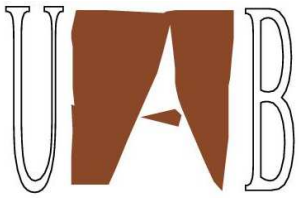

Universidad Autónoma de Barcelona

Escuela Técnica Superior de Ingeniería

Departamento de Arquitectura de Computadores y Sistemas Operativos

Planificación de Aplicaciones Best-Effort y Soft Real-Time en NOWs

Memoria presentada por **José R. García** correspondiente al Trabajo Experimental dentro del Programa de Doctorado en Informática, Opción A: "Arquitectura de Ordenadores y Procesamiento Paralelo".

Barcelona, 10 de julio de 2007

El Dr. Porfidio Hernández Budé TU del Departamento de Arquitectura de Computadores de la Universidad Autónoma de Barcelona,

CERTIFICA:
Que la presente memoria "Planificación de Aplicaciones Best-Effort y Soft Real-Time en NOWs" ha estado realizada bajo su dirección por D. José R. García Gutiérrez, y constituye el Trabajo Experimental dentro del Programa de Doctorado en Informática, Opción A: "Arquitectura de Ordenadores y Procesamiento Paralelo".

Barcelona, 10 de julio de 2007

Dr. Porfidio Hernández

Es la libertad la esencia de la vida.
José Martí

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La ciencia y la industria siempre han necesitado más potencia de cómputo de la que proporcionan los ordenadores. Aunque una tendencia para resolver este problema ha sido mejorar el hardware, no ha sido la única vía a seguir. La computación paralela, con proyectos tan ambiciosos y hermosos como SETI@Home [5], donde usuarios cooperan desde sus hogares en la búsqueda de vida inteligente en otros planetas, se perfila cada vez con más fuerza, como una de las posibles soluciones.

Al mismo tiempo que crece la potencia de cómputo de los ordenadores, y se desarrollan nuevas técnicas de paralelización cambia de forma radical el mundo de las aplicaciones. Nuevos tipos de aplicaciones paralelas con requerimientos de cómputo más estrictos [84, 65, 88] son cada día más comunes en el mundo científico e industrial. Este tipo de aplicaciones pueden requerir un tiempo de retorno (*turnaround*) específico o una calidad de servicio (QoS, *Quality of Service*) determinada, haciendo más complejos los sistemas y modelos de predicción e imponiendo nuevas pautas en el desarrollo de los mismos.

En el caso de los clusters no dedicados, son de especial importancia las aplicaciones locales, cuya evolución se traduce en más requerimientos de recursos de memoria, CPU y ancho de banda de red [26]. Un usuario local puede estar visualizando un vídeo almacenado en su ordenador, lo cual implica necesidades de CPU periódicas y un mayor uso de memoria que los tipos de aplicaciones Best-effort habituales hasta la fecha.

La aparición de nuevos tipos de aplicaciones, como vídeo bajo demanda, realidad virtual, aprendizaje a distancia y videoconferencias entre otras, se caracterizan por la necesidad de cumplir sus *deadlines* y por lo tanto presentan requerimientos periódicos de recursos. Este tipo de aplicaciones, donde la pérdida de deadlines no se considera un fallo severo, aunque ha de ser evitada en lo posible, han sido denominadas en la literatura aplicaciones *soft-real time* (SRT) periódicas.

El paralelismo es una de las grandes apuestas en la gran carrera para mejorar el tiempo de ejecución de las aplicaciones [33]. El crecimiento progresivo en el rendi miento de los procesadores en los últimos años se había visto frenado por las barreras físicas del espacio y la velocidad de las señales. Hoy en día se producen procesadores de dos núcleos incluso para los portátiles. Las previsiones son que el número de núcleos por procesador se incrementará paulatinamente con el tiempo.

Imaginando este escenario que se avecina: nuevas aplicaciones de escritorio y paralelas, así como plataformas hardware cada vez más potentes y complejas, el problema de la planificación temporal de dichas aplicaciones en clusters no dedicados supone un nuevo reto a asumir por los investigadores y conforma el núcleo de este trabajo.

1.1. Clusters no dedicados

Con objeto de satisfacer las necesidades de cómputo masivo existentes en la industria o la ciencia pura, la paralelización se ha convertido en una de las vías más estudiadas y aceptadas. En el fondo, el objetivo básico de esta estrategia consiste en mejorar el tiempo de ejecución de las aplicaciones para poder así afrontar problemas más complejos y aumentar el factor de escala. Cuando paralelizamos, es especialmente atractiva la idea de lograr una gran capacidad de cómputo a coste mínimo, siendo las redes de estaciones de trabajo (*Network of Workstations*, NOW) no dedicadas una de las opciones. En nuestro enfoque, una NOW es un laboratorio docente universitario, un ambiente controlado donde conocemos y podemos estudiar a fondo las necesidades y costumbres de nuestros usuarios locales [9]. En las universidades existen todas las condiciones para este tipo de estudios, por un lado grandes redes de ordenadores pertenecientes a instituciones y por el otro, grupos científicos con necesidades de cómputo importantes.

Queremos destacar que el uso de los períodos de inactividad de los ordenadores pertenecientes a NOWs para ejecutar aplicaciones paralelas no es una utopía, pues en los escenarios antes descritos los ordenadores están ociosos entre el 80 y el 90 porciento del tiempo. Multitud de trabajos se han centrado en este enfoque desde diferentes perspectivas, que representan diferentes formas de utilizar esta capacidad de cómputo. Destacamos los estudios centrados en clusters no dedicados, destacamos que nuestro trabajo se enfoca en la *Planificación de Aplicaciones*:

- *Cluster Computing on The Fly* [56]: Técnica de cómputo oportunista.
- *Cluster-US* [67]: Basado en computadores hibernables, que utiliza los nodos para el para el cómputo paralelo en horarios en que nos se utilizan para otras tareas.

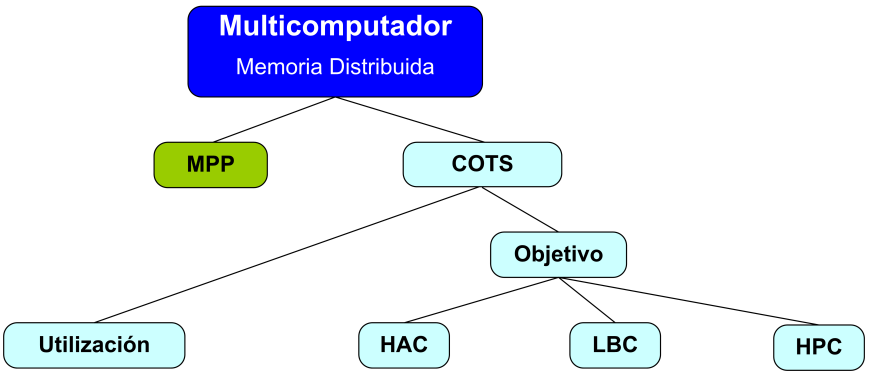

Figura 1.1: Taxonomía de Arquitecturas Paralelas.

- *Grid* [62, 53]: Aprovechamiento de nodos ociosos para cómputo paralelo, en esta variante hay tanto clusters dedicados como no dedicados.
- **Planificación espacial y temporal de aplicaciones**: *Gang Scheduling* [28], Coplanificación dinámica [70], Coplanificación Dinámica [70] o sistemas como SLURM [45], diseñados para clusters Linux.

La Figura 1.1 presenta una taxonomía de Arquitecturas Paralelas. Nuestro campo de interés está centrado en los Multicomputadores, donde las máquinas tipo *Massive Parallel Processors* (MPP) son una opción. Los MPP proporcionan una elevada capacidad de cómputo a un alto coste económico, ya que para su funcionamiento requiere de hardware y software específico. Por este motivo es cada vez más común el uso de clusters COTS (*Commodity Of-The-Shelf*) debido principalmente a la disminución de los costes. Este tipo de entorno se construye a partir de componentes comerciales fácilmente accesibles y el nivel de acoplamiento que alcanzan es bastante menor que los MPP. Es también una opción válida proveer los clusters COTS de redes de conexión más rápidas, aunque hacerlo conlleva un aumento importante en el presupuesto.

Siguiendo la taxonomía propuesta, los clusters COTS son divididos de acuerdo a su **objetivo** o a su **utilización**. En el caso de la división por **objetivos**, ejemplos de cada clase son HA-OSCAR [61] en clusters de alta disponibilidad (HAC, *High Availability Clusters*), MOSIX [58] en la clase de clusters centrados en el balanceo de cargas (LBC, *Load Balancing Clusters*) y Maui [43] como ejemplo de cluster centrado en obtener altas prestaciones (HPC, *High Performance Computing*). En la clasificación por su **utilización** podemos destacar dos clases, dedicados y no dedicados. Los sistemas dedicados suelen conocerse como clusters *Beowulf*, caracterizandose por la no existencia de aplicaciones no controladas por el sistema y la sintonización de todos los componentes a la ejecución de una única aplicación paralela.

En esta útima vertiente de la clasificación (**Utilización**) es donde centramos nuestros estudios (Figura 1.1), en los *clusters no dedicados*. Este tipo de clusters está formados por recursos computacionales preexistentes, abaratándolos aún más con respecto a ordenadores tipo MPP. Hemos de destacar que en los clusters no dedicados podemos encontrar aplicaciones locales no controladas, y es precisamente la aparición de nuevos tipos de aplicaciones locales de tipo SRT una de las principales dificultades, en base al nuevo grado de complejidad que introduce en el sistema. Esto unido a los nuevos requerimientos de las aplicaciones paralelas en cuanto al establecimiento de un turnaround determinado, constituyen el entorno donde se van a desarrollar las propuestas que se presentarán en este trabajo. La *planificación temporal* de aplicaciones Best-effort y SRT, tanto locales como paralelas, en entornos no dedicados de tal forma que no afectemos los niveles de interactividad necesarios para la comodidad de los usuarios locales, configurará el objetivo de las propuestas a desarrollar.

Son de suma importancia para nuestra finalidad estudios como [35, 24], focalizados tanto en el confort de los usuarios locales como en los recursos que emplean en sus aplicaciones. Aún cuando estos estudios se centran en aplicaciones tipo Best-effort, podemos complementarlos con estudios que caracterizan los nuevos tipos de aplicaciones locales SRT [27]. Nuestra finalidad es determinar la viabilidad de la ejecutación de cómputo mediante un usuario paralelo sin que los niveles de interactividad del usuario local se vean afectados.

Nuestro trabajo implica la coexistencia de la carga paralela con la presencia de usuarios locales(y sus aplicaciones) en los ordenadores. Esto condiciona nuestro problema de planificación a dos niveles, **planificación espacial** y **planificación temporal**. Es decir, necesitamos decidir dónde ejecutaremos nuestro cómputo paralelo y como planificaremos los recursos en los nodos compartidos, siendo este último tipo de planificación nuestro objetivo central.

1.2. Planificación en clusters no dedicados

Como ya hemos establecido anteriormente, la planificación de aplicaciones paralelas tiene dos áreas claramente diferenciadas, la *planificación espacial* y la *planificación temporal*. De estas dos variantes, la primera es la que decide en qué conjunto de nodos se va a ejecutar una aplicación paralela y la segunda realiza la planificación temporal a corto plazo en cada nodo perteneciente al cluster. La Figura 1.2 muestra de forma general la problemática que abordaremos en esta sección.

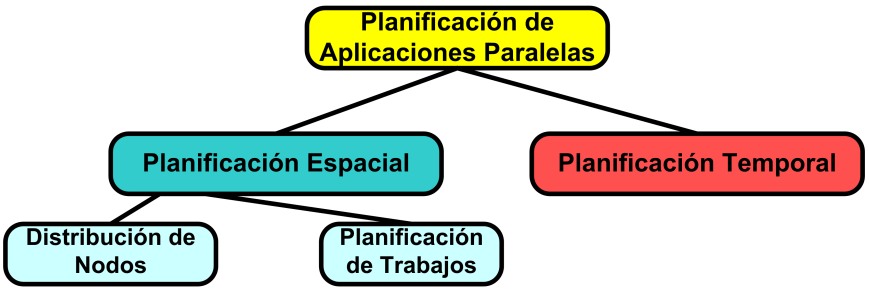

Figura 1.2: Taxonomía General de la Planificación de Aplicaciones Paralelas

1.2.1. Planificación espacial

Las políticas de planificación espacial son las encargadas de decidir en qué nodos se ejecutan los trabajos y cómo se planificarán. En esta subsección abordaremos los temas relacionados con la selección de nodos y su posterior planificación. La selección de nodos podemos abordarla desde dos vertientes principales, la distribución de los nodos y la planificación de los trabajos.

1.2.1.1. Distribución de los nodos

Profundizando en la distribución de los nodos encontramos la necesidad de *particionar* y *seleccionar* los nodos para una correcta distribución, de la forma mostrada en la Figura 1.3. Entre las alternativas de **particionamiento** de nodos encontradas en la literatura, los más simples para su implementación son el estático y el variable, siendo este último una de las mejores opciones por el balance entre la simplicidad a la hora de implementarlo y las desventajas que presenta.

Por otro lado la **selección** intenta elegir los nodos donde ejecutaremos las aplicaciones paralelas de acuerdo a políticas de selección. La política de selección más simple, la *binaria*, considera que un nodo o bien está libre u ocupado. Este tipo de política no considera la posibilidad de que las aplicaciones puedan compartir nodos, ya que no tiene en cuenta el grado de ocupación de los nodos. Como ejemplo de política binaria encontramos las de tipo *buddy* [77, 55].

Las llamadas políticas de *selección discreta* [37, 30, 87] son aquellas dónde se considera un grado de multiprogramación (*Multi Programming Level*, **MPL**) mayor que 1. Lógicamente, al ser el MPL >1, las políticas de este tipo han de combinar el espacio compartido con el tiempo compartido, es decir, trabajar espacial y temporalmente.

En caso de presentar un MPL >1, es necesario de alguna forma poder estimar el grado de disponibilidad de los nodos para cómputo paralelo. Un nodo con

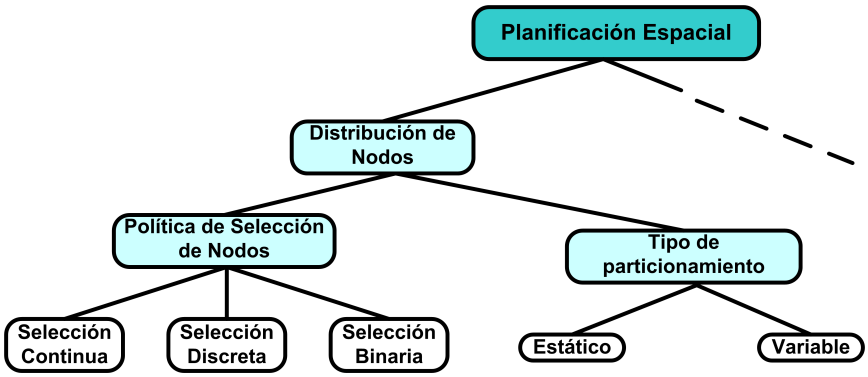

Figura 1.3: Distribución de Nodos en la Planificación Espacial.

cierto grado de ocupación podrá ejecutar una aplicación paralela o no de acuerdo a las necesidades de cómputo y memoria de esta. Dada la posibilidad de ejecutar aplicaciones paralelas en nodos donde se estén ejecutando aplicaciones locales, han de desarrollarse políticas que tengan en cuenta esta situación. Este tipo de políticas se conocen como de *selección continua*.

1.2.1.2. Planificación de Trabajos

La planificación de trabajos se centra en el ordenamiento de los trabajos en espera de ser ejecutados y la forma de seleccionarlos de la *cola de espera*. Primero abordaremos las formas de ordenamiento de las colas de espera y luego la forma de seleccionar los trabajos a ejecutar, la Figura 1.4 muestra las diferentes políticas que mencionaremos.

Con la llegada al sistema de un nuevo trabajo paralelo que no se puede ejecutar en el momento, tenemos un incremento de la cola de espera del cluster. Algunas de las políticas de ordenamiento son: **FCFS** (*First Come First Served*, los trabajos son ejecutados en el orden en que llegan al sistema [86, 76]), **SJF** (*Shortest Job First*, los trabajos se ordenan de forma creciente en función del tiempo de ejecución estimado [4]) y **SNPF** (*Smallest Number of Processes First*, los trabajos se ordenan de acuerdo a la cantidad de procesadores que se solicitan [57]).

Una vez ordenada la cola de espera necesitamos *seleccionar los trabajos* que se encuentran en ella para su ejecución. Aunque elegir el primero parece la opción más justa, no siempre proporciona buenos resultados. Podría ocurrir que un trabajo con muchos requerimientos esté a la cabeza de la cola y frene innecesariamente los demás trabajos en espera.

También ha de considerarse si además del orden existente en la cola de espera tendremos en cuenta el *estado actual* del cluster. El objetivo es lograr un equilibrio entre las métricas de utilización del sistema relacionadas con

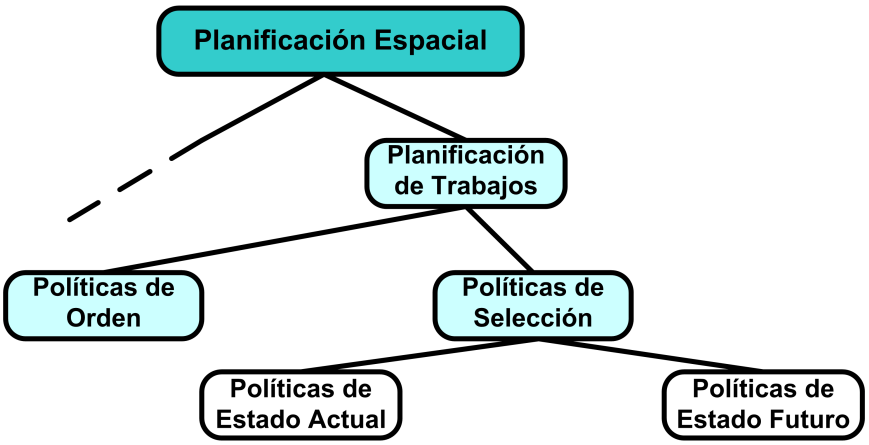

Figura 1.4: Planificación de Trabajos en la Planificación Espacial.

los usuarios y el rendimiento del cluster. Lo usual es intentar utilizar el conocimiento del estado del cluster para predecir el estado futuro y sacar ventajas de este conocimiento.

Entre las políticas más simples, que no necesitan información adicional excepto el conocimiento de la cola de espera podemos citar: *First Fit* y *Best Fit*. La primera de ellas, como su nombre indica, funciona buscando el primer trabajo existente en la cola cuyos requerimientos de recursos sean menores o iguales que la disponibilidad de recursos del cluster [69, 4]. Empleando *Best Fit* el criterio de selección es que el trabajo elegido tenga los requerimientos de recursos más similares a la disponibilidad de recursos del cluster en el momento dado [86].

Entre las políticas que intentan obtener ventaja del *conocimiento del estado* del cluster podemos observar dos grupos fundamentales. Las que se basan en mantener una calidad de servicio (QoS, *Quality of Service*) durante la ejecución de la aplicación [17] y las que intentan adelantar trabajos utilizando técnicas de *Backfilling* [80, 40, 38]. La desventaja natural de este tipo de políticas reside en la necesidad de un **tiempo estimado** de ejecución de las aplicaciones. El problema representado por la imprecisión propia de las estimaciones de los usuarios [60] se ha intentado solucionar de utilizando diversas técnicas, ya sea mediante sistemas históricos [47, 82] o de modelos analíticos [44].

1.2.2. Planificación temporal

La necesidad de compartir los nodos entre los dos tipos de usuarios (local y paralelo) para lograr la planificación el clusters no dedicados, nos obliga a disponer de métodos para hacerlo de forma equitativa. Hemos de contemplar

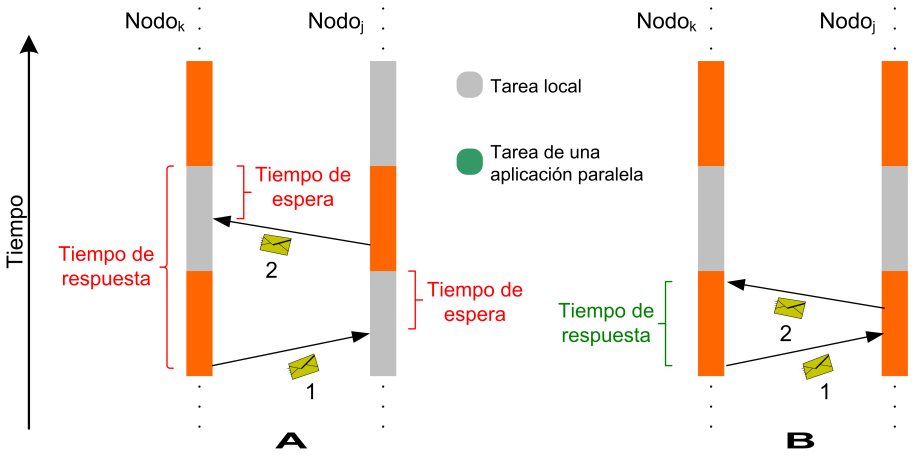

Figura 1.5: Efecto producido por la comunicación sobre la ejecución.

tanto las necesidades de los usuarios locales que brindan sus ordenadores, y encontrar formas de favorecer el progreso de las aplicaciones paralelas sin afectarlos. Debido a que los nodos en un cluster no dedicado son compartidos, ha de mantenerse la interactividad necesaria para el usuario local.

La forma de estimular el progreso de las aplicaciones paralelas se basa en su propia naturaleza. Podemos decir que las aplicaciones paralelas son procesos cooperantes, lo cual implica el intercambio de mensajes en mayor o menor medida. La Figura 1.5 muestra dos posibles casos de comunicación entre dos procesos cooperantes. En el caso A vemos claramente como el hecho de que dichos procesos cooperantes no dispongan de la CPU de forma simultánea provoca dos tiempos de espera adicionales. En cambio en el caso B los dos procesos cooperantes se planifican al mismo tiempo lo cual aumenta las probabilidades de disminuir los tiempos de espera provocados por la comunicación.

1.2.2.1. Coplanificación tradicional

La coplanificación, nombre recibido por las técnicas que intentan estimular la planificación simultánea de procesos cooperantes, fue introducida por [63] y ha sido abordada en múltiples estudios [34, 72, 11, 71, 28]. En su estudio embrionario, John Ousterhout propone un algoritmo conocido como **Algoritmo de la Matriz de Ousterhout** basado en la analogía existente entre la gestión de memoria en un entorno monoprocesador multiprogramado y la gestión de los procesadores en un entorno multiprocesador y multiprogramado. En un entorno monoprocesador y multiprogramado obtenemos una clara ventaja al tener todas las páginas del *working set* de una aplicación en memoria a la vez cuando se ejecuta. El estudio de Ousterhout mostró que

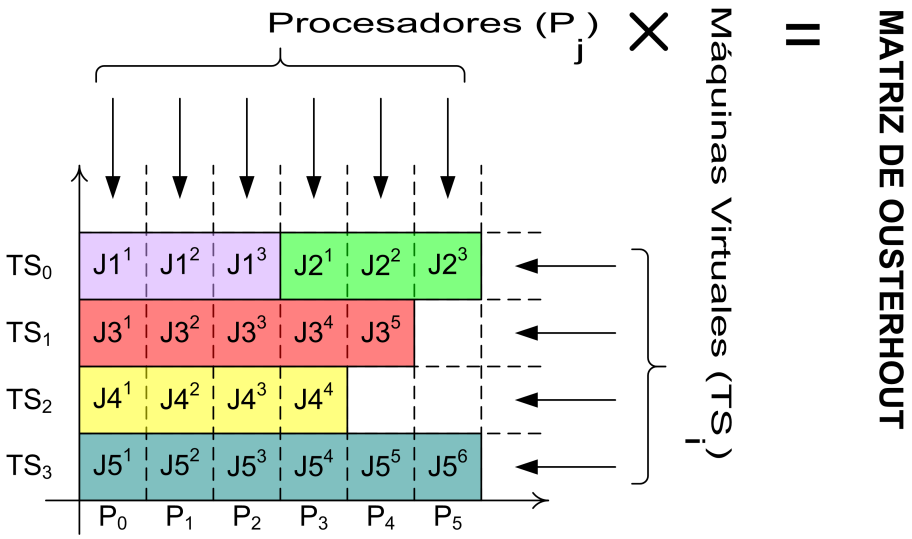

Figura 1.6: Matriz de Ousterhout

el rendimiento de las aplicaciones paralelas se ve seriamente afectado si las mismas no reciben suficientes procesadores y las tareas pertenecienteas a las mismas no son planificadas a la vez. Esto se debe a que los requerimientos de comunicación y sincronización existentes entre los procesos cooperantes de una aplicación paralela, pueden afectar ralentizando su ejecución debido a las esperas provocadas por la no planificación simultánea de sus procesos cooperantes.

