Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Consenso en ambientes no homogéneos en base a una plataforma de tiempo real

Tesis
Que para optar por el grado de
Maestro en Ingeniería
(Computación)

Presenta:

José Ángel Hermosillo Gómez

Tutor: Dr. Héctor Benítez Pérez

México, D.F. abril 2013

Dedicado a
Isabel, Rigoberto,
Rigoberto y Joel

Agradecimientos

Agradezco a todas aquellas almas bondadosas que tuvieron la paciencia y el temple para soportarme, y guiarme por el camino del entendimiento durante este periodo y, además, hacerme ver que la investigación es una labor colaborativa que requiere de concentración, pasión e interés.

Gracias al Dr. Héctor Benítez Pérez, al Dr. Jorge L. Ortega Arjona; a los miembros del jurado, por invertir su tiempo en la lectura de esta tesis y aportar sus ideas para mejorarla (Dra. Ma. Elena Lárraga Ramírez, Dr. Víctor Rangel Licea, Dr. D. Fabian García Nocetti y Dr. Javier Gómez Castellanos), y a todos aquellos que por falta de memoria no cito aquí.

Índice general

Introducción

En esta tesis se presenta una estrategia de planificación global de tareas a partir de acciones de consenso *en línea* entre subconjuntos de nodos en un sistema distribuido heterogéneo.

Un sistemas de tiempo real es frecuentemente modelado como un conjunto finito de tareas independientes, las cuales pueden ser vistas como una sucesión infinita de subtareas (o jobs). Cada subtarea es caracterizada por un tiempo de liberación, tiempo de ejecución y un plazo. El requisito en los sistemas de tiempo real para cada subtarea consiste en que éstas se realicen después de su tiempo de liberación y antes de su plazo.

La planificación es la asignación de tiempo de procesador a las subtareas, mientras que un algoritmo de planificación es la estrategia empleada para determinar dicha asignación. Para determinar que un sistema es planificable es necesario definir un test de planificabilidad que considere sus restricciones, y con él determinar que será posible, bajo cualquier circunstancia, cumplir con todos los plazos impuestos a una tarea. Este trabajo se enfoca en la planificación de tres tipos de tareas: periódicas, aperiódicas y esporádicas, en un sistema distribuido compuesto por nodos con capacidades diferentes. Sólo se consideran sus velocidades y las resoluciones a las que trabajan.

La planificación, en un entorno distribuido, se puede llevar a cabo de dos formas [4]:

1. *Planificación global*: Consiste en asignar una tarea a un grupo de procesadores e implementar una estrategia de planificación local en dicho grupo. En este tipo de planificación una subtarea se puede ejecutar en cualquier procesador del sistema, inclusive una subtarea que ha sido interrumpida por otra de mayor prioridad puede retomar su ejecución en otro nodo distinto del que comenzó su ejecución.

2. *Planificación seccionada*: Consiste en asignar una o varias tareas en un sólo procesador (haciendo una analogía con la planificación global, es asignar una tarea a un grupo de procesadores de tamaño uno) por lo que todas las subtareas son ejecutadas por el mismo procesador.

Las formas de planificación distribuida anteriormente expuestas revelan un componente más: la asignación de tareas a los nodos.

El poder ejecutar una tarea en cualquiera de los procesadores de un grupo *durante la planificación* resulta ser llamativo, sin embargo; es inapropiado debido a la carga que esto podría representar para la red, ya que para que ésto ocurra se debe transferir la tarea entre los procesadores que conforman al grupo, aunque una de las ventajas es la posibilidad de lograr niveles de carga similares.

Por lo anteriormente dicho, la planificación seccionada resulta conveniente ya que de esta forma no habría necesidad de transferir tareas durante la planificación y, entonces, el problema

de planificación se convierte en un problema interno al nodo que le fue asignado un conjunto de tareas, pero con la desavenencia de que el proceso de asignación de tareas entre los procesadores del sistema podría tomar bastante tiempo en encontrar un conjunto de tareas que pueda ser planificado en un procesador y que haga un uso eficiente del mismo.

Por dichas razones, se propone aquí que el conjunto de tareas esporádicas que se desea planificar sea repartido de manera tal que una misma tarea esporádica exista en, al menos, dos nodos distintos del sistema distribuido sin importar la manera en que las tareas son asignadas, esto permitirá un dinamismo, ya que existe más de un lugar (un nodo) posible en donde ejecutar una tarea.

El subconjunto de nodos del sistema que poseen a la misma tarea esporádica se le denomina subconjunto de nodos *emparentados*, pero sólo uno de ellos podrá ejecutarla. Ésta será ejecutada en el nodo más apto según lo determine un acuerdo logrado por *consenso* entre nodos emparentados, y dado que el criterio de decisión consiste en elegir al nodo con menor carga durante un *horizonte* de tiempo, también se logra una carga de trabajo relativamente similar entre los nodos que conforman el sistema.

Una suposición importante para este trabajo es que la saturación de la red con la transferencia de tareas es mayor a la saturación que pudiese haber con los mensajes enviados entre los nodos durante alguna acción de consenso.

Sistema distribuido

Un sistema distribuido consiste de varios procesos distintos repartidos en múltiples nodos (procesadores, computadoras, sensores, etc.) trabajando bajo ciertos requisitos funcionales. Los nodos, generalmente, están geográficamente dispersos, y los procesos distribuidos deben estar coordinados a través de la comunicación y la sincronización.

Los sistemas distribuidos se clasifican en homogéneos y heterogéneos. Los nodos en los sistemas distribuidos homogéneos poseen la misma arquitectura y los soporta el mismo software; en cambio, los sistemas heterogéneos poseen arquitecturas y, posiblemente, software distintos. Los sistemas distribuidos también pueden ser clasificados en descentralizados o centralizados. En los sistemas distribuidos centralizados, los nodos tienen una relación cliente-servidor o manager-workers [41], mientras que en los sistemas descentralizados cada nodo es autónomo.

Sistema de tiempo real

Un sistema de tiempo real es un sistema que trabaja bajo restricciones temporales tales como límites de tiempo, tiempos de liberación de tareas, tiempos de ejecución, entre otros. Los sistemas continuamente muestrean datos del medio que le rodea durante su operación. Una vez que los datos son muestreados, un subsistema computacional los procesa de manera inmediata y regresa una respuesta apropiada al medio que le rodea. El proceso descrito, compuesto por las operaciones de muestreo, procesamiento y emisión de respuestas, debe ejecutarse dentro de las restricciones de tiempo impuestas.

Sin importar si un sistema distribuido en tiempo real es homogéneo o heterogéneo, éste hereda las características de ambos sistemas, de tiempo real y distribuidos, así que la correctez de los sistemas distribuidos de tiempo real debe considerar los requisitos de ambos sistemas. Un sistema distribuido de tiempo real se encarga de la distribución de tareas computacionales

sobre diferentes nodos, mientras que el procesamiento de cada tarea distribuida debe satisfacer las restricciones de tiempo que le fueron asignadas.

Un sistema distribuido tiene como objetivos mejorar el tiempo de respuesta, incrementar la confiabilidad y la predictibilidad. Un sistema de tiempo real puede descomponerse en un conjunto de procesos concurrentes regulando la comunicación y la sincronización. Se espera que los sistemas distribuidos de tiempo real operen continuamente y sean totalmente confiables, más aun si la aplicación que se implementa sobre éste debe cumplir con plazos de tiempo estrictos. Aplicaciones de suma importancia como los algoritmos de planificación juegan el papel primordial de determinar cómo y cuándo ha de ejecutarse una tarea dentro del sistema operativo.

Consenso

Ahora, es necesario establecer las bases de otro elemento constituyente para el desarrollo de este trabajo, el consenso, que dependiendo del contexto del problema puede ofrecer una solución. El consenso no solamente es utilizado en cuestiones de tolerancia a fallas, es tan flexible que de hecho lo aplicamos aquí como el mecanismo que nos permite acordar de entre un subconjunto de nodos cual resulta el más apto para hacerse cargo de la ejecución de una tarea.

Para el problema de lograr consenso consideremos un conjunto de N procesos p_i, donde $i = 1, \ldots, N$, que se comunican mediante el paso de mensajes y con ausencia de fallas. Cada proceso p_i comienza con un estado de indecisión y *propone* inicialmente un valor v_i, luego todos los procesos correctos tienen que *decidir* sobre un valor común v que es igual a alguno de los valores v_i propuestos.

Los acuerdos en el consenso no sólo permiten establecer el valor de algunas variables de importancia en los sistemas distribuidos, también es posible prevenir fallas. Como ejemplo, en la práctica, el consenso juega un papel importante para lograr la consistencia en las bases de datos; es un proceso que permite a sistemas multiagente establecer distancias y velocidades[46], o bien controlar el problema de lograr niveles justos de calidad de servicio (QoS) evitando condiciones de sobrecarga [21]; inclusive, el consenso se hace presente en los sistemas de seguridad conformados por cámaras de video [49], etcétera.

El consenso ha sido estudiado con base a diferentes tipos de modelos de sistemas, tales como los modelos de tiempo síncronos y asíncronos; bajo fallas bizantinas, crash failures y las fallas debidas al medio de comunicación; además de los esquemas de paso de mensajes y memoria compartida. En los sistemas asíncronos, el consenso se ve restringido por el resultado producido por Fischer, Lynch y Patterson, el cual asegura que es imposible alcanzar consenso de manera determinista [16]. Sin embargo, existen trabajos en los que se realizan ajustes que permiten alcanzar el consenso utilizando la aleatoriedad [45], haciendo suposiciones de sincronía o del tiempo sobre los sistemas [14, 15], usando detectores de fallas [9], mediante oráculos [44] e imponiendo condiciones sobre las entradas [40].

Sistema distribuido heterogéneo

Un problema latente en éstos modelos de computación distribuidos es la heterogeneidad, ya que resulta ser un factor que restringe el uso efectivo de los recursos. Cada vez es más probable encontrar sistemas que no poseen una base homogénea, es decir, los sistemas están constituidos por una amplia variedad de dispositivos interconectados con sistemas operativos distintos, que soportan aplicaciones desarrolladas en diferentes lenguajes de programación, cuyo intercambio

de información se da a través de diferentes medios que se auxilian de software de comunicación.

En los sistemas heterogéneos las tareas deben especificar que recursos son necesarios para poder ser ejecutadas. Por lo tanto, en los sistemas heterogéneos, cada procesador debe proveer una lista de sus recursos disponibles y cada tarea debe especificar, de forma completa, todos sus requerimientos, además, como los procesadores y los medios de transmisión varían en la velocidad, predecir el tiempo de ejecución para las tareas se vuelve imposible, característica que nos remite a los sistemas asíncronos.

La problemática

En este punto se revela una peculiar intersección de los tres temas mencionados: tiempo real, consenso y heterogeneidad; que es el motivo del trabajo desarrollado en la presente tesis. La problemática por atender es cómo aplicar el consenso en sistemas distribuidos no homogéneos para lograr satisfacer los requerimientos de tiempo de un sistema de tiempo real.

La estrategia que, en este trabajo de tesis, se propone tiene lugar en un sistema no centralizado, heterogéneo, de tiempo real, que no contempla fallas y que se apoya en el consenso para planificar globalmente un conjunto de tareas que son repartidas a lo largo del sistema conformado por nodos sincronizados. La heterogeneidad en el sistema trae consigo aspectos adicionales que deben ser considerados para este trabajo: las diferentes velocidades de procesamiento de los nodos en el sistema y la resolución del procesador a la que cada nodo funciona. Las particularidades de cada uno de los nodos limitan la manera de como se hace la asignación de las tareas, es decir, la asignación de una tarea a un nodo no es aleatoria, asignaremos una tarea a un nodo siempre que el nodo ofrezca los recursos que la tarea requiere para ser ejecutada.

Cuando un conjunto de tareas en un nodo se vuelve incompatible, es decir, que la ejecución de una tarea torna imposible la ejecución de alguna otra, es necesaria la participación de más unidades de procesamiento que sean capaces de atender las necesidades de las tareas en el sistema, de no contar con más procesadores se prescinde, preferentemente, de la tarea con mayor número de incompatibilidades o de aquella con menor prioridad según los requerimientos del sistema.

Cuando las decisiones de planificación se deben tomar *en línea* y la cantidad de tareas crece al grado de requerir de los servicios de diferentes componentes externos, con capacidades diferentes de procesamiento y comunicación, surge el problema de encontrar una estrategia que permita hacer un uso prudente de los recursos disponibles en el sistema distribuido, razón por la que se considera el consenso.

Al haber varios nodos con el objetivo de planificar en tiempo de ejecución (en línea), se requiere de una interacción entre éstos capaz de permitir que el sistema lleve a cabo las tareas que le fueron asignadas. El consenso se vuelve una manera viable que permite hacer un uso prudente de los componentes, seguramente diferentes, que conforman al sistema distribuido.

Objetivos

En el presente trabajo se muestra un nueva propuesta de planificación, nombrada como *"Planificación global de tareas confinadas basado en consenso"* y se contempla realizar lo siguiente:

1. Proponer un modelo de ejecución global de tareas, en el que dadas algunas tareas con

restricciones de tiempo estrictas (p.e. deadlines, tiempos de activación y ejecución) y un conjunto de procesadores heterogéneos, estos últimos sean capaces de lograr acuerdos mediante el consenso para garantizar la ejecución de un subconjunto de tareas esporádicas, cumpliendo con las restricciones de tiempo establecidas. Considerando que existen elementos externos que provocan cambios tanto topológicos como estructurales en las relaciones de comunicación y procesamiento local.

2. Estudiar y analizar los mecanismos existentes de planificación de tareas y de consenso.
3. Identificar las propiedades y limitaciones de los mecanismos del punto previo.
4. Realizar un análisis exhaustivo del modelo propuesto.
5. Plantear un algoritmo bajo el cual sea posible planificar un conjunto de tareas con más de una política de planificación local.
   a) Identificar las condiciones que permitan elegir el nuevo algoritmo de planificación que permita aprovechar lo mejor posible el procesador para la ejecución de un conjunto de tareas; es decir, elegir la política de ejecución de tareas que no afecte la ejecución de las tares en el sistema.
   b) Proponer un criterio que permita determinar el momento en que se debe efectuar el proceso de consenso.
6. Verificar la correctez del modelo.

Metodología

Para el logro de los objetivos de esta tesis se propone la siguiente metodología:

1. Realizar un análisis minucioso de los algoritmos existentes de consenso y planificación de tareas.
2. Diferenciar las cualidades de los diferentes algoritmos estudiados de consenso y planificación.
3. Hacer una revisión de los algoritmos de planificación más utilizados, actualmente implementados en los sistemas operativos.
4. Especificar el nuevo algoritmo de planificación con base a una estrategia previa de consenso.
5. Probar la correctez del algoritmo.

Metas

- Definir un algoritmo de planificación global, para el cual se establecerán las condiciones que permitan efectuar el consenso en un sistema heterogéneo con múltiples procesadores para el cambio oportuno de las políticas de planificación que definen su comportamiento de acuerdo a los requerimientos que vayan surgiendo conforme transcurre el tiempo.
- Diseñar e implementar una herramienta de software para la obtención de resultados, basada en programación orientada a objetos (lenguaje $C\sharp$).

Hipótesis

Es posible satisfacer los requerimientos de tiempo de un conjunto de tareas planificables en un sistema distribuido, y con una política de planificación que se auxilie del consenso, sin que se retrase la ejecución de alguna de las tareas que forman parte del sistema.

Estructura del trabajo

Este escrito está conformado por tres capítulos. En el capítulo uno se hace una breve revisión de los aspectos que deben ser considerados para el diseño de sistemas de tiempo real en general y de algoritmos teóricos básicos de planificación. En el capítulo dos se encuentra la contribución de este trabajo, se detallan las especificaciones y la definición de un nuevo algoritmo de planificación, así como de proporcionar una revisión de trabajos que se han hecho alrededor del tema de planificación en tiempo real, relacionados con este trabajo. El último capítulo contiene los resultados de las pruebas realizadas sobre el simulador programado en lenguaje $C\sharp$. Al final, una última sección se ha agregado con las conclusiones generales, desventajas y trabajo a futuro.

Capítulo 1

Aspectos y conceptos relacionados

En el presente capítulo se revisan algunos conceptos necesarios para conocer los sistemas de tiempo real, algoritmos de planificación de tareas, consenso, entre otros.

1.1. Introducción

Existen disponibles varias maneras de diseñar algoritmos de planificación para sistemas de tiempo real, pero podemos englobar estas formas de planificación en tres vertientes: en el que las tareas se planifican de manera dinámica o estática, cuando las tareas se planifican de acuerdo a prioridades y permitir la apropiación y, cuando el planificador es dinámico, se deberían planificar tareas orientándonos a la asignación o a una secuencia.

El diseño de cualquier sistema de tiempo real debe contemplar una amplia variedad de aspectos, todos ellos relevantes, se hace un recuento de ellos en la sección 1.2. Para el desarrollo de un sistema en tiempo real existen restricciones que deben ser examinadas y algunas de ellas han sido listadas en la sección 1.3. En la sección 1.4, se encuentra una clasificación de las diferentes maneras en que es posible planificar un conjunto de tareas. En la sección 1.4.2, se hace una revisión de algunos algoritmos de planificación de tareas periódicas y de los que se derivan la mayoría de algoritmos actuales. Dado que el consenso es una herramienta de la que se auxilia este trabajo, en la sección 1.7 se menciona el tema de consenso y propiedades que deben ser consideradas en algoritmos que hagan uso de este recurso. Por último, dado que el algoritmo presentado en el capítulo 4 es asíncrono (en el aspecto de que los nodos son independientes e interaccionan con otros, sólo cuando una tarea esporádica es liberada, para llegar al acuerdo que es requerido para asignar la ejecución de la tarea a un sólo nodo), en la sección 1.8 se hace una distinción de estos tipos de sistemas.

1.2. Sistemas de tiempo real

Los sistemas de tiempo real son aquellos sistemas predecibles que poseen la característica de proporcionar una respuesta correcta a una determinada entrada de datos dadas ciertas condiciones de tiempo, esto es, en un tiempo definido o *plazo* es posible contar con un resultado. Si las restricciones de tiempo no se han completado, el sistema ha fallado o degradado su ejecución.

El manejo de restricciones de tiempo se extienden a muchas áreas de aplicación tales como la bolsa de valores, control de reacciones en plantas nucleares [6], sistemas médicos [6], monitoreo

de los océanos [18], control digital[36], procesamiento de señales [36], control del tráfico aéreo[36], etcétera.

1.2.1. Aspectos del diseño

Un sistema de tiempo real es un sistema computacional cuyo comportamiento es fijado por las características de la aplicación. Se han identificado muchas características para analizar e implementar sistemas de tiempo real. Algunas de ellas son:

1. Identificar todas las tareas que conforman a la aplicación así como sus restricciones temporales (plazos, períodos, tiempos de activación, entre otros) que se pretenden cumplir.
2. Especificar la manera en la que se llevará a cabo el manejo de interrupciones y cambios de contexto.
3. Establecer la manera en la que serán asignadas las prioridades a las tareas.
4. Especificar la forma en la que se manipularán las tareas que presentan algún tipo de relación.
5. Realizar el análisis correspondiente de planificabilidad. El análisis para determinar que un conjunto de tareas es planificable depende de la movilidad de las tareas entre las unidades de procesamiento, es decir, si el conjunto de tareas es dedicado a un nodo, entonces el análisis es local y cuando las tareas pueden ser transferidas entre las unidades de procesamiento que conforman al sistema distribuido, el análisis es global.
6. Implementar el modelo propuesto. Se deben considerar varios aspectos como son la arquitectura del hardware como la del software, las características del sistema operativo, los requisitos de la aplicación, el lenguaje de programación principalmente.

Algunas de las desventajas que se presentan al programar este tipo de aplicaciones son:

- *Programación tediosa.* En ocasiones, cuando se desarrollan sistemas con restricciones rigurosas de tiempo es necesario programar en lenguaje ensamblador; la programación requiere de mayor tiempo y es más difícil.
- *Dificultad para comprender el código.* Excepto por los programadores quienes desarrollaron la aplicación, muy poca gente puede ser capaz de entender enteramente el código de la aplicación producida.
- *Dificultad en la manutención el software.* Según crece el tamaño del software incrementa su complejidad, la modificación de grandes piezas de código se vuelve difícil aún para los programadores originales.
- *Dificultad de verificación en las restricciones de tiempo.* Sin el soporte de tecnologías y de metodologías para el código y análisis de planificación, la verificación de las restricciones de tiempo llega a ser imposible.

El campo de la computación en tiempo real es especialmente rica en retos de investigación, debidos a todos los problemas en arquitectura de computadoras, tolerancia a fallas, sistemas operativos, sincronización y concurrencia. Estos problemas tienen en su haber la característica agregada de que las restricciones de tiempo real se deben satisfacer (plazos y tiempos de activación, por ejemplo).

Los sistemas de tiempo real se diferencian de sus homólogos de propósito general en dos partes importantes. Primera, son mucho más específicas en sus aplicaciones. Segunda, las consecuencias de sus fallas tienen efectos drásticos, por lo que necesitan ser cuidadosamente definidos y su ejecución debe ser validada.

En general, el de diseño de sistemas distribuidos de tiempo real cubre las siguientes características [56]:

1. **Sincronización del reloj:** El primer aspecto es el mantenimiento de la propia hora. Con varias computadoras, cada una con su propio reloj local, la sincronización de relojes es un aspecto fundamental.

   La sincronización es más compleja en sistemas distribuidos que en los centralizados, puesto que los primeros deben utilizar algoritmos distribuidos. Por lo general, no es posible (o recomendable) reunir toda la información relativa al sistema en un lugar y después dejar que cierto proceso la examine y tome una decisión, como se hace en el caso centralizado.

   En la práctica, cuando un sistema tiene n computadoras, los n relojes correspondientes oscilarán a tasas distintas, lo que genera una perdida de sincronía en los relojes. La diferencia entre los valores de tiempo se llama **distorsión del reloj**, δ. Como consecuencia de esta distorsión, podrían fallar los programas que esperan que el tiempo asociado a un archivo, objeto, proceso o mensaje sea correcto e independiente del sitio donde haya sido generado.

   Lamport demostró que la sincronización de relojes es posible y presentó un algoritmo [25, 26] para lograrlo. Lamport señaló que la sincronización de relojes no debe ser absoluta. Si dos procesos no interaccionan no es necesario que estén sincronizados puesto que la carencia de sincronización no sería observable y por tanto no podría provocar problemas.

   Desafortunadamente, la estrategia de sincronización de Lamport sólo es útil cuando importa el orden del procesamiento de una sucesión de eventos, hecho que no está ligado al tiempo.

2. **Sistemas activados por eventos y sistemas activados por el tiempo:** En un sistema de tiempo real **activado por eventos**, cuando ocurre un evento significativo en el mundo exterior, es detectado por algún sensor, lo que entonces provoca que la UCP tenga una interrupción. Los sistemas controlados por eventos están controlados por las interrupciones.

   El principal inconveniente de los sistemas activados por eventos es que pueden fallar bajo condiciones de carga pesada, es decir, cuando muchos eventos ocurran a la vez.

   Un diseño alternativo son los sistemas de tiempo real **activados por el tiempo**. En este tipo de sistemas ocurre una interrupción del reloj cada ΔT unidades de tiempo, a lo que se le denomina **marca de reloj**. En cada marca de reloj ciertos sensores (o todos) se muestrean y ciertos actores (o todos) se controlan. No hay más interrupciones que las marcas de reloj.

   La detección de un evento en estos sistemas ocurre en la primera marca del reloj posterior al evento, pero la carga de interrupciones no modificaría ni contaría para el problema, evitando la sobrecarga del sistema.

   La elección del valor de ΔT debe hacerse con precaución. Si ΔT es muy pequeño, el sistema tendrá muchas interrupciones del reloj y desperdiciará mucho tiempo durante las revisiones. Si es muy grande, los eventos cruciales no son notados sino más tarde. Además, la decisión acerca de los sensores que deben verificarse en cada marca de reloj, y cuáles verificar en otra marca de reloj, etc., son críticas. Por último algunos eventos podrían ser más cortos que una marca de reloj, por lo que deberán guardarse para no ser omitidos.

En resumen, los diseños activados por eventos dan respuesta rápida con carga baja, pero tienen mayor costo y probabilidad de fallar con carga alta. Los diseños activados por el tiempo tienen las propiedades opuestas y sólo son adecuados en un ambiente estático en donde se conozca mucho y se de antemano el comportamiento del sistema. Cuál de ellos es el mejor, depende de la aplicación.

3. **Predictibilidad:** Una propiedad inherente de los sistemas de tiempo real es la predictibilidad. Debe ser claro que el sistema cumple con todos sus tiempos límite, incluso con cargas pico. Los análisis estadísticos del comportamiento, suponiendo que los eventos son independientes, conducen con frecuencia a errores, pues existen correlaciones inesperadas entre los eventos. El comportamiento aleatorio rara vez ocurre en un sistema de tiempo real. Lo más frecuente es que, cuando se detecta el evento E, el proceso X se debe ejecutar, seguido por los procesos Y y Z, en ese orden o en paralelo. Además, con frecuencia se conoce (o debe conocerse) el comportamiento de estos procesos en el peor de los casos.

   Este tipo de razonamiento y modelado lleva a un sistema determinista, más que estocástico.

   Dado que se está proponiendo un algoritmo de tiempo real que trabaje *on line*, y el cumplimiento de plazos estrictos es uno de sus objetivos, el algoritmo no puede ocupar tiempo de procesamiento en procedimientos muy elaborados, ya que existe la posibilidad de afanar el tiempo a la actividad principal del algoritmo, que es la de ejecutar en su totalidad las tareas una vez que son aceptadas.

4. **Tolerancia a fallas:** La tolerancia a fallas es un aspecto opcional del diseño, ya que depende de los requerimientos del sistema. Muchos sistemas de tiempo real controlan dispositivos donde la seguridad es crítica, en vehículos, hospitales y plantas de energía, por lo que la tolerancia a fallas es con frecuencia un aspecto a considerar. Los protocolos de los esquemas de tolerancia a fallas no pueden ser extensos (ya que consumen tiempo) para que todos coincidan en todo, todo el tiempo.

   En un sistema donde la seguridad es crítica, es de particular importancia que el sistema pueda controlar el peor de los escenarios. Además, las fallas no siempre son independientes, cabe la posibilidad de que una falla origine otra y se de un efecto en cascada que termine el funcionamiento del sistema temporalmente. En consecuencia, los sistemas de tiempo real tolerantes de fallas deben poder enfrentar el número máximo de fallas y la carga máxima al mismo tiempo.

   Si el diseño del sistema siempre da espacio suficiente para evitar un desastre y si este se recupera de manera gradual, entonces, decimos que el sistema *es seguro contra las fallas.*

   Los modelos de fallas tienen diferentes niveles de severidad: $Canales \prec Crash \prec Omision \prec Bizantinos$; en el sentido en que dado un protocolo capaz de resolver el problema en un modelo de fallas, α, entonces éste es capaz de resolver un modelo menos severo β, es decir que $\beta \prec \alpha$ [20].

5. **Requisitos del lenguaje:** El lenguaje utilizado debe permitir expresar una tarea como una colección de tareas más pequeñas (procesos ligeros o hilos) que puedan planificarse de manera independiente, sujetos a la precedencia definida por el usuario y las restricciones de exclusión mutua.

   El lenguaje debe ser tal que permita calcular el tiempo máximo de ejecución de una tarea en el momento de su compilación. Este requisito implica que no deben usarse ciclos *while* generales. La iteración se realiza mediante ciclos *for* con parámetros constantes. La recursión tampoco está permitida. Incluso estas restricciones no son suficientes para calcular el tiempo de ejecución de las tareas de antemano, puesto que la falta de caché, el fallo de página y el robo de ciclos por canales DMA[1], por mencionar algunos, afectan el

[1] Los canales DMA (Direct Memory Access) son rutas del sistema usados por muchos dispositivos para transferir información directamente a la memoria en ambos sentidos.

desempeño.

Los lenguajes utilizados para la programación de estos sistemas deben trabajar con el tiempo mismo. Se debe disponer de una variable especial, *clock*, que contiene el tiempo actual en marcas de tiempo. Sin embargo hay que tener cuidado en las unidades en que se expresa el tiempo. Mientras más fina sea la resolución, más rápido tendrá la variable *clock* un desbordamiento.

El lenguaje debe tener forma de expresar los retrasos mínimo y máximo. El conocimiento de estos retrasos puede prevenir algún evento inesperado. Por ejemplo, si un proceso se bloquea en un semáforo durante un tiempo mayor a uno ya establecido, se debe hacer que expire y libere los recursos.

Características deseadas en un sistema de tiempo real

- *Timelines*. Los resultados no sólo tienen que ser correctos sino que también éstos deben darse dentro de sus restricciones de tiempo.
- *Diseño para carga máxima*. Los sistemas de tiempo real no deben colapsar cuando están sometidos a condiciones de carga máxima de trabajo, así que estos sistemas deben estar diseñados para manejar todos los escenarios (el sistema es completo).
- *Predictibilidad*. Para garantizar un mínimo nivel de desempeño, el sistema debe ser capaz de predecir las consecuencias de cualquier decisión de planificación.
- *Tolerancia a fallas*. Ni el hardware y tampoco el software deben ser las causas de que el sistema falle, se deben desarrollar las estrategias necesarias para evitar estos problemas.
- *Manutención*. La arquitectura de un sistema de tiempo real debería ser diseñada de acuerdo a una estructura modular para asegurar que las modificaciones se harán de una forma más sencilla.
- *Operación continua*. La mayoría de los sistemas distribuidos de tiempo real tienen que operar continuamente para mantener una operación normal del medio ambiente controlado y debe estar listo para realizar una acción en contra de una operación anormal.
- *Interacción con procesos asíncronos*. Los sistemas son desarrollados para interactuar con medios ambientes físicos, en donde los procesos están geográficamente distantes. Los procesos asíncronos se comunican mediante el paso de mensajes. Las secuencias de los eventos de procesos asíncronos son muy difíciles de predecir y las decisiones tomadas en la fase de diseño son usualmente violadas.
- *Reloj global y estado global*. Un sistema distribuido comprende de una colección de procesos que se comunican vía variable compartida o paso de mensajes a través de una red de comunicaciones. Los procesos que construyen al sistema pueden estar en ejecución en diferentes procesadores de manera paralela. Cada procesador puede tener su propio reloj, que es independiente de los demás. Es difícil determinar un tiempo global preciso, que es de suma importancia para determinar un estado global exacto y encontrar problemas debidos al tiempo.

1.3. Conceptos básicos

A continuación se presentan la terminología y los conceptos que serán utilizados a lo largo de este capítulo para comprender los algoritmos que se presentan en las secciones ulteriores.

Una entidad intrínseca de cualquier sistema operativo es el *proceso*. Un proceso[2] también es denominado *hilo de ejecución*, los procesos pueden estar compuestos por un conjunto finito de **tareas**. Una tarea es la ejecución secuencial de código que no se suspende a sí misma, a su vez éstas se subdividen en unidades más pequeñas de trabajo relacionado denominadas **jobs**, **instancias de ejecución** o subtareas.

De manera informal, cuando un sólo procesador tiene que ejecutar un conjunto de tareas concurrentes, esto es, que traslapan en tiempo su ejecución y no hay dependencia entre tareas, la UCP tiene que ser asignada a varias tareas de acuerdo a un criterio predefinido, llamado *política de planificación*. El conjunto de reglas que, a cualquier momento, determinan el orden en que las tareas son ejecutadas es llamado *algoritmo de planificación*.

Por lo tanto, las tareas que podrían ejecutarse en la UCP pueden estar en ejecución, si éstas han sido elegidas por el algoritmo de planificación, o bien en espera de ser asignadas a la UCP, cuando hay otra tarea actualmente en ejecución. Una tarea que está haciendo uso actualmente del procesador es llamada tarea en *ejecución*. Una tarea que se encuentra en espera de ser asignada al procesador es llamada una tarea *lista* para ser ejecutada; por último, una tarea *activa* es aquella tarea que ha sido colocada en la cola de tareas listas. Todas las tareas listas para ejecución se mantienen en una cola. En los sistemas operativos, dependiendo del criterio que éstos ocupen, pueden haber varias de estas colas, la imagen 1.1 muestra las transiciones principales que una tarea puede tener a lo largo de su ejecución en el sistema.

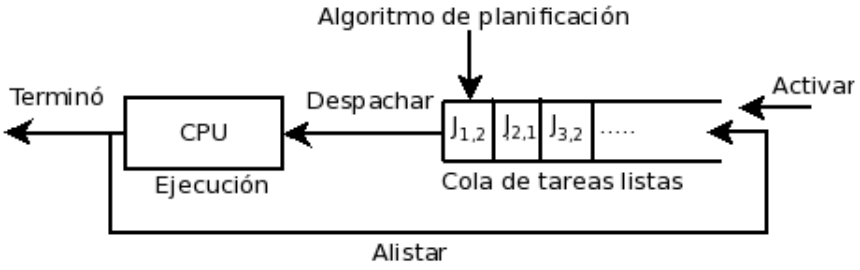

Figura 1.1: Esquema de una cola que contiene las transiciones que una tarea posiblemente sufra durante su ejecución.

En muchos sistemas operativos que permiten la activación dinámica de tareas, aquella tarea en ejecución puede ser *suspendida* temporalmente a cualquier momento, por ende si una tarea de mayor importancia necesita ser ejecutada por el sistema puede hacer uso del procesador inmediatamente. En este caso, la tarea que está en ejecución se interrumpe y se coloca en una cola de tareas listas, mientras que la UCP es asignada a la tarea de la cola de tareas listas más importante que acaba de llegar al sistema. La operación de suspender la tarea en ejecución y colocarla en la cola de listas es llamada **apropiación** (*preemption*, en inglés). En los sistemas de tiempo real la apropiación es importante por tres razones [47]:

1. El manejo de excepciones para el desempeño de las tareas puede que necesite adelantarse a las tareas existentes, así que las respuestas a las excepciones deben ser expedidas de manera oportuna.

2. Cuando las tareas de una aplicación tienen diferentes niveles de importancia, la apropiación permite que las tareas de mayor importancia sean adelantadas.

3. La eficiencia en la planificación puede ser producida para mejorar la capacidad de respuesta del sistema.

[2]Interacción entre el procesador y la memoria, en la cual el procesador es instruido para ejecutar una secuencia de instrucciones sobre un conjunto de datos.

Aunque, también impone ciertas restricciones en la implementación:

1. La apropiación incrementa el cambio de contexto entre tareas,
2. La apropiación no siempre es válida. Existe la posibilidad de que no se pueda detener una tarea para adelantar otra de mayor prioridad, inclusive existen algoritmos que abordan tareas en las que existen secciones no apropiativas [34], etc.

1.3.1. Restricciones

Los sistemas de tiempo real requieren de algoritmos que especifiquen la tarea que debe ejecutarse en cierto instante de tiempo. Para la definición de un algoritmo se deben especificar restricciones, algunas de ellas reducen la complejidad del problema de planificación, algunas restricciones consisten en estrategias como restringir la planificación a un sólo procesador, planificar sólo tareas apropiativas, usar prioridades fijas, remover la precedencia, remover las restricciones de recursos, asumir la activación simultánea de tareas, considerar conjuntos homogéneos de tareas (conjuntos con sólo tareas periódicas o sólo tareas aperiódicas, etc.), veánse [53, 54, 35, 28].

Son básicamente cuatro tipos de restricciones a considerar:

1. Restricciones impuestas por los sistemas operativos,
2. Restricciones de tiempo,
3. Relaciones de precedencia y
4. Restricciones de exclusión mutua en recursos compartidos.

Restricciones impuestas por sistemas operativos

La mayoría de los sistemas de tiempo real usados para el control de aplicaciones se basan en kernels, los cuales son versiones modificadas de sistemas operativos *timesharing*[3]. Como consecuencia, tienen las mismas características básicas encontradas en los sistemas timesharing. Las principales características de tales sistemas son:

- *Multitasking.* Provee de soporte para programación concurrente, a través de un conjunto de llamadas a sistema.
- *Planificación basada en prioridad.* Consiste en implementar múltiples algoritmos para la administración de los instantes de tiempo en los que se hará la asignación del procesador a los procesos.
- *Responder rápidamente a interrupciones externas.* Esta característica puede provocar retrasos en la ejecución de eventos, ya que las prioridades para atender a las interrupciones son más altas que las de los procesos.

[3]Un sistema operativo multiusuario es un sistema operativo que tiene la capacidad de permitir que muchos usuarios compartan simultáneamente los recursos de proceso de la computadora. Centenas o millares de usuarios se pueden conectar a la computadora que asigna un tiempo de computador a cada usuario, de modo que a medida que se libera la tarea de un usuario, se realiza la tarea del siguiente, y así sucesivamente. Dada la alta velocidad de transferencia de las operaciones, la sensación es de que todos los usuarios están conectados simultáneamente a la UCP con cada usuario recibiendo únicamente un tiempo de máquina.

- *Comunicación y sincronización entre procesos.* El mecanismo más conocido son los semáforos; aunque, su uso no es trivial y puede generar problemas como la inversión de prioridades, bloqueo encadenado, deadlocks los cuales pueden producir retrasos indefinidos.
- *Rápido cambio de contexto y un kernel pequeño.* El rápido *switching* ayuda a reducir el *overhead*[4] del sistema, mejorando el tiempo de respuesta del procesamiento de un conjunto de tareas. Mientras que, un kernel pequeño implica funcionalidad limitada y esto a su vez afecta a la predictibilidad del sistema.
- *Soporte de un reloj de tiempo real.* Es un mecanismo para administrar el tiempo del sistema, aunque en muchas de las veces no existen primitivas para especificar las restricciones de tiempo de una tarea como los plazos.

Restricciones de tiempo

Una restricción de tiempo típica en una tarea de tiempo real es el plazo, el cual representa el tiempo en el que dicha tarea debe de completar su ejecución sin causar algún daño al sistema. Si un plazo es especificado con respecto a su tiempo de activación, entonces se le llama *plazo relativo*, mientras que si este es definido con respecto al tiempo cero es llamado *plazo absoluto*. Dependiendo de las consecuencias de perder un plazo, es común dividir las restricciones de tiempo en dos clases: flexibles (*soft deadlines*) y estrictos (*hard deadlines*).

