UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TESIS DOCTORAL

Técnicas y extensiones para Java de tiempo real distribuido

Autor: Pablo Basanta Val
Directora: Marisol García Valls

20 de diciembre de 2006

Tribunal nombrado por el Mgfco. y Excmo. Sr. Rector de la Universidad Carlos III de Madrid, el día ___ de _____ de _____.

Presidente

Vocal

Vocal

Vocal

Secretario

Realizado el acto de defensa y lectura de la Tesis el día ___ de _____ de _____ en _____.

Calificación:

EL PRESIDENTE EL SECRETARIO LOS VOCALES

A meu pai,
onde queira que ande argallando,
e a miña mai e a miña irmá.

Caminante no hay camino,
se hace camino al andar.
Antonio Machado.

Un viaje de mil millas empieza con un paso.
Lao Tse.

Cuando llegue la inspiración,
que me encuentre trabajando.
Pablo Picasso.

Ora et labora.
S. Benito.

Agradecimientos

Hay quien compara el trabajo de realizar una tesis con el viaje de Dante a través del Infierno, el Purgatorio y el Paraíso. Durante su elaboración hay momentos de gran preocupación relacionados con el qué hacer por dónde empezar y otros en los cuales se trabaja sin llegar a vislumbrar ningún tipo de horizonte o meta final. Pero al final, tras un proceso más o menos complicado, se llega al Paraíso, olvidándonos de lo pasado en el Purgatorio y en el Infierno. El producto final, unas cuantas hojas, convertidas en delicatessen, pese a todo el esfuerzo invertido y debido a nuestra naturaleza humana, sigue siendo imperfecto y susceptible de ser mejorado una vez más. Es un proceso sin fin, similar al que realizaba Penélope cuando tejía por el día y destejía por la noche su tela esperando a que retornase Ulises de la guerra de Troya.

Un especial papel, similar al del poeta Virgilio o al de las anónimas ayudantes de Penélope, lo ha jugado Marisol, mi directora de tesis, ayudando a que mi tela vea la luz en algo menos de los veinte años invertidos por Penélope.

Otro papel también importante, no tanto en lo tecnológico sino más en un plano laboral y personal, ha sido el jugado por el Departamento de Ingeniería de Telemática de la Universidad Carlos III de Madrid. Tanto el buen ambiente de trabajo así como los buenos momentos compartidos han ayudo a la buena marcha de este trabajo. En este sentido también quiero agradecer al grupo de trabajo GAST, y en especial al laboratorio de tiempo real DREQUIEM, el apoyo prestado a esta tesis. En especial vayan mis agradecimientos dirigidos a José Jesús, buen compañero de despacho; Norberto, Iria y a Jesus compañeros de fatigas y buenos amigos; a Manolo, Pablo y Carlos Jesus, socios de la nevera; a Paco y al par de Ignacios; a Andrés, a Celeste, a Carlos y a Florina del equipo ubicuo; a Maricarmen y su pequeña; a Rosa y a Guillermo y a Carol; a Abelardo y a Luis; a David y a Alberto; a Carlos García Garcia, a Richi, a Iván y a Guillermo; a Goyo y a Rafa, los técnicos de laboratorio; a Jaime; y a Mario y a Pedro las partidas de tenis.

Quiero también agradecer de una forma también especial a mis tres compañeros de promoción, compañeros de trabajo y algunos también de piso: Iria, Norberto y Jesús el apoyo que siempre me han dado durante los buenos y malos momentos por los que ha atravesado esta tesis.

También quiero agradecer a la gente del grupo de tiempo real de Aveiro, con la cual viví la última etapa de la tesis, su gran acogida. En especial a la amistad mostrada por Luís Almeida, Paulo Pedreiras, Ricardo Marau y Valter. También un recuerdo para mi familia adoptiva en Portugal: a Javi, a su mujer y a sus dos hijos. Y por último, a mis amigas Bea y Katrin, por esos momentos de Mercado Negro.

A mi familia y amigos por el apoyo y el cariño que siempre me han demostrado tanto en los momentos más difíciles de mi vida como en los mejores.

Y a ti amigo lector, por el interés que muestras por esta tesis y por si acaso eres una de las muchas personas que deberían de aparecer explícitamente en estos agradecimientos y, por el contrario, has sido víctima de un olvido. Espero que disfrutes de la lectura de este documento.

Resumen

Al igual que las aplicaciones de propósito general, las de tiempo real no hacen más que aumentar en complejidad, forzando a que se estén explorando nuevas vías tecnológicas no consideradas previamente, como puede ser el empleo del lenguaje de propósito general Java para tales propósitos. En ese camino y a día de hoy, ya existen soluciones maduras que permiten el desarrollo de sistemas centralizados haciendo uso del lenguaje de programación Java, pero en el dominio de los sistemas distribuidos de tiempo real se sigue careciendo de soluciones que integren dichos lenguajes con los paradigmas de distribución de los diferentes middlewares de distribución.

En esta tesis se aborda dicha cuestión mediante una aproximación basada en la extensión de tecnologías ya existentes. Se propone un modelo computacional que está dotado de cierto grado de independencia de la tecnología de implementación, pero que a la vez está dotado de ciertas abstracciones como son el soporte para la recolección distribuida de basura o un servicio de nombres, propias del modelo de distribución Java RMI (Remote Method Invocation), y de otras como es la posibilidad de utilizar mecanismos de comunicación asíncronos, de utilidad en el desarrollo de muchos sistemas de tiempo real. Fijado un modelo, se proponen extensiones para el desarrollo de aplicaciones distribuidas de tiempo real basadas en el paradigma de distribución de la tecnología RMI y en la extensión de tiempo real RTSJ (Real Time Specification for Java), estableciéndose relaciones con otras aproximaciones ya existentes en el estado del arte. También se proponen una serie de extensiones al lenguaje RTSJ (al modelo de regiones, al de referencias y al de entidades concurrentes) que facilitan el desarrollo tanto de aplicaciones centralizadas como distribuidas de tiempo real. Y por último, se realiza una validación empírica en la que se verifica experimentalmente que el control de las prioridades de ejecución en el servidor, los modelos de gestión de memoria basados en regiones, el control del establecimiento de las conexiones, la inclusión de mecanismos de comunicación remota asíncronos, o incluso el tipo de datos intercambiados entre el cliente y el servidor durante la invocación remota, son capaces de influir significativamente en los tiempos de las aplicaciones Java de tiempo real distribuido.

Abstract

Like general purpose applications, the real-time ones are constantly increasing in complexity; this requires the exploration of new technological alternatives not previously considered, as it could be the use of the general-purpose and trendy development languages such as Java. Up to now, there are some fairly mature solutions for real-time Java; however, the area of distributed real-time systems still lacks similar support, thus requiring an extraordinary effort in order to integrate the current centralized real-time Java languages with the traditional middleware distribution paradigms.

This thesis address this problem by giving an approach based on the extensions inspired in already existing technologies. This work proposes a computation model that offers technological independence; at the same time, it provides a set of important abstractions such as support for a distributed garbage collection and a naming service, more related to the RMI (*Remote Method Invocation*) model, and other contributions like the possibility of using asynchronous remote invocations which are useful in the development of many real-time systems. Moreover, this work also proposes programming interfaces for this computation model; these interfaces are based on the RMI (Remote Method Invocation) and the RTSJ (Real-time Specification for Java) technologies, that are compared with other ones of the current state-of-the-art. Also, in the context of the RTSJ, a set of three extensions (one for the region model, another for the reference model and a last one for the concurrent entity model) are proposed to simplify the development of both, centralized and distributed real-time applications. Eventually, an empirical validation is presented in order to experimentally observe that the control of the execution priority at the server, the region-based memory management techniques, the use of pre-established connections or even the type of data exchanged between a client and a server, may influence significantly in the response time of the distributed real-time Java applications.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Desde ya el primitivo mundo Internet, con antecedentes que se remontan a la década de los 70 [84], los desarrolladores de aplicaciones comenzaron a familiarizarse con la complejidad presente en el proceso de construcción de sistemas distribuidos, y la percepción de que durante su desarrollo se repetían, una y otra vez, una serie de pasos altamente propensos a error, y que a su vez eran susceptibles de ser automatizados, dio pie a la aparición con el tiempo de nuevas tecnologías -DCE [57], DCOM [48], CORBA[132], RMI [190] o los actuales Servicios Web [120]- que comúnmente son denominadas middleware de comunicaciones [114].

Empujados también por una creciente demanda de mayores funcionalidades, los primitivos sistemas de tiempo real vienen sufriendo, desde sus primeros tiempos, una notable transformación que los está trasladando desde el mundo de lo cerrado y centralizado a otros escenarios más abiertos y distribuidos. Así, en este camino hemos visto como diferentes tecnologías reductoras de complejidad gestadas para dominios de propósito general -los sistemas operativos [165], los lenguajes de programación [5] y el propio middleware de comunicaciones [178]- han sido adaptadas a los especiales requisitos del tiempo real.

En esa continua e imparable carrera hacia algo más potente que nos permita abordar el desarrollo de nuevas aplicaciones, hoy en día, estamos siendo testigos de ciertos cambios en los lenguajes de programación de tiempo real. Así, si mientras en el pasado se han venido utilizando lenguajes de bajo, medio nivel o incluso alto nivel -C/C++ o ADA [5]-, en la actualidad existe un creciente interés en otros lenguajes de tiempo real con mayor grado de abstracción y popularidad. Y en especial, hacia el uso del lenguaje Java [88] para el desarrollo de sistemas de tiempo real [124] [25]. Todo ello, con la esperanza puesta en que el empleo de dichos lenguajes pueda servir como un nuevo elemento que permita amortiguar o bien reducir los costes de desarrollo y mantenimiento de unas aplicaciones de tiempo real cada vez más complejas y exigentes.

Desde el punto de vista investigador, este cambio plantea la cuestión de cuál ha de ser la relación que se ha de establecer entre estos nuevos lenguajes de programación y el middleware de distribución. Hoy en día, dos son las vías más estudiadas, una más conservadora que explora la utilización de RTCORBA [133] para tal fin y otra más innovadora -DRTSJ [93]- basada en el empleo de RMI.

En este escenario y de forma muy general, el objetivo perseguido por esta tesis se puede enunciar tal y como sigue: *estudio, análisis y definición de nuevas soluciones a la integración entre los lenguajes Java de tiempo real y los modelos del middleware de distribución tradicionales, tanto de propósito general como de tiempo real, con el fin último de contribuir al avance tecnológico de Java de tiempo real distribuido.*

El resto de este capítulo introductorio presenta una serie de secciones que introducen la tesis yendo desde una motivación general hasta la concreción de una aproximación y unos objetivos abstractos para su conjunto. La primera de ellas, la sección 1.1, especula sobre cuáles pueden ser los motivos que pueden haber impulsado la utilización de Java para el desarrollo de aplicaciones de tiempo real, presentando además tanto los diferentes retos que ya han sido abordados así como aquellos que aún se encuentran a la espera de soluciones específicas. La segunda es la sección 1.2 donde se fijan unos objetivos generales para la tesis. La sigue la visión de la tesis, en la sección 1.3, donde de forma resumida se presenta el trabajo realizado así como la aproximación seguida. Después, en la sección 1.4, se define la estructura de capítulos de esta tesis. Por último, en la sección 1.5, aparece la denominada historia de la tesis, recapitulando los diferentes estados por los que ha pasado este trabajo.

1.1. Motivación

Debido a la especial situación en la que se encuentran las tecnologías Java de tiempo real, no resulta difícil justificar la realización de una tesis en esta área de conocimiento. Si tuviésemos que justificar en un único párrafo la realización de una tesis doctoral en el dominio de las tecnologías Java de tiempo real distribuido, quizás, la mejor forma de hacerlo sería la de mencionar la existencia de una importante carencia tecnológica. El hecho de que en la actualidad existan fuertes líneas de trabajo encaminadas a la consecución de especificaciones para *Java de tiempo real distribuido* -DRTSJ [93] RTCORBA-RTSJ [129] RTCORBA-RTCORE [130]- que aún no han generado sus respectivas especificaciones así parece ponerlo de manifiesto.

Dejando un poco de lado lo que son estas carencias tecnológicas y yendo un poco más a lo que es el escenario en el cual se están gestando estas futuras especificaciones, lo cierto es que en un mundo como el de los sistemas embebidos de tiempo real, sean éstos o no distribuidos, donde cada día aparecen nuevas aplicaciones que son más complejas que las de antaño, cualquier tecnología que nos permita manejar esta creciente complejidad de forma más sencilla es bien recibida y resulta de interés. Y ésta, tal y como veremos a lo largo de esta sección, parece ser la razón subyacente que sustenta a Java.

Java y los sistemas de tiempo real: atractivos económicos y retos tecnológicos

Una de las cuestiones que seguramente nos deberíamos de plantear en primera instancia es quizás la de cuáles son los motivos que nos empujarían a utilizar un lenguaje como Java para el desarrollo de un tipo de aplicaciones, las de tiempo real, para las cuales no ha sido inicialmente concebido. Pues bien, el principal motivo es de

índole económica y tiene que ver con el coste de desarrollo y mantenimiento ofrecido por Java frente al de otros lenguajes.

En [139] se presenta un estudio sobre los costes de desarrollo de aplicaciones en Java y en C++, intentando comparar el rendimiento de ambos lenguajes. Durante el experimento en cuestión se analizaban dos proyectos de desarrollo software, uno desarrollado en Java y el otro en C++, comparándolos mediante el análisis de parámetros tales como el número de errores por línea de código, el número de horas totales empleado en la codificación de los programas, el tiempo empleado en arreglar errores y defectos, así como la productividad en número de líneas por hora. Los resultados mostraron claras ventajas de Java frente a C++ pues mientras el código Java analizado presentaba 61 defectos por cada mil líneas, en el código en C++ este valor se eleva hasta 82, siendo por lo general el tiempo medio necesario para subsanar un error en C++ el doble que el de un error en Java. A partir de estos resultados parciales y como una de las conclusiones más generales a las que llega, el informe estima que la productividad aumenta entre un 20 % y un 200 %, cuando en vez de utilizar C++, se toma como lenguaje de desarrollo a Java.

Otro trabajo [141], encargado por una de las mayores organizaciones del sector aeronáutico americano, basándose en el análisis de cuatro proyectos diferentes: uno desarrollado en Ada 95, otro en C++, otro en Java con pequeñas porciones de código en C, y otro realizado en Ada 83 estima que la productividad de Java, en número normalizado de líneas por esfuerzo del programador, es de 19,73, siendo la de C++ de 7,7, la de ADA 95 de 8,09 y la de ADA 83 de 0,95. Lo que porcentualmente significa que la productividad de Java, en los casos explorados, es un 150 % mayor que la de C++ y un 140 % mayor que la de ADA 95.

A este atractivo de productividad le hemos de sumar el hecho de que el negocio de los sistemas embebidos, segmento en cual se encuadran muchos de los sistemas de tiempo real, es un sector marcado por un fuerte crecimiento económico. Y es que aunque el mayor crecimiento del sector se produjo en la década de los años noventa donde se llegaron a alcanzar tasas de crecimiento de un 25 % anual [90], en el futuro más cercano aún se esperan crecimientos ciertamente fuertes. Las previsiones para el período 2003-2009 -ver la figura 1.1 y consultar en [104]- nos auguran un crecimiento en sus tres subsectores, siendo el del software para sistemas embebidos el que, porcentualmente, experimentará un mayor crecimiento: el negocio de los circuitos integrados (IC) crecerá un 14,2 % anualmente hasta alcanzar un volumen de 78,7 billones de dólares en el año 2009, el de los sistemas embarcados un 10 % y el del software para sistemas embebidos a un ritmo del 16 % anual hasta alcanzar los 3,5 billones de dólares en el año 2009.

Además, existe un último motivo relacionado con la especial situación que tradicionalmente ha acompañado al desarrollo de sistemas de tiempo real que también parece acrecentar el interés en Java. En la actualidad existe una gran gama de lenguajes, de dialectos y de herramientas altamente específicas que permiten el desarrollo de sistemas de tiempo real, pero no hay ninguna solución consolidada que se haya impuesto de forma contundente a las demás. Tal y como se recuerda en el prólogo de la especificación RTSJ [4], el éxito tecnológico no ha ido acompañado del comercial, encontrándose el mercado repartido entre diferentes tecnologías con mayor o me-

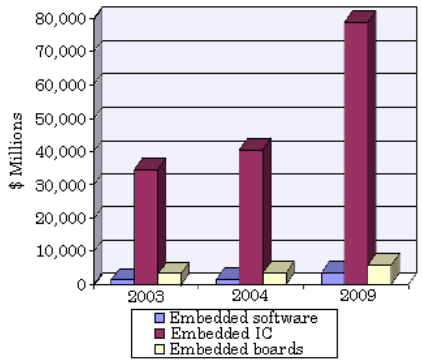


Figura 1.1: El mercado de los sistemas embebidos durante el período 2003–2009

nor cuota de mercado. En este escenario, el especial interés que suscita Java radica en la posibilidad de que una buena adecuación tecnológica, acompañada de la gran popularidad de la que en la actualidad gozan las tecnologías Java, sean capaces de desplazar o relegar a un segundo plano al resto de lenguajes específicos utilizados en la actualidad.

Dejando de lado lo que son los beneficios económicos y concentrándonos más en aspectos tecnológicos, debemos de percatarnos de que esta reducción en los costes deriva de una serie de características de diseño presentes en el modelo Java que no se encuentran disponibles en otros lenguajes de alto/medio nivel como C/C++. Tal y como refleja el documento NIST sobre los requisitos para Java de tiempo real (ver cuadro 1.1 y [112]), las características de Java que ayudan a marcar esta diferencia son: la alta abstracción, la facilidad de uso, la seguridad, el dinamismo, el nivel de reutilización, el grado de validación, la portabilidad, la distribución y la claridad en la semántica.

Pero desafortunadamente, pese a presentar un alto atractivo económico, Java, tal como aparece descrito en la especificación [88], no resulta de utilidad práctica para el desarrollo de sistemas de tiempo real debido a la carencia de ciertos mecanismos básicos de uso común en las aplicaciones de tiempo real. Tal y cómo se pone de relieve en el cuadro 1.2, existe una diferencia conceptual y filosófica fuerte entre Java y lo que es la programación de tiempo real tradicional. No se trata ya de que un tipo de lenguajes sean orientados a objetos y otros basados en objetos o en procedimientos, o que se soporte la escalabilidad o no, pues en último término éstas son características de los lenguajes de propósito general que son de interés para los de tiempo real. El principal problema radica en que en Java el control fino sobre los recursos de bajo nivel -memoria y procesador-, característico de los sistemas de tiempo real, no se encuentra presente.

Adaptar Java para que sea acorde con los requisitos de las aplicaciones de tiempo real no es tarea sencilla. En un principio podríamos pensar que el problema se puede reducir a la definición de nuevas interfaces que permitan el control fino de los recursos, pero la definición de estas interfaces puede entrar en serio conflicto con lo que es la filosofía de Java, pudiendo incluso llevarnos a escenarios donde las ventajas económicas anteriormente mencionadas ya no son válidas.

Característica	Razonamiento
Abstracción:	El alto nivel de abstracción de Java permite una mejor productividad del programador (aunque con mermas de eficiencia en tiempo de ejecución).
Facilidad de uso:	Java es relativamente más sencillo de dominar que C++.
Seguridad:	Java es relativamente seguro, salvaguardando los componentes software (incluyendo la propia máquina virtual) protegidos unos de otros.
Dinamismo:	Java soporta la carga dinámica de clases, y la creación dinámica de objetos e hilos.
Reutilización:	Java está diseñado para soportar integración entre componentes y su reutilización.
Grado de validación:	La tecnología Java ha sido desarrollada con especial consideración, pecando de prudencia en la utilización de conceptos y técnicas que han sido examinadas por la comunidad.
Portabilidad:	El lenguaje de programación Java y las plataformas Java soportan portabilidad de aplicaciones.
Distribución:	La tecnología Java soporta aplicaciones distribuidas.
Semántica:	Java provee una semántica de ejecución bien definida.

Cuadro 1.1: Ventajas de Java en el desarrollo de aplicaciones de tiempo real. Extraído y traducido del documento NIST sobre los requisitos para Java de tiempo real.

Un buen ejemplo de que esas controversias de índole filosófica pueden acabar desembocando en fuertes contradicciones tecnológicas nos lo ofrece la cuestión de la gestión automática de memoria [112] en Java. Este mecanismo, desde el punto de vista del programador, resulta vital pues reduce de forma altamente significativa los tiempos de desarrollo, pero desde el punto de vista de los sistemas de tiempo real también resulta ser difícil de utilizar pues introduce unas latencias en la ejecución que impiden el uso del lenguaje Java en una buena parte de las aplicaciones de tiempo real de la actualidad.

Según un informe de Xerox fechado en el 1980, la reducción en el número de horas de trabajo del programador que se puede conseguir mediante el uso de técnicas de recolección de basura suele rondar el 40 % y las latencias que puede introducir en las tareas de tiempo real del sistema, en algunas aplicaciones de tiempo real que requieren tamaños de montículo que varían entre los 100 Mb y 1 Gb de memoria, pueden superar los 30 segundos. Estos intereses altamente contrapuestos, desde un

Programación Java	Programación de tiempo real
Alto nivel	Bajo nivel
Popular	Arte especializado
Orientada a objeto	Orientada a procedimientos
Gestión automática de memoria	Gestión de memoria asumida por el programador
Componentes reusables	Componentes poco reusables
Flexible y adaptable	Comportamiento cableado
Escalable	No escalable
Portable	Dependiente de la plataforma
Optimiza rendimiento medio	Optimiza el peor de los casos
Control grueso sobre la concurrencia	Control fino sobre la concurrencia

Cuadro 1.2: Diferencias entre la programación Java y la de tiempo real

punto de vista práctico, nos obligan a buscar puntos de equilibrio entre el alto y el bajo nivel.

A la luz de estos hechos, en el dominio de los sistemas de tiempo real, no queda claro cuáles son las ventajas económicas que ofrece Java frente a otras alternativas ya más asentadas en este dominio como pueden ser el estándar RT-POSIX o las extensiones ADA de tiempo real [5]. El hecho de que sean necesarios cambios en los algoritmos que gobiernan dichos mecanismos, como el de la gestión de memoria, tan beneficiosos desde el punto de vista de la productividad, siembra cierta duda razonable sobre cuál va a ser el rendimiento final de estos lenguajes, una vez hayan sido convenientemente adaptados. En la actualidad, se carece de estudios comparativos serios que lo cuantifiquen y lo único de lo que se dispone es de conjeturas sobre sus bondades y sus inconvenientes. Ello nos permite afirmar que el principal reto al cual se enfrentan las tecnologías Java de tiempo real es el siguiente: *transferencia de las ventajas competitivas alcanzas en los entornos de propósito general al campo específico de los de tiempo real.*

Consciente tanto del atractivo como de las limitaciones tecnológicas de Java, la comunidad investigadora, ya desde finales de los noventa [124] ha dedicado muchos esfuerzos a solventar lo que son estas limitaciones tecnológicas, gestando una tecnología que se conoce como Java de tiempo real. En parte, el trabajo realizado ha sido de consenso filosófico entre lo que es el espíritu de Java y el de los sistemas de tiempo real, tratando de mantener las ventajas competitivas de Java en el marco de las aplicaciones de tiempo real. Como fruto de este trabajo se han ido gestado una serie de especificaciones de tiempo real -RTSJ [4] y RTCORE [3]- que se encuentran en proceso de madurez relativamente avanzado, existiendo incluso implementaciones comerciales para algunas de ellas. Sin embargo, la gran limitación tecnológica de estas soluciones, común a todas ellas y de gran interés para esta tesis, es la de que se concentran *únicamente en sistemas centralizados*, dejando de lado las cuestiones relacionadas con la distribución.

Java y los sistemas distribuidos de tiempo real: retos emergentes y carencias tecnológicas

Si el sector de los sistemas embebidos se encuentra en pleno proceso de explosión, el de los sistemas embebidos distribuidos no se queda atrás, proponiendo nuevos retos provenientes de la emergencia de una nueva generación de aplicaciones. Y es que aunque tanto los sistemas de tiempo real como los sistemas embebidos han sido considerados, históricamente, como sistemas de pequeña escala y autónomos, en la actualidad la tendencia es hacia incrementos notables en su funcionalidad, complejidad y escalabilidad. Más particularmente, los sistemas embebidos y de tiempo real están siendo unidos, a través de redes alámbricas o incluso inhalámbricas, para crear sistemas embebidos y distribuidos de tiempo real de gran escala, tales como los entornos de *tele-inmersion*, los sistemas *fly-by-wire aircraft*, los procesos de *automatización industrial* o los entornos *total ship computing*.

Todos estos sistemas se caracterizan por tener una serie de niveles de interdependencia, así como interconexiones de red, que coordinan sistemas finales locales y remotos y una serie de capas software que exhiben retos de su triple naturaleza. Como *sistemas distribuidos*, requieren de las capacidades necesarias para manejar conexiones y mensajes entre, posiblemente, redes heterogéneas. Como *sistemas de tiempo real*, requieren un control eficiente y predecible de los recursos extremo a extremo, tales como la memoria, el procesador y el ancho de banda. Y por último, como *sistemas embebidos*, presentan limitaciones tales como el tamaño, el peso, el coste o el consumo de energía que muchas veces limitan su capacidad de cómputo o de almacenamiento.

Aunque durante la década pasada se han realizado notables avances en cada uno de estos campos, en la actualidad aún existen grandes retos pendientes que esperan a ser abordados [150], destacando los de las tecnologías de componentes, el GRID o la propia Web. Las tecnologías de componentes, con cierto arraigo en sistemas distribuidos, han servido como un importante mecanismo reductor de la complejidad del software en sistemas de propósito general, pero sin embargo su nivel de adopción en los sistemas de tiempo real sigue siendo relativamente bastante bajo [180]. Tampoco, la muy emergente tecnología de computación distribuida GRID, ha explotado todas las ventajas que le ofrecen el middleware de comunicaciones y la tecnología de componentes [71]. Y por último, la popular *WWW*, a menudo asociada al significado *World Wide Wait*, sigue sin proporcionar modelos de calidad de servicio que sean capaces de satisfacer a los usuarios finales, que ven como sus navegadores se bloquean, algunas veces, arbitrariamente. En todos estos casos, el principal problema subyacente parece seguir radicando en una carencia de metodologías que posibiliten el desarrollo de sistemas de gran escala embebidos, distribuidos y de tiempo real.

Estas carencias metodológicas vienen acompañadas de ciertas carencias tecnológicas. Así, una de las soluciones middleware de tiempo real más conocidas y utilizadas en la actualidad, RTCORBA [151], aún presenta serias limitaciones a la hora de hacer frente a estos nuevos escenarios debido principalmente a su alta complejidad, a la falta de soporte específico para ciertas redes de tiempo real y también a dificultades a la hora de utilizar lenguajes de alto nivel para el desarrollo de sistemas distribuidos de tiempo real. Es conocido que la alta complejidad del modelo de interfaces de

CORBA [132], del cual RTCORBA es una extensión para tiempo real, resulta a veces difícil de compaginar con lo que son las restricciones de recursos impuestas por los sistemas embebidos, siendo necesarias simplificaciones en las especificaciones que nos permitan abordar exitosamente su inclusión. A esto se le ha de añadir el hecho de que muchas de las redes específicas de tiempo real utilizadas en la actualidad -buses CAN o TDMA- aún no han encontrado un soporte específico adecuado dentro del estándar [106] que aún sigue, en gran medida, orientado a las redes de propósito general como ATM [61]. Pero quizás de todas las limitaciones y para la que esta tesis propone soluciones, sea la imposibilidad de utilizar, de forma práctica, Java como lenguaje de desarrollo de aplicaciones debido, en gran parte, a que las nuevas especificaciones para Java de tiempo real aún no han sido armonizadas con RTCORBA [101] [67].

Consciente de que Java puede jugar un importante papel en el desarrollo de sistemas distribuidos de tiempo real, la comunidad investigadora ha empezado a trabajar de forma activa en la definición de especificaciones que faciliten el desarrollo de aplicaciones distribuidas de tiempo real. Existen varios procesos en marcha orientados en esa dirección -DRTSJ [93], RTCORBA-RTSJ [130] y RTCORBA-RTCORE [129]- que haciendo uso de las diferentes especificaciones existentes para sistemas centralizados -RTSJ [4] o RTCORE [3]-, definen soluciones intentando conjugar, de la mejor manera posible, lo que son las ventajas ofrecidas por las soluciones centralizadas con lo que son los modelos de distribución aplicables a Java. Las principales vías que se están explorando están bien basadas en RMI -DRTSJ [93]- o bien lo están en el modelo RTCORBA -RTCORBA-RTSJ [130] y RTCORBA-RTCORE [129]. Sin embargo, y a día de hoy, todos estos procesos parecen evolucionar lentamente y *se carece de estándares para Java de tiempo real distribuido* debido, en gran parte, a que los estándares centralizados sobre los que los distribuidos se deberían de asentar aún no han alcanzado el grado de madurez adecuado.

1.2. Objetivos de la tesis

Los objetivos específicos perseguidos en esta tesis son los siguientes:

- Caracterizar un modelo middleware con soporte para tiempo real basado en RMI, de tal manera que:
  - Se soporte un comportamiento tanto síncrono como asíncrono para la invocación remota.
  - Se incluya algún tipo de soporte para la recolección distribuida de basura.
  - Se incluya algún tipo de soporte para el servicio de nombres.
  - En todos los casos, se caracterice el funcionamiento interno del middleware de comunicaciones durante las comunicaciones remotas.
- Definir extensiones de tiempo real para RMI, de tal manera que:
  - Se haga el mayor uso posible del modelo de distribución de RMI.
  - Se haga uso también de las ventajas proporcionadas por la especificación de tiempo real RTSJ.

  - Se definan extensiones para el programador bajo la forma de nuevas clases.
  - Se definan extensiones en el protocolo de comunicaciones entre nodos mediante la inclusión de modificaciones en el protocolo de comunicaciones de RMI.

- Definir extensiones para el lenguaje de tiempo real RTSJ, de tal manera que:
  - Se definan interfaces de programador para cada una de las extensiones propuestas.
  - Se estudien las ventajas que implica su utilización desde el punto de vista del programador.
  - Se estudien los cambios que habría que realizar dentro de las máquinas virtuales de tiempo real existentes en la actualidad a la hora de darles soporte.

- Estudiar experimentalmente los tiempos de respuesta que son capaces de ofrecer las tecnologías Java de tiempo real distribuido, de tal manera que:
  - Se estudie la influencia que los diferentes esquemas de prioridades extremo a extremo pueden llegar a tener en el coste de la invocación remota.
  - Se estudie si el empleo de regiones en el servidor puede reducir los tiempos de respuesta de la invocación remota.
  - Se estudie cómo varía el consumo de memoria en función de los parámetros intercambiados en una invocación remota.
  - Se estudie el cómo varían las latencias introducidas por el middleware de distribución en función de los parámetros intercambiados.
  - Se estudie la influencia que el establecimiento de una conexión puede tener en el coste total de la invocación remota.
  - Se estudie cómo los mecanismos de comunicación asíncrona son capaces de reducir el tiempo de bloqueo máximo experimentado por los clientes.