La Figura 1.6 muestra un ejemplo de aplicación del Algoritmo de la Matriz de Ousterhout. La matriz está formada en el eje de las ordenadas por los procesadores ($j = 1 - n$) y en el eje de las abcisas por el número de máquinas virtuales en ejecución ($i = 1 - k$). Las máquinas virtuales tienen una potencia de cálculo igual a Pot_j/n donde Pot_j es la potencia de cálculo del procesador j y n es el grado de MPL de la máquina virtual. Cada columna contiene los procesos asignados a cada procesador Jp^k y cada fila los trabajos que serán ejecutados durante el quantum (TS_k) del procesador. Siguiendo el algoritmo, cada vez que se ha de asignar un nuevo trabajo (Ji) al sistema, se busca una fila con la misma cantidad de celdas libres que procesos tiene el trabajo a ser asignado. Una vez asignado el trabajo, se utiliza una política de *round-robin* para planificar las diferentes filas de la matriz. En este ejemplo, para un $n = 6$ un $MPL = 4$ y las condiciones mostradas, durante el quantum de tiempo TS_0 se ejecutarán los procesos pertenecientes a los trabajos $J1$ y $J2$, de manera que al finalizar dicho quantum se producirá un cambio de contexto global, de modo que el trabajo $J3$ será planificado en los procesadores $P_0, P_1, P_2, P_3 \text{y} P_4$ durante el siguiente quantum TS_1, y así sucesivamente.

El término *Gang Scheduling* también ha sido empleado ampliamente en la literatura para referirse a la necesidad de la coplanificación. En [29] se define este término como un esquema de planificación que organiza sus tareas en grupos, de tal forma que las aplicaciones paralelas los conformen y que los grupos sean planificados simultáneamente en los procesadores de nodos diferentes. Esta definición concuerda completamente con lo ya establecido anteriormente por Ousterhout y utilizaremos este término para referirnos a ambas técnicas.

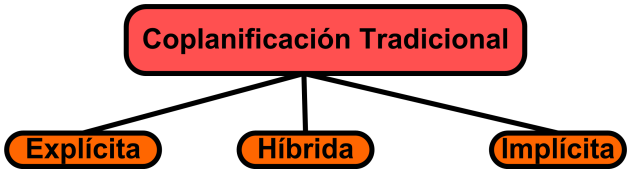

Figura 1.7: Clasificación de la Coplanificación en relación al método de control.

Una relajación del concepto del Gang Scheduling fue propuesta en [71], en el cual se establece que sólo es necesario coplanificar los procesos que están cooperando en un instante determinado.

Una posible forma de clasificar el extenso trabajo llevado a cabo en esta área es de acuerdo al método de control empleado (Figura 1.7) para lograr la coplanificación de los procesos cooperantes:

- Coplanificación con control **explícito**: esta implementación de la coplanificación requiere de un cambio de contexto global simultáneo a lo largo de toda la máquina paralela [31]. Este enfoque es más apropiado para ambientes dedicados y se ajusta a la definición de Gang scheduling.
- Coplanificación con control **implícito**: Las decisiones de planificación son tomadas por los planificador locales de acuerdo con la aparición de eventos locales o remotos. Los eventos pueden ser de comunicación, de memoria, de CPU, de actividad de usuarios locales o grado de multiprogramación (MPL). Como alternativas podemos citar la *coplanificación predictiva* [74, 71] basada en aumentar la probabilidad de coplanificación cambiando las prioridades de los trabajos en función de los eventos de comunicación recibidos y la *coplanificación dinámica* [11, 73], que planifica un proceso si recibe un evento de comunicación, expropiando la CPU al proceso en ejecución.
- Coplanificación con control **híbrido**: como su nombre indica, se hace uso de una combinación de las dos técnicas antes expuestas. Algunos de los resultados son: Buffered Coscheduling (BC) [64], Flexible Coscheduling (FCS) [30]y CoScheduling Cooperativo (CSC) [34].

Suele considerarse que las técnicas basadas en control híbrido son las que aportan más flexibilidad y facilidades de implementación.

1.3. Tiempo Real estricto y débil

Nuestro trabajo se centra en nuevos tipos de aplicaciones con características de tiempo real débil (*soft real-time*, SRT), motivo por el cual en esta sección introduciremos algunos modelos SRT y de tiempo real estricto (*real-time*, RT). Es válido destacar que los sistemas SRT usualmente son considerados como una derivación o relajación de RT, como efectivamente ocurre. Debido a que nos centraremos en los sistemas SRT, en este trabajo colocaremos ambas teorías al mismo nivel en nuestro texto. La intención es profundizar en sistemas RT sólo lo necesario, dado el volumen de información existente y la orientación de nuestro trabajo.

TIEMPO REAL		
Asignación de Prioridades	Fija	Dinámica
Algoritmo Representativo	RMS	EDF
Admisión de Peticiones	$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$	$\sum_{i=1}^{n} U_i \leq 1$
Tareas aperiódicas	Polling Server, Slack Stealing Algorithm	Total Bandwidth Server

Tabla 1.1: Resumen de los temas a tratar en la sección sobre sistemas tiempo real estricto.

1.3.1. Sistemas de Tiempo Real Estricto

Un sistema con requerimientos de tiempo explícitos, ya sea de naturaleza probabilística o determinística, es considerado de RT. La noción de prioridad es comúnmente utilizada para establecer orden en el acceso a recursos, tanto en la CPU como en la Red. La planificación de tareas RT será dividida en dos grupos de acuerdo a la forma en que tratan la prioridad, ya sea con prioridad **fija** y o con prioridad **dinámica**. La Tabla 1.1 muestra un resumen de las características y algoritmos que trataremos en esta subsección.

1.3.1.1. Planificación con Prioridad Fija

En el modelo de Planificación con Prioridad Fija todas las tareas de un mismo trabajo tienen la misma prioridad, que no cambia en el tiempo. La

nomenclatura usualmente empleada denomina a cada tarea como τ_i, dónde i es la prioridad de la tarea. Una tarea es *periódica* (Figura 1.8) si ocurre cada cierto intervalo regular de tiempo, siendo la longitud entre los sucesivos arribos de los trabajos que componen la tarea τ_i constante, llamado el *período* de la tarea y denominado T_i. Cabe destacar que la prioridad i se calcula como la inversa del período ($i = \frac{1}{T_i}$). El *deadline* (plazo) de una tarea periódica se define como D_i, representando este valor el maximo valor de tiempo que puede transcurrir antes de que el trabajo i de la tarea τ_i consuma su *tiempo de cómputo* (C_i).

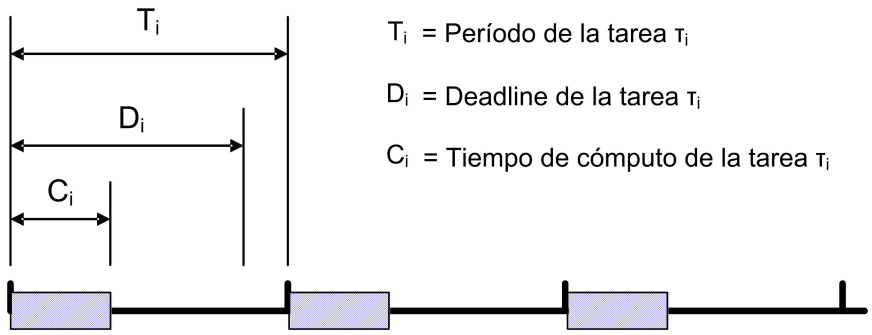

Figura 1.8: Esquema de una Tarea Periódica.

El modelo inicialmente propuesto en la teoría RT (conocido como Modelo de Liu y Layland [54]) asume que:

1. Todas las tareas son periódicas
2. Todas las tareas llegan al inicio de su período y tienen un deadline igual a su período.
3. Todas las tareas son independientes, es decir, no tienen relaciones de precedencia en relación a los recursos que utilizan.
4. Todas las tareas tienen un tiempo de cómputo fijo, o al menos limitado, que es menor o igual que su período.
5. Ninguna tareas se puede suspender a si misma.
6. Todas las tareas son completamente desalojables.
7. No se consideran *overheads*, relacionados con el sistema operativo.
8. Solo existe un procesador.

Bajo este modelo, las tareas de los trabajos periódicos ocurren a lo largo del tiempo a intervalos regulares de longitud constante T_i (el período de la

tarea). Cada tarea tiene un deadline D_i unidades de tiempo después de su liberación. Llamamos *rígida* a una tarea de tiempo real (*hard real-time*) si debe cumplir su plazo (tanto a nivel de tiempo de comienzo como de final); de no cumplirlos se producirán daños no deseados o un error fatal en el sistema.

Una tarea RT es llamada *flexible* si tiene un plazo asociado, que es conveniente, pero no obligatorio; aunque haya vencido el plazo, aún tiene sentido planificar y completar la tarea.

Una tarea *aperiódica* debe comenzar o terminar en un plazo o bien, puede tener tanto una restricción para el comienzo como para la finalización.

Los *análisis de admisión* son empleados para predecir si las restricciones temporales de una tarea serán satisfechos en tiempo de ejecución. Los que tienen en cuenta todos los elementos necesarios (test suficiente y necesario) alcanzan complejidad NP completa, por lo que son impracticables. Son generalmente de menor conplejidad algoritmica los tests que son suficientes pero no necesarios. Los tests suficientes pero no necesarios tienen la desventaja de que son pesimistas.

El hecho de que las prioridades no varien hace más efectivos los análisis de admisión, siendo el Teorema del Instante Crítico [54] el empleado en este caso. El instante crítico para una tarea es el tiempo de liberación para el cual el tiempo de respuesta es el máximo (o excede su deadline, para el caso en el cual el sistema está tan sobrecargado que los tiempos de respuesta crecen sin límites). Este teorema establece que, para un conjunto de tareas periódicas con prioridades fijas, el instante crítico de una tarea ocurre cuando es invocada simultáneamente con todas las tareas de mayor prioridad que ella. El intervalo de 0 a D_i es entonces uno en el cual la demanda de tareas de mayor prioridad $\tau_1 ... \tau_{i-1}$ está en un máximo, creando la situación más difícil para que τ_i cumpla su deadline. Este teorema ha probado ser robusto, siendo verdadero incluso cuando muchas de las restricciones antes listadas son relajadas.

El grupo de políticas de asignación de trabajos con prioridad fija es conocido como RMS (*Rate-Monotonic Scheduling*), en el cual a la tarea con el menor perído se le asigna la mayor prioridad, a la próxima tarea de menor período la siguiente prioridad y así sucesivamentes. Se ha probado que para un conjunto de n tareas periódicas con política de asignación RMS la asignación es posible si:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(2^{\frac{1}{n}} - 1\right) \tag{1.1}$$

Como ejemplo podemos decir que un par de tareas es viable si su utilización de CPU combinada no excede el 82,84 %. Si n tiende al infinito, el valor

$n\left(2^{\frac{1}{n}}-1\right)$ se aproxima a ln(2) para un valor aproximado de utilización de 69,31 %. Muchas veces se asume que bajo estas condiciones el valor antes mencionado defina la máxima utilización posible, lo cual es errado, ya que esto es solo una condición de suficiencia. Este límite es cerrado en el sentido de que existe algún conjunto de tareas inviables cuya utilización arbitrariamente se acerca a $n\left(2^{\frac{1}{n}}-1\right)$. Por este motivo es posible encontrar multitud de conjuntos de tareas con utilización mayor que el 69,31 %. En [49] encontramos un interesante estudio sobre RMS y los niveles de utilización de esta técnica, que como promedio es del 88 %.

No obstante a su utilidad, los análisis de admisión tienen limitaciones, como lo son:

1. La condición de admisión es necesaria pero no suficiente (es decir, pesimista).
2. Impone restricciones poco reales a las características de la tareas, es decir $D_i = T_i$.
3. Las prioridades de la tareas han de ser asignadas utilizando RMS, caso contrario el el análisis de admisión es insuficiente.

Por estas razones se han desarrollado pruebas de admisión más complejas, pero que no tienen las limitaciones antes expuestas. En [15] se propone una prueba de admisión de complejidad polinomial menos pesimista que el representado por la Fórmula 1.1. Esta prueba (Ecuación 1.2) ha demostrado ser fuerte.

$$\prod_{i=1}^{n}\left(\frac{C_i}{T_i}+1\right) \leq 2 \tag{1.2}$$

Existen también tareas, denominadas ***aperiódicas*** que no cumplen con los requerimientos del modelo anterior y han de ser contempladas también en los modelos de RT. Este tipo de tareas puede ser diferente de las periódicas en que los tiempos de arribo o de cómputo sean significativamente diferentes, que no tengan deadlines estrictos o bien alguna combinación de las características antes expuestas.

Si el modelo de tareas periódicas es ligeramente relajado, siendo C_i el máximo tiempo de ejecución y T_i el tiempo mínimo entre los arribos, el modelo [54] sigue siendo válido. Sin embargo es poco práctico y eficiente hacer reservas si los tiempos de cómputo o de arribo de las tareas aperiódicas son muy variables.

Una de la soluciones a este problema es asignar prioridades bajas a las tareas aperiódicas, es decir, relegarlas a procesamiento en *background*. Con

esta acción, intentamos aprovechar el tiempo de cómputo dejado por las tareas periódicas para ejecutarlas. Esta aproximación es válida siempre y cuando, las tareas aperiódicas relegadas a procesamiento en background no tengan requerimientos de QoS, o de obtener un tiempo de respuesta promedio significativo.

Una solución posible al problema de las tareas aperiódicas es implementar un servidor que se ejecute como una tarea periódica normal y que se encargue de ejecutar las tareas aperiódicas. Esta técnica en la literatura se denomina *Polling Server* [68]. La capacidad del servidor se calcula off-line y en la mayoría de los casos se asigna el mayor tiempo de cómputo, que permita el análisis de admisión. En tiempo de ejecución, el servidor se ejecuta periódicamente y su tiempo de cómputo se emplea en ejecutar las tareas aperiódicas. Una vez consumido su tiempo de cómputo su ejecución se suspende hasta su próximo arribo programado, también periódico. Como el servidor se comporta como una tarea periódica, los análisis de admisión diseñados para ellas se pueden aplicar normalmente.

Para las tareas aperiódicas los polling servers significan una mejora sustancial respecto al procesamiento en background. Lógicamente, si llegan demasiadas tareas aperiódicas la capacidad del servidor se verá sobrecargada y algunas tareas tendrán tiempos de respuesta peores. El caso inverso también ocurre, si no llegan tareas aperiódicas la capacidad de cómputo reservada al servidor se infrautiliza. Una posible solución a este último problema es variar la prioridad del servidor de acuerdo a si tiene o no tareas pendientes [79].

Partiendo de la idea anterior, servidores que se ejecutan como tareas periódicas, se han realizado varios trabajos germinales para mejorar el procesamiento de tareas aperiódicas en entornos RT. Entre los más interesantes están el algoritmo de *Slack Stealing* [81], que es óptimo en el sentido de que minimiza el tiempo de respuesta para las tareas aperiódicas manteniendo los deadlines de todas las tareas RT.

Finalmente mostraremos los resultados encontrados en la literatura para **RT** en **multiprocesadores**. Vemos reflejadas dos aproximaciones a la planificación de tareas RT en múltiples procesadores, particionada y global. En la aproximación por particiones cada tarea es asignanada estáticamente a un procesador y en la global las tareas compiten por el uso de los procesadores. En [22] se muestra que la planificación global de tareas para m-procesadores utilizando RMS de un sistema de $m + 1$ tareas no puede ser garantizado para utilizaciones del sistema por encima de 1. Por otro lado, utilizando particionado con RMS *next-fit* podemos garantizar la viabilidad de los sistemas de tareas para utilizaciones por encima de $m/(1+2^{1/3})$. Este límite es conocido como el efecto Dhall, en referencia al investigador que lo determinó.

Dado que el problema de particionamiento óptimo de tareas entre múltiple procesadores es de tipo NP completo, las soluciones óptimas es posible solo

para los casos más simples. Por lo tanto, han de usarse heurísticas para encontrar soluciones aproximadas, siendo la más empleada RMFF (*Rate Monotonic First-Fit*). Para este caso, se ha determinado que la máxima utilización del sistema es de $(m+1)\left(2^{1/(m+1)}-1\right)$. Otro resultado interesante es que para una planificación con prioridades fijas en un sistema multiprocesador de m nodos, sin importar si es global o particionado o el esquema de asignación de prioridades, la utilización garantizada no puede ser mayor que $(m+1)/2$ [8].

Es válido destacar que los avances en la teoría RT aplicados a multiprocesadores o ambientes distribuidos no van a la par los logrados para monoprocesadores y que muchos de los resultados alcanzados para monoprocesador aún necesitan ser generalizados a multiprocesadores, en caso de ser posible.

1.3.1.2. Planificación con Prioridad Dinámica

Planificando con prioridades estáticas, todas las tareas pertenecientes a un mismo trabajo tienen la misma prioridad, si empleamos prioridades dinámicas no será así. Otorgando las prioridades de forma dinámica, cada trabajo perteneciente a una tarea tiene diferentes prioridades, en función de cuán cerca esté su deadline. Uno de los algoritmos con prioridad dinámica más estudiados es el EDF (*Earliest Deadline First*).

EDF es un algoritmo dinámico que no requiere que los procesos (tareas) sean periódicos, lo cual constituye un requerimiento del algoritmo RMS. Tampoco es necesario que sea uniforme el tiempo de ejecución por ráfaga de CPU (como si ocurre con RMS). Cada vez que un proceso necesita la CPU, anuncia su presencia y su plazo. El planificador mantiene una lista de los procesos ejecutables en orden por plazo. El algoritmo ejecuta el primer proceso de la lista, el que tiene el plazo más cercano. Cada vez que un nuevo proceso está listo, el sistema verifica si su plazo se va a cumplir antes que se cumpla el del proceso que se está ejecutando. En tal caso, el nuevo proceso expropiará al actual. Para este algorimo, suponiendo las condiciones del Modelo de Liu y Layland, la prueba de admisión para un conjunto de n tareas periódicas se establece por la utilización del procesador 1.3.

$$\sum_{i=1}^{n} U_i \leq 1 \tag{1.3}$$

En esta ecuación, el *nivel de utilización*, denotado como U_i se define como $U_i = \frac{C_i}{T_i}$.

Aunque existe otro algoritmo que emplea prioridad dinámica, denominado LLF (*Least Laxity First*)[59], este introduce un mayor overhead al sistema, razón por la cual la mayor parte de la investigación se centra en mejorar el

algoritmo EDF. Las mejoras se centran en mejorar los análisis de admisión y en relajar algunos de los postulados simplistas del algoritmo. En [20] se muestra que el algoritmo EDF es óptimo en el sentido de que si existe un algoritmo que puede construir una planificación viable en un solo procesador, entonces el algoritmo EDF también puede construir una planificación viable.

Cuando se utiliza EDF el análisis de admisión puede ser realizado teniendo en cuenta el criterio de *Demanda de Procesador*. La demanda se calcula para un conjunto de trabajos RT y un intervalo de tiempo $[t_1, t_2)$ como

$$h_{[t_1,t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k. \tag{1.4}$$

Es decir, la demanda de procesador es el valor representado por la cantidad de tiempo de cómputo pedido por todos los trabajos con tiempo de arribo en o después de t_1 y deadline antes o en t_2. A partir de este valor, el análisis de admisión puede ser efectuado considerando que la demanda de procesador no puede superar el tiempo disponible, es decir, podemos establecer la viabilidad teniendo en cuenta 1.5.

$$\forall t_1, t_2 \quad h(t_1, t_2) \leq (t_2 - t_1) \tag{1.5}$$

Como las prioridades son dinámicas, la planificación de tareas aperiódicas mejora ya que puede reaccionar mejor a la llegada de una tarea no periódica. Uno de los principales enfoques es el del *Total Bandwidth Server* (TBS) [75], que es una de las técnicas más eficientes para planificar tareas aperiódicas bajo EDF. TBS funciona asignando a cada trabajo aperiódico un deadline de tal forma que la carga total aperiódica no exceda un valor máximo U_s. El deadline asignado se calcula mediante la expresión asociada 1.6, nótese que esta toma en cuenta las asignaciones a tareas anteriores (representadas por d_{k-1}). Una vez asignado el deadline, el requerimiento es insertado en el sistema como el de cualquier otra tarea periódica, pero respetando el umbral U_s antes establecido. Podemos afirmar que dado un conjunto de n tareas periódicas con una utilización del procesador de U_p y un TBS con utilización U_s todo el conjunto es viable para su planificación si y solo si $U_p + U_s \leq 1$. Es válido mencionar que el proceso de asignación de deadlines puede ser optimizado para minimizar el tiempo de respuesta a las aplicaciones aperiódicas [16]. En esta aproximación, el *ancho de banda* del servidor define como un umbral (U_s), que representa la capacidad de cómpto disponible respetando las tareas periódicas.

$$d_k = max\,(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{1.6}$$

La Figura 1.9 muestra un ejemplo de uso de un TBS. Dos tareas periódicas con períodos $T_1 = 6$, $T_2 = 8$ y tiempos de cómputo $C_1 = 3$, $C_2 = 2$ respectivamente se planifican bajo EDF para una utilización $U_p = 0,75$ implicando que el ancho de banda del servidor disponible es de 0,25 (calculado mediante $U_s = 1 - U_p$). Los deadlines son calculados utilizando la Ecuación 1.6 en el momento de los arribos de las tareas aperiódicas. El primer arribo de tarea aperiódica ocurre en $t = 3$ y se le asigna un deadline $d_1 = 7$, como $d_1$es el deadline más cercano a expirar globalmente es servido inmediatamente. La próxima tarea aperiódica arriba en $t = 9$ y recibe un deadline $d_2 = 17$, pero no es servida de forma inmediata, ya que en ese momento está en ejecución una tarea con deadline más urgente (τ_2, con deadline en $t = 16$). Por último llega una tarea aperiódica en $t = 14$ que recibe un deadline $d_3 = 21$, que no es servida de forma inmediata ya que en el momento de su llegada la tarea periódica τ_1 está activa y tiene un deadline más bajo.

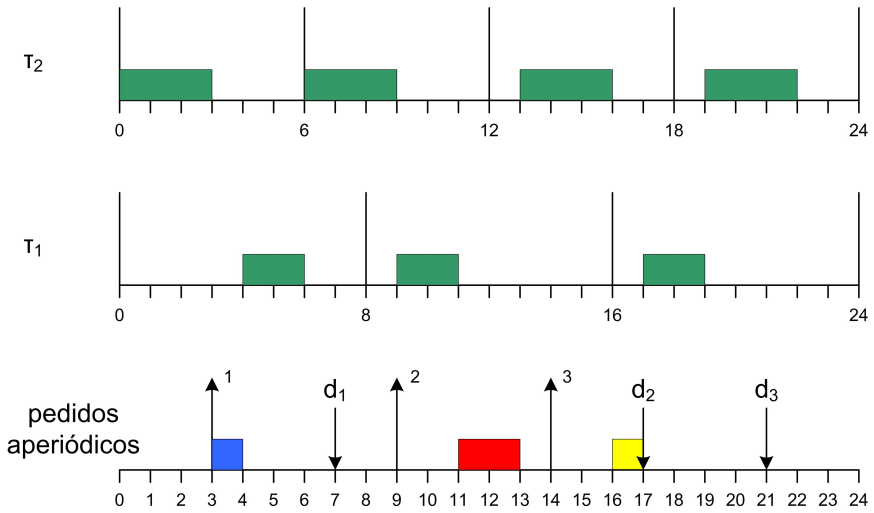

Figura 1.9: Ejemplo de planificación bajo EDF empleando *Total Bandwidth Server*.