Dependiendo de las consecuencias de perder un plazo, un sistema de tiempo real es usualmente distinguido en dos clases:

Estrictos. Se dice que un sistema de tiempo real es estricto, si el usuario requiere de la validación de que el sistema siempre satisface sus restricciones de tiempo. La validación se refiere a un procedimiento probablemente correcto y eficiente que ha sido exhaustivamente simulado y probado.

Flexibles. Si no se requiere de validación, sólo de una prueba de que la tarea satisface algunas restricciones estadísticas.

En general, una subtarea, J_i, que se ejecuta bajo un esquema de tiempo real puede ser caracterizada por un subconjunto de los siguientes parámetros (vea la figura 1.2):

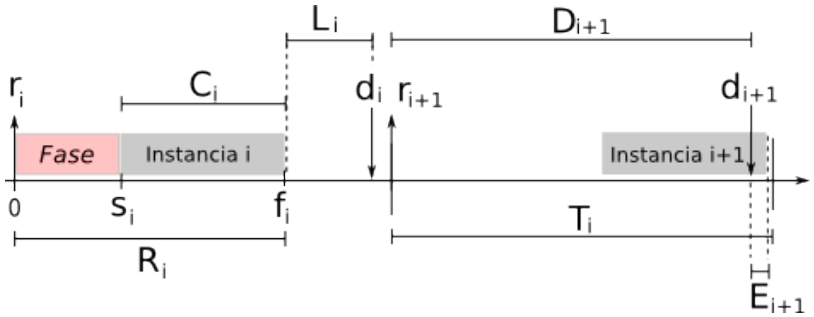

Figura 1.2: Parametros de tareas de tiempo real.

- **Tiempo de activación** r_i: es el tiempo en el que una tarea se encuentra lista para su ejecución; también, se puede referir a ésta como tiempo de liberación (*release time*).

[4]Es la cantidad de tiempo de procesamiento usada por un sistema de software, tal como los sistemas operativos para cargar todos los recursos necesarios antes de ser ejecutados.

- **Consumo** C_i: es el tiempo de procesador necesario para ejecutar una tarea.
- **Periodo** T_i: es lapso de tiempo entre liberaciones sucesivas de instancias de una misma tarea.
- **Plazo absoluto** d_i: es el instante de tiempo en el que la instancia de una tarea debe terminar.
- **Plazo relativo** D_i: es el tiempo de respuesta máxima permitido para ejecutar la instancia de una tarea, o bien, es la diferencia entre el plazo absoluto y el tiempo de activación: $D_i = d_i - r_i$.
- **Tiempo de inicio** s_i: es el tiempo en el que una tarea comienza con su ejecución.
- **Tiempo de termino** f_i: es el tiempo en el que una tarea termina su ejecución.
- **Tiempo de respuesta** R_i: es la diferencia entre el tiempo de termino y el tiempo de activación: $R_i = f_i - r_i$.
- **Latencia** L_i: $L_i = f_i - d_i$ representa el retraso de que una tarea sea completada con respecto a su plazo; note que si una tarea se termina antes de su plazo, su latencia es negativa.
- **Laxitud** o *excedente* E_i: $E_i = max(0, L_i)$ es el tiempo en el que una tarea permanece activa aún después de concluir con su plazo.
- **Fase** ϕ_i: es el tiempo de activación de la primer instancia periódica (*phase* en inglés). Si ϕ_i es la fase de la tarea periódica τ_i, el tiempo de activación de la k-ésima instancia está dado por $\phi_i + (k-1)T_i$, donde T_i es llamado el *periodo* de la tarea.

A lo largo de los capítulos subsecuentes se hará uso de la notación recientemente definida.

Dependiendo de la regularidad de activación de una tarea se clasifican en:

Periódicas. Consisten de una secuencia infinita de actividades idénticas, llamadas instancias, que son normalmente activadas en proporciones constantes de tiempo. A este tipo de tareas se les representará por τ_i.

Aperiódicas. Son tareas que tienen plazos flexibles o carecen de estos. Consisten de una secuencia de instancias, que puede ser infinita, en el sentido de que tienen el mismo comportamiento estadístico y los mismos requisitos de tiempo; sin embargo, sus tiempos de activación no se realizan de manera regular.

Esporádicas. Son tareas que son activadas en instantes de tiempo aleatorios, de acuerdo a una función de probabilidad, y tienen plazos estrictos.

Relaciones de precedencia

En ciertas aplicaciones, las actividades de cómputo no pueden ejecutarse siguiendo un orden arbitrario ya que tienen que respetar algunas relaciones de precedencia definidas en la etapa de diseño [31]. Dichas precedencias suelen representarse en un grafo dirigido acíclico, donde las tareas son representadas por nodos y las relaciones de precedencia por flechas. La precedencia en el grafo induce a un orden parcial en el conjunto de tareas.

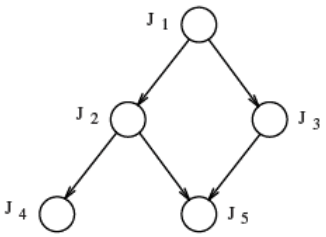

Figura 1.3: Relación de precedencia entre cinco tareas.

Figura 1.4: Una posible secuencia de ejecución de las cinco tareas.

Como ejemplo, vea la figura 1.3, ésta muestra un grafo acíclico dirigido que describe las relaciones de precedencia entre cinco tareas (J_i tal que $i = 1, \ldots, 5$). De la estructura del grafo se observa que la tarea J_1 es la única tarea que se puede comenzar a ejecutar ya que ésta no tiene predecesores. Al terminar la ejecución de J_1, tanto J_2 como J_3 pueden comenzar. La tarea J_4 sólo puede comenzar su ejecución cuando J_2 se ha terminado, mientras que J_5 debe esperar a que terminen J_2 y J_3. Una posible secuencia de ejecución de dicho conjunto de tareas se muestra en la figura 1.4.

Restricciones de exclusión mutua en recursos compartidos

Para asegurar la *consistencia de datos* en recursos compartidos, cualquier sistema concurrente debe usar los protocolos adecuados para el acceso a dichos recursos para garantizar la *exclusión mutua* cuando surgen las condiciones de competencia entre tareas. Una pieza de código que se ejecuta garantizando la exclusión mutua es conocida como *sección crítica*. Generalmente los recursos compartidos no permiten accesos simultáneos pero requiere de exclusión mutua para que concluya con su ejecución un conjunto de tareas. Para asegurar acceso secuencial sobre recursos mutuamente excluyentes, los sistemas operativos usualmente proveen de mecanismos de sincronización que pueden ser usados para generar las secciones críticas.

Si una tarea se encuentra en espera de un recurso que es mutuamente excluyente, se dice que la tarea está bloqueada en dicho recurso. Todas las tareas bloqueadas en el mismo recurso se mantienen en una cola asociada al semáforo que protege a dicho recurso. Cuando una tarea en ejecución activa la primitiva[5] *wait* sobre un semáforo bloqueado, éste entra en un estado de espera, hasta que otro proceso ejecute la primitiva *signal* que desbloquea al semáforo. Cuando una tarea abandona el estado de espera, éste lo hace no yendo al estado de ejecución, pero si al estado de tareas listas, y de esta manera la UCP puede ser asignado a la tarea de mayor prioridad por el algoritmo de planificación.

1.4. El problema de planificación

Para definir un problema de planificación es necesario definir tres conjuntos: un conjunto de n tareas $\Gamma = \{J_1, \ldots, J_n\}$, un conjunto de m procesadores $P = \{P_1, \ldots, P_m\}$ y un conjunto de r tipos de recursos $R = \{R_1, \ldots, R_r\}$. Además, pueden ser especificadas las relaciones de

[5]Instrucción atómica y básica de algún lenguaje de programación sobre la cual es posible implementar instrucciones más complejas.

precedencia entre tareas y no olvidemos las restricciones de tiempo asociadas con cada tarea. Por lo tanto, planificar significa asignar los recursos R y procesadores P a las tareas de Γ con el objetivo de completar todas las tareas bajo las restricciones impuestas. Se ha probado que este problema, en su forma general[6], es computacionalmente intratable por lo que entra dentro de la categoría de los problemas *NP-completo* [19, 23, 32].

Definición: Se dirá que un algoritmo A es **factible** sólo cuando un conjunto de tareas Γ, dado como entrada a dicho algoritmo, haya completado su ejecución respetando las restricciones de tiempo previamente impuestas.

Entre los algoritmos clásicos para calendarizar o planear los eventos, en este caso tareas, se pueden identificar las siguientes clases de algoritmos principales:

Apropiativos. Las tareas pueden ser interrumpidas en cualquier momento para asignar el procesador a otra tarea *lista* para ejecución de mayor prioridad, de acuerdo a la política de planificación. Este termino proviene del inglés *preemptive*.

No apropiativos. El procesamiento es del tipo *batch*. Una vez que la instancia de una tarea se comienza a ejecutar bajo esta política, es ejecutada hasta su culminación. En este caso, todas las decisiones se toman hasta que la instancia termina su ejecución.

Fuera de línea. Un algoritmo es *off line* si éste es ejecutado en el conjunto entero de tareas antes de activar cualquier tarea. La calendarización (planificación) generada bajo este esquema es almacenada en una tabla y después ejecutada por un despachador[7].

En línea. Decimos que un algoritmo de planificación es usado *on line* si las decisiones de planificación se ejecutan durante la ejecución, cada momento que una tarea entre o cuando la ejecución de una tarea termine.

Dinámicos. Son aquellos en los que las decisiones de planificación se basan en parámetros dinámicos, que pueden cambiar durante la evolución del sistema.

Estáticos. Las decisiones de planificación se basan en parámetros fijos, asignados a las tareas antes de su activación.

Óptimos. Un algoritmo es óptimo si minimiza (o maximiza) una función de costo definida sobre el conjunto de tareas. Cuando no se define función de costo y lo único que se desea lograr es una calendarización factible, entonces un algoritmo es óptimo si éste siempre encuentra una calendarización siempre que exista una.

Heurísticos. Si la búsqueda de una planificación factible se basa en el uso de una función (heurística); sin embargo, éstas no garantizan una planificación óptima.

De las clases presentadas, nos enfocaremos principalmente en aquellos algoritmos en los que las decisiones para la selección de la tarea de mayor prioridad, se toman al mismo tiempo que la planificación está en curso (algoritmos ejecutados en línea); sin embargo, a continuación se presenta una breve introducción de algunos algoritmos de planificación que generan una calendarización antes de comenzar la ejecución de un conjunto de tareas (algoritmos ejecutados fuera de línea).

[6] Cuando la apropiación no es permitida y las tareas pueden tener tiempos de llegada arbitrarios.
[7] Es el criterio encargado de asignar la UCP a la tarea elegida por el algoritmo de planificación.

1.4.1. Algoritmos fuera de línea

El problema de planificar tareas sobre múltiples procesadores puede ser visto como la búsqueda de una calendarización factible de tareas. Usualmente, la factibilidad de la calendarización se define en términos del cumplimiento de restricciones (plazos cumplidos, dependencias de tareas, por mencionar algunos). En los algoritmos ejecutados fuera de línea (off line) el espacio de búsqueda puede ser representado como un *árbol de búsqueda* (algoritmo de Bratley [5]), donde cada nodo del árbol representa la asignación de una tarea a un procesador. Cualquier ruta, de la raíz a una hoja del árbol es una *calendarización*, aunque no necesariamente es factible. Una *planificación parcial* es la ruta que se forma entre cualquier par de nodos del árbol.

Una vez hecha la asignación de las tareas en cada uno de los nodos del árbol, el algoritmo recorre dicho árbol mediante búsquedas en profundidad, y se podan todas aquellas ramas que no ofrecen una planificación factible (no cumplen las restricciones especificadas). Cuando una búsqueda finaliza, cada posible recorrido representa una posible planificación. En los casos en los que hay más de una planificación factible, el objetivo es encontrar la planificación óptima, donde la optimalidad se define como una función que debe ser maximizada o minimizada (la carga promedio, la máxima carga de trabajo, reducir el tiempo de ejecución, disminuir la latencia de ejecución, etc.).

La búsqueda en el árbol puede ser categorizada entre búsqueda basada en la *asignación* o en *secuencias* [17], dependiendo del significado que tenga una arista en el árbol (como precedencia, costo, etc.).

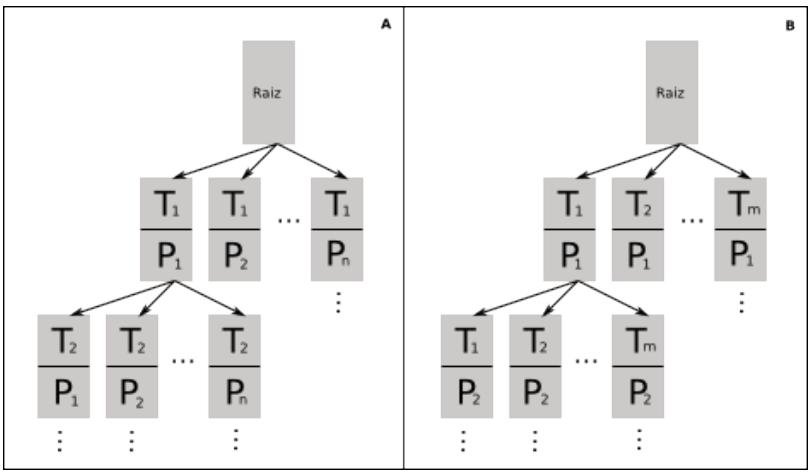

Figura 1.5: Algoritmos de búsqueda sobre árboles: A) búsqueda basada en asignación y B) búsqueda basada en secuencias.

Tanto en la búsqueda basada en asignación como en la basada en secuencias se recorre el árbol en profundidad, en búsqueda de una planificación factible, pero con la desventaja de que en ambos se realiza en tiempo secuencial. Ambos algoritmos producen resultados idénticos. Se ha probado que la aproximación basada en secuencias no es escalable [17], esto es debido a que durante cada iteración, el algoritmo trata de elegir una tarea para ejecutarla en el siguiente procesador disponible, un algoritmo basado en secuencias se enfoca inherentemente a la distribución de carga; por el otro lado, los algoritmo basados en asignación siempre buscan la tarea con plazo próximo a caducar.

1.4.2. Algoritmos de planificación en línea

La lista de algoritmos que a continuación se detalla, hacen referencia a todas esas estratégias cuyas decisiones de planificación se toman al vuelo durante la ejecución del algoritmo.

Planificación de tareas periódicas

La existencia de tareas periódicas se debe a la necesidad de los sistemas para emitir una respuesta, adquirir y procesar datos con regularidad. Cuando una aplicación consiste de varias tareas concurrentes con restricciones temporales se debe garantizar que cada tarea satisfaga sus restricciones. Básicamente las estrategias de planificación reparten el tiempo de procesador de tal forma que se satisfagan las restricciones temporales.

En esta sección se revisan tres técnicas de planificación de tareas periódicas: *tasa monótona*, *el plazo próximo primero* y *límite de tiempo monótono*. Los algoritmos son basados en prioridad lo que quiere decir que el procesador se mantiene ocioso sólo cuando ninguna tarea en ejecución requiere de dicho recurso. Los algoritmos basados en prioridad pueden ser dinámicos o estáticos. En un algoritmo estático todas las tareas se dividen en subconjuntos. Cada subconjunto es asignado a un procesador, y las tareas en cada procesador son planificadas por si mismas, pero tomando en cuenta las restricciones de dependencia. En contraste, en un algoritmo dinámico, las tareas listas para ejecución son colocadas en una cola y son despachadas a los procesadores para su ejecución tan pronto como los procesadores queden disponibles.

Las técnicas mencionadas son ampliamente utilizadas en los sistemas de tiempo real, ya que por medio de estos algoritmos es posible *especificar un criterio para el uso de los recursos* disponibles en un sistema. Los algoritmos de planificación son indispensables ya que estos permiten la utilización racional de los recursos, y en ocasiones con las condiciones apropiadas es posible lograr soluciones óptimas. Desgraciadamente no es posible establecer reglas que definan cuando se darán las condiciones óptimas para implementar un enfoque especifico y aprovechar los recursos en su totalidad, todos los algoritmos que se listan adelante forman parte de una categoría denominada *greedy* [10], en la que se trata de construir una solución óptima a partir de decisiones locales.

Tasa monótona El algoritmo de tasa monótona o mejor conocido en inglés como *rate monotonic* [35], al que también nos referiremos como RM, se ejecuta bajo las siguientes suposiciones:

1. Las tareas son totalmente apropiativas y, además, el intercambio entre tareas tiene un costo insignificante.
2. Sólo los requerimientos de procesamiento son significantes; memoria, I/O y otros requerimientos son despreciables.
3. Todas las tareas son independientes; no hay relaciones de precedencia.
4. Todas las tareas son periódicas.
5. Los plazos relativos de las tareas son iguales a sus periodos.

En este algoritmo de planificación, las prioridades son fijas ya que se asignan en base al periodo: la instancia con el ciclo de duración más corto es la instancia de mayor prioridad. Es decir, para un conjunto finito de tareas periódicas $\Gamma = \{\tau_i : i = 1, \ldots, N\}$ cuyos periodos T_i cumplen que $T_1 < T_2 < \ldots < T_N$, la tarea con identificador $i = 1$ es la de más alta prioridad.

Para verificar si un conjunto de tareas es calendarizable bajo este algoritmo, existe una prueba de planificabilidad, que consiste en verificar que la utilidad total de las tareas no es mayor que $N(2^{1/N} - 1)$:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} < N(2^{1/N} - 1) \leq 1$$

donde N es el número de tareas a ser planificadas, si cumple con tal restricción entonces el algoritmo RM podrá planificar todas las tareas y, por ende, satisfacer sus respectivos plazos. Esta es una condición suficiente pero no necesaria [6]. Lo que quiere decir que habrá conjuntos de tareas cuya utilidad total rebase la cota establecida de $N(2^{1/N} - 1) < 1$ pero que podrán ser planificadas por RM.

El algoritmo se lista a continuación bajo la suposición de que el conjunto de tareas pasó la prueba de planificabilidad:

1. Al inicio del algoritmo (cuando $t = 0$) cada tarea periódica, $\tau_i \in \Gamma$ ($i = 1, \ldots, N$ indica el identificador de la tarea), envía una instancia de ejecución a una cola de tareas, cuyo estado en el sistema operativo es el de *activa* (vea figura 1.1), en donde esperarán a ser ejecutadas.

2. Cuando las instancias llegan a la cola se ordenan de acuerdo a su periodo. En caso de haber tareas con el mismo periodo, romper el empate insertando las instancias de manera arbitraria. Una vez que la ejecución de la instancia j de la tarea i, $\tau_{i,j}$, haya concluido en tiempo t, la tarea τ_i esperará $T_i - t$ para enviar la siguiente instancia.

3. Si no hay tareas activas, el algoritmo ha terminado. Si aún hay tareas activas, entonces regresamos al punto anterior.

Nótese que habrá momentos en el tiempo en los que la UCP estará ociosa, por lo que regularmente se implementa algún mecanismo para aprovechar ese tiempo, un ejemplo es el procesamiento en segundo plano (véase la sección B.1.1 de la página 74).

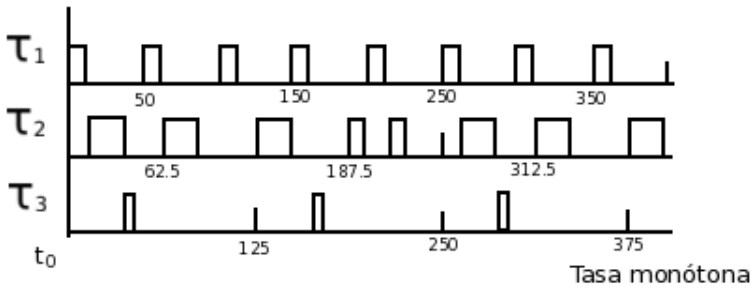

Figura 1.6: Diagrama de Gantt para el conjunto de tareas planificables $\Gamma = \{\tau_1 = (50, 12.5), \tau_2 = (62.5, 25) y \tau_3 = (125, 10)\}$.

La figura 1.6 muestra tres tareas periódicas, y las representaremos como $\tau_i = (T_i, C_i)$, en donde T_i y C_i son el periodo y el valor del tiempo de consumo para la tarea i, respectivamente. Las tareas son $\tau_1 = (50, 12.5)$, $\tau_2 = (62.5, 25)$ y $\tau_3 = (125, 10)$, todas comienzan en $t_0 = 0$. Como puede observarse la tarea τ_1 posee la prioridad mas alta, en seguida tenemos a τ_2 y por último a τ_3. Para este caso en particular la tarea τ_1 no se suspende en ningún momento, cada tarea comienza su ejecución cada que comienza un nuevo periodo. La tarea τ_2 debe suspender su ejecución en $t = 200$, ya que es de menor prioridad que τ_1, y continúa hasta $t = 212.5$, que es cuando τ_1 termina con la ejecución de una de sus instancias. La tarea τ_3 debe esperar 37.5 unidades de tiempo antes de poder ejecutarse la primera vez, nótese que la ejecución depende

del instante en el que las tareas τ_1 y τ_2 liberan el procesador. El análisis de planificabilidad para este conjunto de tareas periódicas arrojó lo siguiente $U = \Sigma_{i=1}^{3} \frac{C_i}{T_i} = 0.73 \leq n(2^{1/n} - 1) = 0.779$.

Límite de tiempo monótono El algoritmo de límite de tiempo monótono (*deadline monotonic* en inglés) [33], al que nos referiremos como DM, se ejecuta bajo las siguientes suposiciones:

1. Las tareas son totalmente apropiativas y el costo de dicha propiedad es insignificante.
2. Sólo los requerimientos de procesamiento son significantes; memoria, I/O y otros requerimientos son despreciables.
3. Todas las tareas son independientes; no hay relaciones de precedencia.
4. Todas las tareas son periódicas.
5. Los periodos de las instancias de las tareas periódicas no tienen el mismo plazo relativo.

Esta política de planificación se lleva a cabo bajo prioridades fijas: la prioridad que se asigna a las tareas se basa en los *plazos relativos* de las mismas, esto es, la tarea con el plazo de terminación más temprano es el de mayor prioridad. El algoritmo DM es una extensión del algoritmo Rate Monotonic donde las tareas pueden tener plazos relativos menores que sus periodos.

Cuando el plazo relativo de cada tarea es proporcional a su periodo[8] el RM y el DM son idénticos. Cuando los plazos relativos son arbitrarios, el algoritmo DM tiene un mejor desempeño en el sentido de que éste a veces produce una planificación factible cuando RM falla.

Al igual que RM, es posible planificar un conjunto de tareas siempre que la utilidad total sea menor o igual a $n(2^{1/n} - 1)$.

El algoritmo se lista a continuación bajo la suposición de que el conjunto de tareas es planificable:

1. Al inicio del algoritmo (cuando $t = 0$) cada tarea periódica, $\tau_i \in \Gamma$ ($i = 1, \ldots, N$ indica el identificador de la tarea), envía la instancia j, $\tau_{i,j}$, a una cola de tareas, cuyo estado en el sistema operativo es el de *activa* (vea figura 1.1), en donde esperarán a ser ejecutadas.
2. Cuando las instancias llegan a la cola se ordenan de acuerdo a su plazo relativo ($D_i = d_i - r_i$, véase la sección 1.3.1). En caso de haber tareas con el mismo plazo relativo, romper el empate insertando las instancias de manera arbitraria. Una vez que la ejecución de la instancia j de la tarea i, $\tau_{i,j}$, haya concluido en tiempo t, la tarea τ_i esperará para enviar la siguiente instancia.
3. Si no hay tareas activas, el algoritmo ha terminado. Si aún hay tareas activas, entonces regresamos al punto anterior.

[8] Es decir, que los periodos de todas las tareas, excepto el periodo de la tarea con el periodo más corto, son iguales o múltiplos del periodo más corto, y además las instancias deben ser liberadas por las tareas ya sea que todos se liberan en $t = 0$ o bien de la siguiente manera, por los efectos que la fase tiene sobre los plazos: la tarea de más alta prioridad, τ_1, libera su primera instancia en $t = 0$, luego τ_2, la siguiente tarea de más alta prioridad, libera su primera instancia en $t = T_2$, y así sucesivamente hasta llegar a la N-ésiama tarea que libera su primer instancia en $t = (N - 1)T_N$.

El plazo temporal próximo primero El algoritmo del plazo temporal próximo primero (en inglés conocido como *earliest deadline first*) [35], al que nos referiremos como EDF, se ejecuta bajo las siguientes suposiciones:

1. Todas las instancias de una tarea τ_i tienen el mismo peor tiempo de ejecución C_i.
2. Las tareas son totalmente preemptive (apropiativas) y el costo de apropiar es insignificante.
3. Todas las tareas son independientes; no hay relaciones de precedencia.

Un procesador bajo los influjos del algoritmo EDF siempre ejecuta la tarea con *plazo absoluto* inmediato. EDF es un algoritmo de planificación dinámica; las prioridades de las tareas no están fijas pero cambian dependiendo de la cercanía de su plazo absoluto. EDF es igual a DM bajo ciertas condiciones ([24], p. 73). Ya que el plazo absoluto de una tarea periódica, τ_i, depende la instancia j, lo calculamos de la siguiente manera:

$$d_{i,j} = \phi_i + (j-1)T_i + D_i \tag{1.1}$$

donde ϕ_i es la cantidad de tiempo transcurrido desde la primera vez que la tarea τ_i es liberada hasta que ésta es ejecutada por primera vez (la fase); T_i, el valor del periodo y D_i su plazo relativo.

El algoritmo EDF asigna tareas dinámicamente a los procesadores, aunque la prioridad de cada tarea es fija. Este puede ser usado para planificar tanto tareas periódicas como aperiódicas.

Siempre que EDF sea aplicado a un conjunto de tareas periódicas, Γ, podemos garantizar la planificabilidad de Γ si y sólo si $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, donde T_i es el periodo (tiempo de activación entre cualesquiera dos instancias) de una tarea τ_i.

Listemos el algoritmo:

1. En cada comienzo de su nuevo respectivo periodo (cada múltiplo $k \in \mathbb{N} \cup \{0\}$ del periodo T_i), la tarea periódica τ_i envía la instancia $\tau_{i,j}$ a una cola, en donde esperará a ser ejecutada.
2. Cuando las instancias llegan a la cola se ordenan de acuerdo a su plazo absoluto (mediante la ecuación 1.1). En caso de haber empates, se rompen de manera arbitraria.
3. En caso de haber instancias listas, se toma una de la cola y es asignada a la UCP e ir al paso 1. En caso contrario terminar.

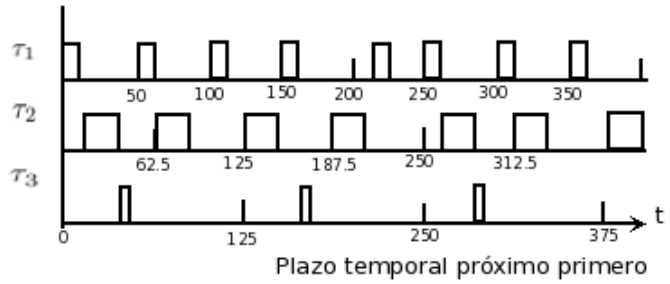

Figura 1.7: Diagrama de Gantt para el conjunto de tareas planificables $\Gamma = \{\tau_i : i = 1, 2, 3\}$: $\tau_1 = (50, 12.5)$, $\tau_2 = (62.5, 25)$ y $\tau_3 = (125, 10)$ bajo EDF.

El ejemplo de la figura 1.7 muestra la planificación generada por EDF de tres tareas. Las tareas τ_1, τ_2 y τ_3 tienen periodos $T_1 = 50$, $T_2 = 62.5$ y $T_3 = 125$ cuyos peores tiempos de

ejecución son $C_1 = 12.5$, $C_2 = 25$ y $C_3 = 10$. Todas las tareas periódicas comienzan su ejecución en $t = 0$ por lo tanto $\phi = 0$ en cada una de ellas. Obsérvese que las prioridades se van modificado en el tiempo de acuerdo al plazo absoluto de cada instancia.

1.5. Planificación de tareas en múltiples procesadores y sistemas distribuidos

Sistemas conformados por múltiples procesadores son conocidos como sistemas fuertemente acoplados. Esta característica denota la existencia de memoria física compartida en el sistema. En contraste, un sistema es llamado débilmente acoplado cuando carece de algún tipo de memoria física. En los sistemas fuertemente acoplados, el tiempo de comunicación entre procesadores (IPC) es muy pequeño, en comparación con el tiempo de procesamiento de una tarea, y puede ser ignorado. Esto es posible a que la comunicación se reduce a operaciones de lectura y escritura sobre la memoria compartida. Sin embargo, no ocurre lo mismo en sistemas distribuidos en donde el tiempo utilizado para la comunicación es comparable al tiempo de ejecución. Debido a esta razón, un sistema con múltiples procesadores puede usar un planificador *centralizado*; mientras que esto no sería factible en un sistema distribuido. Un planificador centralizado requeriría mantenimiento del estado de varias tareas en el sistema en una estructura de datos centralizada. Esto requeriría de varios procesadores en el sistema para actualizarlo cada vez que el estado de las tareas cambie y en consecuencia esto resultaría en altos costos de comunicación.

La planificación en tiempo real en sistemas distribuidos de múltiples procesadores consiste en dos subproblemas: asignación de tareas a los procesos y planificar las tareas en procesadores individuales. El problema de asignación de tareas se refiere a como partir un conjunto de tareas y después como asignarlas a los procesadores. El problema de asignación de tareas puede ser *dinámico* o *estático*:

- En el esquema de planificación estática, la asignación de tareas a los nodos es permanente y no cambia con el tiempo. El conjunto de tareas es dividido en varios subconjuntos, luego cada subconjunto es asignado a cada procesador.
- El esquema de asignación dinámica, las tareas son asignadas a los nodos conforme éstas son liberadas en el sistema. El conjunto de tareas listas para ser ejecutadas se encuentra en una estructura de datos común (una cola) del sistema y éstas son asignadas a los nodos conforme los procesadores se vuelven disponibles.

Por lo tanto, en el caso dinámico diferentes instancias de una misma tarea pueden ser asignadas a diferentes nodos (inclusive instancias de tareas periódicas). Después de una asignación adecuada de tareas a los procesadores, se puede considerar a las tareas en cada procesador de manera individual y con esto reducir el problema del procesamiento en un sólo procesador.

La asignación de tareas a múltiples procesadores en sistemas distribuidos es un problema *NP-duro* y el problema de determinar una solución óptima tiene complejidad exponencial. Así que la mayoría de los algoritmos desarrollados son algoritmos basados en funciones heurísticas.

1.5.1. Asignación de tareas a múltiples procesadores

En las siguientes secciones se revisan algunas estrategias de asignación estática de tareas de tiempo real a los procesadores. Estas estrategias de asignación no tienen como propósito

minimizar los costos de comunicación entre procesadores y por lo tanto su implementación no sería factible en un sistema distribuido, además estos algoritmos son centralizados [39].

Asignación estática de tareas

Algoritmo de distribución basado en balanceo de utilización Su nombre en inglés es *utilization balancing algorithm* [30]. Este algoritmo consiste en una estructura de datos centralizada (p. e. una cola) que mantiene las tareas en orden incremental, de acuerdo a su medida de utilización. Éste remueve tarea por tarea de la estructura de datos para luego enviarlas al nodo con menor utilidad cada vez. En un sistema perfectamente balanceado la utilidad de cada procesador es igual a la utilidad del resto de los nodos del sistema.

La función objetivo que se desea minimizar en este algoritmo es $\sum_{i=1}^{n} |u - u_i|$, donde u es el valor de la utilidad promedio de las utilidades de los procesadores en el sistema y u_i es el valor de la utilidad en el procesador i.

Este algoritmo es factible cuando el número de procesadores en el sistema es fijo. Este algoritmo conviene ser implementado cuando los procesadores, de manera independiente, utilizan el algoritmo EDF.

Algoritmo Next-Fit para tasa monótona En este algoritmo [39], un conjunto de tareas es dividido, así que cada subconjunto de tareas es asignado a un procesador que utiliza el algoritmo de tasa monótona para planificar. Este algoritmo trata de utilizar la menor cantidad de procesadores posibles. Este algoritmo a comparación del algoritmo de balanceo no requiere que el conjunto de procesadores sea dado de antemano ni predeterminado. Este clasifica a las diferentes tareas de acuerdo a su utilización. Uno o más procesadores son asignados a un tipo de tarea. La esencia de este algoritmo es agrupar a las tareas con utilidades similares para su ejecución.

Las tareas son clasificadas de acuerdo a su utilización mediante la siguiente política: si las tareas van a ser divididas en m clases, una tarea τ_i pertenece a la clase k, tal que $0 \leq k < m$, si y sólo si

$$(2^{\frac{1}{k+1}} - 1) < \frac{C_i}{T_i} \leq (2^{\frac{1}{k}} - 1)$$

Asignación dinámica de tareas

Asignación centralizada basada en subasta En este método [39], cada procesador mantiene dos tablas llamadas *tabla de estados* y *tabla de carga en el sistema*. La tabla de estados contiene información acerca de las tareas que un nodo debe ejecutar, incluyendo información del resto de los procesadores en el sistema. Con dicha información, es posible calcular la capacidad de computo excedente de los otros procesadores.

Este método funciona mediante ventanas de tiempo de igual duración. Al final de cada ventana, cada procesador transmite su tiempo restante de computo en la siguiente ventana de tiempo a todos los procesadores del sistema. Cada procesador que recibe la información actualiza su tabla de estados. Cuando una tarea es liberada en un nodo, el nodo primero verifica si la tarea puede ser ejecutada de manera interna. Si la tarea puede ser procesada, el nodo actualiza su estado. En caso contrario, busca en su tabla para determinar que nodo puede procesarla.

Mientras se realiza la búsqueda de un nodo capaz de atender a la tarea, el procesador consulta

su tabla de carga y determina en que nodos es posible asignar la tarea. Entonces el procesador envía una petición a estos nodos. Existe una probabilidad de que en el procesador al que se le envío la información haya cambiado su estado y se encuentre con alta carga de trabajo ahora.

El problema con esta información obsoleta puede ser sobrellevado considerando factores como la proximidad, conociendo la carga exacta de trabajo, etc.

Algoritmo del colega Su nombre en inglés es *buddy algorithm* [39], tiene como propósito sobrellevar la alta cantidad de comunicaciones del algoritmo de subasta.

En este algoritmo, un procesador puede estar en cualquiera de dos estados: *sobrecargado* o *subcargado*. Un procesador (P_i) tiene estatus subcargado si su utilización no sobrepasa un valor establecido, Th, $u_i < Th$. Un procesador se dice sobrecargado si éste está por encima de un valor ya establecido ($u_i \geq Th$).

En este algoritmo un procesador envía su información sólo cuando su estatus cambia de sobrecargado a subcargado o viceversa. Además, cuando ocurre el cambio de estado este envía su información a un subconjunto de nodos llamado el conjunto de colegas, generalmente este subconjunto contiene a los vecinos inmediatos.

1.6. Planificación de tareas coordinadas

Esta estrategia de planificación se suscita en una red *ad hoc* [8], donde los nodos pueden entrar o salir de ésta en cualquier momento. Además, existen diferentes procesos[9] en cada uno de los nodos que forman parte de la red y la interacción de estos procesos generan nuevas tareas (tareas coordinadas). La ejecución de estas nuevas tareas requiere de una configuración al vuelo (on line) de los nodos de la red durante la ejecución.

El protocolo detonado por la coordinación (Coordination Triggered Protocol) o CTP posibilita a los nodos que conforman al sistema de la capacidad de coordinarse con la finalidad de llevar a cabo una tarea compuesta por diferentes procesos distribuidos en los nodos de la red. Las principales funciones desempeñadas por CTP son [42]:

1. mapear los procesos con los nodos,
2. proveer los mecanismos, que requieren los nodos, para que la tarea coordinada pueda llevarse a cabo,
3. proveer de una infraestructura de comunicación,
4. monitorizar el estatus de la ejecución de una tarea coordinada y
5. garantizar que los nodos hayan liberado todos los recursos utilizados.

Si se modela una tarea coordinada como una red de procesos Khan [22] entonces se añade la restricción de que cada uno de los procesos son independientes, lo que conlleva a un análisis individual, por nodo, para garantizar la planificabilidad del sistema.

[9] Un proceso es visto como una tarea menor o muy específica con restricciones temporales definidas: deadline, periodo, tiempo de ejecución, tiempo de liberación y otras.

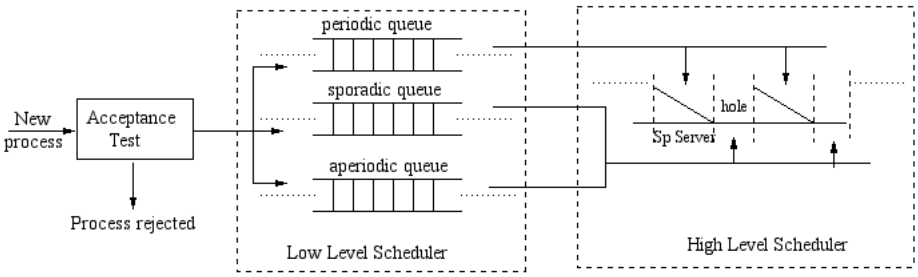

Figura 1.8: Arquitectura del planificador coordinado de tareas (imagen tomada del articulo [42]).

Como se observa en la imagen 1.8, cuando llega una petición a algún nodo del sistema con la finalidad de ejecutar una tarea coordinada, el nodo receptor ejecuta un test con el fin de determinar si puede permitirse la ejecución del nuevo proceso.

Si el test es aprobado, el proceso es colocado en la cola correspondiente, al tipo de tarea (periódica, aperiódica o esporádica). Luego cada cola es planificada de acuerdo a las políticas de planificación internas al nodo. Todo ello compone al planificador de bajo nivel; mientras tanto, el planificador de alto nivel se compone de las políticas con las que se determina la siguiente tarea de mayor prioridad en el nodo.