1.3. Aproximación y visión general de la tesis

Partiendo de la idea de que la obtención de soluciones generales haciendo uso de tecnologías particulares resulta bastante difícil de emplear, esta tesis aborda la problemática de Java de tiempo real distribuido, de una forma más general. La idea principal es dejar en segundo plano tanto al lenguaje de programación como paradigma de distribución. Y frente a la posibilidad de definir integraciones basadas en las diferentes interfaces de las diferentes tecnologías ya existentes, en esta tesis se traslada la cuestión de Java de tiempo real distribuida, llevándola al campo de los recursos y su gestión; campo donde el grado de arbitrariedad es menor.

La gran ventaja de este enfoque es que nos permite abordar los problemas de forma mucho más general, evitándonos caer en las limitaciones particulares de una u

otra tecnología de tiempo real, pues bajo esta aproximación las tecnologías actuales no son más que medios que permiten llegar a implementaciones.

De forma gráfica, la figura 1.2 presenta tanto la aproximación tomada en esta tesis como el modelo se pretende desarrollar a lo largo de ella. En la parte izquierda se puede observar como la arquitectura aparece dividida en dos tecnologías: Java de tiempo real y Java de tiempo real distribuido.

Java de tiempo real es la tecnología mediante la cual el programador puede controlar una serie de recursos subyacentes como pueden ser la memoria, el procesador y el acceso a las comunicaciones remotas. Y Java de tiempo real distribuido se encarga, haciendo uso de gran parte de la tecnología Java de tiempo real centralizada, de enmascarar el proceso de invocación remota de tal forma que éste sea predecible extremo a extremo, mediante el control coordinado de los recursos de los diferentes nodos.

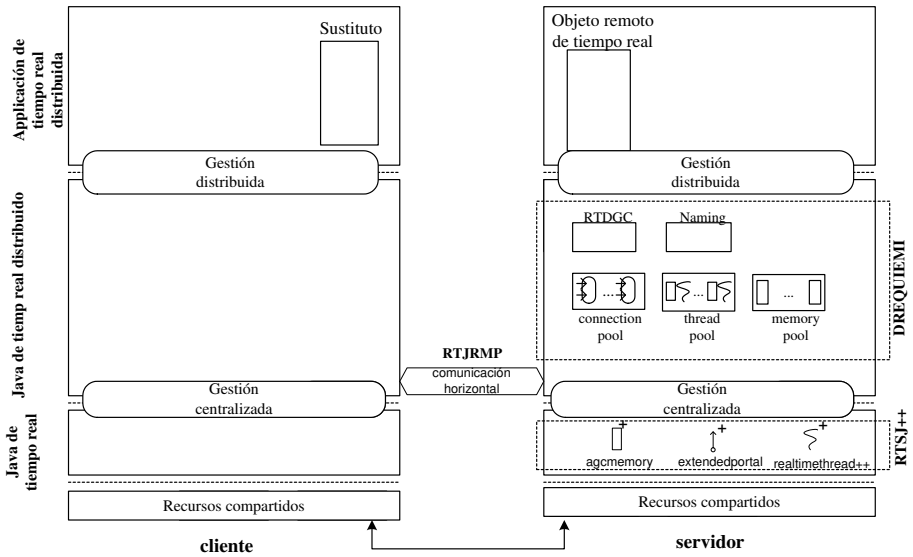


Figura 1.2: Aproximación a Java de tiempo real distribuido y visión general de la tesis

Ya en la parte de la derecha aparecen los diferentes elementos que esta tesis incorpora en este modelo de dos capas y a cuya caracterización y experimentación práctica se dedica el resto de esta tesis. Dentro del middleware de distribución esta tesis caracteriza el comportamiento interno del proceso de invocación remota decidiendo cómo se gestionan los recursos durante su ejecución y permitiendo su control mediante una interfaz denominada DREQUIEMI que está basada en el modelo computacional proporcionado por el modelo de distribución RMI. Y dentro del middleware de infraestructura esta tesis define un modelo mejorado del middleware RTSJ denominado RTSJ++.

En el middleware de distribución RMI, las principales modificaciones que intro-

duce son dos:

- La definición de un modelo de computación que caracteriza el comportamiento interno del middleware de distribución. Este modelo define cómo se utilizan los recursos tanto en las invocaciones remotas síncronas como asíncronas y cómo el recolector distribuido de basura y el servicio de nombres hacen uso de los recursos del sistema. Internamente, en este modelo, el middleware de distribución asume el control de la gestión de los recursos mediante la incorporación de tres nuevas entidades: un *connectionpool*, un *threadpool* y un *memorypool*. Esta caracterización incluye además dos servicios: uno interno encargado de la recolección distribuida de basura (RTDGC) y otro de nombres (Naming).
- La definición de un sistema de interfaces de programador denominado DREQUIEMI. Este sistema de interfaces de programador permite parametrizar el comportamiento interno del middleware de distribución. Ofrece soporte para sustitutos y objetos remotos de tiempo real y además tiene un nivel de parametrización adicional que le es proporcionado por el *DistributedScheduler*. También permite configurar los recursos que internamente son gestionados por el middleware de distribución. Por último, en su definición se han incorporado también una serie de cambios en el protocolo de comunicaciones de RMI, el JRMP, que permiten el intercambio de información necesaria para realizar la comunicación.

Y en el middleware de infraestructura RTSJ, incorpora tres nuevas extensiones:

- `AGCMemory`. Esta extensión proporciona un modelo de regiones con más capacidades de detectar basura que el proporcionado en la actualidad por RTSJ, acercando el modelo de regiones al de recolección de basura mediante la incorporación de soporte para la recolección de basura flotante.
- `ExtendedPortal`. Esta extensión proporciona un mecanismo que permite el acceso a regiones prohibidas más versátil que el propuesto por el actual portal de RTSJ, incorporando además la posibilidad de manejar una semántica de tipo *strong* o *weak*.
- `RealtimeThread++`. Esta extensión generaliza el modelo de entidades concurrentes de RTSJ simplificando el modelo de concurrencia del actual RTSJ, permitiendo que un hilo decida durante su ejecución y de forma totalmente dinámica el tipo de relación que desea mantener con el recolector de basura.

Ya de una forma más empírica y dentro de lo que es el middleware de distribución Java, se estudia cómo:

- El empleo de un esquema de prioridades en el servidor puede reducir el tiempo de respuesta de los clientes de mayor prioridad.
- El empleo de regiones en el servidor puede reducir el tiempo de respuesta de las aplicaciones distribuidas Java de tiempo real.

- El flujo de datos intercambiado entre el cliente y el servidor influye en el coste (memoria y procesador) de la invocación remota.

1.4. Estructura de la tesis

Esta tesis se ha organizado mediante capítulos que de forma más o menos incremental van desarrollando los diferentes objetivos generales que se han fijado para ella.

Capítulo 1 Este capítulo es el actual y aparece dedicado a la presentación del problema que se pretende abordar en esta tesis. En él se define el escenario de trabajo, presentando el reto tecnológico que se pretende abordar en esta tesis, y partir de éste, se define una aproximación o línea de trabajo para lo que va a ser el resto de la tesis, una serie de objetivos generales para la misma, la presente estructuración en capítulos y la historia de la tesis.

Capítulo 2 Este capítulo es el estado del arte y aparece dedicado a la revisión crítica de aquellas técnicas y tecnologías que son relevantes para el desarrollo de la presente tesis. Esto incluye una revisión de los diferentes middlewares de comunicación, las técnicas de gestión de recursos más utilizadas en el desarrollo de sistemas de tiempo real, los diferentes lenguajes Java de tiempo real centralizado, el middleware de distribución de tiempo real y los trabajos que abordan específicamente la cuestión de Java de tiempo real distribuido.

Capítulo 3: Éste es el primero de los capítulos donde se realizan contribuciones. En él se construye un modelo computacional que ofrece una funcionalidad básica para el desarrollo de aplicaciones de tiempo real y que contempla, de alguna manera, la gestión de recursos distribuidos. El modelo proporciona soporte a la invocación remota síncrona, la asíncrona, a un servicio de sistema síncrono de recolección distribuida de basura y a otro de usuario de nombres.

Capítulo 4: En este capítulo se proponen extensiones para Java de tiempo real distribuido basadas en el modelo de distribución RMI y RTSJ denominadas DREQUIEMI. Estas extensiones están basadas en el modelo definido en el capítulo anterior y han sido concebidas para ser fácilmente extensibles. Tras ello, se realiza una comparación entre el modelo de interfaces desarrollado y el de otras aproximaciones a RTRMI, tratando de establecer relaciones entre ambas.

Capítulo 5: En este capítulo se proponen una serie de mejoras de orden general que ayudan no sólo a la hora de implementar DREQUIEMI sino que además pueden ser beneficiosas para el conjunto de las tecnologías Java de tiempo real. Se trata de tres extensiones todas ellas de alguna manera ligadas a diferentes aspectos del modelo de gestión de memoria de RTSJ. En la primera de ellas se propone un modelo extendido para el modelo de regiones; en la segunda para el sistema de referencias y en la tercera para el de entidades concurrentes. De forma conjunta reciben el nombre de RTSJ++.

Capítulo 6: En este capítulo se pretende verificar de forma empírica que diferentes de las técnicas propuestas para DREQUIEMI resultan interesantes. Esto es, que pueden ayudar a reducir notablemente los tiempos de respuesta de las tareas distribuidas. Se analizará cómo la gestión del procesador basada en prioridades, la recolección de basura en el servidor realizada mediante el empleo regiones, la gestión de conexiones realizada en el cliente, la posibilidad de realizar comunicaciones asíncronas y el empleo de diferentes tipos parámetros de invocación influyen en el coste total de la invocación remota.

Capítulo 7: En este capítulo se ofrecen una serie de conclusiones sobre el trabajo realizado en esta tesis, recapitulando cuáles han sido los principales resultados que se han obtenido así como cuáles han sido las principales contribuciones que han sido realizadas al estado del arte. También incluye una serie de líneas futuras de trabajo a seguir explorando.

1.5. Historia de la tesis

Esta tesis comienza en curso académico 2002-03, donde el doctorando comenzó sus estudios en el programa de doctorado de Tecnologías de las Comunicaciones de la Universidad Carlos III de Madrid. En ese mismo año se produjo el primer contacto con las tecnologías Java de Tiempo Real y se vio que esa área de conocimiento, relativamente poco explorada, podía ser un buen campo en el que realizar una tesis doctoral. Así, ese mismo curso y durante el segundo cuatrimestre del primer año de doctorado se realiza un trabajo analizando la especificación RTSJ.

De este análisis surgió la convicción de que la gestión automática de memoria y predecible basada en regiones era un tema bastante novedoso. Y ya en el segundo año del programa de doctorado donde se realizaban tres trabajos tutelados, se exploró en dos de ellos la gestión automática de memoria, dando lugar a dos contribuciones específicas: el `NoHeapRemoteObject` [19] [20] y la `AGCMemory` [16] [17].

Tras haber recibido diversos comentarios de los revisores sugiriendo que se incorporasen tanto una serie de mecanismos que permitiesen controlar el establecimiento de las conexiones como el empleo de esquemas de prioridades en el servidor, se empezó a trabajar en la caracterización de RTRMI, dando lugar a un modelo para el comportamiento interno y a interfaces de programación. El objetivo perseguido era el de obtener una caracterización del funcionamiento del middleware de distribución tal que pudiese ser aplicada en otros modelos de distribución, en un intento de dotar a la aproximación de un alto grado de abstracción. Para ello se complementa el modelo de RMI con ideas tomadas de RTCORBA, incorporando un esquema de prioridades extremo a extremo, control sobre las conexiones y otras más novedosas como la utilización de bloques de memoria reutilizables, no presentes en RTCORBA. Tras haber definido el modelo de computación para el modelo de distribución y haciendo uso de él se procede a aplicarlo al modelo de RMI y de RTSJ, dando lugar a DREQUIEMI.

Durante el proceso de implementación de DREQUIEMI se fue creando la impresión de que ciertos aspectos relacionados con el modelo de gestión de memoria de RTSJ eran claramente mejorables y de utilidad para el programador, motivo por el

cual se desarrollaron extensiones, juntándose con la extensión `AGCMemory`, bajo el nombre epígrafe común RTSJ++. Tras la finalización de la escritura de este capítulo publica el `ExtendedPortal` [18] en el JTRES'06.

Por último, tras realizar las últimas modificaciones de la implementación RMI-OP disponible para jTime, ya en el 2006 y en la Universidad de Aveiro, se realizaron las mediciones empíricas que corroboraban la utilidad de buena parte de los mecanismos propuestos en esta tesis. También se realiza un artículo [15] con diversas medidas experimentales obtenidas de la evaluación empírica a la que es sometido DREQUIEMI.

El proceso de escritura de la tesis se ve entremezclado con el de desarrollo y el de publicación. Comienza el 1 de septiembre del 2005 y a finales de mayo del 2006 se envía oficialmente el proyecto de tesis doctoral que finalmente es aprobado en octubre del mismo año. Ya a mediados de diciembre del mismo año, se terminan de realizar las últimas modificaciones a la presente memoria.

Capítulo 2

Estado del Arte

El hecho de que se pretenda realizar una aproximación amplia a Java de tiempo real distribuido impide restringir el análisis a las diferentes aproximaciones de integración- DRTSJ [93], RTCORBA-RTSJ [130], RTCORBA-RTCORE [131]- citadas en el capítulo de introducción, siendo necesario estudiar conceptos más generales.

Así pues, el estado del arte aparece organizado en una serie de secciones de forma incremental; partiendo de los conceptos más abstractos -el tiempo real y el middleware- se va concretando hacia el trabajo más relacionado con esta tesis: el middleware de distribución Java de tiempo real. Arranca la sección 2.1 donde se hace un recorrido por los conceptos básicos de los sistemas de tiempo real. Después, la sección 2.2 presenta el concepto de middleware, de capas y de modelos de comunicación, y dos tecnologías altamente relevantes para la tesis: el middleware de distribución de propósito general RMI y el de tiempo real RTCORBA. Ya bajo el sesgo de las tecnologías Java y en la sección 2.3, se hace una revisión del estado actual de Java de tiempo real para sistemas centralizados. Le sigue la sección 2.4 en la cual se presentan las diferentes aproximaciones existentes en la actualidad para desarrollar Java de tiempo real distribuido. Por último, la sección 2.5 ofrece un resumen y conclusiones sobre cuáles son las oportunidades investigadoras ofertadas en la actualidad por las tecnologías Java de tiempo real.

2.1. Sistemas de tiempo real

Esta sección versa sobre los diferentes algoritmos más utilizados en diferentes sistemas de tiempo real tales como sistemas operativos, lenguajes y middlewares tanto centralizados como distribuidos de tiempo real. Para ello, se hace un repaso de las principales técnicas de gestión existentes tanto para sistemas centralizados como distribuidos relacionadas con la gestión del procesador, viendo también la gestión de memoria en tiempo real. Se comienza con conceptos básicos de los sistemas de tiempo real -sección 2.1.1 - para después ver las técnicas de gestión de procesador más utilizadas en entornos centralizados -sección 2.1.2 -, los protocolos de sincronización de tiempo real -sección 2.1.3-, así como los cambios que la planificación distribuida introduce en la centralizada -sección 2.1.4-; para después -sección 2.1.5- tratar el tema de la gestión de memoria de tiempo real. Pero antes de empezar, se tratará de

definir qué se entiende por un sistema de tiempo real.

Un sistema de tiempo real se suele definir[1] como: *Cualquier sistema en el cual el instante temporal donde se produce la respuesta es significativo.* Significatividad que suele venir impuesta por el entorno físico en el que está circunscrito; permitiéndonos afirmar que sistemas de control de vuelo, las cadenas de montaje de las plantas de manofacturación, las telecomunicaciones, los sistemas de control o los marcapasos pueden ser de tiempo real, pues en todos ellos una respuesta fuera de tiempo hace que el funcionamiento del sistema sea incorrecto o peligroso. La gran diferencia para con uno de propósito general es por tanto que los de tiempo real exigen garantías adicionales sobre el comportamiento temporal del sistema.

Por lo general, con el fin de conseguir un mayor grado de determinismo, con el cual se sea capaz de asegurar que las restricciones temporales de las aplicaciones se satisfacen adecuadamente, los sistemas de tiempo real suelen proporcionar un control fino sobre los recursos compartidos tales como el procesador, la memoria o incluso la red. Este control fino de los recursos combinado adecuadamente con los requisitos de las diferentes aplicaciones de tiempo real, es lo que nos permite garantizar que los diferentes plazos de las tareas se ven satisfechos.

Tradicionalmente uno de los recursos cuya gestión eficiente y predecible más se ha investigado es el procesador. A su análisis se ha dedicado una disciplina que generalmente se conoce como la *planificación de tiempo real* [155] que aparece consagrada a la búsqueda de disciplinas de gestión del procesador tanto centralizadas como distribuidas que permitan la satisfacción de los diferentes plazos de las diferentes tareas de tiempo real de forma óptima. Sin embargo, a medida que los sistemas se han hecho más y más complejos -este es el caso de Java- otros mecanismos de gestión de recursos, tradicionalmente no considerados, como podría ser el caso de la gestión de memoria en tiempo real, comienzan a despertar un mayor interés.

2.1.1. Tareas de tiempo real

Un sistema de tiempo real se suele modelar como un conjunto de tareas $\tau_1, \ldots, \tau_n$ donde cada una de las tareas presenta una serie de requisitos y restricciones temporales.

Dependiendo de las especiales características de éstas, tradicionalmente, se suele distinguir tres tipos de tareas básicas: *periódicas*, *esporádicas* y *aperiódicas*. Las tareas periódicas son aquellas en las cuales su activación se produce con una periodicidad fija. Los parámetros que las suelen caracterizar son: el instante en el cual la tarea se encuentra por primera vez en el sistema (A); el instante en el que la tarea está lista por primera vez en el sistema (J); el consumo de procesador realizado en cada una de las activaciones (C); el plazo de la tarea (D) y el período de activación (T). En contraposición a las periódicas, las esporádicas y las aperiódicas carecen de esta propiedad de periodicidad. Las tareas esporádicas se caracterizan por tener períodos de activación variables, que están acotados por un T_{min} y las tareas aperiódicas por la ausencia de dicho tipo de cotas.

[1] Oxford Dictionary of Computing

Además, en función del tipo de los requisitos de las diferentes tareas de tiempo real, se suele distinguir dos grandes tipos de tareas de tiempo real. Las que tienen unos requisitos extremos o *hard real-time*. Y las que poseen unos requisitos de predictibilidad más bien bajos y donde se transige alguna pérdida de plazo o *soft real-time*.

La planificación clásica, la más conocida y estudiada hasta la actualidad, suele trabajar con tareas periódicas que además tienen unos requisitos de predictibilidad extremos, *hard real-time*. En este tipo de casos, las tareas esporádicas y aperiódicas se suelen aproximar, muchas veces de forma pesimista pero segura, por tareas periódicas. Otras veces son agrupadas en servidores [108].

2.1.2. Planificación centralizada

Una vez caracterizadas las tareas de tiempo real, el siguiente paso a dar es establecer cómo repartir, en función de los requisitos de cada una de las tareas, el procesador. Este reparto consiste en decidir en cada instante temporal cuál de las diferentes tareas de tiempo real, existentes en el sistema, es la que accede al procesador. En una primera aproximación se suele considerar un modelo de tareas periódicas en el que tan sólo existe un procesador y en el que las tareas son independientes entre si.

Planificación estática

La planificación estática se caracteriza por realizar un reparto del procesador en función de la descripción de las tareas periódicas del sistema. Dos técnicas, EC y FPS, son las más relevantes en el estado del arte.

La técnica más sencilla, que estuvo plenamente vigente hasta la décadas de los 80, fueron los *ciclos ejecutivos* **(EC)** [14]. Su principal limitación, que fue la que motivó la aparición de la planificación basada en prioridades, es la dificultad que este tipo de mecanismo tiene a la hora de generar ciclos a partir de la definición de las tareas.

A fin de mitigar estas limitaciones, se trabajó en algoritmos encuadrables en lo que son los algoritmos de tipo *Fixed Priority Scheduling* **(FPS)**, siendo el trabajo de Liu y Layland [113] del *rate-monotonic* uno de los que mayor impacto ha alcanzado. La idea básica de este algoritmo es asignar una prioridad a cada una de las tareas en función del plazo máximo de éstas, de tal manera que a las tareas con un mayor plazo, D_i, se les asignan prioridades mayores, siendo la tarea con menor D_i la más prioritaria de nuestro sistema. Esta asignación es estática y se mantiene en cada período de activación de la tarea.

Basándose en esta asignación, se desarrolló un *test de planificabilidad* que a partir de las prioridades de las tareas, nos permite garantizar el cumplimiento de sus plazos.

Sea la tarea τ_i; su tiempo de respuesta, R_i, se puede obtener mediante la resolución de la siguiente ecuación recursiva:

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil C_j$$

Su mayor limitación es que, en el peor de los hipotéticos casos, la cota de utilización máxima del procesador puede caer asintóticamente hasta valores próximos al 69.3 %.

Planificación dinámica

La siguiente aproximación existente consiste en utilizar el instante de activación de cada tarea para realizar un cálculo de la prioridad de forma dinámica. Su principal ventaja sobre la planificación estática radica en el hecho de que es teóricamente más eficiente y su mayor inconveniente es la dificultad adicional que conlleva la implementación eficiente de este tipo de algoritmos. Los algoritmos más conocidos son EDF, LLF y MAU.

El algoritmo *Early Deadline First* **(EDF)** [113] es el algoritmo más popular de todos los dinámicos. En él, la idea básica es que en vez de considerar todas las tareas de forma global, se observan en cada instante temporal aquellas tareas que están activas, asignándole el procesador a aquella de menor período. Con esta técnica la utilización máxima teórica del procesador crece hasta el 100 %. Su principal inconveniente deriva de su mal comportamiento ante sobrecargas del sistema.

Este problema es mitigado por el algoritmo *Least Laxity First* **(LLF)** [118]. Este algoritmo asigna el procesador a aquella tarea con menor laxitud donde ésta se define como la diferencia entre tiempo máximo de finalización de la tarea y el actual.

Por último, es de destacar el algoritmo *Maximum Accrued Utility* **(MAU)**[166], capaz de emular a FPS, a EDF o a LLF mediante una apropiada configuración de sus parámetros.

2.1.3. Algoritmos de sincronización

Todos los algoritmos definidos con anterioridad están basados en la idea de que las tareas son independientes y de que no hay ningún tipo de interdependencia entre ellas. Pero lo cierto es que éste no suele ser el caso más general, pues muchas de las tareas de tiempo real necesitan compartir datos haciendo uso de mecanismos de acceso en exclusión mutua. Y el hecho de que los datos puedan ser accedidos por dos tareas de diferente prioridad de tal manera que la tarea de mayor prioridad tenga que esperar a que la de menor prioridad abandone la zona no compartida provoca un efecto conocido como *inversión de la prioridad*, perjudicial a la hora de garantizar el cumplimiento de los plazos de las tareas de mayor prioridad. A fin de reducir dicho efecto se han desarrollado una serie de algoritmos -PIP, PCP, SRP, WFO y LSFO- que reducen la inversión de prioridad experimentada extremo a extremo.

El protocolo *Priority Inheritance Protocol* **(PIP)** [156] ha sido una de las primeras soluciones desarrolladas y su principal limitación es el elevado número de cambios de contexto al cual nos puede conducir su utilización.

El protocolo *Priority Ceiling Protocol* **(PCP)** [157] es capaz de reducir este número de cambios de contexto mediante la asignación de una prioridad, *prioridad de techo*, a cada uno de los recursos compartidos.

Por último, otra alternativa es utilizar los *Lock Free Shared Objects* **(LSFO)** [9] o los *Wait Free Objects* **(WFO)** [10] que son no bloqueantes.

2.1.4. Planificación distribuida

Los algoritmos presentados hasta ahora están pensados para sistemas centralizados, con un único procesador, y no contemplan la posibilidad de que existan múltiples procesadores. A la hora de generalizar los resultados vistos hasta ahora a este nuevo escenario es necesario volver a tratar con una serie de cuestiones como son la asignación de prioridades a tareas, la asignación de éstas a procesadores, la sincronización distribuida o el análisis de planificabilidad.

Uno de los trabajos de más impacto en planificación distribuida ha sido el realizado por Dhall y Liu [95], que aborda la planificación distribuida mediante la extensión de las técnicas basadas en prioridades fijas. En él se diferencian dos tipos de sistemas: *particionados*, donde cada tarea es asignada a un procesador y los *globales*, donde las tareas compiten por todos los procesadores.

Otro de los grados de libertad que ofrece el empleo de múltiples procesadores es la asignación de tareas a un procesador. La asignación óptima de tareas a cada uno de los procesadores del sistema distribuido, tal y como se demuestra en [64] es un problema np-completo. En consecuencia, lo que muchos autores han hecho es recurrir al empleo de heurísticos que ofrecen resultados bastante desiguales. El más conocido es el *Rate Monotonic First Fit* **(RMFF)** [95].

Al igual que sucede en los sistemas centralizados, en los distribuidos se pueden obtener beneficios de la existencia de algoritmos que controlen la inversión de prioridad. Pero sin embargo, aunque algunos autores han abordado su adaptación [143], a día de hoy, aún no existe ningún tipo de solución de carácter general que pueda compararse con PIP o PCP. En consecuencia, lo más normal es que en la comunicación entre los diferentes nodos del sistema se utilicen protocolos no bloqueantes de tipo WFO o LSFO.

Por último, otra línea de investigación consiste en determinar la planificabilidad de un sistema distribuido. En la actualidad existen varias técnicas de análisis, destacando el análisis de Tindell [176] junto a la optimización de Palencia [72].

Para concluir, se puede afirmar que los resultados existentes para sistemas distribuidos tienen un grado de madurez mucho menor que los existentes para sistemas centralizados. Ello es así debido a que si bien para sistemas centralizados existe un buen conocimiento de los diferentes mecanismos -FPS, PIP- que permiten el desarrollo de sistemas de tiempo real de una forma óptima; en sistemas distribuidos aún no se disponen de resultados similares, teniendo muchas veces que recurrir a heurísticos.

2.1.5. Gestión de memoria

Al contrario de lo que sucede con la gestión del procesador, la gestión de la memoria, tradicionalmente, no ha sido tan considerada a la hora de construir sistemas de tiempo real. Esto era así porque existía la posibilidad de realizar una reserva inicial de la memoria que se iba a utilizar. Pero lo cierto es que, a día de hoy, las tareas son cada vez más dinámicas y que en muchas de ellas en vez de realizar una reserva inicial de memoria puede ser más interesante la realización de una reserva de memoria de forma dinámica, durante fases de la ejecución sujetas a restricciones temporales. Este cambio hace que los diferentes algoritmos empleados a la hora de gestionar la

memoria tengan un nuevo requisito: el de ser predecibles.

De forma general, la gestión de memoria se enfrenta al problema de tener que asignar y liberar, bajo demanda, bloques de memoria. Se distinguen dos familias de técnicas: la *gestión dinámica* [188] y la *gestión automática* [187] de memoria. La gestión dinámica de memoria se caracteriza por introducir una serie de primitivas, tales como `malloc` y `free` que permiten la reserva y la liberación de bloques de memoria de tamaño arbitrario. En la gestión automática de memoria el sistema subyacente asume la liberación de la memoria y las primitivas de tipo `free` son reemplazadas por técnicas de recolección de basura. Según se hable de un tipo u otro de gestión, la naturaleza de las impredictibilidades y la forma de abordarlas variará.

Desde el punto de vista del tiempo real, el principal problema que existe a la hora de utilizar gestión de memoria durante la ejecución de las tareas de tiempo real deriva del hecho de que el coste de ejecución de estos algoritmos es altamente variable. Algunas de las operaciones realizadas por estos algoritmos de gestión como pueden ser el defragmentado de la memoria [188] o la propia necesidad de algunos recolectores de basura de explorar la estructura de memoria principal [187] para determinar qué objetos se pueden recolectar y cuáles no, introducen costes temporales tan significativos que pueden provocar la pérdida de plazos de muchas tareas de tiempo real.

Para solucionar esta serie de problemas lo que la comunidad investigadora ha hecho es refinar los diferentes algoritmos de gestión dinámica y automática de memoria existentes en el estado del arte, de tal manera que los costes extraordinarios o bien desaparecen o bien no afectan al cumplimiento de los plazos de las tareas de tiempo real. Dentro del campo de la gestión dinámica trabajos relevantes son la optimización TLSF [116] y la técnica de pre-fragmentación de Lindblad [110]. Por último, dentro del área de la gestión automática de memoria uno de los trabajos más completos realizados a tal respecto es el de Henrikson [75] donde se modela al recolector de basura como una tarea de tiempo real periódica, deduciendo matemáticamente cotas para su coste y su periodicidad.

Para concluir con la gestión de memoria en tiempo real podemos decir que el empleo de estas técnicas supone un aumento en la cantidad de recursos requeridos a la hora de proporcionar un soporte adecuado a un sistema de tiempo real. Por lo general, se sigue la máxima de que a mayor funcionalidad soportada en la gestión de la memoria, mayores son los costes computacionales adicionales que se han de asumir. Así, el tener gestión automática de memoria de tiempo real suele ser más costoso, en términos de procesador y de memoria, que el tener gestión dinámica de memoria de tiempo real, y ésta última suele ser más costosa que el no tener ningún tipo de elemento gestor.

2.1.6. Conclusiones

Tras analizar el estado del arte se podría decir que en sistemas de tiempo real no existe ningún tipo de resultado general, aplicable en un amplio rango de aplicaciones que nos permita hacer una gestión de los recursos óptima. Más bien lo que existe son una serie de técnicas dependientes del dominio de aplicación, con mayor o menor

grado de aplicabilidad, que son capaces de permitirnos acotar el tiempo máximo de respuesta de las diferentes tareas de tiempo real.

Las técnicas descritas en esta sección constituyen pues el superconjunto de los algoritmos de gestión de recursos que de alguna forma deberían de ser considerados por Java de tiempo real distribuido. No parece esperable que la inclusión de mecanismos reductores de la complejidad, como puede ser el propio middleware de distribución, sean capaces de ocultar dicha gestión al desarrollador completamente. Por tanto, idealmente cualquier aproximación a Java de tiempo real distribuido debería de, en la medida de lo posible, dar cabida a todas ellas.

2.2. Middleware de comunicaciones

Tras haber caracterizado los sistemas de tiempo real, la segunda componente a estudiar es la del middleware de comunicaciones. En su estudio se tratarán conceptos más abstractos como es el de los modelos de comunicación extremo a extremo y otros más ligados a tecnologías concretas. Así, se comenzará caracterizando cuáles son las diferentes capas o niveles en los que se puede trabajar cuando hablamos de middleware -sección 2.2.1- dando ejemplos de tecnologías representativas. Después se verá cuáles son los diferentes paradigmas de comunicación extremo a extremo utilizados más comúnmente -sección 2.2.2- en los principales middlewares de distribución. Y para terminar se estudiarán dos casos particulares: el del middleware de comunicaciones de propósito general RMI -sección 2.2.3- y el de tiempo real RTCORBA -sección 2.2.4-, de gran interés para la presente tesis. Pero al igual que se hizo en la sección previa, se comenzará por la definición de middleware.