Para la **implementación de reservas** bajo EDF disponemos como una opción válida del algoritmo *Constant Bandwidth Server* (CBS) [3]. Un CBS se caracteriza por un presupuesto c_s, un deadline dinámico d_s y un par ordenado (Q_s, T_s), donde Q_s es el presupuesto máximo y T_s el período del servidor. Llamamos a el cociente $U_s = Q_s/T_s$ el ancho de banda del servidor. A cada trabajo servido por el CBS se le asigna un deadline conveniente e igual al deadline actual del servidor, calculado para no sobrepasar el ancho de banda reservado. Mientras el trabajo se ejecuta, el presupuesto c_s es decrementado en el tiempo consumido por el trabajo. Cada vez que $c_s = 0$ se recarga el presupuesto del servidor a Q_s y el deadline del servidor se pospone en T_s, para reducir la interferencia a otras tareas.

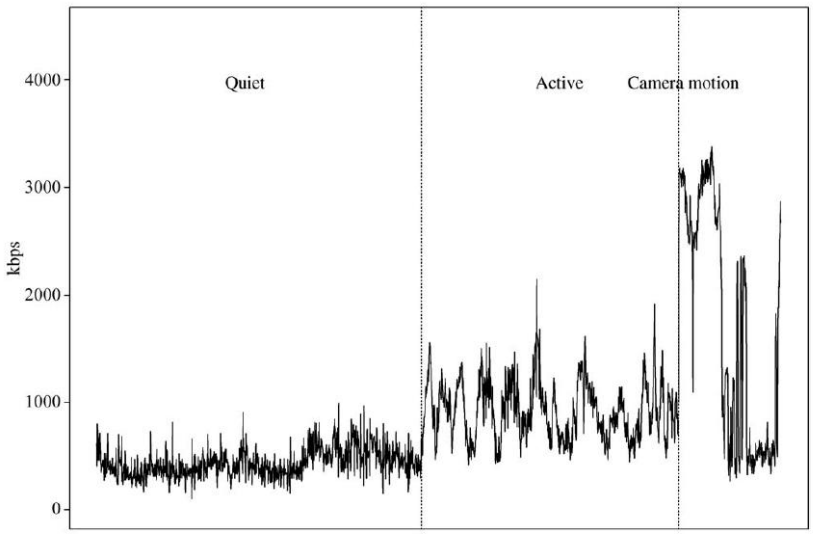

Figura 1.10: Caracterización del uso de ancho de banda de red durante una vídeo conferencia.

1.3.2. Sistemas Tiempo Real Débil

La teoría para RT está concebida tomando como axioma que la pérdida de un deadline ha de considerarse un fallo en el sistema. Para lograr respetar de forma estricta los deadlines de la tareas, la teoría RT se basa en la formulación estricta del peor caso. Este enfoque permite tener una cota superior para la carga en cualquier instante de tiempo, lo que nos permite conocer si los deadlines de las tareas se cumplirán.

Pero qué ocurre si el peor caso no está cerca del caso promedio, como sucede en la mayoría de los sistemas de control para lo que está diseñada la teoría RT estricta. Un ejemplo claro y ampliamente utilizado en la literatura para ejemplificar esta situación es el de un vídeo online. Si no hay cambios bruscos en las escenas, los *frames* transmitidos son de menor tamaño, debido a la codificación de la información en frames I, P o B. Esto tiene como consecuencia que la media de uso del ancho de banda de red sea bastante menor, diferencia que puede llegar a ser de varios órdenes de magnitud. Si tratamos este caso de forma estricta, deberíamos de reservar recursos que una parte importante del tiempo estarían ociosos.

La Figura 1.10 muestra un caso parecido, el uso de ancho de banda de red durante una video conferencia. Claramente podemos observar que los valores oscilan entre una media de 500-600 kbps cuando la cámara de video y los participantes de la video conferencia están quietos hasta 3500-4000 kbps si se mueve la cámara de video. La variaciones se deben al algoritmo

utilizado, basado en enviar sólo las diferencias con la escena anterior, lo que motiva que la cantidad de información a transmitir en cada caso se diferente. Resulta evidente que para esta situación reservar el peor caso es un desperdicio importante de recursos, esta sección intenta profundizar en la teoría relacionada con el tratamiento de estos casos, la teoría de tiempo real débil (*Soft Real-Time*, SRT).

1.3.2.1. Aplicaciones SRT

El caso anteriormente descrito presenta una situación que cada vez es más común, tanto para usuarios locales como para los usuarios paralelos. En esta sección mostraremos varios ejemplos de aplicaciones, tanto locales como paralelas, que requieren especial atención dados sus requerimientos de recursos periódicos.

En [27] se estudian varios métodos para identificar aplicaciones de tipo *Human Centered* (HuC), i.e.: reciben el foco de atención del usuario local, razón por la cual, según se plantea en ese trabajo, deberían de recibir especial atención. La caracterización de las aplicaciones estudiadas en este trabajo nos permite conocer mejor las diferencias en los requerimientos de recursos de las aplicaciones a lo largo de un lapso de tiempo significativo para nuestros objetivos. El cambio más significativo se refleja en el paso de aplicaciones con interactividad, basada en tiempos de respuesta del teclado o el ratón, como a los editores de texto de diferentes tipos y a las aplicaciones multimedia. Estas últimas necesitan más recursos de forma periódica para su correcta ejecución, y podría ocurrir que durante largos períodos de tiempo no reciban ningún evento originado por el usuario, como un clic de ratón u otros. Aplicaciones de estas características (mayores requerimientos periódicos de recursos) componen el grupo de aplicaciones que denominaremos aplicaciones locales SRT (*local_ SRT*).

Otros componentes de este grupo son los tipos de juegos con algoritmos de visualización complejos, como los conocidos por *First Person Shooter* (FPS). Es una tendencia que los juegos de ordenador consuman cada vez más recursos y en caso de no recibirlos de forma periódica, su ejecución no sea satisfactoria para el usuario. Este tipo de aplicación se emplea en los estudios como aplicación comparativa.

Por otro lado encontramos que es cada vez más común que los usuarios paralelos necesiten ejecutar aplicaciones con necesidades temporales. Este es el caso descrito en [83], donde rutinas de detección de obstáculos en secuencias de *frames* hacen que el volumen de cálculo sea alto y de acuerdo a la finalidad del resultado, ha de obtenerse con urgencia. Este tipo de aplicaciones requiere de hardware especializado o de cómputo paralelo de altas prestaciones. También en [65] encontramos un caso novedoso de aplicación paralela.

En este estudio encontramos un tipo de aplicaciones con una alta cantidad de eventos, generados por instrumentación científica, y una ausencia casi total de usuarios. Para lograr recolectar todos los eventos necesitamos que las tareas que se generan con cada evento gocen de prioridades en el sistema.

Otro ejemplo de aplicación paralela SRT, bastante más común que los anteriormente mencionados, es la posibilidad de que el usuario paralelo necesite los resultados de la ejecución de su aplicación paralela dentro de un intervalo de tiempo específico. Las aplicaciones que requieran de tiempos de cómputo periódico en diferentes nodos de un sistema distribuido, ya sea dedicado o no, recibirán en este estudio el tratamiento de aplicaciones paralelas SRT (*par_SRT)*.

1.3.2.2. Modelos SRT

¿Cómo representamos estas tareas de tipo SRT, ya sea locales o paralelas, en forma de modelos? Han existido muchos enfoques basados en la teoría RT existente, como lo es mezclar tareas periódicas con aperiódicas sin deadlines o complejos modelos que le asignan a cada tarea un *valor de utilidad* en función de la QoS requerida por la tarea. El disponer de modelos para este caso nos permite predecir, calcular e incluso garantizar algún recurso a este tipo de tareas. Describiremos algunos de los modelos (Figura 1.11) encontrados en la literatura.

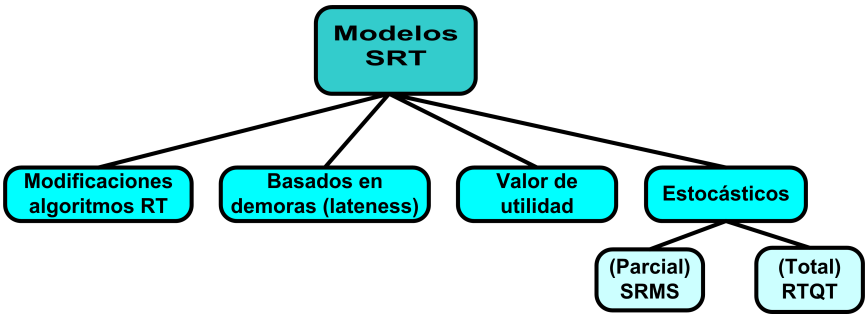

Figura 1.11: Taxonomía: Modelos SRT.

El modelo basado en demoras (*lateness*) se formula asociando a los deadlines de las tareas SRT una restricción que representa la demora permisible. Esta restricción puede tener varias formas e incluso podemos encontrarla en forma compuesta. Por ejemplo, si definimos $\alpha(x)$ como la parte de los trabajos que pierden su deadline por más de x unidades de tiempo, entonces se suele definir la demora en la forma $\alpha(x) \leq \beta$. Esta notación representa una restricción en el sentido de que limitamos la cantidad de trabajos que pierden su deadline a β. En este modelo, cada deadline perdido se considera un fallo y β limita la fracción de trabajos que pueden fallar. El valor $\alpha(-\infty)$ representa la cantidad

de fallos, incluidos los rechazados en los análisis de admisión. En general, para un conjunto de valores de tiempo denotado por $\{x_1, \ldots, x_m\}$ y una lista de restricciones $\{\beta_1, \ldots, \beta_m\}$ podemos requerir que $\alpha(x_i) \leq \beta_i, 1 \leq i \leq m$. Esta especificación de las restricciones nos permite tener en cuenta la naturaleza estocástica de los tiempos de arribo y de cómputo de las tareas tipo SRT, lo cual a su ves nos lleva a la formulación del concepto de *viabilidad del caso promedio*, es decir, el mayor monto de carga promedio que el sistema puede procesar cumpliendo las restricciones de demora.

También basados en la naturaleza estocástica de las aplicaciones SRT se han formulado otros modelos, como el Modelo Estocástico RMS, *Stochastic RMS* (SRMS) [12, 13]. Este modelo está especialmente designado para ser utilizado en sistemas en los cuales las tareas periódicas tienen tiempos de cómputo y requerimientos de QoS altamente variables, además de que los deadlines de las tareas sean débiles, no duros. Este último requerimiento significa que algunos deadlines pueden perderse, aunque con restricciones en las pérdidas. El diseño del algoritmo SRMS también se pensó de tal forma que maximice el uso de los recursos a la vez que minimice el uso de recursos de las tareas que pierden sus deadlines.

Por último mencionaremos los modelos que consideran que todos los parámetros de las tareas SRT son estocásticos, es decir, que tienen tiempos de arribo entre las tareas, tiempos de cómputo y deadlines estocásticos. Aunque en este caso es difícil derivar un análisis completo de la viabilidad para el caso promedio, se han desarrollado modelos para calcular la fracción de tareas demoradas (*late tasks*) para los casos con tráfico pesado, en los cuales los modelos tienen altos niveles promedio de utilización. Este método es conocido como RTQT (*Real-Time Queueing Theory*) [25, 50, 51], Teoría de Colas para Tiempo Real debido a que es una extensión de la Teoría de Colas que tiene en cuenta de forma explícita los requerimientos temporales de las tareas SRT. RTQT asume que las tareas se planifican bajo EDF y su métrica de rendimiento se calcula en base a la fracción de las tareas que terminan dentro de su deadline.

Este algoritmo ha de mantener información del tiempo restante hasta que el deadline de cada tarea finalice (*lead time*). Este requerimiento combinado con la necesidad del algoritmo EDF de mantener información de los deadlines de cada tarea, hace que este problema sea analíticamente intratable. El problema se resuelve parcialmente ya que se pueden obtener buenas aproximaciones para el caso de tráfico pesado, que servirá de cota para cualquier otro caso más ligero. En RTQT este caso se alcanza cuando ($\rho = 1$), donde ρ es la intensidad del tráfico e intenta significar el momento de mayor necesidad de cómputo a través del momento en el que llegan más tareas.

1.4. Sistemas de Planificación para Aplicaciones de Múltiples Tipos

Una vez introducido nuestro problema, la necesidad de una plataforma de experimentación flexible para llevar a cabo nuestros estudios se impone. Necesitamos estudiar la planificación temporal de aplicaciones con características SRT, tanto locales como paralelas, en entornos no dedicados. Cabe destacar que además hemos de proveer soporte para la ejecución de aplicaciones Best-effort, que también podrán ser paralelas o locales.

Nuestro grupo ha desarrollado CISNE (*Cooperative & Integral Scheduling for Non-dedicated Environments*) [36, 39], una propuesta para la utilización de recursos no dedicados, que implementa una Máquina Virtual Paralela (MVP) y utiliza técnicas de planificación de aplicaciones. Este sistema proporciona una doble funcionalidad (Figura. 1.12): ejecutar aplicaciones paralelas de tipo Best-effort y aplicaciones locales (Best-effort), pertenecientes a los usuarios locales del cluster no dedicado.

En la implementación inicial de CISNE, cada nodo del cluster es compartido en el tiempo por ambos tipos de carga Best-effort: local y paralela. En consecuencia el sistema ha de gestionar el uso de los recursos entre las aplicaciones que se ejecutan, considerando que las tareas locales no pueden verse ralentizadas. CISNE debe garantizar el progreso de las tareas de las aplicaciones paralelas Best-effort en ejecución, de forma tal que el usuario local no note una intrusión en su ordenador. Esta propuesta (CISNE) se basa en una técnica de *Planificación de Aplicaciones Paralelas*, que considera las características de las aplicaciones distribuidas y el estado del entorno para ejecutar las aplicaciones de los usuarios paralelos.

Para aplicar la Planificación de Aplicaciones se analiza el problema desde dos dimensiones opuestas y complementarias: el espacio y el tiempo. Como se ha dicho, cada nodo de la MVP ha de ser capaz de gestionar el uso de CPU entre las tareas en ejecución, aspecto que que se conoce como ***Planificación Temporal*** (P.T.). Desde el punto de vista del espacio, el sistema ha de ser capaz de asignar el conjunto de nodos que conforman el cluster no dedicado a las aplicaciones paralelas que los necesiten, garantizando que ningún nodo será sobrecargado de forma que las tareas locales vean alterada su capacidad de respuesta. Este tipo de planificación es conocida como ***Planificación Espacial*** (P.E.) y es el principal objetivo tomado en cuenta en el diseño original del sistema CISNE.

Para evaluar nuestras propuestas, hemos de extender las funcionalidades de CISNE, principalmente siguiendo las directivas enumeradas a continuación:

1. Los tipos de cargas de trabajos soportados por el sistema (aplicaciones locales y paralelas de tipo Bes-effort) han de ser ampliados para so-

portar nuestras necesidades de experimentación. Hemos de dotar a CISNE de soporte para aplicaciones con características SRT, tanto locales como paralelas, pues nuestro escenario de experimentación está compuesto por los cuatro tipos de carga de trabajo.

2. Ha de extenderse el diseño de CISNE para permitir un marco flexible en la planificación temporal, ya que su idea inicial de diseño está enfocada en la planificación espacial.

3. Mantener la acertada filosofía de ser capaces de predecir, con niveles aceptables de precisión, los tiempos de turnaround de las aplicaciones paralelas. Cabe destacar que el nuevo escenario implica el desarrollo de nuevos métodos de estimación.

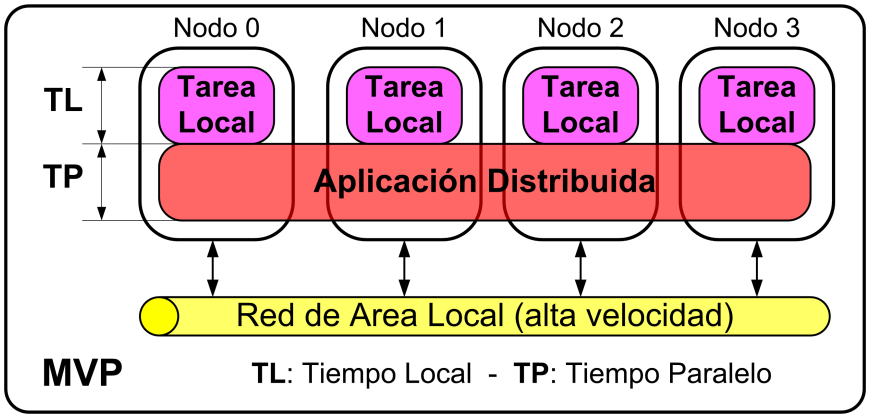

Figura 1.12: Máquina virtual paralela.

Recalcamos que al grupo inicial de aplicaciones previstas en el diseño de CISNE (locales y paralelas de tipo Best-effort), hemos de agregar los nuevos tipos de aplicaciones (descritas con más detalles en la sección 1.3.2.1) que han aparecido, siendo deseable mantener la alta capacidad de predicción lograda en CISNE [41]. Finalmente, los tipos de aplicaciones con las que ha de ser capaz de trabajar el sistema para desarrollar nuestros estudios serían:

- *Locales Best-effort*: Tipo de aplicaciones locales "comunes", usualmente editores de texto, compiladores y aplicaciones con niveles de interactividad que pueden ser medidos con respecto a la respuesta en un tiempo acotado por la capacidad de reacción del ser humano al utilizar el teclado o el ratón. Contempladas en el diseño original de CISNE.

- *Paralelas Best-effort*: Aplicaciones paralelas para las cuales no existen limitaciones en el turnaround o exigencias de QoS. La principal cualidad deseada consistía en predecir lo mejor posible el turnaround

para lograr una mejor planificación y brindarle información al usuario paralelo. Contempladas en el diseño original de CISNE.

- ***Locales SRT***: Aplicaciones con requerimientos de recursos determinados, usualmente necesitan la CPU de forma periódica para su correcta ejecución. Es también una buena idea garantizarles la cantidad de memoria principal que necesitan, ya que en caso contrario podrían no ejecutarse correctamente. Descritas en la sección 1.3.2.1.El soporte para este tipo de aplicaciones ha de ser **añadido** al sistema CISNE.

- ***Paralelas SRT***: Aplicaciones paralelas con un turnaround determinado debido a sus características. Descritas en la sección 1.3.2.1. El soporte para este tipo de aplicaciones ha de ser **adicionado** al sistema CISNE.

Es válido destacar que aún cuando en la literatura encontramos trabajos que estudian el comportamiento de las aplicaciones paralelas SRT en clusters, nuestro trabajo se diferencia por el hecho de contemplar carga local, tanto de tipo Best-effort como SRT. En [84, 88] se estudia el comportamiento de aplicaciones SRT en clusters dedicados y [23, 78] describen herramientas dedicadas a la planificación espacial de tareas paralelas SRT. En [23] los nodos se seleccionan de acuerdo a estudios probabilísticos y en [78] se intenta garantizar los deadlines de las tareas creando las tareas como parejas. Este enfoque permite tener dos niveles de prioridad, para poder lanzar la tarea con más prioridad si se estima que perderá su deadline.

Por otro lado, estudios como [1] intentan garatizar un recurso crítico, en este caso el ancho de banda de red, empleando mecanismos de QoS. En [85] también se hace uso de mecanismos de QoS para intentar garatizarles recursos a algunos tipos de aplicaciones, la novedad es que particionan los recursos disponibles para lograr hacerlo.

Una vez introducido el estado del arte y los conceptos básicos para comprender el alcance y enfoque de nuestro trabajo, pasamos a introducir los objetivos que nos proponemos.

1.5. Objetivos

Este trabajo es una reacción ante las posibilidades de investigación que se abren al redefinirse un escenario muy conocido y estudiado. El nacimiento de *nuevos tipos de aplicaciones* (locales y paralelas de tipo SRT) impone un estudio en las técnicas de *planificación temporal* para lograr la coexistencia de tipos de aplicaciones disimilares en clusters no dedicados.

Se ha llevado a cabo un esfuerzo de *investigación en la literatura existente*, considerable si tenemos en cuenta que unimos dos campos de investigación bien definidos. La planificación de aplicaciones, ya sea temporal o espacial, está en el punto de mira de un amplio sector de la comunidad científica. Nuestro trabajo *mezcla la planificación temporal en clusters no dedicados con aspectos de tiempo real débil*, siendo este último campo también muy explorado y pujante en el mundo científico.

Con este trabajo, aún en etapa embrionaria, hemos intentado sembrar la semilla para una investigación futura rica en posibilidades. Es nuestra finalidad principal poder *analizar el comportamiento de nuevos tipos de aplicaciones, con características SRT en clusters no dedicados*. Su interacción con las aplicaciones de tipo Best-effort, ampliamente contempladas en estudios previos, es un objetivo también incluido entre los nuestros. Para este fin hemos creado una plataforma de simulación que intentamos sea lo más flexible posible, teniendo en cuenta que conecta con prototipos de investigaciones previas no diseñados para este fin específico.

Con el fin de contrastar nuestro método de simulación, creamos *dos nuevos núcleos* para realizar nuestra experimentación, uno analítico y otro basado en dos niveles de simulación. Ambos proyectos están en desarrollo, pero brindan la grandes oportunidades para nuestros fines. Estos dos métodos son capaces de procesar los tipos de aplicaciones que necesitamos, paralelas y locales de tipo SRT y Best-effort.

1.6. Organización de la Memoria

En este capítulo introductorio hemos descrito las dos áreas generales en las que se enmarca nuestra investigación, la *Planificación Temporal en Clusters No Dedicados* en la sección 1.2 y los nuevos tipos de *aplicaciones de Tiempo Real Débil*, en la sección 1.3. Para facilitar la comprensión de los sistemas SRT, hemos introducido antes los conceptos básicos de Tiempo Real dividiendo la sección previa en dos partes. En la sección 1.4 mostramos la conjunción de los conceptos antes introducidos, y es nuestro primer intento de describir nuestro problema. Posteriormente enumeramos los objetivos de este trabajo.

El Capítulo 2 describe nuestra aproximación para estudiar la problemática antes introducida. En este capítulo comenzamos por introducir la arquitectura que tomamos como base (subsección 2.2). La necesidad de calcular el tiempo remanente de ejecución ($RExT$) es descrita en la subsección 2.3.1 y luego los dos métodos de estimación de $RExT$ propuestos, uno analítico (subsección 2.3.2) y el otro consiste en simulación a dos niveles (subsección 2.3.3).

El núcleo de estimación simulado consiste en un software de simulación independiente, y su implementación es descrita en el Capítulo 3. La implementación emplea una librería de simulación orientada a eventos, basada en entidades y eventos. Los principales eventos y entidades definidas, así como una introducción a la librería, son tratados en la sección 3.1. Posteriormente describimos con más detalles la forma de agregar nuevos algoritmos RT al software (sección 3.2) y la forma de comunicación entre los dos procesos de simulación (sección 3.3).

Una vez definido el problema, los modelos propuestos (analítico y simulado) y los detalles de implementación, mostramos la experimentación realizada y los resultados en el Capítulo 4. El mismo comienza por la validación de método (sección 4.2) y la escalabilidad (sección **??**). En la sección 4.3 mostramos la experimentación realizada incluyendo cargas de tipo SRT, tanto local como paralela y la combinación de ambos tipos de cargas.

Finalmente, el Capítulo 5 está dedicado a las conclusiones de este estudio y las contribuciones realizadas.

Cabe destacar que los Apéndices incluidos (A y B) describen los pasos realizados para incluir en nuestros estudios las nuevas plataformas hardware fácilmente accesibles hoy en día, los procesadores multicore.

Capítulo 2

Arquitectura General del Sistema

En el presente capítulo se describe la arquitectura general del método de simulación propuesto, basado en una serie de extensiones al entorno de planificación CISNE. Este entorno fue diseñado con el objetivo de estudiar la planificación espacial de aplicaciones paralelas en clusters de ordenadores no dedicados. Este sistema (CISNE en su versión original) se construye a partir de dos subsistemas, al nivel más alto se realiza de forma centralizada la planificación espacial (P.E.) empleando *LoRaS*, el cual distribuye la carga paralela en el cluster. El subsistema de bajo nivel implementa un entorno de planificación temporal (P.T.) en cada nodo, que ha de lograr la coplanificación y balanceo de recursos asignados a las tareas pertenecientes a un mismo trabajo paralelo. A su vez es responsable de preservar el rendimiento del usuario local.