1.7. Consenso

El consenso es un proceso, esto es que hay un número finito de pasos cuyo propósito es el de alcanzar un acuerdo sobre algún valor. Cada uno de los procesos en una red comienza con un valor inicial de un tipo particular y se supone que eventualmente cada proceso en la red dará como salida el mismo valor. Se requiere que las salidas sean las mismas aún cuando las entradas sean valores arbitrarios. Un aspecto importante para lograr el consenso (aplicado a muchas situaciones prácticas) es que éste se debe lograr aún con la presencia de fallas.

Lograr el consenso en un sistema distribuido sin fallas no es un reto que requiera de un gran análisis para desarrollar un algoritmo que lo resuelva, basta con el intercambio de mensajes con la información necesaria para lograr un acuerdo [38]. En cambio, cuando se toma una decisión en presencia de fallas o si el consenso interacciona con un algoritmo de planificación, las cosas se tornan más interesantes.

Los acuerdos en el consenso no sólo permiten establecer el valor de algunas variables de importancia en los sistemas distribuidos. En la práctica el consenso es de suma importancia para lograr la consistencia en algunos sistemas (las bases de datos son un ejemplo, ya que existen copias de la base de datos dispersas y conectadas por una red, dichas copias se actualizan con los mismos datos generados con cada transacción efectuada en cualquiera de las copias); es un proceso que permite a sistemas multiagente establecer distancias y velocidades, o bien controlar el problema de lograr niveles justos de calidad de servicio (QoS) evitando condiciones de sobrecarga [21]; inclusive, el consenso se hace presente en los sistemas de seguridad conformados por cámaras de video [49], etcétera.

El consenso ha sido estudiado en base a diferentes tipos de modelos de sistemas, tales como los modelos de tiempo síncronos y asíncronos; bajo fallas bizantinas, crash failures y las fallas debidas al medio de comunicación; además de los esquemas de paso de mensajes y memoria compartida. En los sistemas asíncronos, el consenso se ve restringido por el resultado producido por Fischer, Lynch y Patterson (FLP), el cual asegura que es imposible alcanzar consenso de manera

determinista en un sistema asíncrono [16], aun con una caida. Sin embargo existen trabajos en los que se realizan ajustes que permiten alcanzar el consenso: utilizando la aleatoriedad [45], haciendo suposiciones de sincronía o del tiempo sobre los sistemas [14, 15], usando detectores de fallas [9], mediante oráculos [44] e imponiendo condiciones sobre las entradas [40].

Algunos de los problemas más conocidos que suelen surgir al tratar de lograr acuerdos son *atomic broadcast*, *group membership*, *leader election*, *atomic commitment*, entre otros. Para saber en que consisten estos problemas revise el apéndice A.

1.7.1. El problema de lograr consenso

Consideremos un conjunto de N procesos p_i, donde $i = 1, \ldots, N$, que se comunican mediante el paso de mensajes. A grandes rasgos, para alcanzar un consenso, con ausencia de fallas, cada proceso p_i comienza con un estado de indecisión y *propone* un valor v_i. Los procesos se comunican unos con otros intercambiando valores. Después de algún número de rondas cada proceso p_i obtiene los valores de los demás procesos, y aplicando un algoritmo como el de votación por mayoría el proceso p_i *decide* un valor para su variable de decisión v_i, entonces el proceso p_i entra a un estado de decisión, del cual no puede cambiar. Como todos los nodos utilizan los mismos valores y ejecutan el mismo algoritmo, todos deciden el mismo valor y llegan a un consenso.

Cualquier algoritmo de consenso debe cumplir los siguientes requisitos:

***Terminación*:** eventualmente cada proceso correcto obtiene un valor para la variable de decisión.

***Acuerdo*:** el valor de decisión de todos los procesos correctos es el mismo; es decir, si p_i y p_j son correctos y han decidido un valor, entonces $v_i = v_j$ $(i, j = 1, 2, \ldots, N \ y \ j \neq i)$.

***Validez o integridad*:** si un proceso decide v, entonces v fue propuesto por alguno de los procesos.

1.8. Modelos de tiempo

Bajo el esquema del tiempo los sistemas se clasifican como síncronos y asíncronos. Los sistemas síncronos cumplen con las siguientes características:

- Se asume que todos los mensajes tienen un límite superior en el retardo de los mensajes, tomando en cuenta, el envío, transporte y recepción, sobre un canal.
- Todos los nodos tienen un reloj con una tasa acotada con deriva $\rho \geq 0$ con respecto al tiempo real.
- Hay un conocimiento de la frontera de tiempo superior e inferior que un proceso requiere para ejecutar un paso.
- Los relojes aproximadamente sincronizados, pueden servir para simular relojes perfectamente sincronizados.
- Con un retardo en los mensajes, y los relojes aproximadamente sincronizados, se puede implementar un modelo síncrono por rondas.

En cambio, un sistema es asíncrono si posee las siguientes características:

- No tiene límite el retardo de cada mensaje.
- Existe un desvío en el reloj.
- No hay un tiempo necesario para ejecutar un paso en el programa.

1.9. Comentarios del capítulo

La información presentada en éste capítulo da un panorama general de las principales consideraciones tomadas para el diseño del algoritmo definido en este trabajo de tesis y presentado en el capitulo 3. El algoritmo ha sido diseñado para un sistema distribuido en tiempo real, por lo que se repartirá un conjunto de tareas con restricciones temporales entre un conjunto de nodos heterogéneos, que satisfacerán las restricciones de las tareas aceptadas dentro de cada uno.

Aunque los algoritmos mostrados en la sección 1.4.2 y en el apéndice B pueden ser implementados para planificar el medio de comunicación y otros recursos, este trabajo sólo se enfoca en el uso de la UCP. El algoritmo desarrollado no se limita a alguna estrategia de planificación dinámica en especifico.

Para efectos de este trabajo, dentro de las características del diseño de sistemas en tiempo real (listadas en la sección 1.2.1) se ignorarán aspectos como la sincronización de relojes y, en su lugar, se supondrá que los nodos ya están sincronizados; tampoco se considerarán aspectos relacionados con tolerancia a fallas y, de igual manera, se supondrá que los nodos y medios de comunicación son confiables; no hay duplicidad de mensajes o cualquier otro tipo de fallas. El algoritmo se limita a la planificación de tareas independientes, totalmente apropiativas cuyos tiempos de cambio de contexto son nulos (condiciones que no son ciertas en la implementación de un verdadero sistema de tempo real).

Uno de los aspectos que forman parte de la planificación distribuida, la distribución de las tareas entre los nodos que conforman al sistema. Sin embargo, ninguna de ellas es utilizada en este trabajo, ya que como se explicará más adelante, la estrategia propuesta distribuye las tareas de manera aleatoria y de manera repetida entre los nodos.

Capítulo 2

Planificación global de tareas confinadas basada en consenso

En el presente capítulo se presenta la contribución de este trabajo, una estrategia nueva de planificación de tareas, *on line*, distribuida que trabaja de manera local; a dicha estrategia se le referirá como "planificación global de tareas confinadas basada en consenso" o *PGC*.

2.1. Introducción

Los sistemas de tiempo real son sistemas cuyo desempeño depende del resultado dado así como del cumplimiento las restricciones de tiempo definidas.

Los sistemas de tiempo real suelen clasificarse en dos categorías con respecto a las restricciones temporales: sistemas *estrictos* y sistemas *flexibles* [6, 36]. Los sistemas de tiempo real estrictos son susceptibles a daños cuando la ejecución de una tarea no ha sido completada antes de su *deadline* o restricción temporal. Por otro lado, los sistemas de tiempo real flexibles provocan degradación en el desempeño cuando una tarea no se ejecuta dentro del límite de tiempo, pero es capaz de continuar con la operación del sistema.

En los sistemas de tiempo real, el desempeño de una tarea ejecutada es evaluada como un nivel de *QoS* (calidad de servicio). Generalmente, los niveles de *QoS* dependen de los recursos asignados a una tarea tales como la utilización de la UCP (unidad central de procesamiento), ancho de banda de la red, memoria, etcétera. Lo que significa que se requiere asignar más recursos a las tareas para así mejorar sus niveles de calidad de servicio. Sin embargo, el mejorar los niveles de *QoS* de algunas las tareas llega a causar condiciones de sobrecarga.

El trabajo propuesto en [21] considera una aplicación del problema de consenso modelado con funciones no lineales para garantizar un control justo de la calidad del servicio, *QoS*, para sistemas de tiempo real con restricciones flexibles. Se propone un algoritmo adaptativo para la asignación de recursos con un controlador basado en agentes para lograr niveles equitativos de calidad del servicio. Se modela el controlador como un sistema multiagente donde cada agente tiene la información de algún recurso del sistema y un nivel de *QoS* de cada tarea, y actualiza la información de su recurso de acuerdo al protocolo de consenso. La desventaja de este modelo es que no es posible garantizar, en todo momento, la ejecución de las tareas en el sistema dentro de sus restricciones temporales, ya que su labor principal es la de lograr un uso equitativo de los recursos que el sistema provee entre los procesos ahí presentes.

En este trabajo se utiliza el consenso para lograr un acuerdo *en linea* que permita decidir el nodo apropiado para ejecutar una tarea justo después del instante en el que ésta fue liberada. Una vez aceptada la tarea en el nodo se garantiza su ejecución mediante un análisis que permite saber si dentro del nodo existe la cantidad de tiempo necesaria tomando en consideración la velocidad de los procesadores en cada nodo involucrado.

La mayoría de las aplicaciones de tiempo real son implementadas sobre *stand-alone*[1], sistemas embebidos[2], nodos dedicados, etc. Para la planificación de tareas se han propuesto esquemas teóricos que proveen soporte a aplicaciones de tiempo real. Una aplicación es el conjunto de tareas relacionadas que brindan alguna función al sistema o también se les denomina flujos (*streams*). La planificabilidad de las aplicaciones en un sistema se determina por medio del análisis de todas las aplicaciones en conjunto, lo que implica realizar un análisis que tarda tiempo proporcional al número de tareas, es decir, el análisis para n tareas en el sistema tarda tiempo $O(n)$. Para lograr una reducción en el tiempo en dicho proceso de verificación se han desarrollado estrategias de planificación que se auxilian de modelos jerárquicos.

En [12] y [11] se propone un esquema de planificación jerárquico para un sistema abierto de aplicaciones independientes que satisface los objetivos que según Deng *et al.* un sistema (o aplicación) de tiempo real debe satisfacer:

1. Permitir al desarrollador de cada aplicación de tiempo real validar la planificabilidad de las tareas en la aplicación de manera aislada.
2. Tener un criterio de aceptación con el que el sistema operativo puede determinar si es posible admitir una nueva aplicación de tiempo real sin tener que analizar la planificabilidad de todas las aplicaciones existentes en conjunto con la nueva.
3. Una vez que el sistema haya aceptado una aplicación de tiempo real, garantiza la planificabilidad de las tareas en la aplicación.
4. El sistema debe mantener cierto nivel de respuesta ante aplicaciones que no sean de tiempo real (conjuntos de tareas que no poseen restricciones temporales, tales como las tareas aperiódicas).
5. Se debe llevar a cabo todo lo antes mencionado sin omitir la asignación adecuada de tiempo y recursos o las restricciones temporales para las aplicaciones, con demandas de tiempo y recursos variables.

El esquema mencionado asume que cuando el sistema operativo admite una nueva aplicación de tiempo real, éste crea un servidor de utilidad constante (*Constant Utilization Server*) para ejecutar la aplicación. El esquema jerárquico se basa en dos partes. El primero, el nivel elevado de la jerarquía, se encarga de la asignación del tiempo a una tarea, fija su deadline y luego la ejecuta según las políticas del algoritmo *Earliest Deadline First*. El segundo, el nivel bajo, sólo se encarga de acomodar las tareas de un sólo *stream* (una aplicación) de acuerdo a sus prioridades en su cola correspondiente.

Para este trabajo, el esquema jerárquico de planificación ha sido modificado, ya que se ha agregado la capacidad de interacción y entonces llevar a cabo el consenso.

Más trabajos como el presentado en [42] muestran la aplicación del esquema jerárquico para sistemas distribuidos móviles, en los que los nodos pueden salir y entrar al sistema en

[1] Programa que no requiere de los servicios de un sistema operativo para ejecutarse.

[2] Un sistema embebido es un sistema basado en un microprocesador o un microcontrolador que está desarrollado para controlar una función o un rango de funciones.

cualquier momento. La interacción de los procesos en ejecución en los diferentes nodos da a lugar a nuevas tareas, denominadas *coordinadas*. La ejecución de estas nuevas tareas requiere de la reconfiguración del sistema, dicha reconfiguración está a cargo del protocolo *Coordination Triggered Protocol*.

Para nuestros fines el conjunto de nodos permanece inamovible durante un lapso de tiempo. En el tiempo en el que el sistema conserva su configuración se debe lograr la ejecución de las tareas esporádicas que han sido aceptadas, por algún test de aceptación que involucra al consenso al que llega un subconjunto de nodos relacionados.

Actualmente existe una amplia diversidad de algoritmos de planificación, cada uno de ellos está diseñado para resolver algún aspecto específico. Dentro de la lista de aspectos con los que se trata de lidiar se encuentran el brindar una repartición justa de los recursos a las diferentes tareas, brindar una mejor calidad de servicio (*QoS*) [21], sobrellevar las condiciones que provocan sobrecarga de trabajo al sistema, [37], combatir la inanición [43], garantizar exclusión mutua, [48], entre otros.

La planificación en tiempo real no necesariamente cubre los criterios antes expuestos. La planificación de tareas en tiempo real tiene el objetivo primario de ejecutar una tarea dentro de sus restricciones temporales: plazo, tiempo de liberación, un tiempo de ejecución por cada instancia, etcétera.

2.2. Modelado del sistema

Para el desarrollo de este algoritmo se considera una red de nodos que no cambian su configuración durante un intervalo de tiempo definido por $\gamma = [t_{ini}, t_{fin})$. Definimos un nodo como aquella entidad física del sistema distribuido dotada de un procesador capaz de soportar, al menos, *dos hilos de ejecución en paralelo* con su respectiva memoria, así como con la capacidad de comunicarse con el resto de los nodos en el sistema, y capaz de soportar, al menos, dos hilos de ejecución simultáneamente: uno de ellos encargado de la planificación de tareas; mientras, el segundo está destinado a la recepción de información y peticiones.

El algoritmo presentado en este capítulo tiene como principal objetivo utilizar el consenso para lograr la planificación global de tareas independientes y apropiativas [47], sobre un sistema distribuido heterogéneo en tiempo real sin fallas, a partir de la planificación local dinámica que tiene lugar en cada uno de los nodos que forman parte del sistema. El reto es lograr satisfacer los plazos de tiempo de las tareas esporádicas aceptadas para ser ejecutadas en un nodo a pesar del consenso.

Sobre cada nodo hay implementada una estrategia dinámica de planificación basada en prioridades. La tarea con la prioridad más alta se asigna al procesador dentro del nodo N_{id} cada *marca de reloj*, ΔT_{id}. Cada nodo se hace cargo de la planificación de un conjunto de tareas que le es asignado previamente (las tareas confinadas), dicho conjunto de tareas consta de tareas periódicas, aperiódicas y esporádicas.

El caso de estudio considera sistemas heterogéneos, lo que trae consigo aspectos adicionales que deben ser considerados para este trabajo: las diferentes velocidades de procesamiento de los nodos en el sistema y la sincronización de sus relojes, principalmente. La escala de tiempo (o resolución del procesador) sobre la que cada nodo funciona es, probablemente, diferente por efectos de la heterogeneidad del sistema. Las particularidades de cada uno de los nodos limitan la manera de como se hace la asignación de las tareas, es decir, la asignación de una tarea a un nodo no es aleatoria, asignaremos una tarea a un nodo siempre que el nodo ofrezca los

recursos que la tarea requiere para ser ejecutada. Cabe aclarar que la forma en que las tareas son designadas a los nodos no es parte del alcance de este trabajo; sin embargo, resulta ser un tema importante.

El algoritmo comienza su ejecución justo después de que las tareas hayan sido asignadas a los diferentes nodos, de acuerdo a sus capacidades y características (capacidad de memoria, demanda de procesador, velocidad del procesador). Al comenzar el algoritmo, en cada nodo, las tareas se mantienen en estado de espera hasta que sus respectivos tiempos de activación les indiquen que es momento de ser ejecutadas si son tareas periódicas, o bien, si son tareas esporádicas o aperiódicas, el planificador determina que ha llegado el momento de asignar a alguna de ellas al procesador[3]. El algoritmo de planificación sólo puede asignar el procesador a una sola tarea a la vez, por lo que en cada nodo hay, en todo momento, a lo más una tarea en *ejecución*.

Consideremos el problema de planificar un conjunto finito de tareas totalmente apropiativas e independientes en el nodo N_{id}:

$$\Gamma_{id} = \{\tau^S\} \cup \Gamma_{periódicas} \cup \Gamma_{esporádicas} \cup \Gamma_{aperiodicas} \tag{2.1}$$

compuesto de una tarea servidor τ^S (se explicará más adelante en la sección 2.4), un subconjunto de tareas periódicas, esporádicas y aperiódicas, respectivamente, sobre un nodo

$$N_{id}(\Gamma_{id}, V_{id}, \Delta T_{id}, factor_{id})$$

donde Γ_{id} es el conjunto de tareas confinadas, V_{id} es la velocidad y ΔT_{id} es el valor de la marca de reloj y $factor_{id}$ es el nivel de importancia de las tareas aperiódicas en el nodo cuyo identificador es id y, además, $id \in \mathbb{N}$ y es único en el sistema. Existen en el sistema un número finito de nodos, m, y todos *tienen la capacidad de comunicarse con todos*.

A continuación, se definen los parámetros que permiten definir a cada tarea τ_i.

La ejecución de tareas se hace por instancias que se liberan con regularidad cada cierto tiempo. Sean i, j dos valores tales que $i \in \mathbb{N}$ y $j \in \mathbb{N} \cup \{0\}$, donde i representa el identificador único de una tarea en el sistema y j es el valor que determina la instancia que será ejecutada y que depende del tiempo. Para conocer la instancia de la tarea en $\Gamma_{periódicas}$ que se encuentra en ejecución en tiempo t, $t \in \mathbb{N} \cup \{0\}$ se calcula:

$$j_i(t) = \left\lceil \frac{t}{T_i} \right\rceil \tag{2.2}$$

T_i: Es el valor del periodo, o regularidad con la que se libera una instancia (job) de la tarea i, τ_i.

$d(\tau_{i,j})$: Es la función que devuelve el *plazo absoluto de la instancia j* de la tarea periódica i, $\tau_{i,j}$, pero será denotada por $d_{i,j}$:

$$d_{i,j} = d(\tau_{i,j}) = (j+1) \cdot T_i \tag{2.3}$$

d_i: Es el plazo de tiempo dentro del que debe ejecutarse la tarea esporádica, τ_i^E, en su totalidad.

C_i: Es el número de unidades de tiempo que tiene cualquier instancia de una tarea, τ_i, para utilizar el procesador.

$r_{i,j}$: Es el tiempo de liberación o activación de la instancia j de la tarea i, $\tau_{i,j}$.

[3]Las frases ejecutar una tarea y asignar una tarea al procesador tienen el mismo significado.

El conjunto de tareas, Γ_{id}, asignado a un nodo, N_{id}, esta compuesto (como se definió antes) por tres subconjuntos que se definen en base a los parámetros antes mencionados:

- $\Gamma_{periódicas} = \{\tau_i^P(T_i, C_i) : T_i, C_i, i \in \mathbb{N}, r_{i,0} = 0,\ d_{i,j} = k \cdot T_i$, donde $k \in \mathbb{N}$ e i es un identificador único del sistema $\}$,
- $\Gamma_{esporádicas} = \{\tau_i^E(d_i, C_i, r_{i,0}) : d_i, C_i \in \mathbb{N}, r_{i,0} \in \mathbb{N} \cup \{0\}$ y aún no ha sido liberada$\}$ y
- $\Gamma_{aperiodicas} = \{\tau_i^A(C_i, r_{i,0}) : C_i \in \mathbb{N}, r_{i,0} \in \mathbb{N} \cup \{0\}\}$.

Como condición para este algoritmo se exigirá que el único subconjunto que tiene permitido ser vacío es el conjunto de tareas aperiódicas, $\Gamma_{aperiodicas}$. Si $\Gamma_{periódicas}$ o $\Gamma_{esporádicas}$ son vacíos $(= \phi)$, entonces el nodo no tendría una participación activa en el algoritmo de manera global.

Por el momento se limitará a definir la tarea servidor, τ^S, como una tarea periódica más y es necesaria una en cada nodo. Dicha tarea se mantendrá agrupada junto con el resto de tareas periódicas como se define enseguida:

$$\Gamma'_{periódicas} = \Gamma_{periódicas} \cup \{\tau^S\} \tag{2.4}$$

Si la instancia j de la tarea i, $\tau_{i,j}^{tipo}$, (donde *tipo* puede ser periódica, P; aperiódica, A, o esporádica, E) se completa en tiempo t, entonces su tiempo de respuesta se define como:

$$t_{respuesta} = t - r_{i,j} \tag{2.5}$$

la demanda de procesador en un intervalo de tiempo, $\alpha = [t_1, t_2)$, para una tarea, τ_i, se define como:

$$demanda(\alpha, C_i, T_i) = \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor \cdot C_i \tag{2.6}$$

La tarea esporádica τ_i^E no solamente se encuentra alojada en un sólo nodo. Existe la posibilidad de que τ_i^E también se encuentre almacenada en otros nodos, por lo que es necesario que los nodos que contengan a dicha tarea conozcan al resto de los nodos en el sistema que la almacenan para lograr el acuerdo que permitirá su activación. Al conjunto de nodos en el sistema que poseen la misma tarea esporádica τ_i^E se le denomina **emparentados** o nodos relacionados, y se define como sigue:

$$\Upsilon(\tau_i^E) = \{N_{id} : \tau_i^E \in \Gamma_{id}\} \tag{2.7}$$

Para efectuar el consenso, para cualquiera de las tareas esporádicas que han sido confinadas antes de comenzar el algoritmo, en el nodo N_{id}, se requiere del conocimiento de los nodos con los que se debe llegar al acuerdo que define al nodo apto para la ejecución de una tarea esporádica liberada, τ_i^E. Por lo tanto, cada nodo lleva el registro de las tareas esporádicas que almacena y que también otros nodos almacenan, dicho registro se define a continuación:

$$\upsilon(N_{id}) = \{\Upsilon(\tau_i^E) : \tau_i^E \in \Gamma_{id}\} \tag{2.8}$$

En el comienzo del algoritmo, el nodo N_{id} conoce un subconjunto de las tareas esporádicas en el sistema que deben ser ejecutadas, $\Gamma_{esporádicas}$, en dicho conjunto las tareas permanecen inactivas. Cuando alguna tarea, $\tau_i^E \in \Gamma_{esporádicas}$ es liberada y aceptada (después de un consenso previo) pasa a formar parte del conjunto de tareas esporádicas *listas*, en tiempo t, dentro de alguno de los nodos en $\Upsilon(\tau_i^E)$:

$$\Gamma'_{esporádicas} = \{\tau_i^E : r_{i,0} \geq t \text{ y haya sido aceptada}\} \tag{2.9}$$

igualmente ocurre para las tareas aperiódicas, excepto que éstas al ser liberadas de la lista $\Gamma_{aperiódicas}$, en el instante t, son aceptadas inmediatamente dentro del nodo donde aparecen y agregadas a una nueva lista, en la que esperarán a ser ejecutadas:

$$\Gamma'_{aperiódicas} = \{\tau_i^A : r_{i,0} \geq t\} \tag{2.10}$$

Como se mencionó antes, para los acuerdos sobre la activación de las tareas esporádicas es indispensable el consenso, que requiere del intercambio de información. La información de la tarea esporádica, τ_i^E, que se aloja en nodo N_{id} se transmite por medio de la red de comunicaciones a través de un mensaje, msg_i, que se define como una tupla con los siguientes campos:

$$msg_i = (i, r_{i,0}, id, V_{id}, H_{id}, tipo) \tag{2.11}$$

i: identificador de la tarea esporádica τ_i^E.

id: identificador del nodo que envía el mensaje.

$r_{i,0}$: tiempo de liberación de la tarea τ_i^E.

H_{id}: el horizonte de tiempo calculado por el nodo (N_{id}) que envía el mensaje.

$tipo$: que indica la manera en que el mensaje será procesado: como *información* o como *petición*.

Cuando un mensaje es enviado como información quiere decir que el mensaje ha sido enviado por primera vez; mientras que una petición es la información de una tarea para la que se requiere de una ronda más de consenso, debido al cambio de estado que no permite a dicha tarea ser asignada a algún nodo todavía (la tarea se encuentra en búsqueda de un nodo apto).

Los mensajes recibidos por el nodo N_{id} son almacenados en listas. Una lista es un conjunto de mensajes de un sólo tipo: respuestas o peticiones.

$$lista = \{msg_i : i \text{ es el identificador de la tarea } \tau_i^E\} \tag{2.12}$$

Cuando el mensaje recibido por N_{id} contiene información se almacena en una lista de acuerdo con el identificador de la tarea indicada en el mensaje, formando así un conjunto de listas (una por tarea esporádica alojada en N_{id}) denominado *respuestas*:

$$respuestas = \{lista_i : i \text{ es el identificador de } \tau_i^E\} \tag{2.13}$$

En cambio, cuando el mensaje recibido por N_{id} es una petición se almacena en una lista denominada *peticiones*:

$$peticiones = \{msg_i : i \text{ es el identificador de } \tau_i^E \text{ y es una petición}\} \tag{2.14}$$

, en la que las peticiones serán atendidas en el orden en el que son recibidas (estrategia FIFO).

2.2.1. Estados de una tarea

Durante su existencia, una tarea τ_i pasa por diferentes estados, los cuales indican la actividad actual de τ_i en el nodo, como se muestra en la figura 2.1, a continuación se listan los estados:

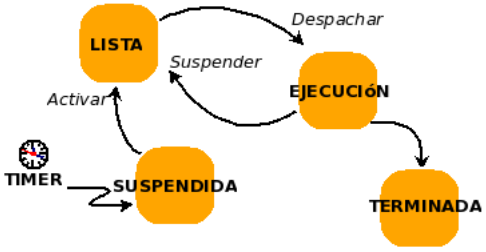

Figura 2.1: Diagrama de transiciones de estados por los que pasa una tarea dentro de un nodo.

Activa, si la tarea τ_i está activa en tiempo t, entonces $r_i \leq t < d_i$ y ha sido aprobada por un test de aceptación.

Lista, es una tarea que potencialmente podría ser ejecutada. Es cualquier tarea, τ_i, cuyo tiempo de activación a comenzado en tiempo $t(\geq 0)$, es decir $t \geq r_i$ y aún falta consumo por ejecutar, $C_i > 0$, además, en caso de tareas esporádicas o aperiódicas, han sido aceptadas después de un examen de aceptación previo. Nótese que toda tarea lista está activa.

Ejecución, una tarea está ocupando el procesador.

Suspendida, es el estado inicial en el que se encuentran las tareas de un nodo, en otras palabras, es cualquier tarea τ_i almacenada en memoria que aún no cumple con su tiempo de activación en tiempo t, o bien, $r_i < t$.

Terminada, ocurre una vez que la tarea ha concluido de manera definitiva su ejecución.

2.3. Descripción de la planificación

El propósito de este algoritmo es garantizar la ejecución de tareas esporádicas aceptadas en un nodo, dentro de un presupuesto de tiempo, C^S_{AE}, asignado cada periodo, T^S, según se especifique para la tarea servidor de cada nodo. Enfoquemos la atención en el trato de tareas esporádicas dado que éstas poseen plazos que hay que cumplir dentro de un tiempo específico; mientras que las tareas aperiódicas sólo se deben ejecutar sin que necesariamente se cumpla un plazo; a menos, que el programador así lo decida para evitar la condición de inanición de la que pueden ser objeto, en este caso se utiliza un método de penalización (sección 2.6).

El ambiente está conformado por tareas esporádicas redundantes en el sistema, que a causa de la heterogeneidad de los nodos en el sistema no pueden ser asignadas de cualquier forma, así que tal asignación en los nodos debe hacerse de manera que se satisfagan sus requerimientos de procesamiento y memoria.

Una condición necesaria para este algoritmo consiste en garantizar que al final de la asignación de tareas, en cada nodo, debe haber, al menos, una tarea esporádica y una periódica, de lo contrario el nodo no se considera para la ejecución del algoritmo en el sistema distribuido, además, en cada nodo, es necesario guardar un registro de las tareas esporádicas almacenadas y el del resto de los nodos que también las contienen. Dicha información se almacena, durante el lapso definido por γ.

Para que el algoritmo local pueda comenzar necesita que cada nodo posea su respectivo conjunto de tareas independientes, al que denominaremos Γ_{id}, donde $id \in \mathbb{N}$ es el identificador del nodo en el sistema. Existe una variedad restringida entre las tareas de Γ_{id}, a saber:

1. Periódicas: con periodo; igual a su tiempo límite, consumo y tiempo de activación definidos.

2. Aperiódicas: con consumo y tiempo de activación definidos.

3. Esporádicas: con consumo, tiempo de activación y tiempo límite definidos.

Las tareas son repartidas bajo criterios del programador entre los nodos siguiendo un orden. Las tareas periódicas son asignadas primero en cada nodo según se requiera o lo permitan sus capacidades. Después es el turno de las tareas esporádicas, éstas son asignadas a todos los nodos en el sistema distribuido que sean capaces de soportar sus requerimientos (memoria, velocidad de procesamiento, disponibilidad del procesador), por lo que pueden existir varios nodos en el sistema con la misma tarea esporádica. Por último se hace la repartición de tareas aperiódicas dependiendo de la funcionalidad que éstas provean a los nodos y de la capacidad de éstos para soportar todos sus requerimientos.

El conjunto de tareas periódicas asignado a cada nodo debe cumplir lo siguiente:

- Se satisfacen los plazos de tiempo de las instancias de tareas periódicas, es decir, sus plazos son estrictos y se cumplen antes de cada periodo. Se asume que los plazos son iguales a cada múltiplo de su periodo (p.e. si tuviésemos una tarea con periodo definido, T, entonces los plazos de sus instancias ocurrirán en Tn, donde $n \in \mathbb{N}$).
- El conjunto de tareas es planificable, pero éste nunca ocupa en su totalidad el tiempo del procesador, es decir, su utilidad (U_p) cumple que $0 \leq U_p < 1$ dejando tiempo de procesador libre, mismo que es destinado a la tarea servidor.

Por consiguiente, el tiempo disponible $(1 - U_p)$ se destina para uso exclusivo de tareas esporádicas y aperiódicas. El manejo de éstas se hace por medio de una tarea denominada tarea **servidor** (para más detalle vaya a la sección 2.4).

Las tareas aperiódicas comienzan su ejecución en cada nodo cada vez que su tiempo de activación lo indique y haya presupuesto en la tarea servidor para atenderlas; en cambio, las tareas esporádicas deben aprobar un test antes de pertenecer al conjunto de tareas susceptibles a ser ejecutadas, $\Gamma'_{esporádicas}$, para el cual es necesario conocer la carga de trabajo del resto de los nodos emparentados, el nodo que ofrezca la menor carga de trabajo dentro de un lapso de tiempo para una tarea esporádica será el que ejecutará la tarea.

Se desea garantizar la ejecución de las tareas esporádicas aceptadas durante el lapso de tiempo que el sistema permanece invariante[4], por lo tanto para eliminar la posibilidad de que la misma tarea se ejecute casi simultáneamente en varios de los nodos del sistema y, con esto, quitar la oportunidad de utilizar el procesador a otras tareas que también lo requieren (en todos los nodos que contienen a la mencionada tarea), introducimos la idea del *consenso*.

La función principal del consenso en este trabajo será acordar que nodo (el nodo apto) se encargará de la ejecución de una tarea esporádica, garantizando así que sólo uno de los nodos la ejecutará, y a su vez aseguramos un mejor uso de los recursos de los nodos en el sistema.

El consenso y lo que éste implique también consumen tiempo de procesador por lo que es necesario considerar más labores dentro de cada nodo:

1. Atender peticiones (figura 2.2),

2. intercambio de información (figura 2.3) y

[4]Tiempo en el que la disposición geográfica de los nodos que conforman el sistema es la misma y no se han agregado tareas extra.

3. decidir.

Las tres acciones mencionadas permiten que se llegue a una decisión sobre que nodo es "apto", de entre un subconjunto de nodos *emparentados*, para hacerse cargo de la tarea. El nodo apto es el nodo que provee la menor carga de trabajo para ejecutar una tarea dentro de un subconjunto. Para dar una garantía de la ejecución de dichas labores es necesario reservar tiempo de la tarea servidor, al que se le denomina $C_{consenso}$.

El consenso comienza de manera independiente en cada nodo, con el envío de la información de una tarea esporádica. El evento que lo inicia es la liberación de cualquier tarea esporádica, y el consenso comienza siempre y cuando haya presupuesto para la tarea servidor, y termina cuando se decide (en base a la información recibida) que nodo ofrece la menor carga de trabajo en cierto horizonte, H.

Observemos que el consenso sólo tiene sentido entre aquellos nodos que tienen en común a la misma tarea esporádica almacenada, τ^E. Al conjunto de nodos que poseen a la tarea, τ^E, lo denotaremos $\Upsilon(\tau^E)$ y lo llamaremos conjunto de nodos **emparentados** o relacionados por τ^E.

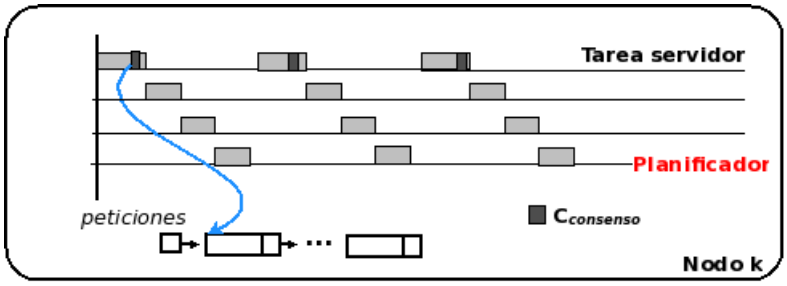

Figura 2.2: Cuando hay tiempo disponible en la tarea servidor, en específico $C_{consenso} > 0$, se revisa la existencia de peticiones para atenderlas.

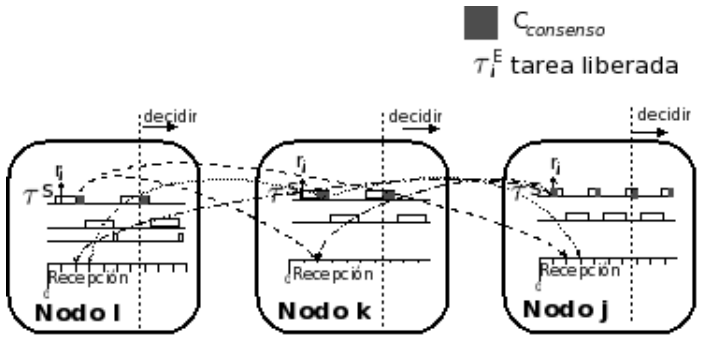

Figura 2.3: Una vez liberada la tarea τ_i^E en tiempo r_i, los nodos del conjunto $\Upsilon(\tau_i^E) = \{N_j, N_k, N_l\}$ comienzan el envío de la información para τ_i^E, siempre que se disponga de presupuesto para actividades de consenso. Hasta que cada nodo en $\Upsilon(\tau_i^E)$ recibe el último mensaje de información y haya presupuesto ($C_{consenso} > 0$) es posible tomar una decisión.

2.4. La tarea servidor

En un nodo, las tareas esporádicas y aperiódicas son liberadas en instantes desconocidos de tiempo y aparecen como respuesta a algún evento en el entorno. Dichas tareas no tienen periodos asociados, por lo que sus instancias deben ser tratadas siguiendo algún criterio. De otra manera

no hay límite en la cantidad de carga de trabajo que una tarea esporádica pueda agregar y por ende el análisis de planificabilidad se torna imposible.

Una manera de tratar con tareas esporádicas o aperiódicas con tiempos de liberación desconocidos es imponiéndoles características que permitan su manejo, una de las formas más comunes es asignándoles un periodo "artificial" (por llamarlo de algún modo): igual al intervalo mínimo de tiempo entre la liberación de instancias sucesivas de ejecución de un conjunto de tareas periódicas residentes en el nodo.

Una forma simple de incorporar tareas esporádicas y aperiódicas es agregando una tarea servidor, τ^S; ya que de esta manera es posible su manipulación como si de tareas periódicas se tratara. Así, es posible asignar al procesador cualquier tarea esporádica o aperiódica aceptada en espera de atención durante lapsos definidos por C_{AE}, cada periodo artificial de tiempo T^S, como si se tratase de una tarea periódica más. Fuera de este tiempo de ejecución (C_{AE}) el procesador está a disposición de tareas periódicas, únicamente.

En cuanto a las características que definen a la tarea servidor, tales como un periodo y un consumo (o presupuesto), no hay una regla que establezca como debe hacerse la asignación de sus valores. Por lo que la obtención de dichos valores se propone y acota en las siguientes dos subsecciones (secciones 2.4.1,2.4.2).

La labor del servidor no sólo se limita a la ejecución de tareas esporádicas y aperiódicas, sino que también tiene encomendada la labor de *enviar la información*, de las tareas esporádicas que han cumplido con sus tiempos de liberación (comienza el consenso) y una vez que se ha reunido toda la información sobre alguna tarea esporádica *decide* cuál nodo resulta apto.

En resumen, la tarea servidor, tiene la función principal de reservar y administrar el tiempo del procesador para ejecutar tareas aperiódicas y esporádicas, cuando lo indiquen las reglas del algoritmo planificador.

2.4.1. Obtención del periodo para la tarea servidor

El valor del periodo, $T^S > 0$, es sólo uno de los atributos que describen a la tarea servidor, a la que denotaremos por τ^S.

