejecuciones y sus prestaciones actuales) impiden el uso de la implementación realizada en los sistemas comerciales actuales, pero, en un futuro, cuando se solventen esos problemas la implementación podría ser realizada exitosamente.

ANEXO I. Invocación de la máquina virtual TimeSys

A continuación se muestra un ejemplo de un script que podría usarse para inicializar las variables necesarias para la invocación del binario tjvm que implementa la versión 1.0.3 de la RTSJ.

```
#!/bin/bash


INSTALLDIR=/opt/timesys/rtsj-ri
LOCALDIR=`pwd`
CLASSPATH=${INSTALLDIR}/lib/foundation.jar
MAIN_CLASS=$1

shift

while [[ $1 = "-l" ]]
do
  shift
  CLASSPATH=${CLASSPATH}:${LOCALDIR}/$1
  shift
done


MEMCONFIG=
TIMESCALE="-Djavax.realtime.tck.timeScale=40"
SCOPESIZE="-Djavax.realtime.tck.bigScopeSize=2000000"
SMALLSCOPE="-Djavax.realtime.tck.smallScope=100"
MEDIUMSCOPE="-Djavax.realtime.tck.mediumScope=2000"
LARGESCOPE="-Djavax.realtime.tck.largeScope=28000"

cd ${INSTALLDIR}/bin;
export LD_ASSUME_KERNEL=2.4.1;
export IMMORTAL_SIZE=6000000;

./tjvm -version

./tjvm -Xms10M ${MEMCONFIG} ${TIMESCALE} ${SCOPESIZE} ${SMALLSCOPE} ${MEDIUMSCOPE}
${LARGESCOPE} -Xbootclasspath=${CLASSPATH} ${MAIN_CLASS} $@
```

Un ejemplo de invocación de este script sería:

```
~>./runTJVM Example –l library1.jar –l library2.jar mainClass
```

En este ejemplo *library1.jar* y *library2.jar* son dos archivos JAR que contienen el código a ejecutar. Dentro de uno de estos archivos JAR debe estar la clase *mainClass* que será la clase cuyo método main se invoque.

Para compilar el código Java durante el desarrollo de este proyecto se ha usado el entorno de desarrollo Eclipse añadiendo la librería *foundation.jar* en la lista de librerías a incluir en la compilación(*Library Path*) y configurando la opción *JDK Compliance* en versión 1.3.

ANEXO II. Código Fuente del planificador y de las Tareas usadas en las pruebas

Código fuente de la superclase de la tarea usada para las pruebas:

```
public abstract class WaitTask extends RealTimeTask {
	protected boolean mustSleep = false;

	protected SchedulerRealTimeTask waitScheduler;

	public WaitTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
			HighResolutionTime deadline, HighResolutionTime start,
			HighResolutionTime period) {
		super(DDBBid, priority, cost, deadline, start, period);
	}

	public WaitTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
			HighResolutionTime deadline) {
		super(DDBBid, priority, cost, deadline);
	}

	public WaitTask(Integer DDBBid, Priority priority, HighResolutionTime cost,
			HighResolutionTime deadline, HighResolutionTime minimumInterarrival) {
		super(DDBBid, priority, cost, deadline, minimumInterarrival);
	}

	public void sleepAsSoonAsPosible(){
		this.mustSleep = true;
	}

	public void setWaitScheduler(SchedulerRealTimeTask scheduler){
		this.waitScheduler = scheduler;
	}

	public synchronized void sleep(){
		while(mustSleep){
			try {
			  wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}

	public synchronized void awake(){
		this.mustSleep = false;
		this.notify();
	}

}
```