CISNE es un entorno que consta de dos modos de ejecución, uno en el cual se realizan ejecuciones reales y otro para realizar simulaciones off-line. Aunque nuestro objeto de interés a largo plazo es dotar al sistema de soporte para lanzar aplicaciones SRT o simulaciones que contemplen este tipo de carga (SRT), el progreso hecho hasta ahora se relaciona sólo con el modo de simulación off-line. Hemos modificado el modo de simulación off-line adicionándole dos nuevos núcleos de estimación, el soporte para la entrada de datos que caracterizan aplicaciones SRT y además ha sido necesario definir un método de intercambio de datos para el núcleo simulado. Se incluye también una breve descripción de la arquitectura del modo de simualción off-line de CISNE, de tal forma que permita comprender la extensión de las modificaciones.

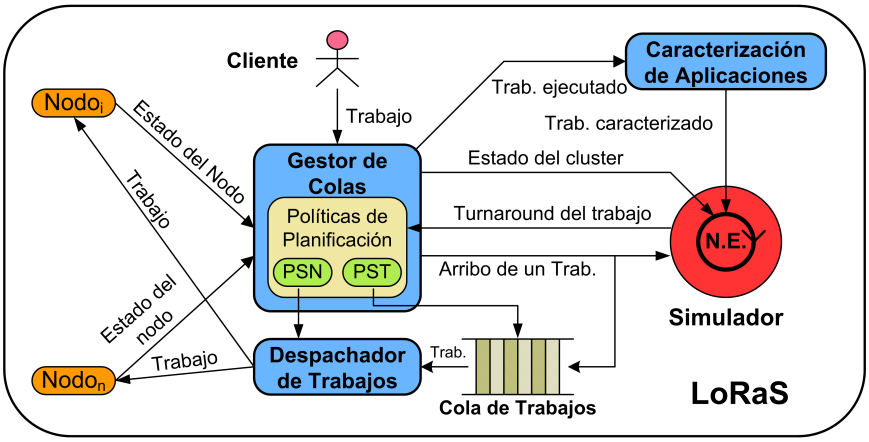

Figura 2.1: Arquitectura del sistema de predicción por simulación integrado en LoRaS.

2.1. Subsistema LoRaS

LoRaS es uno de los dos componentes fundamentales de CISNE, siendo responsable de aceptar peticiones de ejecución de aplicaciones paralelas y definir el mejor momento y lugar para ejecutar tales aplicaciones. La arquitectura del sistema LoRaS es centralizada con demonios en cada nodo para controlar sus estados.

La arquitectura del sistema de simulación se divide en dos partes: por un lado la simulación de las políticas de planificación configurables en el entorno, y por otro lado la estimación del tiempo de ejecución de las aplicaciones, una vez que éstas han sido planificadas por el simulador. Cabe destacar que las extensiones realizadas al entorno son dos nuevos métodos de estimación, que se acoplan en el sistema en la forma de núcleos de estimación (N.E en la Figura 2.1).

Los N.E. requieren de dos tipos de información: una caracterización de las aplicaciones a ejecutar (recursos de memoria y CPU consumidos, cantidad de nodos necesarios, tiempo de ejecución en dedicado, etc.) y el estado actual del entorno (cantidad de recursos totales y ocupados en los nodos, cuantas aplicaciones paralelas y en que nodos se encuentran en ejecución, la actividad local en el entorno, etc.). Esta información es provista por los módulos de Caracterización de Aplicaciones y el Gestor de Colas, respectivamente. La utilización del módulo de Caracterización de Aplicaciones permite desligar el proceso de estimación, de valores de caracterización provistos por el usuario paralelo, que suelen encontrarse alejados de la realidad según estudios como [86, 60].

2.2. Arquitectura del simulador fuera de línea

La arquitectura del simulador off-line, se basa en la "duplicación" de módulos, reemplazándolos por otros (dummies), para proveer al gestor de colas de un entorno en el que pueda planificar aplicaciones como si realmente éstas fuesen ejecutadas. Ha de destacarse que todos los módulos dummies se encuentran bajo el control del simulador.

Para realizar las simulaciones off-line en CISNE es necesario proveer en la configuración del sistema información sobre las características del entorno (configuración del entorno en la Figura 2.2). Por ejemplo hemos de entrar en los ficheros de configuración la memoria total, memoria inicialmente en uso, la potencia de la CPU y si existe, cuál es la carga inicial de CPU para cada nodo incluido en el sistema a simular. Esta caracterización es importante, porque en el sistema en producción las características de los nodos son obtenidas por el sistema desde los nodos reales cuando se inicia.

En este caso sin embargo, si tratamos con una simulación el sistema no podrá interrogar a los nodos reales para obtener información sobre sus características y carga actual. De la misma forma y por las mismas razones, también ha de proveerse para la simulación una caracterización de las tareas locales (nodo en que se ejecutarán, tiempo de inicio y fin y consumo de recursos de memoria y CPU) que se tendrán en el entorno a lo largo de la ejecución de la simulación (configuración de la carga Paralela (Local en la Figura 2.2). Finalmente y debido a que el sistema ahora no presta un servicio de planificación a usuarios reales, hemos de proveer la carga de aplicaciones paralelas a ejecutar en ficheros de entrada, como hemos hecho antes con al carga local. Para esto, se provee al sistema de una lista de aplicaciones a ejecutar, junto con el tiempo de llegada de cada una.

2.3. Extensiones incluidas

En esta sección describiremos las modificaciones, en forma de extensiones, añadidas al sistema CISNE hasta el momento. Hemos de destacar que las extensiones sólo cubren el modo de simulación off-line (descrito en la sección 2.2) del sistema. Este grupo de modificaciones va dirigido a estudiar el comportamiento de las NOWs frente a los nuevos tipos de cargas SRT, descritos en la sección 1.3.2.1.

Dado que de momento no somos capaces de ejecutar carga SRT de tipo local o paralelo en nuestro entorno (CISNE) con las características de QoS exigidas, a efectos de poder experimentar con propuestas de planificación que contemplen los nuevos requerimientos de las aplicaciones SRT; se han desarrollado modelos analíticos y núcleos de simulación que nos permitirán

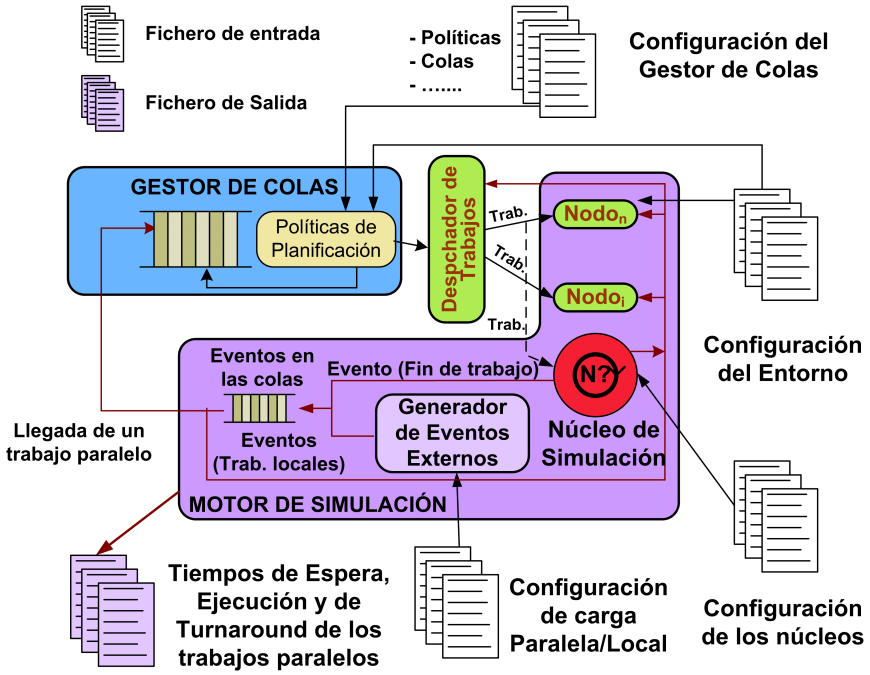

Figura 2.2: Vista modular de la arquitectura del sistema de simulación fuera de línea en LoRaS.

formular nuevas propuestas en este sentido. De esta forma logramos establecer cierto nivel de validación en nuestros métodos, los cuales son descritos en esta sección.

2.3.1. Tiempo Remanente de Ejecución

Los dos módulos fundamentales de la arquitectura en modo de simulación off-line son mostradas en la Figura 2.2, el Gestor de Colas (descrito en la sección 2.1) y el Motor de Simulación.

El simulador de LoRaS funciona dirigido por eventos discretos, como los son la llegada, arribo o finalización de una tarea de cualquier tipo de trabajo. Para realizar las simulaciones, el motor necesita de tres ficheros de configuración:

- **Configuración del Entorno**: En este fichero especificamos el conjunto de nodos a utilizar por el Gestor de Colas para ejecutar la carga paralela. Contiene la cantidad de nodos disponibles y sus principales características (poder de cómputo, tamaño de la memoria principal, cantidad de recursos consumidos por las cargas locales, etc.).

- **Configuración del Núcleo de Simulación**: Permite elegir el método empleado para calcular el valor del Tiempo Remamente de Ejecución (*RExT*, *Remanent Execution Time*) por el Gestor de Colas. Puede ser tanto un método analítico (los incluidos en el diseño original), como el método simulado implementado para este trabajo.

- **Configuración de la Carga**: Contiene la lista de trabajos, tanto paralelos como locales, a simular por el entorno. De cada trabajo se necesita información detallada, que por ejemplo incluye su tiempo de ejecución en isolación, tiempo de arribo y requerimientos máximos de memoria y CPU.

Una vez que todos los ficheros de configuración se han cargado, el entorno está listo para comenzar el proceso de simulación. Es importante destacar que los valores estáticos utilizados para describir la carga a simular, tales como su tiempo de ejecución de forma aislada o requerimientos máximos de CPU o memoria, son recolectados por el mismo entorno para su uso futuro. Toda esta infomación, junto al estado del cluster, es empleada en generar los conjuntos de datos que conforman las diferentes etapas del proceso de simulación. Hemos de destacar que el evento de llegada de una aplicación paralela marca el comienzo de una nueva etapa, ya que implica una reestimación de los tiempos de turnaround de todas las aplicaciones en ejecución, pues lógicamente todas ser verán afectadas por los recursos que esta consumirá. Cada vez que llega una aplicación paralela, se crea un conjunto de datos que incluye la nueva aplicación paralela y se procesa por el motor de simulación.

Como se ha descrito en la introducción (sección 1.3.1) de este trabajo, los sistemas RT y SRT necesitan realizar análisis de admisión en el momento de arribo de una tarea SRT, para saber si aceptándola en el sistema no se afectan a las demás tareas. Dado el progreso de nuestro trabajo, un análisis de admisión exhaustivo aún no es necesario. En el entorno se hace un análisis rudimentario de capacidad de CPU para aceptar o no una aplicación SRT, además, los niveles de aplicaciones SRT que arriban al sistema son acotados.

Todos los núcleos de estimación que se integran al entorno han de ser capaces de retornar una estimación del Tiempo Remanente de Ejecución (*RExT*, *Remanent Execution Time*) de las aplicaciones paralelas incluidas en el sistema, que es la base del funcionamiento del modelo de estimación del simulador. La idea es la de estimar para un entorno dado, y con un conjunto de aplicaciones en ejecución, cuál es la próxima aplicación que se espera finalice y cuales serán los recursos que se liberarán en tal caso. Los nuevos núcleos de estimación, con capacidad de procesar carga con características SRT, conforman el resto de este capítulo.

El Algoritmo 1 muestra el proceso de simulación de forma simplificada. El primer paso consiste en duplicar el estado del sistema, a los módulos *dummies* creados a tal efecto e inicializar las colas de trabajos y lista de nodos. El control de finalización del algoritmo es la condición del **wihle** de la línea 3, que controla los trabajos en ejecución. Dentro de este lazo principal se calcula $RExT$ cada trabajo en ejecución (línea 4) y se asume que la próxima aplicación en terminar será J_i, en el instante de tiempo t_i (línea 5). El siguiente paso consiste en actualizar el tiempo de finalización de la aplicación J_i y eliminarla de la cola de aplicaciones en ejecución DRQ. También han de actualizarce los tiempos que las restantes aplicaciones han estado en la cola DRQ. El lazo que se ejecuta entre las líneas 8 *y* 13 es el encargado de seleccionar las aplicaciones en la cola de espera (DQ) de acuerdo a las condiciones del sistema y las políticas en uso y pasarlas a la cola DRQ.

Algoritmo 1 Proceso de Simulación

1: Duplicar el estado del sistema a *dummy*: siendo DQ una copia de la cola de espera de trabajos, DRQ una copia de la cola de trabajos en ejecución y CL_{sim} una copia de los nodos que conforman el cluster y sus respectivos estados
2: Guardar el momento actual (t_0), como el momento en que la simulación ha comenzado.
3: **while** ($\exists J_k \in DRQ$) **do**
4: **forall** ($J_k \in DRQ$) **do** Calcular $RExT$ de J_k.
5: Asumir que la aplicación J_i es la próxima que finalizará en el tiempo t_i.
6: Actualizar el tiempo de finalización de J_i a t_i (i.e.: calcular el tiempo de ejecución para J_i), y eliminarlo de DRQ.
7: **forall** ($J_k \in DRQ$) **do** Calcular *tiempo de CPU usado* para $J_k \in t_i - t_0$ y actualizarlo en las respectivas aplicaciones J_k.
8: **while** ($\exists$ recursos disponibles Cl_{sim} y algún trabajo en DQ) **do**
9: Buscar una aplicación $J_x \in DQ$ que pueda ser ejecutada en el estado actual del sistema (Cl_{sim}).
10: Seleccionar el mejor subconjunto de Cl_{sim} para ejecutar J_x, empleando la política del sistema.
11: Ejecutar la aplicación J_x en el subconjunto seleccionado de Cl_{sim} y adicionarla a DRQ.
12: Incrementar el tiempo de espera estimado de J_x en $t_i - t_0$.
13: **end while**
14: **forall** ($J_j \in DQ$) **do** Incrementar el tiempo de espera estimado de J_j en $t_i - t_0$.
15: Asignar $t_0 = t_i$.
16: **end while**

2.3.2. Núcleo analítico

Para facilitar la comprensión del método analítico propuesto en este trabajo, explicaremos antes otro método similar sin capacidad SRT, que llamaremos CPU. Introducimos también la notación $RExT_{ANL-SRT}$, que denotará nuestro método analítico capaz de realizar estimaciones con cargas SRT.

Nuestro método analítico sin capacidad SRT (CPU) comienza por calcular el $RExT$ que la aplicación necesitaría si se ejecutara de forma aislada, a este valor lo llamaremos $RExT_{isol}(j)$ y se calcula según la Ecuación 2.1.

$$RExT_{isol}(j) = \frac{t_{total}(j) \times (t_{total\_CPU}(j) - t_{used\_CPU}(j))}{t_{total\_CPU}(j)}, \tag{2.1}$$

En esta ecuación, $t_{total}(j)$ es el tiempo total de ejecución, $t_{total\_CPU}(j)$ el tiempo de CPU de la aplicación ejecutada de forma aislada y $t_{used\_CPU}(j)$ el tiempo de CPU que ha empleado desde su comienzo, todos estos valores asociados a la aplicación paralela j .

Ha de destacarse que la Ecuación 2.1 asume que $RExT_{isol}(j)$ es proporcional al tiempo total de ejecución de forma aislada ($t_{total}(j)$) limitado por el tiempo de CPU que consumirá ($t_{total\_CPU}(j) - t_{used\_CPU}(j)$) y el tiempo total de CPU que necesita la aplicación ($t_{total\_CPU}(j)$).

El próximo paso en el método CPU es considerar el tanto porciento de CPU requerido por las tareas. De acuerdo a esto, el valor del $RExT(j)$ se calcula de acuerdo la la siguiente ecuación:

$$RExT_{CPU}(j) = RExT_{isol}(j) \times \frac{CPU(j)}{CPU_{feasible}(j)}, \tag{2.2}$$

donde $CPU(j)$ es el tanto porciento de CPU ($t_{total\_CPU}(j)/t_{total}(j)$) que la aplicación puede utilizar y

$$CPU_{feasible}(j) = min(CPU(j), \frac{CPU(j)}{CPU_{max}(j)}) \tag{2.3}$$

es el máximo tanto porciento de CPU que esperamos la aplicación j consuma. Finalmente

$$CPU_{max}(j) = max(CPU_{par}(n) + CPU_{loc}(n) \mid n \in N(j)) \tag{2.4}$$

donde $CPU_{loc/par}(n)$ es la suma del uso de CPU de cada tarea local/paralela ejecutándose en el nodo n. Destacamos que estos valores representan los requerimientos máximos de uso de CPU (en porciento) entre los nodos donde la aplicación paralela j está en ejecución.

Una vez descrito el método $RExT_{CPU}$, lo usaremos como base para el método analítico capaz de lidiar con cargas SRT. El método $RExT_{ANL\_SRT}$ se basa en considerar cuales de los requerimientos de la tareas, ya sea locales o paralelas, son SRT. Para lograr esto, se redefine la expresión $CPU_{feasible}(j)$ de $RExT_{CPU}(j)$ de la siguiente manera:

$$CPU_{feasible\_SRT}(j) = \begin{cases} CPU(j) & j \in App_{SRT} \\ min(CPU(j), CPU_{min\_SRT}(j)) & j \notin App_{SRT} \end{cases} \quad (2.5)$$

donde App_{SRT} denota el conjunto de aplicaciones paralelas SRT en actualmente ejecución en el cluster y $CPU_{min\_SRT}(j)$ representa la cantidad mínima de CPU de la que dispondrá la aplicación paralela j a lo largo de todo el cluster y se calcula con la siguiente ecuación:

$$CPU_{min\_SRT}(j) = min(\frac{CPU(j) \times (100 - CPU_{SRT}(n))}{CPU_{no\_SRT}(n)}) \mid n \in N(j) \quad (2.6)$$

donde $CPU_{SRT}(n)$ y $CPU_{no\_SRT}(n)$ representan las sumas de CPU requeridas por cada tarea SRT y no SRT, respectivamente, ejecutándose en el nodo n. Las tareas pueden ser locales o paralelas.

2.3.2.1. Método de estimación MPL

MPL es un método de estimación del $RExT$ propuesto en [36] y al igual que el método CPU (descrito anteriormente como introducción al núcleo analítico $ANL - SRT$), se basa en multiplicar por un factor el tiempo remanente de ejecución que la aplicación necesitaría en caso de ejecutarse de forma aislada, que se calcula usando la Ecuación 2.1.

En este caso, el factor se calcula utilizando la Ecuación 2.7, la cual retorna el número máximo de tareas, considerando tanto locales ($MPL_{local}(n)$) como paralelas ($MPL_{paral}(n)$), que se ejecutan concurrentemente con j entre todos los nodos utilizados por j para su ejecución ($nodos(j)$).

$$MPL_{max}(j) = max(MPL_{paral}(n) + MPL_{local}(n) \mid n \in nodes(j)). \quad (2.7)$$

Finalmente, el valor del $RExT$ en el método MPL se calcula utilizando la Ecuación 2.8.

$$RExT_{MPL}(J) = RExT_{isol}(j) \times MPL_{max}(j) \quad (2.8)$$

Cabe destacar que el objetivo de la Ecuación 2.8, es el de ponderar el tiempo de ejecución que tendría la aplicación j ejecutada en un entorno dedicada a la misma (calculado empleando la Ecuación 2.1), considerando la mayor carga actual con la que alguna de sus tareas ha de compartir el nodo, y por lo tanto los recursos de cómputo.

2.3.3. Método simulado

Nuestra alternativa al modelo de analítico propuesto la constituye un nuevo núcleo de estimación desarrollado, capaz de procesar carga con características SRT está basado en la simulación. Análogamente a como hicimos con el método anterior, lo denotaremos $RExT_{SIM\_SRT}$ y es importante destacar que es un simulador *externo* al entorno.

Aún siendo un programa externo al entorno, ha de devolver una estimación del $RExT$, al igual que los métodos analíticos. Para lograr este objetivo, es necesario establecer una interfaz de comunicación entre los dos programas (simuladores), para que el simulador externo ($RExT_{SIM\_SRT}$ de ahora en adelante) pueda recibir los datos de entrada y devolver los resultados. Debido a que los dos programas estaban escritos en diferentes lenguajes de programación, se optó por comunicarlos mediante ficheros, una vía cómoda y simple. En nuestro caso, empleamos el formato XML ya que permite crear plantillas y comprobar la consistencia de los datos con facilidad. El funcionamiento de la interfaz mediante ficheros XML es mostrado en la Figura 2.3.

Al igual que el método análitico antes descrito, se toma una instantánea del estado del sistema (conformada por el estado del cluster y los datos de las aplicaciones) y se realiza una simulación. Con la diferencia de que el nivel de detalle alcanzado es mucho mayor que en el método analítico. Esto se debe a que este método de estimación del $RExT$ es en realidad un motor de simulación completo; siendo capaz, por ejemplo, de planificar las tareas con diferentes políticas de asignación teniendo en cuenta si las tareas son SRT o no.

Hemos de destacar también que $RExT_{SIM\_SRT}$ lleva a cabo su estimación del $RExT$ realizando una simulación del estado de cada nodo, tomando en cuenta políticas diferentes políticas de asignación (diferentes si un trabajo es SRT o no), recursos disponibles y los mensajes intercambiados por las aplicaciones paralelas. A continuación analizamos un conjunto de casos de uso representativos de aplicaciones SRT contempladas en este estudio y posteriormente describimos la manera en la se administran los principales recursos en nuestro simulador.

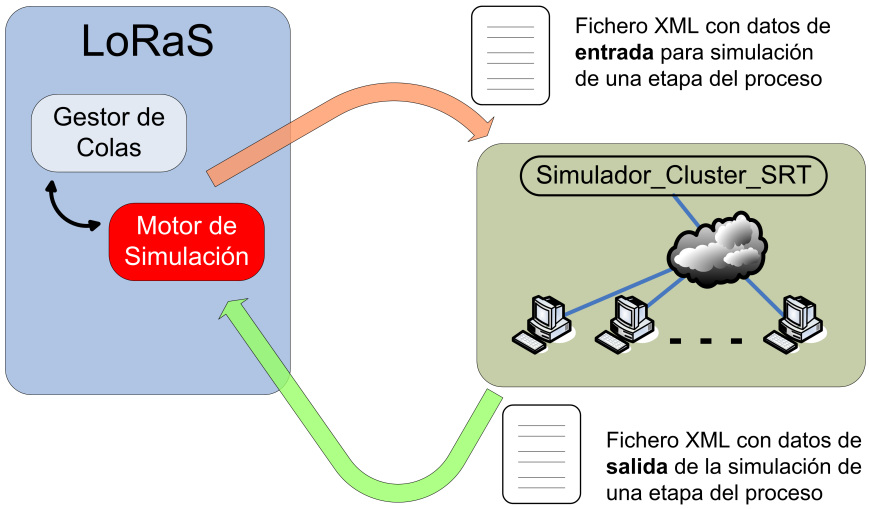

Figura 2.3: Esquema de la simulación a dos niveles.

2.3.3.1. Análisis de casos representativos de aplicaciones SRT

Dada la complejidad del entorno de planificación, hemos de centrarnos en lograr una correcta predicción de los fallos en trabajos pertenecientes a tareas (aplicaciones) SRT. La Figura 2.4 muestra dos casos representativos de aplicaciones SRT. El caso A se incluye con fines comparativos y muestra el comportamiento de la planificación para aplicaciones RT, además de las variables que lo definen. El caso B es un ejemplo de aplicación local periódica SRT, en el cual podemos observar que los tiempos de cómputo (los etiquetados como C_i *real*) de la aplicación local varían para cada arribo de un trabajo. Este comportamiento corresponde con los requerimientos de CPU de un vídeo en formato *mpeg*, el cual tiene frames de diferente tamaño, por lo cual cada uno puede requerir C_i diferentes. Para lograr una correcta predicción y planificación de las tareas Best-effort y SRT tanto locales como paralelas, nos vemos en la necesidad de otorgarle la CPU en tiempos conocidos (representados por los C_i *deseado* en la figura).