Dado que utilizaremos una estrategia dinámica de planificación, basada en *el plazo próximo primero* (vea la sección 1.4.2), con el fin de lograr la prioridad más alta, la mayor parte del tiempo, conviene asignar a la tarea servidor, τ^S, de cada nodo el menor de los periodos de las tareas periódicas ya existentes de manera interna, τ_l^P (donde l es el identificador de la tarea periódica en el nodo), es decir, se hace una revisión de todas la tareas periódicas dentro del nodo y se determina el valor del periodo mínimo, luego ese valor mínimo se debe ajustar según la marca de reloj definida en el nodo N_{id}, ΔT_{id}, el valor obtenido caracterizará el periodo de la tarea servidor, T_{id}^S, que residirá en el nodo. En resumen el valor del periodo se obtiene así:

$$T_{id}^S = \left\lfloor \left(\min_{\tau_l^P \in \Gamma_{periódicas}} \{T_l\} \right) \frac{1}{\Delta T_{id}} \right\rfloor \cdot \Delta T_{id} \tag{2.15}$$

donde T_l es el valor del periodo de la tarea $\tau_l^P \in \Gamma_{periódicas}$.

Nótese que esta estrategia funciona, porque la política para EDF se basa en elegir a la tarea cuya instancia tiene el límite de tiempo próximo y porque suponemos que los periodos son iguales a los tiempos límite de cada instancia, además de ser totalmente apropiativas.

2.4.2. Cálculo del presupuesto de la tarea servidor

El presupuesto es un concepto utilizado para referirse a la cantidad de tiempo de procesador reservado para la ejecución de tareas que no tienen un patrón de liberación definido (es decir, tareas cuyo tiempo de liberación es desconocido), en contraste con las tareas periódicas.

Para el algoritmo aquí propuesto, recordemos que inicialmente un subconjunto de tareas periódicas ($\Gamma_{periódicas}$) ya se encuentra en cada nodo (N_{id}) y la utilización de este subconjunto es menor a uno (sección 2.10.1):

$$U_P = \sum_{i=1}^{K} \frac{\lceil \frac{C_i}{\Delta T_{id}} \rceil}{\lceil \frac{T_i}{\Delta T_{id}} \rceil} < 1 \tag{2.16}$$

donde C_i y T_i son los valores del consumo y periodo de la tarea $\tau_i^P \in \Gamma_{periódicas}$, respectivamente y ΔT_{id} es el valor de la marca de reloj del nodo N_{id} (donde $id = 1, \ldots, m$ es el identificador del nodo en el sistema). Por lo tanto, el valor de la utilidad para la tarea servidor es $U^S = 1 - U_P$. Por otro lado, se conoce el valor del periodo de la tarea servidor (vea ecuación 2.15).

Sabiendo que la utilidad de una tarea se obtiene mediante el cálculo $u = consumo/periodo$, despejando el consumo, luego sustituyendo los valores conocidos (T^S y U^S) y considerando el valor de la marca de reloj, se define el valor para el consumo de la tarea servidor, $C^S < T^S$, como:

$$C_{id}^S = \left\lfloor \frac{U^S \cdot T^S}{\Delta T_{id}} \right\rfloor \Delta T_{id} \tag{2.17}$$

el resultado se debe a la ejecución de la tarea con mayor prioridad cada ΔT_{id} unidades de tiempo, que corresponden a la resolución más fina a la que trabaja el algoritmo planificador, a la que se referirá como *marca de reloj*.

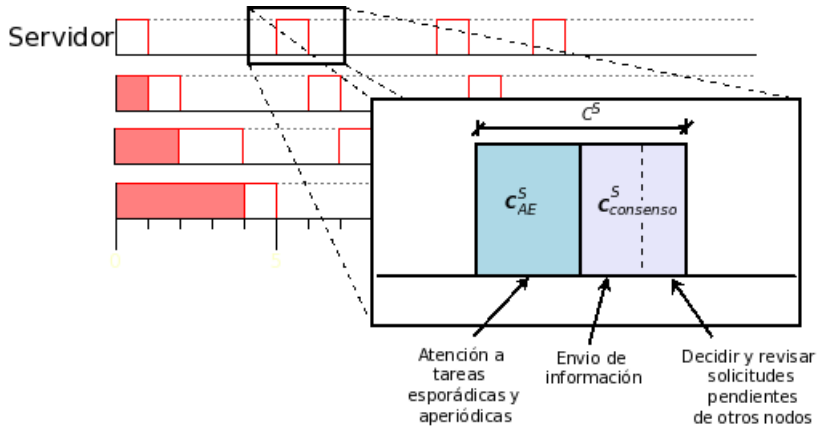

Figura 2.4: División del presupuesto asignado a la tarea servidor.

Esta tarea considera, como se menciono antes, la ejecución de consenso, así como revisar solicitudes pendientes para repetir el consenso, para tareas que aún no encuentran lugar para ejecutarse, hechas por otros nodos. El tiempo de ejecución (o consumo), C^S, destinado para la tarea servidor se subdivide en dos partes (como se muestra en la figura 2.4): la primera, C^S_{AE}, la cantidad de tiempo destinada a la ejecución de tareas esporádicas y aperiódicas; la segunda, $C^S_{consenso}$, de carácter variable; en el aspecto de que las operaciones aquí realizadas podrían tardar menos del tiempo reservado.

El consumo de la tarea servidor se divide en dos partes, ahora es el momento de explicar como asignar la duración de ambas partes. Lo primero que se debe hacer es definir que porción del consumo C^S (calculado con la ecuación 2.17) se desea reservar para la ejecución de tareas

aperiódicas y esporádicas, digamos x ($x \in (0,1)$), entonces el valor para C_{AE} será, considerando el efecto de la marca de reloj (ΔT_{id}), de

$$C_{AE} = \left\lfloor \frac{xC^S}{\Delta T_{id}} \right\rfloor \Delta T_{id} \tag{2.18}$$

una vez que se conoce el valor de C_{AE} se procede a calcular $C_{consenso}$:

$$C_{consenso} = C^S - C_{AE} \tag{2.19}$$

bajo la siguientes restricciones

$$1 \leq \Delta T \leq C_{consenso} \leq C_{AE} < C^S$$

2.5. Asignación de prioridades

Los resultados que se obtengan en este trabajo pertenecen al algoritmo EDF (el plazo próximo primero). En un nodo, planificado por el algoritmo EDF, los tiempos de liberación de las instancias de una tarea son controlados con base a sus plazos: la tarea τ_b^{tipo} tiene mayor precedencia que τ_a^{tipo} si y sólo si $a, b \in \mathbb{N}$, $a \neq b$ y $\tau_b^{tipo} \prec \tau_a^{tipo}$. Entonces, $\tau_b^{tipo} \prec \tau_a^{tipo}$ si y sólo si $d_{b,j_b} < d_{a,j_a}$ o $d_{b,j_b} = d_{a,j_a} \ \wedge \ b < a$. Por lo tanto, cuando los plazos sean iguales, los empates se rompen a favor de la tarea con el identificador menor. Si *tipo* es aperiódica o esporádica, entonces su plazo se calcula mediante la ecuación 2.3, en la que se reemplaza el valor de T_i por el valor del periodo, T^S, de la tarea servidor, τ^S, es decir, el plazo de cada instancia j liberada, τ_{a,j_a}^{tipo} (o τ_{b,j_b}^{tipo}), es un múltiplo del valor del periodo de la tarea servidor.

Para el algoritmo que aquí se presenta, las tareas esporádicas tienen mayor precedencia que las tareas periódicas, las tareas periódicas tienen mayor precedencia que las aperiódicas; sin embargo, con dicha condición las tareas aperiódicas podrían nunca ser atendidas, ya que comparten el presupuesto con las tareas esporádicas. Para asegurar que no sólo se atiende a tareas esporádicas se define una nueva condición que penaliza su ejecución dentro de γ.

2.6. Penalización de tareas esporádicas

Procurando siempre dar cabida tanto a la ejecución de tareas esporádicas como de aperiódicas, surge la necesidad de imponer una nueva restricción, con la que se evita que sólo tareas esporádicas acaparen el tiempo que también le pertenece a tareas aperiódicas (dado que las esporádicas tienen mayor precedencia que las aperiódicas).

Si se ejecuta una tarea esporádica mientras haya tareas aperiódicas *listas* en $\Gamma_{aperiodicas}$ ($\neq \phi$), entonces, la ejecución de tareas esporádicas es sancionada con tiempo de ejecución menor, t^A.

La reducción de tiempo de ejecución para tareas esporádicas se hace de manera gradual. Por cada tarea esporádica, τ_i^E, ejecutada en el nodo N_{id}; siempre y cuando en el instante, t, haya tareas aperiódicas esperando su turno en $\Gamma_{aperiodicas}$, incrementa el valor de la penalización, t^A, disminuyendo la demanda de trabajo para tareas esporádicas en el intervalo γ, una cantidad proporcional a la cantidad de tiempo empleada por la tarea esporádica ejecutada (C_i), además, dicha cantidad se va acumulando:

$$t^A = factor_{id} \cdot \sum_{\tau_i^E \in \Gamma_{esporádicas}} C_i \tag{2.20}$$

donde $0 \leq factor_{id} \leq 1$, es un parámetro que se define dentro de cada nodo y *se establece dependiendo de la importancia de ejecutar las tareas aperiódicas dentro del sistema.*

Cada vez que ocurra la marca de reloj, en tiempo t, se verifica la siguiente condición:

$$\lceil t^A \rceil \geq \left\lfloor \frac{t_{fin} - t}{T^S} \right\rfloor \cdot C^S_{AE} \tag{2.21}$$

cuando sea verdadera, la ejecución de tareas esporádicas se detendrá, y sólo se ejecutarán tareas aperiódicas. Nótese que cuanto más cercano es el valor de $factor$ a uno, tareas esporádicas como aperiódicas se ejecutan, aproximadamente, en igual prioridad.

Por otro lado, no por el hecho de haber reservado una cantidad de tiempo para las tareas aperiódicas quiere decir que éstas vayan a ser ejecutadas en su totalidad, o bien, que por lo menos se ejecute alguna de las que están en espera de ser ejecutadas; entonces, antes de comenzar la ejecución de alguna tarea τ^A_k, primero se revisa que sea verdadera la condición:

$$\exists \tau^A_k \in \Gamma_{aperiodicas} \text{ tal que } C_k \leq t^A \tag{2.22}$$

de lo contrario el tiempo es asignado nuevamente a tareas esporádicas, de haberlas, o se declara como ocioso, de no haber alguna otra labor que atender.

2.7. Arquitectura del planificador

Ya se ha hablado de la interacción entre los nodos del sistema, ahora toca turno de conocer los componentes constituyentes de cada nodo.

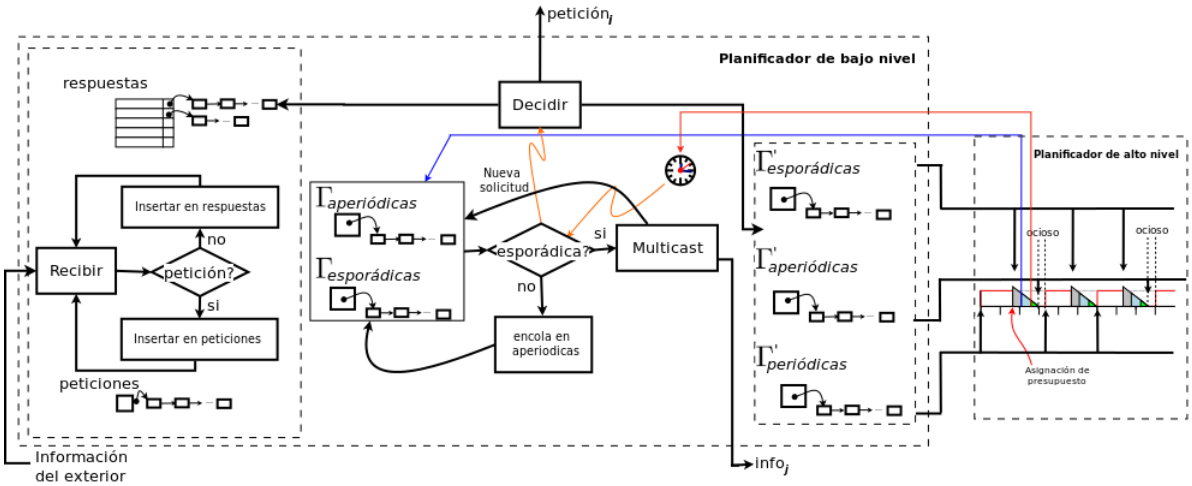

Figura 2.5: Arquitectura del planificador jerarquico, con modificaciones agregadas.

Al igual que en [42] se utiliza un planificador de dos niveles por nodo, sólo que a éste se le han agregado modificaciones al *planificador de bajo nivel* que le permiten a un nodo realizar consenso. La figura 2.5 muestra la arquitectura del planificador, en ella se muestran las dos partes que lo componen: un planificador de alto nivel y otro de bajo nivel.

A comparación del trabajo antes citado, [42], para este trabajo la tarea se encuentra íntegra, es una sola entidad y no se encuentra fragmentada en procesos y dispersa en el sistema (no es una tarea coordinada). Las únicas tareas para las cuales se requerirá de un consenso previo para ser ejecutadas son las tareas esporádicas, la razón del consenso es porque una sola de estas tareas es asignada a diferentes nodos.

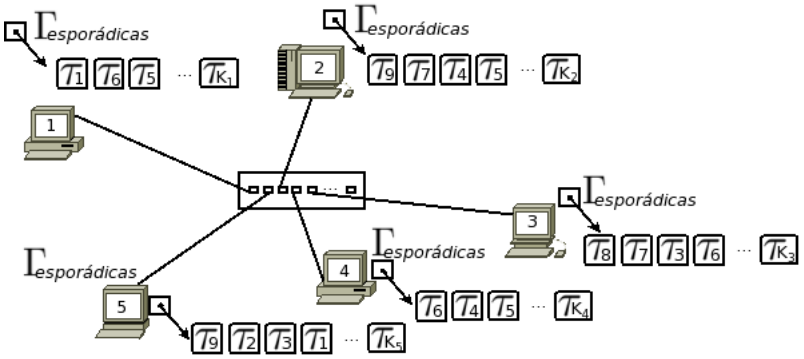

Figura 2.6: Ejemplo de la distribución de las tareas esporádicas en el sistema.

Los nodos que almacenan a la misma tarea están emparentados por alguna tarea esporádica, τ^E, el consenso comienza cada vez que haya presupuesto para la tarea servidor y que, al menos, haya una tarea esporádica liberada. Véase, por ejemplo, la imagen 2.6 que muestra una red cualquiera en donde los nodos 1, 2 y 4 poseen en su respectiva lista a la tarea 5, por lo tanto serán sólo estos nodos quienes efectúan el consenso para determinar en cual de ellos debe ser activada, justo después de que su tiempo de activación se haya cumplido.

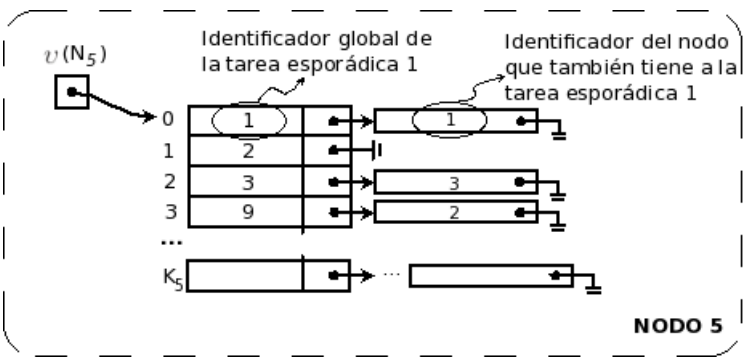

Figura 2.7: Configuración de la asignación de tareas en el sistema distribuido para el nodo 5 (del ejemplo de la figura 2.6), N_5, durante un intervalo de tiempo, almacenada en $\upsilon(N_5)$.

Se ha agregado también una nueva estructura de datos, una tabla en la que cada nodo almacena toda la información que otros nodos relacionados le envían, este estructura ha sido llamada *respuestas*.

En cada nodo, N_{id}, también existe una tabla, $\upsilon(N_{id})$, que contiene los identificadores de todas las tareas esporádicas que posee el nodo N_{id} y para cada tarea se indica un listado con el resto de los nodos del sistema que también la almacenan. El propósito de este conjunto es el de ayudar a determinar que se tiene la información completa de alguna tarea para, entonces, *decidir*. Esta tabla almacena temporalmete una parte de la configuración del sistema (la manera en que las tareas esporádicas han sido distribuidas por el sistema y, por lo tanto, las relaciones entre los nodos durante γ). Si retomamos el ejemplo de la figura 2.6, el conjunto de nodos *emparentados* para el nodo N_5 ($\upsilon(N_5)$) es como la mostrada en 2.7, obsérvese que no es necesario colocar para cada lista el identificador del nodo N_5, ya que por el hecho de contener al conjunto de identificadores, contiene a las tareas.

Para que cualquier nodo, N_{id}, sea capaz de llegar al consenso sobre una tarea esporádica, τ_i^E, debe tener la información completa, para ello el planificador de bajo nivel revisa que se tengan los mensajes del resto de los nodos emparentados en el conjunto *respuestas* para la tarea con identificador i (con la ayuda de $\upsilon(N_{id})$) para luego determinar el nodo en el que se habrá de

activar la tarea τ_i^E.

2.7.1. Planificador de bajo nivel

Este nivel consta, a su vez, de otras dos partes. La primera, conformada por el mecanismo de ordenar las tareas confinadas en colas, dependiendo de su tipo (esporádica, aperiódica o periódica), y cada cola se planifica de manera independiente de las demás. En el planificador de bajo nivel se encuentra la tarea servidor, τ^S, que está en la cola de tareas periódicas($\Gamma'_{periódicas}$) por lo que ésta entra en acción cuando el planificador de alto nivel la elige como la tarea de mayor prioridad.

La segunda, consiste en la recepción y clasificación de la información recibida por el nodo, ya sea para llegar a un consenso o para recomenzar con el consenso de alguna tarea que aún no se ha podido activar (petición).

Dado que se desconoce el tiempo de llegada de las tareas esporádicas, éstas se planificarán en linea [5] (*online*).

2.7.2. Planificador de alto nível

Una vez que el planificador de bajo nivel ha ordenado las tareas en sus respectivas colas, el planificador de alto nivel tiene el cometido de ejecutarlas. La ejecución de tareas se efectúa mediante una técnica dinámica de planificación. El algoritmo a cargo de seleccionar la tarea de mayor prioridad es EDF, por lo que se elegirá la tarea con el plazo próximo cada marca de reloj (ΔT), mientras que el manejo del presupuesto para la tarea servidor puede hacerse con alguna de las técnicas revisadas en el anexo B (las estrategias ahí mostradas mejoran el tiempo promedio de ejecución de tareas).

2.8. Análisis de aceptación de tareas

Para cualquier tarea aperiódica, τ^A (confinada en cualquiera de los nodos), el criterio es muy sencillo, cuando su tiempo de liberación se cumple sólo se cambia su estado a *lista*, se encola en $\Gamma'_{aperiodicas}$ y ésta espera a ser elegida por el planificador de alto nivel; la memoria no es infinita así que se mantendrán tantas tareas aperiódicas como le sean posibles a cada nodo, y serán atendidas cuando haya tiempo disponible en la CPU.

Las tareas esporádicas requieren de presupuesto para comenzar el consenso ($C_{consenso} > 0$). En los nodos, la liberación de una tarea esporádica no necesariamente coincide con el momento en que el presupuesto les es asignado a sus respectivas tareas servidor, por lo que es necesario que esperen hasta que les sea asignado otra vez, y así se podrá comenzar con el intercambio de información de la tarea ya liberada (vea la figura 2.8), para luego llegar a un consenso (determinar el nodo que activará a la mencionada tarea).

El análisis realizado en el planificador de bajo nivel para aceptar una nueva tarea esporádica, τ_i^E, es más complejo ya que un nodo debe considerar la carga de trabajo a la que está sometido. Para calcular éste valor es necesario el conocimiento de dos valores: el horizonte de tiempo libre (H) y la velocidad a la que funciona la Unidad Central de Procesamiento (UCP). El horizonte,

[5] En la búsqueda *off-line* el flujo de ejecución del programa se conoce a priori, mientras que en la búsqueda *on-line*, éste se conoce a posteriori, por ello también se le llama búsqueda en ambientes desconocidos.

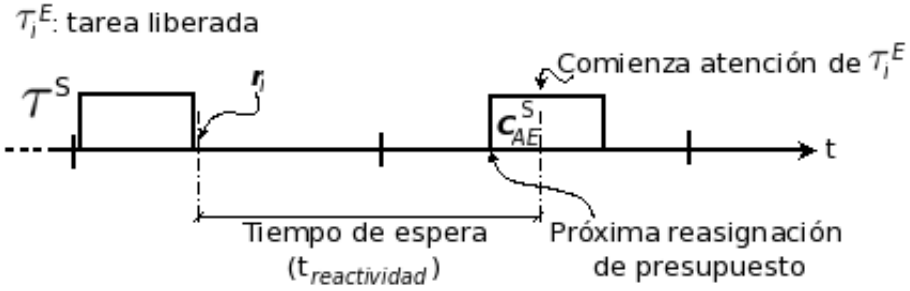

Figura 2.8: Tiempo que una tarea esporádica, τ_i^E, debe esperar antes de que comience el consenso.

H, debe ser mayor al valor del consumo de la nueva tarea esporádica ($C_i < H$) y la velocidad de cada UCP, representada por un valor entero positivo tal que $V_{id} > 0$. Antes de activar τ_i^E, en alguno de los nodos emparentados, ocurre un intercambio de información entre ellos, es decir, comienza el consenso.

El consenso termina cuando cada nodo emparentado posee toda la información del resto de los nodos y, con ésta, tomar una decisión, nótese que el intercambio de información entre los nodos no ocurre de manera sincronizada ya que es posible que las características de las tareas servidor en los nodos sean diferentes (los atributos T^S y C^S son diferentes), además de haber una posible distorsión de reloj en cada nodo N_{id}, δ_{id}.

2.8.1. Cálculo del horizonte de tiempo

Cuando en tiempo t, en algún nodo se encuentre una tarea esporádica, τ_i^E, cuyo tiempo de liberación se haya cumplido (i.e. cuando $r_i \geq t$) y haya presupuesto ($C^S > 0$), debe calcularse la cantidad de tiempo de la que dispone la tarea servidor para su ejecución, a dicha cantidad de tiempo se le nombrará **horizonte**. El valor del horizonte junto con la información de las velocidades de cada nodo emparentado, ayudarán al criterio que determina que nodo en $\Upsilon(\tau_i^E)$ es el más "apto" para ejecutar a dicha tarea.

La prioridad de este algoritmo es cumplir los plazos de las tareas esporádicas que ya fueron aceptadas, por un test de aceptación, después de ocurrido el consenso en cada uno de los nodos. Dichas tareas se encuentran en espera de ser ejecutadas en la lista $\Gamma'_{esporádicas}$.

En cada nodo, es necesario conocer el tiempo del que se dispone, antes de activar una nueva tarea, τ_i^E, primero se obtiene el valor del plazo más lejano de entre las tareas en $\Gamma'_{esporádicas}$ y la nueva tarea, τ_i^E:

$$max\_plazo = \max_{\tau_k^E \in \Gamma'_{esporádicas} \cup \{\tau_i^E\}} \{d_k\} \tag{2.23}$$

y luego, a dicha cantidad se le resta el tiempo actual t :

$$t_{disponible}(t) = \left\lfloor \frac{max\_plazo - t}{T^S} \right\rfloor * C_{AE}^S \tag{2.24}$$

donde T^S y C_{AE}^S son el periodo y consumo (destinado a la ejecución de tareas esporádicas y aperiódicas) de la tarea servidor, respectivamente (vea la figura 2.9).

Ahora que ya se conoce de manera local el tiempo disponible en el intervalo, que va del momento en el tiempo en el que termina la ejecución de tareas esporádicas y aperiódicas (el tiempo, a partir del cual, es posible ejecutar la nueva tarea) hasta el valor del mayor de los plazos de las tareas esporádicas en $\Gamma'_{esporádicas} \cup \{\tau_i^E\}$, procedemos a calcular el tiempo reservado para las tareas esporádicas en $\Gamma'_{esporádicas}$ de la manera mostrada en el algoritmo 1.

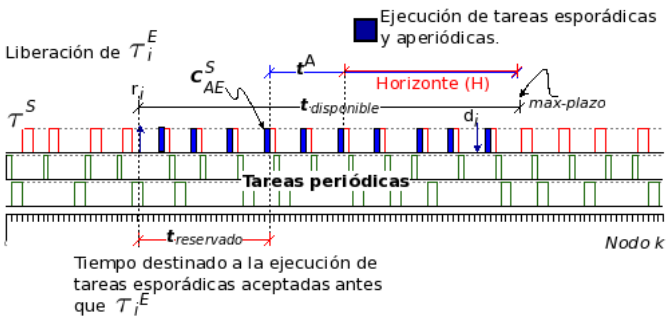

Figura 2.9: Horizonte para la tarea τ_i^E. El horizonte es la sumatoria del número de veces que se puede asignar presupuesto a la tarea servidor enteramente dentro del lapso definido por $t_{disponible} - (t_{reservado} + t_A)$, para la ejecución de tareas esporádicas y aperiódicas, C_{AE}^S.

Función 1 tiempo reservado

Entrada: $\Gamma_{esporádicas}$ y la tarea esporádica liberada τ_i^E

1: $t_{reservado} \leftarrow 0$
2: **para todo** $\tau_k^E \in \Gamma_{esporádicas} \cup \{\tau_i^E\}$ **hacer**
3: $t_{reservado} \leftarrow C_k + t_{reservado}$
4: **si** $t_{reservado} > d_k$ **entonces**
5: error("No es posible planificar τ_i^E")
6: **devolver** -1
7: **fin si**
8: **fin para**
9: **devolver** $t_{reservado} - C_i$

Cuando la nueva tarea τ_i^E es planificable (en el algoritmo 1) es posible calcular el tiempo restante, que es el intervalo de tiempo que todavía puede ser destinado para ejecutar la nueva tarea τ_i^E, es decir, el horizonte, H, y se calcula mediante la diferencia entre el tiempo disponible ($t_{disponible}$), el tiempo reservado ($t_{reservado}$) y la penalización t^A, veamos

$$H(t) = \begin{cases} t_{disponible}(t) - t_{reservado} - t^A & \text{si } \tau_i^E \text{ es planificable} \\ 0 \text{ si } \tau_i^E & \text{de lo contrario} \end{cases} \tag{2.25}$$

Una vez determinado el valor del horizonte, H, se envía a todos los nodos emparentados, y cuando cada uno de los nodos tenga la información completa sobre la nueva tarea, τ_i^E, se decide el lugar en el que se activarán las tareas para las que se efectuó el intercambio de información previo al intercambio que hubo para τ_i^E, incluyendo a ésta última.

Después de que el envío de información ocurrió entre los nodos emparentados del sistema para la tarea τ_i^E, cada nodo, N_{id}, procede a revisar la información en el conjunto *respuestas* para verificar, que se tenga la información completa de cualquiera de las tareas para las que ya se hizo el envío de información (para toda tarea τ_k^E con tiempos de activación menores o iguales al de τ_i^E, r_i), auxiliada por $v(N_{id})$, dicho en otros términos, se busca por toda las listas contenidas en el conjunto de *respuestas* (con la ayuda de $v(N_{id})$, con la finalidad de confirmar que se tiene la información proveniente de todos los nodos emparentados, para decidir finalmente), y por cada lista con la información completa se decide si la tarea se ejecuta internamente (dentro de N_{id}) cuando existe el tiempo suficiente en el nodo.

La manera en que un nodo determina que una tarea puede ser internamente ejecutada se describe en la función decidir (función 2): cuando un nodo, N_{id}, constata que es el apto (línea 4) y, también, verifica que es verdadera la condición de que su estado no haya cambiado desde la última

Función 2 decidir

Entrada: `respuestas`

1: **para todo** $lista_k \in$ `respuestas` y $lista_k \neq \phi$ **hacer**
2: **si** $completa(lista_k) = TRUE$ **entonces**
3: $id\_nodo \leftarrow nodo\_apto(lista_k)$
4: **si** $id\_nodo =$ mi_ID_NODO **entonces**
5: $H \leftarrow calcula\_horizonte(\Gamma'_{esporádicas}, \tau_i^E)$
6: **si** $C_i \leq H$ **entonces**
7: activar_tarea(τ_i^E)
8: **si no**
9: **si** en el momento actual es imposible la ejecución de la tarea indicada por $id(lista_k)$ **entonces**
10: $peticion(info\_actual(), \tau_i^E,$ petición)
11: **fin si**
12: **fin si**
13: **si no**, **si** $id\_nodo = -1$ **entonces**
14: $mostrar($" no es posible activar τ_i^E" $)$
15: **fin si**
16: Eliminar la información de τ_i^E de la tabla `respuestas`
17: **fin si**
18: **fin para**

vez que se realizó el intercambio de información (línea 6), entonces, se tendrá una nueva tarea *lista* en este nodo; en caso contrario, si el nodo constata que es el nodo apto (según la información enviada antes), pero su estado ha cambiado desde la última vez que se hizo el intercambio de información, entonces debe enviar peticiones para comenzar el consenso, nuevamente, para la misma tarea esporádica (línea 10), las peticiones son enviadas a todos los nodos emparentados, con la información actualizada de la tarea que no pudo ser activada en el nodo N_{id}.

La función 3 (nodo apto), que auxilia a la función 2 (decidir), describe cómo se obtiene el nodo que ofrece la menor demanda de trabajo para la tarea analizada y, por lo tanto, puede hacerse cargo de su ejecución. Resumiendo, la función devuelve la información del nodo que presente la menor carga de trabajo:

$$V_{min} = \min_{N_{id} \in \Upsilon(\tau_i^E)} \{V_{id}\} \tag{2.26}$$

$$\min_{N_{id} \in \Upsilon(\tau_i^E)} \left\{ id \text{ tal que } \min_{N_{id} \in \Upsilon(\tau_i^E)} \left\{ \frac{\frac{V_{min}}{V_{id}} * C_i}{H_{id}} \right\} \right\} \tag{2.27}$$

para todo nodo N_{id} que contenga a τ_i^E, donde H_{id} es el valor del horizonte de tiempo de cada nodo $N_{id} \in \Upsilon(\tau_i^E)$.

El criterio utilizado para determinar el nodo *apto* toma en cuenta tanto el tiempo que se tiene para ejecutar una nueva tarea (el horizonte), como la velocidad a la que trabaja el procesador.

El mensaje ($info_{id}$) que un nodo transmite al resto de nodos emparentados, por alguna tarea esporádica, que recientemente se activó, contienen: el identificador de la tarea y velocidad del nodo; de donde se origina el mensaje, identificador del nodo; que hace el envío del mensaje, el horizonte de tiempo y el instante en el que la tarea fue liberada.

Cuando la información del resto de los nodos emparentados es recibida se almacena en una estructura de datos denominada *respuestas*, en el que la información se agrupa en listas de

Función 3 nodo apto

Entrada: $lista_i$

Salida: identificador de nodo "apto"

1: $false \leftarrow -1$
2: $info \leftarrow extraer(lista_i)$
3: $\tau_{aux} \leftarrow info$
4: $H_{apto} \leftarrow calcula\_horizonte(\Gamma'_{esporádicas} \cup \{\tau_{aux}\})$
5: $apto \leftarrow info$
6: $V_{min} \leftarrow \text{mín}_{info \in lista_i}\{vel\_nodo(info)\}$
7: **mientras** $lista_i \neq \phi$ **hacer**
8: $info\_aux \leftarrow extraer(lista_i)$
9: $\tau_{aux} \leftarrow info\_aux$
10: $H_{aux} \leftarrow calcula\_horizonte(\Gamma'_{esporádicas} \cup \{\tau_{aux}\})$
11: **si** $\frac{\frac{V_{min}}{vel\_nodo(info)} * consumo(info)}{H_{apto}} < \frac{\frac{V_{min}}{vel\_nodo(info\_aux)} * consumo(info\_aux)}{H_{aux}}$ **entonces**
12: $info \leftarrow info\_aux$
13: $H_{apto} \leftarrow H_{aux}$
14: **fin si**
15: **fin mientras**
16: **si** $H_{apto} \geq consumo(info)$ **entonces**
17: **devolver** $id(info)$
18: **si no**
19: **devolver** $false$
20: **fin si**

acuerdo con el identificador de la tarea. Cuando se toma la decisión de que nodo debe activar alguna tarea se examina en el conjunto *respuestas* la lista con la información perteneciente a la tarea. Cada mensaje, seguramente, contiene un valor distinto para la velocidad, así que para identificar que nodo puede ejecutar la tarea en el tiempo más corto posible se calcula, además del horizonte, la proporción que indique que tan veloces son el resto de los nodos con respecto al nodo más lento en el sistema, y se calcula de la siguiente forma:

$$\frac{V_{min}}{vel\_nodo_{id}} \tag{2.28}$$

donde id denota el identificador de los nodos emparentados. Nótese que vel_nodo_{id} es la información extraída de cada uno de los menajes de la lista en el conjunto *respuestas*, luego para saber que tanto se reduce el tiempo de ejecución de una tarea se multiplica el consumo de la tarea esporádica, el valor resultante nos dice que tanto tiempo se requiere para ejecutar dicha tarea en el nodo N_{id} y, por último, dicha cantidad se divide por el horizonte de tiempo, que es información que también forma parte de los mensajes recibidos para alguna tarea, lo que resulta en la carga de trabajo que puede tenerse en cada nodo emparentado, por lo tanto, el lugar en el que conviene ejecutar la nueva tarea será aquel que presente la menor carga de trabajo (función 2.27).

2.9. Consenso

Una ronda es la serie de pasos que permiten a un subconjunto de nodos emparentados llegar a un acuerdo para la activación de una tarea esporádica. Comienza con el *multicast* de la información de la tarea, cuyo tiempo de activación se ha cumplido, y termina cuando es posible recolectar la información del resto de los nodos relacionados por la misma tarea, para decidir localmente sobre que nodo es apto.

De manera formal el algoritmo que aquí se propone cumple las siguientes propiedades de consenso:

1. Validez: se decide en base a la información intercambiada. El nodo que decide activar la tarea (el nodo apto) lo hace debido a que el análisis de la información recibida así lo indicó.
2. Terminación: cada nodo sin fallas decidirá paulatinamente. Cada vez que un nodo posea la información completa para decidir los nodos emparentados expresan su decisión sobre la activación local de una tarea votando por el nodo que ofrece la menor carga de trabajo.
3. Acuerdo: cuales quiera dos nodos no decidirán diferente, es decir, si no ocurre un cambio de estado en el nodo apto y todos los procesos ya decidieron sobre que nodo debe activar la tarea, entonces el nodo apto se encargará de activar la tarea.

La importancia de especificar estas propiedades es tal que nos dan un panorama general de que es lo que exactamente debe pasar y con ello determinar si el algoritmo es correcto en su ejecución.

Como ya se mencionó, el propósito de lograr un acuerdo es para encontrar el nodo "apto" en un subconjunto de nodos, para la ejecución de una tarea esporádica. Ésto también consume tiempo de procesador, por lo que es necesario reservar un porcentaje del presupuesto asignado a la tarea servidor para el consenso, $C^S_{consenso}$.

El proceso de consenso se lleva a cabo mediante rondas. Cada ronda consta de tres acciones:

1. **Intercambiar**, consiste en enviar y recibir la información de la tarea que se pretende activar, τ_i^E, entre aquellos nodos emparentados de $\Upsilon(\tau_i^E)$, por lo que no es necesario inundar la red de comunicaciones (se realiza un *multicast*).
2. **Decidir**, una vez concluido el intercambio de información correspondiente a la tarea τ_i^E, se revisa la información que otros nodos enviaron de tareas previas a τ_i^E, e inclusive a ésta última.
3. **Atender**, se procesan las peticiones de consenso de otros nodos emparentados, las peticiones se generan cuando en un nodo no es posible ejecutar una tarea más ya que su carga de trabajo actual se ve sobrepasada por la nueva tarea, lo que quiere decir que el nodo cambio su estado.

2.9.1. Límites de tiempo para el consenso

Recordemos que uno de los aspectos que caracterizan a los sistemas de tiempo real es la predictibilidad, lo que nos lleva a realizar un análisis para identificar retrasos que el consenso puede provocar en la ejecución del algoritmo.

2.9.2. Cotas mínima y máxima para el consenso

Cuando un subconjunto de nodos del sistema distribuido se encuentran relacionados por la posesión común de una tarea esporádica, τ_i^E, seguro llegará un momento en el tiempo en que éstos interaccionarán (a partir de r_i o posteriormente), para determinar el nodo idóneo para la ejecución de dicha tarea. Al subconjunto de nodos, del sistema distribuido, relacionados por

la posesión de una misma tarea, τ_j^E, lo denominaremos conjunto de nodos emparentados y lo representaremos como $\Upsilon(\tau_j^E)$.

La interacción entre los nodos de $\Upsilon(\tau_i^E)$ comienza con el intercambio de la información que permite determinar el estado de la tarea τ_i^E en cada uno de los nodos relacionados (emparentados); una vez que cada nodo haya colectado la información del resto de los nodos para dicha tarea, se decide que nodo se hace cargo de su ejecución, o la de tareas previas a ésta; y cuando el tiempo lo permita se atenderán peticiones de tareas hechas por nodos en los que el estado cambio.

Para determinar el tiempo mínimo en que el consenso puede llevarse a cabo se propone lo siguiente: el tiempo que al menos tarda el consenso en llevarse a cabo ocurre cuando una tarea esporádica, τ_i^E, se libera en todos los nodos emparentados (no necesariamente al mismo tiempo) antes de que el algoritmo de planificación asigne el presupuesto para la ejecución de actividades relacionadas con el consenso ($C_{consenso}$), de esta manera, cuando se asigne el presupuesto a la tarea servidora comenzará el consenso para la tarea τ_i^E.

$$\max_{N_{id} \in \Upsilon(\tau_i^E)} \left\{ \left\lfloor \frac{2 \cdot (T_{id}^S - C_{id}^S) + C_{AE_{id}}^S}{\Delta T_{id}} \right\rfloor \Delta T_{id} \right\} \tag{2.29}$$

donde T_{id}^S, C_{id}^S son, respectivamente, los valores del periodo y consumo de la tarea servidora del nodo N_{id}. Dicho valor se basa en el nodo que ofrece el mayor de los lapsos posibles entre liberaciones sucesivas de dos instancias, como se observa en la figura 2.10.