Código fuente del planificador usado en las pruebas:

```
public class SchedulerTask extends WaitTask {
	public static int quantum = 300;

	// SortedMap<Integer,LinkedList>
	SortedMap priorities;
	Vector finished;

	double total_ratio = 0;
	AbsoluteTime total_endtime;

	public SchedulerTask() {
	  priorities = new TreeMap();
	  finished = new Vector();
	}

	/**
	 * Add a new thread in the suitable queue depending on
```

```
 * the priority of thread
 */
public void addThread(WaitTask t){
  Integer priority = new Integer(t.getPriority());
  LinkedList queue;

  Log.RTdebug("Adding Thread: " + t.getName());

  if(!priorities.containsKey(priority)){
    queue = new LinkedList();
    priorities.put(priority, queue);
  }
  else queue = (LinkedList)priorities.get(priority);

  Stat s = new Stat(t);
  queue.addLast(s);
}

public void run(){
  // start all threads and sleep it
  Vector lists = new Vector(this.priorities.values()); // Collection<LinkedList>
  for(int i = 0; i < lists.size(); i++ ){
   LinkedList l = (LinkedList)lists.elementAt(i); //LinkedList<Stat>

   Iterator it = l.iterator();

    while(it.hasNext()){
      Stat st = (Stat)it.next();

      st.thread.sleepAsSoonAsPosible();
      st.thread.start();

      st.initTime = Clock.getRealtimeClock().getTime();
    }
  }

  // wait for all threads to sleep
  try {
    Thread.sleep(2000);
  }
      catch (InterruptedException e) {
  }

  // while there is any thread active in system
  while(this.priorities.size() > 0){
    // get the highest priority
   Object key = this.priorities.lastKey();
    // get the queue of threads with that priority
    LinkedList l = (LinkedList)this.priorities.get(key); //LinkedList<Stat>

    // get the first element of queue
    Stat st = (Stat)l.remove(0);

    AbsoluteTime before = null;
    AbsoluteTime after = null;

    // awake the thread
    st.thread.awake();
    before = Clock.getRealtimeClock().getTime();
     // sleep the quantum
     try {
       Thread.sleep(quantum);
     }
          catch (InterruptedException e) {
       e.printStackTrace();
      }
      // if thread is still alive we ask for it to sleep
      // scheduler sleeps while waits for it
      if(st.thread.isAlive()){
        st.thread.sleepAsSoonAsPosible();
        sleep();
      }
      after = Clock.getRealtimeClock().getTime();

      // compute cputime
      st.cputime += (
        (after.getMilliseconds() * 1000000 + after.getNanoseconds())
```

```
                -
                 (before.getMilliseconds() * 1000000 + before.getNanoseconds())
                );

              // if thread is still alive push into queue as the last element
              if(st.thread.isAlive()){
                l.addLast(st);
              }
              // thread has finished. Push into queue of finished threads
              else {
               AbsoluteTime now = Clock.getRealtimeClock().getTime();
               st.endTime = now;
               total_endtime = now;

               finished.add(st);
              }

              // if there is no more threads with that priority remove
              // that entry in priority list.
              if(l.size() == 0)
                this.priorities.remove(key);
           }

            // print results
            Log.RTdebug("Results: ");
            for(int i = 0; i < finished.size(); i++){
              Stat st = (Stat)this.finished.elementAt(i);

              long realtime = (
                  (st.endTime.getMilliseconds() * 1000000 + st.endTime.getNanoseconds())
                   -
                  (st.initTime.getMilliseconds() * 1000000 + st.initTime.getNanoseconds())
                );

              total_ratio += ((double)st.cputime
                  /
                   (double)(
                (total_endtime.getMilliseconds() * 1000000 + total_endtime.getNanoseconds())
                  -
                (st.initTime.getMilliseconds() * 1000000 + st.initTime.getNanoseconds())
                             ));

             Log.RTdebug("Thread  "+  st.thread.getName()  +  "("  +  st.thread.getPriority()  +  ")
cputime " + st.cputime +
              " real time " + realtime + " ratio " + ((double)st.cputime/(double)realtime));
             }

             Log.RTdebug("total cputime ratio " + total_ratio);
            }

            public synchronized void sleep(){
             this.mustSleep = true;
             super.sleep();
            }
}
```

Código fuente de la tarea usada en las pruebas:

```
public class TestTask extends WaitTask {

	public TestTask() {
		this.mustSleep = true;
	}

	public void run(){
	  Log.RTdebug("Thread " + this.getName() + " started");


	  sleep();

	  double i = 1e3;
	  while(i-- > 0){

	    double j = 1e6;
	    while(j-- > 0);

	    if(mustSleep)
	      this.waitScheduler.awake();

	    this.sleep();

	  }
	  Log.RTdebug("Thread " + this.getName() + " finished");
	}

}
```

Referencias

[1] A. Vogel, B. Keherve, G. Bochmann y J. Gescei. "Distributed Multimedia and QoS: A Survey". IEEE Multimedia Magazine, 2(2): 10-19. 1995.
[2] The Real.Time Specification for Java, 2003
[3] JamaicaVM 3.2 – User Documentation
[4] M. García Valls. Calidad de servicio en sistemas multimedia empotrados mediante gestión dinámica de recursos. Tesis doctoral. Universidad Politécnica de Madrid. Julio 2001.
[5] Sun Microsystems. JRE user manual 1.3.0