La Figura 2.4.C, muestra el caso correspondiente a una aplicación paralela con turnaround acotado, un caso que podría ser tratado como una aplicación. Este tipo de aplicación podría planificarse otorgándole la CPU con C_i largos en algún momento de su ejecución, de acuerdo a la carga que tenga el nodo donde se ejecuta, para intentar que termine dentro de su deadline. Dado que es una aplicación aperiódica, no se vería afectada por este tratamiento. Este enfoque, aunque tal vez sea ventajoso desde algunos aspectos, afecta nuestra capacidad de predicción y dificulta enormemente todo el proceso de planificación. Creemos que es mejor asignarle un C_i que le permita terminar dentro de su deadline (representado por C_i *deseado*), aunque podríamos otorgarle

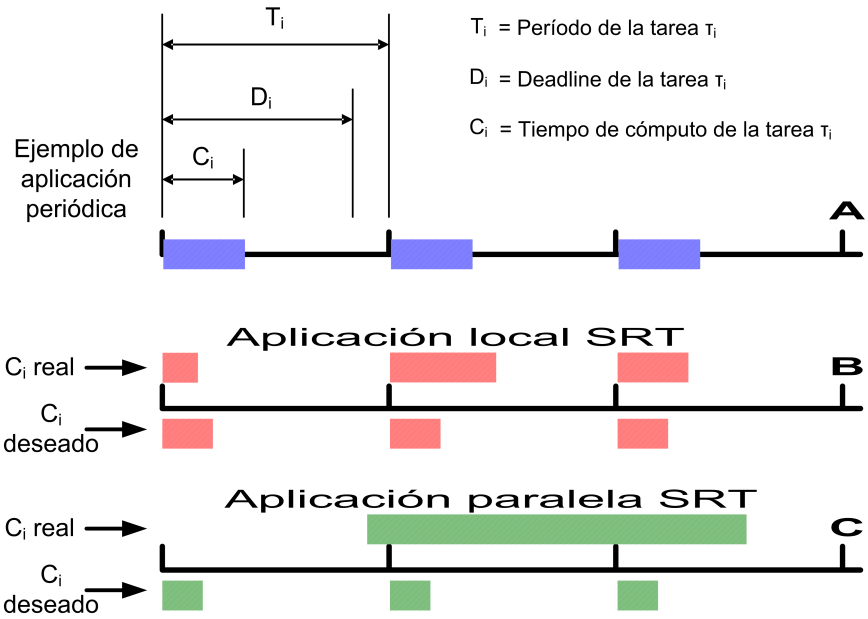

Figura 2.4: Ejemplo de modelos de planificación para aplicaciones SRT

más de acuerdo al estado del nodo y utilizar el C_i *deseado* como cota mínima del tiempo de cómputo que podemos otorgarle a esta aplicación paralela.

2.3.3.2. Planificación de la CPU

La gestión eficiente de la CPU, uno de los recursos más estudiados tanto en la de la de la planificación de aplicaciones RT como SRT, merece un aparte en este trabajo. En $RExT_{SIM\_SRT}$ la planificación de la CPU se hace de acuerdo al tipo de tareas, es decir, las tareas SRT y las Best-effort se planifican de acuerdo a diferentes políticas y criterios de prestaciones.

Para la *planificación de las aplicaciones Best-effort*, se utiliza el tiempo de cómputo después luego de procesarse las necesidades de cómputo de todas las aplicaciones SRT. Este tiempo de cómputo asignado a las tareas Best-effort se puede planificar con alguna de las dos políticas: *Round Robin* o *Coscheduling Cooperativo* [34]. Cabe destacar que la extensión realizada en el simulador contempla la posibilidad de testear y analizar nuevas políticas con un esfuerzo razonable.

De acuerdo con los análisis antes expuestos, la forma de *gestionar la CPU en presencia de aplicaciones SRT* se hace de acuerdo a si es local o paralela. En caso de ser local, solo necesitamos garantizarle sus requerimientos de CPU de manera periódica. El problema torna a ser más complicado cuando estamos en presencia de una aplicación paralela SRT, que en nuestro estudio está

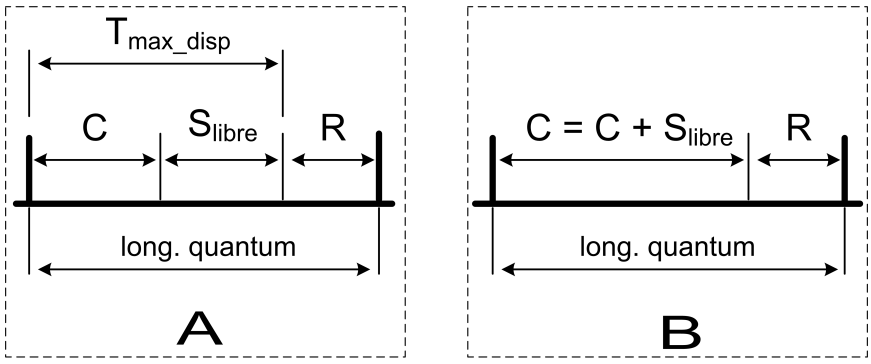

Figura 2.5: Asignación dinámica del quantum de CPU en el núcleo simulado.

representada por una aplicación paralela con turnaround acotado. En una aproximación inicial, parecería que es suficiente con asignarle de forma periódica la CPU en cada época (según la definición adoptada en sistemas Linux), con un $C_i = C_{par}(j,n)$ calculado con una ecuación similar a la Ecuación 2.9.

$$C_{par}(j,n) = \frac{T_{par}(j,n) \times (t_{used\_CPU}(j)}{(D(j) - t_{exec}(j))} \times 100. \tag{2.9}$$

Con esta ecuación podemos calcular el tiempo de cómputo ($C_{par}(j,n)$) periódico (cada vez que transcurra el tiempo $T_{par}(j,n)$) que necesitaría la aplicación paralela SRT j en el nodo n para terminar dentro del deadline correcto ($D(j)$). El valor de $t_{used\_CPU}(j)$, al igual que la Ecuación 2.1 de la sección 2.3.2, representa el tiempo de CPU que ha empleado desde su comienzo la aplicación paralela y $t_{exec}(j)$ es el tiempo que ha pasado en ejecución la aplicación paralela j desde su comienzo.

Sin embargo, aunque este enfoque permite reservar tiempo de cómputo para una aplicación paralela, es en extremo pesimista. Con un valor calculado en base al deadline deseado, sólo se logra que la aplicación paralela SRT termine cerca del valor usado como base del cálculo. La Figura 2.5 muestra dos casos de asignación del quantum de la CPU. En esta figura C representa el valor de tiempo de cómputo reservado a una aplicación paralela SRT (calculado con la Ecuación 2.9), R el slice del quantum reservado a otras aplicaciones SRT locales o paralelas y S_{libre} el segmento de quantum de CPU que no está en uso por ninguna aplicación SRT. En el caso **A** podemos notar que la aplicación paralela podría recibir un slice mayor, representado por T_{max_disp}, y de esta forma terminar antes del deadline asignado. El caso **B** muestra el comportamiento de nuestro simulador, en el cual la aplicación paralela SRT de mayor prioridad recibe el máximo slice del quantum posible. Empleando este enfoque, las aplicaciones paralelas SRT pueden tomar ventaja de cualquier momento de baja carga en los nodos y finalizar antes.

2.3.3.3. Gestión de Memoria y Red

Ha de destacarse que aún cuando LoRaS entrega una carga balanceada al núcleo de estimación simulado, teniendo en cuenta el estado del nodo y sus recursos, $RExT_{SIM\_SRT}$ es capaz de controlar la memoria usada en el nodo de acuerdo al Contrato Social (límite en los recursos que podemos emplear para cómputo paralelo respetando al usuario local, estudiado en [10]) definido en el momento de su ejecución.

El recurso Red también ha sido tomado en cuenta en el diseño del núcleo simulado. En nuestro simulador, las aplicaciones paralelas generan mensajes de acuerdo a su caracterización. Logramos esto empleando una implementación booleana de la distribución de Bernoulli, que inicializamos con el resultado de $100 - getCPUUsage(job)$, basándonos en la idea de que el tiempo que no se gasta en CPU se gasta en comunicaciones.

En base al valor generado por la distribución de Bernoulli, podemos decidir si la aplicación comunica o no. En caso de que comunique, generamos mensajes para ella en todos los nodos en los que hay trabajos de la aplicación paralela y los guardamos en el buffer local del nodo que los genera. Posteriormente cuando la aplicación paralela para la cual generamos los mensajes tiene asignada la CPU, revisa el buffer buscando mensajes y los envía a los nodos donde están el resto de los trabajos. Estos mensajes se guardan en los buffers de los nodos remotos, siendo este proceso retrasado para simular la demora de la red. A continucación cuando los trabajos en sus respectivos nodos tienen la CPU, procesan los mensajes.

Capítulo 3

Implementación del Simulador para Aplicaciones SRT

Como ya hemos mencionado antes (Capítulo 2), el núcleo de estimación simulado es una aplicación independiente al motor superior de simulación (LoRaS). Esta aplicación recibe el nombre de Simulador_Cluster_SRT y sus características principales son mencionadas en la sección 2.3.3. En este capítulo describiremos su arquitectura e implementación.

3.1. Arquitectura

La arquitectura del Simulador_Cluster_SRT está basada en un *framework* orientado a objetos diseñado para los programadores que usan Java y desarrollan modelos de simulación. A partir de este framework se estructura todo el modelo empleado, razón por la cual comenzaremos por introducirlo.

3.1.1. Framework DESMO-J

Según [48], DESMO-J es un framework orientado a objetos diseñando para los programadores que desarrollan modelos de simulación. "DESMO-J" significa "*Discrete-Event Simulation and MOdelling in Java*" (Simulación dirigida por eventos discretos y modelado en Java). Esta forma de nombrar el framework destaca las dos características más significativas de DESMO-J:

- DESMO-J funciona bajo el paradigma de la simulación dirigida por eventos discretos. En modelos de este tipo, todos los cambios de estado del sistema se supone sucederán en puntos discretos del tiempo. Entre dichos acontecimientos, el estado del sistema se asume seguirá siendo

el mismo. La simulación dirigida por eventos discretos es por lo tanto particularmente conveniente para los sistemas en los cuales los cambios del estado relevantes ocurren de forma repentina e irregular.

- DESMO-J está implementado en Java. Usar este framework para construir modelos de simulación implica la escritura de un programa en Java.

DESMO-J ha sido desarrollado en la Universidad de Hamburgo y en la actualidad es mantenido por un equipo de investigadores [21]. Este framework adiciona características que simplifican el desarrollo de simuladores dirigidos por eventos discretos. Entre ellas podemos mencionar:

- Clases para modelar componentes comunes de los modelos, como por ejemplo: colas y distribuciones estocásticas basadas en números aleatorios.
- Clases abstractas que pueden ser adaptadas a comportamientos específicos (modelos, entidades, eventos, procesos de simulación y otras).
- Una infraestructura de simulación lista para emplearse que comprende los planificadores, lista de eventos y reloj de simulación, todas encapsuladas en una clase llamada *Experiment*.

Cabe destacar que esta última clase denota una separación entre el modelado y la experimentación, lo cual facilia su uso. Todas las clases están contenidas en paquetes de Java para organizarlas y hacerlas más accesibles.

En el Simulador_Cluster_SRT se emplean parte de las clases brindadas por este framework, siendo las principales.

- *public abstract class* ***Entity***: representa la superclase para todas las entidades en un modelo. Se supone que las entidades serán programadas en cierto punto de simulación de acuerdo a eventos compatibles.
  Las clases que heredan de Entity encapsulan usualmente toda la información de entidades del modelo relevantes al modelador. Empleando los eventos, podemos cambiar el estado del modelo en cierto momento programable del tiempo.
- *public abstract class* ***Event***: provee la superclase para eventos definidos por el usuario que pueden cambiar el estado del modelo. Al ser un framework dirigido por eventos, los cambios de estado son generados por eventos que son programados en distintos puntos del tiempo de simulación. Un evento puede actuar solo en una entidad, cambiando su estado de acuerdo a la reacción programada de la entidad al evento específico.

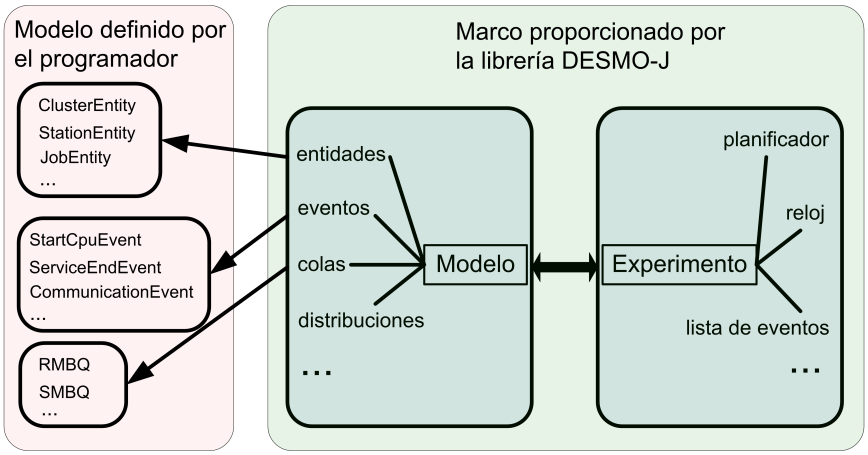

Figura 3.1: Interacción Modelo-Experimento en demoj.

- *public abstract class* ***Model***: las clases que heredan de esta superclase contienen todas las referencias a todos los componentes del modelo a simular.

Una vez dados a conocer los elementos básicos necesarios para comprender el framework usado, pasamos a describir las clases implementadas y su interacción. La Figura 3.1 muestra el esquema general de funcionamiento, en el mismo podemos apreciar la interacción existente entre los experimentos a ejecutar y el API base para definir modelos. También queda especificada la alto modularidad presente gracias al uso de desmoj.

3.1.2. Entidades relevantes

Agruparemos las entidades presentes en el modelo en dos grupos, el primero conteniendo las entidades que representan elementos de hardware y el segundo las entidades relacionadas con los trabajos y tareas representadas.

Antes de entrar en la descripción de las entidades relevantes, queremos destacar la clase base del simulador, nombrada ***SimCluster*** y mostrada en la Figura 3.3. Esta clase es el contenedor principal de todas las entidades incluidas en el diseño y la responsable de cargar los datos de entrada, generar todos los nodos con sus respectivos estados e iniciar el proceso de simulación. El proceso de inicio de la simulación incluye la creación de una instancia de la clase *desmoj.core.simulator.Experiment*, que es la que provee la infraestructura para la ejecución de una simulación. El Algoritmo 2 muestra de forma general las principales acciones a realizar para hacer simulaciones empleando el modelo definido. En este algoritmo, primero definimos las instancias y principales métodos que intervienen (líneas de la 1 a la 5) y luego mostramos

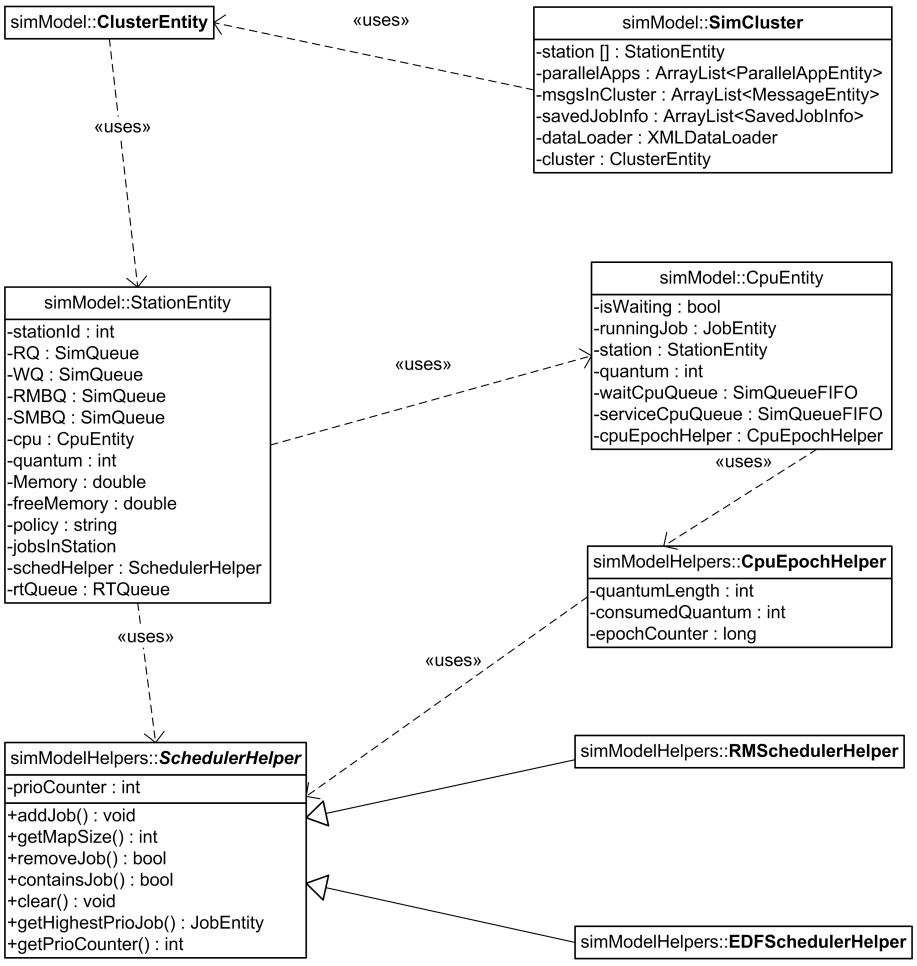

Figura 3.2: Interacción general de entidades ”hardware” presentes en el modelo.

las principales acciones a realizar para realizar un experimento. Con este algoritmo queremos mostrar los primeros pasos a realizar para poder ejecutar un experimento luego de definir un modelo.

Es interesante mencionar también la manera de detener los experimentos, que se basa en condiciones de parada. Las condiciones de parada han de ser clases que hereden de la clase *desmoj.Condition* e implementen el método *check* que es el empleado para determinar si las condiciones se cumplen o no. En nuestra implementación la condición de parada es que todos los trabajos paralelos (SRT o no) cargados en el fichero de datos de entrada terminen su ejecución. Para lograr esto, mantenemos un contador de las tareas de cada trabajo paralelo que van terminando su ejecución y al ser igual este contador al valor inicial de la cantidad de tareas del nodo, aumentamos el contador de cantidad de trabajos paralelos concluidos. El método check sólo ha de comparar el valor de este contador con la cantidad inicial de trabajos paralelos incluidos y detener la simulación en caso de ser iguales.

3.1.2.1. Entidades que modelan elementos "hardware"

Algoritmo 2 Algoritmo general de la simulación

1: SimulationModel → SimCluster Contiene las entidades y eventos del modelo
2: Experiment → Clase que provee la infraestructura para ejecutar simulaciones con los modelos definidos.
3: StopCondition → Condición de Fin, en este caso el experimento se detiene si todas las aplicaciones paralelas han terminado
4: SimCluster.**init**() → Inicializa el model, también carga los datos de *inputFile.xml*
5: SimCluster.**doInitialSchedules**() → Planifica los eventos iniciales del modelo, en este caso también crea las instancias de *StationEntity* y de *JobEntity* cargadas del fichero *inputFile.xml* y planifica los primeros eventos *StartCpuEvent*
6: **create new** Instancia de Experiment → *experiment*
7: **create new** Instancia de SimulationModel → *simModel*
8: *simModel*.**connectToExperiment**(*experiment*)
9: **create new** Instancia de StopCondition → *stopCondition*
10: *experiment*.**stop**(*stopCondition*)
11: simModel.**init**()
12: simModel.**doInitialSchedules**()
13: *experiment*.**start**()
14: *simModel*.**printResults**(*outFile.xml*)
15: *simModel*.**finish**()

class ***ClusterEntity***: entidad que representa a un cluster, es el contenedor principal de todos los elementos presentes en la simulación. Mantiene la lista de los nodos pertenecientes al cluster y otras informaciones de carácter global, como una lista con todos los mensajes que están en movimiento en el cluster en cada momento.

class ***StationEntity***: entidad que representa a un nodo del cluster. Encapsula los objetos que representan los recursos del ordenador, entre ellos la CPU y sus modelos de planificación. Mantiene las siguientes colas:

- **RQ** (*Ready queue*): representa la cola de trabajos listos para ser ejecutados, de acuerdo a la planificación de eventos de la CPU, los trabajos pasan a la misma y luego a la cola de espera hasta que se termine la época actual. Hemos de hacer notar que en esta implemención los trabajos de tipo SRT nunca pasan a la cola de espera.
- **WQ** (*Wait queue*): contiene los trabajos que ya se han ejecutado y han de esperar a que termine la época para volver a la RQ. Cada vez que termina una época de la CPU esta cola es vaciada y todos los trabajos que hay en ella pasan a la RQ.
- **RMBQ** (*Receive Messages Buffer Queue*): los mensajes que arrivan al nodo son guardados en esta cola, a la espera de que la tarea a la cual han sido enviados entre en la CPU y los pueda procesar.
- **SMBQ** (*Send Messages Buffer Queue*): los mensajes que han de ser enviados desde este nodo son guardados en esta cola. En este implementación, los eventos de comunicación se revisan con cada evento de terminación de la CPU, por lo que en caso de generarse alguno ha de ser guadado a la espera de que la tarea que los generó entre en la CPU y los pueda procesar.

La entidad StationEntity conoce las clases que implementan las políticas de planificación de las tareas Best-effort y SRT. Como ya se mencionó anteriormente (sección 2.3.3), los tipos de políticas para tareas Best-effort son Round Robin y Coscheduling Cooperativo; y para SRT son RMS y EDF.

class ***CpuEntity***: representa la CPU de un nodo y encapsula toda la información relacionada con la misma. Los eventos *StartCpuEvent* y *ServiceEndEvent* son los que controlan la entrada y salida de trabajos a la CPU. El control de las épocas es realizado a través de la clase *CpuEpochHelper*, que se auxilia de las clases que implementan los algoritmos de planificación RT (RMS y EDF) para saber cuál es la próxima tarea en entrar en la CPU.

Las relaciones entre estas entidades se observan en la Figura 3.2. *StationEntity* mantiene referencias a las instancias de las clases *SchedulerHelper* y

CpuEntity conteniéndolas y haciendo posible que *CpuEntity* pueda acceder la información necesaria para la planificación de aplicaciones SRT de la instancia de la clase que hereda de *SchedulerHelper* que esté en uso en el modelo. La informanción del estado de la época es mantenida por *CpuEpochHelper* y de ella se auxilia *CpuEntity* para controlarlas. Todo el conocimiento del estado de la simulación es accesible desde el modelo, representado por una instancia de la clase *SimCluster*.

3.1.2.2. Entidades que modelan la carga de trabajo

class ***JobEntity***: Encapsula toda la información relacionada con una tarea. De acuerdo a nuestras necesitdades, una tarea puede ser local o paralela y para cada uno de estos tipos, podemos tener características SRT. Además de los campos necesarios para controlar estas características de la tarea, hemos de llevar el control de la cantidad de jiffies consumidos por todas las tareas en cada momento, para controlar su terminación.
Al ser cargado un fichero de entrada, marcamos la entidad de acuerdo a si es local Best-effort ($LOCAL$), local SRT ($LOCAL\_SRT$), paralela Best-effort ($PARALLEL$) o paralela SRT ($PARALLEL\_SRT$). Estas marcas definen cuales campos tendrán valores, la forma en que son tratados durante la planificación de la CPU, si revisan o no las colas de mensajes (RMBQ ó SMBQ) y otros comportamientos propios de cada tipo de tarea.

class ***ParallelAppEntity***: Contiene toda la información relacionada con una aplicación paralela. A partir de los valores que la caracterizan se construyen las tareas que la conforman en los diferentes nodos.

class ***MessageEntity***: Entidad que representa a un mensaje. Conoce su nodo origen y su nodo destino, además de la tarea que lo originó. Las caracterización de las aplicaciones paralelas son tomadas en cuenta para decidir si generan o no mensajes, pues se generan de acuerdo a sus necesidades de cómputo.

La interacción de las entidades mencionadas en esta subsección con las entidades "hardware" es mostrada en la Figura 3.3. En esta figura podemos apreciar que el conocimiento de las tareas es propio de los nodos y que en cambio información de más alto nivel se conoce desde la perspectiva del cluster. En el cluster se mantiene la información de las aplicaciones paralelas y los mensajes entre ellas, para que sea accesible a todas los nodos para su uso.