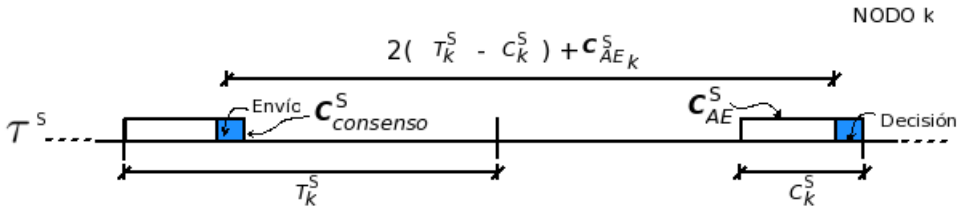

Figura 2.10: Mayor de los lapsos de tiempo posibles entre dos instancias de la tarea esporádica τ_i^E en el nodo N_K.

En cuanto al tiempo máximo en un subconjunto de nodos emparentados, sólo basta tomar el tiempo mínimo que va desde que la tarea τ_i^E es liberada ($r_{i,0}$) hasta el valor de su plazo (d_i) menos su consumo (C_i) y se ajusta con el valor de la marca de reloj:

$$\min_{N_{id} \in \Upsilon(\tau_i^E)} \left\{ \left\lfloor \frac{\left\lfloor \frac{di - r_{i,0}}{T_{id}^S} \right\rfloor C_{id}^S}{\Delta T_{id}} \right\rfloor \Delta T_{id} - \left\lceil \frac{C_i}{\Delta T_{id}} \right\rceil \Delta T_{id} \right\} \tag{2.30}$$

2.9.3. Criterio de cambio de estado

El **estado** de un nodo es la información que refleja la demanda de trabajo que éste tiene dentro de un horizonte en un instante de la ejecución del algoritmo. La información empleada consta de dos datos: el valor del horizonte de tiempo y la velocidad del nodo.

Para un nodo, el conocimiento del estado de los nodos con los que se encuentra relacionado (o emparentado) es requerido para la toma de la decisión que determina el nodo que se ocupará de realizar la tarea (el nodo apto).

La necesidad del conocimiento del estado ocurre en el momento en el que una tarea esporádica

ha sido liberada y es necesario definir su situación en cada uno de los nodos que pertenecen al conjunto que define la tarea, siempre que haya presupuesto ($C_{consenso} > 0$).

El criterio, para decidir que un nodo es *apto* para hacerse cargo de la ejecución de una tarea esporádica, τ_i^E, consiste en elegir al nodo que ofrece la menor carga de trabajo dentro de un horizonte válido (H), es decir, dentro de un horizonte que cumpla con $H > C_i$. Si al terminar el consenso, el horizonte es insuficiente para ejecutar a τ_i^E dentro de sus restricciones temporales $H \leq C_i$, se dirá que el estado ha cambiado para el nodo y éste último comienza una nueva ronda de consenso para determinar que otro nodo es capaz de ejecutar la tarea. De lo contrario, la tarea es activada y comienza su ejecución cuanto antes.

Para entender mejor recurramos a un ejemplo: en un sistema se tienen, al menos, ocho nodos; en el hay dos tareas esporádicas (τ_k y τ_j) cuyos tiempos de liberación son r_j y r_k, de tal forma que $r_j \leq r_k$, con tiempos de consumo (mostrados entre paréntesis) $C_j = 31$ y $C_k = 20$ respectivamente. Las tareas se encuentran en más de un nodo, por lo que se forman dos subconjuntos de nodos emparentados:

1. $\Upsilon(\tau_k^E) = \{N_1, N_2, N_3, N_4\}$
2. $\Upsilon(\tau_j^E) = \{N_5, N_6, N_7, N_8, N_4\}$

la información que refleja el estado de cada nodo, antes de comenzar el intercambio de información para las tareas en ese momento se muestra en la figura 2.11 en tuplas (H, V_{id}), donde H es el lapso de tiempo disponible en ese momento, para ejecutar tareas no periódicas (esporádicas y aperiódicas) y V_{id} es la velocidad de cada nodo (representada por un número entero positivo).

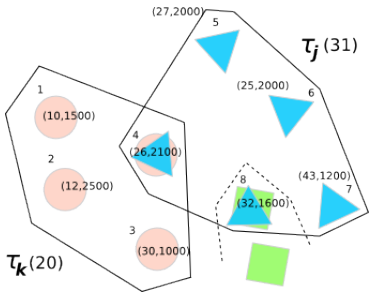

Figura 2.11: Estado inicial de los nodos emparentados por las tareas τ_k^E y τ_j^E, $\Upsilon(\tau_k^E)$ y $\Upsilon(\tau_j^E)$ respectivamente.

Supongamos que el instante en el reloj de los nodos N_4, N_5, N_6 y N_7 es r_j (tiempo de liberación de τ_j^E) y hay presupuesto para consenso ($C_{consenso} > 0$) en cada uno de los nodos mencionados, entonces empieza el intercambio de información como se muestra en la figura 2.12. Por alguna razón, el nodo N_8 no envía su información (no hay presupuesto, el reloj está desfasado con el resto de los nodos o se encuentra ocupado llegando a un acuerdo con otro subconjunto de nodos emparentados), por lo que es imposible decidir para la tarea τ_j^E. Justo después, en tiempo r_k el subconjunto $\Upsilon(\tau_k^E)$ comienza el intercambio de información (vea la figura 2.13) y concluye instantes después (es tácita la existencia de presupuesto, $C_{consenso} > 0$, en los nodos N_1, N_2, N_3 y N_4), por lo que el nodo N_4 decide, según el criterio explicado en la sección 2.8.1, que es el nodo apto. Como consecuencia el nodo N_4 incrementa su carga y disminuye su horizonte, como se muestra en la figura 2.14.

Instantes después de haber concluido el consenso en $\Upsilon(\tau_k^E)$, termina el intercambio de información para $\Upsilon(\tau_j^E)$ como se muestra en la imagen 2.15, por lo que es posible decidir para éste último. Según la información que contiene cada nodo en $\Upsilon(\tau_j^E)$ el nodo apto es, nuevamente,

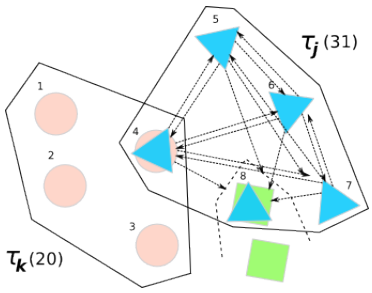

Figura 2.12: El consenso comienza en el subconjunto $\Upsilon(\tau_j^E)$.

N_4. El horizonte disponible en N_4 no es válido, por lo que éste debe comenzar una nueva ronda de consenso, para lo que envía una petición al resto de los nodos en $\Upsilon(\tau_j^E)$ y, con ello, encontrar otro posible destino para la tarea τ_j^E (vea la figura 2.16).

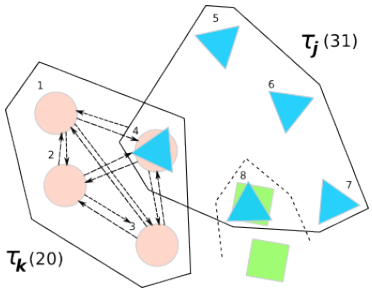

Figura 2.13: Los nodos del subconjunto $\Upsilon(\tau_k^E)$ llevan a cabo el intercambio de información.

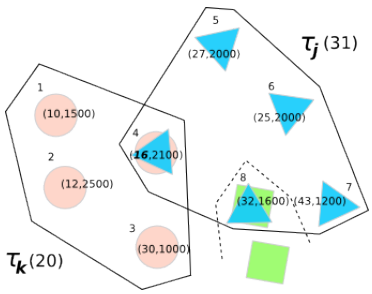

Figura 2.14: El consenso termina para el subconjunto $\Upsilon(\tau_k^E)$ por lo que cada nodo toma la decisión de elegir al nodo poseedor del menor valor de carga. Todos coinciden en que el nodo apto es el nodo N_4.

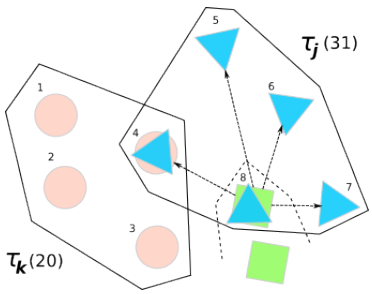

Figura 2.15: El intercambio de información concluye para la tarea τ_j^E.

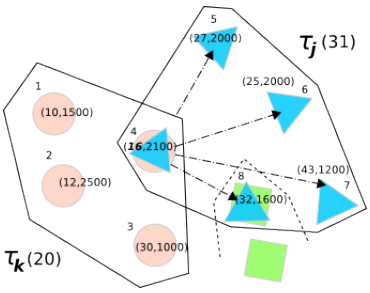

Figura 2.16: Los nodos que conforman $\Upsilon(\tau_j^E)$ acuerdan que el nodo apto es el nodo N_4 nuevamente. Pero dado a que el horizonte ha cambiado en N_4 y es imposible ejecutar τ_j^E en dicho nodo, se envía desde éste una petición al resto de los nodos en el subconjunto $\Upsilon(\tau_j^E)$ para comenzar de nueva cuenta el consenso.

2.10. Algoritmo PGC

Se ha llegado al punto cumbre de este trabajo: enunciar la estrategia de planificación. Las secciones precedentes se enfocaron a explicar algunos de los detalles que envuelven al algoritmo nombrado como *Planificación Global de tareas confinadas basada en Consenso* o, bien, **PGC**. En las siguientes secciones se procede a dar orden a cada uno de los pasos mencionados.

2.10.1. Análisis de suficiencia de la planificación de tareas

La **planificabilidad** es un análisis formal que permite garantizar que un conjunto de tareas pueden (o no) ser asignadas a un recurso compartido (p.e. el procesador) tal que cada una de las tareas cumpla con su plazo de respuesta.

El análisis de factibilidad para tareas periódicas con plazos absolutos iguales a sus periodos bajo el algoritmo EDF se presenta en [35], el cual prueba que un conjunto de n tareas periódicas es planificable si y sólo si la utilización del procesador es menor o igual a uno:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

nótese que el análisis previo no considera la resolución a la que funciona el procesador (ΔT).

Para garantizar que un conjunto de tareas puede ser planificado, bajo PGC, considérense las restricciones que delimitan sus características. El algoritmo de planificación se restringe a tareas con las siguientes características:

1. Todas las tareas son totalmente apropiativas (o en inglés, *preemptive*), de manera que es posible interrumpir cualquier tarea en cualquier instante.
2. El sistema sobre el que se lleva a cabo la planificación es un sistema detonado por el tiempo, por lo que en cada nodo ocurre una interrupción cada ΔT unidades de tiempo con la que se realiza la búsqueda de la próxima tarea de mayor prioridad.
3. Todas las tareas son independientes; no hay precedencia de tareas.
4. Todas las tareas son periódicas; todas aquellas que no lo sean están a cargo de la tarea servidor (τ^S).

5. El plazo absoluto de una tarea es igual a su periodo.

6. Todas las tareas periódicas ($\Gamma'_{periódicas}$) son liberadas en el instante $t = 0$.

Obsérvese lo que sucede con sólo una tarea periódica, τ_1^P. Dicha tarea se puede ejecutar sin ningún problema siempre que se respete la siguiente condición dentro de un nodo N_{id}

$$\left\lceil \frac{C_1}{\Delta T_{id}} \right\rceil \Delta T_{id} \leq \left\lfloor \frac{T_1}{\Delta T_{id}} \right\rfloor \Delta T_{id} \tag{2.31}$$

para este caso se trata de una condición suficiente y necesaria (ver figura 2.17).

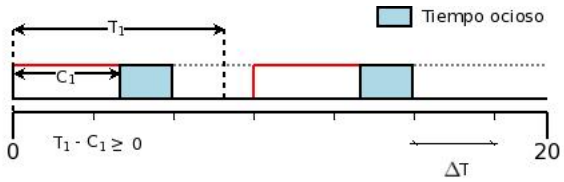

Figura 2.17: Análisis de planificabilidad de una tarea.

Si ahora agregamos una tarea más, τ_2, tal que $T_1 \leq T_2$, en el nodo N_{id}. Ésta será planificada satisfactoriamente si en la primera iteración se puede encontrar suficiente tiempo en el intervalo $[0, T_2]$ (vea figura 2.18). Supongamos que τ_2 termina en tiempo t. Contemos el número de veces que la tarea τ_1 ha sido liberada y completada en $[0, t]$, el número de veces es $\lceil t/T_1 \rceil$ y nótese que, además, debe haber tiempo disponible para ejecutar τ_2 y no se debe olvidar el efecto que tiene la marca de reloj, ΔT_{id}. Por lo tanto, se debe cumplir la siguiente condición:

$$t \geq C'_2 + \begin{cases} \left\lfloor \frac{t}{T_1} \right\rfloor C_1 & si\ C_i \text{ mód } \Delta T_{id} = 0, i = 1, 2 \\ \left\lfloor \frac{t}{T_1} \right\rfloor \left\lceil \frac{C_1}{\Delta T_{id}} \right\rceil \Delta T & si\ C_i \text{ mód } \Delta T_{id} \neq 0, i = 1, 2 \end{cases} \tag{2.32}$$

donde

$$C'_2 = \begin{cases} \left\lceil \frac{C_2}{\Delta T_{id}} \right\rceil \Delta T_{id} & si\ C_2 \text{ mód } \Delta T_{id} \neq 0 \\ C_2 & si\ C_2 \text{ mód } \Delta T_{id} = 0 \end{cases} \tag{2.33}$$

si es posible encontrar el valor de $t \in [0, T_2]$ que satisfaga esta condición entonces es posible garantizar la ejecución de ambas tareas.

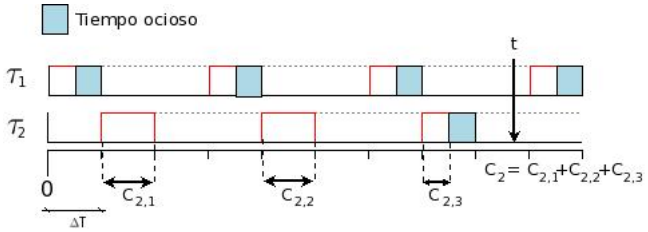

Figura 2.18: Análisis de planificabilidad de dos tareas.

Generalizando esta propiedad para la tarea τ_n, bajo el supuesto que ésta termina la ejecución de la primera iteración en tiempo t (vea la figura 2.19), obtenemos que se debe encontrar un tiempo $t \in [0, T_n]$ tal que

$$t \geq C'_n + \sum_{i=1}^{n-1} \begin{cases} \left\lfloor \frac{t}{T_i} \right\rfloor C_i & \text{si } C_i \text{ mód } \Delta T_{id} = 0 \\ \left\lfloor \frac{t}{T_i} \right\rfloor \left(\left\lceil \frac{C_i}{\Delta T_{id}} \right\rceil \Delta T_{id} \right) & \text{de lo contrario} \end{cases} \tag{2.34}$$

si $T_n > T_{n-1} > \cdots > T_1$, donde

$$C'_n = \begin{cases} \left\lceil \frac{C_n}{\Delta T_{id}} \right\rceil \Delta T_{id} & \text{si } C_n \text{ mód } T_{id} \neq 0 \\ C_n & \text{si } C_n \text{ mód } T_{id} = 0 \end{cases} \tag{2.35}$$

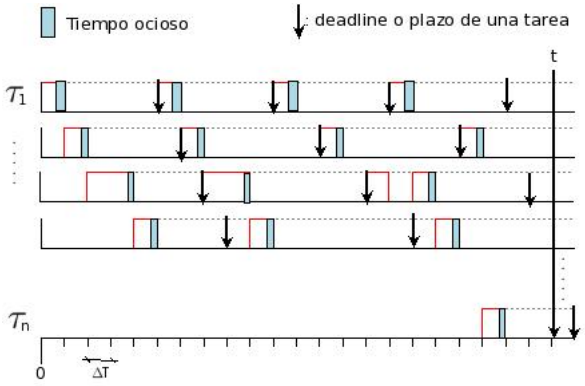

Figura 2.19: Analisis de planificabilidad para n tareas.

El análisis previo nos es útil para finalmente concluir que dadas n tareas periódicas son planificables si

$$U_P = \sum_{i=1}^{n} \frac{\lceil \frac{C_i}{\Delta T_{id}} \rceil}{\lceil \frac{T_i}{\Delta T_{id}} \rceil} < 1 \tag{2.36}$$

2.10.2. Condiciones necesarias del algoritmo

Los algoritmos de tiempo real estricto, como el que aquí se propone, deben garantizar que se cumplen los tiempos límite del conjunto de tareas dentro del sistema.

Uno de los objetivos de los sistemas distribuidos de tiempo real es lograr un mejor desempeño; sin embargo, la predictibilidad y el determinismo son características que siempre se deben cumplir. Para lograr la predictibilidad en los sistemas de tiempo real es común que durante el diseño de un algoritmo de planificación se consideren cotas, basadas generalmente en los peores tiempos de ejecución, o sea, se diseña en base a intervalos de tiempo lo suficientemente amplios para permitir la ejecución de las tareas ante cualquier circunstancia. Ésto obviamente ocasiona que no se aproveche el tiempo de procesador en su totalidad, pero es uno de los costos que se deben pagar para el cumplimiento de las restricciones temporales en el sistema.

El algoritmo de planificación global de tareas confinadas con base al consenso (PGC) es activado por tiempo, por lo que la sincronización de los relojes es crítica. El tiempo es discreto, por lo que la revisión de la siguiente tarea de mayor prioridad se hace con cada marca de tiempo. Nótese que esta marca de reloj no es lo suficientemente fina por lo que será necesario el manejo de buffers que permitan almacenar las interrupciones que ocurran entre marcas de tiempo.

A continuación se hace un recuento de todas las consideraciones necesarias para el funcionamiento correcto del algoritmo PGC:

1. El conjunto de tareas esporádicas que se busca planificar (esporádicas) se distribuye entre los nodos de tal forma que una tarea se encuentre en varios nodos (pero no necesariamente en todos), la razón de ésta decisión está fuertemente ligada a la siguiente condición.

2. Supone que el tiempo que el consenso puede durar es mucho menor al tiempo que puede durar la transferencia de tareas entre nodos.

3. La planificación sólo se ejecuta dentro del tiempo que el sistema se encuentre estático, es decir, el tiempo durante el cual el sistema no cambia su configuración, γ.

4. No ocurren fallas que inhabiliten a los nodos o al medio de transmisión durante el lapso de tiempo γ.

5. No hay duplicación ni pérdida de mensajes.

6. El sistema es activado por el tiempo.

7. Las tareas son totalmente apropiativas y el tiempo de cambio de contexo es despreciable.

8. Cada nodo tiene una lista de las tareas que almacena, y con cada tarea una lista de los nodos en el sistema que también la contienen (la lista sólo es válida dentro de γ). Durante el tiempo que el sistema permanece estático un nodo conoce al resto de los nodos que comparten las mismas tareas que éste.

9. Los nodos están sincronizados, es decir el tiempo del reloj en cada nodo se encuntra dentro del intervalo $[-\delta, \delta]$, entonces cualesquiera dos relojes en el sistema distribuido difieren en a lo más 2δ unidades de tiempo. La desviación de reloj, δ, no se encuentra acotada, es un valor que el programador define.

10. El orden de llegada (al nodo destino) de los mensajes se conserva: el mensaje de una ronda posterior en cualquier otro nodo origen no llega antes que el mensaje de la última ronda ejecutada en el mismo nodo origen al nodo destino.

11. El algoritmo de planificación local utilizado se basa en EDF (el próximo tiempo límite primero) para la asignación de prioridades entre las tareas.

12. Los tiempos de comunicación son valores despreciables (aunque, no es así en la realidad).

2.10.3. Algoritmo de planificación de tareas confinadas basado en consenso

La sucesión de pasos que comprende este algoritmo se lista a continuación (vea la figura 2.20):

1. Preparar la alarma que interrumpirá en ΔT unidades de tiempo.

2. Extraer la instancia $\tau_{i,j}$ de la tarea de mayor prioridad de acuerdo al plazo próximo en $\Gamma'_{perioicas}$.

3. En caso de no haber tarea ($\tau_{i,j} = \phi$), ir al paso *3.c.1.* En caso contrario ($\tau_{i,j} \neq \phi$) verificar el tipo de tarea que es $\tau_{i,j}$:

   a) Si $\tau_{i,j}$ es *periódica*, entonces, ejecutar $\tau_{i,j}$ hasta que ocurra la interrupción al agotarse ΔT o al concluir con el consumo de la instancia en ejecución (cuando el tiempo de ejecución es menor a ΔT). Después de ocurrida la interrupción de la marca de tiempo ir al paso 4.

b) Si $\tau_{i,j}$ es la *tarea servidor* y aún hay presupuesto para ejecutar tareas esporádicas o aperiódicas ($C_{AE}^{S} > 0$), entonces:

1) Si existe alguna tarea *lista* en $\Gamma_{esporádicas}$, τ_i:

a' Si existe alguna tarea en $\Gamma_{aperiodicas}$ y, además, la tarea τ_i es liberada por primera vez, entonces, actualizar el valor de t^A:

$$t^A \leftarrow t^A + (factor) \cdot C_i \tag{2.37}$$

b' Ejecutar la tarea τ_i hasta que ocurra la interrupción de la próxima marca de reloj, o agotarse el tiempo de la instancia liberada de la tarea esporádica τ_i (cuando el tiempo de ejecución es menor a ΔT), en caso de agotarse C_{AE}^{S} vaya al paso *c*. Ir al paso 4 después de ocurrida la marca de tiempo, de lo contrario suspender τ_i y esperar a que la marca de tiempo ocurra.

2) De no haber tareas esporádicas *listas*. Buscar en $\Gamma_{aperiodicas}$ la tarea cuyo tiempo de ejecución sea el menor, τ_i. En seguida, ejecutar τ_i hasta que ocurra la interrupción de la marca de reloj o el tiempo de ejecución termine (cuando el tiempo de ejecución es menor a ΔT). En caso de tampoco haber tareas aperiódicas, entonces, realizar una búsqueda en $\Gamma_{periódicas}$ y extraer la tarea periódica con igual o mayor prioridad que la tarea servidor. Si no se encontró tarea, entonces destinar el tiempo a la atención de peticiones yendo al paso *3.c.1.* Una vez ocurrida la marca de tiempo ir al paso 4.

c) Si se ha agotado el presupuesto $C_{AE}^{S}(= 0)$ y $C_{consenso}^{S} > 0$, entonces:

1) Buscar las tareas esporádicas en $\Gamma_{esporádicas}$ para las cuales el tiempo de liberación se haya cumplido y comenzar el envío de información para cada una de ellas hacia los nodos emparentados.

2) Revisar el conjunto *respuestas* y, para cada tarea ahí registrada que contenga la información completa, entonces *decidir*.

3) Hacer la búsqueda de peticiones y enviar la información de la tarea solicitada de cada petición, a los nodos emparentados.

4) De no haber tareas esporádicas liberadas, información en respuestas y tampoco peticiones, entonces diríjase al paso *3.b.1* (es decir, no ocurrió ninguno de los tres pasos anteriores).

4. Verificar que el tiempo actual sea igual o mayor al tiempo de liberación de la instancia de la tarea servidor. Si la condición anterior es verdadera, entonces, asignar presupuesto para la tarea servidor. Ir al paso 1.

El algoritmo tiene la finalidad de ejecutar las tareas periódicas que inicialmente fueron asignadas a cada nodo, y de tareas esporádicas aceptadas y por último dar atención a la mayor cantidad de tareas aperiódicas.

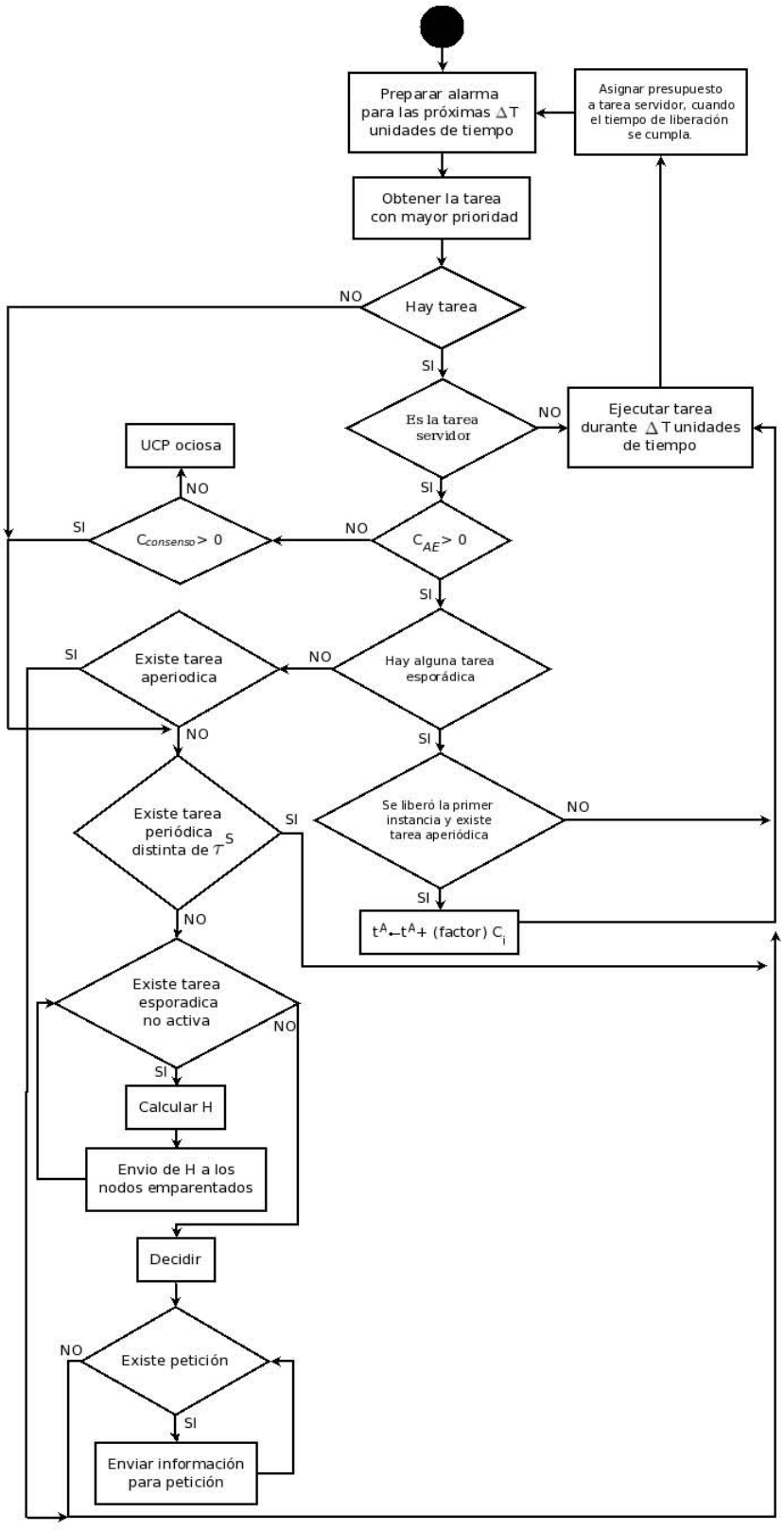

Figura 2.20: Diagrama de flujo del hilo planificador.

Ejemplo

Para este ejemplo, primero fijamos el valor de los parámetros como se lista en la tabla 2.1:

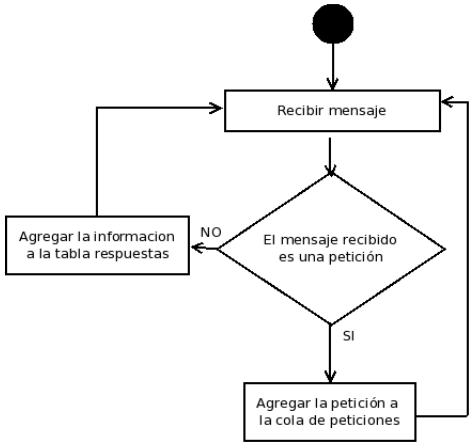

Figura 2.21: Diagrama de flujo que muestra al segundo hilo encargado del tratamiento de los mensajes recibidos en todo momento por el planificador.

Parámetro	Valores
γ	$[0, 40)$
$factor$	0
δ	0
m	6

Cuadro 2.1: Valores de los parámetros globales que requiere el algoritmo.

Los conjuntos de tareas que cada nodo debe planificar se muestran en el cuadro 2.2, los datos se eligieron de manera pseudoaleatoria, pero cuidando en todo momento que cada conjunto de tareas periódicas fuese planificable y su demanda menor a la unidad mediante la ecuación 2.16. La figura 2.23 muestra a los seis nodos que conforman al sistema distribuido representados como los vértices de un grafo, las aristas indican una relación: la existencia de la misma tarea esporádica almacenada en los nodos.

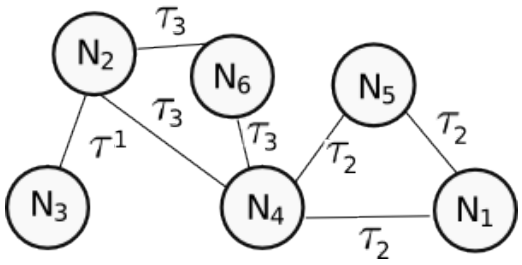

Figura 2.22: Grafo que muestra la relación entre los seis nodos que conforman al sistema.

En base a la información de la tabla 2.2 se calculan los consumos y periodos de las tareas servidor en cada nodo, según las ecuaciones 2.15 y 2.17, véase la tabla 2.3.

Por último para cada nodo N_{id} en el sistema, se definen los parámetros de marca de reloj (ΔT_{id}) y velocidad (V_{id}), tal que $id \in \{x \in \mathbb{N} : 1 \leq x \leq 6\}$, como se muestra en el cuadro 2.4.

Como se observa en el cuadro 2.4, el manejo de la velocidad de cada uno de los nodos se hace mediante un valor entero positivo.

La planificación ocurre en un sistema constituido por 6 nodos, todos ellos perfectamente sincronizados ($\delta = 0$) y con características diferentes de velocidad y marca de reloj y, además, la planificación interna de los nodos que conforman el sistema se basa en el algoritmo EDF.

Nodo	Γ		
	$\Gamma_{esporadicas}$	$\Gamma_{aperiodicas}$	$\Gamma_{periodicas}$
N_1	$\tau_2^E(101, 10, 11)$	$\tau_1^A(10, 0)$	$\tau_1^P(20, 5)$, $\tau_2^P(30, 8)$
N_2	$\tau_1^E(90, 11, 8)$, $\tau_3^E(67, 10, 12)$	ϕ	$\tau_3^P(25, 7)$, $\tau_4^P(30, 5)$, $\tau_5^P(20, 4)$
N_3	$\tau_1^E(90, 11, 8)$	ϕ	$\tau_6^P(15, 5)$
N_4	$\tau_2^E(101, 10, 11)$, $\tau_3^E(67, 10, 12)$	ϕ	$\tau_7^P(35, 15)$
N_5	$\tau_2^E(101, 10, 11)$	ϕ	$\tau_8^P(10, 4)$
N_6	$\tau_3^E(67, 10, 12)$	ϕ	$\tau_9^P(30, 5)$, $\tau_{10}^P(40, 17)$

Cuadro 2.2: Distribución de las tareas en los nodos.

Nodo	Tarea servidor			
	Nombre	C^S(Consumo)	T^S(Periodo)	$C_{consenso}$
N_1	τ_1^S	9	20	3
N_2	τ_2^S	6	24	2
N_3	τ_3^S	9	15	3
N_4	τ_4^S	18	33	6
N_5	τ_5^S	6	10	2
N_6	τ_6^S	10	30	2

Cuadro 2.3: Valores obtenidos para el consumo y periodo de las tareas servidor de cada nodo.

Dado que el tiempo de comunicación es despreciable en el sistema, cuando se envía la información relacionada a alguna tarea (liberada) en un instante de tiempo t, ocurrirá que dicha información se ha de recibir en el nodo destino en el mismo instante en el que la información fue liberada.

Para describir lo que ocurre en el algoritmo nos auxiliaremos de la figura 2.23. La primer tarea esporádica que se libera en el sistema es τ_1^E, en tiempo 8, dentro de los nodos 3 y 2 (N_3 y N_2). La segunda tarea en ser liberada es τ_2^E en los nodos 5,4 y 1 (N_5, N_4 y N_1 respectivamente) en el instante 11. Por último se libera τ_3^E, en el instante 12, en los nodos 6,4 y 2 (N_6, N_4 y N_2).

Cuando se libera τ_1^E en el nodo N_3 no es posible atenderla de manera inmediata, debido a ΔT_3, así que hasta que concluya la marca de reloj, es decir, en el instante 9 (hubo un retraso de una unidad de tiempo); en cambio, la misma tarea es atendida de manera inmediata en N_2.

Nodo	V_{id}	ΔT_{id}
N_1	40	1
N_2	35	2
N_3	30	3
N_4	38	3
N_5	40	1
N_6	43	2

Cuadro 2.4: Parámetros internos de velocidad y marca de reloj.

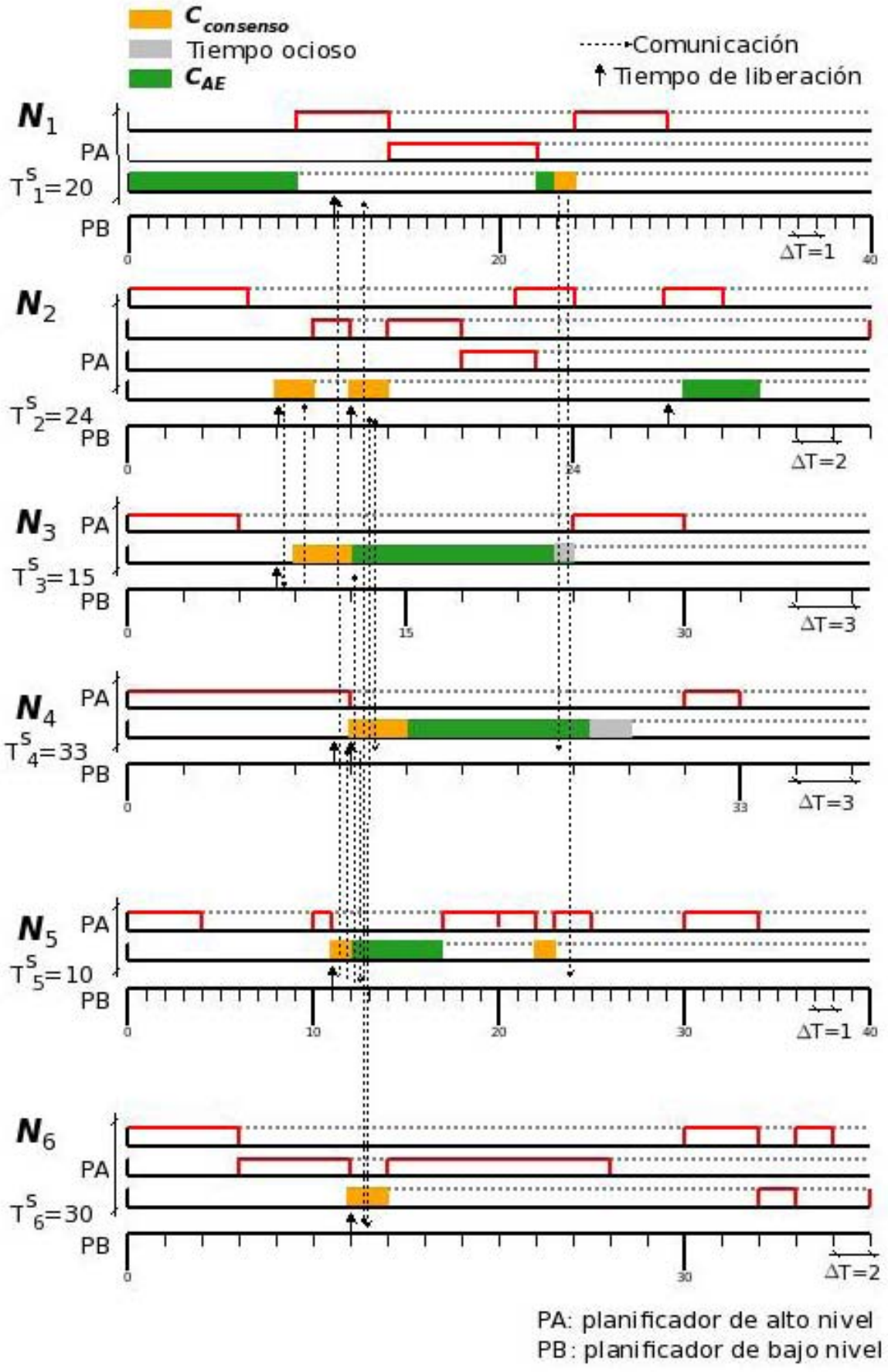

Figura 2.23: Ejemplo de la planificación de un conjunto de tareas en un sistema de 6 nodos.

Los valores intercambiados entre N_2 y N_3 son, por supuesto, los valores de velocidad (V_{id}) y horizonte (H_{id}):

1. El nodo N_2 envía a N_3, en tiempo 8, la siguiente información $H_2 = 12$ y $V_2 = 35$
2. El nodo N_3 envía a N_2, en tiempo 9, la siguiente información $H_3 = 30$ y $V_3 = 30$

Una vez que cada nodo posee la información del resto de los nodos emparentados, determina que nodo ofrece la menor proporción de carga de trabajo (< 1), de acuerdo a la fórmula 2.27:

$$\min_{N_{id} \in \Upsilon(\tau_1^E)} \left\{ id \text{ tal que } \min_{N_{id} \in \Upsilon(\tau_1^E)} \left\{ \frac{\frac{30}{35} * 11}{12} \approx 0.786, \frac{\frac{30}{30} * 11}{30} \approx 0.367 \right\} \right\}$$

el nodo al que corresponde la menor de las cargas de trabajo en ese momento es N_3, por lo tanto será ese nodo el que se hará cargo de τ_1^E.