Debido a que las tecnologías middleware han sido utilizadas por diversos grupos y en diferentes dominios de aplicación [178], eg. el acceso a bases de datos remotas, en sistemas de transacciones distribuidas o en sistemas cooperativos, se han generado múltiples concepciones de lo que es el middleware. Sin embargo, la idea subyacente en todos los casos es común, la de proveer alivio a los problemas de heterogeneidad y de distribución provenientes de la existencia de multiples tipos de hardware y de sistemas operativos. Y por tanto, se podría definir como: *una capa entre el sistema operativo distribuido y las aplicaciones que intenta resolver el problema de la heterogeneidad y de la distribución.*

2.2.1. Estructura de capas del middleware

A la hora de analizar más en detalle lo que son las diferentes capas presentes en el middleware, se toma como punto de partida el modelo descrito por Schantz en [149]. En este modelo se reparte lo que es la problemática de la construcción del middleware de comunicaciones entre cuatro capas. Tal y como se muestra en la figura 2.1, las capas que se identifican son las siguientes: *middleware de infraestructura*, *middleware de distribución*, *middleware de servicios comunes* y *middleware de servicios específicos de dominio.*

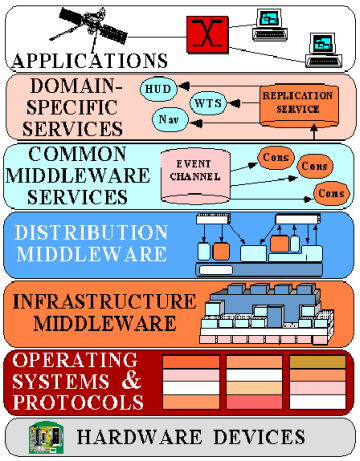


Figura 2.1: Modelo en capas del middleware

Así, en la parte más cercana al hardware se encuentra lo que denominamos *middleware de infraestructura*. Esta capa encapsula y mejora la comunicación con el sistema operativo nativo, con el fin de crear componentes reutilizables. Su misión es eliminar muchos aspectos tediosos y propensos a error relacionados con el mantenimiento y el desarrollo de las aplicaciones distribuidas a través de la definición de interfaces de bajo nivel. En este tipo de middleware podríamos situar a la máquina virtual de java (JVM) [111] y a .NET [146], pues ambos proveen una plataforma de ejecución capaz de abstraer al programador de las peculiaridades del sistema operativo. Incluso algunas interfaces de sistema operativo tales como POSIX [41], cabrían en esta categoría dado que estas interfaces también proveen un cierto grado de abstracción de los detalles particulares del sistema operativo.

Justo encima del middleware de infraestructura se sitúa el *middleware de distribución* que es el encargado de proveer la abstracción de programación distribuida. El middleware de distribución posibilita que los clientes puedan programar aplicaciones distribuidas de forma sencilla, ocultando detalles como son la localización del servidor o el lenguaje de programación utilizado. CORBA [132], Sun RMI [190], DCOM [48] y SOAP [179] son tecnologías que proporcionan tal funcionalidad y que por tanto pueden ser clasificadas como middleware de distribución. La arquitectura CORBA del OMG permite que los objetos inter-operen a lo largo de la red, sin importar el lenguaje de programación en el que hayan sido escritos o la plataforma en la que se ejecuten. RMI permite el desarrollo de aplicaciones Java-a-Java, proporcionando un modelo en el cual los métodos de los objetos pueden ser invocados desde otras máquinas virtuales remotas. DCOM permite la comunicación de componentes software sobre la red a través de la instanciación de componentes en plataformas Windows. Por último, SOAP permite el intercambio de información estructurada, haciendo uso del lenguaje XML, en la Web.

Muchos middlewares proveen, además de la de distribución, de una capa adicional denominada como *middleware de servicios comunes* que es la encargada de definir las abstracciones de uso más generalizado. La existencia de este tipo de servicios

permite que el programador se concentre en la lógica de negocio sin necesidad de que tenga que escribir el código necesario para realizar ciertas tareas comunes. Dentro de esta capa caben tecnologías tales como los CORBAServices [132], J2EE [168], el CORBA Component Model (CCM) [132] o los Servicios Web de .NET [146].

Por último, la última capa estaría formada por el *middleware específico de dominio* que resulta ser la capa que soporta aquella funcionalidad más dependiente de la aplicación. De forma contraria a las otras tres capas, que proveen mecanismos horizontales de integración, esta capa está relacionada con cuestiones más dependientes del dominio de aplicación. Este tipo de middleware es quizás el menos comprendido pues los dominios específicos de aplicación del middleware aún no han sido tan estudiados como los generales. Así, en el caso del OMG, las *Domain Tasks Forces* se concentran en la estandarización de servicios middleware específicos de domino, manejando dominios tan específicos como pueden ser el *Electronic Commerce Domain* o el *Life Science Research Domain*.

2.2.2. Clasificación

Atendiendo al tipo de comunicación establecida extremo a extremo entre el cliente y el servidor, y de forma totalmente independiente de la capa en la que se encuentre, se puede establecer la existencia de dos grandes familias de middlewares: los *síncronos* y los *asíncronos*. Los síncronos se caracterizan por que el cliente no puede continuar con su ejecución hasta que el servidor no finaliza con la suya. En el caso del modelo asíncrono se permite que el cliente retenga el control del proceso, desligando la ejecución del cliente de la del servidor.

A su vez, estos dos modelos generan dos grandes paradigmas de programación: los paradigmas basados en *invocación remota* y los paradigmas basados en *paso de mensajes*.

En la invocación remota se recrea la ilusión de que tanto el cliente como el servidor residen en el mismo entorno de ejecución, abstrayendo la transferencia de datos entre ambos mediante el uso de sustitutos y mediadores, que son generados por herramientas a partir de interfaces remotas. El primer tipo de sistema en aparecer fue el *Remote Procedure Call* **(RPC)** [119] que con la llega de la orientación a objetos se transformó en el paradigma que en la actualidad conocemos por *Remote Method Call* **(RMC)** [23].

Por otro lado, en el caso de paso de mensajes se deja bien claro la existencia de dos entidades, una cliente y otra servidora, que se comunican haciendo uso de un canal de comunicaciones que se emplea para el intercambio de mensajes entre ambas. Dependiendo del uso que se realice del canal de comunicaciones, se suelen diferenciar dos grandes paradigmas: el *Producer-Consumer Message Oriented Middleware* (PC-MOM) orientado a la comunicación punto a punto y el *Publisher-Subscriber Message Oriented Middleware* (PS-MOM) orientado a la comunicación multipunto [114].

Cada una de las dos familias de paradigmas tiene un dominio de aplicación propio, no pudiendo establecerse una condición de superioridad de un tipo de paradigma frente a otro, sin fijar antes un escenario.

2.2.3. El modelo de distribución RMI

Aunque existen numerosos modelos de distribución de propósito general, no todos tienen el mismo interés para esta tesis. Y así de un superconjunto formado por el metamodelo RMODP (Reference Model of Open Distributed Processing) [86], el DCE (Distributed Computing Environment) [57], el DCOM (Distributed Component Object Model) [34] de Microsoft, el CORBA (Common Object Request Broker Architecture) [164] del OMG, el DSA (Distribution Systems Annex) de Ada [96], el RMI (Remote Method Invocation) [167] de Sun y los Servicios Web [120] del W3C, esta sección se concentra en aquel más utilizado en esta tesis; esto es, RMI.

RMI ha sido concebido como modelo de distribución del modelo de objeto Java y su especificación, *Remote Method Invocation* (RMI) [167], ha sido desarrollada por Sun. Actualmente y bajo el control del *Java Community Process (JCP)* 66 hay un subperfil [94] que lo adecúa a los requisitos de los entornos dotados de pocos recursos, mientras que en la comunidad 50 [93] hay otro esfuerzo, aún en curso y que será estudiado más adelante, que intenta hacerlo adecuado a las necesidades de los sistemas de tiempo real.

La idea subyacente bajo el modelo de distribución de RMI es la de proporcionar un modelo de distribución para el modelo de objeto de Java, haciendo uso de una interfaz, `Remote`, que diferencia a aquellos objetos que pueden ser accedidos remotamente de aquellos que no. De todas las características del modelo de objeto Java, el modelo RMI provee una versión distribuida de los siguientes métodos: `equals`, `hashCode`, `toString`, `cloneable`, `finalize`, consiguiendo que tanto el cliente como el servidor compartan una semántica común. Para el mecanismo de sincronización basado en `synchronized`, `wait`, `notify` y `notifyAll` no se proporciona un modelo distribuido, presentando un comportamiento distinto en el cliente y otro en el servidor.

Además, adaptando parte de los mecanismos centralizados, se proporciona un conjunto de servicios generales y utilidades específicas. Existe un servicio de nombres, denominado en terminología inglesa *rmiregistry*; un servicio de activación[191], denominado *rmid*; un servicio de descarga dinámica de código distribuido; un servicio de recolección distribuida de basura [190] y otro de gestión de conexiones, transparente al programador. Salvo el de nombres, ninguno de los citados puede ser controlado directamente por el programador, sino que como mucho es parametrizable mediante atributos globales.

A nivel arquitectónico se caracterizan tres capas [190]. La primera capa, *stub/skeleton*, es la encargada de ocultar el envío y la recepción de datos entre el cliente y el servidor. La segunda, *remote reference*, es la encargada de soportar diferentes semánticas para los objetos remotos (e.g. multicast, unicast). La tercera, la de transporte, abstrae del manejo de bajo nivel del protocolo de red (e.g. TCP/IP) al programador.

El modelo de comunicación que presenta RMI es síncrono. Aunque existen investigaciones sobre los cambios necesarios para dar cabida a modelos asíncronos [105], éstos no han impactado suficientemente como para ser incorporados a la especificación.

Una de las mejoras hechas a la especificación ha sido la definición de un perfil reducido para entornos con capacidades limitadas, dentro de lo que se conoce en ter-

minología Java como J2ME [186]. Existe un paquete opcional, llamado RMIOP [94], que reduce la complejidad del actual RMI mediante la eliminación de características complejas tales como el subprotocolo multiplexado, la activación dinámica y la encapsulación del protocolo JRMP dentro de mensajes HTTP.

2.2.4. El modelo de predictibilidad de RTCORBA

De todos los modelos de distribución citados en en la sección previa, no todos han alcanzado el mismo grado de evolución dentro del área de los sistemas de tiempo real. DCOM no ha experimentado una gran evolución debido a que es un modelo en desuso, careciendo de extensiones para tiempo real. Tampoco existe una especificación de tiempo real para el modelo de distribución de ADA: el DSA, pero existen ciertos trabajos de la Universidad de Cantabria [38] analizando diferentes aspectos relativos a su soporte. De la misma manera, tampoco existe ningún tipo de especificación de tiempo real para los Servicios Web pero recientemente se han publicado algunos trabajos sobre la viabilidad de utilizar SOAP en el desarrollo de algunos sistemas embebidos de tiempo real [73] [74]. En el mundo Java tampoco existe ningún tipo de especificación RTRMI, pero tal y como veremos más en detalle en la sección 2.4, existen esfuerzos encaminados a su consecución. Y por último, está la tecnología CORBA [132],quizás una de las más estudiadas, contando incluso con extensiones (RTCORBA) de tiempo real.

La especificación RTCORBA [58] [151] [103] aparece dividida por motivos históricos en dos: una para planificación estática [151] llamada RTCORBA 1.0 y otra para planificación dinámica [103] denominada RTCORBA 2.0. En la actualidad ambas aparecen descritas en un documento único [133].

RTCORBA 1.0: Planificación estática basada en prioridades

La especificación RTCORBA 1.0 define una serie de características estándar que permiten obtener predictibilidad extremo a extremo en aplicaciones CORBA haciendo uso de un esquema de prioridades fijas. Se identifican los recursos del sistema que deben de ser gestionados para poder conseguir que el sistema responda de forma predecible, definiendo interfaces que permiten parametrizar la gestión realizada en cada una de las capas del modelo.

- *Capa de comunicaciones.* A este nivel, se distinguen dos tipos de mecanismos, los que permiten la gestión de conexiones y los que permiten aprovechar las posibilidades que algunos protocolos de comunicaciones (e.g. los circuitos virtuales de ATM [61]) ofrecen a la hora de establecer las conexiones.

- *Capa del sistema operativo.* A este nivel, se identifican los mecanismos de planificación y sincronización soportados por los principales sistemas operativos de tiempo real (eg. IEEE POSIX 1003 [41]).

- *Capa CORBA.* A este nivel, se proveen interfaces que permiten que el programador pueda especificar los requisitos de sus aplicaciones. Mediante el uso

de interfaces especiales, se puede configurar y controlar hasta tres tipos de recursos:

- **Recursos de procesamiento** a través del *threadpool*, los *mecanismos de prioridad*, el sistema de control basado en cerrojos[2] y el servicio de *planificación global*.
- **Recursos de comunicación** a través del uso de *vinculaciones explícitas*[3], de *bandas de prioridades* y de las *conexiones privadas*.
- **Recursos de memoria** a través de los *threadpools*.

Analicemos cada uno de estos mecanismos más en detalle.

- **Mecanismo de prioridades** RTCORBA establece un mecanismo de prioridades propias y mecanismos de declaración, de propagación y de transformación de prioridades. En la especificación, se identifican dos tipos de prioridades: *nativas* y *globales*. Las prioridades nativas son aquellas que maneja el sistema operativo y las globales son un conjunto de 32768 prioridades utilizadas por el programador CORBA. En cada ORB se realizan correspondencias entre las prioridades nativas y las globales.

  Se consideran dos modelos para la elección de la prioridad a la cual se ejecutará el servidor: *prioridades declaradas* por el servidor y *prioridades propagadas* por el cliente. En el modelo declarado, es el objeto remoto el que define la prioridad a la cual son ejecutados todos sus métodos, y en el propagado, la prioridad RTCORBA a la que ejecuta el servidor es tomada del cliente.

  Con el objetivo de proveer un modelo más flexible, RTCORBA, soporta *transformadores de prioridad*[4]. Éstos permiten la definición de esquemas de prioridades más complejos que los proporcionados por las prioridades declaradas y las propagadas.

- **Threadpool** A fin de controlar el modelo de concurrencia utilizado en el servidor, se define una nueva entidad denominada *threadpool*. Cada POA tendrá asignado un *threadpool* que será el que proporcione los hilos utilizados para invocar al objeto remoto. En RTCORBA se define un modelo de *threadpool* particular, distinguiéndose dos tipos de casos. Uno *sin carriles* orientado a la obtención de una alta eficiencia en la utilización de recursos y otro *con carriles* más enfocado a la obtención de una alta predictibilidad.

- **Cerrojos** El mecanismo de cerrojos de RTCORBA permite la definición de cerrojos que son gestionados mediante los algoritmos de techo o de herencia de prioridad. Estos cerrojos operan sobre cada uno de los ORBs. No está soportada la sincronización distribuida.

[2] En el especificación denominado mutex
[3] En la especificación explicit bindings
[4] En la especificación aparece descrito como *priority transforms*.

- **Gestión de comunicaciones inter-ORB** Tanto en el cliente como en el servidor se definen ciertas interfaces que permiten configurar las propiedades del protocolo de transporte utilizado en la comunicación extremo a extremo. Estos mecanismos permiten elegir el tipo de conexiones que serán utilizadas para conectar al cliente con el servidor y las características que éstas tendrán. Adicionalmente, se define un mecanismo de *vinculaciones explícitas*, las *bandas de prioridades* y las *conexiones privadas*. Las vinculaciones explícitas permiten reservar todos los recursos necesarios para realizar la invocación remota, reduciendo así la inversión de prioridad típicamente experimentada durante la primera invocación remota. Las bandas de prioridades permiten dividir el tráfico de datos entre el cliente y el servidor entre varias conexiones en función de la prioridad de ejecución del cliente. Y las conexiones privadas permiten obtener una máxima predictibilidad mediante la no compartición del canal de comunicaciones.

- **Servicio de planificación** Este servicio permite definir los requisitos de las aplicaciones de tiempo real, de una forma más intuitiva, utilizando parámetros como son el tiempo de ejecución en el peor de los casos, en vez de utilizar otros de más bajo nivel, tales como las prioridades del sistema operativo. Este servicio se encarga de establecer estas relaciones de forma más sencilla. En la actualidad, con la definición del servicio de planificación dinámica de RTCORBA 2.0, este servicio está obsoleto.

RTCORBA 2.0: Planificación dinámica

RTCORBA 2.0, también conocido como DSRTCORBA [103], define un marco mucho más general para la gestión del procesador que el definido por RTCORBA 1.0. Dos son las principales novedades que incorpora a RTCORBA 1.0: el hilo distribuido y un servicio de planificación dinámica.

- **Hilo distribuido** RTCORBA 2.0 introduce el concepto de hilo distribuido de tiempo real tomándolo del kernel alfa [127]. Este mecanismo permite que el programador pueda modificar tanto el estado de ejecución de un hilo distribuido arrancándolo, parándolo o abortándolo, como sus parámetros de planificación en tiempo de ejecución. El middleware transmite los cambios realizados localmente a los nodos remotos de la red donde se esté ejecutando el hilo distribuido de forma transparente al programador.

- **Servicio de planificación dinámico** El servicio de planificación de RTCORBA 1.0 presenta dificultades a la hora de adaptarse a los requisitos de algunas de las aplicaciones de tiempo real. Así que se ha propuesto un nuevo servicio de planificación donde el programador puede utilizar sus propios algoritmos de planificación que cuenta con soporte para los siguientes algoritmos: FPS, EDF, LLF y MAU.

Líneas de investigación

Por último, unido a RTCORBA existe una fuerte tradición de grandes grupos de trabajo. Uno de los de referencia es el DOC de la Universidad de Washington, el cual ha desarrollado un ORB de código libre denominado TAO. Este ORB ofrece soporte tanto a la comunicación síncrona [70] como a la asíncrona [8] basada en paso de mensajes, a una serie de patrones de programación ([1] [2] [142]) que mejoran la predictibilidad de la invocación remota, así como a servicios especiales como son el de eventos de tiempo real [134] o el de planificación [68].

La Universidad de Rhode Island también ha contribuido a RTCORBA proponiendo servicios de tiempo real ([189] [177] [62] [189]) así como con una propuesta de encapsulación de parámetros de tiempo real denominada *time distributed method invocations* (TDMI) [177], centrándose últimamente en el soporte a la planificación dinámica basada en el paradigma de hilo distribuido [35].

Otro de los esfuerzos importantes en el desarrollo de RTCORBA ha sido el realizado por el grupo del MITRE emplazado en Bedford que ha caracterizado a RTCORBA desde el punto vista de los sistemas de control [171] [43].

Por último, el proyecto ROFES intenta mejorar la especificación mediante la incorporación de modelos de comunicación no soportados por ésta como son el *time-triggered ethernet* [107], los buses CAN [106] o las interfaces SCI [148], de uso bastante común en muchos sistemas embebidos.

2.2.5. Resumen y conclusiones

En su conjunto, el middleware de comunicaciones se nos presenta como una maraña de tecnologías enfocadas a la mitigación de las complejidades asociadas al desarrollo de las aplicaciones distribuidas, con diferentes capas que progresivamente van enmascarando una importante parte de los problemas asociados a la distribución.

De ellas, no todas tienen la misma importancia a la hora de desarrollar Java de tiempo real distribuido. Las más relevantes son las que se han caracterizado como *middleware de distribución* y complementando a ésta, aunque en menor grado, tanto el *middleware de infraestructura* como el *middleware de servicios comunes*.

Por último, tanto los mecanismos de comunicación síncronos como los asíncronos son interesantes para el middleware de distribución para Java de tiempo real. Y por tanto Java de tiempo real distribuido no debería de restringirse al empleo exclusivo de mecanismos síncronos o asíncronos, debiendo de dar cabida a ambos tipos de mecanismos.

2.3. Middleware de infraestructura para Java de tiempo real

Hasta el momento, en este estado del arte se ha analizado el middleware y los sistemas de tiempo real llegándose a la conclusión de que las capas más interesantes para la construcción de Java de tiempo real distribuido eran dos: la de infraestructura y la de distribución. En esta sección se procederá a analizar el estado del arte

relacionado con el middleware de infraestructura de tiempo real Java, dejando para la siguiente las cuestiones relacionadas con la distribución.

Así, con el fin de conocer las principales tecnologías Java de tiempo real para sistemas centralizados, esta sección presenta, analiza y compara las principales propuestas existentes que están basadas en la extensión de la jerarquía de clases de Java. Comienza la sección 2.3.1, presentando diferentes limitaciones presentes en el modelo tradicional de máquina virtual Java que dificultan el desarrollo de sistemas de tiempo real centralizados. Le sigue la sección 2.3.2 presentando los requisitos NIST para Java de tiempo real, que dan paso a las dos principales especificaciones Java de tiempo real existentes en la actualidad: RTCORE (sección 2.3.3) y RTSJ (sección 2.3.4). Después, en la sección 2.3.5, se presentan otras aproximaciones menores. Y por último, el estado del arte relativo a las tecnologías Java de tiempo real finaliza, en la sección 2.3.6, realizando una pequeña comparativa entre las diferentes aproximaciones presentadas así como, en la sección 2.3.7, con unas conclusiones generales.

2.3.1. Limitaciones de Java para los sistemas embebidos y de tiempo real

El hecho de no haber sido concebido como lenguaje de tiempo real, sino más bien como lenguaje orientado hacia la programación de propósito general, hace que Java no presente, o a veces lo haga de forma muy vaga, control sobre la gestión realizada sobre los recursos internos, no siendo de aplicación directa las técnicas de utilizadas en las aplicaciones de tiempo real descritas anteriormente en la sección 2.1. A estas carencias generales hemos de sumar otras más específicas derivadas de la existencia de otros mecanismos, incorporados por el modelo computacional de Java, como son la carga y la descarga de clases, su inicialización o la propia gestión automática de memoria que se convierten en nuevas fuentes específicas de impredictibilidad e indeterminismo.

- **Planificación** Una de las debilidades de Java, típica de los lenguajes de propósito general, para con el tiempo real es su modelo de planificación. La especificación de la máquina virtual [111] define un modelo de procesamiento basado en prioridades pero, en ella, no se llega a concretar si el modelo es expulsivo o no, recayendo dicha decisión en el implementador de la máquina virtual. Esta falta de concreción le permite adaptarse a una gran variedad de sistemas operativos, pero también impide el uso de técnicas basadas en prioridades expulsivas tan básicas como las basadas en prioridades fijas. Tal y como veremos en el transcurso de esta sección, lo que las diferentes aproximaciones existentes en la actualidad han hecho para solventarlo es clarificar y a veces extender el modelo de planificación de Java.

- **Protocolos de sincronización de tiempo real** Otra de las características clave de Java, el soporte de mecanismos de sincronización dentro del propio lenguaje, presenta una definición demasiado vaga para los sistemas de tiempo real. La palabra reservada `synchronized` [88] permite limitar la concurrencia estableciendo zonas de código de acceso en exclusión mutua. Al igual que

sucede en el modelo de planificación, su definición ha sido relajada deliberadamente para facilitar la implementación de la máquina virtual. En Java no se especifica qué hilo, de los múltiples que pueden estar esperando por el acceso al cerrojo, es el que accede finalmente a él, siendo compatibles con esta definición comportamientos FIFO, LIFO o cualquier otra disciplina de gestión de procesador conservativa. Tal y como se verá, esto se ha solventado mediante el uso de protocolos de sincronización de tiempo real -PIP y PCP- y se ha visto complementado, en algunos casos, con la inclusión de otros mecanismos de sincronización clásicos como son el semáforo, el mútex o la cola de mensajes no bloqueante.

- **Gestión automática de memoria** Otra de las características de Java que obstaculiza el desarrollo de aplicaciones de tiempo real es la gestión automática de memoria. La gestión automática de memoria facilita la programación, evitando los problemas asociados a las fugas de memoria mediante el uso de algoritmos de recolección de basura. Esto constituye una ventaja para el programador, pues se reducen notablemente los tiempos de desarrollo, pero supone un inconveniente para las aplicaciones de tiempo real pues los algoritmos que ejecutan los recolectores se convierten en nuevas fuentes de inversión de prioridad. A la hora de abordarla, no existe una solución única sino que se han utilizado tanto técnicas basadas en recolección de basura de tiempo real ([13] [163]) como modelos intermedios basados en regiones ([78] [28] [26]), en el almacenamiento de objetos en pila ([3] [66]) o la gestión de memoria delegada en el programador ([87] [54]).

- **Carga dinámica de clases** Otra de las ventajas de Java para sistemas embebidos, la carga dinámica de clases, presenta serios inconvenientes a la hora de ser utilizada en el dominio del tiempo real. Este mecanismo posibilita la carga y la descarga de forma dinámica de las clases que no son utilizadas durante la ejecución de un programa suponiéndonos grandes ventajas para sistemas embebidos donde la memoria física disponible es un bien escaso y caro. Sin embargo, desde el punto de vista del tiempo real, esto puede suponer un serio problema pues el proceso de carga y de descarga de código puede interferir con la ejecución de las diferentes tareas de tiempo real, provocando pérdidas de plazos. Esto es debido a que el tiempo que normalmente tarda la máquina virtual en cargar una clase no suele ser despreciable frente a los plazos de las tareas de tiempo real. Por lo general, lo que han hecho la mayor parte de los investigadores es evitar este mecanismo proponiendo que no sea utilizado durante fases críticas de la aplicación.

- **Inicialización de clases** En la misma línea, la inicialización de clases es también problemática. La especificación de la máquina virtual [111] obliga a que se ejecute el código de una clase, ésta se encuentre inicializada, implicando entre otras muchas tareas la ejecución de los constructores estáticos de las clases Java. Esto, al igual que sucede con la carga y la descarga de clases, puede interferir con la ejecución de una tarea de tiempo real provocando la pérdida

de plazos. Para eliminar dicha interferencia, al igual que ocurre en la carga dinámica de clases, lo que generalmente se ha propuesto es exigir que la totalidad de las clases necesarias para el desarrollo de una aplicación de tiempo real esté disponible cuando comienza la ejecución de dicha tarea de tiempo real.

- **Manejo de eventos** El manejo de eventos en Java es también bastante limitado. Generalmente, los lenguajes de tiempo real incluyen mecanismos que permiten el manejo de eventos asíncronos capaces de romper la ejecución síncrona de las diferentes aplicaciones. Estos eventos aparecen asociados bien a eventos externos, generados por ejemplo por interrupciones hardware, o a internos, generados por otros hilos residentes en la misma máquina virtual. El modelo Java proporciona mediante el modelo de eventos del AWT [60], las excepciones y algunos métodos de la clase `Thread` (como por ejemplo `Thread.stop()`), un cierto soporte a dicho tipo de funcionalidad. Sin embargo, dichos mecanismos resultan insuficientes a la hora de construir ciertas aplicaciones de tiempo real. Motivo por el cual, muchas aproximaciones a Java de tiempo real han incluido clases especiales que le proporcionan soporte específico.

- **Acceso a hardware** También es común que los sistemas embebidos necesiten acceder a recursos de bajo nivel mediante, por ejemplo, el uso de puertos específicos o el acceso a posiciones fijas de memoria, para interacturar con sistemas físicos. En Java dicha interacción no está directamente soportada y para realizarla se suele recurrir a JNI (*Java Native Interface*) [109]. Este mecanismo sirve de enlace entre las aplicaciones Java y otros lenguajes -típicamente C/C++- y nos permite realizar mediante rutinas de bajo nivel el acceso al hardware subyacente. Aunque resulta de utilidad, el hecho de que se tenga que recurrir a otro lenguaje en el desarrollo de aplicaciones embebidas, ha sido suficiente para que muchas aproximaciones definan nuevas jerarquías de clases que posibilitan el acceso directo desde el lenguaje Java.

Desde el punto de vista práctico los puntos que hemos expuesto sintetizan en gran medida la problemática encerrada en Java de tiempo real y cualquier solución que se pretenda diseñar en dicho entorno debería de abordar las limitaciones anteriormente presentadas.

2.3.2. Requisitos NIST para Java de tiempo real

Motivados por las ventajas económicas que el uso de Java para el desarrollo de sistemas embebidos podía acarrear y también conscientes de sus grandes limitaciones, en la década de los noventa se iniciaron una serie de esfuerzos encaminados a la obtención de soluciones operativas. Uno de los hitos más importantes se produce en 1999, cuando se realiza un *workshop* soportado por el NIST, en el que participaron tanto fabricantes como desarrolladores de sistemas de tiempo real, con el objetivo de definir cuáles serían los requisitos para Java de tiempo real. Los resultados fueron recopilados en un documento [112] y han sido utilizados en varias de las especificaciones.

Tomando los requisitos como punto de partida se desarrollaron dos alternativas. Una de ellas, la del JConsortium, liderada por HP y NewMonics, trabajó en la especificación RTCORE. Y otro grupo denominado *Real-Time for Java Experts Group* (RTJEG), bajo el amparo de Sun Microsystems, gestó RTSJ. Por último los requisitos, pensados inicialmente para sistemas Java, han transcendido a las tecnologías Java y han sido aplicados al lenguaje .NET con el objetivo de determinar la viabilidad tecnológica de un hipotético RT.NET [192].

2.3.3. Real Time CORE Extensions

The Real-Time CORE Extension (RTCORE) [3] es una especificación Java de tiempo real desarrollada por el JConsortium. En su definición aparecen dos entornos de ejecución: el *core* que permite el desarrollo de aplicaciones de tiempo real y el tradicional de Java, denominado en la especificación como *baseline*. Para la comunicación entre ambos entornos cada objeto *core* tiene dos interfaces de aplicación, una para el domino *core* y otra para el *baseline*. La sincronización entre ambos entornos se realiza mediante semáforos.

Características

- **Memoria** En el *core* se define una jerarquía de objetos que tiene el `CoreObject` como raíz. Para sobrecargar los objetos finales de la clase `Object` se han realizado cambios en la semántica del cargador de clases, reemplazando estos métodos con métodos especiales del `CoreObject`. Un `CoreObject` puede ser creado en dos bloques de memoria: *la pila del hilo en ejecución* o en un *contexto de creación*. La creación en la pila se consigue definiendo el objeto como `stackable`. El contexto de creación es un tipo de región donde los objetos pueden ser liberados de forma totalmente explícita por el programador. La principal característica común a todos los objetos creados en el *core* es que, de forma contraria a los creados en el *baseline*, no sufren las impredictibilidades del recolector de basura.

- **Tareas y asincronismo** Las tareas del *core* son análogas a los hilos Java tradicionales (`java.lang.Thread`). Todas las tareas de tiempo real deben de extender `CoreTask` o una de sus subclases. Las tareas son planificadas siguiendo un modelo expulsivo basado en prioridades (128 niveles) con orden FIFO dentro de cada prioridad. Una `CoreTask` dispone de la posibilidad de utilizar el método `stop()` para la realización de una transferencia asíncrona de control. Existe un tipo especial de tareas, `SporadicTask`, que permiten implementar tareas esporádicas. No existe ningún tipo de clase que implemente tareas de tiempo real periódicas y el programador ha de recurrir al empleo de los métodos `sleep()` y `sleepUntil()` de la clase `CoreTask` para conseguir periodicidad.

- **Excepciones** La jerarquía de clases utilizada para tratar las excepciones en el *core* no es la misma que en el *baseline*. En el *core* la jerarquía de excepciones Java, `java.lang.Throwable`, es reemplazada por el cargador de clases, de forma transparente, por otra equivalente.

- **Sincronización** El *core* restringe el modelo de sincronización de Java. En él, la sincronización mediante `synchronized` sólo puede ser realizada sobre el objeto actual (`this`). A fin de paliar esta limitación se soportan nuevos mecanismos de sincronización como pueden ser los semáforos y el mútex. En los monitores y los cerrojos las tareas entrantes son ordenadas según prioridad y orden de llegada. La inversión de prioridad se evita mediante el uso de un protocolo de tipo PCP.