3.1.3. Eventos relevantes

Como se ha establecido anteriormente, los eventos controlan los cambios de estado interno, y han de estar asociados a una entidad. En esta sección

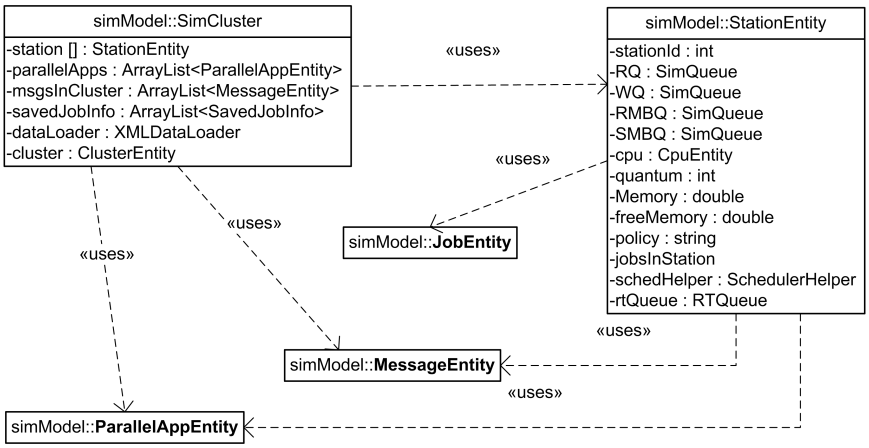

Figura 3.3: Interacción de las entidades que modelan la carga de trabajo con las entidades "hardware".

describimos los eventos más importantes incluidos en este modelo, el Algoritmo 3 muestra la interacción entre ellos. Hemos desglosado este algoritmo para facilitar su comprensión, siendo el Algoritmo 5 el que explica el funcionamiento de la generación de los mensajes y el Algoritmo 4 el que describe el procedimiento seguido al finalizar una tarea.

El Algoritmo 3 muestra la interacción general de los eventos, la primera acción a realizar es cargar los datos del fichero de entrada en formato XML, crear las instancias necesarias de *JobEntity* y *StationEntity* y generar el primer evento de StartCpuEvent para cada tarea (líneas 6 a 9). Luego para cada instancia de JobEntity presente en cada instancia de StationEntity generamos el primer evento StartCpuEvent. Posterior a esto, el control de los eventos para cada instancia de JobEntity relacionada con las instancias de StationEntity pasa a los dos eventos principales del modelo, StartCpuEvent y ServiceEndEvent. El primero realiza las acciones necesarias (línea 13), entre las cuales está la planificación del evento ServiceEndEvent asociado a su ejecución. Es durante la ejecución del evento ServiceEndEvent (líneas 14 a 23) que se chequea la ocurrencia de eventos de RT (líneas 18 a 20), comunicación (línea 22) o terminación de tarea (líneas 15, 16).

El Algoritmo 4 ocurre cuando una tarea consume todo su tiempo de cómputo (línea 15 del Algoritmo 3). En caso de ser una tarea local, no se hace nada (línea 12). En cambio si es una tarea paralela ha de incrementarse el contador de tareas de la respectiva aplicación paralela terminadas (línea 3). De acuerdo a si esto implica que todas las tareas de la aplicación paralela han terminado o no, se incrementa el contador de aplicaciones paralelas terminadas (línea 6) o no se hace nada. Si el contador de aplicaciones paralelas terminadas es igual a la cantidad de aplicaciones cargadas del fichero de entrada, se detiene

Algoritmo 3 Interacción general entre los eventos en el modelo

1: **StartCpuEvent** → Evento que ocurre cada vez que una tarea necesita la CPU, durante su ejecución dispara un evento **ServiceEndEvent**
2: **ServiceEndEvent** → Evento que ocurre cada vez que una tarea deja la CPU, implica la posible generación de: *Fin de Tarea*, *Condición RT* y *Comunicación*.
3: **JobArrivalEvent** → Ocurre cada vez que una tarea arriba al sistema, durante la carga de datos del fichero de entrada.
4: **CommunicationEvent** → Representa un evento de comunicación, implica que se revisen las colas locales de mensajes.
5: **RTEvent** → Representa un evento de RT.
6: **for all** $Job \in inputFile.xml$ **do**
7: Generar la correspondiente instancia de ($JobEntity \rightarrow job_{i,j}$) del Job_i en $station_j$
8: Planifica para $job_{i,j}$, su primer evento **StartCpuEvent** en $node_j$, en orden de prioridad para las tareas SRT y las Best-effort luego.
9: **end for**
10: **for** $j = 0$ **to** $simCluster.getCantStations()$ **do**
11: **for** $i = 0$ **to** $station[j].getCantJobs()$ **do**
12: **while** ($job_{i,j}.remCV > 0$) **do**
13: Durante el evento **StartCpuEvent**:
Planificación de un evento **ServiceEndEvent** en $t = t_{actual} + job_{i,j}.assignedQuantumSlice$
Chequeo de las colas de mensajes, $SMBQ$ y $RMBQ$ (envío y recepción de mensajes).
Control de Época de la CPU y del quantum a asignar a las tareas
14: Durante el evento **ServiceEndEvent**:
Cálculo del tiempo de cómputo restante para la tarea → $job_{i,j}.remCV$
15: **if** ($job_{i,j}.remCV == 0$) **then**
16: Ejecutar **Algoritmo 4** (Fin de Tarea)
17: **else**
18: **if** ($job_{i,j}.RT$) **then**
19: Generar evento **RTEvent** → $rtEvent$
20: Planificar $rtEvent$ para $t = t + job.getPeriod()$
21: **end if**
22: Ejecutar **Algoritmo 5** (Generación de Evento de Comunicación)
23: **end if**
24: **end while**
25: **end for**
26: **end for**

la simulación (línea 8), proceso que implica la generación del fichero de salida en formato XML .

Algoritmo 4 Fin de Tarea

```
1:  La instancia de JobEntity, job ha consumido todo su volumen de cómputo
2:  if ((job.type == PARALLEL) or (job.type == PARALLEL_SRT)) then
3:     Incrementar el contador de tareas terminadas (finishedJobs) de la
       aplicación paralela correspondiente.
4:     if (finishedJobs == parallelAppJobsCounter) then
5:        Guardar la información de la aplicación paralela para generar el
          fichero de salida (outFile.xml)
6:        Incrementar el contador de tareas paralelas terminadas finishedPa-
          rallelApps
7:        if (finishedParallelApps == loadedParallelApps) then
8:           Condición de Finalización alcanzada, detener la simulación y ge-
             nerar el fichero de salida (outFile.xml)
9:        end if
10:    end if
11: else
12:    Fin de Tarea Local → Se descarta su información
13: end if
```

Finalmente, el Algoritmo 5 describe el proceso de la generación de un evento de comunicación. Cabe mencionar que es un requerimiento para la ejecución de este algoritmo que la tarea sea de tipo paralelo, ya sea SRT o no. Si a tarea es paralela, procedemos a muestrear una distribución booleana de Bernoulli creada de acuerdo a la relación cómputo/comunicación de la apliacación paralela. Si el valor devuelto es verdadero, se pone en marcha el proceso de crear mensajes en los otros nodos del cluster que tienen tareas de esta aplicación paralela. Han de mencionarse que esto es solo el inicio del proceso de comunicación, que consta de más partes. Luego de ser introducidos estos mensajes en las colas de mensajes del nodo, se procesan cuando la tarea paralela recibe la CPU y se envían a los nodos a los cuales están destinados. El proceso de comunicación concluye cuando las respectivas tareas de los otros nodos con tareas cooperantes revisan sus colas de mensajes recibidos, los procesan y envían las respuestas.

class ***StartCpuEvent***: Evento que gestiona la inserción de las tareas en la CPU, calculando el quantum que le corresponde en caso de ser necesario. Conjuntamente con el evento ServiceEndEvent controlan la entrada y salida de las tareas en la CPU. En caso de tener tareas SRT, asigna sus slices del quantum de la CPU de acuerdo a sus requerimientos, el slice disponible del quantum se emplea en las aplicaciones Best-effort.
Al acceder una tarea paralela la CPU, revisa las colas de mensajes para saber

si han ocurrido eventos de comunicación. En caso de ser así, se gestionan en ese momento.

class ***ServiceEndEvent***: cada vez que ocurre un evento StartCpuEvent, se genera un evento de este tipo, que se encarga de expulsar la tarea de la CPU. Al ocurrir este evento se actualizan todas las variables que controlan las cantidades de CPU recibidas por cada tarea, en caso de terminar la tarea, se trata de acuerdo a si es local o paralela. En caso de ser local, nada ocurre, solo se guardan sus datos y se borra de las colas del nodo. Por otro lado, en caso de ser paralela también se quita de las colas del nodo y además se actualiza el contador de tareas en el trabajo paralelo. Si este contador vale 0, se notifica el fin del trabajo y se guarda hasta la terminación de la simulación.
Los datos de las aplicaciones paralelas conforman un fichero de salida en formato XML, que recibe LoRaS y constituyen los valores de $RExT$ generados por Simulador_Cluster_SRT.
En este evento se gestionan los eventos de tiempo real, que es descrito a continuación.

class ***RealTimeEvent***: representa a un evento RT, es planificado solo en dos situaciones:
1− Al arribar una tarea con características SRT al sistema.
2− Cuando expulsamos alguna tarea SRT de la CPU.
Al ocurrir, desaloja (si no es SRT) el trabajo que se encuentra en la CPU y se planifica un evento StartCpuEvent.

Algoritmo 5 Generación de Evento de Comunicación

Require: ($job.type$ == PARALLEL) or ($job.type$ == PARALLEL_SRT)
1: Muestrear la distribución de Bernoulli (creada de acuerdo a la relación cómputo/comunicación) de la tarea paralela para decidir si comunica o no → $generateCommEvent$
2: **if** ($generateCommEvent$) **then**
3: **for** ($i = 0$ to $simCluster.stationCounter$) **do**
4: **if** (**stationHasJob**(job, i) and ($i \neq station$.**getStationId()**)) **then**
5: **create new** $MessageEntity \rightarrow msg$
6: Adicionar msg a la cola de mensajes en el cluster
7: $station$.**SMBQ.insert**(msg);
8: **end if**
9: **end for**
10: **end if**

class ***JobArrivalEvent***: Debido a que los datos de los trabajos son cargados de ficheros de entrada, el peso de los datos recae en las tareas. Para cada nodo recibimos una lista de las tareas presentes en él, de cualquier tipo y

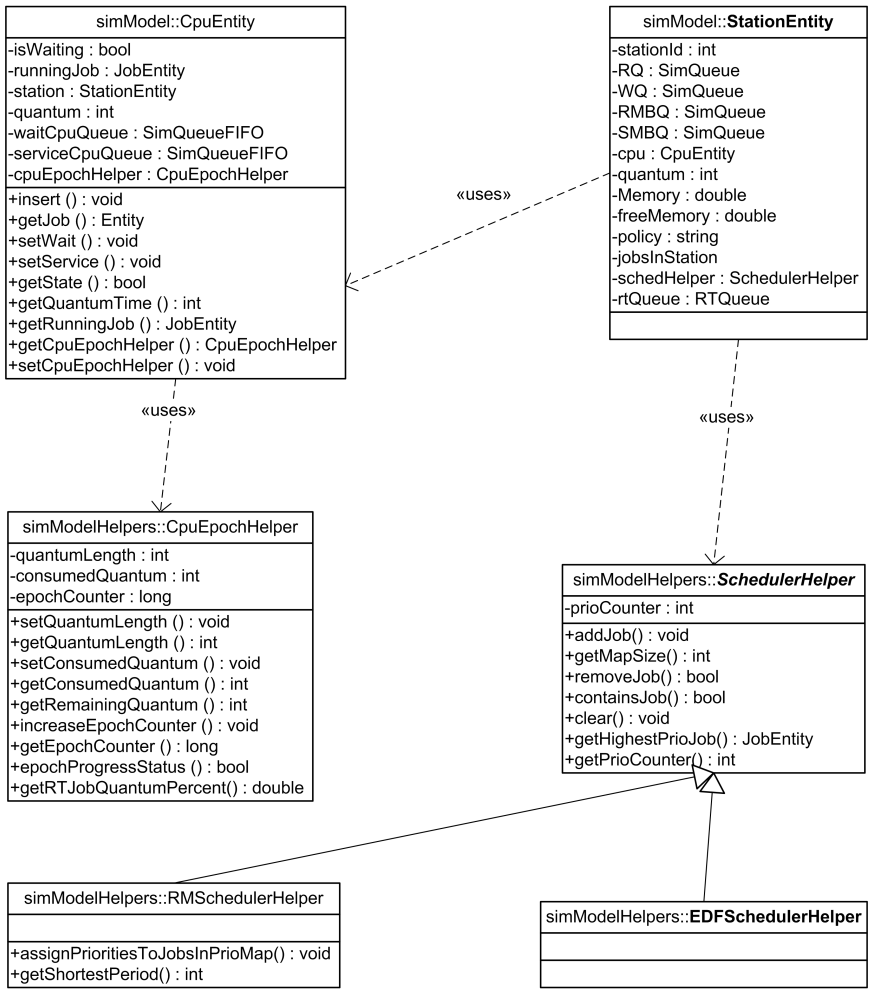

Figura 3.4: Interacción de las clases, soporte para adición de algoritmos de planificación RT.

con cualesquiera características (SRT o Best-effort). A partir de esta lista, se cargan los datos y el modelo se inicializa con ellos, este evento gestiona la colocación de cada tarea en el nodo que le corresponde.

class ***CommunicationEvent***: Representa un evento de comunicación, al ocurrir, se revisan las colas de mensajes del nodo y los mensajes pertenecientes a la tarea se procesan.

3.2. Soporte para nuevos algoritmos RT

Para facilitar la experimentación de nuevos algoritmos de planificación RT, se diseñó el grupo de clases que conforman esta parte del software de forma

modular. Esta implementación, permite añadir nuevos algoritmos heredando de una clase abstracta, nombrada *SchedulerHelper* e incluida en el paquete *simModelHelpers*. La Figura 3.4 muestra las clases que representa a los algoritmos inicialmente incluidos en la jerarquía (*RMSchedulerHelper* y *EDFSchedulerHelper*) y la interacción entre las clases necesarias para realizar la planificación.

La clase *StationEntity* es el contenedor principal de toda la información de las tareas que se encuentran en ella y la forma de planificarlas. Por esta razón guarda referencias a los *helpers* para la planificación usando algorimos RT y a la CPU. La clase *CpuEntiy* se apoya de la información brindada por la clase *CpuEpochHelper* (también definida en el paquete *simModelHelpers*) a la hora de controlar sus épocas. Esta a su vez, se nutre de la información que brindan las clases mostradas en la Figura 3.4 (*RMSSchedulerHelper* o *EDFSchedulerHelper*) para decidir datos relacionados con el quantum o cual es el la próxima tarea RT en entrar en la CPU. Implementando esta parte del software de esta forma logramos una mayor extensibilidad y flexibilidad en el código, que se traduce en ahorro de tiempo y esfuerzo para desarrollos futuros.

3.3. Comunicación entre los procesos

La comunicación entre LoRaS en modo de simulación off-line y Simulador_Cluster_SRT ocurre a través de ficheros en formato XML. Como ya se ha explicado anteriormente, la simulación funciona a dos niveles, en el superior, LoRaS genera ficheros XML con el estado del cluster y realiza la ejecución de Simulador_Cluster_SRT. Una vez leídos de los ficheros XML, los datos generados por LoRaS son guardados en clases (Figura 3.5), y posteriormente empleados para generar los eventos de arribo de tareas. Cabe destacar que, al igual que con el soporte de nuevos algoritmos RT, la modularidad del código implica mayor facilidad a la hora de extender las funcionalidades a nuevos tipos de tareas.

Las razones por las que elegimos XML como formato para los ficheros de datos son:

- La existencia de APIs que facilitan su uso en los lenguajes de programación implicados en el desarrollo.
- Permite comprobar la validez y consistencia de los ficheros de datos de forma rápida y segura. Para cada fichero XML podemos establecer su fichero de formato, contra el cual podemos validarlo y comprobar su consistencia, de acuerdo a la forma en la que lo definimos.

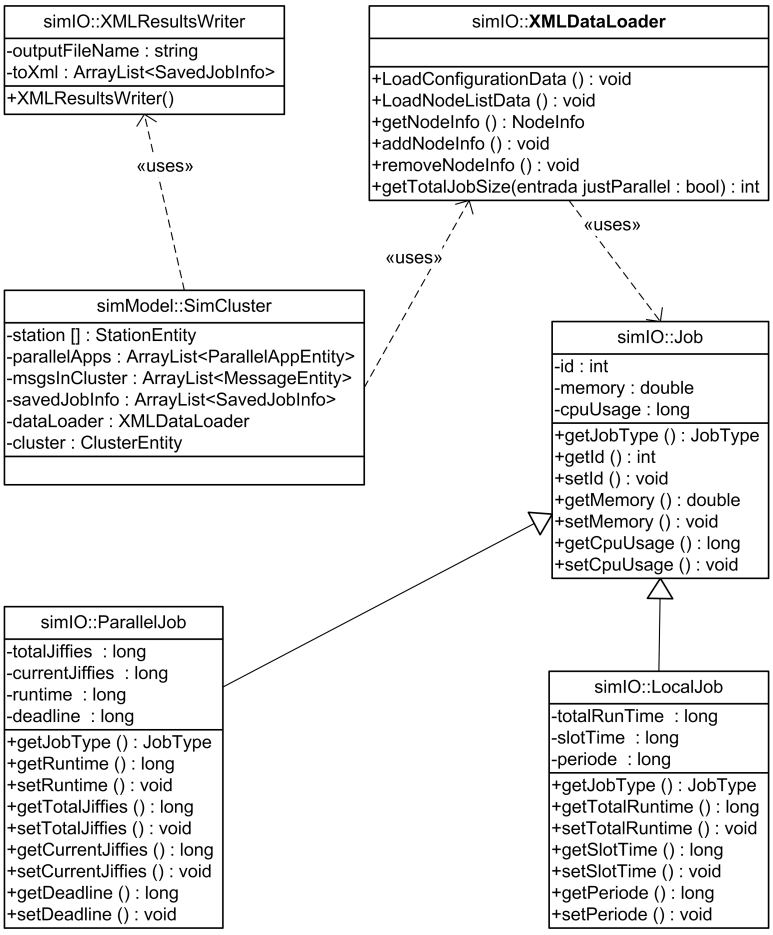

Figura 3.5: Jerarquía de manejo de datos y su interacción general.

Por las razones antes expuestas, ahorra tiempo de desarrollo y hace más claros y legibles los ficheros de intercambio de datos. Es también un estándar ampliamente utilizado.

Capítulo 4

Experimentación realizada y resultados obtenidos

En este capítulo mostramos la experimentación realizada, principalmente dirigida a validar el sistema, comprobar su escalabilidad y observar el desempeño de nuestros métodos de estimación con capacidad SRT ante cargas locales o paralelas SRT.

4.1. Caracterización de los entornos de ejecución

Antes de pasar a mostrar y comentar los resultados alcanzados, es necesario describir el entorno utilizado para realizar las ejecuciones. Cabe destacar que se han realizado ejecuciones reales, i.e.: con carga de tipo Best-effort, para todos los métodos descritos en el capítulo 2. La caracterización de este entorno de ejecución es mostrado a continuación (subsección 4.1.1), que incluye la forma de representar la carga paralela y las aplicaciones paralelas junto con sus tiempos de llegada al sistema.

Ha de mencionarse que tanto las ejecuciones reales como las simuladas fueron realizadas empleando la política FCFS (*First Come First Serve*) para la selección de trabajos y la política *Normal* para la selección de los nodos. Destacamos que la política *Normal* intenta seleccionar el mejor conjunto de nodos para ejecutar una aplicación paralela teniendo en cuenta el uso de recursos en los nodos del cluster. De esta forma esta política no sobrecarga los nodos en detrimento de la carga local que pueda estar presente en ellos.

4.1.1. Entorno de las ejecuciones reales

En el caso de las ejecuciones reales, es necesario simular la presencia de usuarios locales y además, aplicaciones paralelas que lleguen al sistema en

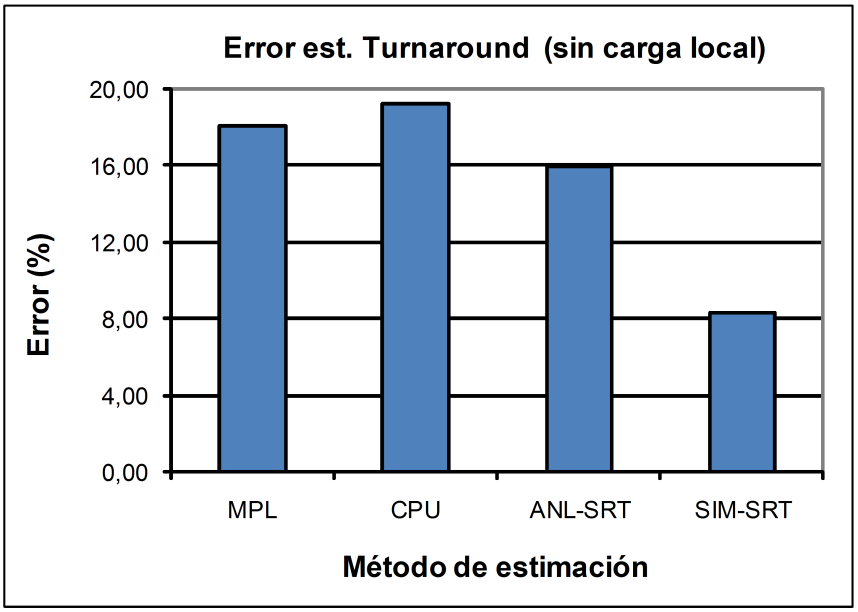

Figura 4.1: Validación parcial de los métodos SRT, contra ejecuciones reales sin carga local.

intervalos de tiempo representativos. Por estas razones, las aplicaciones han de estar correctamente caracterizadas, para garantizar luego que las comparaciones con los métodos de estimación empleados en las simulaciones (analíticos o simulado) sean justas.

La actividad local Best-effort es modelada por un benchmark parametrizable. Los valores de CPU, memoria y uso de red de las aplicaciones son dados como parámetros a este benchmark y el simula el uso de estos recursos. Para conseguir una mayor similitud con valores reales, se realizaron medidas en laboratorios y con los valores obtenidos creamos los parámetros modelo para aplicaciones Best-effort. En este caso, los valores para el benchmark son 15 % de CPU, 35 % de memoria y 0.5 KB/s como uso de red.

La carga paralela está representada por una lista de aplicaciones NAS ejecutadas usando PVM, que emplean 2, 4 y 8 nodos y llegan al sistema siguiendo una distribución de Poisson. Estas aplicaciones han sido mezcladas de tal forma que sea balanceada en cuanto a cómputo y comunicación. La Tabla 4.1 muestra la caracterización de las aplicaciones paralelas utilizadas.

El cluster donde se llevaron a cabo las ejecuciones estaba compuesto de 8 nodos Pentium-IV (1.8 GHz), con 512 MB de memoria RAM e interconectados por una red Fast-Ethernet. El sistema operativo instalado en estos nodos es Linux. Las simulaciones son realizadas de acuerdo a la caracterización de estos nodos.

	NAS-IS (CPU (%) - Mem. (MB) - Tiempo (seg.))								
	Clase A			**Clase AB**			**Clase B**		
Nodos	CPU	Mem.	Tiempo	CPU	Mem.	Tiempo	CPU	Mem.	Tiempo
2	44	112	50	57	220	99	58	380	240
4	29	72	49	26	136	109	25	260	240
8	26	44	39	25	88	58	24	150	179
	NAS-MG (CPU (%) - Mem. (MB) - Tiempo (seg.))								
	Clase A			**Clase AB**			**Clase B**		
Nodos	CPU	Mem.	Tiempo	CPU	Mem.	Tiempo	CPU	Mem.	Tiempo
2	72	220	49	86	220	129	90	220	209
4	57	113	29	83	113	65	78	113	119
8	36	60	19	62	60	49	70	60	75

Tabla 4.1: Caracterización de las aplicaciones paralelas para el proceso de simulación.

4.1.2. Entorno de las ejecuciones simuladas

Para el caso de la actividad local de tipo SRT empleamos los resultados mostrados en [26]. En este estudio se muestran varias aplicaciones con características SRT, entre las cuales seleccionamos una para nuestro estudio, el visualizador multimedia Xine. Esta aplicación local se caracteriza por presentar diferentes niveles en el uso de recursos de acuerdo al tamaño de la ventana de visualización, siendo los consumos de recursos representados como 11 % de CPU y 15 % de memoria para visualización de $1x$ y 41 % de CPU y 20 % de memoria para visualización a $2x$ (donde x representa el tamaño de la ventana de visualización).