Lo mismo se hace para τ_3^E y τ_2^E:

$$\min_{N_{id} \in \Upsilon(\tau_3^E)} \left\{ id \text{ tal que } \min_{N_{id} \in \Upsilon(\tau_3^E)} \left\{ \frac{\frac{35}{35} * 10}{8} \geq 1, \frac{\frac{35}{38} * 10}{12} \approx 0.38, \frac{\frac{35}{43} * 10}{8} \geq 1 \right\} \right\}$$

dado que la proporción calculada, a partir de la información compartida por N_2 y N_6, informa que su carga de trabajo se excede con la nueva tarea τ_3, ya que sus valores fueron mayores que uno, tomamos al nodo N_4 como el nodo más apto.

$$\min_{N_{id} \in \Upsilon(\tau_2^E)} \left\{ id \text{ tal que } \min_{N_{id} \in \Upsilon(\tau_2^E)} \left\{ \frac{\frac{38}{40} * 10}{18} \approx 0.527, \frac{\frac{38}{38} * 10}{24} \approx 0.417, \frac{\frac{38}{40} * 10}{36} \approx 0.264 \right\} \right\}$$

según los cálculos obtenidos el nodo que ofrece la menor carga de trabajo es N_5, por lo que es el más apto para ejecutar τ_2^E.

Para concluir, nótese que en la ejecución de esta instancia propuesta no hubo envió de peticiones por cambio de estado (revisar la sección 2.9.3), y por lo tanto sólo se ejecutó una ronda de consenso para cada tarea.

2.11. Comentarios del capítulo

En este capítulo se presentó la parte medular de este trabajo de tesis. La contribución aquí presentada consiste en una estrategia de planificación distribuida, basada en consenso y cuyas decisiones de planificación son tomadas durante la ejecución. También se especifican las condiciones bajo las cuales esta estrategia puede llevarse a cabo y; para concluir, se muestra el criterio de planificabilidad.

Se discutió el criterio propuesto para la elección del nodo apto para ejecutar una tarea esporádica en el sistema distribuido, éste se basa en la elección del nodo que ofrece la menor carga de trabajo: la menor proporción de la velocidad, de un nodo con las del resto de los nodos emparentados, en un horizonte.

En los algoritmos de planificación en tiempo real existe una constante indiscutible, es el de garantizar en todo momento el cumplimiento de los plazos de las tareas. Este trabajo no es la excepción, ya que las tareas aceptadas por algún nodo, previamente aprobaron un examen que garantiza su ejecución.

Uno factor que afecta el desempeño del algoritmo es la elección de la marca de reloj ΔT para un nodo. Si ΔT es muy grande, con respecto al tamaño del periodo más largo del conjunto de tareas periódicas en un nodo, entonces la mayoría de los eventos no serían atendidos inmediatamente sino hasta ocurrida la próxima marca, provocando la perdida de los deadlines de todas las tareas con periodos más cortos, y si es muy pequeño podría ocasionar muchas interrupciones al sistema que afectarían gravemente su desempeño.

En cuanto a la arquitectura que modela a cada nodo, una arquitectura jerárquica, de alguna forma da modularidad de los componentes del algoritmo y sus funcionalidades.

Dado que cada tarea esporádica es apropiativa (y sin perdidas) e inclusive el consenso, entonces la tarea servidor se ve en este análisis de suficiencia como una tarea periódica más, de hecho debe ser la tarea con el menor de los periodos.

El algoritmo propuesto es muy limitado ya que considera que el tiempo de comunicación es un valor despreciable y no hay retardos en las comunicaciones ni duplicados y ningún tipo de falla ocurre en alguno de los componentes. Dentro de cada nodo el cambio de contexto es instantáneo, es decir, el tiempo del cambio entre tareas es nulo y cada tarea planificada es totalmente apropiativa. Dado que existen varias tareas esporádicas iguales, entre los nodos que conforman el sistema distribuido, se requiere de mayor cantidad de memoria por nodo.

Capítulo 3

Implementación y resultados

Este capítulo detalla algunos de los aspectos considerados en la implementación de esta estrategia de planificación y se muestran los resultados obtenidos de las ejecuciones realizadas así como el análisis de los mismos.

3.1. Introducción

Se ha desarrollado un caso de estudio que explota las capacidades de un sistema distribuido heterogéneo para comprobar que el algoritmo propuesto funciona y satisface el principal objetivo que es el de lograr la ejecución de tareas esporádicas aceptadas dentro de sus respectivos plazos después de haber ocurrido su respectivo consenso.

Los resultados son obtenidos de un simulador que recibe como entrada un archivo con las características de los diferentes tipos de tareas independientes que se pretenden ejecutar dentro de algún sistema distribuido heterogéneo libre de fallas.

El algoritmo utilizado por cada nodo del sistema distribuido para elegir a la tarea de mayor prioridad es EDF, el cual trabaja conjuntamente con el algoritmo del servidor aplazable [29, 55] para el manejo del presupuesto destinado a la tarea servidor (C^S).

3.2. Detalles de la implementación

Para la manipulación de las tareas en el simulador, se utilizará la clase `Task`, que contiene la información que permite identificar y conocer el avance de la ejecución de una tarea. En particular, la clase `Task` contiene todos los parámetros especificados por el programador en tiempo de creación, más información temporal necesaria que permitirá administrar una tarea. Los miembros principales de esta clase contienen la siguiente información:

- Un *identificador global* con el que se distingue a una tarea a través del sistema distribuido, dicho valor es asignado por el programador.
- El *estatus* que refleja el estado actual en el que se encuentra la tarea dentro del nodo, en la figura 2.1 se muestran los estados por los que pasa una tarea durante su ejecución.
- Un *consumo* que especifique el número de unidades de tiempo que una tarea debe ser ejecutada.

- El tiempo de *liberación* que delimita el tiempo en el que una tarea puede comenzar su ejecución.

Dado que existen varios tipos de tareas en el sistema la clase `Task` se especializa en otros tres tipos de tareas (vea figura 3.1): tareas periódicas (clase `Periodic`), tareas aperiódicas (clase `Aperiodic`), tareas esporádicas (clase `Sporadic`) y, por último, la tarea servidor (clase `Server`) responsable de brindar atención a los dos últimos tipos de tareas mencionados, a las actividades de consenso y dar atención a las peticiones.

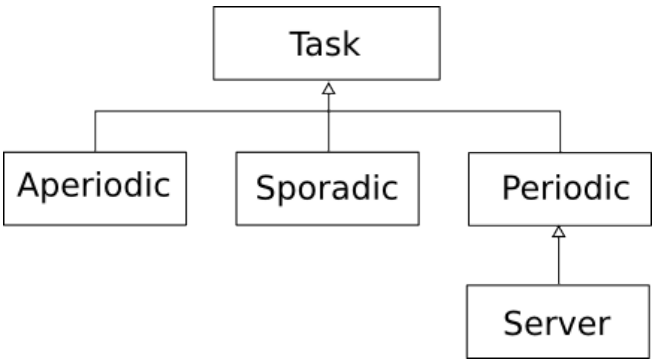

Figura 3.1: Especializaciones de la clase Task para el manejo de los diferentes tipos de tareas en el simulador.

Recordemos que una misma tarea esporádica es asignada a varios nodos, que a causa de la heterogeneidad no pueden ser asignadas de cualquier forma entre los nodos, así que tal asignación debe hacerse de manera que los requerimientos de cada tarea (procesamiento, memoria, etc.) se puedan satisfacer en el nodo que la contendrá.

Al final de la asignación de las tareas especificadas en el archivo, en cada nodo debe haber, al menos, una tarea esporádica y una periódica, de lo contrario el nodo no se considera para de la ejecución del algoritmo en el sistema distribuido, además, en cada nodo, es necesario guardar un registro de las tareas esporádicas con la posibilidad de ser ejecutadas y el del resto de los nodos que también las contienen. Dicha información se almacena, durante el lapso definido por γ, en una estructura de datos como la mostrada en la figura 3.2, además, dicha estructura es necesaria, al momento de decidir, para conocer cuando se ha recibido la información completa de una tarea en la tabla `respuestas`.

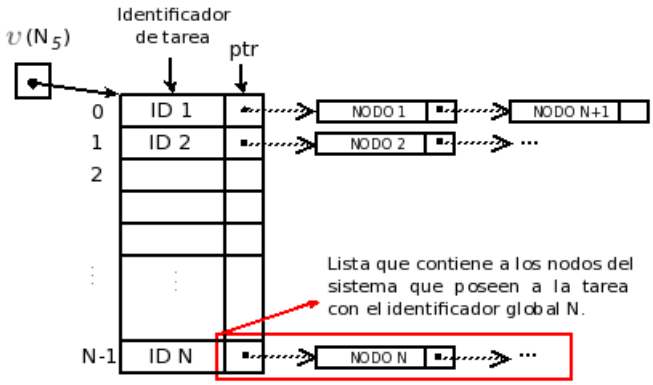

Figura 3.2: Tabla que indica los subconjuntos de nodos que poseen una misma tarea.

Las tareas esporádicas aún suspendidas con posibilidad de ser ejecutadas se colocan en una lista ordenada, de forma ascendente según el valor de sus tiempos de liberación. Esta estructura de datos ha sido nombrada como `sporadicInitList` ($= \Gamma_{esporadicas}$) en la implementación.

El propósito de este algoritmo es garantizar la ejecución de tareas esporádicas aceptadas en un nodo, dentro de un presupuesto de tiempo, C^S_{AE}, asignado cada periodo, T^S, según se especifique para la tarea servidor de cada nodo. Enfoquemos la atención en el trato de tareas esporádicas dado que éstas poseen plazos que hay que cumplir dentro de un tiempo específico; mientras que las tareas aperiódicas sólo se deben ejecutar sin que necesariamente se cumpla un plazo; a menos, que el programador así lo decida para evitar la condición de inanición de la que pueden ser objeto, en este caso se utiliza un método de penalización (sección 2.6).

Como ya se mencionó, el propósito de lograr un acuerdo es para encontrar el nodo más "apto" de entre un subconjunto de nodos, para la ejecución de una tarea esporádica. Ésto también consume tiempo de procesador, por lo que es necesario reservar un porcentaje del presupuesto asignado a la tarea servidor para el consenso, $C^S_{consenso}$.

El consenso se lleva a cabo mediante rondas de forma asíncrona, porque un nodo emparentado no espera la información (de una tarea esporádica) del resto de nodos emparentados, por el contrario, continua con su ejecución. Cada ronda consta de tres acciones:

1. **Intercambiar**, consiste en enviar y recibir la información de la tarea que se pretende activar, τ^E_i, entre aquellos nodos emparentados de $\Upsilon(\tau^E_i)$.
2. **Decidir**, una vez concluido el intercambio de información correspondiente a la tarea τ^E_i, se revisa la información que otros nodos enviaron de tareas previas a τ^E_i, e inclusive a ésta última.
3. **Atender** las peticiones de consenso que otros nodos hicieron al notar que su estado actual no permite la ejecución de tareas previas a τ^E_i (cambio su estado).

Inicialmente, todas las tareas consideradas dentro del sistema han sido asignadas a los diferentes nodos, y éstas se encuentran *suspendidas* mientras no se cumplan sus tiempos de liberación, es decir, no están ejecutando trabajo útil para el sistema distribuido, pero están cargadas en memoria en espera de la señal de reloj que las active.

Sólo cuando se extrae una tarea esporádica de $\Gamma_{esporadicas}$ comienza el intercambio de información con el resto de los nodos en el subconjunto de nodos emparentados, para tomar la decisión sobre *cual es el nodo más apto del subconjunto para ejecutar aquella tarea esporádica.*

Cada nodo tiene dos hilos de ejecución en paralelo. Existe un hilo encargado de planificar y otro hilo que recibe mensajes que contienen información o peticiones enviadas por otros nodos, en todo momento. Los mensajes con la información de alguna tarea esporádica se recibirán en una tabla, respuestas (en la implementación tendrá el nombre de `answers`) como la mostrada en la figura 3.3, mientras que las peticiones serán recibidas en otra cola (llamada `requests` en la implementación, vea la figura 3.4).

La información que se encuentra en la tabla *respuestas* (`answers`) se utiliza, en conjunto con $v(N_{id})$, para decidir localmente si una tarea τ^E debe ejecutarse en un nodo N_{id}. La información en la tabla *respuestas* se actualiza al final de cada lapso definido por el intervalo γ.

La información que se almacena en la lista *peticiones* (`requests`) corresponde a las solicitudes hechas por otros nodos, para comenzar, de nueva cuenta, el consenso para determinar un nuevo lugar donde ejecutar una tarea para la que ya se había decidido el nodo que la activaría pero que, por el incremento en su utilidad (vaya a la sección 2.9.3), no puede hacerse cargo de la tarea actualmente.

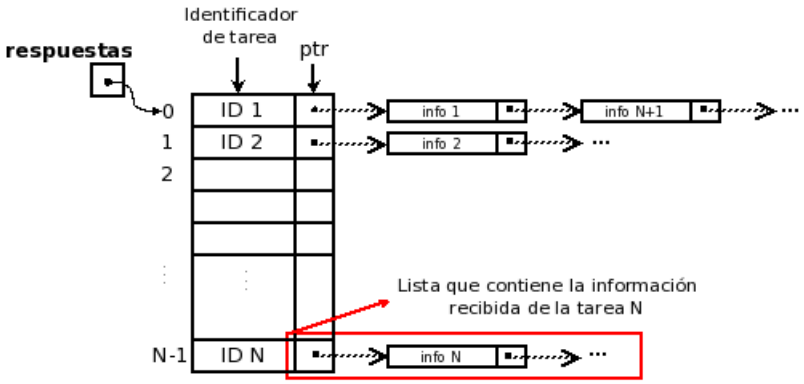

Figura 3.3: Tabla *respuestas*, en ella se almacena la información recibida del resto de los nodos emparentados.

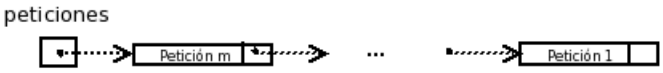

Figura 3.4: Cola en la que se almacenan las peticiones enviadas por aquellos nodos que al percatarse de la imposibilidad de ejecutar una tarea localmente debido a que hubo un cambio de estado en el nodo.

3.3. Descripción del caso de estudio

Cada nodo del sistema emplea la misma estrategia de planificación y la misma para el manejo del presupuesto. Se decidió llevar a cabo la experimentación para planificar un conjunto de tareas mediante el algoritmo EDF, que es auxiliado por el servidor aplazable [29, 55] para el manejo del presupuesto.

Las velocidades varían entre los nodos del sistema, así como sus marcas de reloj y los relojes de los nodos en el sistema se encuentran sincronizados. Diremos que un conjunto de nodos se encuentra sincronizado si la diferencia de los relojes de cualesquiera dos nodos en el sistema se encuentra acotada por el intervalo $[-\delta, \delta]$.

El simulador implementado (bajo el paradigma de la programación orientada a objetos) recibe un archivo, como el mostrado en la figura 3.5, con un listado de las tareas que son asignadas a los nodos del sistema distribuido mediante probabilidades (la probabilidad asumirá la responsabilidad de determinar si el nodo es capaz de cubrir las necesidades de una tarea).

Antes de comenzar con la simulación, se creó un archivo con un listado extenso de tareas que el planificador podría ejecutar (esporádicas, aperiódicas y periódicas). Luego las tareas ahí especificadas fueron asignadas a los nodos cuidando que al menos una tarea periódica fuera asignada a cada nodo, las tareas aperiódicas se asignaron de manera aleatoria y las tareas esporádicas fueron asignadas de la siguiente forma: se hizo un recorrido por el conjunto de nodos que conforman al sistema distribuido, para cada tarea esporádica especificada en el archivo se iba generando un valor aleatorio, dentro de un intervalo definido por el programador, $[0, \varepsilon]$, tal que $\varepsilon << 1$. Si el valor generado caía dentro del intervalo, era posible asignar la nueva tarea esporádica al nodo. De esta manera era posible garantizar la existencia de la misma tarea esporádica dentro de diferentes nodos (se garantiza la redundancia de una misma tarea).

Sin embargo, para la asignación de tareas periódicas, cada nodo hace un análisis para determinar que la tarea periódica que será agregada no sobrepasará un valor de utilidad, σ, también definido por el programador. Cada vez que se agrega una tarea periódica, τ_j, a un nodo N_{id}, se

```
.....
s 81,7,390,6
s 82,4,491,1
s 83,9,459,8
a 84,4,89
p 85,9,384
p 86,2,229
s 87,19,46,17
p 88,10,438
s 89,6,160,4
s 90,20,120,14
p 91,3,129
a 92,14,43
s 93,8,263,0
a 94,17,33
p 95,15,114
p 96,1,158
.....
```

Figura 3.5: Fragmento del listado en un archivo utilizado para la inicialización de una instancia. Los caracteres a, p y s indican que se trata de tareas aperiódicas, periodicas y esporádicas respectivamente. Las secuencias de números ahí mostradas son los valores de los parámetros de configuración de las tareas. Las tareas esporádicas se representan con la secuencia s id,C,d,r; las tareas aperiódicas como a id,C,r y, por último, las tareas periódicas p id,C,T; donde, id es el identificador global, C es el consumo, r el tiempo de liberación, d es el plazo de tiempo y T el valor del periodo.

revisa que se cumpla la condición (ecuación 2.16)

$$\sum_{\tau_i \in \Gamma_{periodicas}} \frac{\lceil \frac{C_i}{\Delta T_{id}} \rceil}{\lceil \frac{T_i}{\Delta T_{id}} \rceil} + \frac{\lceil \frac{C_j}{\Delta T_{id}} \rceil}{\lceil \frac{T_j}{\Delta T_{id}} \rceil} < \sigma << 1$$

con esto las tareas periódicas no ocupan en su totalidad la utilidad de un nodo y, como consecuencia, se asegura una utilización de, al menos, $1 - \sigma$ para la tarea servidor.

Una vez que se han agotado las tareas en el archivo y que cada nodo en el sistema posee su conjunto de tareas, el algoritmo comienza su labor. Cada nodo comienza su ejecución, pero no al mismo tiempo, a causa del criterio definido de la sincronización de relojes algunos nodos comienzan antes y otros después de un tiempo, que se toma como referencia.

3.4. Experimentación y resultados

Las condiciones sobre las que se realizó la experimentación se listan a continuación:

1. La velocidad de los procesadores se representa con un número entero elegido arbitrariamente entre 1000 y 2000.
2. La marca de reloj de cada nodo en el sistema fue elegida aleatoriamente: el valor de la marca de reloj para el nodo N_{id} se eligió de manera aleatoria de un conjunto de tres valores $\Delta T_{id} \in \{1, 2, 3\}$, donde id es el valor del identificador del nodo.
3. No se efectuó la penalización a la ejecución de tareas esporádicas ($factor = 0$).
4. El desfase de los relojes de los nodos que conforman al sistema es de $\delta = 3$, es decir, a lo más habrá 6 unidades de tiempo de desfase entre cualesquiera dos nodos del sistema.
5. La repartición de las tareas esporádicas a los nodos se realizó de forma aleatoria.
6. El intervalo de tiempo sobre el que se ejecutó el algoritmo es $\gamma = [0, 500]$

Profundicemos un poco más en la manera en que las tareas fueron asignadas a los nodos. Por cada tarea esporádica, se realizó un recorrido de los nodos en el sistema, luego por cada nodo se generó un valor aleatorio en el rango $[0, 1]$, si éste era menor o igual a un valor ε, entonces la tarea era insertada en el nodo. Dicho algoritmo se detalla en la función 4.

Función 4 Repartición de tareas

Entrada: Listado de tareas *listado*

```
1:  n ← 0
2:  para todo τ_j ∈ listado hacer
3:    si τ_j es esporádica entonces
4:      para todo nodo (N_K) en el sistema distribuido hacer
5:        aleatorio ← genera_aleatorio()
6:        si aleatorio < ε entonces
7:          insertar(N_K, τ_j, Γ_esporadicas)
8:        fin si
9:      fin para
10:   si no
11:     si τ_j es aperiódica entonces
12:       aleatorio ← genera_aleatorio()  mód m
13:       insertar(N_aleatorio, τ_j, Γ_aperiodicas)
14:     si no
15:       insertar(N_n mód m, τ_j, Γ_periodicas)
16:       n ← n + 1
17:     fin si
18:   fin si
19: fin para
```

Para los resultados que se mostraran más adelante se eligió el valor de $\varepsilon = 0.2$, la elección de este valor se hizo para garantizar una baja incidencia de la misma tarea sobre varios de los nodos que conforman al sistema.

3.5. Eficiencia en el uso del horizonte

Para determinar la factibilidad del cálculo del horizonte como medida del tiempo disponible en un nodo para la ejecución de una tarea esporádica recientemente liberada analicemos lo que ocurre en algunas ejecuciones de 5, 10, 20 y 30 nodos.

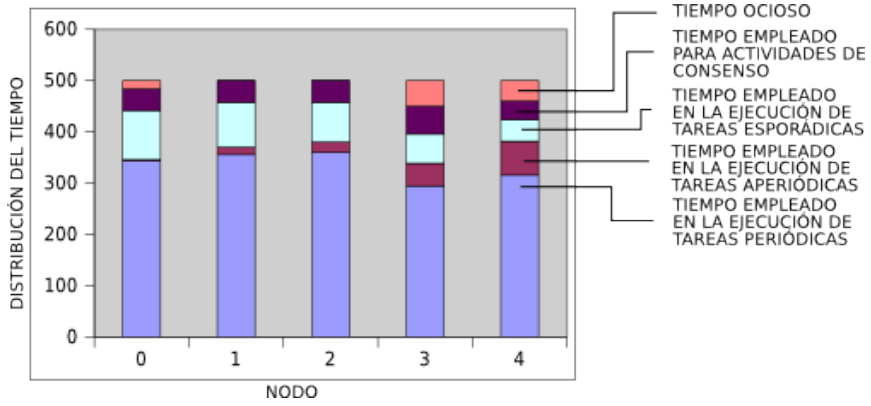

Figura 3.6: Uso del tiempo en un sistema conformado por 5 nodos durante el intervalo γ.

Como se observa en la gráfica 3.6, se tiene un sistema conformado por 5 nodos, lo que implica que un nodo puede, a lo más, estar emparentado con 4 nodos, dicha gráfica muestra la sumatoria de los tiempos de las actividades desarrolladas por nodo durante el intervalo definido por $\gamma = [0, 500]$.

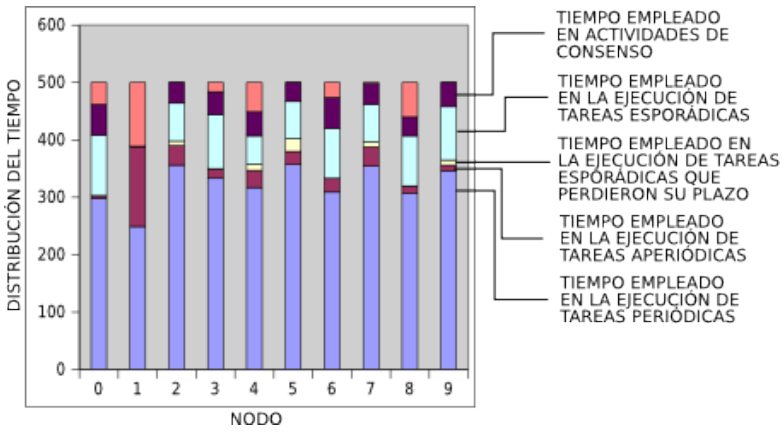

Figura 3.7: Uso del tiempo en un sistema conformado por 10 nodos durante el intervalo γ.

La figura 3.7 muestra la forma en que 10 nodos emplearon el tiempo durante el intervalo de ejecución (γ), en este caso cada nodo puede estar, a lo más, emparentado con otros 9 nodos. Como se observa, aparece una sección de tiempo nueva que indica la cantidad de tiempo que fue empleada ejecutando tareas esporádicas cuyos plazos no fueron satisfechos. Esto evidencia que la medida del horizonte de tiempo (ecuación 2.25) ha fallado, sin embargo, según los resultados obtenidos y registrados en la tabla 3.4 la cantidad de tiempo mal empleado (el tiempo que es utilizado para ejecutar tareas esporádicas aceptadas por consenso cuyos plazos no fueron cumplidos) es menor comparado con el tiempo empleado para la ejecución de tareas esporádicas ejecutadas correctamente (tareas aceptadas por consenso que se ejecutaron dentro de sus plazos).

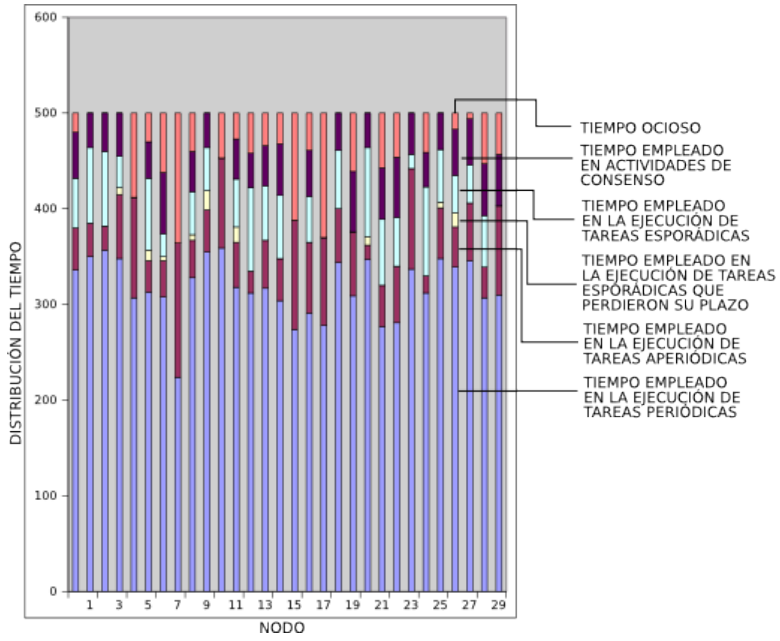

Figura 3.8: Uso del tiempo en un sistema conformado por 30 nodos durante el intervalo γ.

La figura 3.8 muestra la manera en que un conjunto de 30 nodos emplean el tiempo durante el intervalo γ, en este caso un nodo puede, a lo más, estar emparentado con otros 29 nodos.

Los resultados mostrados por las gráficas 3.8, 3.7 y 3.6 se resumen en la tabla 3.4.

	5 nodos	10 nodos	20 nodos	30 nodos
Tareas periódicas (%)	66.84	64.52	66.44	63.57
Tareas aperiódicas (%)	5.84	6.6	4.29	11.67
Tareas esporádicas no concluidas (%)	0.00	1.18	1.12	0.63
Tareas esporádicas (%)	14.44	14.22	14.96	8.73
Actividades de consenso (%)	8.64	7.4	8.98	7.65
Tiempo ocioso (%)	4.24	6.08	4.21	7.75

Cuadro 3.1: Porcentaje del aprovechamiento promedio del presupuesto asignado a las tareas servidoras de los diferentes nodos del sistema, durante el intervalo γ para la ejecución de tareas esporádicas y aperiódicas.

3.6. Otros resultados

Además de los resultados anteriormente mostrados, vale la pena mostrar otro tipo de relaciones entre los parámetros, como la cantidad de tiempo invertido en la ejecución de tareas esporádicas conforme aumentan los nodos.

3.6.1. Relación entre tareas esporádicas y el número de nodos

Para determinar que la cantidad de nodos en el sistema influye en la cantidad de tiempo empleado para la ejecución de tareas esporádicas se realizó un nuevo experimento.

Se generaron dos conjuntos de experimentos, en cada uno se cambió la probabilidad de asignar una tarea esporádica entre los nodos del sistema distribuido, para el primer conjunto la probabilidad fue de $\varepsilon = 0.8$, mientras que para el segundo conjunto la probabilidad se especificó en $\varepsilon = 0.2$. La forma de asignar las tareas se resume en el algoritmo de repartición de tareas 4. Para ambos conjuntos se generaron once instancias cuyo número de nodos variaba entre 5 y 50, en múltiplos de cinco. Es decir, hubo 220 instancias distintas ejecutadas.

El listado de tareas era lo suficientemente grande como para garantizar una carga de trabajo que mantuviese ocupados, la mayor parte de tiempo, a los procesadores de cada nodo a lo largo del tiempo definido por γ. Se comenzó la experimentación con 5 nodos, y por cada nodo se generaron 170 tareas (entre ellas hay tareas periódicas, esporádicas y aperiódicas), luego el número de nodos se fue incrementando en múltiplos de cinco hasta llegar a 50 (las siguientes instancias, de 55 nodos, tomaban aproximadamente 30 min. en concluir).

Para entender mejor el proceso analicemos una de las ejecuciones de este algoritmo. Esta instancia consta de cinco nodos ($m = 5$) y la manera en que fue aprovechado el presupuesto de la tarea servidor se muestra en la figura 3.9. Para el primer nodo (N_1) se contabilizó un lapso de tiempo de 150 unidades de tiempo disponible, de las cuales 21 unidades fueron aprovechadas por tareas esporádicas y 99 unidades aprovechadas por tareas aperiódicas, los tiempos del resto de los nodos se muestran en la tabla 3.4.

Luego, para cada instancia, el tiempo disponible total obtenido de cada nodo se sumó, por un lado, mientras que lo mismo se hizo para el tiempo aprovechado por tareas esporádicas y, lo mismo, en aperiódicas como se reporta en la tabla 3.3.

Por cada instancia se obtuvo la sumatoria mostrada en la gráfica 3.10. Continuando con la instancia antes mencionada, resulta que es la número 7, si realizamos la sumatoria de los tiempos de aprovechamiento y la de tiempos libres entonces podremos observar que se tuvo un total de

Nodo	Aprovechado por esporádicas	Aprovechado por aperiódicas	Tiempo total disponible
N_1	21	99	150
N_2	13	113	167
N_3	15	98	166
N_4	30	82	112
N_5	60	40	100

Cuadro 3.2: Aprovechamiento del presupuesto asignado a la tarea servidor durante el intervalo γ para la ejecución de tareas esporádicas y aperiódicas.

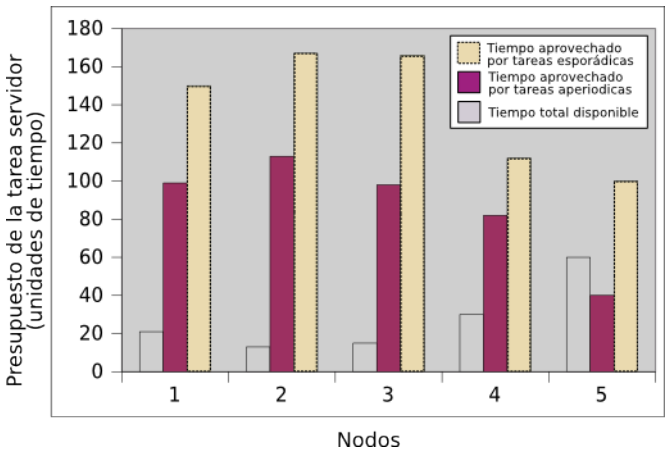

Figura 3.9: Gráfica que muestra el aprovechamiento del presupuesto para la tarea servidor en cada uno de los nodos, en la simulación de una instancia con cinco nodos en el sistema.

139 unidades de tiempo utilizadas por tareas esporádicas, 432 unidades de tiempo utilizadas para la ejecución de tareas aperiódicas y que el sistema distribuido dentro del que se ejecutó el algoritmo hubo un tiempo disponible de 695 unidades en el sistema visto de manera global.

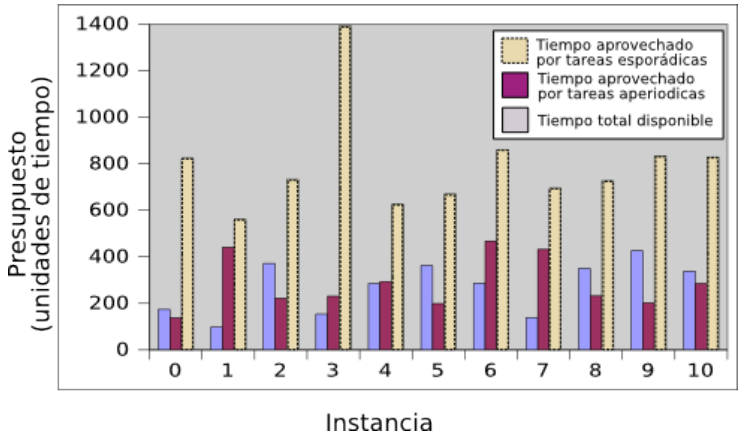

Figura 3.10: Tiempos totales de presupuesto por instancia en el intervalo γ.

De las instancias se obtuvieron resultados que después fueron promediados con los resultados del resto de las instancias (once instancias totales) y se listan en la tabla 3.4.

Para mostrar de manera global el comportamiento del algoritmo. Los resultados mostrados en la tabla 3.4 muestran los porcentajes del promedio del presupuesto aprovechado dentro del intervalo γ, obtenido después de la ejecución de 11 instancias por cada conjunto de nodos de

Nodo	Aprovechado por esporádicas	Aprovechado por aperiódicas	Tiempo total disponible
0	174	138	824
1	98	441	561
2	372	222	733
3	154	231	1393
4	285	294	626
5	362	199	670
6	286	468	861
7	139	432	695
8	350	232	727
9	426	202	834
10	338	285	829

Cuadro 3.3: Aprovechamiento del presupuesto asignado a la tarea servidor durante el intervalo γ para la ejecución de tareas esporádicas y aperiódicas.

Número de nodos	% Presupuesto servidor	% Esporádicas	% Aperiodicas
5	78.121	40.917	37.204
10	77.328	18.587	58.742
15	72.963	12.342	60.621
20	65.203	3.574	61.629
25	76.608	3.137	73.471
30	63.870	3.827	60.043
35	70.288	2.133	68.154
40	75.329	0.376	74.953
45	53.718	3.289	50.429
50	57.230	1.597	55.633
55	85.251	0.201	85.050

Cuadro 3.4: Valores promedio del presupuesto aprovechado para la ejecución de tareas esporádicas y aperiódicas dentro de γ.

tamaño $5n$, donde $n \in \mathbb{N}$. Si observamos los resultados obtenidos para 5 nodos, en la columna que muestra el porcentaje promedio del presupuesto del servidor se observa que el 78.121 % del presupuesto destinado a la ejecución de tareas esporádicas y aperiódicas fue aprovechado y el tiempo restante fue tiempo que los procesadores de esas instancias se mantuvieron ociosas dentro del presupuesto destinado a la tarea servidor. De este presupuesto aprovechado se desglosan dos rubros: el porcentaje promedio que fue ocupado por tareas esporádicas y el ocupado por aperiodicas. Para 5 nodos, se observa que el 40.917 % del presupuesto destinado, dentro de γ, fue para la ejecución de tareas esporádicas y el otro 37.204 % restante para la ejecución de tareas aperiódicas.

Si observamos en la tabla 3.4, los mejores porcentajes de aprovechamiento del presupuesto de la tarea servidor para tareas esporádicas fueron obtenidos cuando el número de nodos emparentados era relativamente pequeño (por ejemplo, para instancias de 5 nodos, el tamaño máximo en un conjunto de nodos emparentados era 5).

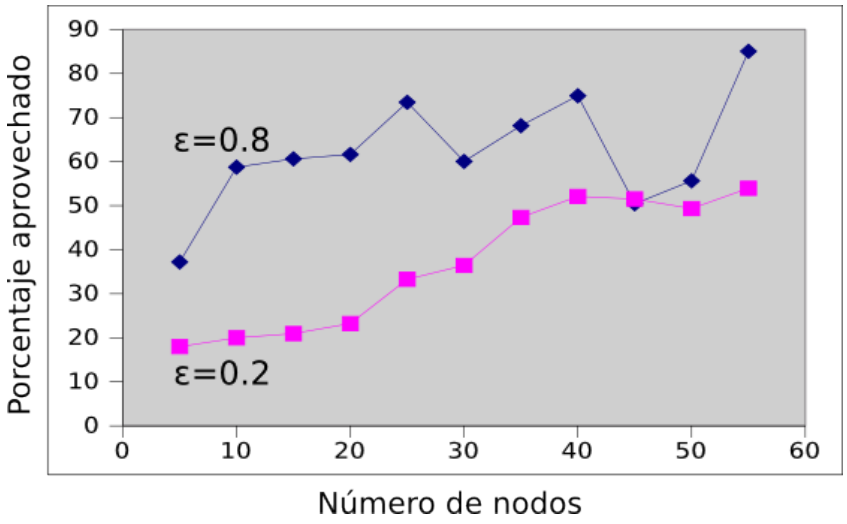

Figura 3.11: Gráfica que muestra el porcentaje utilizado del presupuesto destinado, durante γ, para tareas aperiodicas.

Nótese en la figura 3.11 como va incrementando el porcentaje del tiempo promedio aprovechado por las tareas aperiódicas conforme incrementa el número de nodos en el sistema. El fenómeno se debe a que al existir mayor cantidad de nodos en el sistema, el tamaño de los subconjuntos de nodos emparentados aumenta, provocando que el consenso tuviera mayor duración entre los subgrupos (de nodos emparentados). Este retardo causado por la etapa de decisión para lograr el consenso provoca en los nodos una carencia de tareas esporádicas en estado de lista, por lo que el planificador procede a ejecutar tareas aperiódicas.

Obsérvese la figura 3.12 el decremento en la ejecución de tareas esporádicas, éste efecto se da gracias a que el aumento en el número de nodos en el sistema conlleva a un aumento en el tamaño de los conjuntos de nodos emparentados y a un aumento en el número de subgrupos de nodos emparentados, a saber, tantos subconjuntos de nodos emparentados como tareas esporádicas haya en el sistema. Todo ello, a su vez, implica que los retardos en la toma de decisiones sean cada vez mayores y que ocurran los cambios de estado con frecuencia (tratado en la sección 2.9.3).

Los resultados obtenidos para la planificación de tareas esporádicas y aperiódicas muestran que del presupuesto disponible, dentro del intervalo γ, hubo un aprovechamiento del presupuesto que oscila entre el 53 % y 86 %, diríjase a la gráfica 3.13.