- **Acceso a Hardware** El reloj del sistema se puede acceder con una precisión de hasta 64 bits con una resolución de nanosegundos. Además, la clase `Time` provee conversiones de tipos. Los diferentes puertos del hardware pueden ser accedidos a través de la clase `IOPort`.

Actualmente, su principal limitación es de índole práctica. Y aunque en la literatura aparece alguna propuesta de implementación, como por ejemplo [69], lo cierto es que no existe ningún tipo de máquina virtual que soporte la especificación.

2.3.4. The Real Time Specification for Java

The Real Time Specification for Java (RTSJ) [4] define una alternativa basada en la modificación de la máquina virtual Java, soportando tanto aplicaciones de tiempo real como de propósito general. Esta especificación se desarrolló bajo el amparo del modelo de comunidades Java, siéndole asignado al grupo JCP-1 [91]. En la actualidad la especificación se encuentra en un proceso de refinamiento, siendo el grupo JCP-282 [92] el encargado de tal tarea.

Sus principios de diseño son los siguientes:

- No restringir el entorno de ejecución Java.
- Mantener la compatibilidad hacia atrás.
- No incluir ningún tipo de extensión sintáctica en el lenguaje Java.
- Ejecución predecible.
- Dar cobertura a las necesidades de los sistemas de tiempo real actuales.
- Permitir a futuras implementaciones la incorporación de características avanzadas.

Características

- **Hilos y planificación** El comportamiento del planificador es clarificado, requeriéndose un mínimo de 28 prioridades expulsivas, siendo las tareas en cada prioridad ordenadas siguiendo una disciplina FIFO. Aunque se impone como requisito el tener prioridades expulsivas, el modelo es abierto, pudiendo potencialmente soportar otros planificadores (e.g. EDF o LLF). Sus nuevas clases posibilitan dos modelos de programación: la basada en hilos y la basada en

eventos. La programación basada en hilos la soportan las clases `RealtimeThread` y `NoHeapRealtimeThread`, siendo la principal diferencia entre ambas su grado de dependencia para con el recolector de basura. Los mecanismos de asincronismo están soportados por la clase `AsyncEventHandler` y todas sus subclases. Los hilos de tiempo real pueden soportar tanto un comportamiento periódico como uno aperiódico.

- **Memoria** Dado que el recolector de basura es difícil de predecir, RTSJ refina más el modelo de programación de memoria de Java definiendo el concepto de *memoryarea*. Un *memoryarea* es un espacio de almacenamiento que está sujeto a una disciplina de gestión automática de memoria particular, donde el programador puede crear objetos. En RTSJ hay tres tipos básicos de *memoryarea*: la `HeapMemory`, cuyos objetos son eliminados por un recolector de basura; la `ImmortalMemory`, cuyos objetos nunca son recolectados; y la `ScopedMemory` que permite la recolección de objetos haciendo uso de un mecanismo de regiones.

  RTSJ incluye además una serie de interfaces que permiten que el programador seleccione en qué tipo de memoria son creados los objetos Java. Esta elección se realiza de forma indirecta mediante el uso de una estructura propia de cada hilo, denominada *scopestack*, y una serie de métodos auxiliares, `executeInArea()` y `enter()`, que interactúan con dicha estructura. Los objetos son creados en la región que se encuentra en la cima del *scopestack* y los métodos auxiliares permiten tanto la creación como la modificación de la cima, apilando y desapilando *memoryareas*.

  Siguiendo el modelo de referencias seguras de Java, en RTSJ se evita la posibilidad de que existan referencias inseguras mediante la inclusión de dos reglas: la regla del padre único (*single parent rule*) y la regla de asignación (*assignment rule*). La regla del padre único evita que aparezcan ciclos en el modelo de regiones y la regla de asignación impide el establecimiento de referencias que potencialmente son inseguras. Tal y como se esquematiza en la tabla 2.1, un objeto almacenado en `ScopedMemory` sólo puede ser referenciado desde variables locales (las que residen en la pila), desde la misma instancia de `ScopedMemory` donde reside el objeto referenciado o en una region que ocupe una posición más interna en el *scopestack*, no pudiendo realizarla si el objeto se encuentra almacenado en la memoria inmortal o en el montículo Java.

Desde:\ a:	`Heap`	`Immortal`	`Scoped`
`Heap`	√	√	NO
`Immortal`	√	√	NO
`Scoped`	√	√	Sólo si es la misma o es externa
`Local`	√	√	√

Cuadro 2.1: Las reglas de asignación de RTSJ

- **Sincronización** En RTSJ se definen dos tipos de mecanismos de sincronización: una versión mejorada del `synchronized` y las colas de mensajes no blo-

queantes. La versión mejora del `synchronized` utiliza por defecto un protocolo de herencia de prioridad, posibilitando también el uso de techo de prioridad. Las colas de mensajes posibilitan la comunicación entre hilos `RealtimeThread` y `NoHeapRealtimeThread`, evitando la propagación de la inversión de prioridad del recolector de basura entre ambos.

- **Acceso a hardware** Se incluyen clases que permiten manejar el tiempo con una precisión de nanosegundos. Un nuevo tipo de clase, `RationaleTime` puede ser utilizada para describir frecuencias dentro de períodos. La clase `RawMemory` permite el acceso a posiciones físicas de memoria, posibilitando el acceso a puertos. Además, las clases de tipo `AsyncEvent` permiten procesar señales externas posix.

Implementaciones

En el momento de escribir estas líneas, las implementaciones de RTSJ son aún poco comunes y la mayor parte de ellas están aún en proceso de desarrollo o bien proporcionan un soporte parcial a la especificación. Aún así, existen implementaciones tanto comerciales como de código libre y abierto.

La más popular, desarrollada por la empresa TIMESYS y descargable desde la página web de la empresa, es jTime [173], también a veces denominada como RTSJ-RI por ser desarrollada para la validación de la especificación. De código libre, jRate [47] [45] soporta parcialmente la especificación y puede ser descarga desde *sourceforge* [44]. OVM [135] [136] [140] es al igual que jRate de código abierto y puede ser descargada desde la página oficial del proyecto [56]. En el MIT se ha extendido el compilador Flex [32] [33] con extensiones para Java de tiempo real descargables desde la página del proyecto [147]. En Europa, la empresa AICAS tiene un producto propio, Jamaica [163], que puede ser descargado libremente desde su página web [162]. Desde primavera del 2005 Sun Microsystems ofrece Mackinack [7] [24], una plataforma de desarrollo para Java de tiempo real disponible para entornos Sparc®y plataformas Solaris™. Desde la primavera del 2006 Apogee ofrece la máquina virtual Aphelion [12], basada en la máquina virtual J9. La última implementación, disponible desde el verano del 2006, es la de IBM que ofrece un producto denominado WebSphere Real Time [174]

Quizás, de todas las máquinas virtuales, la que ofrece un mayor atractivo de cara a la hora de ser tomada como punto de partida a la hora de realizar prototipos es jTime. A su favor pesan el hecho de que es una de las pocas implementaciones que soporta la totalidad de la especificación RTSJ y también que su código fuente, muy previsible en un futuro no muy lejano, se podrá consultar y utilizar con fines investigadores. Su mayor inconveniente es quizás es su bajo rendimiento [27], lo que introduce un cierto pesimismo en los experimentos realizados con ella.

Dialectos orientados hacia la alta integridad

Pese a tratarse de una especificación de tiempo real, algunas veces, ciertas características específicas de RTSJ pueden resultar demasiado complejas de predecir.

Esto ha hecho que la comunidad investigadora, especialmente aquella que se dedicada al desarrollo de sistemas de alta integridad -sistemas de frenado, antibloqueo para automóviles, apagado automático de centrales nucleares, intercambiadores ferroviarios asistidos por computadora-, se encuentre trabajando en versiones simplificadas y extendidas de la especificación capaces de ofrecer unas garantías extremas de predictibilidad.

Ravenscar-Java [83] constituye un entorno restringido de RTSJ que trata de extender las ventajas del modelo Ravenscar Ada [36] a RTSJ. De forma similar a ravenscar-Java, Expresso [65] define un perfil para aplicaciones de alta integridad de tiempo real que añade a las dos fases del ravenscar-Java -inicialización y misión- una de finalización. Basándose en las restricciones que presentan los sistemas embebidos, Martin Schoerl ha diseñado una máquina virtual para sistemas embebidos de tiempo real [152] [154] [153] de pequeño tamaño llamada JOP. Y por último la empresa Aonix ha lanzado una especificación denominada *Scalable Java Development of Real-Time Systems* (SJDRTS) [123] en la que se proponen interfaces para los sistemas de seguridad crítica, así como una serie de librerías de propósito general, alineadas filosóficamente con el modelo RTCORE, recuperando así parte de las ideas de este modelo dentro del contexto de RTSJ.

Proyectos que utilizan RTSJ

En la actualidad, existen una serie de proyectos de mayor o menor calado que se dedican a la migración de aplicaciones de tiempo real, previamente realizadas en otros lenguajes de programación, a Java de tiempo real. En la mayor parte de los casos, partiendo de un modelo computacional ya creado, se comprueba si éste puede ser soportado o no por las diferentes tecnologías Java de tiempo real existentes en la actualidad.

Así, científicos de Sun Labs y del NASA/JPL (Jet Propulsion Laboratory) están trabajando de forma conjunta para implementar la arquitectura *Mission Data System* (MDS) del JPL empleando RTSJ como lenguaje de desarrollo [54] [42]. Y dentro del proyecto AOCS [137] se contempla también la utilización de Java de tiempo real. Y por último cabe citar al programa *Real-Time Java for Embedded Systems* (RTJES) que se ha encargado de portar la arquitectura BoldStroke de Boeing [158] a Java de tiempo real, obteniendo resultados [159] que apuntan a que la máquina virtual jTime es capaz de satisfacer adecuadamente los requisitos impuestos por BoldStroke.

Se podría decir que los diferentes proyectos desarrollados alrededor de RTSJ muestran que esta tecnología es capaz de satisfacer adecuadamente los requisitos de algunas de las aplicaciones de tiempo real existentes en la actualidad. La existencia de aplicaciones de complejidad relativamente alta, tales como MDS o BoldStroke, así parece ponerlo de manifiesto.

Líneas de investigación

RTSJ es aún una tecnología bastante joven y aún no ha alcanzado un elevado grado de madurez, existiendo en la actualidad varias líneas de investigación abiertas que tratan de solventar sus principales limitaciones tecnológicas. Analizando lo que

son los diferentes artículos publicados en revistas y sobre todo en conferencias especializadas, y tratando de proceder a su clasificación, se ha visto que algunos de los temas en los que se está trabajando son los siguientes: (1) la eficiencia de ejecución ([79] [44] [147]); (2) la programación con `ScopedMemory` ([46] [144] [22] [53] [140] [19] [54] [26] [87][20] [26]); (2) la validación eficiente de las reglas de asignación y del padre único ([21] [33] [45] [76]); (3) extensiones al modelo de referencias de RTSJ ([31] [172]); (4) extensiones al modelo de regiones ([47] [16] [172]); (5) la revisión del modelo de eventos [183]; y (6) algoritmos de planificación para Java de tiempo real ([37] [59] [182] [115]).

De todas ellas, las más novedosas son aquellas relacionadas con la gestión automática de memoria, en especial aquellas relacionadas con el modelo de regiones y de referencias de RTSJ.

Ya para finalizar el estado del arte relacionado con RTSJ cabría reflexionar, al igual que se hizo anteriormente con RTCORE, sobre el estado actual de esta tecnología. Aunque a día de hoy se puede decir que el grado de evolución de RTSJ es mayor que el de RTCORE, RTSJ aún no ha alcanzado un grado de madurez óptimo. Y pese a que existen numerosas implementaciones, a día de hoy, las herramientas de desarrollo y de depuración de uso más común aún no han sido convenientemente adaptadas para facilitar el trabajo con características tan específicas como la `ScopedMemory`.

2.3.5. Otras aproximaciones

Al margen de las soluciones basadas en los requisitos NIST, existen otras, basadas al igual que RTSJ y RTCORE en extender las interfaces Java, que también permiten el desarrollo de aplicaciones Java de tiempo real. A continuación las presentaremos brevemente, sin entrar en grandes detalles.

- **Portable Executive for Reliable Control (PERC)** Esta solución ha sido desarrollada por la empresa NewMonics ([125] [121] [124] [122]) y define dos paquetes: `Real-Time` y `Embedded`. El paquete `Real-Time` provee abstracciones para sistemas de tiempo real, mientras el `Embedded` provee abstracciones de bajo nivel para acceder al hardware subyacente. Aunque el modelo permite que cada usuario defina su propio planificador; por defecto, se soporta un planificador basado en prioridades gestionadas mediante un algoritmo de reparto de procesador de tipo *round-robin*. No se soporta ningún tipo de protocolo de herencia de prioridad para resolver el problema de la inversión de prioridad. Por último, en el modelo se soporta un recolector de basura de tiempo real.

- **Real time Java Threads (RTJThreads)** Ésta es una solución bastante simple que consta únicamente de tres clases [117]: `RtThread`, `RtHandler` y `Time`. El modelo de planificación soportado está basado en prioridades y se soportan protocolos de inversión de prioridad para permitir la compartición de datos entre las diferentes aplicaciones. A fin de evitar la inversión de prioridad introducida por el recolector de basura se le asigna una prioridad inferior a la de todas las tareas de tiempo real.

- **Communicating Java Threads (CJThreads)** Una solución totalmente diferente a todas las anteriores son las extensiones CTJ [80]. Éstas están basadas en el modelo CSP propuesto por Hoore [82]. El acceso al hardware subyacente se realiza mediante el empleo de memoria compartida. El reparto del procesador se realiza mediante un modelo basado en prioridades. La sincronización se realiza mediante canales que se gestionan de forma no expulsiva. No se utilizan pues protocolos para controlar la inversión de prioridad. Tampoco se trata el problema de la recolección de basura.

2.3.6. Comparación

Para analizar las soluciones Java de tiempo real que se han ido presentando, es posible utilizar múltiples criterios de comparación. Así, algunos autores [77] han tomado como criterio la diferente cobertura que cada una de las soluciones da a problemas concretos como son la planificación, la sincronización, el acceso al hardware, el soporte dado a eventos asíncronos o la posibilidad de realizar negociaciones. Otros [25], han tomado como criterio cómo las diferentes soluciones cubren los requisitos NIST. Y por último, también se han utilizado, en [126], ciertos criterios provenientes de la ingeniería del software.

En el presente caso no se ha querido compararlos en ninguno de estos términos, sino que se ha querido evaluar el grado de completitud de las soluciones. Para ello, en este estado del arte, se ha optado por analizar el grado de cobertura que cada una de las aproximaciones ofrece a cada una de las limitaciones presentadas en la sección 2.3.1. Con los resultados de esta evaluación se ha construido la tabla 2.2, donde para cada una de las soluciones se ha asignado el signo $\surd$, en caso de que se aborde una limitación de Java, o un signo -, en el caso contrario. El signo †, que aparece en RTSJ, indica que el problema es abordado por sus dialectos.

	Planificación	Sincronización	Recolección de basura	Carga dinámica de clases	Inicialización de clases	Manejo de eventos	Acceso hardware
RTSJ	$\surd$	$\surd$	$\surd$	$\surd$†	$\surd$†	$\surd$	$\surd$
RTCore	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
PERC	$\surd$	-	$\surd$	-	-	$\surd$	$\surd$
RTJThreads	$\surd$	$\surd$	$\surd$	-	-	-	-
CJThreads	$\surd$	-	-	-	-	-	$\surd$

Cuadro 2.2: Comparación entre las diferentes soluciones Java de tiempo real centralizado

El primer resultado que salta a la vista es que se puede considerar que tanto RTSJ como RTCORE son las especificaciones más completas, ya que son las que más

limitaciones abordan. PERC es la segunda que más aborda, pero su soporte carece de algoritmos que eviten la inversión de prioridad, permitiendo caracterizarla como más incompleta. Finalmente, el soporte proporcionado por los RTJThreads y los CJThreads puede ser catalogado como el más pobre pues el número de limitaciones abordado es mucho menor.

Otro resultado interesante es que todas las soluciones soportan planificación basada en prioridades y algunas de ellas -RTCORE, RTSJ y los RTJThreads- incluso soportan planificación dinámica. Esto muestra que el grado de comprensión de este tipo de limitación es muy alto y que además existe un cierto consenso generalizado sobre la necesidad de incorporar técnicas que solventen este problema.

El uso de mecanismos de sincronización de tiempo real es también bastante generalizado. Salvo PERC y los CJThreads, el resto de las soluciones estudiadas soporta herencia de prioridad o techo de prioridad. En el caso de PERC se proporcionan dos nuevos modificadores, `timed` y `atomic`, que ofrecen una cierta alternativa de alto nivel a la sincronización. En caso de los CJThreads, la baja inversión de prioridad que introducen las operaciones realizadas sobre las colas de mensajes, hace que puedan servir como alternativa a la sincronización de tiempo real.

La gestión automática de memoria es abordada por todas las aproximaciones salvo por los CTJThreads. En el caso de PERC y los RTJThreads la solución descansa en la existencia de un algoritmo de recolección de basura de tiempo real. RTSJ y RTCORE añaden soporte para regiones. Además, RTCORE permite el almacenamiento de objetos en la pila, mediante el modificador `stackable`.

Las impredictibilidades derivadas de la carga y la descarga dinámica de código así como las derivadas de la inicialización de las clases Java tan sólo son abordadas por RTSJ y por RTCORE. RTSJ no dice nada al respecto, pero sus dialectos para la alta integridad, como el ravenscar-Java, si que tratan el tema imponiendo restricciones adicionales sobre la ejecución de los programas. RTCORE modifica el cargador de clases para los objetos de tipo `CoreObject` de tal manera que las clases utilizadas pasan a ser otras diferentes.

La posibilidad de interactuar con eventos externos está cubierta por RTCORE, RTSJ y por PERC. En RTSJ existe la posibilidad de que la máquina virtual responda a las señales externas. RTCORE proporciona también un cierto soporte mediante los manejadores de interrupciones. Y por último PERC soporta este tipo de interacción mediante el uso de excepciones asíncronas.

Por último, la única aproximación que no proporciona ningún tipo de mecanismo para realizar accesos al hardware subyacente es la de los RTJThreads, más centrada en los sistemas de tiempo real que en los embebidos. El resto de las soluciones incluyen clases especiales que sirven para realizar dichos accesos de bajo nivel.

2.3.7. Conclusiones

Desde su aparición, RTSJ y RTCORE, se nos han presentado como competidoras [90] y según nuestra opinión, a día de hoy, la que posiblemente presenta un mejor futuro es RTSJ pues tanto desde el punto de vista industrial como investigador, directa o indirectamente, se está prestando más atención a RTSJ que a RTCORE. El

sector industrial pone de manifiesto su interés en RTSJ mediante grandes proyectos -Golden Gate o RTJES- y máquinas virtuales comerciales de tiempo real -jTime, Jamaica o Mackinack- vinculadas a RTSJ para las cuales no existe un equivalente en el mundo RTCORE. También, el sector de la investigación parece mostrar cierta preferencia a favor de RTSJ. Realizando estadísticas a partir de los artículos publicados en *workshops* y conferencias especializadas, podemos ver que la mayoría de los artículos relacionados con Java de tiempo real opta por RTSJ en vez de RTCORE como tecnología base hacia la que dirige su investigación. Por tanto, de triunfar alguna, la que más bazas tiene en su haber es RTSJ.

Desde el punto de vista particular de esta tesis, el modelo que resulta más interesante es el proporcionado por RTSJ. En un principio la elección de una tecnología base sobre la que cimentar una solución se podría reducir a RTSJ y a RTCORE, dejando de lado aproximaciones menores como son los CJThreads, los RJThreads y PERC. A la hora de elegir entre ambas lo que ha resultado más determinante han sido los modelos arquitectónicos. Se ha optado por RTSJ en vez de RTCORE por ser el primero el que presenta un modelo arquitectónico más flexible y más próximo al de Java tradicional. Esta elección, en principio, nos debería de allanar el camino a la hora de aplicar dicha tecnología a diferentes modelos de distribución, pues a mayor grado de generalidad, más sencilla debería de ser su integración con el modelo de distribución.

Esta elección se ve reforzada por la existencia de múltiples implementaciones de RTSJ que pueden ser utilizadas para el desarrollo de aplicaciones Java de tiempo real, facilitándonos pues la construcción de prototipos.

Y de todas las implementaciones de RTSJ existentes en la actualidad, la que resulta más atractiva es la de referencia, jTime.

2.4. Middleware de distribución para Java de tiempo real

El último paso que se da en este análisis del estado del arte, es el de analizar las diferentes propuestas existentes para Java de tiempo real distribuido. Los trabajos descritos en esta sección constituyen el núcleo duro sobre el que se construye esta tesis y su análisis es importante pues permite identificar cuáles son las principales aproximaciones presentes en este área de conocimiento y sus respectivas carencias.

Tal y como resulta lógico, el número de trabajos que aborda la temática de Java de tiempo real distribuido -véase el cuadro resumen 2.3- es menor que el existente para los sistemas centralizados, lo que permite su análisis en bastante detalle. Siguiendo el mismo esquema que se utilizó en Java de tiempo real centralizado, en primer lugar, en la sección 2.4.1, se verán cuáles son los problemas específicos a los que se enfrenta Java de tiempo real distribuido. Tras ello aparecerán una serie de secciones -2.4.2, 2.4.3, 2.4.3, 2.4.5, 2.4.6, 2.4.7, 2.4.8 y 2.4.9- que presentan, al igual que se hizo en la sección Java de tiempo real para sistemas centralizados, una a una, las diferentes aproximaciones. Luego vendrán dos secciones donde se analiza tanto colectivamente -sección 2.4.10- como individualmente -sección 2.4.11- dichas

Proyecto/Institución	Distribución	Lenguaje	Objetivos
DRTSJ	RMI	RTSJ	Definición de una especificación.
JCP RTSJ and RTCORBA Synthesis	CORBA	RTSJ	Definición de una especificación.
JConsortium RTCORE and RTCORBA Synthesis	CORBA	RTCORE	Definición de una especificación.
RTZen	CORBA	RTSJ	Diseño de un ORB para sistemas embebidos de tiempo real.
Universidad de York	RMI	RTSJ	Definición de un middleware de distribución de tiempo real.
Universidad Politécnica de Madrid	RMI	RTSJ	Perfiles para sistemas distribuidos RTRMI de tiempo real estricto y con calidad de servicio.
Universidad de Texas	RMI	RTSJ	Modelos de distribución de tiempo real para componentes móviles en RTRMI.
Universidad de Twente	CSP	Java	Construcción de sistemas distribuidos de tiempo real basados en paso de mensajes.

Cuadro 2.3: Trabajo más relacionado con Java de tiempo real distribuido

aproximaciones, identificando sus principales limitaciones. Y por último está la sección 2.4.12 de conclusiones.

2.4.1. Retos a abordar por Java de tiempo real distribuido

Si bien en sistemas centralizados Java de tiempo real existe un conocimiento más o menos amplio de cuáles son los problemas a abordar, en sistemas distribuidos, posiblemente debido a la no existencia aún de soluciones totalmente cerradas, no existe un conocimiento tan exhaustivo de los problemas que han de ser abordados. Recopilando y combinando la información de las diferentes aproximaciones que a continuación se presentarán y teniendo en mente también el modelo RTCORBA, se ha llegado a que los principales retos que debería de abordar Java de tiempo real distribuido son los siguientes:

1. **Gestión distribuida del procesador** Por lo general, en los middlewares de propósito general, no existe ningún tipo de garantía sobre la gestión del procesador realizada en el servidor durante la atención de una invocación remota. Esto puede provocar que las tareas de mayor prioridad sufran fuertes inversiones de prioridad durante una invocación remota en el caso de que tengan que

esperar a que finalicen otras de menor prioridad. Por lo general esta limitación se solventa introduciendo algún tipo de mecanismo de gestión del procesador en el servidor encargado de mantener un esquema de prioridades extremo a extremo coherente.

2. **Gestión de memoria predecible extremo a extremo** Por lo general en un middleware de distribución basado en Java tanto el middleware como el objeto remoto consumen memoria de forma dinámica. Lo que hace que resulte necesario introducir algún tipo de mecanismo que recicle esa memoria cuando ya no está en uso. Desde el punto de vista de las tecnologías Java de tiempo real que se han visto, esto obliga a la incorporación de algún tipo de técnica -regiones, recolección de basura, objetos en pila, gestión delegada- que se pueda utilizar de forma conjunta con la funcionalidad ofertada por el middleware de distribución.

3. **Gestión de conexiones** Por lo general tanto el uso que se haga de las conexiones de red como el tipo de red subyacente sobre la cual se apoya el middleware pueden repercutir en los tiempos máximos de respuesta de la invocación remota significativamente. Este tipo de limitación, tal es el caso de RTCORBA, se suele abordar introduciendo algún tipo de mecanismo que permita reducir la inversión de prioridad sufrida a la hora de enviar y de recibir datos a través de una conexión así como introduciendo mecanismos que reduzcan la inversión de prioridad asociada a la gestión de conexiones durante su apertura, compartición y multiplexación.

4. **Gestión de la concurrencia** Por lo general, los middlewares de propósito general tampoco suelen especificar ningún tipo de modelo de concurrencia en el servidor, lo cual, una vez más, puede provocar nuevas inversiones de prioridad en los clientes. Este tipo de limitación se suele solventar introduciendo, tal y como se hace en el *threadpool* de RTCORBA, algún tipo de mecanismo de control fino que posibilite su gestión en el servidor.

5. **Recolección de basura distribuida** Algunos middlewares de distribución Java -RMI [191] entre ellos- ofrecen versiones distribuidas del recolector de basura que, aunque evitan que el programador destruya un objeto remoto de forma prematura, suponen un coste computacional extra que puede interferir en la ejecución de las diferentes tareas de tiempo real. En estos casos resulta necesario introducir algún tipo de algoritmo de gestión de memoria distribuida de tal manera que la interferencia que éste introduce en las tareas de tiempo real se encuentre acotada. Hasta el momento, éste es uno de los retos pendientes que no ha sido plenamente abordado por las diferentes aproximaciones RTRMI existentes en el estado del arte.

6. **Descarga dinámica de código** Adicionalmente, RMI también hace uso de mecanismos que son capaces de descargar el código necesario para ejecutar una aplicación de forma dinámica. Al igual que sucede con la carga y la descarga de clases en sistemas centralizados, esta descarga puede provocar la pérdida

de plazos porque los tiempos que suele implicar la descarga suelen ser muchas veces mayores que los plazos de respuesta de las aplicaciones. Por lo general, al igual que su homólogo centralizado, este tipo de mecanismo hasta el momento no ha sido aún muy estudiado, sugiriéndose en la mayor parte de los casos evitar su empleo.

7. **Modelo de eventos distribuidos** Como complemento a la invocación remota síncrona, muchos middlewares de tiempo real ofrecen diferentes fórmulas de asincronía que permiten desligar la ejecución del cliente de la del servidor. La gran ventaja que presentan estas fórmulas suele ser una reducción en los tiempos máximos de bloqueo experimentados por el cliente y además la posibilidad de independizar la ejecución de la lógica utilizada para procesar el evento de la lógica que lo genera. Al igual que en los modelos de invocación remota síncrona, resulta necesario que el comportamiento de estos mecanismos esté dotado de algún grado de predictibilidad a fin de favorecer su utilización en sistemas de tiempo real.

Estos siete puntos constituyen los retos que en mayor o menor medida las diferentes aproximaciones a Java de tiempo real distribuido que a continuación pasaremos a presentar y a analizar deberían de abordar.

2.4.2. DRTSJ

Dentro del mundo de la especificación el esfuerzo más relevante, encaminado hacia la definición de una especificación RTRMI, es el denominado como *Distributed Real-Time Specification for Java* (DRTSJ). Este proceso se encuadra dentro del modelo de comunidades de Java, los *Java Community Processes*, y su *Java Specification Request* asociado es el JSR-50 [93]. En la actualidad, no existe ningún tipo de borrador público de la especificación y los únicos documentos que se encuentran disponibles son la página web del grupo de trabajo [93] y una serie de artículos ([89], [185], [184] y [11]) que lo describen a grandes rasgos.

Basándose en el hecho de que las aplicaciones de tiempo real pueden presentar diferentes tipos de requisitos, se definen tres niveles de integración [185]: el nivel 0, el 1 y el 2.

- **Nivel 0** Este nivel representa aquellos escenarios donde no es necesario realizar ningún tipo de añadido ni a RMI ni a RTSJ. En este nivel las interfaces, el modelo de implementación, las herramientas de desarrollo, el significado del tiempo y la semántica de fallo utilizadas son las de RMI estándar. Independientemente de cual sea el hilo cliente que realice la invocación, el hilo remoto se comportará como si se tratase de un hilo Java normal; incluso en el caso de que el cliente sea un hilo de tiempo real. Y por tanto, se puede decir que en este modelo no se requieren cambios en las diferentes clases de RMI pero que tampoco se obtienen garantías sobre el tiempo máximo de respuesta extremo a extremo.

- **Nivel 1** El nivel 1 de integración es el primero que proporciona garantías sobre el tiempo de respuesta extremo a extremo. A tal fin, se realizan cambios en los elementos clave de RMI. Así, el modelo de objeto remoto, asociado a la interfaz `java.rmi.Remote`, se ve complementado con dos nuevas interfaces: `Realtime-Remote` y `NoHeapRealtimeRemote` que garantizan cotas máximas temporales a la ejecución de una invocación remota. La inclusión de estas interfaces también requiere cambios en las herramientas de desarrollo de aplicaciones RMI. Y por tanto, se puede decir que se garantiza un tiempo máximo de respuesta extremo a extremo a costa de introducir nuevas interfaces en el plano del programador.

- **Nivel 2** El nivel 2 integra la predictibilidad del nivel 1 con la flexibilidad que provee el paradigma de programación del *hilo distribuido* de tiempo real. La jerarquía de clases de RMI se enriquece con nuevas clases para la manipulación de hilos distribuidos: `DistributedRealtimeThread` y `Distributed-NoHeapRealtimeThread` y de eventos asíncronos: `RemoteAsyncronousEvent` y `RemoteAsyncronousEventHandler`. Y por tanto, se podría decir que se garantizan unos tiempos de respuesta extremo a extremo máximos y un modelo de programación más flexible que el proporcionado por el nivel 1, a costa de introducir cambios en el núcleo de la máquina virtual de tiempo real.

2.4.3. JCP RTSJ and RTCORBA Synthesis

Aunque el propio RTCORBA [133] incorpora una correspondencia para el lenguaje Java, ésta tiene un carácter más testimonial que funcional. El modelo de prioridades, la falta de mecanismos de herencia de prioridad y la recolección de basura hacen que la predictibilidad que se puede obtener extremo a extremo con Java tradicional, cuando se combina con RTCORBA, sea muy baja. Con la aparición de RTSJ, esta situación se ha visto alterada, volviendo a relanzarse el problema de cómo realizar una adecuada correspondencia entre ambas especificaciones.

En la actualidad, este proceso no ha concluido, encontrándose en sus primeras fases. Dos son los documentos disponibles que lo describen: un *request for proposal* [128] y un *initial submission* [130].