Al no contar con aplicaciones paralelas SRT con la caracterización necesaria por el simulador Off-line de LoRaS, hemos tenido que construirnos la carga paralela SRT. En este enfoque inicial, las aplicaciones paralelas SRT son aplicaciones paralelas Best-effort a las que se les definimos un deadline o tiempo de finalización máximo. Cabe destacar que este deadline lo calculamos sumándole al tiempo de ejecución obtenido mediante la ejecución de la aplicación paralela isolada un 20 % del valor obtenido (Ecuación 4.1).

$$deadline(j) = turnaround_{isol}(j) + \frac{1}{5} \times turnaround_{isol}(j) \tag{4.1}$$

Al construir la carga paralela SRT de esta forma, podemos reusar las caracterizaciones disponibles de trabajos previos. Cabe destacar que debido a que aún no disponemos de soporte en el sistema de ejecuciones reales para aplicaciones SRT, ya sean paralelas o locales, esta era una de las pocas opciones viables. Creemos que la pérdida de generalidad introducida esta asunción es permisible a este nivel de nuestro trabajo.

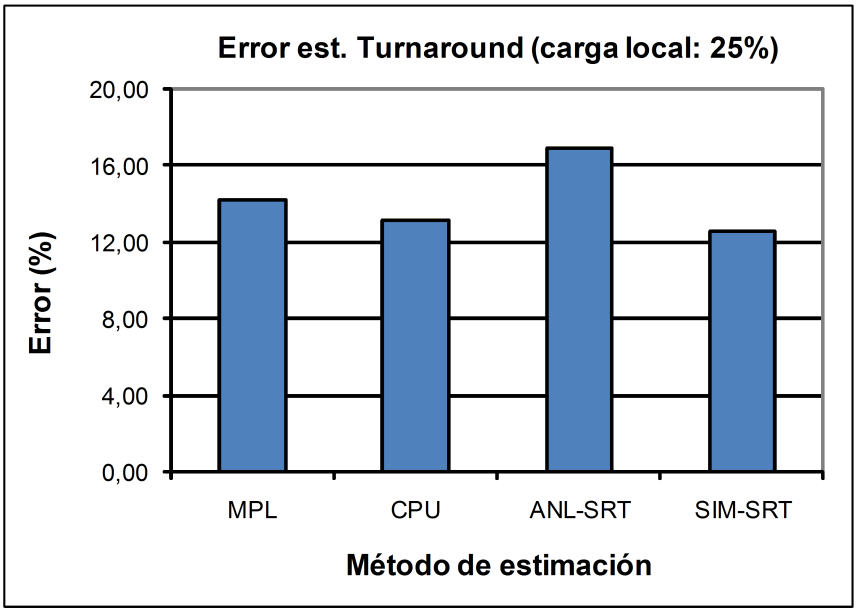

Figura 4.2: Validación parcial de los métodos SRT, contra ejecuciones reales con un 25 % de carga local.

Para las ejecuciones simuladas se utilizan caracterizaciones de los nodos (Pentium-IV a 1.8 GHz con 512 MB de memoria RAM) empleados para las ejecuciones reales.

4.2. Validación del Simulador

Debido a que el entorno CISNE aún no es capaz de ejecutar carga con características SRT, la validación posible en este momento del trabajo es *parcial*. Es decir, podemos comparar los resultados de nuestros métodos de estimación con resultados de ejecuciones reales solo para carga Best-effort. Por otro lado cabe destacar que gran parte de los estudios consultados en la literatura [42, 12, 84, 88] usan la simulación como único método de trabajo, siendo la excepción mayoritariamente los trabajos que hacen modificaciones a los SO para mejorar sus capacidades SRT [19, 14, 66, 2]. Para este caso, la cantidad de aplicaciones NAS ejecutadas fue de 30.

Las Figuras 4.1, 4.2 y 4.3 muestran la precisión que nuestros métodos pueden alcanzar para diferentes niveles de carga local. En este experimento tanto la carga local como la paralela son de tipo Best-effort, debido a que son comparados con métodos incapaces de procesar carga (paralela o local) de timpo SRT. Para realizar esta experimentación, fueron empleados 8 nodos para ejecutar tanto la carga local como la paralela, variando los niveles de

presencia de usuario local (en valores de 0 %, 25 % y 50 %) en los diferentes casos. La diferencia entre los valores obtenidos de las ejecuciones reales y las simuladas por los diferentes métodos son mostradas aquí en la forma de tanto porciento de error en el turnaround.

En esta figura podemos apreciar que el método simulado (SIM_SRT, correspondiente a $RExT_{SIM\_SRT}$) se comporta siempre mejor que los demás métodos, como era de esperar dado su mayor nivel de detalle. Los valores de error máximos en este tipo de experimento para SIM_SRT son siempre menores al 12 %. Cabe destacar que los métodos denotados por MPL y CPU han sido contrastados contra otros métodos de estimación presentes en la literatura, brindado al menos resultados tan buenos como ellos.

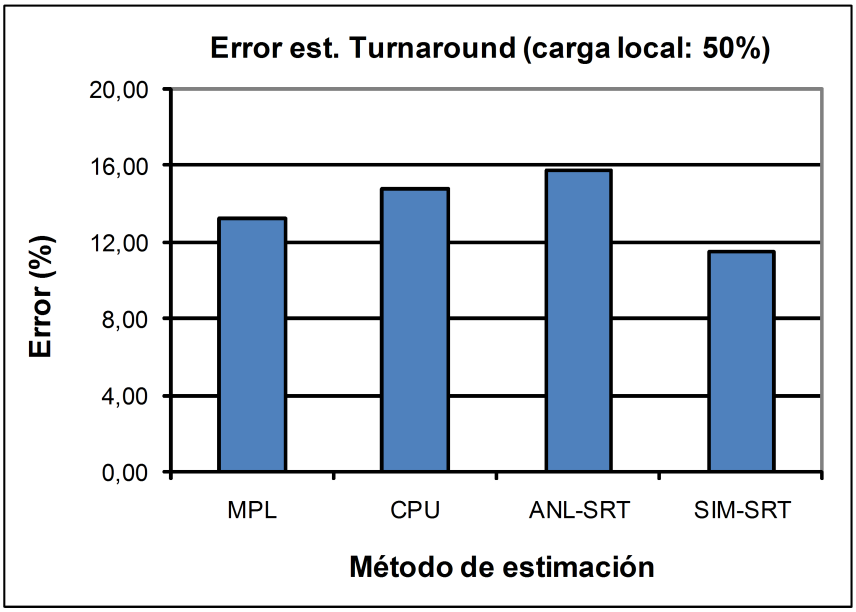

Figura 4.3: Validación parcial de los métodos SRT, contra ejecuciones reales con un 50 % de carga local.

4.3. Inclusión de carga SRT

La inclusión de carga con características SRT en la experimentación implica que solo podemos emplear los métodos de estimación del $RExT$ capaces de trabajar con este tipo de tareas, que en nuestro caso son *ANL-SRT* y *SIM-SRT*. Estudiaremos primero el efecto de la carga local SRT sobre las aplicaciones paralelas Best-effort y luego la coexistencia de varios tipos de cargas en un cluster no dedicado.

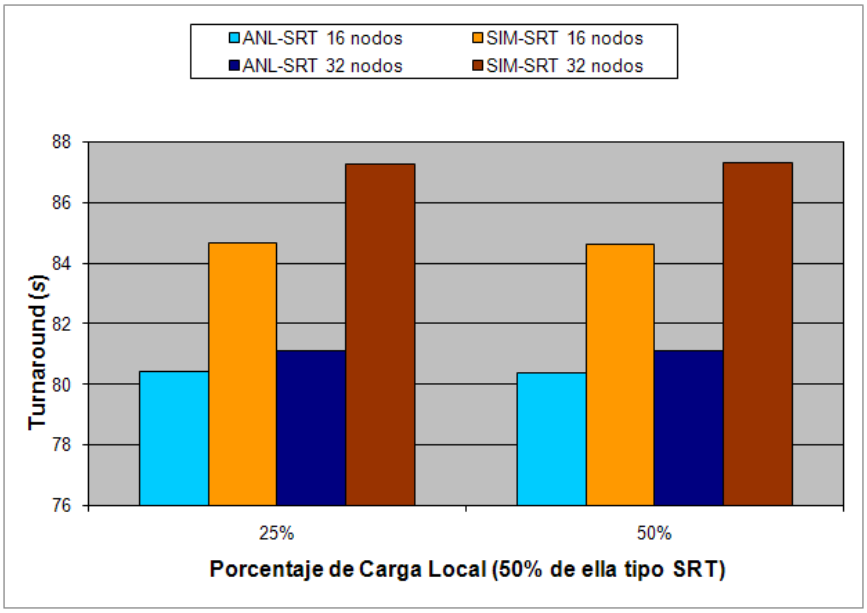

Figura 4.4: Comparación de métodos en presencia de carga local SRT, turnaround para 16 y 32 nodos.

4.3.1. Carga local SRT

Este experimento compara los dos métodos de estimación del $RExT$ en presencia de carga local Best-effor y SRT. Para generar la gráfica mostrada en la Figura 4.4 realizamos experimentos para 16 y 32 nodos, con diferentes niveles de presencia de usuario local. Es importante destacar que para cada porciento de carga local, la mitad es de tipo SRT.

Creemos que esta experimento también nos permite observar el efecto de la carga local de tipo SRT en el turnaround (tiempo que espera el usuario paralelo hasta que concluye la ejecución de la aplicación paralela que ha lanzado) de las aplicaciones paralelas. La Figura 4.4 permite apreciar que ambos métodos siguen la misma tendencia, lo cual es un resultado alentador, pues nuestros dos métodos de estimación se comportan parecido. Cabe destacar que el método simulado genera resultados con tiempos de turnaround mayores que el análitico, consideramos que esto ocurre debido a su mayor nivel de detalle, lo cual refleja las mayores exigencias de recursos del usuario local con más presición.

4.3.2. Carga paralela SRT

En este experimento, el 15 % porciento de la carga paralela es de tipo SRT, la presencia de usuario local es de tipo Best-effort y los experimentos fueron

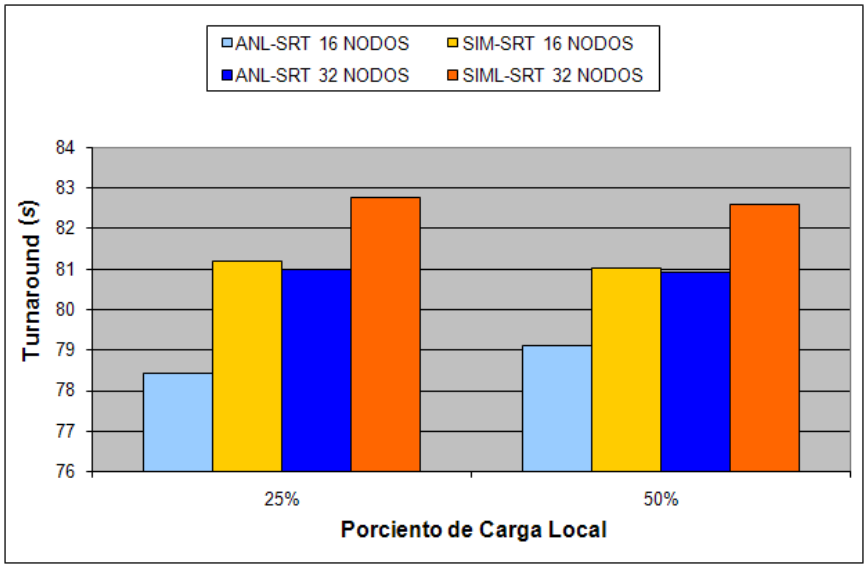

Figura 4.5: Comparación de métodos en presencia de carga local Best-effort, turnaround para 16 y 32 nodos. Aplicaciones paralelas tipo SRT (15 % del total).

hechos para 16 y 32 nodos. Como podemos apreciar en la Figura 4.5, los tiempos de ejecución de las tareas paralelas disminuyen. Esto es un resultado lógico y esperado, pues ahora las aplicaciones paralelas disponen de más tiempo de CPU, debido a la disminución de los requerimientos de CPU y memoria de las aplicaciones locales.

Capítulo 5

Conclusiones y Trabajo Futuro

En este capítulo enunciamos las conclusiones alcanzadas y las líneas de trabajo futuro.

5.1. Conclusiones

Nuestro trabajo incluye elementos de planificación temporal en clusters no dedicados y sistemas tiempo real débil. La unión de estas dos líneas se hace necesaria para reflejar los cambios ocurridos en las aplicaciones a ejecutar en clusters no dedicados, tanto las locales como las paralelas. Nuevos tipos de aplicaciones locales SRT, cuyo mejor ejemplo son las aplicaciones multimedia, implican una redefinición de las pautas que garantizan su coexistencia con la carga paralela. Las aplicaciones paralelas también evolucionan, requiriendo en muchos casos QoS para una correcta ejecución.

Creemos que las aulas de ordenadores presentes en cualquier universidad hoy en día son una fuente de poder de cómputo de la que muchos sistemas intentan hace uso eficiente. Un enfoque como el nuestro, que hace coexistir la carga paralela y la local implica un mejor uso de estos recursos. El cambio en las aplicaciones locales y paralelas anteriormente mencionado implica crear nuevos esquemas y métodos de planificación para que los usuarios locales no vean afectada la interactividad de sus ordenadores.

El principal objetivo de este trabajo es proveer un sistema que permite estudiar la ***planificación temporal*** de aplicaciones de varios tipos que han aparecido en la literatura. Para este fin, se ha modificado un sistema diseñado para estudiar la planificación espacial de aplicaciones paralelas, que contempla dos tipos de aplicaciones, locales y paralelas Best-effort. Se ha añadido ***soporte para aplicaciones paralelas y locales SRT***, en dos métodos de estimación, uno analítico y otro simulado.

Se ha realizado una ***revisión bibliográfica*** dirigida a la consideración de aplicaciones SRT, tanto locales como paralelas, en clusters no dedicados. Cabe destacar que aún cuando es un tema ampliamente estudiado, no hemos encontrado trabajos donde se aborde la problemática de la planificación espacial en clusters no dedicados de aplicaciones locales y paralelas de los tipos que contempla este trabajo, Best-effort y SRT.

Se han ***creado dos nuevos métodos de estimación con capacidad SRT***, que permiten estudiar el comportamiento de aplicaciones con estas características en clusters no dedicados. Este trabajo ha significado una serie de extensiones y modificaciones a sistemas previamente desarrollados en el grupo.

El método de estimación por simualción es un simulador independiente, que podría ser extendido para realizar simulaciones sin necesidad de otros sistemas. Su implementación modular permite incorporar nuevos algoritmos de planificación con un esfuerzo razonable. Este método de estudio de la planificación temporal de aplicaciones paralelas y locales, tanto Best-effort como SRT ha dado origen a la publicación:

- J.García, P. Hernández, J. Lérida, F.Giné, F.Solsona & M. Hanzich. ***Using Simulation for Job Scheduling with Best-Effort and Soft Real-Time Applications on NOWs.*** XVIII Jornadas de Paralelismo, JP'2007.

La experimentación desarrollada hasta este momento refleja la necesidad de combinar las métricas existentes con otras que nos permitan evaluar los algoritmos de planificación SRT que incluyamos en nuestro sistema. Es importante mencionar que es necesaria una *redefinición de la carga paralela*, para reflejar de manera fideligna nuestro entorno de estudio. Creemos que en esta nueva situación, las colas de espera han de ser cortas para poder satisfacer los requerimientos de las aplicaciones paralelas SRT.

5.2. Trabajo Futuro

En este tipo de estudios, la caracterización de la carga, ya sea local o paralela, es de primordial importancia. Además, los resultados alcanzados demuestran que hemos de prestar especial importancia a las aplicaciones paralelas, de tal forma que reflejen nuestro entorno de una forma más específica. Por lo tanto una de nuestras líneas abiertas más importantes es el encontrar aplicaciones paralelas, tanto SRT como Best-effort, que encajen mejor en nuestro entorno.

Aún cuando incluimos un modelo analítico en nuestro trabajo, creemos que este es susceptible a mejoras, siendo esta dirección de trabajo también importante en nuestro futuro. Ha de trabajarse también en el desarrollo de nuevas

métricas y su combinación con las ya existentes, pues al combinar dos direcciones de investigación (planificación temporal en clusters no dedicados y planificación aplicaciones SRT), han de proveerse medios para evaluar los resultados en las dos direcciones. Creemos que el desarrollo de modelos analíticos para la predicción relacionada con las métricas para aplicaciones SRT es también importante.

Las nuevas métricas a contemplar en los modelos han de ser capaces de evaluar el comportamiento de los algoritmos de planificación estudiados, en base a las pérdidas de deadlines en las planificaciones. Hasta ahora solo somos capaces de medir deadlines de aplicaciones paralelas SRT, hemos de ser capaces de medir los deadlines que pierde cada tarea SRT, ya sea paralela o local. También hemos de desarrolla métodos analíticos para predecir las pérdidas de deadlines de tareas SRT.

Finalmente, el hecho de que los procesadores multicore sean fácilmente accesibles en el mundo comercial nos hace plantearnos con fuerza su inclusión en nuestra línea de trabajo. Cabe mencionar que se ha hecho un esfuerzo considerable de investigación en la literatura y ya se perfilan algunas de las estrategias para su uso. Los resultados alcanzados hasta el momento en esta línea de trabajo están incluidos en forma de anexos (Anexos A y B). Cabe destacar los procesadores multicore es un tema en auge en la actualidad, reflejado en los recientes artículos sobre el tema [32, 6, 7, 18].

También incluidos en el apéndice B tenemos los estudios de rendimiento relacionados con el uso de procesadores Pentium D para la ejecución de aplicaciones PVM NAS.

Apéndice A

Propuestas para la planificación

Hemos de destacar que nuestra propuesta para la planificación ha de tener en cuenta el hardware en el cual se implementará y las necesidades de los tipos de aplicación a planificar. Los detalles del hardware disponible están en el Apéndice B.

A.1. Implementación del planificador en espacio de usuario

Es un requerimiento importante de diseño que no se haga ningún tipo de modificación en el kernel de Linux. Esto nos permite una mayor portabilidad de nuestro sistema, además de facilitar enormemente su implantación y mantenimiento. Un grupo de pruebas y reflexiones que incluimos en esta subsección nos llevaron a nuestra opción de implementación del planificador.

Queremos destacar que la opción que parece más evidente para la implementación de un planificador en Linux, es decir, el uso de las colas RT incluidas en las últimas versiones, ha quedado descartada en este trabajo. La principal razón por la cual descartamos las colas RT es porque la única forma de ejecutar una aplicación RT antes que otra es que esté en una cola de mayor prioridad. El planificador tendría que utilizar una llamada al sistema (*sched_setscheduler*) cada vez que hiciese falta otorgarle la CPU a alguna aplicación, con el agravante de que no todas las aplicaciones son RT . Este planificador resultaría más complejo que la propuesta que presentaremos más tarde.

A.1.1. Planificador basado en renice

Las primeras pruebas realizadas giraron en torno al uso de la función *renice* de Linux, la cual permite cambiar la prioridad estática de los procesos de forma dinámica. Este método tiene como ventajas que:

- Genera poca intrusión, dado que la modificación del valor *nice* de los procesos no influye en el rendimiento del planificador y una vez asignado se utiliza durante largos períodos de tiempo.
- El quantum asignado a cada proceso es calculado de forma precisa a partir del valor del nice (empleando el Algoritmo 6). Esta forma de calcular es la empleada en el kernel 2.16.x.

Algoritmo 6 Cálculo de la prioridad a partir del *nice*

1: $scale\_prio(\text{X}, prio) \leftarrow max(\text{X} \times \frac{(\text{MAX\_PRIO} - prio)}{(\text{MAX\_USER\_PRIO/2})}, \text{MIN\_TIMESLICE})$
2: **if** $(\text{STATIC\_PRIO} < nice\_to\_prio(0))$ **then**
3: $\quad return\ scale\_prio(\text{DEF\_TIMESLICE} \times 4, \text{STATIC\_PRIO})$
4: **else**
5: $\quad return\ scale\_prio(\text{DEF\_TIMESLICE}, \text{STATIC\_PRIO})$
6: **end if**

Y como desventajas:

- No podemos controlar exactamente que proceso se ejecutará, pues al consumir todo su quantum los procesos salen de la cola de procesos listos y no se ejecutan hasta que termina la época.
- Los valores de nice negativos generan problemas con la interactividad del ordenador.

A.1.2. Planificador basado en STOP-CONT

Esta forma de planificar las aplicaciones se basa en el envío de señales de STOP/CONT para detener y reanudar los procesos. De esta forma logramos controlar el momento concreto del tiempo en el cual un proceso inicia y se detiene, además de la cantidad de cómputo que recibe.

Este método tiene una única desventaja, el hecho de genera más intrusión que el método del renice antes descrito. Esto es debido a la ejecución periódica del planificador para controlar las aplicaciones en ejecución, aunque es válido destacar que en las pruebas realizadas no logramos medir la intrusión generada por este tipo de planificador.

Las ventajas de este método son:

- Al comparar el nivel de intrusión generado por este método lo comparamos con el anterior, por lo que vale la pena destacar que en comparación al renice, es mucho más fiable y ajustable para las tareas de planificación a realizar.
- Proporciona una mayor resolución en la planificación (en el orden de los μseg) que el renice (resolución máxima en el orden de los jiffies $\approx 10\ mseg$).
- La planificación es mucho más fácil ya que podemos enviar señales a grupos de procesos (padre y todos sus hijos).

A.2. Planificador propuesto

Finalmente en esta sección enunciamos nuestra propuesta de planificación, que consiste en combinar los métodos expuestos. Esta combinación aporta lo bueno del método del renice (intrusión extremadamente baja) y lo positivo del método del STOP-CONT (su alta resolución) justo donde más falta hace. Podemos implementar un planificador haciendo uso del método del STOP-CONT para las aplicaciones SRT, que son las que necesitan de una alta resolución temporal para su correcta ejecución. Y luego, las aplicaciones tipo Best-effort son planificadas utilizando el método del renice, lo cual genera poca intrusión en el sistema y no afecta las aplicaciones SRT.

Tenemos a nuestro favor que a partir del kernel de Linux 2.6.18 se incluyen mejoras relacionadas con el tiempo real. Estas mejoras están principalmente relacionadas con una mayor resolución del temporizador y mejoras en la capacidad de desalojo de algunos segmentos del kernel. Es válido destacar que aún queda mucho por hacer en estos dos sentidos.

Apéndice B

Procesadores multicore

Los procesadores multicore, que son ya una realidad hoy en día, comienzan a ser objeto de muchos estudios. Entre los trabajos que se enfocan en tomar ventaja de las posibilidades que abren al emplear procesadores multicore encontramos [32, 6, 7, 18]. En este apéndice mostramos el estado de arte relacionado con los procesadores multicore, la forma en la que pensamos tomar ventajas de ellos y algunos experimentos realizados.

B.1. Estado del arte procesadores multicore

La planificación de aplicaciones paralelas SRT en procesadores multicore es un tema de reciente interés en la comunidad científica. Las ventajas que pueda aportar la coplanificación en un entorno multicore no son evidentes y han de ser estudiadas detalladamente. Además de la planificación, otros temas de relevancia en esta área están asociados al nuevo cuello de botella existente en estas arquitecturas, la memoria principal y las caches.

En [32] se aborda el problema desde una óptica introductoria, mostrando el panorama general de hardware y las limitaciones de los sistemas operativos (SO) comerciales al enfrentar este problema. Las conclusiones más importantes de este estudio son la necesidad de desarrollar aplicaciones capaces de aprovechar el paralelismo existente en el hardware y que los planificadores de los SO sean más competentes a la hora de aprovechar las capacidades multicore de los procesadores. Sus recomendaciones pasan por tres puntos principales, la **clasificación** de los procesos, la capacidad de **adaptarse** y **detectar** su tipo de forma **automática** y la necesidad de incrementar la ***cooperación*** entre los procesos.