La figura 3.13 muestra un panorama general de la actividad de las unidades de procesamiento

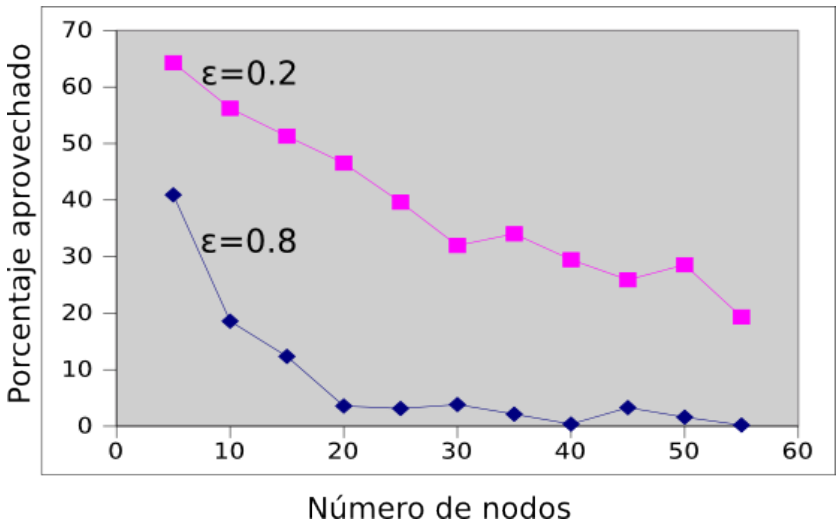

Figura 3.12: Porcentaje aprovechado para la ejecución de tareas esporádicas.

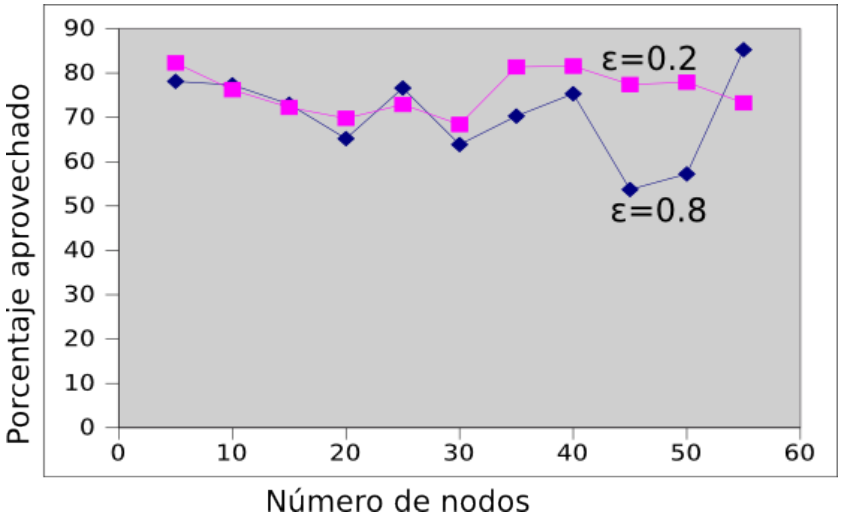

Figura 3.13: Porcentajes del presupuesto promedio del tiempo utilizado en la ejecución de tareas esporádicas y aperiódicas.

de las instancias producidas y ejecutadas. Observemos como los procesadores se mantuvieron ocupados por encima del 53 % según el promedio obtenido de las once instancias ejecutadas.

3.7. Conclusiones

La razón de ser de este algoritmo surge a partir de la necesidad de encontrar una estrategia de planificación distribuida cuyas decisiones se tomen en tiempo de ejecución y que permita obtener resultados satisfactorios en cuanto a la ejecución de tareas esporádicas aceptadas (después de haber llegado a un acuerdo) dentro de sus restricciones temporales.

El objetivo, de satisfacer los plazos de tareas esporádicas aceptadas, fue alcanzado, pero con un error, ya que existe la posibilidad de que los nodos acepten tareas esporádicas que después no son ejecutadas dentro de sus plazos temporales. Al parecer, los errores arrojados por las simulaciones muestran que dicho error no sobrepasa el 4 % de tiempo de la ejecución dentro de un nodo, es decir, dentro del intervalo γ, en este caso de longitud 500 (unidades de tiempo), la cantidad de tiempo, inútilmente utilizado, máximo identificado es de 20 unidades de tiempo (el 4 % del intervalo γ).

La distribución de tareas realizada de manera aleatoria implica que no necesariamente se tiene una repartición justa de tareas entre nodos, ya que no se están considerando sus capacidades, pero esta forma es veloz a comparación de las estrategias que realizan algún tipo de procesamiento previo a la repartición para lograr una planificación (la planificación se hace offline y de manera centralizada).

Conforme aumenta el número de nodos en el sistema distribuido, aumenta el número y, posiblemente, el tamaño de los subconjuntos de nodos emparentados, por la posesión común de una misma tarea; por lo que cada nodo podría participar en un mayor número de acuerdos, lo que afecta considerablemente a la ejecución de tareas esporádicas, sin embargo favorece a la ejecución de tareas aperiódicas. Por lo tanto el número de tareas esporádicas sí ve mermada su ejecución cuando hay consenso de por medio.

Conclusiones

Los algoritmos de planificación permiten el trabajo en sistemas de computación, garantizando la factibilidad en el uso de los recursos, ya que varias aplicaciones y sistemas dependen de estrategias que determinen el orden en el que una serie de acciones, eventos o tareas deban ser llevadas a cabo. Ejemplos claros de ello son los sistemas operativos, aplicaciones para tiempo real, etcétera.

La estrategia de planificación utilizada se basa en el uso de los recursos disponibles entre los nodos que conforman al sistema distribuido para lograr un acuerdo mediante consenso.

Se propuso una estrategia de planificación global a partir de posibles acciones de consenso locales entre diversos nodos. La técnica de consenso aquí usada nos permite tomar decisiones de la manera más rápida posible a costo de inundar la red de comunicaciones.

Una buena parte de los objetivos perseguidos en este trabajo fueron logrados. El principal objetivo, que era el de definir una estrategia global de planificación basada en consenso fue alcanzado, bajo ciertas condiciones:

- Existe la posibilidad de aceptar tareas que después no podrán ser ejecutadas, lo que representa una porción de tiempo inútilmente empleada por nodo.
- No se consideran tiempos de cambios de contexto entre tareas y los tiempos de comunicación entre nodos son despreciables.
- La planificación se lleva a cabo durante el tiempo en el que el sistema se encuentra estático (γ), y dicho lapso de tiempo se conoce de antemano.
- Los nodos que componen el sistema se encuentran sincronizados (la diferencia de tiempo entre cualesquiera dos nodos del sistema queda acotada dentro de un intervalo de $[-\delta, \delta]$ unidades de tiempo).
- No hay variaciones en los tiempos de ejecución de las tareas, ya que son siempre constantes.
- Cada nodo puede tener implementada una estrategia de planificación distinta, pero ésta siempre debe reservar tiempo para una tarea servidor.
- El consumo de cualquier tarea esporádica es el tiempo que lleva ejecutar la tarea en el nodo emparentado más lento. La decisión del nodo más apto, para una tarea, se toma mediante dicha consideración.
- Cada nodo tiene un procesador capaz de soportar dos hilos de ejecución paralelos: uno dedicado únicamente a la recepción de los mensajes y otro encargado de planificar.

No fue necesario establecer más de una estrategia de planificación dentro de un mismo nodo. Uno de los objetivos de este trabajo era el de modificar la estrategia de planificación de acuerdo

a las condiciones que se iban dando durante la ejecución del algoritmo, sin embargo como sólo se buscaba que el algoritmo ejecutará las tareas aceptadas antes de su plazo, no fue necesaria dicha implementación, a menos, que hubiese cambiado el propósito de la planificación por el de reducir el tiempo de ejecución promedio de las tareas, por ejemplo.

Gracias al conjunto de pruebas realizadas se constata que es posible ejecutar tareas esporádicas aceptadas mediante el criterio de la carga de trabajo en un horizonte de tiempo, pero con pérdida de tareas esporádicas previamente aceptadas. Eso lo revela el tiempo empleado para la ejecución de tareas esporádicas que perdieron su plazo.

En cuanto a la planificación de tareas esporádicas, los resultados obtenidos son:

- Entre mayor sea el número de nodos poseedores de una misma tarea esporádica, mayor es la posibilidad de que ésta sea ejecutada pero a costa de una reducción considerable en la ejecución de tareas esporádicas debido al aumento de tiempo que tarda un nodo en recolectar toda la información necesaria para llegar a un consenso, condición que favorece a la ejecución de tareas aperiódicas.
- El cálculo de la carga de trabajo existente en un nodo en un horizonte de tiempo es suficiente para garantizar que es altamente probable que una nueva tarea esporádica puede llevarse a cabo, y no ser rechazada aun después de haber sido aceptada.

Los resultados de la ejecución de las tareas aperiódicas fueron alentadores:

- Gracias a la condición que permite la ejecución de tareas aperiódicas en ausencia de tareas esporádicas, se consiguió hacer uso de una buena parte del presupuesto designado a la ejecución de tareas esporádicas y aperiódicas.

En general, el uso del procesador se mantuvo en un nivel aceptable:

- Se logró aprovechar el presupuesto, de la tarea servidor disponible, para tareas esporádicas y aperiódicas, $_{AE}$, durante el intervalo de tiempo especificado por γ, de manera tal que el procesador se mantiene ejecutando alguna tarea (siempre que ésta estuviera en espera).

Desventajas

1. La ocurrencia de fallas, de cualquier tipo, en el sistema evitará alcanzar acuerdos y con ello la ejecución de tareas esporádicas no se realizará.
2. Cada nodo en el sistema requiere de una gran cantidad de memoria, ya que debe conocer al resto de los nodos con los que se encuentra emparentado, así como las tareas esporádicas que podrá ejecutar.
3. Este algoritmo se limita a la ejecución de tareas independientes.
4. El consenso se lleva a cabo mediante la inundación de la red.
5. Un nodo no puede decidir, sino hasta haber recolectado toda la información del resto de los nodos emparentados.

Trabajo a futuro

Con el fin de mejorar este algoritmo (*"Planificación global de tareas confinadas basado en consenso"*), se deben agregar condiciones que en los requerimientos de cualquier sistema heterogéneo real pueden existir, lo que vuelve posible su implementación, entonces se debe extender el presente trabajo con los siguientes puntos:

- Identificar las condiciones por las que un nodo acepta tareas esporádicas que después no puede ejecutar y, posteriormente, hacer las modificaciones al criterio de aceptación de tareas para evitar este error.
- Modelar a través de modelos matemáticos el comportamiento de ejecución un conjunto de tareas, para luego proporcionar la mejor estrategia de planificación que optimice alguna función objetivo dada para dicho conjunto.
- Introducir incertidumbres en los tiempos de ejecución.
- Agregar un mecanismo para que el algoritmo sea capaz de adaptarse en el tiempo al recibir retroalimentación de las ejecuciones previas de un conjunto de tareas.
- Incluir y modelar el efecto de los retardos de comunicación y cambios de contexto durante la ejecución del algoritmo.
- Mejorar la estrategia de consenso, de tal forma que no sea necesario inundar la red de comunicaciones y agregar la habilidad de decidir en caso de pérdida de mensajes.
- Agregar la movilidad en los nodos del sistema para extender el modelo a redes ad hoc.
- Extender el modelo para que sea capaz de planificar tareas de diferentes tipos y permitir la existencia de dependencias entre tareas.

Apéndice A

Consenso

Los algoritmos que se presentan en las siguientes secciones son síncronos en el sentido de que cada nodo envía un conjunto de mensajes y entonces recibe replicas para aquellos mensajes. La instrucción enviar mensaje incluye solamente el identificador del nodo destino seguido de la información enviada, y la instrucción recibir sólo incluye variables para almacenar los datos que son recibidos.

A.0.1. Algoritmo de una ronda

El algoritmo de consenso más simple para $N > 2$ nodos se muestra enseguida.

Función 5 Consenso de una ronda

```
1:  accion: entero
2:  arreglo[N]: entero
3:  arreglo[miID] ← aleatorio MOD 2
4:  para todo otros nodos G hacer
5:      enviar(G,miID,arreglo[miID])
6:  fin para
7:  para todo otros nodos G hacer
8:      recibir(G, arreglo[G])
9:  fin para
10: accion ← mayoria(arreglo)
```

En este algoritmo, función 5, cada nodo elige uno de dos posibles valores (linea 3), lo envía al resto de los nodos y después recibe los valores de los otros nodos. El valor elegido finalmente es el valor que más veces haya aparecido. Nótese que cada nodo además de conocer su propio valor, conoce el valor de los otros nodos en el sistema. También es posible extender este escenario puede ser extendido a un número arbitrario de nodos.

Una observación importante es que este algoritmo no es válido para exactamente dos nodos ([38], pp.: 82-86).

A.0.2. Algoritmo de los generales bizantinos

Uno de los problemas al que los sistemas distribuidos se enfrentan es el desconocimiento de la fuente de los mensajes, cuando se desea lograr un acuerdo en un sistema distribuido se debe

asegurar que la información recibida en cada nodo sea la misma que en cualquier otro nodo que forma parte del sistema distribuido para así llegar a un mismo acuerdo. El algoritmo de los generales bizantinos [27] logra esto usando rondas extra de envío de mensajes: una primer ronda en la que los nodos envían su propio valor, y en las rondas subsecuentes los nodos se dedican a enviar la información que otros nodos le enviaron. Por definición un nodo correcto siempre retransmite lo que recibe; de esta manera, si hay suficientes nodos correctos, éstos pueden superar los intentos de los nodos incorrectos para evitar que se llegue a un consenso.

El siguiente algoritmo (mostrado en la función 6) consta de dos rondas. La primer ronda es la misma que la mostrada en la sección anterior; cuando ésta concluye, el arreglo contiene los valores del resto de los nodos. En la segunda ronda, la información recabada es enviada a los otros nodos. Obviamente, un nodo no se envía información a si mismo, ni tampoco envía de regreso a otro nodo que le reportó acerca de si mismo. Por lo tanto, en cada ronda se reduce en uno el número de mensajes que un general necesita enviar.

Función 6 Consenso de dos rondas

```
1:  accion: entero
2:  arreglo[N]: entero
3:  planReportado[N,N]: entero
4:  plan[N,N]: entero
5:  arreglo[miID] ← aleatorio MOD 2
6:  para todo otros nodos G hacer
7:     enviar(G,miID,arreglo[miID])
8:  fin para
9:  para todo otros nodos G hacer
10:    recibir(G, arreglo[G])
11: fin para
12: para todo otros nodos G hacer
13:    para todo otros nodos G' exceto G hacer
14:       enviar(G',miID,G,arreglo[G])
15:    fin para
16: fin para
17: para todo otros nodos G hacer
18:    para todo otros nodos G' exceto G hacer
19:       recibir(G,G',planReportado[G,G'])
20:    fin para
21: fin para
22: para todo otros nodos G hacer
23:    plan[G] ← mayoria(arreglo[G]∪planReportado[*,G])
24: fin para
25: plan[miID] ← arreglo[miID]
26: accion ← mayoria(plan)
```

La línea 14: `enviar(G',miID,G,arreglo[G])` significa enviar al nodo G' lo que el nodo con identificador `miID` recibio del nodo G, es decir, el conjunto de valores arreglo[G]. Cuando dicho mensaje es recibido es almacenado en `planReportado`, donde el valor del elemento del arreglo `planReportado[G,G']` es el valor que G reportó recibido de G'.

Las votaciones se efectúan en dos etapas. Para cualquier otro nodo G, se toma la mayoría de votos del plan directamente recibido de G (el valor en arreglo[G]) junto con los planes reportados por G recibidos de los otros nodos (denotado por `planReportado[*,G]`). Ésto es tomado como la intención del nodo G y es almacenado en `plan[G]`. La decisión final se toma tomando el valor que más veces se repite en `arreglo[G]`.

A.0.3. Consenso en sistemas síncronos

El que un sistema sea síncrono significa que cada una de sus ejecuciones consisten de una secuencia de *rondas*. Cada una de éstas se enumera con un valor r, donde $r \in \mathbb{N}$. Una ronda esta compuesta por tres fases consecutivas:

1. La fase de *envío* en la que cada proceso envía mensajes.
2. La fase de *recepción* en la que cada proceso recibe mensajes.
3. Una fase de *computo* durante la cual cada proceso procesa los mensajes recibidos en una ronda y realiza cálculos locales.

La propiedad fundamental de un modelo síncrono recae en el hecho de que *un mensaje enviado por un proceso p_i a un proceso p_j en la ronda r, es recibida por p_j en la misma ronda.*

Nótese que la propiedad de acuerdo sólo se aplica a procesos correctos[1], lo que permite a procesos con alguna falla decidir un valor diferente al dado por los procesos correctos. El *acuerdo uniforme* es una propiedad más restrictiva que previene dichos escenarios. De manera más precisa, el *consenso uniforme* se define por los requerimientos, anteriormente descritos, de terminación y validez más la siguiente propiedad de acuerdo uniforme: *cuales quiera dos procesos (correctos o no) deciden el mismo valor.*

Recordemos que los procesos con fallas bizantinas pueden decidir cualquier valor. Así que una especificación razonable para un protocolo de consenso bajo fallas bizantinas incluye terminación, acuerdo y la siguiente propiedad débil de validez: *Si todos los procesos correctos proponen el mismo valor v, entonces v es el valor decidido por un proceso correcto.*

A.0.4. Consenso en sistemas asíncronos

La asincronía se refiere a la imposibilidad de definir una frontera superior de tiempo sobre la planificación de los retardos y sobre los retardos en la transferencia de mensajes. Esto se debe al hecho de que ni la carga entrante de los usuarios ni la carga precisa de la red de comunicaciones puede ser predecida de manera precisa.

Cuando nos enfrentamos a resolver un problema que implica acuerdos en los sistemas reales es necesaria una transformación hacia el problema de consenso. A continuación se presentan sólo algunos ejemplos de ello.

Broadcast atómico

Generalmente los protocolos de broadcast atómico imponen un reparto de los mensajes en orden total, en inglés *total order delivery*, con el objetivo de garantizar la linearizabilidad, en inglés *linearizability*, el criterio que garantiza un comportamiento de secuencia aceptable. Generalmente este tipo de estrategias son indispensables en el manejo de transacciones en las que se desea mantener el orden en el que los eventos ocurrieron (p.e. una base de datos: el broadcast atómico asegura que un conjunto de réplicas se mantendrán consistentes siempre que apliquen las mismas transacciones en el mismo orden).

[1] Cuando el proceso sigue las especificaciones prescritas por el algoritmo.

Ha sido probado que este problema y el de consenso son equivalentes en sistemas asíncronos propensos a caídas por parte de los procesos [9]. Desde un punto de vista teórico, ésto significa que todos los problemas que involucran al consenso también son aplicables al broadcast atómico. Desde el punto de vista práctico, ésto significa que la solución al broadcast atómico puede darse a partir de una rutina de consenso, en otras palabras el problema del broadcast atómico se reduce a un problema de consenso.

Atomic Commitment

Formalmente el problema del *atomic commitment* en sistemas asíncronos puede definirse por las siguientes propiedades:

1. Terminación: cada proceso sin fallas eventualmente decidirá. Todos los procesos involucrados expresan su exito o falla sobre la ejecución de una transacción local votando por ejecutar o abortar.
2. Acuerdo: cualesquiera dos procesos no decidirán diferente. Es decir, si ningún proceso falla y todos los procesos votan por ejecutar entonces ejecutar será decidido. Si algún proceso decide abortar, abortar será decidido.
3. Validez: esta propiedad da su significado al valor decidido. Éste está compuesto de tres partes.
   a) Dominio de la decisión: el valor de la decisión es ejecutar o abortar.
   b) Justificación: si un proceso decide ejecutar, entonces todos los procesos han votado por ejecutar.
   c) Obligación: si todos los procesos votaron por si y ninguno de ellos fue percibido con fallas, entonces el valor de la decisión debe ser ejecutar.

Un posible algoritmo se describe a continuación:

1. El proceso coordinador envía una solicitud de voto (vote request) a los procesos participantes en la ejecución de la transacción.
2. Cuando los participantes reciben la solicitud de voto, responden enviando al coordinador un mensaje con su voto (*si* o *no*). Si un participante vota *no*, la transacción se aborta (abort).
3. El coordinador recoge los mensajes con los votos de todos los participantes. Si todos han votado *si*, entonces el coordinador también vota *si* y envía un mensaje *commit* a todos los participantes. En otro caso, el coordinador decide abandonar y envía un mensaje *abort* a todos los participantes que han votado afirmativamente.
4. Cada participante que ha votado si, espera del coordinador un mensaje commit o abort para terminar la transacción de forma normal o abortarla.

El algoritmo anterior consiste de dos fases porque se distinguen dos partes distintas, la fase de los votos (pasos 1 y 2) y la fase de decisión (pasos 3 y 4).

Los participantes tienen un periodo de incertidumbre que comienza cuando envían al coordinador el voto afirmativo (paso 2) y termina cuando reciben del coordinador los mensajes de commit o abort (paso 4), sin embargo, este periodo no existe para el coordinador que vota tan pronto como decide.

A.0.5. Técnicas de solución de consenso

Una manera de evitar el resultado de imposibilidad de FLP es considerando sistemas parcialmente síncronos, que son mas flexibles que los sistemas síncronos para ser usados como modelos de sistemas prácticos, y mas robustos que los sistemas asíncronos para dar solución al consenso en base en ellos [14]. Sin embargo, existen otras técnicas que dan solución al consenso.

Enmascaramiento de fallas

Para evitar el resultado de imposibilidad FLP es posible ocultar las fallas que ocurren en algunos procesos. Por ejemplo, los sistemas de transacciones implementan almacenamiento persistente, el cual es capaz de sobrellevar las fallas provocadas por caidas. Si un proceso se cae, entonces éste es reiniciado (de manera automática o por un admisnistrador). Los procesos colocan la información suficiente en memoria persistente de los puntos críticos del programa (información de su estado), de esta manera, si el programa se cae será reiniciado, el programa al iniciar encontrará la información necesaria como para ser capaz de retomar al ejecución de su tarea. Esto se verá como un proceso que trabaja de manera correcta pero que tarda un largo tiempo en realizar su procesamiento.

El acceso a esta memoria estable es usualmente una fuente de ineficiencia y debe ser evitada tanto como sea posible. Aguilera, Chen y Toueg [1] han probado que, el consenso puede ser resuelto sin utilizar memoria si se garantiza que el número de procesos que nunca fallan es mayor al número de procesos que presentan una falla, ya sea de omisión o caída.

Detectores de fallas

Otro método para eludir el problema de alcanzar consenso se basa en el uso de detectores de fallas. En esta estrategia los procesos pueden estar de acuerdo en considerar un proceso que no ha respondido a una petición, después de un tiempo dado, como incorrecto. Un proceso que no responde, no necesariamente es un proceso incorrecto, pero el resto de los procesos actúan como si hubiese cometido fallas. Éstos descartan cualquier mensaje posterior recibido de cualquier proceso incorrecto. En otras palabras, se ha transformado un sistema asíncrono a uno síncrono. Este método se basa en que los detectores de fallas son usualmente exactos. Cuando los detectores son inexactos, el sistema tiene que proceder sin un grupo de miembros que de otra manera hubieran podido contribuir a la efectividad del sistema. Desafortunadamente, para hacer a un detector de fallas razonablemente exacto implica usar grandes valores de tiempo, forzando a los procesos a esperar tiempos relativamente largos, en lugar de realizar trabajo útil, antes de concluir que un proceso ha fallado.

Una aproximación un poco diferente es la de utilizar detectores de errores poco fiables, y alcanzar consenso mientras se permite a los procesos sospechosos comportarse correctamente en lugar de excluirlos. Chandra y Toueg [9] analizaron las propiedades que un detector de fallas debe tener con el propósito de resolver el problema de alcanzar el consenso en un sistema asíncrono. Mostraron que el consenso se puede lograr en un sistema asíncrono, aun con estos detectores poco confiables, siempre y cuando no haya más de $N/2$ procesos incorrectos (debido a fallas por caídas) y la comunicación sea confiable.

El algoritmo de consenso de Chandra y Toueg permite a los procesos falsamente sospechosos continuar su operación normal y permite a los procesos que han sospechado de ellos recibir los mensajes de ellos y de los procesos que funcionan de manera normal. Esta estrategia tiene la ventaja de que los procesos correctos falsamente sospechosos no son excluidos.

La aproximación anterior de Chandra *et al.* puede tener una pequeña mejora si se adaptan los tiempos de espera a las peticiones, de acuerdo a los tiempos de respuesta observados.

Consenso basado en aleatoriedad

El resultado de Fischer *et al.* depende sobre que podemos considerar que pueda ser un adversario. El adversario manipula la red para retrasar los mensajes, así que éstos llegan en momentos no adecuados, o bien, pueden acelerar o retrasar a los procesos lo suficiente para que cuando éstos reciban los mensajes no se encuentren en un estado correcto.

Esta técnica consiste en introducir un elemento de cambio en el comportamiento del proceso, así que el adversario no puede ejercer su estrategia efectivamente. El cosenso puede no ser alcanzado en algunos casos, pero este método habilita a los procesos alcanzar el consenso en un tiempo esperado de tiempo. Un algoritmo probabilístico que resuelve el consenso aun si ocurren fallas bizantinas se presenta en [7].

Apéndice B

Algoritmos de planificación

B.1. Planificación de tareas aperiódicas

B.1.1. Servidores de prioridad fija

Los algoritmos de planificación anteriormente descritos sólo tratan con conjuntos de tareas homogéneas, la actividad computacional se centra en planificar ya sea tareas periódicas o aperiódicas. Sin embargo, muchas de las tareas realizadas por los sistemas de tiempo real tienen que lidiar con aplicaciones que requieren ambos tipos de tareas.

Un algoritmo de prioridad fija basa la organización de la ejecución de sus tareas en el algoritmo Rate Monotonic, asignando la misma prioridad a todos los jobs de una tarea. Los algoritmos que se presentan a continuación tienen el propósito de manejar conjuntos compuestos de un subconjunto de tareas periódicas con plazos estrictos y de un subconjunto de tareas aperiódicas con plazos flexibles. Además, todos los algoritmos aquí presentados cumplen con las siguientes restricciones:

- Las tareas periódicas serán planificadas bajo el algoritmo Rate Monotonic (RM), esto quiere decir que estarán bajo un modelo fijo de planificación.
- Todas las tareas periódicas comenzarán simultáneamente su ejecución en tiempo $t = 0$ y, además, los plazos serán iguales a sus períodos.
- Las peticiones de ejecución para tareas aperiódicas son desconocidas.
- Cuando no se especifique explícitamente el tiempo mínimo de ejecución entre dos instancias de una tarea aperiódica, se asumirá que será igual a su plazo.
- Todas las tareas pueden ser interrumpidas en cualquier momento (preemptive).
- Cada tarea periódica τ_i tiene un período T_i, un tiempo de cómputo C_i y un plazo relativo D_i al que consideraremos igual a su período.

Planificación en segundo plano

Este método por si sólo no es implementado, como su nombre lo indica trabaja de manera conjunta con otros algoritmos con el afán de auxiliarlos. Cuando el algoritmo que lo implementa

ha hecho todo lo que debía hacer con el presupuesto y hay tiempo ocioso en el procesador el procesamiento en segundo plano (*background*, en inglés) se activa. Este mecanismo se aplica cuando no hay instancias de tareas periódicas listas para ser ejecutadas. Esta forma de planificar presenta un inconveniente: para grandes cargas de trabajo (por tareas periódicas) las tareas aperiódicas deben esperar grandes cantidades de tiempo antes de ser ejecutadas por primera vez. Por dichas razones, se recomienda el uso de este método de planificación cuando las tareas aperiódicas no tienen restricciones rigurosas de tiempo o bien la carga de trabajo de tareas aperiódicas no es tan alta.

Para implementar este método se necesita de dos colas (ver figura B.1). La primera, de alta prioridad, estará dedicada a las tareas periódicas. La segunda, que posee la prioridad más baja, estará reservada para las peticiones de ejecución de las tareas aperiódicas, éstas sólo se ejecutarán cuando la cola de tareas periódicas este vacía y en caso de que una tarea aperiódica esté en ejecución en el momento en el que una tarea periódica pida ser ejecutada, se suspenderá la tarea aperiódica y se le cederá el procesador a la tarea periódica. Las dos colas son independientes y por lo tanto puede haber un algoritmo de planificación distinto actuando en cada una de ellas.

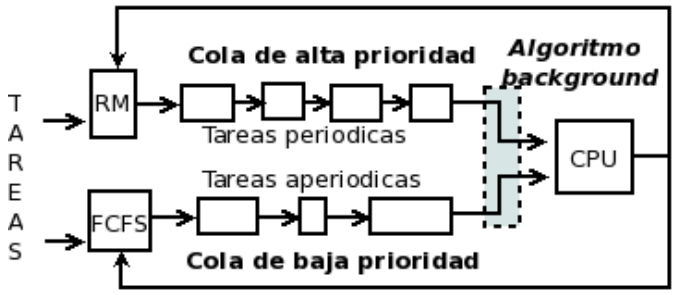

Figura B.1: Estructuras de datos necesarias para el procesamiento de tareas en segundo plano.

Los modos principales de acceso a los recursos del sistema [3] por las tareas que se encuentran en espera en la cola son:

FCFS (First Come First Served), también llamado *Fisrt In First Out*. Sirve a las tareas en el orden de llegada al sistema. No siempre es la más conveniente, ya que si hay una tarea de larga duración monopolizará el uso de los recursos que ocupe.

SJF (Shortest Job First), también conocida como *Sortest In First Out*. Sirve primero a aquellas tareas cuya demanda de servicio es menor.

LCFS (Last Come First Served) llamada *Last In First Out*. La última tarea en llegar es la primera en servirse.

RR (Round Robbin) en la que se reparte el tiempo de recurso equitativamente entre todas las tareas que esperan.

A continuación se listan las actividades principales que realiza el algoritmo:

1. Cada comienzo de un nuevo periodo de la i-ésima tarea periódica, esto es cada múltiplo $k \in N \cup \{0\}$ de su respectivo periodo T_i, envia el siguiente job a la cola de mayor prioridad.

2. Los jobs de tareas periódicas son recibidos por el bloque encargado de agregar los jobs a la lista de mayor prioridad, estos se ordenan en la cola de alta prioridad para su ejecución de acuerdo a algún criterio, digamos RM; si hay tareas aperiódicas deberán esperar a ser

ejecutadas en la cola de baja prioridad, bajo el criterio FCFS (First Come First Served) por ejemplo.

3. Ejecutar los jobs, mientras los haya en la cola de alta prioridad. En caso de no haber jobs en la cola de alta prioridad se tomarán las tareas en la cola de baja prioridad y se ejecutarán hasta que hayan jobs, de nueva cuenta, en la cola de alta prioridad.

Nótese que el punto tres implica la ejecución de dos procesos: uno que se encargue de la ejecución de las treas aperiódicas y otro que, concurrentemente, recibirá la notificación que detiene la ejecución de la tarea aperiódica para dar lugar a los jobs de tareas periódicas.

Nota : para su implementación debe considerarse que la cola de alta prioridad debe manejar la inserción de nuevos jobs, tal vez simultáneamente, por lo que debe considerar la inclusión de un mecanismo de exclusión mutua durante la inserción, lo mismo para la cola de baja prioridad.

Servidor de consulta

Si comparamos este algoritmo con el de *planificación en segundo plano* resulta que este mejora la respuesta a peticiones de ejecución de tareas aperiódicas. Este algoritmo realiza la planificación a través de un *servidor*; esto es, una tarea periódica cuyo propósito es el de procesar peticiones de tareas aperiódicas tan pronto como le sea posible. Como cualquier otra tarea periódica, un servidor τ_s consta de un período T^S y un tiempo de computación C^S, llamado capacidad del servidor e indica la cantidad de presupuesto disponible. Una vez que éste se encuentra activo trata de atender las peticiones dentro de los límites de su capacidad C^S. La decisión de en que orden el servidor atenderá las peticiones de las tareas aperiódicas dependerá del programador.

El algoritmo de servidor de consulta o *Polling Server* [50, 29] (PS) se activa cada T^S unidades de tiempo y ejecuta alguna petición pendiente hecha por alguna tarea aperiódica, pero lo hace en medida del presupuesto disponible. Si no hay peticiones pendientes de tareas aperiódicas, PS se suspende a si mismo hasta que comience el siguiente período, y el tiempo que originalmente había sido asignado para dar servicio a las tareas aperiódicas es cedido a la ejecución de tareas periódicas. Nótese que si la petición de una tarea aperiódica llega justo después de que el servidor PS se ha suspendido, éste debe esperar hasta que PS comience un nuevo periodo T^S, que es cuando la capacidad del servidor se repone (replenish time); esto es, recupera su presupuesto.

La planificación de un conjunto de tareas periódicas, cuya planificación está a cargo del algoritmo PS, bajo los criterios del algoritmo RM se garantiza si se cumple que

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C^S}{T^S} \leq (n+1)[2^{1/(n+1)} - 1] \tag{B.1}$$

Para aclarar veamos la figura B.2, que muestra la calendarización de las tareas periódicas $\tau_1 = (20, 5), \tau_2 = (10, 2), \tau_3 = (14, 1)$ y para el servidor PS tenemos $C^S = 1$ y $T^S = 16$ con tareas aperiódicas A_1 y A_2 cuyos tiempos de activación son de $r_{A1} = 6$ y $r_{A2} = 16$, con tiempos de ejecución son $C_{A1} = 2.5$ y $C_{A2} = 1.5$. Las prioridades en este ejemplo tienen las siguientes jerarquías $\tau_2 > \tau_3 > \tau_s > \tau_1$. Obsérvese que al tiempo $t = 0$ no hay peticiones de llamadas aperiódicas, así que el presupuesto se pierde. A pesar de que la primer tarea aperiódica se activa por primera vez en $t = 6$ no se puede ejecutar por falta de presupuesto y se comienza a ejecutar en $t = 16$, pero sólo es posible ejecutarla en la medida del presupuesto, terminando en $t = 17$ con un tiempo de ejecución sobrante de 1.5 unidades de tiempo, después retoma su ejecución

en $t = 32$ y termina una unidad de tiempo después, la tarea aperiódica A_1 termina su ejecución de manera definitiva hasta $t = 48.5$ e inmediatamente después comienza su ejecución la tarea aperiódica A_2. La tarea A_2 fue activada en $t = 16$, pero su ejecución no es posible hasta que no termine A_1.

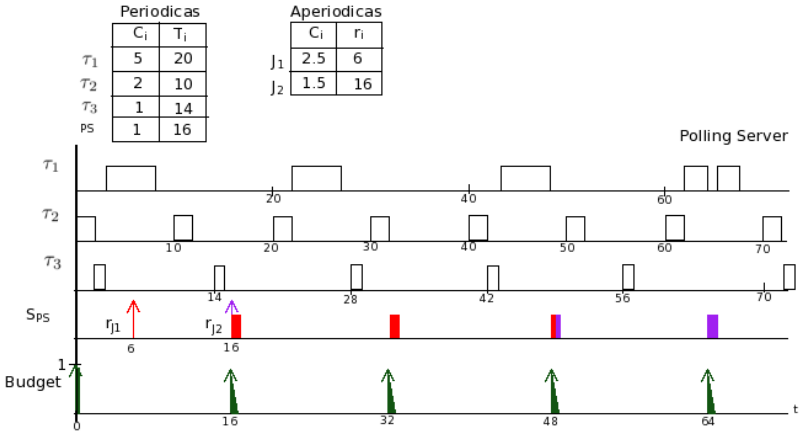

Figura B.2: Ejemplo de la planificación que produce el servidor $S_{PS} = (16, 1)$.

Servidor aplazable

Este algoritmo, llamado en inglés *deferrable server*, propuesto por Lehoczky, Sha y Stronsnider [29, 55] mejora el tiempo de respuesta promedio del algoritmo PS para atender a peticiones hechas por tareas aperiódicas con respecto al PS. Al igual que el algoritmo PS, el algoritmo de servidor aplazable (DS) atenderá las peticiones de tareas aperiódicas como si se tratasen de tareas periódicas. Pero, a diferencia del algoritmo PS, éste conserva su capacidad, aún cuando no se hayan atendido peticiones de tareas aperiódicas. La capacidad (presupuesto) del servidor se mantiene hasta el final del período, por lo que puede atender peticiones en cualquier momento, siempre y cuando su capacidad se lo permita. Al inicio de cada período el algoritmo recupera toda su capacidad.

Dado un conjunto de n tareas periódicas y las restricciones impuestas por el algoritmo servidor aplazable con factores de utilidad U_p y U_s, respectivamente, la planificabilidad del conjunto se garantiza bajo RM si

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right) \tag{B.2}$$

Un mejor desempeño puede lograrse si al servidor se le asigna una mayor prioridad. El efecto de incrementar la prioridad (asignándole un periodo (T^S) menor a DS) trae como consecuencia reducir los tiempos de respuesta del servidor.

Un ejemplo de la planificación que produce este algoritmo se muestra en la figura B.3. Son dos las tareas periódicas, τ_1 y τ_2, cuyos periodos son $T_1 = 8$ y $T_2 = 10$ con tiempos de ejecución $C_1 = 1$ y $C_2 = 2$, y el algoritmo DS con los valores $T^S = 6$ y $C^S = 2$, presentes en este sistema.

Al tiempo $t = 0$ no hay peticiones pendientes de tareas aperiódicas, por lo que la tarea τ_1 es asignada al procesador y en cuanto ésta termina ($t = 1$) τ_2 se comienza a ejecutar, pero no concluye ya que en $t = 2$ la primer tarea aperiódica J_1 se activa y tiene la más alta de las prioridades. La tarea J_1 se ejecuta en su totalidad terminando con el presupuesto, por lo que

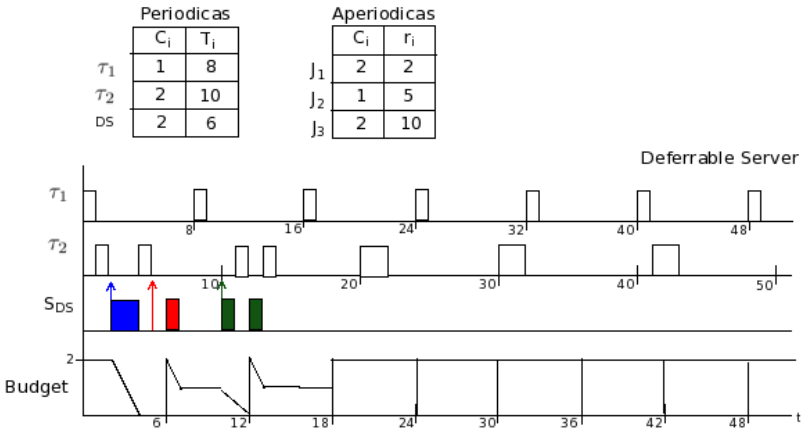

Figura B.3: Ejemplo de planificación producida por DS.