La única respuesta a la petición de propuesta, la [130], está aún en vías de desarrollo. Por el momento se han definido las interfaces Java de la correspondencia, a partir de las del modelo RTCORBA, pero la forma en que éstas son soportadas por RTSJ, es aún un tema muy abierto tal y como se concluye en [67]. Uno de los problemas más difíciles a los que se enfrenta es cómo acomodar características tan particulares como son la `ScopedMemory` o los diferentes tipos de hilos de RTSJ dentro del modelo arquitectónico de CORBA.

2.4.4. JConsortium RTCORE and RTCORBA Synthesis

El JConsortium, al igual que previamente había hecho el RTJEG, propuso la integración de su especificación con RTCORBA mediante una correspondencia. Actualmente este proceso no se ha completado y la información a la que se tiene acceso

es la petición de propuesta, *request for proposal* [129], y un envío revisado, *revised submission* [131].

La petición de propuesta [129] establece, análogamente a [128], los principales puntos que han de abordar las diferentes soluciones que se propongan a la integración entre RTCORE y RTCORBA.

A día de hoy el trabajo no ha concluido y de lo que se dispone es de un documento [131] que lo describe a grandes rasgos. El documento recoge de forma parcial tanto aspectos relativos a las interfaces como a la implementación. En el plano de las interfaces, se establece una correspondencia entre el sistema de prioridades de RTCORBA y el de RTCORE. Y en el plano de la implementación, se identifican las características de RTCORE que resultan relevantes a la hora de soportar RTCORBA, sin dar pautas de cómo podrían ser utilizadas.

2.4.5. RTRMI: Universidad York

Uno de los grupos de tiempo real, que ha trabajo de forma activa en la definición de RTRMI, ha sido el grupo de tiempo real de la Universidad de York. Su principal línea de trabajo se ha centrado en la definición de un marco de integración entre las especificaciones RMI y RTSJ [30] [29]. En la actualidad el marco carece de implementación software.

El primer paso dado es la definición de interfaces para RTRMI. El modelo de clases propuesto, siguiendo las pautas de DRTSJ, replica la estructura de clases de RMI, caracterizándose éstas por la inclusión del prefijo `Realtime`. La principal diferencia entre estas clases y las antiguas es que las segundas proveen garantías de tipo temporal. Además, se añaden nuevos métodos a la jerarquía de clases. En el servidor se introduce un método, `export`, que permite la exportación de objetos remotos con características de tiempo real. En el cliente se incluye un nuevo método, `invokeRealtime`, que posibilita el envío de datos en tiempo real desde el cliente al servidor. El segundo paso dado consiste en la definición de los algoritmos internos que permiten garantizar la obtención de cotas máximas al tiempo de respuesta de la invocación remota. Estos mecanismos son implementados por la jerarquía de clases de tiempo real y atacan dos tipos de impredictibilidades: las producidas por la gestión del procesador y las derivadas de la gestión de memoria.

En el caso del control del procesador se proponen dos soluciones distintas, una para el cliente y otra para el servidor. En el cliente el modelo está basado en la existencia de un única hebra que se bloquea síncronamente hasta que recibe el resultado de la invocación remota desde el servidor. En el servidor el modelo está basado en el de RTCORBA e incorpora un *threadpool*.

En el caso del control predecible de la memoria también se diferencian dos soluciones distintas para el cliente y para el servidor. En el cliente se propone que todos los objetos instanciados durante el proceso de invocación remota sean creados, de forma similar a lo hecho en la invocación local, en la memoria del hilo invocante. En el servidor la solución es más compleja y consiste en crear los parámetros de la invocación en la misma área de memoria donde se creó la instancia del objeto remoto invocado.

En [29] se complementa el modelo con la definición de un prototipo. El prototipo se construye a partir de las clases de RMI disponibles para el entorno J2ME. En este prototipo se explica cómo los diferentes hilos utilizan las estructuras internas del middleware para realizar una invocación remota. Finalmente, este trabajo también identifica cinco potenciales mejoras realizables en el marco desarrollado: (1) extensiones al servicio de nombres de RMI (el *rmiregistry*) para que sea de tiempo real; (2) la descarga de código distribuido; (3) la inclusión de un recolector de basura distribuido de tiempo real; (4) mecanismos que permitan determinar el tamaño de los objetos serializados; y (5) un mecanismo de eventos distribuidos capaz de funcionar en tiempo real.

El último trabajo realizado ha sido la definición de un modelo extendido de referencias para RTSJ (ver [31]) con el fin de facilitar la implementación de aplicaciones complejas.

2.4.6. RTRMI y QoS: Universidad Politécnica de Madrid

El grupo de tiempo real de la Universidad Politécnica de Madrid mantiene una línea de investigación relacionada con RTRMI. Dentro de esa línea de investigación, se están desarrollando para el proyecto europeo HIJA (High-Integrity JAva) [6], dos perfiles de tiempo real: uno para aplicaciones de tiempo real crítico y otro para las acríticas [169] [170]. Con anterioridad, integrantes del grupo, analizaron la integración de protocolos de comunicación de tiempo real dentro de RMI [49].

A la hora de definir cada perfil, se consideran cuatro cuestiones principales: (1) el modelo computacional; (2) las adaptaciones requeridas en RMI, (3) en el modelo de concurrencia y (4) en el modelo de gestión de memoria. El modelo computacional hace referencia al funcionamiento interno del middleware de distribución. Las adaptaciones requeridas en RMI hacen referencia al modo en que se integra la nueva funcionalidad, ofrecida por los diferentes modelos, dentro de la jerarquía de clases de RMI. El modelo de concurrencia hace referencia al esquema de hilos que da soporte al mecanismo de comunicación remota. Y finalmente, el modelo de memoria hace referencia a qué tipo de memoria, de las múltiples que nos podemos encontrar en RTSJ, es la que se utiliza durante el proceso de invocación remota.

El primer perfil desarrollado se denomina HRTRMI y aparece orientado hacia aquellas aplicaciones que requieren de una alta predictibilidad.

Sus principales características son las que siguen. Su modelo está basado en el del perfil ravenscar-Java [83]. Desde el punto de vista del programador se incorporan nuevas clases: `RtRemoteStub`, `RtRemoteServer` y `UnicastRtRemoteObject`, que permiten garantizar cotas máximas en el tiempo de respuesta extremo a extremo. En el servidor se hace uso de dos hilos: aceptor, que espera peticiones entrantes desde el cliente; y el manejador, que realiza la invocación local sobre la instancia del objeto remoto correspondiente. En el cliente, al igual que ocurre en RMI estándar, el hilo invocante se bloquea síncronamente hasta recibir el resultado proveniente del servidor. Por último, la única memoria que se puede utilizar es la `ScopedMemory` y la `ImmortalMemory`, estando prohibida la `HeapMemory`.

El otro modelo desarrollado se denomina QoSRMI y aparece orientado hacia los

sistemas de tiempo real que exhiben requisitos de tiempo real más laxos que los de HRTRMI.

Sus principales características son las que siguen. Este perfil está orientado hacia los sistemas acríticos y es mucho más flexible que el de HRTRMI. Incorpora mecanismos que permiten controlar el ancho de banda, la memoria, el procesador y la disponibilidad de manejadores en el servidor. Desde el punto de vista del programador, se incorporan nuevas clases en el modelo de RMI: `RtQoSRemote`, `RtQoSRemoteStub` y `RtQoSUnicastRemoteObject` similares a las descritas para HRTRMI. El modelo de concurrencia de QoSRMI es más complejo que el de HRTRMI, permitiendo que el programador realice reservas que se asocian a referencias a objetos remotos en los clientes. Y por último, el modelo de gestión de memoria soportado está basado en el uso exclusivo de `HeapMemory`, prohibiéndose la utilización de la `ScopedMemory` y de la `ImmortalMemory`.

Con anterioridad al trabajo desarrollado en HIJA, de Miguel analizó cómo integrar adecuadamente los protocolos de transporte de tiempo real en RMI [49]. Más particularmente, el escenario estudiado fue el proporcionado por el modelo de integración de servicios de internet - *intserv* [85]- y la versión 1.2 de las clases de RMI de Sun.

2.4.7. RTRMI: Universidad de Texas

El grupo de tiempo real de la Universidad de Texas ha enfocado el problema de la definición de un modelo RTRMI hacia el dominio de los entornos móviles basados en tecnología de componentes [145] [39] [40] [181]. Su trabajo se ha concentrado especialmente en la propuesta de metodologías para la provisión de capacidades de tiempo real estricto (hard real-time) en sistemas móviles.

En el capítulo IV de la thesis de Rho [145] se definen las principales características del soporte RTRMI. La primera en aparecer es el esquema de concurrencia utilizado en el servidor. Éste se caracteriza por la existencia de tres hebras: *escuchadora* que espera peticiones entrantes; *trabajadora* que atiende a la primera etapa de la invocación remota; y la *manejadora* que ejecuta la lógica del método remoto. La hebra escuchadora es la que se ejecuta a mayor prioridad, la trabajadora ejecuta a una prioridad media y la manejadora a una inferior. Las hebras manejadoras reparten el procesador haciendo uso de una política EDF. Las hebras escuchadoras realizan el control de admisión. El trabajo no caracteriza ningún otro tipo de interfaz de programación ni horizontalmente ni verticalmente.

Tomando el soporte RTRMI descrito, los autores construyen servicios avanzados. En [39] se propone REALTOR, un protocolo de descubrimiento de servicios especialmente definido para entornos hostiles. En [40] se estudia el problema de cómo definir una tecnología de componentes con capacidades de migración. Y por último, en [181], se presenta un modelo de componentes para aplicaciones de tiempo real distribuidas.

2.4.8. RTZen: Universidad de California

El grupo *Distributed Object Computing* (DOC) de la Universidad de California mantiene abierta una línea de investigación en la que trabaja con los modelos RTSJ y

RTCORBA. Más concretamente, dentro del proyecto RTZen ([97] [98] [99] [100] [101] [102]) se están desarrollando una serie de herramientas que facilitan el desarrollo de aplicaciones distribuidas y embebidas de tiempo real. Su objetivo es la consecución de un ORB para sistemas embebidos que sea sencillo de utilizar y que haciendo uso de RTSJ y de patrones de diseño permita reducir las inversiones de prioridad que experimentan las aplicaciones distribuidas de tiempo real.

Basándose en los patrones definidos anteriormente por Pyarali [142] dentro del proyecto TAO (ORB para sistemas de tiempo real codificado en C++), RTZen ha incorporado una serie de estrategias que lo dotan de predictibilidad extremo a extremo además de eficiencia. Éstas aparecen en el nivel ORB, donde se han diseñado técnicas que simplifican el proceso de invocación remota [102], y en el nivel POA de CORBA, donde se han propuesto mecanismos de demultiplexación de complejidad máxima acotable por una función $\Theta(1)$ y donde además se utiliza el *threadpool* de RTCORBA.

RTZen también obtiene ventajas provenientes del uso de RTSJ. Para ello hace uso del modelo de hilos y de gestión de memoria de RTSJ dentro del ORB y del POA. En el ORB se ha buscado una alta predictibilidad utilizando para ello los mecanismos de RTSJ que resultan más predecibles: el `NoHeapRealtimeThread` y la `ScopedMemory`. A nivel POA se ha buscado el preservar el paradigma de programación de Java para el programador y en vez de utilizar el `NoHeapRealtimeThread` y la `ScopedMemory`, se ha preferido utilizar hilos de tipo `RealtimeThread` y memoria de tipo `HeapMemory`.

En sus últimos trabajos, el proyecto ha empezado a desarrollar una serie de herramientas que facilitan el desarrollo de aplicaciones de tiempo real haciendo uso de RTZen. Una de ellas, RTZen-kit [160], posibilita la realización de una configuración fuera de línea de los componentes que estarán disponibles durante la ejecución de la aplicación distribuida. Otra de ellas, Isoleak [144], permite la detección de fugas de memoria dentro de una región.

2.4.9. Otras aproximaciones

Hildenrik [81], del laboratorio de control de la Universidad de Twente, ha propuesto una aproximación al problema de generar sistemas de tiempo real distribuidos basados en Java haciendo uso del paradigma *Communication Sequential Process* (CSP) [82]. La gran ventaja de este modelo es que su código no es sólo sencillo de utilizar, sino que también es seguro por estar basado en las reglas de CSP. El principal problema abordado es la inversión de prioridad dentro de los canales de comunicación. Para solventarlo se propone el empleo de una técnica basada en prioridades, la técnica del *rate-monotonic* y un protocolo que controla la inversión de prioridad.

2.4.10. Análisis conjunto

Aplicando como criterios para la comparación los diferentes retos descritos en la sección 2.4.1, se ha evaluado las diferentes aproximaciones a Java de tiempo real descritas, a excepción de las correspondencias RTCORE-RTCORBA y RTSJ-RTCORBA por ser muy pobre la información que sobre ellas hay disponible. En el caso de RTSJ-RTCORBA existe un buen sustituto, RTZen, que al estar basado en

	Gestión de procesador	Gestión de memoria	Gestión de conexiones	Gestión de concurrencia	Recolección de basura distribuida	Descarga dinámica de código	Modelo de eventos distribuidos
RTCORBA-RTZen	√	-	√	√	-	-	-
DRTSJ	√	I	-	-	-	-	√
RTRMI-Univ. York	√	√	√	√	I	-	I
RTRMI-Univ. Politécnica	√	√	√	√	I	-	-
RTRMI-Univ. Texas	√	-	-	I	-	-	-
RTRMI-Univ. Twente	√	-	-	-	-	-	√

Cuadro 2.4: Análisis conjunto de las diferentes aproximaciones a Java de tiempo real distribuido

RTSJ y RTCORBA nos ofrece un buen suplente de la correspondencia. En el caso de RTCORE-RTCORBA, al no existir un proyecto que vaya en esa misma línea, no hay sustituto.

Se ha evaluado cada una de las aproximaciones con los criterios definidos y con los resultados obtenidos se ha construido la tabla 2.4. En ella, a cada una de las aproximaciones se le ha asignado o bien un √, en el caso de que aborde la limitación; o un I, en el caso de que tan sólo se identifique; o un -, en el caso de que no se identifique ni se aborde. Por identificar se entiende que se tome conciencia de uno de los retos y por abordar se entiende que además de ello se proponga algún tipo de medida encaminada a su solución.

De todas las limitaciones presentadas, al igual que sucedía en el caso de Java de tiempo real centralizado, la mejor comprendida es la de la gestión del procesador; tal y como se pone de manifiesto en el hecho de que todas las aproximaciones la identifiquen y ofrezcan mecanismos que posibiliten la realización de una gestión predecible basada en prioridades.

No ocurre lo mismo con el uso predecible de la memoria extremo a extremo. Éste es un problema que podríamos decir que ha sido identificado pero para el cual no existen grandes soluciones. Aunque en el trabajo realizado por DRTSJ se identifica este problema, sólo la Universidad de York y la UPM lo abordan mediante el empleo de regiones. En el caso de RTZen la aproximación híbrida utilizada (`ScopedMemory` dentro del ORB y `HeapMemory` dentro del POA) rompe la línea de predictibilidad extremo a extremo, motivo por el cual decimos que no llega a abordar completamente el problema.

La necesidad de realizar una gestión fina sobre la conexión, identificada ya en el modelo RTCORBA, está presente en RTZen y las aproximaciones de tipo RTRMI de las Universidades de York y Politécnica de Madrid. En el caso de la Universidad de York se propone un modelo en el que cada invocación remota implica el establecimiento de una nueva conexión. En el caso de la UPM, dependiendo del perfil al cual vaya dirigida la aplicación -QoSRMI o HRTRMI-, se aboga o bien por el uso de un modelo flexible de gestión de las conexiones o se tiende más hacia la creación de un conjunto de conexiones en una fase de inicialización.

La importancia de tener un modelo de concurrencia en el servidor, ya recogido en el modelo de RTCORBA mediante la definición de un *threadpool*, está también presente en muchas de las soluciones. Así, RTZen, York y la UPM dan una cobertura específica a este tipo de mecanismos, ofreciendo la Universidad de Texas un menor grado de cobertura.

La inclusión de mecanismos de gestión de memoria distribuida de tiempo real es uno de los retos que aún no ha sido abordado. Así, aunque algunas aproximaciones, como por ejemplo la de la Universidad de York, han identificado el problema que supone para el cumplimiento de los plazos dicho mecanismo, se sigue sin ofrecer ningún tipo de solución operativa, sugiriéndose la supresión de dicho mecanismo.

La descarga de código distribuido, pese a ser un mecanismo que potencialmente puede introducir unas altas latencias en los tiempos de respuesta de las aplicaciones distribuidas, tampoco ha sido muy estudiado y sólo algunos trabajos identifican parcialmente este problema, proponiendo en su mayor parte que no se utilice.

Por último, la necesidad de un modelo de eventos distribuido ha sido aborda por DRTSJ, que ha propuesto un esquema de clases que extiende el paradigma ya existente para RTSJ a RMI. También ha sido identificado como un mecanismo de interés por la Universidad de York que lo propone como una extensión. Por último, podemos decir que la aproximación de la Universidad de Twente le proporciona, al estar basada en el paso de mensajes, soporte de forma nativa.

2.4.11. Análisis crítico

A continuación, trabajo a trabajo, se procede a ir analizando cuales son los puntos más fuertes y los más débiles de cada una de las aproximaciones a Java de tiempo real distribuido descritas. El objetivo perseguido es determinar cuales son las principales limitaciones que lastran la evolución de cada una de las aproximaciones. Se han excluido del análisis las diferentes propuestas de integración con RTCORBA y el trabajo realizado por la Universidad de Twente.

- **DRTSJ** Quizás la gran ventaja que nos ofrece el modelo DRTSJ es su alta fidelidad para con el modelo de programación Java. La elección que se hace de RMI, en vez de RTCORBA, le confiere unas posibilidades de integración que difícilmente podrán ser superadas por el modelo de distribución CORBA. Sin embargo, a la hora de definir el modelo, el grado de integración alcanzado no es excesivamente elevado debido, en gran parte, a una sobre-especificación de ciertas características.

Un punto criticable lo constituye el modelo de distribución propuesto que diferencia entre dos tipos de objetos remotos: aquellos que son susceptibles de ser invocados con garantías temporales de aquellos que no. En principio, el razonamiento empleado para tal diferenciación es seguir el modelo de RMI donde se marcan los objetos que pueden ser invocados remotamente con `Remote`, para diferenciarlos de aquellos que no pueden recibir invocaciones remotas. Siguiendo esa línea argumental, en DRTSJ, se diferencia entre aquellos objetos remotos que son capaces de recibir invocaciones remotas predecibles de aquellos que no son capaces, justificándose así la interfaz `RealtimeRemote`. Sin embargo, esta aproximación presenta una limitación importante ya que la reutilización de objetos remotos legados no resulta inmediata pues éstos implementan `Remote` en vez de `RealtimeRemote`. Una alternativa como por ejemplo decidir el tipo de hilo que realiza la invocación remota en el servidor en función del hilo que es utilizado en el cliente, facilitaría la reutilización de objetos remotos diseñados para sistemas de propósito general en los de tiempo real.

Otra de las características de DRTSJ que puede llegar a ser perjudicial es el elevado número de entidades concurrentes que el programador puede tener que llegar a manejar. Tal y como se nos presenta DRTSJ, el programador tendrá la posibilidad de elegir entre un total de cinco tipos de hilos: el primero de ellos -`Thread`- proveniente de Java tradicional; otros dos -`RealtimeThread` y `NoHeapRealtimeThread`- provenientes de RTSJ; y por último, otros dos más -`DistributedRealtimeThread` y `NoHeapDistributedRealtimeThread`- provenientes del propio modelo DRTSJ.

Pero su punto más débil es su propio estado de madurez. Es de esperar, al igual que sucedió con RTSJ, que desde las actuales aproximaciones a las que finalmente se implanten se produzcan cambios significativos tanto en el modelo computacional como en el conjunto de interfaces actualmente caracterizado.

- **RTRMI: Universidad de York** El marco definido por la Universidad de York realiza una buena identificación de los principales problemas de integración existentes entre RMI y RTSJ, siendo éste su punto más fuerte. Su punto más débil aparece a la hora de abordar dichos problemas mediante la definición de algoritmos de gestión de sencilla implementación.

  En el modelo propuesto se emplea un mecanismo de *threadpool*, pero sin embargo las posibles ventajas que puede aportar su uso no están claras. En el modelo de RTCORBA la posibilidad de mantener conexiones de red multiplexadas permite desacoplar asíncronamente el procesado de una petición entrante de la invocación al objeto remoto, permitiendo la obtención de una mayor eficiencia en el uso de los recursos del sistema. Sin embargo, en el caso del modelo de integración de Borg, la elección de RMIOP impide el uso de la técnica de multiplexación de conexiones. Es más, el cambio de contexto requerido por los hilos manejadores supone una carga computacional extra para la cual no se explica ningún tipo de contrapartida.

  En el servidor el modelo de gestión de memoria propuesto presenta una alta complejidad a la hora de ser implementado. En el modelo propuesto [29] se

crean los objetos remotos de la invocación remota en la misma región en la cual el objeto remoto ha sido creado, lo que hace que sea difícil de soportar. Partiendo del hecho de que el número potencial de invocaciones que es capaz de recibir un objeto remoto no está acotado, resulta necesaria la inclusión de un mecanismo de gestión de memoria en la región para evitar el agotamiento de la memoria disponible, pudiendo este mecanismo requerir cambios dentro de la propia máquina virtual de tiempo real.

Uno de los puntos más flacos del marco de integración es su estado de madurez. Los propios autores manifiestan que su trabajo aún requiere de mucha investigación [30] para conseguir buenas soluciones, apuntando a la integración de la gestión de memoria de RTSJ dentro de RMI como uno de los retos más novedosos.

- **RTRMI: Universidad Politécnica de Madrid**

  Los puntos más fuertes de la aproximación de la Politécnica de Madrid son dos: por un lado la identificación de una serie de requisitos para dos entornos de computación diferentes y por otro lado, la detección de una serie de limitaciones presentes en RMI que dificultan la construcción de aplicaciones de tiempo real. Sin embargo, presenta limitaciones a la hora de aprovechar los componentes desarrollados en un perfil en otro.

  La realización de dos perfiles excluyentes: HRTRMI y QoSRMI, impone restricciones a la hora de reutilizar los componentes de una aproximación en la otra. En el modelo propuesto, un componente realizado para la arquitectura HRTRMI no podrá ser utilizado directamente en la QoSRMI. La imposibilidad de que QoSRMI utilice la `ScopedMemory` impide que un componente HRTRMI pueda ser utilizado en el perfil QoSRMI pues éste tan sólo soporta la `HeapMemory`. La adopción de otro modelo, en el cual el tiempo real estricto fuese un caso particular de la calidad de servicio y donde se pudiese utilizar la `ScopedMemory` en QoSRMI, propiciaría un mejor aprovechamiento de los componentes HRTRMI dentro de QoSRMI.

  A nivel de interfaz, el modelo propuesto comparte con DRTSJ el problema de una sobre-especificación. La inclusión de múltiples, en este caso habría hasta tres, tipos de sustitutos y de interfaces remotas hace que los grados de reutilización bajen y que no se puedan aprovechar las aplicaciones realizadas en un entorno en el otro. Al igual que en DRTSJ, una aproximación menos verbosa, con un menor número de interfaces, mejoraría la aproximación.

- **RTZen: Universidad de California**

  Uno de los puntos más fuertes de la solución RTZen deriva del hecho de que sus resultados son fruto de la implementación, lo que les proporciona un alto grado de validez. A esto ha de sumársele que al estar basada en RTCORBA, no resulta necesario definir nuevas interfaces para el desarrollo de aplicaciones de tiempo real. Esto hace que esta línea sea una de las de mayor grado de madurez. Pero esa ventaja tiene también sus propios inconvenientes.

La principal limitación que se ha encontrado en RTZen ha sido la carencia de mecanismos que eviten las inversiones de prioridad del recolector de basura dentro de la invocación remota. En el servidor, la elección que se hace de la `HeapMemory` en vez de la `ScopedMemory` rompe la cadena de predictibilidad extremo a extremo. Y así, un cliente, independientemente de que éste sea o no un hilo de tipo `NoHeapRealtimeThread`, puede llegar a sufrir las inversiones de prioridad debidas al recolector de basura del servidor.

- **RTRMI: Universidad de Texas**

  El modelo propuesto por la Universidad de Texas aparece mucho más centrado en los problemas que son introducidos por la movilidad de código, los protocolos de descubrimiento o su modelo de componentes que en la definición de un modelo de distribución de tiempo real para RTRMI. No se llega a discutir realmente sobre la forma de integrar RTSJ con RMI, sino que RTRMI se presenta más como punto de partida para la construcción del modelo de componentes y del protocolo de descubrimiento.

2.4.12. Conclusiones

Resulta difícil tratar de aventurar cuál de las dos grandes líneas de integración, RTRMI o RTCORBA, será la que tendrá un mejor futuro.

A corto plazo el camino más sencillo de recorrer puede ser la integración de RTSJ con el modelo CORBA pues la existencia del modelo RTCORBA facilita mucho este proceso, reduciéndolo en un principio a una simple correspondencia entre especificaciones, no resultando necesario pues, tal y como ocurre con el modelo RMI, la definición de un modelo de gestión de recursos extremo a extremo. Pero por el otro lado, la propia existencia de RTCORBA puede dificultar el proceso porque hay características de RTSJ, como pueden ser el modelo de regiones, cuya integración escapa a la trivialidad.

Así, no resultaría extraño que siguiendo este camino se llegase a perder algunas de las buenas cualidades ofertadas por RTSJ, en favor del modelo RTCORBA.

A largo plazo, la vía de integración que potencialmente parece ser capaz de alcanzar una integración más sinérgica es la de DRTSJ. Esto se ve favorecido tanto por la sencillez arquitectónica de RMI, menos complejo que CORBA, como por la carencia de especificaciones RTRMI que impongan grandes condiciones de partida.

Por último, desde un punto de vista de la investigación, la solución que más innovación requiere es RTRMI. Aunque en un principio muchos de los mecanismos de gestión de recursos disponibles en RTCORBA pueden ser utilizados en el contexto RTRMI, estos no serán suficientes para soportar características especiales, propias de RMI, como la recolección distribuida de basura o la descarga dinámica de clases.

En esta tesis aparece alineada tecnológicamente con RTRMI siendo los trabajos más relevantes para ella dos. El primero de ellos es el realizado por DRTSJ cuyo objetivo es ofrecer una especificación lo suficientemente general para el desarrollo de aplicaciones distribuidas de tiempo real. El segundo es el marco de trabajo desarrollado en la Universidad de York que intenta, tomando como base a DRTSJ, caracterizar

los mecanismos internos que permiten el desarrollo de aplicaciones distribuidas de tiempo real.

El resto de trabajos juega, para esta tesis, un papel algo más secundario. RT-Zen, aunque basado en RTCORBA, también resulta interesante para el desarrollo de esta tesis pues parte de los algoritmos de gestión empleados en este modelo son de aplicación directa en el modelo que se pretende desarrollar. La relación con las aproximaciones HRTRMI y QoSRMI de la Politécnica de Madrid es de complementariedad pues estos dos perfiles ofrecen escenarios o casos de estudio a los que el modelo podría ser enfocado. Por último, el RTRMI de la Universidad de Texas complementaría también al modelo que se pretende desarrollar, con características avanzadas tales como el soporte de modelos de componentes o los protocolos de descubrimiento.

2.5. Resumen y conclusiones

En este capítulo se ha comenzado por realizar un análisis del estado del arte relativo al middleware de distribución de tiempo real, intentando ver cuáles son las diferentes técnicas y tecnologías existentes a la hora de construir sistemas Java de tiempo real distribuido. Se comenzó por el estado del arte relacionado con los sistemas de tiempo real, citando las principales técnicas utilizadas por los diferentes middlewares de tiempo real para continuar con el estudio del middleware de distribución. Tras ello, se ha estudiado con bastante detalle las tecnologías más relevantes para esta tesis: la tecnología Java de tiempo real para sistemas centralizados y la de sistemas distribuidos.

De este análisis se ha llegado a la convicción de que en ambas áreas de conocimiento es posible realizar múltiples contribuciones originales. En Java de tiempo real centralizado, especialmente en RTSJ, existe aún un campo sin explorar suficientemente importante relacionado con el modelo de gestión basado en regiones y la forma en que éstas se pueden utilizar para realizar aplicaciones. Y en Java de tiempo real distribuido una de las mayores limitaciones existente es la carencia de soluciones operativas que permitan el desarrollo de aplicaciones distribuidas de tiempo real. Y de entre todos los problemas específicos que deberá de resolver Java de tiempo real distribuido destaca sobre todo uno: la integración del modelo de gestión de memoria de RTSJ dentro de los modelos arquitectónicos de RTCORBA y de RMI.

De forma práctica este estado del arte también ha servido para identificar tanto las especificaciones que serán utilizadas en la realización de esta tesis, así como las implementaciones software concretas que servirán realizar prototipos software concretos. RTSJ [4] y RMI [167] se nos muestran como las tecnologías más afines a esta tesis y la implementación jTime [173] de TIMESYS y el paquete opcional RMIOP [94] como los productos software más adecuados para realizar validaciones experimentales.

En el siguiente capítulo se comienza a construir el cuerpo de la tesis definiendo un modelo de middleware con soporte de tiempo real basado en RMI.

Capítulo 3

Modelo de middleware con soporte de tiempo real basado en RMI

Uno de los principales problemas que presenta RMI para el desarrollo de aplicaciones de tiempo real es la carencia de un modelo que caracterice cómo funciona internamente. Ello es debido a que en la especificación de RMI no se aclaran ciertos detalles relevantes para la construcción de sistemas de tiempo real, como pueden ser de dónde es tomada la memoria necesaria para realizar una invocación remota tanto en el nodo cliente como en el servidor, ni otros como la prioridad a la que se atienden las diferentes peticiones remotas o el modelo de concurrencia que es seguido en el servidor. Esto se ve agravado por el hecho de que RMI posee una serie de servicios básicos, como pueden ser el de recolección distribuida de basura o el de nombres, que también compiten por los recursos de los que dispone el middleware de distribución, interfiriendo con la ejecución del resto de las aplicaciones de una forma a priori que no está caracterizada. A este hecho se la ha de sumar que de forma práctica existe una carencia funcional de mecanismos de comunicación asíncronos que nos permitan desligar la ejecución del cliente de la del servidor. Y si bien todas estas carencias son aceptables en las aplicaciones de propósito general donde no hay plazos que cumplir, para la mayor parte de los sistemas de tiempo real tal indeterminismo es poco aceptable.

Así pues el objetivo de este capítulo es el de caracterizar el comportamiento interno de un middleware parecido a RMI de tal manera que pueda ser utilizado en la construcción de sistemas de tiempo real. Para ello se definirá un modelo para RTRMI de tal manera que en todo momento se sepa cómo son utilizados internamente los diferentes recursos con los que cuenta el middleware de distribución. La forma de proceder será similar a la seguida en RTCORBA, donde se clarifica el comportamiento del modelo interno de computación CORBA mediante la definición tanto de nuevas entidades como de modelos de gestión de recursos. El modelo dará soporte para características básicas de RMI como son la invocación remota síncrona así como los servicios de recolección distribuida de basura o el servicios de nombres y para otros que actualmente no están presentes en la especificación actual de RMI,

como es la invocación remota asíncrona, pero que aún así resultan interesantes para ciertos sistemas de tiempo real. Lo que de forma práctica permitirá al diseñador de aplicaciones tener la suficiente información para poder ser capaz de configurar la aplicación distribuida de tiempo real de tal manera que se vean satisfechas las diferentes restricciones temporales de sus tareas.