La posibilidad de que los diferentes cores de un procesador multicore sean usados para diferentes tipos de tareas es introducida en [18]. En este estudio,

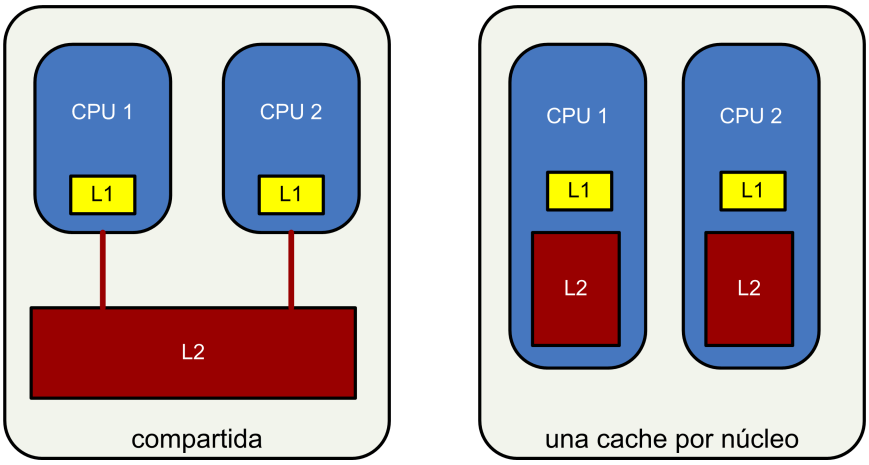

Figura B.1: Ejemplos de tipos de cache en procesadores dual-core.

los cores que conforman el procesador multicore tienen diferente potencia, por lo que se denomina a esta plataforma **asimétrica**. Proponen la planificación de las tareas best-effort y SRT en los diferentes cores del procesador intentando mejorar el rendimiento de las aplicaciones best-effort con una estrategia basada en servidores diferidos DS (*Deferrable Servers*) [79]. El algoritmo propuesto en DS está basado en la premisa de aumentar las prioridades a las tareas aperiódicas mientras las tareas periódicas no pierdan sus deadlines.

Algunos estudios han intentado responder las incógnitas relacionadas con la *memoria principal* y la *cache* en procesadores multicore. Ha de señalarse que todos los estudios realizados se basan en la premisa de la cache de nivel 2 (L2) es ***compartida*** (Figura B.1.izquierda) entre los cores del multiprocesador. Estudios centrados en la planificación de tareas RT [7] y paralelas [6] en plataformas multicore intentan tomar ventaja de esta característica de algunos procesadores multicore. En estas circunstancias los cambios de contexto no son costosos cuando el *working set* de la aplicación está en cache y de esta característica de los sistemas multicore con cache compartida sacan ventaja en estos trabajos.

La influencia del ancho de banda de la memoria cuando se planifican aplicaciones RT o SRT es estudiada en [52]. Como conclusión principal de este estudio encontramos que solo podemos colocar varios procesos RT o SRT en un procesador multicore simétrico si controlamos el ancho de banda del bus de memoria disponible y su uso. Se recomiendo además que las aplicaciones RT o SRT modelen su comportamiento de acceso a memoria en ráfagas, para facilitar la planificación del uso del bus de memoria. En [46] encontramos un estudio sobre el efecto del impacto en la eficiencia en el acceso a memoria en sistemas SMP con diseño de memoria compartida. Contemplando el impacto

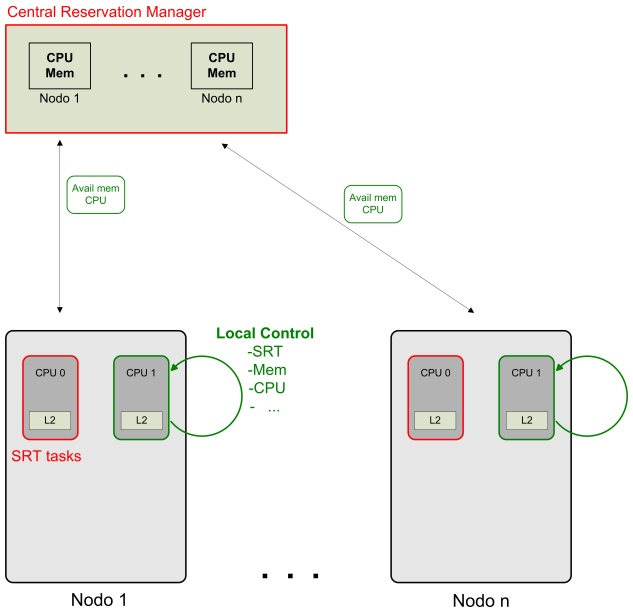

Figura B.2: Esquema de uso de los cores en procesadores Pentium D.

de la consumición de ancho de banda de memoria por parte de las nuevas tecnologías de red, usando como ejemplo Myrinet.

Los trabajos en este tema aún son embrionarios y escasos, siendo algunos de ellos interesantes consideraciones y recomendaciones a seguir.

B.2. Uso de la capacidad multicore de los procesadores

Otro de los desafíos a enfrentar en este trabajo es la forma de tomar ventaja de las bondades de los procesadores multicore, algunos de los cuales están descritos en la sección B.4. En [18], múltiples procesadores de diferente poder de cómputo son planificados con tareas SRT y Best-effort usando servidores diferidos para mejorar el rendimiento de las aplicaciones Best-effort. Esta idea nos parece válida, aún cuando en nuestro caso ambos cores tienen el mismo poder de cómputo.

La Figura B.2 muestra una aproximación a este problema desde nuestra óptica. Dado que hay disponibilidad de dos cores y tenemos dos grandes grupos de aplicaciones, se propone utilizar cada core para cada grupo de aplicaciones (SRT y Best-effort). Lógicamente para generar menos compentencia por los recursos, las tareas de control también serían realizadas por el core dedicado a las tareas Best-effort.

App/Proc	mgb8	mgb4	mgb2	mgab8	mgab4	mgab2	mga8
Pentium IV	75	119	209	49	65	129	19
Pentium D	52	84	138	31	50	84	11
SpeedUp	1,43	1,42	1,52	1,56	1,29	1,53	1,74

Tabla B.1: Tiempos y speedups para Pentium D y Pentium IV, tipo de aplicación NAS CPU bound.

App/Proc	isb8	isb4	isb2	isab8	isab4	isab2	isa8
Pentium IV	179	240	240	58	109	99	39
Pentium D	119	144	103	59	70	50	30
SpeedUp	1,50	1,67	2,33	0,99	1,55	1,97	1,31

Tabla B.2: Tiempos y speedups para Pentium D y Pentium IV, tipo de aplicación NAS IO bound.

B.3. Experimentación en procesadores Pentium D

Los experimentos realizados van destinados a calcular el speedup relativo entre procesadores Pentium D a procesadores Pentium tradicionales. Los procesadores Pentium D destacan por tener dos cores, cada uno a una velocidad de 3.6G Hz y una cache de 2 MB por cada core, lo cual hace un total de 4 MB. También ha de destacarse que el acceso a memoria es mejor, ya que tienen valores del FSB mayores que los procesadores Pentium empleados como comparativa. Los procesadores Pentium tienen una velocidad del reloj de 1.8 GHz y 512 KB de cache.

Las tablas B.3 y B.4 muestran los tiempos de ejecución y speedup de acuerdo a la caracterización de las aplicaciones del NAS. CPU bound significa que las aplicaciones emplean más tiempo calculando que accediento a disco e IO bound lo contrario. El speedup promedio para todas las ejecuciones es de 1,58 y llama la atención que sea tan bajo, teniendo en cuenta que no solo hay presentes dos procesadores, sino que también sus características particulares han mejorado de forma notable.

Para intentar aclarar el por qué del bajo speedup, repetimos el experimento anterior para los Pentiums D, pero con un proceso robándose toda la capacidad de cómputo de uno de los cores. Para asegurarnos que el sistema de balanceo de colas de Linux no intefería en nuestro experimento, ligamos nuestro ladrón de CPU a uno de los cores y el demonio de PVM al otro.

Los resultados parecen sorprendentes, pues el speedup promedio para este experimento (1,56) es apenas dos décimas menor que cuando empleamos los dos cores. Creemos que esto se debe al hecho de que la implementación de PVM es tratada como un proceso más por el SO, razón por la cual no toma gran

App/Proc	mgb8	mgb4	mgb2	mgab8	mgab4	mgab2	mga8
Pentium IV	75	119	209	49	65	129	19
Pentium D	52	82	154	31	50	80	16
SpeedUp	1,44	1,44	1,36	1,57	1,30	1,62	1,22

Tabla B.3: Tiempos y speedups para Pentium D (un core deshabilitado) y Pentium IV, tipo de aplicación NAS CPU bound.

App/Proc	isb8	isb4	isb2	isab8	isab4	isab2	isa8
Pentium IV	179	240	240	58	109	99	39
Pentium D	117	144	102	58	71	49	29
SpeedUp	1,53	1,67	2,35	0,99	1,55	2,00	1,33

Tabla B.4: Tiempos y speedups para Pentium D (un core deshabilitado) y Pentium IV, tipo de aplicación NAS IO bound.

ventaja de la capacidad multicore del procesador. Este situación refuerza la propuesta antes realizada, ya que el hecho de emplear los procesadores por separado no afecta de forma notable la ejecución de las aplicaciones.

B.4. Algunas Arquitecturas SMP Actuales

Symmetric multiprocessing (SMP) es el nombre de una arquitectura de ordenadores multiprocesador en la cual dos o más procesadores idénticos están conectados a una única memoria pricipal compartida. En este tipo de sistema, la ubicación de la cache de nivel 2 (L2), define el coste de las migraciones de tareas entre procesadores. En esta sección repasaremos los procesadores actuales que soportan arquitecturas SMP, cabe destacar que este tipo de arquitectura es bastante común en los procesadores comerciales.

Una amplia variedad de procesadores soportan SMP en la actualidad, entre ellos podemos mencionar:

- *Xeon* de Intel: Soportan sistemas SMP desde mediados del 2001, es la línea de procesadores de Intel destinada a servidores. Actualmente se producen procesadores Xeon en versiones Dual-Core y Quad-Core. En todas las versiones con varios cores la L2 es compartida.

- *Intel Core*: Diseñado para ordenadores portátiles, por lo que se centra en el ahorro energético de los procesadores. Al igual que su sucesor directo (*Intel Pentium Dual-Core*) los cores integrantes del procesador comparten la misma cache.

- *Intel Core 2*: Representa la unión de las dos líneas de procesadores de Intel, para ordenadores portátiles y de escritorio. Puede contener 2 (versión Duo) o 4 (versión Quad) cores, con cache L2 compartida.

- *Opteron* de AMD: Lanzado al mercado en abril del 2003, fue diseñado para competir con los procesadores Xeon de Intel en el mercado de los servidores. Desde mayo del 2005 tiene una versión de dos cores y se espera que para mediados del 2007 salga al mercado la versión Quad-Core. Además de arquitectura SMP soporta arquitecturas NUMA.

- Ultra *SPARC*: Lanzados al mercado desde finales del 2005, en versiones de 4, 6 y 8 cores, soportando 4 hilos por cada uno de ellos. Utiliza una cache L2 compartida de 3 MB de tamaño y técnicas para lograr un uso más eficiente.

- *Pentium D*: Lanzado al mercado en mayo del 2005, incluye dos unidades Pentium completas actuando cada una como un core, que disponen de una cache L2. Los tamaños de las L2 por core están asociados a la generación a la que pertenecen, siendo de 2x1 MB o 2x2 MB.

Ha de destacarse que aún cuándo en los procesadores multicore la tendencia parece ser el uso de la cache L2 compartida, en este trabajo la experimentación habrá de realizarse en procesadores Pentium D "Presler". Estos procesadores conforman la última y más nueva generación de procesadores de esta gama, fabricados en tecnología de 65 nanómetros, con una velocidad del reloj de 3.6 GHz y una cache L2 2 MB por cada core. En esta línea de procesadores, la comunicación entre los cores se hace a través del FSB (*Front Side Bus*), lo cual genera efectos tan negativos como la contención de bus y el hecho de que cuando el procesador está muy cargado ha de dividirse el ancho de banda existente (800 MT/s) entre los cores.

La contención de bus es un efecto ya presente en otros procesadores, como los procesadores Xeon, con este término nos referimos a los efectos no deseados que ocurren cuando más de un core intenta escribir en el bus de memoria. Este efecto no deseado puede llevar a operaciones erróneas o daños al hardware. Cabe destacar que no es un problema solo de los buses de memoria, puede ocurrir también con cualquier bus del sistema con más de un dispositivo asociado al bus.

Bibliografía

[1] TF Abdelzaher, KG Shin, and N. Bhatti. User-level qos-adaptive resource management in server end-systems. *Computers, IEEE Transactions on*, 52(5):678–685, 2003.

[2] L. Abeni and G. Lipari. Implementing resource reservations in linux. 2002.

[3] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, pages 4–13, 1998.

[4] K. Aida. Effect of job size characteristics on job scheduling performance. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1911:1–17, 2000.

[5] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[6] J. Anderson and J. Calandrino. Parallel real-time task scheduling on multicore platforms. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[7] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[8] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 93, Washington, DC, USA, 2001. IEEE Computer Society.

[9] A. Andrzejak, P. Domingues, and L. Silva. Predicting machine availabilities in desktop pools. 2006.

[10] R.H. Arpaci, A.C. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A. Patterson. The interaction of parallel and sequential workloads on

a network of workstations. *ACM SIGMETRICS 1995*, pages 267–277, 1995.

[11] A. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[12] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 123–132, 1998.

[13] A.K. Atlas and A. Bestavros. Slack stealing job admission control. Technical report, Technical Report BUCS-TR-98-009, Boston University, Computer Science Department, 1998.

[14] SA Banachowski and SA Brandt. Best scheduler for integrated processing of best-effort and soft real-time processes. *PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*, pages 46–60, 2002.

[15] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[16] GC Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks inhard real-time environments. *Computers, IEEE Transactions on*, 48(10):1035–1052, 1999.

[17] R. Buyya, M. Murshed, and D. Abramson. A deadline and budget constrained cost-time optimization algorithm for scheduling task farming applications on global grids. 2002.

[18] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[19] Stephen Childs and David Ingram. The linux-srt integrated multimedia operating system: Bringing qos to the desktop. *rtas*, 00:0135, 2001.

[20] M.L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974.

[21] Desmoj Developer Team. Desmo-j project: http://asi-www.informatik.uni-hamburg.de/desmoj/.

[22] S.K. Dhall and CL Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[23] PA Dinda. A prediction-based real-time scheduling advisor. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10–17, 2002.

[24] P. Domingues, P. Marques, and L. Silva. Resource usage of windows computer laboratories. *International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 469–476, 2005.

[25] B. Doytchinov, J. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with earliest-deadline-first queue discipline. *Ann. Appl. Probab*, 11(2):332–378, 2001.

[26] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Desktop scheduling: How can we know what the user wants? *ACM NOSSDAV*, pages 110–115, 2004.

[27] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Process prioritization using output production: Scheduling for multimedia. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2(4):318–342, 2006.

[28] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grained synchronization. *Journal on Parallel and Distributed Computing (JPDC'92)*, 16(4):306–318, 1992.

[29] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1291:1–34, 1997.

[30] E. Frachtenberg, D. G. Feitelson, J. Fernández, and F. Petrini. Parallel job scheduling under dynamic workloads. *Job Scheduling Strategies for Parallel Processing. High Perfomance Distributed Computing (HPDC'03), Seattle, Washington. Lecture Notes in Computer Science*, 2862.:208–227, June 2003.

[31] E. Frachtenberg, F. Petrini, S. Coll, and W. Feng. Gang scheduling with lightweight user-level communication. In *Proceedings of International Conference on Parallel Processing Workshops (ICPPW'01)*, Valencia, Spain, 2001. Workshop on Scheduling and Resource Management for Cluster Computing.

[32] Eitan Frachtenberg. Process scheduling for the parallel desktop. In *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, pages 132–139, Washington, DC, USA, 2005. IEEE Computer Society.

[33] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[34] F. Giné. *Cooperating Coscheduling: a coscheduling proposal for non-dedicated, multiprogrammed clusters.* PhD thesis, Universitat Autònoma de Barcelona, July 2004.

[35] A. Gupta, B. Lin, and P. A. Dinda. Measuring and understanding user confort with resource borrowing. *13th IEEE International Symposium on high Performance and Distributed Compuing (HPDC'04), Honolulu, USA*, 2004.

[36] M. Hanzich. *A Temporal and Spatial Scheduling System for Non-dedicated Clusters.* PhD thesis, Universidad Autónoma of Barcelona, July 2006.

[37] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Coscheduling and multiprogramming level in a non-dedicated cluster. *EuroPVM/MPI 2004, Lecture Notes in Computer Science*, 3241:327–336, 2004.

[38] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. 3DBackfilling: A space sharing approach for non-dedicated clusters. In *Parallel and Distributed Computing and Systems (PDCS'05)*, volume 17, pages 131–138. ACTA Press, 2005.

[39] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Cisne: A new integral approach for scheduling parallel applications on non-dedicated clusters. *EuroPar 2005, Lecture Notes in Computer Science*, 3648:220–230, 2005.

[40] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. What to consider for applying backfilling on non-dedicated environments. *Journal on Computer Science and Technology*, 5(4):189–195, 2005.

[41] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Using on-the-fly simulation for estimating the turnaround time on non-dedicated clusters. *To be appeared in EuroPar 2006, Lecture Notes in Computer Science*, 2006.

[42] L. He, S.A. Jarvis, D.P. Spooner, and G.R. Nudd. Dynamic, capability-driven scheduling of dag-based real-time jobs in heterogeneous clusters. *International Journal of High Performance Computing and Networking*, 2(2):165–177, 2004.

[43] D. Jackson, Q. Snell, and Mark Clement. Core algorithms of the maui scheduler. *Job Scheduling Strategies for Parallel Processing, Cambridge, MA, USA*, 2221:87–102, June 2001.

[44] S. A. Jarvis, D. P. Spooner, H. N. Lim Choi Keung, J. Cao, S. Saini, and G. R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems, Special Issue on System Performance Analysis and Evaluation*, 2004.

[45] M. Jette and M. Grondona. Slurm: Simple linux utiltity for resource management. *ClusterWorld 2003 Conference and Expo*, June 2003.

[46] Evangelos Koukis and Nectarios Koziris. Memory bandwidth aware scheduling for smp cluster nodes. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 187–196, Washington, DC, USA, 2005. IEEE Computer Society.

[47] B. J. Lafreniere and A. C. Sodan. Scopred—scalable user-directed performance prediction using complexity modeling and historical data. *11th Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 2005.

[48] T. Lechler and B. Page. Desmo-j: An object oriented discrete simulation framework in java. In *Proceedings of the European Simulation Symposium '99*, 1999.

[49] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization andaverage case behavior. *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, 1989.

[50] J. P. Lehoczky. Real-time queueing theory. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 186, Washington, DC, USA, 1996. IEEE Computer Society.

[51] J. P. Lehoczky. Real-time queueing network theory. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 58, Washington, DC, USA, 1997. IEEE Computer Society.

[52] Jochen Liedtke, Marcus Völp, and Kevin Elphinstone. Preliminary thoughts on memory-bus scheduling. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210, New York, NY, USA, 2000. ACM Press.

[53] M. Litzkow, M. Livny, and M. Mutka. Condor- a hunter of idle workstations. *8th International Conference of Distributed Computing Systems*, 1988.

[54] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[55] W. Liu, V. Lo, K. Windisch, and B. Nitzberg. Non-contiguous processor allocation algorithms for distributed memory multicomputers. *IEEE/ACM Supercomputing 1994*, pages 227–236, November 1994.

[56] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2P scheduling of idle cycles in the internet. 2004.

[57] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. *ACM SIGMETRICS, Conference on Measurement and Modeling of Computer Systems*, pages 104–113, May 1988.

[58] S. McClure and R. Wheeler. Mosix: How linux clusters solve real-world problems. *Proceedings of the USENIX Annual Technology Conference*, pages 49–56, June 2000.

[59] A. K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. Technical report, Cambridge, MA, USA, 1983.

[60] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transaction on Parallel & Distributed Systems*, 12(6):529–543, 2001.

[61] J. Mugler, T. Naughton, S. L. Scott, B. Barrett, A. Lumsdaine, J. M. Squyres, B. Ligneris, F. Giraldeau, and C. Leangsuksun. Oscar clusters. In *Proceedings of the Ottawa Linux Symposium (OLS'03)*, Ottawa, Canada, 23-26, 2003.

[62] M. Netto, R. N. Calheiros, R. K. S. Silva, C. F. De Rose, C. Northfleet, and W. Cirne. Transparent resource allocation to exploit idle cluster nodes in computational grids. In *Proceedings of The First IEEE International Conference on e-Science and Grid Computing*, pages 238–245, Melbourne, Australia, 2005. IEEE Computer Society.

[63] J. Ousterhout. Scheduling techniques for concurrent systems. *Proceedings of the International Conference on Distributed Computing Systems*, 1982.

[64] F. Petrini and W. Feng. Improved resource utilization with buffered coscheduling. *Conference on High Performance Networking and Computing, Baltimore, Maryland. Proceedings of the 2002 ACM/IEEE conference on Supercomputing.*, 1-26, 2002.

[65] George & Sharma Akshay. Plale, Beth & Turner. Real time response to streaming data on linux clusters. Technical report, Indiana University. Computer Science Department Technical Report TR-569, November 2002.

[66] J. Regehr and J.A. Stankovic. Augmented cpu reservations: Towards predictable execution on general-purpose operating systems. *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 141–148, 2001.

[67] M. A. Rodriguez, F. Diaz del Río, C. Amaya, E. Florido, R. Senhadji, and G. Jiménez. Multicomputador hibernable: una solución para compartir los recursos de computación de los laboratorios docentes. *XIII Jornadas de Paralelismo, Lleida*, pages 117–122, 2002.

[68] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. *IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[69] Q. O. Snell, M. J. Clement, and D. B. Jackson. Preemption based backfill. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 2537:24–37, July 2002.

[70] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1459:231–256, 1998.

[71] P. Sobalvarro and W. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 949:106–126, 1995.

[72] F. Solsona. *Coscheduling Techniques for Non-Dedicated Cluster Computing*. PhD thesis, Universitat Autònoma de Barcelona, 2002.

[73] F. Solsona, F.Giné, P. Hernández, and E. Luque. Implementing explicit and implicit coscheduling in a pvm environment. *EuroPar 2000, Lecture Notes in Computer Science*, 1900:1165–1170, 2000.

[74] F. Solsona, F. Giné, P. Hernández, and E. Luque. Predictive coscheduling implementation in a non-dedicated linux cluster. *EuroPar 2001, Lecture Notes in Computer Science*, 2150:732–741, 2001.

[75] M. Spuri and GC Buttazzo. Efficient aperiodic service under earliest deadline scheduling. *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11, 1994.

[76] S. Srinivasan, R. Kettimuthu, V. Subrarnani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. *International Conference on Parallel Processing Workshops (ICPPW'02)*, pages 514–522, 2002.

[77] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Selective buddy allocation for scheduling parallel jobs on clusters. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2002)*, September 2002.

[78] H. Streich and M. Gergeleit. On the design of a dynamic distributed real-time environment. In *WPDRTS '97: Proceedings of the 1997 Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS / OORTS '97)*, page 251, Washington, DC, USA, 1997. IEEE Computer Society.

[79] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[80] D. Talby and D. G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 84.1, Washington, DC, USA, 2005. IEEE Computer Society.

[81] S.R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. *Real-Time Systems Symposium*, pages 22–33, 12 1994.

[82] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using runtime predictions rather than user estimates. Technical Report TR 2005-5, School of Computer Science and Engineering, Hebrew University of Jerusalem, November 2005.

[83] M.T. Yang, R. Kasturi, and A. Sivasubramaniam. An automatic scheduler for real-time vision applications. *Parallel and Distributed Processing Symposium., Proceedings 15th International*, page 8, 2001.

[84] Y. Zhan and A. Sivasubramaniam. Scheduling best-effort and real-time pipelined applications on time-shared clusters. *Proceedings of the 13th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA'2001)*, pages 209–218, 2001.

[85] J. Zhang, T. Hamalainen, and J. Joutsensalo. A new mechanism for supporting differentiated services in cluster-based network servers. *mascots*, 00:0427, 2002.

[86] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. A comparative analysis of space- and time-sharing techniques for parallel job scheduling in large scale parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[87] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003.

[88] W Zhu. Allocating soft real-time tasks on clusters. *SIMULATION77*, 5-6:219–229, 2001.