J_2 tiene que esperar a que DS recupere el presupuesto al inicio del siguiente periodo; mientras tanto se termina de ejecutar el primer job de τ_2. Para $t = 6$ ya hay pendiente una petición de la tareas aperiódica J_2, hecha en $t = 5$, por lo que es asignada al procesador inmediatamente, consumiendo una unidad del presupuesto. En $t = 10$ se activa la tercer tarea aperiódica J_3 que sólo se puede ejecutar por una unidad de tiempo, ya que el presupuesto no es suficiente, por lo que tiene que esperar al tiempo $t = 12$ y termina en $t = 13$.

Intercambio de prioridad

Esta técnica nos es útil para dar servicio a un conjunto de peticiones hechas por tareas aperiódicas junto a un conjunto de tareas periódicas [29]. A comparación del DS, el algoritmo servidor de intercambio de prioridad (PE) es ligeramente mejor en términos de la capacidad de responder a las peticiones de las tareas aperiódicas, ejecuta la petición en cuanto le es posible y a la medida de su presupuesto. Hace un manejo inteligente del presupuesto sobrante de peticiones hechas en periodos anteriores, si le es posible.

Al igual que otros algoritmos, el PE se auxilia de un servidor periódico (que usualmente posee una prioridad alta) para ejecutar las peticiones de tareas aperiódicas. El algoritmo PE conserva su capacidad de alta prioridad intercambiándola por el tiempo de ejecución de una tarea periódica.

Al comienzo de cada periodo del servidor, el presupuesto es puesto a su capacidad total. Si existen peticiones de tareas aperiódicas y el servidor es la tarea lista con las más alta prioridad, entonces las peticiones son atendidas con el presupuesto disponible; de lo contrario C^S hace un intercambio durante el tiempo de ejecución de una tarea periódica activa con la más alta prioridad en ese momento.

Cuando ocurre uno de esos cambios de prioridad entre el PE y una tarea periódica, la tarea periódica se ejecuta en el nivel de prioridad del servidor mientras el servidor acumula la capacidad en el nivel de prioridad de una tarea periódica. Entonces, la tarea periódica continúa con su ejecución, y la capacidad del servidor se mantiene en baja prioridad. Si no llega alguna petición hecha por tareas aperiódicas para usar la capacidad el cambio de prioridad continua con otras tareas de baja prioridad hasta que la capacidad sea usada para algún servicio a tareas aperiódicas o bien la capacidad se pierda y el PE se degrade para ejecutarse en segundo plano. Ya que el propósito de este algoritmo es el de proporcionar servicio principalmente a peticiones de tareas aperiódicas, todos los empates se rompen a favor de tareas aperiódicas.

Dado un conjunto de n tareas periódicas y el servidor de intercambio de prioridad con factor de utilidad U_p y U_s, respectivamente, la planificación del conjunto de tareas periódicas se garantiza bajo RM si

$$U_p \leq \ln(\frac{2}{U_s + 1}) \tag{B.3}$$

El ejemplo de la planificación que produce este algoritmo se muestra en la figura B.4. Son dos las tareas periódicas, τ_1 y τ_2, cuyos periodos son $T_1 = 8$ y $T_2 = 10$ con tiempos de ejecución $C_1 = 1$ y $C_2 = 2$, y el algoritmo DS con los valores $T^S = 6$ y $C^S = 2$, presentes en este sistema.

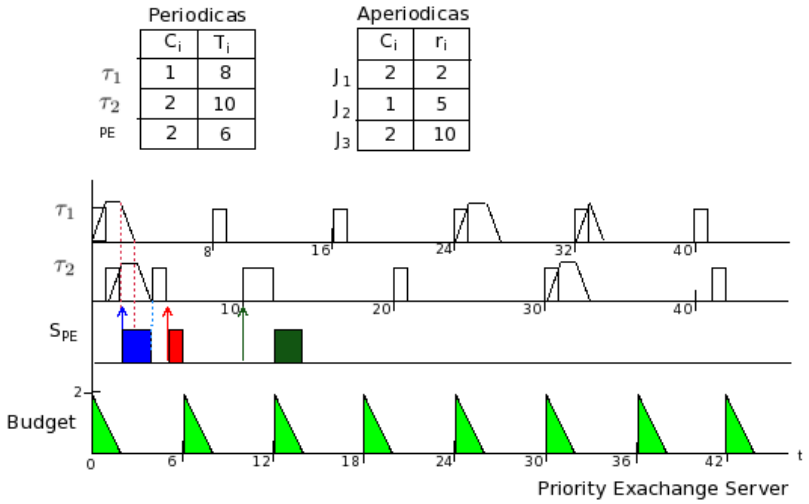

Figura B.4: Ejemplo de pla planificación producida por PE.

Al tiempo $t = 0$ PE asigna un presupuesto de 2 pero como no hay peticiones pendientes de tareas aperiódicas la tarea τ_1 es asignada al procesador y, como es la única tarea activa, recibe el presupuesto. Como resultado el servidor acumula una unidad de tiempo al nivel de prioridad de la tarea τ_1 y en cuanto ésta termina ($t = 1$) τ_2 se comienza a ejecutar y recibe el resto del presupuesto (el servidor acumula una unidad de tiempo al nivel de prioridad de la tarea τ_2). En $t = 2$ se activa la primer tarea aperiódica por lo que comienza a consumir el presupuesto almacenado por τ_1 hasta $t = 3$, y consume el presupuesto de τ_2. En $t = 5$ se activa la tarea aperiódica J_2 a pesar de que no hay presupuesto, pero esto es debido a la ausencia de tareas periódicas (PE degradó su ejecución a segundo plano). Por último, la tarea J_3 genera una petición de ejecución pero no hay presupuesto para asignarla al procesador, así que debe esperar a que el job $\tau_{2,2}$ termine su ejecución en $t = 12$, y es justamente que PE asigna presupuesto por lo que J_3 no se ejecuta en segundo plano, sino lo hace con alta prioridad (de hecho la más alta) consumiendo el presupuesto; es decir, si hubiese visto alguna petición de ejecución (job) de alguna tarea periódica, el planificador hubiese elegido de cualquier forma a la tarea aperiódica.

Servidor esporádico

Algoritmo propuesto por Sprunt, Sha y Lehoczky (en inglés es conocido como *sporadic server*) [50], que permite mejorar el tiempo de respuesta promedio para tareas aperiódicas sin degradar el límite de uso del conjunto de tareas periódicas.

Éste algoritmo (SS) crea una tarea de alta prioridad, la tarea servidor, para atender peticiones de tareas aperiódicas y, al igual que DS, preserva la capacidad del servidor a su más alto nivel hasta la llegada de una petición hecha por una tarea aperiódica. Excepto que SS maneja el llenado de su capacidad en trozos de presupuesto.

Para facilitar la descripción del método de llenado de presupuesto usado por SS, definamos la siguiente notación:

P_{exe} Denota el nivel de prioridad de la tarea que actualmente está en ejecución.

P_s Denota el nivel de prioridad asociada a SS.

Activo Se dice que SS está activo cuando $P_{exe} \geq P_s$.

Ocioso Se dice que SS está ocioso cuando $P_{exe} < P_s$.

RT Denota el tiempo de llenado (replenishment time) al cual la capacidad será aumentada.

RA Denota la cantidad de llenado (replenishment amount) que será sumada a la capacidad existente en el tiempo RT.

La capacidad C^S consumida por la petición de una tarea aperiódica será llenada de acuerdo con las siguientes reglas:

1. El tiempo de llenado RT se produce tan pronto como SS llegue a estar activo y $C^S > 0$. Sea t_A dicho tiempo. El valor de RT es igual a t_A más el periodo del servidor, T^S ($RT = t_A + T^S$).
2. La cantidad de llenado RA que será hecha al tiempo RT se computa cuando SS se encuentra ocioso o C^S ha sido agotado. Sea t_I la variable que represente lo antes descrito. El valor de RA es puesto igual a la capacidad consumida dentro del intervalo $[t_A, t_I]$

Debido a que SS se comporta como una tarea periódica, el test de planificación del conjunto de tareas periódicas se deriva del algoritmo PE. Por lo tanto, un conjunto Γ de n tareas periódicas con factor de utilidad U_p calendarizado con Sporadic Server con factor de U_s son planificables bajo RM si el conjunto de tareas periódicas es pequeño y cumple

$$U_p \leq n \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right] \tag{B.4}$$

Para conjuntos de tareas grandes la planificabilidad se garantiza si la condición $U_p < log\left(\frac{1}{U_s+1}\right)$ es verdadera.

Veamos el conjunto de tareas bajo SS de la figura B.5. Consiste en dos tareas periódicas τ_1 y τ_2, donde sus periodos son $T_1 = 6$ y $T_2 = 10$, con tiempos de consumo $C_1 = 1$ y $C_2 = 2$ más la tarea periódica del algoritmo SS con periodo y tiempo de consumo $T^S = 8$ y $C^S = 2$, respectivamente.

Al no haber peticiones de tareas aperiódicas se asigna el procesador al primer job de τ_1, en cuanto éste último termina se comienza a ejecutar el primer job de τ_2 pero no concluye puesto que ocurre la petición por recursos por parte de la tarea aperiódica J_1. La tarea J_1 está activa a partir de $t = 2$ por lo que se calcula el siguiente $RT = 2 + T^S = 10$, y se consumen dos unidades de presupuesto por lo que la siguiente cantidad de llenado $RA = 2$. La siguiente petición por parte de la tarea aperiódica J_2 se hace en $t = 5$ pero es imposible atenderla ya que no hay el presupuesto, hasta $t = 10$ que es cuando ocurre el RT. La tarea J_2 activa a SS y se calcula el próximo $RT = 10 + T^S = 18$ consumiendo una unidad de presupuesto, en cuanto la tarea J_2 se termina se comienza a ejecutar J_3 pero no concluye debido a la falta de presupuesto, la tarea J_3 retoma su ejecución en $t = 19$ y se termina en $t = 20$, nótese que SS estaba activo desde $t = 18$, por lo que el siguiente tiempo $RT = 18 + T^S = 26$ y se consumió una unidad del presupuesto, entonces $RA = 1$.

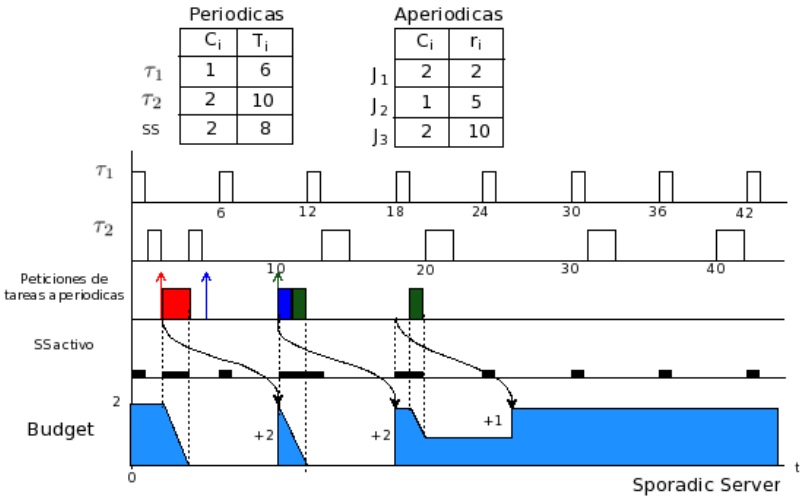

Figura B.5: Ejemplo de pla planificación producida por SS.

B.1.2. Servidores de prioridad dinámica

En estos algoritmos la asignación de prioridades a las instancias (o jobs) de cada tarea se realiza de manera dinámica, a través de algún algoritmo basado en plazos como *Earliest Deadline First*. Esto es, que al algoritmo permite cambiar la prioridad de cada job en cada tarea durante el tiempo de ejecución, eligiendo como al job de mayor prioridad a aquel que tenga plazo relativo más corto. Al igual que los algoritmos de prioridad fija estos son útiles en la planificación de tareas que son aperiódicas y periódicas. Pero, a diferencia de los algoritmos basados en RM (prioridad fija), los algoritmos de planificación dinámicos permiten mejorar el uso del procesador, en promedio mejora la respuesta a peticiones de tareas aperiódicas.

Para la siguiente lista de algoritmos se toman en cuenta las restricciones:

- Toda tarea periódica $\tau_i : i = 1, \ldots, n$ tiene un plazo estricto.
- Ninguna tarea aperiódica $J_i : i = 1, \ldots, m$ tiene plazo.
- Cada tarea periódica τ_i tiene un periodo T_i, un tiempo de cómputo C_i y un plazo relativo D_i, que será *igual* a su período.
- Todas las tareas periódicas se activan simultáneamente al tiempo $t = 0$.
- Las peticiones de las tareas aperiódicas tienen tiempos conocidos de cómputo pero los tiempos de activación (arrival times) se desconocen.

Intercambio dinámico de la prioridad

En inglés es conocido como *Dynamic Priority Exchange*. Esta técnica, propuesta por Spuri y Buttazo [51, 52] es una extensión del algoritmo servidor de intercambio de prioridad [29], sólo que éste está adaptado para trabajar con Earliest Deadline First. La idea principal de este algoritmo es la de *permitirle al servidor intercambiar su tiempo de ejecución con el tiempo de ejecución de la tarea periódica de menor prioridad* en un momento dado sólo si no hay petición alguna de tareas aperiódicas. De esta manera el tiempo de ejecución del servidor se preserva, es decir, los recursos asignados para el servidor no se consumen, a menos que haya tiempos

ociosos en el procesador. Con esta característica el servidor es capaz de reclamar el tiempo de procesador cuando una tarea aperiódica se active.

Describamos con detalle el algoritmo, pero antes se debe hacer notar que el comportamiento del servidor se ve regido por un periodo T^S y una capacidad C^S:

1. Al comienzo de cada período, a la capacidad del servidor (C^S) se le es asignada un presupuesto (budget) para atender a las peticiones de las tareas aperiódicas, C^{Sd}, donde d es el plazo del período actual. Para abreviar, llamaremos a la capacidad para atender las peticiones de las tareas aperiódicas como la *capacidad aperiódica*.
2. Cada plazo d asociado a las instancias, completas o no, de la tarea periódica i tiene una capacidad aperiódica, $C_{S_i}^d$, que inicialmente es puesta a cero.
3. Todas aquellas capacidades aperiódicas mayores que cero reciben prioridades de acuerdo al algoritmo EDF.
4. Siempre que la entidad de mayor prioridad en el sistema (nótese que bien puede tratarse de una tarea aperiódica o el servidor, todo depende de la prioridad de los jobs en ese momento) tenga una capacidad aperiódica de C unidades de tiempo lo siguiente ocurre:
   - Si existe la petición de una tarea aperiódica en el sistema, ésta es servida hasta que se completa o hasta que la capacidad C es agotada (cada petición consume una capacidad igual a al tiempo que se ejecuta);
   - Si no hay peticiones de tareas aperiódicas pendientes, la tarea periódica con el plazo más corto se ejecuta; una capacidad igual a la duración de la ejecución de la tarea periódica es agregada a la capacidad aperiódica $C_{S_i}^d$ de la tarea periódica y dicha capacidad se extrae de C.
   - Si no hay peticiones de tareas aperiódicas o jobs de tareas periódicas pendientes, se dice que el procesador está en estado de ocioso y la capacidad, C, se consume hasta agotarse. Cuando la capacidad C ha sido consumida por completo debemos esperar a que ocurra el siguiente tiempo de llenado: cuando comienza un nuevo período T^S en el servidor.

Para garantizar que un conjunto de tareas puede ser planificado bajo este algoritmo se debe cumplir la condición propuesta por Liu y Layland [35]: $U_p + U_s \leq 1$, donde U_p el el factor de utilidad del conjunto de tareas periódicas y U_s es el factor de utilidad del servidor DPE.

Servidor dinámico esporádico

La principal diferencia entre la versión con prioridades fijas y ésta, es la manera en la que son asignadas las prioridades. Mientras que en SS la prioridad se elige de acuerdo al algoritmo RM, dynamic sporadic server (DSS) tiene una prioridad que le fue asignada en base a su plazo. Las reglas, muy parecidas a las de SS, se listan a continuación:

1. Cuando el servidor es creado, su capacidad C^S se inicializa a su máxima capacidad (le es asignado el presupuesto máximo posible).
2. El siguiente tiempo de llenado RT y el plazo actual del servidor d_s son asignados tan pronto como DSS llegue a estar activo, esto es que $C^S > 0$ y haya una petición periódica pendiente. Sea t_A dicho tiempo, entonces, el valor de RT es igual a t_A más el periodo del servidor, T^S, o sea, $RT = d_s = t_A + T^S$.

3. La cantidad de llenado RA que será hecha al tiempo RT se computa cuando DSS se encuentra ocioso o C^S ha sido agotado. Sea t_I la variable que representa dicho instante en el tiempo. El valor de RA es puesto igual a la capacidad consumida dentro del intervalo $[t_A, t_I]$.

La planificabilidad para un conjunto de n tareas periódicas con este algoritmo se garantiza sólo si $U_p + U_s \leq 1$, donde U_s y U_p son el factor de utilidad de SS y la utilidad de las tareas aperiódicas, respectivamente.

Servidor de ancho total de banda

Para este algoritmo (del inglés *Total Bandwith Server*), al que se referirá desde ahora como TBS, propuesto por Spuri y Buttazzo [51, 52], la asignación del procesador debe hacerse de tal manera que el uso total del procesador para una carga de peticiones de tareas aperiódicas nunca exceda el valor máximo especificado del factor de utilidad U_s. La idea es asignar inmediatamente el ancho de banda total del servidor a cualquier instancia de tarea aperiódica que solicite ser ejecutada.

Cuando la petición de la tarea aperiódica tiene lugar en un tiempo $t = r_k$, ésta recibe un plazo, que se calcula mediante:

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{B.5}$$

donde C_k es el tiempo de ejecución de la petición y U_s es el factor de utilidad de la tarea servidor. Por definición $d_0 = 0$. Nótese que en la regla de asignación del plazo se considera el ancho de banda asignado a tareas aperiódicas previas, d_{k-1}.

Para garantizar que cualquier conjunto de n tareas periódicas será planificado bajo TBS y utilizando el algoritmo EDF debe cumplirse la propiedad:

$$U_p + U_s \leq 1 \tag{B.6}$$

Esta vez, se trata de una condición suficiente y necesaria.

Servidor de utilización constante

Para este algoritmo (en inglés, *Constant Utilization Server*) se define la fracción de utilización del procesador, U, para la tarea servidor. Se asume que el tiempo de ejecución para cada job es conocido después de que es liberado, sea C_i el tiempo de ejecución para cualquier instancia de la tarea i ($J_{i,k}$) en la cola de tareas listas para su ejecución, sean r_i y d las variables que representan el tiempo de liberación y el plazo de la instancia de $J_{i,k}$, respectivamente; por último, consideremos a la variable b, cuyo valor se encuentra en el rango $0 < b \leq C_i$ e indica el valor del consumo de cualquier instancia de cualquier tarea lista para ser ejecutada J_i.

Las reglas para esta estrategia de planificación son las siguientes:

1. Inicialmente, el presupuesto del servidor es cero, y su plazo es, también, colocado en cero.
2. Cuando alguna instancia de la *i-ésima* tarea, J_i, esporádica (o aperiódica) con consumo C_i este lista para ser ejecutada (es decir, ha sido liberada y está en espera al inicio de la cola de tareas listas) en tiempo r_i,

   a) Si $d \leq r_i$, asignar presupuesto de b al servidor, $C^S = b$ y $r_i + \frac{b}{U}$ para el plazo, d.
   b) Si $d > r_i$, hacer nada.

3. En el plazo d del servidor:

   a) Si un job J_i con tiempo de consumo C_i está esperando al inicio de la cola de tareas listas, se asigna un presupuesto de b y movemos el plazo a $d + \frac{b}{U}$;
   b) De otro modo se hace nada.

La tarea servidor se comporta como una tarea cuyo valor de utilización, U, es constante sólo si ésta nunca está vacía, es decir, si siempre hay tareas esporádicas o aperiódicas en espera para ejecución, por ello es el nombre de su denominación.

Este algoritmo en esencia es el mismo que el *Total Banwidth Server* (TBS). A diferencia de que en TBS el presupuesto es recuperado inmediatamente si la cola de tareas esporádicas o aperiódicas están llenas de tareas listas para ser ejecutadas en espera, mientras que en éste algoritmo la próxima reasignación de presupuesto se hace cuando el plazo actual se termina.

B.1.3. Planificación de tareas que comparten recursos

Hasta este momento se han explicado algunos de los algoritmos de planificación más conocidos cuyas tareas son independientes; no obstante, algunas aplicaciones comparten recursos y, por lo tanto, es necesario aplicar alguno de los mecanismos que garantizan la exclusión mutua con algún costo tácito.

Los sistemas concurrentes típicamente proveen de una herramienta de sincronización, llamada *semáforo* [13], que puede ser usado para delimitar secciones críticas. Cuando se utiliza esta herramienta, cada recurso mutuamente excluyente R debe estar protegido por diferentes semáforos S_i y cada sección crítica posee métodos atómicos que permiten cambiar el estado de los semáforos.

El problema consiste en proveer de una planificación, tal que dado un conjunto de tareas (de al menos dos de ellas) que comparten al menos un recurso R sea óptima en el sentido de que no se pierda ningún plazo.

Fenómeno de inversión de prioridad

Una situación muy conocida en sistemas computacionales es la que surge cuando existen al menos dos procesos con la necesidad de compartir algún recurso. La situación más conocida es el acceso a variables de memoria compartida. Para evitar la inconsistencia de los datos, es usual utilizar algún mecanismo de sincronización para garantizar la exclusión mutua, es decir, un mecanismo capaz de garantizar que mientras una tarea está utilizando un recurso otra tarea no pueda tener acceso a dicho recurso. Algunos de los mecanismos más usuales de sincronización son los semáforos, monitores y rendezvous; sin embargo, estos mecanismos pueden dar lugar a inversiones de prioridad, que dificulten o impidan el cumplimiento de las metas temporales de las tareas.

Para dar mayor claridad se da el siguiente ejemplo. Consideremos dos tareas J_1 y J_2 que comparten el recurso R_i. Para garantizar la exclusión mutua el recurso debe estar protegido por algún mecanismo de sincronización.

Si la apropiación de tareas está permitida y la tarea J_1 tiene mayor prioridad que J_2, entonces uno de los escenarios posibles (vea la figura B.6) puede ser en el que J_1 se ve interrumpida por la tarea J_2, la cual comienza su ejecución antes que J_1, esto es $r_2 < r_1$, y antes de que la tarea J_1 sea activada la tarea J_2 comenzó a ejecutar la región crítica (la acotada por el mecanismo de exclusión mutua), durante la ejecución de dicha región crítica la tarea J_1 se activa para su ejecución y por ser de mayor prioridad interrumpe a la tarea J_2 en un tiempo t_0; la tarea J_1 se ejecuta hasta que llega a la región que comparte con la otra tarea, J_1 no puede continuar su ejecución así que debe ceder su tiempo de ejecución en el procesador a la otra tarea en tiempo t_1; por ende, J_2 reanuda su ejecución. J_1 no tiene mas opción que esperar a que J_2 libere la región crítica.

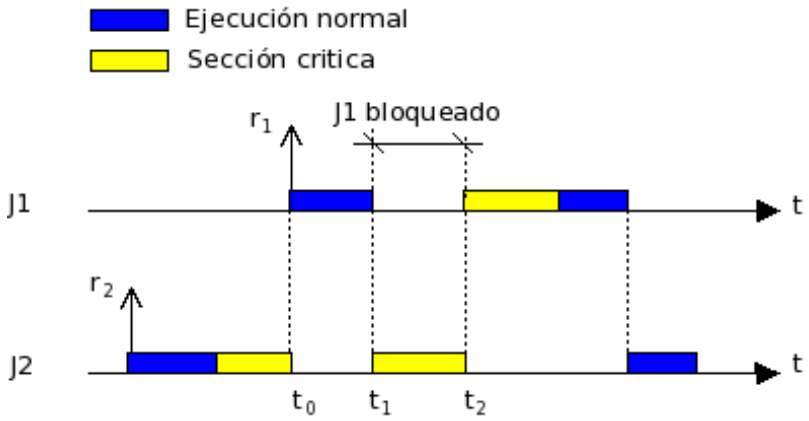

Figura B.6: Ejemplo de inversión de prioridad, debido a un recurso compartido.

Protocolo de herencia de prioridad simple

Conocido en ingles como *priority inheritance protocol*, ofrece una solución al problema que surge del intercambio de prioridades debido a recursos compartidos. A grandes rasgos, el algoritmo consiste en hacer que la tarea de menor prioridad herede la prioridad de la de mayor prioridad en cuanto la tarea de menor prioridad comienza a bloquear a la de mayor prioridad, así eliminamos el bloqueo indirecto. Algunas características del algoritmo se listan a continuación:

- Se aplica a sistemas de prioridad fija, aunque es posible extender el protocolo EDF Spuri.
- Se elimina el bloqueo indirecto y se acota el tiempo máximo de bloqueo.
- Pueden existir bloqueos inducidos en las tareas de prioridad intermedia.
- Puede existir bloqueo encadenado: una tarea es bloqueada por tareas de menor prioridad varias veces en su activación.

El algoritmo está sujeto a las siguientes suposiciones:

- Las tareas no se suspenden a si mismas (operaciones de entrada/salida).
- Las secciones críticas están propia mente anidadas.
- Sólo un job puede estar dentro de una sección crítica a la vez, la cual se encuentra resguardada por un semáforo S_k.

Una descripción aproximada del algoritmo podría ser la siguiente:

1. Comenzamos asignando prioridades a los jobs de acuerdo a algún criterio, se referirá a esta prioridad por *prioridad original* o sólo como prioridad. Cuando existan jobs con la misma prioridad podemos usar el criterio de que al primer job en llegar se le asignará primero el procesador.
2. Cuando un job, digamos J_i, trate de accesar a una sección crítica que actualmente está siendo ejecutada por otro job de menor prioridad, entonces J_i se bloqueará. De lo contrario entrará a la sección crítica.
3. Cuando un job de mayor prioridad J_i es bloqueado por un semáforo, éste transmitirá su prioridad al job J_k y este último modificará su prioridad activa. Entonces, J_k ejecuta el resto de la sección crítica.
4. Cuando J_k termina con la sección crítica libera al semáforo S_k, y con éste al job con la prioridad más alta. J_k actualiza su prioridad activa con su prioridad original si no está bloqueando a otros jobs; de lo contrario, éste asignará su prioridad activa a la prioridad del job bloqueado con mayor prioridad.

Nota: la herencia de prioridad es transitiva, es decir, si un job J_3 bloquea a un job J_2, y J_2 bloquea a un job J_1, entonces J_3 hereda la prioridad de J_1 a través de J_2.

Protocolo de techo de prioridad

Un defecto del protocolo básico de herencia de prioridad es la posibilidad de ocurrencia de un deadlock y bloqueo encadenado cuando se accede a secciones críticas anidadas. Se impide el comienzo de las secciones críticas que puedan bloquear la ejecución de tareas de mayor prioridad.

La idea básica de este algoritmo consiste en que una tarea de menor prioridad herede la prioridad de la mayor prioridad cuando comienza a bloquear a la segunda. Una tarea que quiere utilizar un recurso se bloquea si su prioridad no es mayor que el techo de prioridad de todos los recursos que estén siendo utilizados por otras tareas, de este modo, ninguna tarea puede utilizar un recurso si su prioridad no es mayor que el techo de prioridad de todos los recursos utilizados por otras tareas en ese momento.

Propiedades:

- Una tarea se puede bloquear, como máximo, una vez cada activación.
- No pueden existir bloqueos encadenados.
- No puede haber interbloqueos.
- Con este protocolo, el tiempo de bloqueo máximo de una tarea es igual a la duración de la sección crítica más larga invocada por tareas de prioridad inferior sobre recursos cuyo techo de prioridad es igual o mayor que el de la tarea.

Listemos el algoritmo:

1. Se les es asignada una prioridad a cada semáforo $C(S_k)$ igual a la prioridad del job de más alta prioridad que puede bloquearlo.

   a) Sea J_i el job con la más alta prioridad entre todos los jobs listos para ser ejecutados; entonces, J_i es asignado al procesador.
   b) Sea S^* el semáforo con la más alta prioridad de techo entre todos los semáforos actualmente bloqueados por jobs distintos a J_i y sea $C(S^*)$ este techo.

2. Para entrar a una sección crítica resguardada por por un semáforo S_k, J_i debe tener una prioridad mayor a $C(S^*)$. Si la prioridad del job J_i, $P_i \leq C(S^*)$, el bloqueo de S_k no es permitido y entonces decimos que J_i se encuentra bloqueado por el semáforo S^*, debido al job que mantiene bloqueado a S^*.

3. Cuando un job J_i se encuentra bloqueado por un semáforo, éste transmite su prioridad al job, digamos J_k, que es el que mantiene bloqueado a dicho semáforo. Entonces J_k ejecuta el resto de la sección crítica con la prioridad de J_i.

4. Cuando J_k abandona la sección crítica, éste libera el semáforo y al job de mayor prioridad. La prioridad activa de J_k se actualiza de la siguiente manera: si no hay otros jobs bloqueados por J_k, p_k es asignada a la prioridad original P_k; de lo contrario, ésta es puesta en la prioridad más alta de los jobs bloqueados por J_k.

Bibliografía

[1] Aguilera, M. K., Chen, W., and Toueg, S. Failure detection and consensus in the crash-recovery model. *In Proceedings 11th International Symposium on Distributed Computing (DISC'98, formerly WDAG)* (September 1998), 231–245.

[2] Anderson, J., and Mills, A. A stochastic framework for multiprocessor soft real-time scheduling. In *Scheduling* (Dagstuhl, Germany, 2010), S. Albers, S. K. Baruah, R. H. Möhring, and K. Pruhs, Eds., no. 10071 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[3] Arias, J. J., Suárez-González, A., and Díaz-Redondo, R. P. *Teoría de colas y simulación de eventos discretos.* Pearson Education, S. A., Madrid, 2003.

[4] Baruah, S. Techniques for multiprocessor global schedulability analysis. *28th IEEE International Real-Time Systems Symposium* (2007), 119–128.

[5] Brassard, G., and Bratley, P. *Fundamentos de Algoritmia.* PRENTICE HALL, Madrid, 1997.

[6] Buttazzo, G. C. *Hard Real Time Computing Systems.* Springer, Italy: University of Pavia, 2004.

[7] Canetti, R., and Rabin, T. Fast asynchronous byzantine agreement with optimal resilience. *STOC '93 Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (1993), 42–51.

[8] Cano, J.-C., Calafate, C. T., Malumbres, M. P., and Manzoni, P. Redes inalámbricas ad hoc como tecnología de soporte para la computación ubicua. *Departamento de Informática de Sistemas y Computadores, Univesidad Politécnica de Valencia.*

[9] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *Journal of ACM 43*, 2 (1996), 225–267.

[10] Cormen, T., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms.* Mc Graw Hill, 2001.

[11] Deng, Z., and Liu, J. W. S. Scheduling real-time aplications in an open environment. *Proceedings of the 18th IEEE Real-Time Systems Symposium* (December 1997), 308–319.

[12] Deng, Z., Liu, J. W. S., and Sun, J. A scheme for scheduling hard real-time aplications in open system environment. *Proceedings of 9th Euromicro Workshop on Real-Time Systems* (June 1997), 191–199.

[13] Dijkstra, E. W. Cooperating secuential processes. *Programming Languages Academic Press, New York* (1968).

[14] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *Journal of ACM 35*, 2 (1988), 288–323.

[15] Fetzer, C., and Cristian, F. On the possibility of consensus in asynchronous systems. *In Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems, New Port Beach, USA* (December 1995), 86–91.

[16] Fischer, M. J., Lynch, N., and Paterson, M. S. Impossibility of distributed consensus with one fauty process. *Journal of Association for Computing Machinery 32* (1985), 374–382.

[17] Gantman, A., ning Guo, P., Lewis, J., and Rashid, F. Scheduling real-time tasks in distributed systems: A survey.

[18] Glenn, S. M., Dickey, T. D., Parke, B., and Boicourt, W. Long-term real-time coastal ocean observation networks. *Oceanography 13*, 1 (2000), 24–34.

[19] Graham, R., Lawler, E., Lenstra, J. K., and Kan, A. H. G. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics 5* (1979), 287–326.

[20] Hadzilacos, V., and Toueg, S. Reliable broadcast and related problems. *In Distributed Systems, ACM Press* (1993), 97–145.

[21] Hayashi, N., and Ushio, T. Application of a consensus problem to fair multi-resource allocation in real-time systems. *Proceedings of the 47th IEEE Conference on Decision and Control, Cancun, Mexico* (December 2008), 2450–2455.

[22] Kahn, G. The semantics of a simple language for parallel programming. *Information Processing '74: Proceedings of the IFIP Congress, J. L. Rosenfeld, Ed. New York, NY: North-Holland* (1974), 471–475.

[23] Kise, H., Ibaraki, T., and Mine, H. A solvable case of the one machine scheduling problem with ready and due times. *Operation Research 26* (1978), 121–126.

[24] Krishna, C. M., and Shin, K. G. *Real-Time Systems*. McGraw-Hill, 1997.

[25] Lamport, L. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM 21*, 7 (1978), 558–565.

[26] Lamport, L. Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems 8* (november 1990), 305–310.

[27] Lamport, L., Shostak, R., and Pease, M. Byzantine generals problem. *ACM Transactions Programming Languages and Systems 4*, 3 (1982), 382–401.

[28] Lawler, E. L. Optimal sequencing of a single machine subject to precedence constraints. *Managements Science 19* (1973).

[29] Lehoczky, J. P., Sha, L., and Strosnider, J. K. Enhanced aperiodic responsiveness in hard real-time environments. *In Proceedings os the IEEE Real-Time Systems Symposium* (December 1987).

[30] Leinberger, W., Leinberger, W., Karypis, G., Karypis, G., Kumar, V., Kumar, V., Biswas, R., and Biswas, R. Load balancing across near-homogeneous multiresource servers. 60–71.

[31] Lenstra, J., and Kan, A. R. Complexity of scheduling under precedence constraints. *Operations Research 26*, 1 (1978), 22–35.

[32] Lenstra, J. K., Kan, A. H. G. R., and Brucker, P. Complexity of machine scheduling problems. *Annals of Discrete Mathematics 7* (1977), 343–362.

[33] Leung, J., and Whitehead, J. W. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation 2* (1982), 4.

[34] Liu, C., and Anderson, J. H. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. *Department of Computer Science, University of North Carolina at Chapel Hill* (2010).

[35] Liu, C. L., and Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery 20* (1973), 41–46.

[36] Liu, J. W. S. *Real-Time Systems.* Prentice Hall, New Jersey, 2000.

[37] Locke, C. D. Best-effort decision making for real time scheduling. *PhD Thesis, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA* (1986).

[38] Lynch, N. A. *Distributed Algorithms.* Morgan Kaufmann Plublishers, Inc., 1996.

[39] Mall, R. *Real-Time Systems: Theory and Practice.* Pearson education, Second Impression, 2008.

[40] Mostefaui, A., Rajsbaum, S., and Raynal, M. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of Association for Computing Machinery 50* (2003), 922–954.

[41] Ortega-Arjona, J. L. The manager-workers pattern. an activity parallelism architectural pattern for parallel programming. *9th European Conference on Pattern Languages of Programming and Computing 2004 (EuroPLoP2004)* (july 2004), 7–11.

[42] Palomera-Pérez, M. A., and Benitez-Pérez, H. Scheduling coordinated task. *4th IEEE Conference on Industrial Electronics and Applications (ICIEA 2009)* (July 2009), 3944–3949.

[43] Parvar, M. R., Parvar, M. E., and Safari, S. A starvation free imlfq scheduling algorithm based on neural network. *International Journal of Computational Intelligence Research 4*, 1 (2008), 27–36.

[44] Pedone, F., Schiper, A., Urbán, P., and Cavin, D. Solving agreement problems with weak ordering oracles. *In Proceedings Fourth European Dependable Computing Conf.* (October 2002), 44–61.

[45] Rabin, M. O. Randomized byzantine generals. *24th Annual Symposium on Foundations of Computer Science* (November 1983), 403–409.

[46] Ren, W., Beard, R. W., and Atkins, E. M. Information consensus in multivehicle cooperative control. *Control Systems, IEEE 27*, 2 (April 2007), 71–82.

[47] Schwan, K., and Zhou, H. Dynamic scheduling of hard real-time and real-time threads. *IEEE Transactions on Software Ingineering 18*, 8 (August 1992), 736–748.

[48] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority inheritance protocols: An approach to real time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175 – 1185.

[49] Song, B., Kamal, A. T., Soto, C., Roy-Chowdhury, C. D. A. K., and Farrell, J. A. Tracking and activity recognition through consensus in distributed camera networks. *IEEE* (2010).

[50] Sprunt, B., Sha, L., and Lehoczky, J. P. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems 1* (July 1989).

[51] Spuri, M., and Buttazo, G. Efficient aperiodic service under earliest deadline scheduling. *In Proceedings of the IEEE Real-Time Systems Symposium* (December 1994).

[52] Spuri, M., and Buttazo, G. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems* (December 1996).

[53] Stankovic, J., and Ramamritham, K. The disign of the spring kernel. *In Proceedings of the IEEE Real-Time Systems Symposium* (December 1987).

[54] Stankovic, J., and Ramamritham, K. The spring kernel: a new paradigm for real-time systems. *IEEE Software* (May 1991).

[55] Strosnider, J. K., Lehoczky, J. P., and Sha, L. The deferrable server algorithm for enhancing aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers 4*, 1 (January 1995).

[56] Tanenbum, A. S. *Sistemas Operativos Distribuidos.* Prentice Hall, 1996.