Para ello, este capítulo comienza -en la sección 3.1- caracterizando un modelo de primitivas y de capas para el middleware de comunicaciones RMI, estableciendo también relaciones entre este modelo y el proporcionado por las tecnologías Java/RTSJ y RMI. Tras ello -sección 3.2- se completará el modelo con la definición de un modelo de predictibilidad que definirá el soporte mínimo que habrá de soportar el middleware de infraestructura y las entidades que en el de distribución aparecen para complementarlo. Después vendrán una serie de secciones donde se caracterizará el comportamiento interno del middleware de distribución. Se comienza explicando cómo se atienden las invocaciones remotas tanto síncronas como asíncronas -sección 3.3. Después se caracteriza un posible comportamiento para el recolector de basura distribuido -sección 3.4- y el servicio de nombres -sección 3.5. Cierra el capítulo la sección de conclusiones y líneas futuras -sección 3.6.

3.1. Modelo de capas y de primitivas para RMI

El primer paso que se da a la hora de construir el modelo para el middleware de distribución es caracterizar un esquema de capas y de primitivas para RMI, definiendo la funcionalidad que cada una de las capas ofrece al resto del sistema. El objetivo perseguido es el de crear un modelo lo suficientemente abstracto como para ser aplicado a diferentes middlewares de distribución, pero al mismo tiempo lo suficientemente próximo a RMI y a RTSJ como para ser fácilmente soportable por ambas tecnologías.

Tal y como se muestra en la figura 3.1 y siguiendo el modelo descrito en el estado del arte, se identifican tres capas: *infraestructura*, *distribución* y *servicios comunes*. La capa de infraestructura es la que controla los recursos básicos gestionando la memoria, el procesador y la capacidad de transmitir datos entre diferentes nodos. La de distribución proporciona capacidad de comunicación remota con otros nodos permitiendo tanto el registro y el desregistro de objetos remotos y de sustitutos así como la realización de invocaciones remotas. Y por último, la de servicios comunes consta de un servicio de sistema dedicado a la recolección distribuida de basura y otro de nombres.

3.1.1. Principales primitivas

Cada una de las capas ofrece al resto del sistema una serie de funcionalidades bajo la forma de primitivas:

- **Infraestructura**

  Esta capa permite que el middleware de distribución o la lógica del programador hagan uso de la memoria, el procesador y los recursos de comunicación subyacentes.

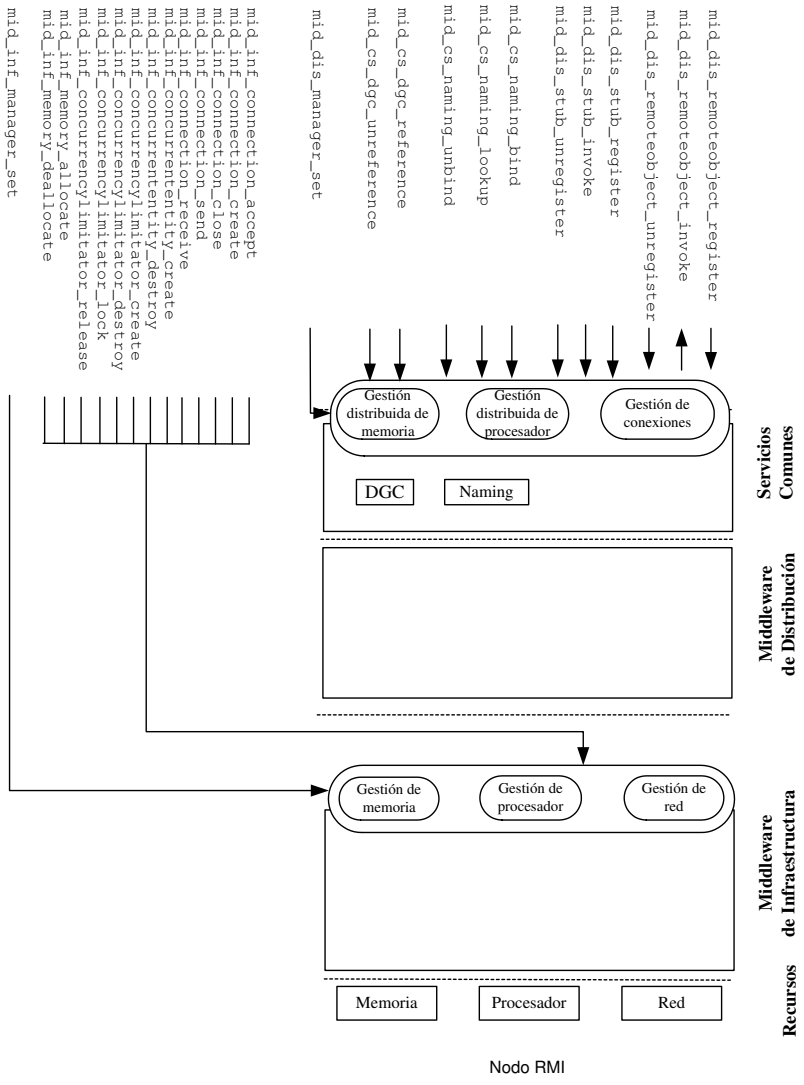


Figura 3.1: Modelo de primitivas y de capas para RMI

Para interactuar con la gestión de memoria define dos primitivas: `allocate` y `deallocate`. La primera permite reservar un bloque de memoria y la segunda liberarlo.

Para interactuar con la gestión del procesador se proporcionan tres conjuntos de primitivas. El primero, con el prefijo común `concurrententity`, contiene las primitivas que permiten la creación (`concurrententity_create`) de entidades concurrentes y su destrucción (`concurrententity_destroy`). El segundo conjunto de primitivas, con prefijo común `concurrencylimitator`, permite la creación (`concurrencylimitator_create`) y la destrucción (`concurrencylimitator_destroy`) de cerrojos, así como la obtención en exclusión mutua (`lock`) de un cerrojo y su liberación (`unlock`). Y por último está el grupo de aquellas que interactúan con la gestión de conexiones y que permiten establecer canales de comunicación entre un cliente y un servidor (`accept` y `connect`), su liberación (`release`) así como el envío (`send`) y la recepción (`receive`) de datos.

- **Distribución**

  Esta capa ofrece al programador la posibilidad de realizar invocaciones remotas previo registro tanto del cliente como del servidor. Para ello dispone de primitivas que permiten el registro (`remoteobject_register`) y desregistro (`remoteobject_unregister`) de objetos remotos así como el registro (`stub_register`) y desregistro (`remoteobject_register`) de sustitutos. Y también hay otras dos primitivas asociadas al proceso de invocación remota: una para el cliente (`stub_invoke`) y otra para el servidor (`remoteobject_invoke`) que posibilitan la realización de invocaciones remotas entre cliente y servidor.

- **Servicios comunes**

  Esta capa ofrece dos servicios: uno de recolección distribuida de basura y otro nombres.

  El servicio de recolección distribuida de basura es un servicio interno que no puede ser accedido directamente por el programador y que consta de dos primitivas. La primera es `reference`. Esta primitiva es lanzada cada vez que una referencia a un objeto remoto abandona un nodo RMI y su principal misión es la de informar sobre la creación de una nueva referencia remota a un objeto remoto al nodo correpondiente. La segunda es `unreference`. Esta primitiva es invocada cada vez que en un nodo RMI desaparece una referencia a un objeto remoto con el objeto de informar al algoritmo de recolección distribuida de basura de su destrucción.

  Por otro lado, el servicio de nombres permite gestionar relaciones entre identificadores lógicos y los diferentes objetos remotos del sistema. Consta de tres primitivas: `bind` que la establece, `unbind` que la destruye y, por último, `lookup` que permite la obtención de una referencia a un objeto remoto a partir de un identificador lógico.

Por último, aparecen dos primitivas especiales, una en el middleware de infraestructura y otra en el de distribución denominadas con el sufijo común `manager_set`. La primera es `mid_inf_manager_set` y la segunda es `mid_dis_manager_set`. La de infraestructura está pensada para ganar un cierto grado de control sobre el comportamiento global del middleware de infraestructura y la de distribución hace lo mismo pero actuando sobre el nivel de distribución.

3.1.2. Relación entre las primitivas propuestas y las tecnologías Java

Tal y como sintetiza en la tabla 3.1 es posible establecer relaciones entre cada una de las primitivas definidas por el modelo descrito y las tecnologías Java RTSJ y RMI. Tal y como se observa la única primitiva para la cual no se ha identificado un equivalente es la `mid_dis_manager_set`. Para el resto sí que existe una buena adecuación, existiendo incluso a veces varias alternativas a la hora de dar soporte a una misma primitiva, tal y como ocurre en el caso de la gestión de memoria o en el de la concurrencia.

En el caso del middleware de infraestructura destaca el hecho de que para algunas operaciones como pueden ser la reserva de memoria, la creación de entidades concurrentes o la creación de elementos limitadores de la concurrencia es posible identificar múltiples alternativas. A modo de ejemplo cabe destacar el caso de la reserva de memoria que puede ser realizada tanto de forma especializada por el `new` aplicado sobre la `HeapMemory` o sobre la `ImmortalMemory` o en crudo mediante la instanciación de objetos de tipo `ScopedMemory`. Otro caso interesante es el de la creación y la destrucción de entidades concurrentes que tal y como se muestra en la tabla 3.1 admite múltiples aproximaciones.

Por último en el caso de la capa de middleware de distribución y la de servicios comunes ha sido posible identificar una única acción dentro del middleware RMI, y algunas veces incluso un método concreto, para cada una de las primitivas propuestas.

3.2. Modelo de predictibilidad para RTRMI

Hasta el momento se ha conseguido un modelo más o menos similar al que nos puede proporcionar el modelo computacional de RMI, definiendo primitivas tanto para la capa de distribución como para una posible capa infraestructura basada en RTSJ. A partir de ahora, se particuliza más el modelo introduciendo una serie de restricciones y de entidades especialmente pensadas para el desarrollo de aplicaciones de tiempo real.

A la hora de particularizar el modelo descrito, la idea clave es la de requerir un soporte mínimo al middleware de infraestructura que es después complementado con nuevas características dentro del middleware de distribución. Esta forma de proceder es similar a la seguida en RTCORBA donde la dificultad de que el middleware de infraestructura (típicamente un sistema operativo de tiempo real) pueda ofrecer un soporte eficiente y predecible para la creación de hilos en el servidor hace que esta tarea sea asumida por el *threadpool* de la capa de distribución.

De esa manera, el modelo reparte las responsibilidades de la gestión de tiempo real de los principales recursos -memoria, procesador y conexiones- entre las diferentes capas de tal forma que el middleware de infraestructura asume aquella más básica y el middleware de distribución, mediante la introducción de ciertas entidades -un *memorypool*, un *connectionpool* y un *threadpool*- la de aquellos aspectos más complejos.

3.2.1. Soporte predecible ofrecido por la capa de infraestructura

El middleware de infraestructura, tal y como se puede ver en la figura 3.2, no ofrece el mismo tipo de garantías sobre el comportamiento temporal de todas sus primitivas.

Así, operaciones complejas como son la creación y la destrucción de entidades concurrentes, la de conexiones o la de de elementos de control de la concurrencia no ofrecen ningún tipo de garantía temporal sobre su ejecución. En un principio y dependiendo de la implementación subyacente, su ejecución podría demorarse un tiempo arbitrariamente elevado.

Las únicas primitivas sobre las que el middleware de infraestructura ofrece garantías temporales son aquellas que resultan más imprescindibles. En el modelo propuesto son cinco. Cuatro ya definidas: `-connection_send`, `connection_receive`, `connection_lock` y `connection_unlock-` junto a otra nueva `-concurrententity-_setpriority`[1]- que permite cambiar la prioridad a la que se está ejecutando una entidad concurrente.

En caso de que se utilicen las primitivas que permiten la creación de entidades concurrentes `-mid_inf_concurrententity_create-`, de elementos de sincronización `-mid_inf_concurrencylimitator_create-` o de conexiones `-mid_inf_connection-_create` y `mid_inf_connection_connect-` o de las asociadas a su liberación `-mid-_inf_concurrententity_destroy`, `mid_inf_concurrencylimitator_create` y `mid-_inf_connection_close-` no se tendrá, a priori, ninguna garantía sobre su comportamiento temporal. Dependiendo de la implementación particular del middleware de infraestructura utilizada estás operaciones son más o menos fáciles de predecir y/o eficientes. Y por tanto, su utilización debe de ser controlada de alguna manera desde el plano del programador.

Se podría decir que el soporte ofertado por el middleware de infraestructura es bastante mínimo. Se proporciona soporte predecible a aquellas primitivas que resultan casi imprescindibles para el funcionamiento del middleware de distribución de tiempo real como es el control de la prioridad a la que ejecuta un hilo, el cierre y la liberación de un cerrojo y el envío y la recepción de datos a través de una conexión. Y para el resto de primitivas, como pueden ser la de creación de recursos de comunicación, la de entidades concurrentes o de elementos limitadores de la concurrencia, no se ofrece ningún tipo de garantía temporal sobre su ejecución.

El que el middleware de infraestructura no ofrezca un soporte predecible para el conjunto de primitivas que maneja hace que resulte necesario que el middleware de distribución asuma parte de la gestión realizada por el middleware de infraestructura,

[1] Se puede asociar al método `setPriority` de la clase `Thread`.

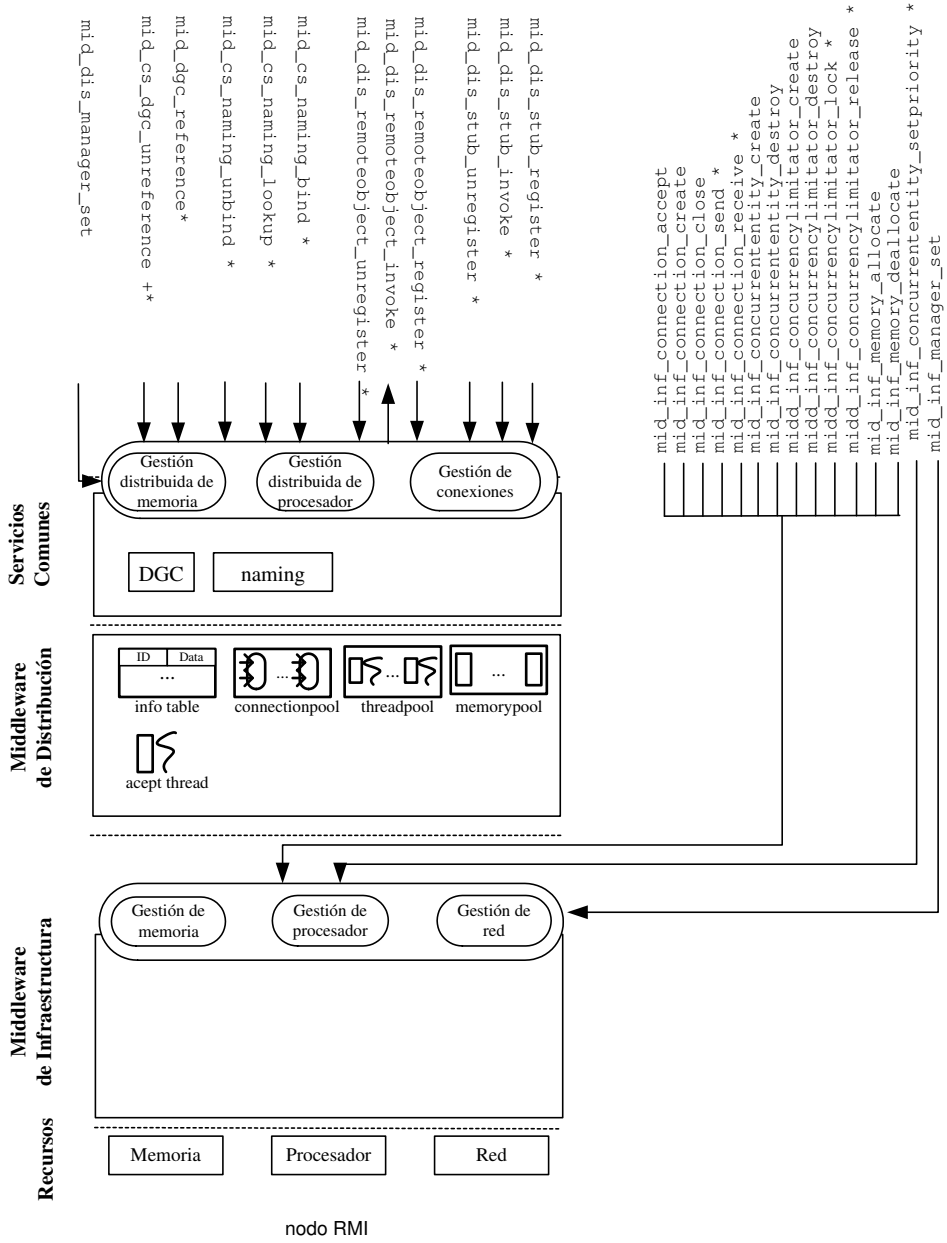


Figura 3.2: Modelo de predictibilidad para RTRMI

mediante la incorporación de nuevas entidades. Éstas participarán de forma activa a la hora de satisfacer diferentes recursos utilizados en la comunicación remota.

3.2.2. Gestión de recursos asumida por la capa de distribución

Observando ahora a la capa de distribución de la figura 3.2 y comparándola con la de la figura 3.1 vemos que aparecen nuevos elementos que antes no estaban presentes. Estos elementos sirven para permitir que la lógica del programador (ya veremos en el siguiente capítulo cómo) decida sobre la gestión de recursos realizada durante las comunicaciones remotas.

Las entidades introducidas por el middleware de distribución, al igual que los tipos de recursos manejados por el middleware de infraestructura, son tres:

- **ConnectionPool** Esta entidad se dedica a la provisión del canal de comunicaciones que es compartido entre el cliente y el servidor. Esta entidad permite controlar cuándo es realizada la creación y la liberación de la conexión así como fijar cuál es el comportamiento que del lado del servidor recibe el canal de comunicación, permitiendo por ejemplo fijar una prioridad inicial de trabajo a la que procesar datos entrantes.

- **MemoryPool** Esta entidad se dedica a la provisión de la memoria temporal requerida por el middleware de distribución para la atención de peticiones que involucran a objetos remotos locales. Al igual que la anterior permite decidir cómo se realiza la gestión de memoria que es requerida de forma dinámica para atender las necesidades del middleware durante las diferentes etapas de la comunicación remota.

- **ThreadPool** Por último, esta entidad se dedica a la provisión de las entidades concurrentes que son requeridas de forma adicional a las provistas por el *connectionpool* para la implementación de mecanismos de asincronía en el servidor. Al igual que las anteriores permite decidir cuándo es realizada la creación y la destrucción de dichas entidades concurrentes así como limitar el grado de asincronismo alcanzable por un servidor.

A efectos prácticos y para el resto del capítulo podemos suponer que la estrategia básica que siguen estos elementos es la prereserva y que existe una única entidad de cada tipo en cada uno de los nodos RMI de cada uno de los tres tipos. En el siguiente capítulo, donde se propondrán interfaces para cada uno de los tres tipos de recursos, se verá que es posible crear múltiples instancias asociables tanto a clientes como a servidores como al propio middleware de distribución.

Una vez se ha fijado el modelo de primitivas y el de las garantías ofrecidas por el modelo al resto del sistema, se procede a explicar el comportamiento interno del middleware. Lo que implica caracterizar más en profundidad tanto a la invocación remota como a los servicios de recolección distribuida de basura y a el de nombres.

3.3. Invocación remota

A la hora de hablar de la invocación remota se tendrán en consideración tres modelos: uno *síncrono* y dos *asíncronos*. El síncrono se caracteriza por que el cliente esperará una respuesta proveniente del servidor. El primero de los asíncronos por no esperar ningún tipo de respuesta del servidor, continuando con su ejecución tras depositar los datos en el nodo local. Y por último, un segundo asíncrono donde el cliente esperará una confirmación de la recepción de los datos de la invocación remota por parte del servidor antes de proseguir con su ejecución.

Para cada uno de estos tres modelos se propondrá un comportamiento interno para el middleware de distribución. En términos prácticos, en todos los casos esto supondrá determinar una serie de cuestiones: de dónde es tomada la *memoria* y las *conexiones* necesarias para realizar la comunicación con el nodo remoto, así como cuál es la *prioridad* de ejecución en cada una de las fases por las que atraviesa la invocación remota. Esta caracterización del comportamiento interno será la que permitirá al desarrollador de aplicaciones distribuidas utilizar los modelos descritos en el estado del arte para el calculo de los tiempos de respuesta máximos de sus aplicaciones.

3.3.1. Invocación remota síncrona

En la invocación remota síncrona el cliente se bloquea a la espera de una respuesta del servidor y no continúa con su ejecución hasta que no son recibidos los resultados del método remoto, estableciéndose una secuenciación lógica entre la ejecución del cliente y la del servidor.

La figura 3.3 nos muestra el proceso de forma gráfica. En ella se ha dividido la invocación remota en siete etapas que comienzan -en 1- con la cesión del control por parte de la lógica del desarrollador a la del middleware de distribución, proceso que es seguido por: -en 2- el procesado y envío de datos del cliente al servidor, -en 3- el procesado en el servidor de la invocación remota entrante, -en 4- la ejecución de la lógica del programador (otra vez en el plano del desarrollador), -en 5- el envío del resultado de la invocación remota al cliente, -en 6- el procesado de los resultados en el cliente y, por último, la devolución del control -en 7- al plano del programador.

A la hora de acceder a los recursos básicos del sistema (memoria, procesador y capacidad de comunicación) necesarios para realizar la invocación remota en el nodo cliente, el modo de proceder es el siguiente:

- *Comunicación.* La conexión necesaria para realizar la invocación remota se toma de un *connectionpool* local -en 2.1- y al finalizar la lectura de los datos provenientes del servidor se devuelve a dicha entidad -en 6.1. La forma en que internamente el *connectionpool* provee este tipo de entidad es dependiente del tipo de *connectionpool* utilizado. Para la recepción de datos por el canal -en 6- se utiliza la primitiva `connection_receive` y para el envío de resultados -en 2- la `connection_send`.

- *Procesamiento.* La entidad concurrente con la que se ejecuta la invocación remota es la hebra con la que se invoca el método del sustituto: *client*. Y la

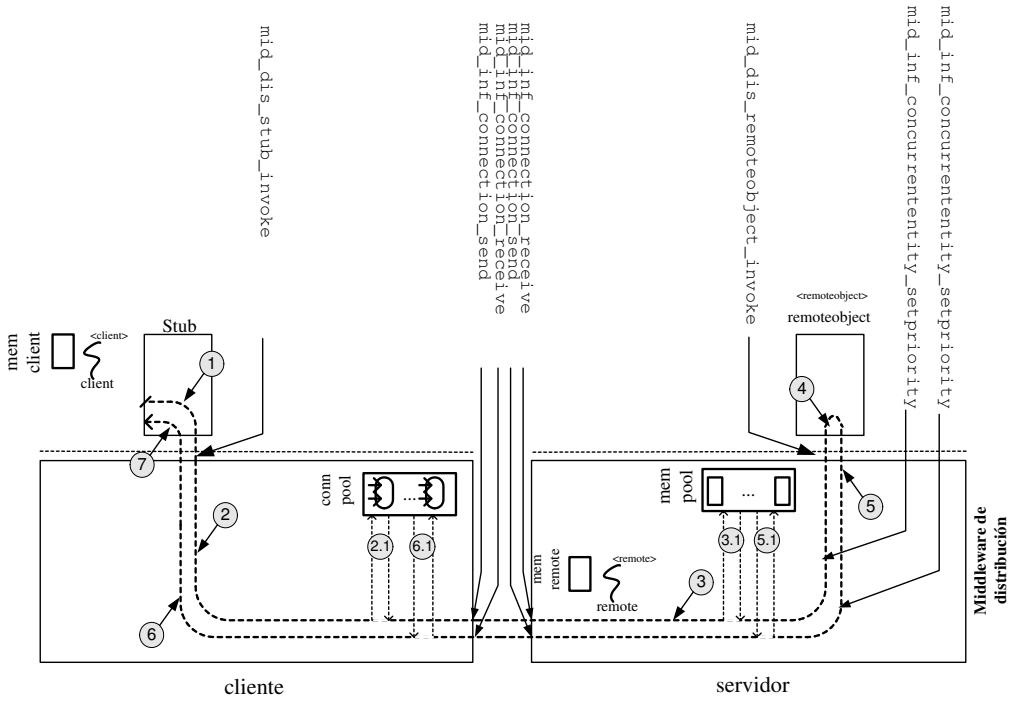


Figura 3.3: Invocación remota síncrona

prioridad $^{<client>}$ [2] a la que ejecuta internamente el middleware de distribución es la misma a la que se encontraba trabajando esa hebra cuando realiza la invocación sobre el sustituto.

El acceso a estructuras de datos compartidas como pudiesen ser el *connpool* u otras tablas internas que contienen información sobre el conjunto de objetos remotos y sustitutos que han sido creados se hace utilizando un protocolo de control de la inversión de la prioridad y las primitivas `lock` y `unlock`.[3]

- *Memoria.* La memoria que de forma temporal es requerida para la realización de la invocación remota, utilizada para serializar -en 2- los datos que son enviados al servidor así como para deserializar los que son devueltos por el servidor -en 6- la proporciona la lógica del programador. Ésta habrá de proveer un bloque de memoria lo suficientemente grande para contener todos los objetos creados de forma temporal durante la invocación. Tras la finalización de la invocación remota esta memoria será retornada -en 7- a la aplicación que podrá liberarla si así lo desea.

Y en el nodo servidor el funcionamiento de la invocación remota es tal y como sigue:

[2] La notación $^{<client>}$ significa prioridad a la que ejecuta la entidad concurrente *client*.
[3] En la figura no se incluye dicha tabla para no entorpecer la comprensión del modelo.

- *Comunicación.* Para la lectura de datos y el envío del resultado de la invocación remota al cliente se utiliza la conexión establecida por el cliente. Esta conexión dispone de una hebra asociada -*remote*- que se encuentra a la espera de datos provenientes del cliente. Para la recepción de datos enviados por el canal -en 3- se utiliza la primitiva `connection_receive` y para el envío de resultados -en 5- la `connection_send`.

- *Procesamiento.* Para el procesado de los datos que llegan por la conexión se utiliza la hebra, *remote*, que es creada cada vez que se establece la conexión. Esta hebra ejecuta con un modelo de dos prioridades[4]: (1) $^{<remota>}$ mientras se está a la espera de datos del cliente y cuando se procesa la cabecera del protocolo de comunicaciones y (2) una prioridad definida por el objeto remoto $^{<remoteobject>}$ o la misma del cliente $^{<client>}$ cuando se produce la invocación del objeto remoto -en 4.

  La asignación de estas prioridades, tal y como es lógico, dependerá de las características de la aplicación distribuida de tiempo real desarrollada y deberá de ser determinada por el desarrollador de aplicaciones distribuidas. Aunque no es estrictamente necesario, típicamente se cumplirá que la prioridad de procesado utilizada será mayor que la que se utilizará para ejecutar la lógica del objeto remoto, a fin de reducir lo que es la inversión de prioridad experimentada extremo a extremo.

  Para cambiar la prioridad a la que se ejecuta el servidor se utiliza `setpriority` dos veces: justo antes de invocar al objeto remoto y justo después, tras enviar los resultados de la invocación remota por el canal de comunicaciones. Por último, al igual que en el cliente, el acceso a estructuras de datos compartidas como pudiese ser el *connpool* se hace utilizando un protocolo de control de la inversión de la prioridad y las primitivas `lock` y `unlock`.[5]

- *Memoria.* En el lado del servidor el middleware dispone de dos bloques de memoria. El primero de ellos es uno privado *memremote* y aparece asociado a cada una de las entidades concurrentes *remote*. Es de utilidad a la hora de obtener de forma dinámica la memoria necesaria para realizar el procesado de las cabeceras del protocolo de comunicaciones. El segundo de ellos es tomado del *mempool*. Es utilizado para deserializar los datos enviados desde el cliente -en 3- y para serializar -en 5- los resultados de la invocación remota que son retornados al cliente. Este bloque de memoria se devuelve -en 5.1- al *mempool* una vez se han enviado los datos al cliente mediante la primitiva `connection_send`.

Desde el punto de vista del código del método remoto -en 4- al comienzo del método remoto la situación con la que se encuentra es la que sigue. La lógica del método remoto dispone de un bloque de memoria tomado del *mempool*, de una en-

[4]Este esquema es similar al existente en muchas implementaciones del modelo de prioridades propagado y definido por el servidor de RTCORBA, como es el caso de TAO.

[5]En la figura no se incluye ningún tipo de tabla para no entorpecer la comprensión del funcionamiento del modelo.

tidad concurrente *remote* prestada por el middleware de distribución cuya prioridad de ejecución es o bien $^{<cliente>}$ o $^{<remoteobject>}$.

Dos hechos son de resaltar en este modelo. El primero de ellos es que el modelo no hace uso de aquellas primitivas del middleware de infraestructura que no ofrecen garantías sobre su ejecución. Para evitar la creación de conexiones y de entidades concurrentes se recurre a las entidades de gestión de recursos presentes en el middleware de distribución. Y el segundo es que en todo momento se sabe la gestión que se está haciendo de los recursos tanto en el cliente como el servidor, lo que permitiría aplicar las técnicas de planificación distribuida al modelo de invocación remota desarrollado.

Sin embargo, la mayor limitación de este modelo es que en muchas aplicaciones, sobre todo en aquellas ligadas a lo que podría ser la realización de una señalización remota de la cual no se espera ningún tipo de respuesta o aquellas donde la lógica del método remoto consume demasiado tiempo, se puede estar esperando en el cliente de forma innecesaria a que finalice la ejecución en el nodo servidor.

A fin de mitigar este problema se propone reducir este tiempo de espera soportando dos fórmulas de asincronismo, una que elimina las latencias asociadas a la comunicación y a la ejecución remota y otra que elimina las introducidas por la ejecución de la lógica del método remoto.

3.3.2. Invocación remota asíncrona

A diferencia del modelo síncrono, en el asíncrono no se espera por ningún resultado proveniente del servidor, sino que se desliga la ejecución de la lógica del cliente de la del servidor en las primeras etapas de la invocación remota. Veamos cuáles son las diferencias que introduce cuando se compara con el caso síncrono presentado en la sección 3.3.1.

La figura 3.4 muestra cómo funciona la invocación remota asíncrona y cómo se van combinando los diferentes recursos a la hora de darle soporte.

Al igual que antes se pueden distinguir siete etapas pero su ejecución ya no es secuencial. Al igual que en el caso síncrono se comienza pasando el control de la capa de aplicación a la de distribución -en 1- para luego enviar la información de la invocación -usando `connection_send`- al servidor. Tras ello se paraleliza la ejecución. Si no ha habido ningún problema en el depósito de datos, el cliente -en 6- retorna el control -en 7- a la lógica del programador. Y del otro lado, -en la fase 3- el servidor procesa los datos que habían sido enviados, invocando a la lógica del objeto remoto -en 4. Después de ello, -en 5- los recursos de computación que habían sido utilizados para satisfacer la invocación remota son liberados a la espera de una nueva invocación remota.

La política seguida en cuanto a la gestión de recursos realizada en el cliente no presenta grandes diferencias entre el caso síncrono y el asíncrono. La memoria *memclient*, la entidad concurrente *client* y la prioridad utilizada en la ejecución de la lógica del cliente -$^{<client>}$- son definidas por el plano del programador y las conexiones necesarias para realizar la comunicación son tomadas de la entidad *connpool*. La principal diferencia es que en el cliente, en el asincronismo, no se hace uso de la

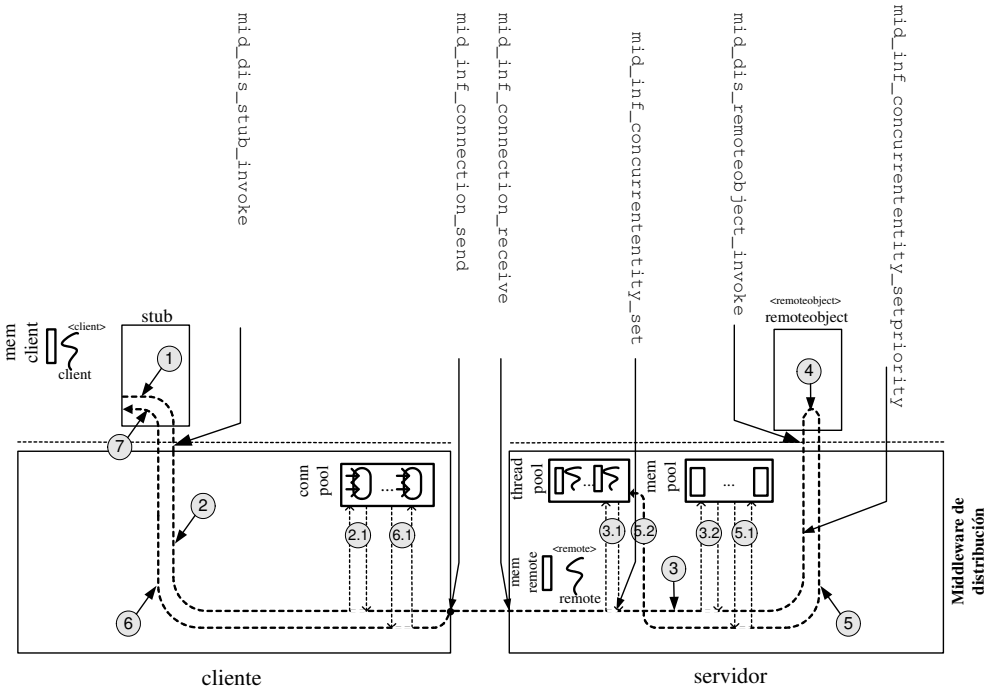


Figura 3.4: Invocación remota asíncrona

primitiva `receive` para recibir ningún tipo de información proveniente del servidor.

Sin embargo la gestión en el servidor presenta ya más cambios pues la política de tener una hebra dedicada a atender las peticiones de forma síncrona ya no es tan válida como en el caso síncrono. Resulta necesario introducir algún tipo de mecanismo que permita desacoplar la ejecución del cliente de la del servidor. Este elemento es el *threadpool*.

Así, la entidad concurrente *remote* cuando comienza a procesar la invocación remota y se da cuenta de que es una asíncrona procede a ir a buscar una entidad concurrente sustituta al *threadpool* -en la fase 3.1. La prioridad a la que permanecerá escuchando esta nueva entidad concurrente será cambiada a $^{<remote>}$ mediante empleo de la primitiva `setpriority`. Lo que le permitirá estar a la espera de otra petición proveniente del cliente. Y tras haber encontrado una hebra sustituta, la entidad concurrente que va a invocar al objeto remoto, ya podrá tomar el bloque de memoria auxiliar del *connectionpool* -en 3.2- así como mudar su prioridad de ejecución con `setpriority` (a bien $^{<client>}$ o a $^{<remoteobject>}$) antes de proceder a invocar al objeto remoto. Por último, tras la invocación al objeto remoto, en vez de volver a escuchar la conexión, la entidad concurrente es retornada al *threadpool* -en 5.2. [6]

Un problema que presenta esta fórmula de asincronismo es que a veces es demasiado insegura pues no provee ningún tipo de mecanismo que permita detectar problemas en el nodo servidor. Por ello, en el modelo desarrollado se contempla la existencia de una segunda forma de asincronismo, confirmada por el servidor, que

[6]Este patrón recibe, siguiendo la terminología utilizada en [2], el nombre de *leader-follower*.

ofrece mayores garantías al cliente sobre lo que ocurre en la parte del servidor, a cambio de reducir el grado de asincronismo máximo alcanzable.

3.3.3. Invocación remota asíncrona con confirmación del servidor

Esta fórmula de asincronismo reduce la capacidad paralelizadora del asincronísmo visto en la sección anterior haciendo que el cliente espere una confirmación proveniente del servidor, lo que ofrece garantías adicionales sobre el estado de la ejecución remota al cliente a costa de reducir el grado de paralelismo alcanzable. Al igual que en el caso anterior, la descripción realizada se limitará a ver los cambios que introduce este tipo de solución en la invocación asíncrona ya estudiada en la sección 3.3.2.

La figura 3.5 nos muestra el esquema de cómo funciona una invocación remota con confirmación por parte del servidor. La principal diferencia existente entre las dos fórmulas de asincronismo es que en la segunda se utiliza el canal de comunicaciones para retornar al cliente una confirmación proveniente del servidor. Lo que permite realizar ciertas comprobaciones -en 3.1- sobre el estado del sistema (sobre si hay suficientes recursos o si el protocolo de comunicaciones utilizado ha sido el correcto).

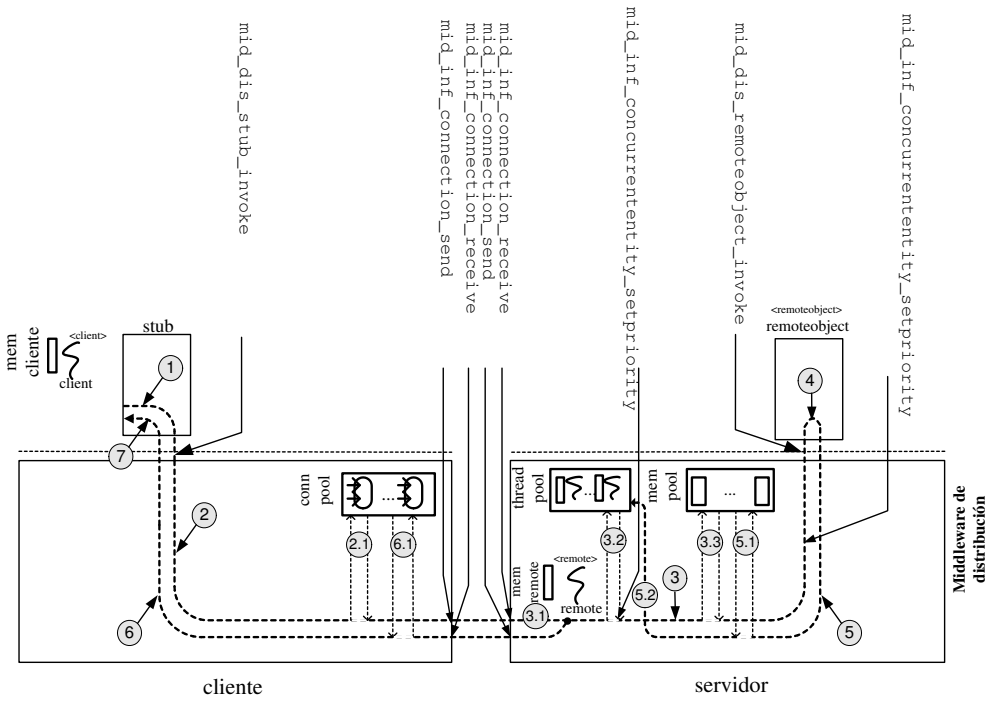


Figura 3.5: Invocación remota asíncrona con confirmación del servidor

En el cliente el funcionamiento es igual al de la invocación asíncrona, con la única salvedad de que se espera en el cliente -en 6- una confirmación del servidor haciendo uso de `receive`.

En el servidor el funcionamiento es igual al de la la invocación asíncrona con la salvedad de que antes -en 3.1- de proceder a obtener una hebra sustituta -en 5.2-, se

procede a enviar, haciendo uso de `send`, un mensaje al cliente indicándole que va a resultar posible realizar la invocación remota asíncrona.

El modelo de asincronía desarrollado excluye la inclusión de mecanismos que nos permitan la devolución de resultados desde el cliente al servidor[7]. El principal motivo para tomar esta decisión ha sido el de mantener la implementación del middleware de distribución relativamente sencilla de realizar. Y como consecuencia de ello, la lógica del programa que durante la ejecución de un método remoto asíncrono desee devolver datos desde el servidor al cliente tendrá que utilizar otro método remoto.

Tras haber caracterizado el funcionamiento de la invocación remota tanto de forma síncrona como asíncrona es la hora de abordar cómo son gestionados los recursos tanto durante el proceso de recolección distribuida de basura como a la hora de obtener referencias a objetos remotos a partir de identificadores lógicos. Empezaremos viendo cómo es integrado el servicio de recolección distribuida de basura -sección 3.4- para luego -sección 3.5- continuar con el de nombres.

3.4. Integración del recolector distribuido de basura

Cada uno de los nodos del modelo posee un servicio de recolección de basura distribuido (DGC). Éste es el encargado de mantener un esquema de referencias a objetos remotos de forma consistente, sirviendo de nexo entre la recolección de basura local realizada en cada nodo y una global donde participan las referencias remotas. Tiene por misión evitar la existencia de situaciones en las cuales existan referencias a objetos remotos inexistentes o que se siga manteniendo un objeto remoto cuando éste ya no es accesible ni remota ni localmente. Para ello lo que hace es mantener referencias a los objetos remotos almacenados en un nodo mientras existan referencias remotas en otros. Pero para conseguir mantener esta lógica necesita intercambiar, haciendo uso de las facilidades provistas por el middleware de distribución, mensajes entre los diferentes nodos de la red con información relativa a la creación y a la destrucción de referencias a objetos remotos. En esta sección se caracteriza un soporte para un algoritmo de recolección de basura distribuida síncrono, implementable con técnicas basadas en contaje.

Aunque resulta posible implementar múltiples esquemas de recolección de basura distribuida, con soporte incluso a la tolerancia de fallos, se ha optado por un esquema sencillo de implementar pues características como la tolerancia a fallos en tiempo real requieren de un tratamiento específico. El modelo escogido es quizás uno de los más sencillos de implementar y está basado en la técnica de recolección de basura síncrona basada en contaje. Este modelo es síncrono con el de gestión de memoria local y está basado en el conocimiento de dos instantes temporales bien definidos: (1) el instante en que una referencia a un objeto remoto abandona un nodo local (para ser creada en otro remoto) y (2) cuando ésta es destruida. Esos dos procesos disparan la ejecución de dos primitivas: (1) `reference` y (2) `unreference`, que comunican de forma síncrona a los correspondientes nodos remotos tales eventos.

Veamos pues cómo el middleware de distribución les proporciona soporte.

[7]El modelo de asincronía de RTCORBA si que los incorpora pero su nivel de utilización es mas bien bajo

3.4.1. Abandono de una referencia remota del nodo local

La figura 3.6 nos ilustra el funcionamiento de la primitiva `reference`, mostrándonos cómo la transmisión de una referencia de un nodo remoto a otro hace que el middleware de forma automática establezca una comunicación con el modo remoto donde reside dicho objeto remoto antes de su transmisión. En el ejemplo mostrado, el nodo cliente intenta transmitir al nodo *servidor* una referencia al objeto remoto $remoteobject_A$, residente en el nodo A. Internamente, el middleware de distribución, antes de transmitir dicha referencia, se lo notifica al algoritmo de recolección de basura remoto (DGC) del nodo A.

A la hora de gestionar los recursos la política seguida por el middleware de distribución es similar a la que se sigue en una invocación remota síncrona. En el nodo local donde reside la referencia remota, los recursos necesarios para notificar el abandono de la referencia los provee la hebra que intenta transmitir la referencia remota por el canal de comunicaciones. Y en el nodo donde reside el algoritmo de recolección distribuida de basura la política seguida es la de utilizar un esquema de dos prioridades para gestionar el procesador y dos bloques de memoria (uno propio y otro del *connectionpool*) para la provisión de la memoria que es requerida de forma dinámica.

En la parte superior-izquierda de la figura, donde la referencia trata de salir de un nodo cliente, el middleware ejecuta con prioridad $^{<client>}$ y cuenta con la memoria -*memclient*- del hilo que está intentando transmitir la referencia remota y la capacidad de comunicación proporcionada por el *connpool* local para comunicarse con el nodo A.

Así, antes de transmitir la referencia al nodo servidor -en 2-, establecerá una comunicación -en 2'- con el nodo A, esperando a una respuesta proveniente de éste -en 6'- antes de transmitir la referencia remota al nodo servidor. Esto le garantizará al cliente que el objeto remoto $remoteobject_A$ no será destruido antes de tiempo pues así se lo garantizará el recolector de basura remoto. De no hacerlo así podría darse el caso de que se eliminase prematuramente $remoteobject_A$.

Y en el nodo donde reside el algoritmo de recolección distribuida de basura (DGC) del nodo A, los recursos se gestionan como si fuesen los de una invocación remota síncrona ordinaria. Al igual que en una invocación remota síncrona, existe una entidad concurrente $remote_A$ que está a la espera de datos escuchando la conexión a prioridad $^{<remote_A>}$. Tras procesar la invocación remota entrante esa prioridad cambia, haciendo uso de `setpriority` a bien la del cliente $^{<client>}$ o a bien una definida para el algoritmo de recolección de basura $^{<dgc_A>}$. Con esta prioridad -en 4'- se invoca al algoritmo de recolección de basura. Su lógica interna es tan sencilla como crear una referencia local al objeto $remoteobject_A$. Y tras ello, le comunica al cliente -en 5'- que el proceso se ha realizado correctamente. Por último, la hebra retorna el bloque de memoria que había tomado y vuelve a escuchar la conexión, restaurando su prioridad a $^{<remote_A>}$.

Al igual que en el caso de la asignación de prioridades al proceso de invocación remota, determinar qué esquema de prioridades se ha de utilizar guarda una fuerte relación con la aplicación desarrollada, no pudiendo presumirse ningún tipo de comportamiento. Pero por lo general y a fin de reducir la inversión de prioridad que la recolección distribuida de basura introduce en las aplicaciones, una primera aproxi-

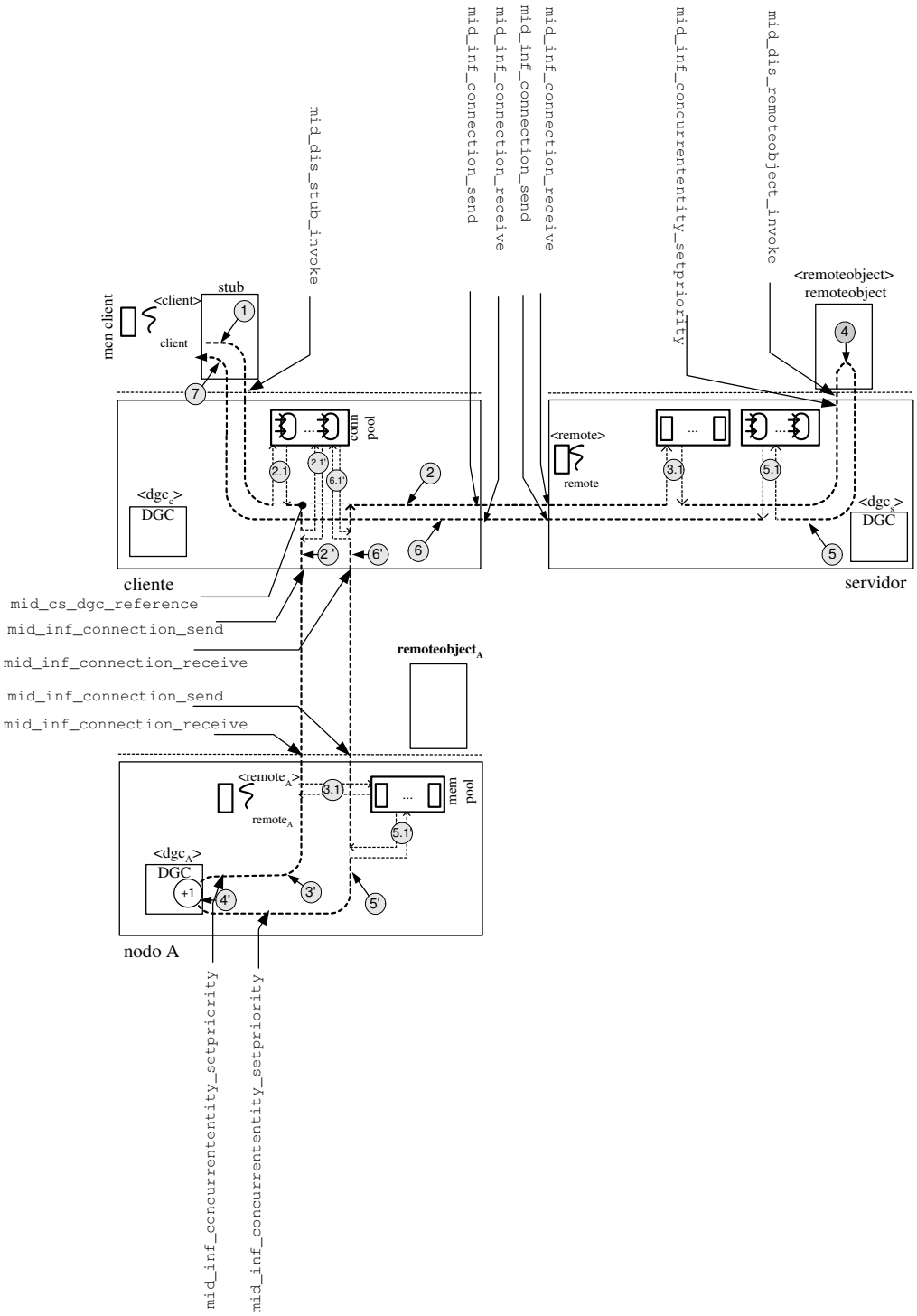


Figura 3.6: Abandono del nodo local de una referencia a un objeto remoto

mación podría ser la de hacer que el algoritmo de recolección distribuida de basura sea ejecutado a una prioridad baja, que no interfiera con las del resto de aplicaciones de tiempo real.

3.4.2. Destrucción de una referencia remota

Cada vez que se destruye una referencia remota, en aras de mantener el esquema de referencias distribuidas funcionando correctamente, dicho evento ha de ser convenientemente notificado por el middleware de distribución al nodo correspondiente, disparando el procesado de la primitiva `unreference`. En esta sección veremos cómo el middleware internamente gestiona los recursos que están involucrados en la atención de dicha primitiva.

Al igual que en el caso del abandono de una referencia remota local, los recursos serán gestionados de una forma similar a la realizada en la invocación remota síncrona. Para ejemplificarlo tomaremos la figura 3.7 como punto de partida. En ella se muestran los pasos que se dan en la destrucción de una referencia remota, residente en el nodo cliente, al objeto remoto $remoteobject_B$, residente en el nodo B.

En cualquiera de los casos, cuando se produce esa destrucción, existe una entidad concurrente que está ejecutando a cierta prioridad y que posee cierta cantidad de memoria cuyos recursos computacionales le son tomados temporalmente para comunicar tal operación al nodo remoto donde reside el objeto remoto. La prioridad a la que ejecutará el nodo local es $^{<client>}$ y la memoria de la que hará uso es *memclient*. El recurso de comunicación que utilizará para comunicarse con el nodo remoto será uno tomado del *connpool* local -en 2.1"'- que será devuelto más tarde -en 6.1"'.

En el nodo donde reside el recolector distribuido de basura, la forma de proceder es similar a realizada en la invocación remota síncrona. Existe una entidad concurrente -*remote*- asociada a la conexión y en una primera fase la prioridad a la que ejecuta es $^{<remote>}$ y la memoria de la que dispone es un bloque *memremote*. Y tras esa primera fase se toma un bloque de memoria de *mempool* para realizar un procesamiento de los datos enviados desde el nodo cliente, mudándose la prioridad de la entidad concurrente a bien la definida por el cliente $^{<client>}$ o bien a otra $^{<dgc_B>}$ definida para el proceso de recolección de basura.

Con esos recursos se ejecutará el algoritmo de recolección distribuida de basura -4"'. Éste es tan sencillo como destruir la referencia local que había sido creada durante el procesado de la primitiva `reference`. Y tras ello, los recursos que habían sido tomados para realizar la invocación remota son restaurados, la prioridad de la hebra se cambiará -en 3"'- a $^{<remote>}$ con `setpriority`, el bloque de memoria será devuelto -en 5.1"'- y la entidad concurrente quedará lista a la espera de nuevas peticiones entrantes.

Al igual que en el caso de la primitiva `reference` existe cierto grado de flexibilidad a la hora de definir la prioridad a la que se ejecuta el algoritmo de recolección de basura. Y una buena recomendación inicial sería fijarla de tal manera que no interfiera con los plazos de tiempo real del resto de las tareas.

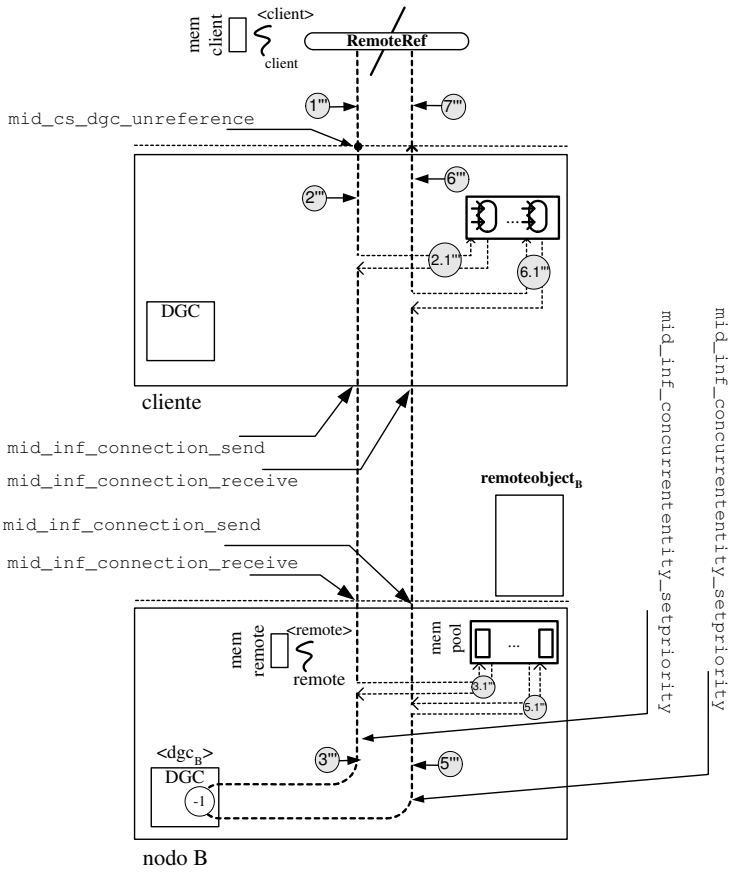


Figura 3.7: Destrucción de una referencia remota

3.5. Integración del servicio de nombres

Y por último, estaría el servicio de nombres. Este servicio no se trata más que de un objeto remoto con interfaz bien conocida que permite realizar la búsqueda de objetos remotos a partir de identificadores lógicos, estableciendo y gestionando pares cadena-referencia_remota. La gestión de recursos realizada sigue la misma política que la utilizada en la invocación remota síncrona. Y la mayor peculiaridad que tiene es que cada una de las tres primitivas que oferta (`bind`, `unbind` y `lookup`), de una forma ligeramente distinta, implican interacciones con el servicio de recolección distribuida de basura. Y por este motivo describimos su funcionamiento brevemente, haciendo hincapié en ver las relaciones existentes entre dichas primitivas y las del servicio de recolección distribuida de basura.

3.5.1. Establecimiento de una relación lógica entre un identificador y una referencia remota

La figura 3.8 muestra el funcionamiento de la primitiva `bind`. Esta primitiva es la encargada de asociar un nombre lógico a una referencia remota. En el ejemplo se intenta fijar un identificador lógico para el $objetoremoto_A$.

Al igual que sucede en el caso de la invocación remota, los recursos para realizar este tipo de operación en el nodo local los provee la entidad concurrente que inicia la operación. En el caso del ejemplo, la prioridad a la que se ejecutará la lógica de la primitiva `bind` será $^{<client>}$, la entidad concurrente utilizada para la invocación será *client* y el bloque de memoria que se utilizará par la creación dinámica de objetos será *memclient*, mientras que la conexión será tomada del *conpool* del nodo local.

La peculiaridad que presenta es que de forma implícita se ha de interactuar con el de recolección de basura pues la referencia al objeto remoto tiene que abandonar el nodo local en dirección al nodo remoto donde reside el servicio de nombres (Naming). Para notificar de esto al nodo A donde reside el algoritmo de recolección distribuida de basura -en la fase 2'y la 6"- se utilizará la prioridad $^{<client>}$, el bloque de memoria *memclient* y una conexión tomada -en 2.1'- del *connectionpool* local. Ya en el nodo A, para su ejecución, el algoritmo de recolección distribuida de basura, dispone de la hebra que se encuentra escuchando la conexión entrante a una prioridad inicial $^{<remote_a>}$ y de los bloques de memoria $memremote_a$ y del proporcionado por el *memorypool* local.

Por último, estaría el nodo remoto donde se hospeda el servicio de nombres. En este nodo la gestión de recursos sería equivalente a la realizada en una invocación remota síncrona. La prioridad inicial a la que trabajaría el hilo que atiende la conexión sería $^{<remote>}$, la cual cambiaría de valor a bien $^{<naming>}$ o a $^{<client>}$ justo antes de invocar la lógica del servicio de nombres. Y la memoria utilizada para realizar la invocación sería la de la propia entidad concurrente *remote* junto a la obtenida del *mempool*. Tras invocar al servicio de nombres, la memoria tomada del *mempool* se retornaría a éste y la entidad concurrente volvería a escuchar la conexión a la espera de nuevas peticiones entrantes, no sin antes volver a restaurar su prioridad de trabajo a $^{<remote>}$.

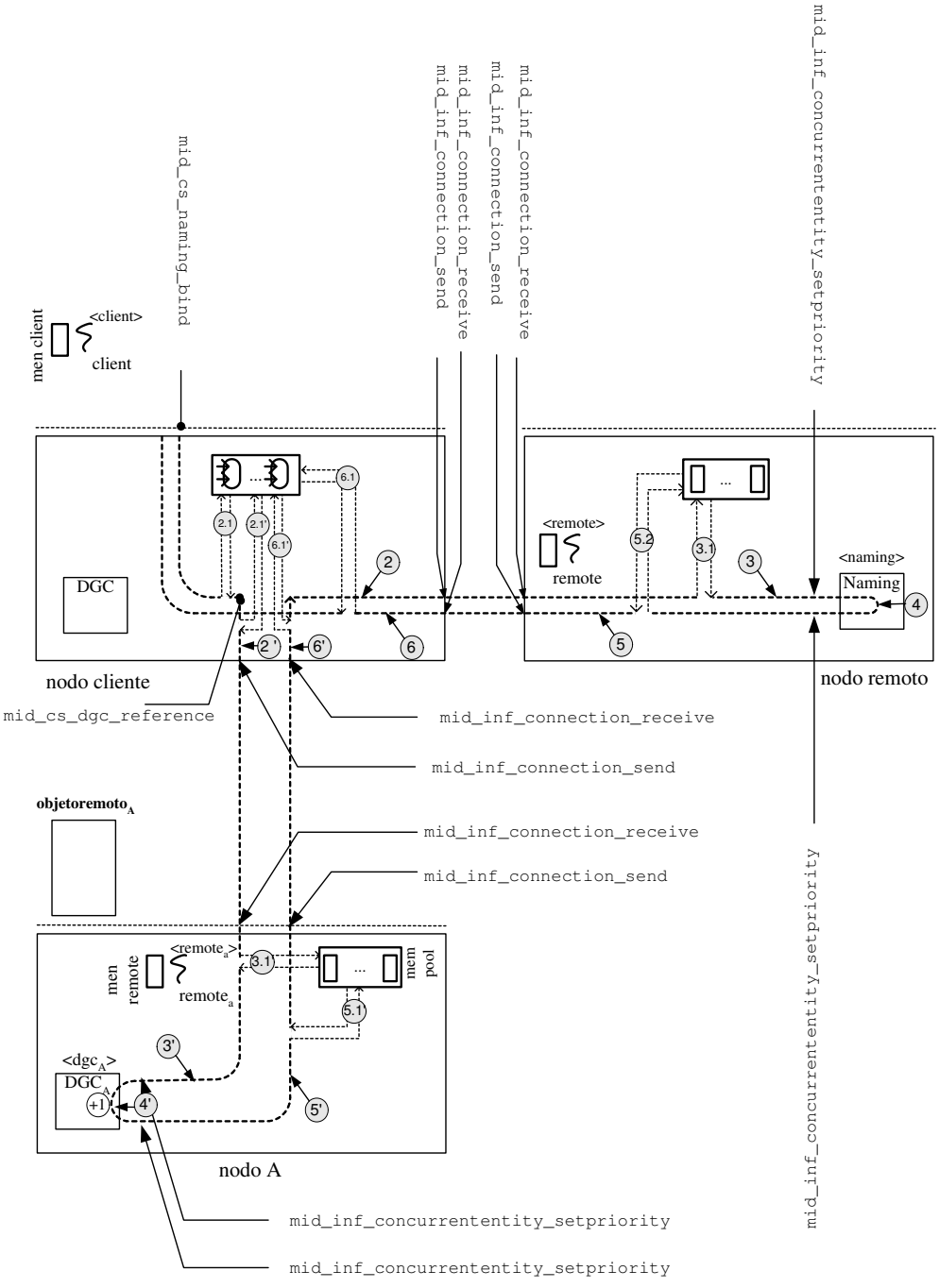


Figura 3.8: Soporte para la primitiva `bind`

Internamente, la lógica del servicio de nombres puede ser tan sencilla como tener una tabla que contenga pares identificador lógico y referencia a objeto remoto.

3.5.2. Destrucción de una relación lógica entre un identificador y una referencia remota

En el caso de la destrucción de una relación lógica entre un identificador y una referencia remota, es el nodo que hospeda al servicio de nombres el que tiene que ponerse en contacto con el servicio de recolección de basura a fin de notificarle que una referencia a un objeto remoto que poseía ha desaparecido.

La figura 3.9 nos muestra el cómo se gestionan los recursos en el caso de que una entidad concurrente, situada en un nodo cliente, decida destruir la relación lógica que había sido establecida con un `bind`, mediante un `unbind`.

Siguiendo la norma básica de la invocación remota síncrona, los recursos necesarios en el nodo que llama a la primitiva `unbind` son tomados de la entidad concurrente que inicia la invocación. En la figura, la prioridad a la que ejecutará el hilo encargado de realizar la invocación remota es $^{<client>}$. La memoria que utilizará para comunicarse con el nodo remoto es *memclient*. Y la conexión utilizada para establecer la comunicación será tomada del *connectionpool*.

Y en el lado del servidor el comportamiento es similar al de la invocación remota síncrona. La única diferencia es que la destrucción de esta referencia implicar realizar invocaciones al recolector de basura del nodo -DGC_A- donde reside el objeto remoto que se pretende destruir. En este caso, los recursos con los que contará el nodo remoto son proporcionados por la hebra que realiza la invocación sobre el servicio de nombres. Y así, la prioridad a la que ejecutará dicha lógica será o bien $^{<naming>}$ o $^{<client>}$. Y por último, la conexión necesaria para comunicarse con el nodo remoto será tomada del *connectionpool* local.

Internamente, la lógica interna del servicio puede ser tan sencilla como eliminar una entrada de una tabla almacenada en el servicio de nombres.

3.5.3. Obtención de una referencia remota a través de su identificador

Y por último estaría la operación que permite buscar un objeto remoto a partir de su identificador lógico, operación que en caso de ser exitosa devuelve una referencia a un objeto remoto al nodo que la invoca.

La figura 3.10 muestra cómo se gestionan los recursos durante una llamada a la primitiva `lookup`.

La gestión de recursos realizada durante esta primitiva en el nodo cliente es similar a la de la invocación remota síncrona. La entidad concurrente utilizada es *client*, la prioridad de ejecución $^{<client>}$ y el bloque de memoria utilizado es *memclient*. Y por último, la conexión necesaria para invocar al servicio de nombres se toma del *connpool*.

La gestión en el nodo del servidor es similar a la realizada en el caso del resto de invocaciones remotas síncronas y la única reseña realizable es la de que cuando los resultados de la operación son devueltos al nodo cliente -en 5-, se ha de establecer

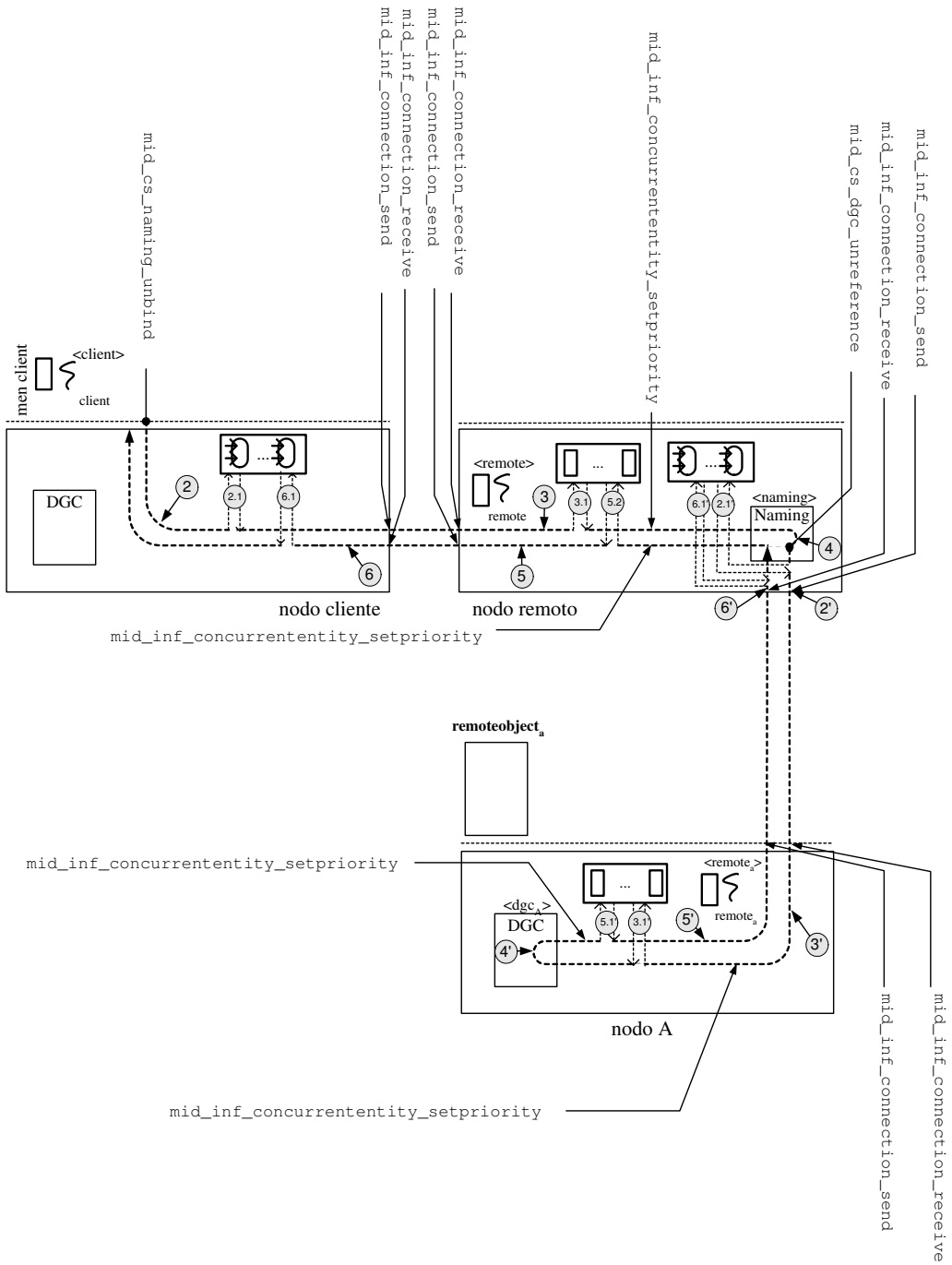


Figura 3.9: Soporte para la primitiva `unbind`

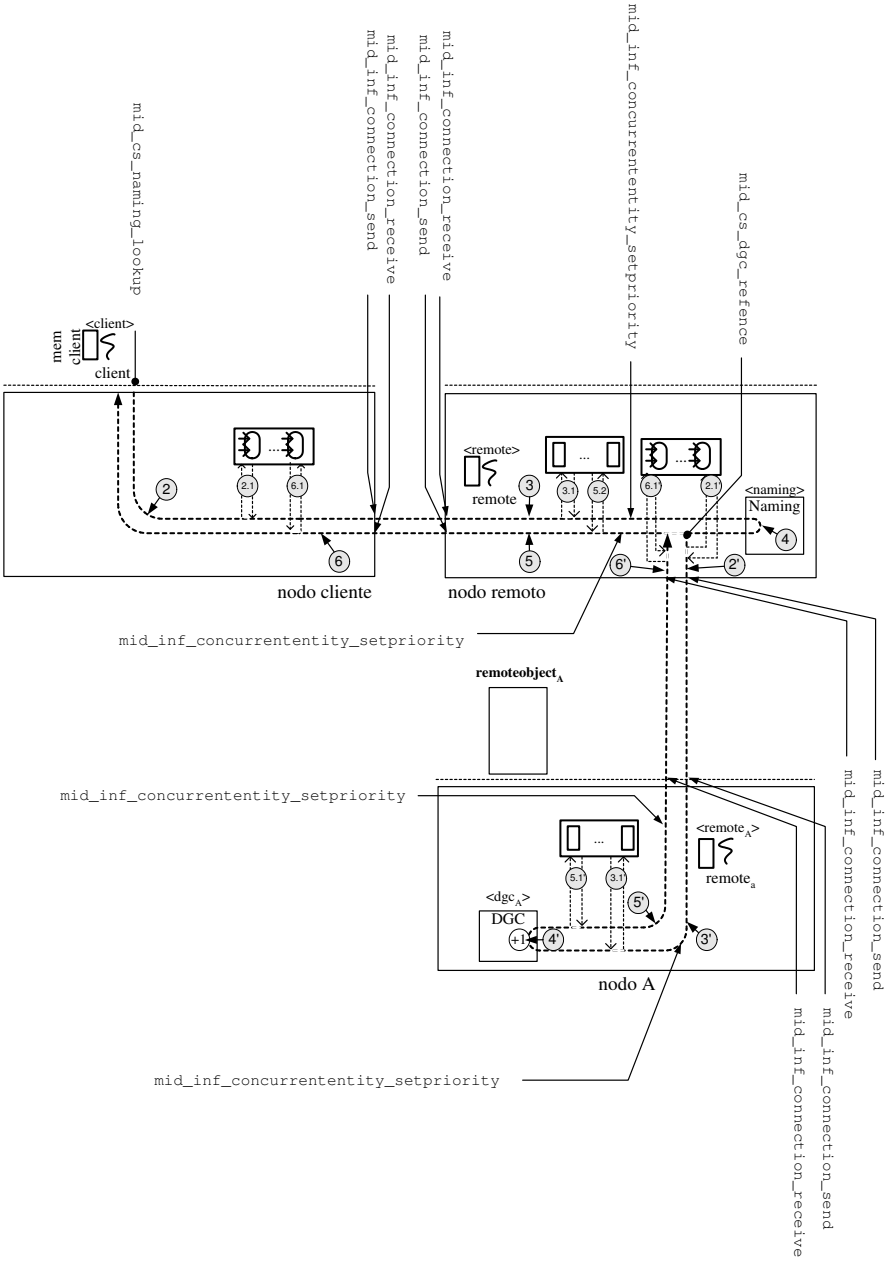


Figura 3.10: Soporte para la primitiva `lookup`

previamente una comunicación con el nodo remoto donde se hospedan los objetos remotos -2" y 6"- a fin de comunicarles la creación de una nueva referencia a un objeto remoto. Tras haber recibido del nodo que hospeda dicho objeto remoto -nodo A- la confirmación de que todo ha ido bien, se continúa enviando los resultados del proceso de búsqueda al cliente.

3.6. Conclusiones

En este capítulo se ha caracterizado un modelo para un middleware de distribución de tiempo real basado en RMI. Éste caracteriza un comportamiento interno para tanto para el proceso de invocación remota síncrona como asíncrona así como para un servicio de recolección distribuida de basura y otro de nombres. El modelo exige que el middleware de infraestructura proporcione un soporte mínimo predecible tanto a la gestión de prioridades, como a la utilización de elementos limitadores de la concurrencia así como al envío y la recepción de datos a través de una conexión. Y a cambio, el middleware de distribución asume el control sobre el establecimiento de las conexiones como sobre la creación de entidades concurrentes como sobre la provisión de memoria de forma dinámica mediante nuevas entidades en el middleware de distribución. Y a partir de este punto se caracteriza cómo funciona internamente el middleware, de tal manera que las técnicas del estado del arte relacionadas con la gestión de recursos de tiempo real son aplicables al modelo.

Si se compara el modelo desarrollado con el modelo RTCORBA se pueden observar dos diferencias clave: que en el propuesto no aparecen elementos arquitectónicos de RTCORBA como son el ORB o el POA y que se incorpora soporte para el modelo de referencias remotas de RMI mediante un recolector distribuido de basura y un servicio de nombres de tiempo real. La gestión de recursos realizada en RTCORBA por el POA y el ORB de tiempo real pasa a ser realizada en el modelo propuesto por tres nuevas entidades: *connectionpool*, *threadpool* y *memorypool* que se encargan de la provisión de recursos (conexiones, entidades concurrentes y memoria) requeridos por las operaciones remotas. Y el modelo de referencias remotas propuesto, al igual que el de RMI, provee abstracciones distribuidas tanto para la recolección distribuida de basura como para el servicio de nombres, que no se encuentran presentes en el modelo de RTCORBA y en los cuales las latencias derivadas de su utilización son asumidas por las tareas que hacen uso directo o indirecto (en el caso de la recolección distribuida de basura) de estos servicios.

La principal contribución al estado del arte es la de definir un comportamiento para el middleware de distribución RMI acorde con los necesidades de ciertos sistemas de tiempo real. El modelo desarrollado mejora el de RMI clásico definiendo un comportamiento para la invocación remota asíncrona y además caracterizando un comportamiento tanto para el servicio de recolección distribuida de basura como para el de nombres. Pero el modelo no considera otras características avanzadas de RMI como puede ser la descarga de código distribuido o el soporte a la activación de objetos remotos, también presentes en la especificación de RMI. Y por tanto, la inclusión de dichas características dentro del modelo desarrollado constituye la principal línea de trabajo futuro a explorar.

El modelo descrito no entra en detalle sobre los cambios necesarios en el sistema de interfaces RMI para poder realizar la configuración del sistema sino que se limita a definir el comportamiento interno para el middleware. Así, no se llega a abordar el cómo se llega a elegir la prioridad con la que ejecutan las diferentes entidades ni tampoco el cómo se configuran las diferentes entidades empleadas para satisfacer las necesidades dinámicas de recursos o los cambios necesarios en el protocolo de comunicaciones, entre nodos, para dar cabida a este modelo. Ése será el objetivo específico del siguiente capítulo donde se definirá un conjunto de interfaces horizontales y verticales altamente alineadas con RTSJ y con RMI: DREQUIEMI.

Primitiva	Java (RTSJ o RMI)
mid_inf_manager_set	Parcialmente soportada por javax.realtime.Scheduler
mid_inf_memory_allocate	Método new en un HeapMemory o en un ImmortalMemory e instanciación de una ScopedMemory
mid_inf_memory_deallocate	Método System.gc() y destrucción de una ScopedMemory
mid_inf_concurrententity_create	Método start de Thread, RealtimeThread, NoHeapRealtimeThread y jerarquía AsyncEventHandler
mid_inf_concurrententity_destroy	Finalización del método run de los anteriores
mid_inf_concurrencylimitator_create	Creación de un objeto java.util.concurrent.locks.Lock y creación de cualquier objeto Java
mid_inf_concurrencylimitator_destroy	Destrucción de un objeto Lock o de otro general
mid_inf_concurrencylimitator_lock	Método lock en la clase Lock y palabra reservada synchronized
mid_inf_concurrencylimitator_unlock	Método unlock en Lock y finalización de un bloque synchronized
mid_inf_connection_accept	Clase java.net.ServerSocket, método accept
mid_inf_connection_create	Clase java.net.Socket, método connect
mid_inf_connection_close	Clase Socket, método close
mid_inf_connection_send	Método write del outputstream del Socket
mid_inf_connection_receive	Método read del inputstream del Socket
mid_dis_manager_set	No hay equivalente
mid_dis_remoteobject_register	Instanciación de java.rmi.server.RemoteServer
mid_dis_remoteobject_unregister	Destrucción de un objeto RemoteServer o llamada al método unexport
mid_dis_remoteobject_invoke	Clase java.rmi.server.ServerRef, método invoke
mid_dis_stub_register	Instanciación de la clase java.rmi.server.RemoteStub
mid_dis_stub_unregister	Destrucción de un objeto java.rmi.server.RemoteStub
mid_dis_stub_invoke	Clase java.rmi.server.RemoteRef, método invoke
mid_cs_reference	Proceso de serialización de un objeto RemoteRef
mid_cs_unreference	Destrucción de un objeto RemoteRef
mid_cs_naming_bind	Clase java.rmi.Naming, método bind
mid_cs_naming_unbind	Clase java.rmi.Naming, método unbind
mid_cs_naming_lookup	Clase java.rmi.Naming, método lookup

Cuadro 3.1: Equivalencias entre el modelo de primitivas propuesto y las tecnologías Java

Capítulo 4

Extensiones de tiempo real para RMI

En cualquier software la definición de interfaces que permitan acceder a cierta funcionalidad supone enfrentarse a la difícil decisión de tener que seleccionar que partes del modelo han de ser conocidas por el programador y cuales no. Por lo general, esto requiere llegar a soluciones de compromiso pues unas interfaces demasiado austeras impiden utilizar toda la funcionalidad subyacente y por el contrario unas demasiado complejas requieren un esfuerzo extra por parte del implementador del middleware. Es más, muchas veces las interfaces suelen condicionar lo que es el éxito final del software desarrollado, siendo causa en algunos casos de grandes fracasos. Y teniendo en cuenta la existencia de este reto, en este capítulo se proponen unas extensiones para Java de tiempo real distribuido acordes con el modelo descrito en el capítulo anterior denominadas DREQUIEMI.

Tal y como ya hemos visto en el estado del arte, ya existen interfaces para el desarrollo de aplicaciones Java de tiempo real distribuido que podrían ser utilizadas como base para esta tarea que sin embargo no lo serán por diferentes motivos. En el proyecto RTZen ([101] y sección 2.4.8 del estado del arte) se adopta una solución que aprovecha las interfaces de RTCORBA pero que presenta ciertas limitaciones a la hora de acomodar plenamente ciertas características especiales de RTSJ como la `ScopedMemory`. Otras siguen el paradigma de distribución RMI y proponen nuevas jerarquías imitando, parcialmente, el trabajo realizado en RTCORBA. Destaca entre ellas DRTSJ ([184] y sección 2.4.2) que aunque esboza una nueva jerarquía de clases no llega a concretar las implicaciones que éstas tienen ni en el programador ni en el middleware de distribución. Otras aproximaciones como por ejemplo la de la Universidad de York ([29] y sección 2.4.5) aunque se aproximan a temas relativos a la implementación no llegan a proveer soluciones completamente operativas. Y por último la Universidad Politécnica de Madrid ([170] y sección 2.4.6) define una jerarquía de clases completa -HRTRMI y QoSRMI- para cada uno de los escenarios a los que enfoca su solución, lo que produce una solución demasiado ligada a la aplicación. Estas carencias, unido al hecho de que ninguna de las aproximaciones RTRMI, contrariamente a RTCORBA, llega a caracterizar las implicaciones que tienen sus soluciones en la comunicación horizontal, sirve de motivación para definir una solu-

ción propia -DREQUIEMI- que toma por punto de partida para su construcción el modelo descrito en el capítulo anterior.

DREQUIEMI afronta el reto que presenta la definición de interfaces consciente de que es necesario llegar a un doble compromiso: el tecnológico y el de la funcionalidad asumida por el middleware de distribución. Las dos tecnologías que toma como punto de partida DREQUIEMI -RTSJ y RMI- nos retan a intentar buscar combinaciones entre ambos modelos de tal manera que el resultado sea lo más armonioso posible. Así, idealmente, las extensiones no deberían de introducir conceptos diferentes de los que implícitamente ahora se encuentran ya presentes en RTSJ o en RMI. También, a la hora de diseñarla deberemos de tener en cuenta la existencia de dos actores principales con intereses contrapuestos: el programador y el implementador del middleware de distribución. Al programador le interesan unas interfaces ricas en funcionalidad que permitan la realización de operaciones altamente complejas y al implementador del middleware le interesa que sean sencillas de soportar. Y por lo tanto, las interfaces DREQUIEMI deben de intentar establecer algún tipo de punto de equilibrio entre ambas posturas.

Principios de DREQUIEMI

Así pues, las líneas maestras que guían el modelo DREQUIEMI son las siguientes:

- *Alineamiento con el modelo de gestión de recursos de RTSJ.* RTSJ proporciona un modelo de programación que permite la gestión de recursos de forma predecible, introduciendo para ello una serie de clases como son las proporcionadas por la jerarquía de clases `MemoryArea` y la `Scheduler` que permiten realizar una gestión predecible de memoria y de procesador. En plena sintonía con este modelo, DREQUIEMI lo aprovecha y lo extiende definiendo *planificadores distribuidos* y nuevos tipos de recursos encargados de gestionar la memoria, el procesador y las conexiones utilizadas durante una comunicación remota.

- *Alineamiento con el modelo de computación de RMI.* RMI proporciona un modelo de computación distribuido basado tanto en una serie de interfaces horizontales como verticales que permiten que el programador caracterice y cree nuevos objetos capaces de ser accedidos remotamente, como que los diferentes nodos intercambien información entre ellos de forma transparente al programador. Combinando lo que es el modelo basado RMI caracterizado en el capítulo anterior, con lo que es el modelo de gestión de RTSJ, DREQUIEMI define nuevas interfaces tanto horizontales como verticales que permiten el intercambio de cierta información no funcional que es utilizada por las aplicaciones de tiempo real entre los diferentes nodos de la red.

- *Regla de la mínima intrusión.* Este último punto hace referencia a lo que es el grado de originalidad que se pretende que tengan las extensiones y se puede entender también como una máxima u objetivo final a conseguir. Desde la óptica de DREQUIEMI las extensiones ideales son aquellas que no introducen ningún tipo de concepto excesivamente nuevo en el programador, o lo que es lo

mismo en el caso de DREQUIEMI, cuyos conceptos se encuentran ya presentes en los modelos de RMI o de RTSJ. Y por tanto, DREQUIEMI pone un empeño especial en que los conceptos que introduce no resulten raros, tratando de vincularlos a otros ya existentes en RTSJ o en RMI.

Fijado lo que es la necesidad, el reto y las líneas maestras del capítulo se puede empezar a desarrollar lo que son tanto las interfaces horizontales como verticales de DREQUIEMI. La sección 4.1 presenta aquellas interfaces que son utilizadas directamente por el programador de aplicaciones distribuidas de tiempo real y la sección 4.2 aquellas ligadas a la comunicación horizontal que son utilizadas por el middleware de distribución. Una vez definidas, en la sección 4.3, se comparan con las de otras alternativas RTRMI presentes en el estado del arte, intentando establecer relaciones. Por último, en la sección 4.4, están las conclusiones y las líneas futuras de DREQUIEMI.

4.1. Interfaces verticales de comunicación

Antes de definir ningún tipo de extensión y en primer lugar, debemos de darnos cuenta de que la definición de extensiones a la interfaz RMI es una tarea necesaria. Aunque sería posible, partiendo del hecho de que en un cliente el middleware puede acceder a los parámetros de planificación del hilo que está realizando la invocación sobre un sustituto, ofrecer un cierto soporte transparente suficiente para ciertos sistemas de tiempo real, desde un punto de vista práctico, este modelo resulta insuficiente. Para justificarlo podemos fijarnos en RTCORBA donde de forma implícita se está reconociendo que son necesarias interfaces especiales para el desarrollo de aplicaciones de tiempo real. Y a esto le podemos añadir el hecho de que para poder parametrizar el modelo descrito en el capítulo anterior, resulta necesario configurar las diferentes entidades de gestión de recursos que para él se han propuesto.

A modo de síntesis, la figura 4.1 muestra la jerarquía de clases que componen la extensión DREQUIEMI. En ella se puede ver que las clases aparecen categorizadas en tres niveles: aquellas más próximas al programador, conformando extensiones al modelo cliente-servidor; aquellas que se dedican a la gestión de recursos en el middleware de distribución, que parametrizan el comportamiento interno del middleware; y por último, los propios recursos que son manejados internamente por el middleware de distribución.

Veamos, poco a poco, cuál es el cometido de cada una de las clases de la extensión.

4.1.1. Extensiones para el servidor

En el servidor la jerarquía estándar de RMI se ha extendido con una nueva jerarquía de clases que permiten la realización de invocaciones tanto síncronas como asíncronas añadiendo la posibilidad de que el servidor decida sobre el contexto en que se procesarán sus peticiones: *heap* o *noheap*, así como la prioridad a que éstas son ejecutadas. Así, para el servidor y tal y como muestra la figura 4.2, DREQUIEMI introduce tres nuevas clases: `RealtimeUnicastRemoteObject`, `NoHeapUnicastRealtimeRemoteObject` y `AsyncUnicastRealtimeRemoteObject`.

Veamos, clase a clase, cuáles son sus principales características.

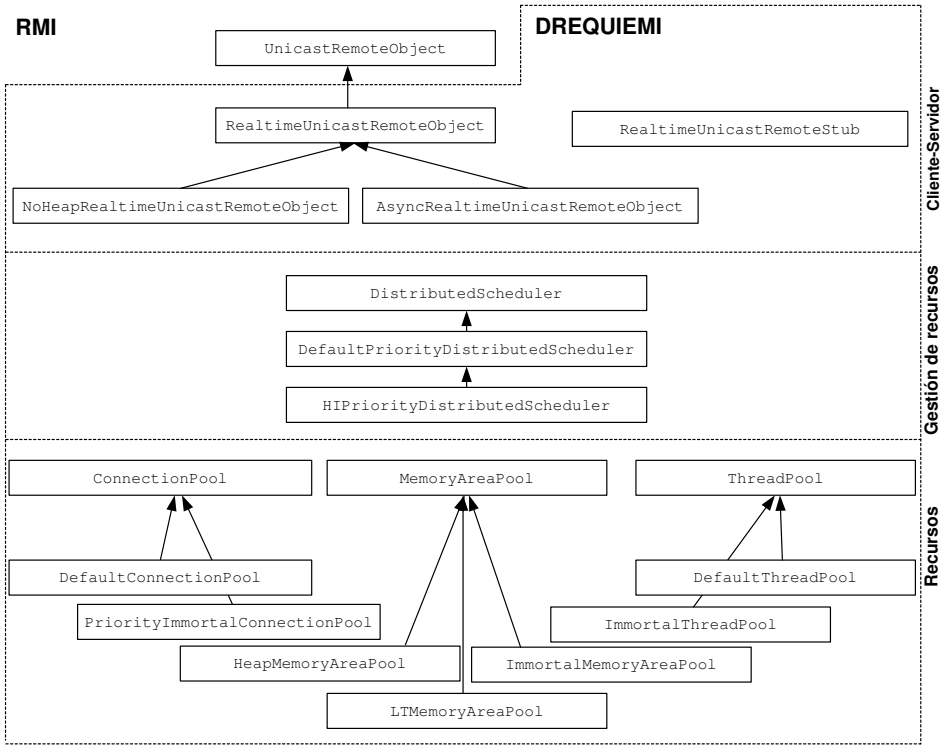


Figura 4.1: Jerarquía de clases de DREQUIEMI y relación con la jerarquía tradicional RMI

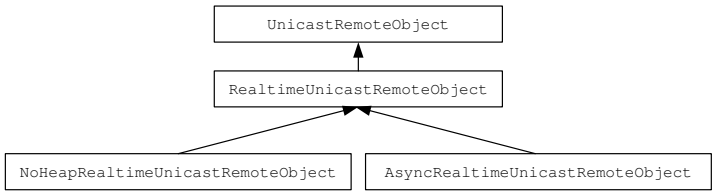


Figura 4.2: Jerarquía de clases definidas para el servidor por DREQUIEMI

- `RealtimeUnicastRemoteObject`. Esta clase es una de las clases maestras que hereda su comportamiento de la clase `java.rmi.server.UnicastRemoteObject` de RMI estándar. Tal y como se ve en la figura 4.4, mediante el uso de constructores especiales es capaz de controlar el tipo de gestión de procesador realizada, considerándose dos posibilidades: la propagada por el cliente y la definida por el servidor. A modo de ejemplo, en el caso de que el parámetro `sp` sea distinto de `null`, los parámetros de planificación utilizados en la invocación del método remoto serán los definidos en el constructor del objeto remoto y en el caso contrario se utilizaran los que son propagados por el cliente.

  También es capaz de decidir sobre cuál es el estado inicial de la memoria utilizada a la hora de invocar al servidor, caracterizando para ello el *scopestack* que es utilizado para invocar al objeto remoto. Éste, de forma similar a lo que ocurre en un `AsyncEventHandler` de RTSJ, constará de dos partes: el contexto de creación del objeto remoto, y una *memoryarea* tomada de un `MemoryAreaPool`. Esta última instancia, tal y como se muestra en la figura 4.3, se apilará en el *scopestack* del hilo invocante justo antes de que comience a ejecutarse la lógica del método remoto y contendrá, entre otros, los parámetros de la invocación del método remoto. En el ejemplo de la figura 4.3 estos parámetros son `a` y `b`.

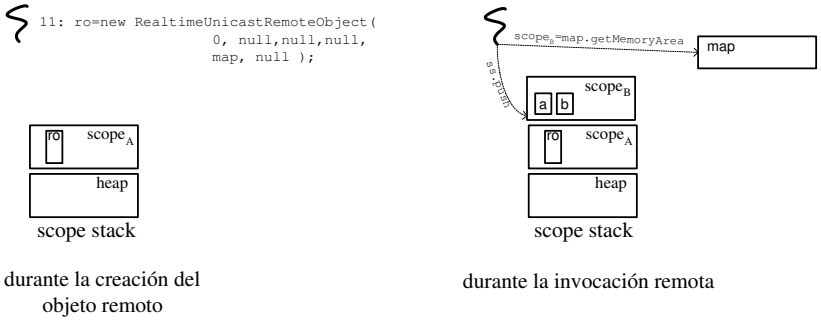


Figura 4.3: Relación entre el *scopestack* utilizado durante la creación del objeto remoto y durante su invocación remota

El objeto remoto permite configurar otros parámetros, tal y como muestra la figura 4.4. Estos parámetros son los de liberación `-rp-`, los de memoria `-mp-`, los de procesado en grupo `-pgp-` (presentes en el modelo RTSJ), otros como el puerto en el cual se esperarán las peticiones entrantes (pertenecientes al modelo de comunicación unicast de RMI) y otros como el *memorypool* `-map-` del cual se

toma la memoria para la realización de la invocación remota (ligados al modelo descrito en el capítulo anterior).

```
package es.uc3m.it.drequiem.rtrmi.server;
public class RealtimeUnicastRemoteObject
                        extends java.rmi.server.UnicastRemoteObject{
public RealtimeUnicastRemoteObject( int port,
                                    SchedulingParameters sp,
                                    ReleaseParameters rp,
                                    MemoryParameters mp,
                                    MemoryAreaPool map,
                                    ProcessingGroupParameters pgp
                                    ) throws java.rmi.RemoteException
                                  . . .
public SchedulingParameters getSchedulingParameters();
public void setSchedulingParameters(SchedulingParameters sp);
                                  . . .
}
```

Figura 4.4: Detalle de la clase `RealtimeUnicastRemoteObject`

Por último, el contexto de ejecución relacionado con el tipo de memoria utilizada para la realización de la invocación remota en el servidor es propagado por el cliente. Si el cliente realiza la invocación haciendo uso de un `NoHeapRealtimeThread`, el servidor será invocado con este tipo de hilo y en el caso contrario de utilizar un `RealtimeThread`, se utilizará este último tipo de hilo para realizar la invocación sobre el objeto remoto.

Además de los métodos incluidos en el constructor, este tipo de objeto remoto permite acceder y modificar dinámicamente, mediante métodos de tipo `get` y `set`, similares a los definidos en la interfaz `Schedulable` de RTSJ, los parámetros que han sido pasados a través de su constructor durante su creación.

- `NoHeapRealtimeUnicastRemoteObject`. Esta clase, particularización de la anterior, posibilita que sea el servidor el que decida, a la hora de crear el objeto remoto, el tipo de hilo que realizará la invocación remota, haciendo caso omiso del tipo de hilo utilizado en el cliente. Tal y como muestra en detalle la figura 4.5, en el constructor de la clase se añade un nuevo parámetro `heap` que selecciona el tipo de hilo -`RealtimeThread` o `NoHeapRealtimeThread`-, encargado de invocar al método remoto. Ésta es la única diferencia que existe entre esta clase y su clase padre, el `RealtimeUnicastRemoteObject`.

- `AsyncRealtimeUnicastRemoteObject`. Esta clase, particularización del `RealtimeUnicastRemoteObject`, permite la realización de invocaciones remotas asíncronas confirmadas por el servidor. Automáticamente, un servidor que la extienda tendrá un comportamiento asíncrono, esto es, el resultado de la invocación remota no será devuelto al cliente. Además de esta característica, al igual que en el `AsyncEventHandler` de RTSJ, en este tipo de objeto remoto el desarrollador ha de decidir sobre si la invocación es realizada por un `No-`

```
package es.uc3m.it.drequiem.rtrmi.server;

public class NoHeapRealtimeUnicastRemoteObject
                          extends RealtimeUnicastRemoteObject {
public NoHeapRealtimeUnicastRemoteObject( int port,
                                          bool heap,
                                          SchedulingParameters sp,
                                          ReleaseParameters rp,
                                          MemoryParameters mp,
                                          MemoryAreaPool map,
                                          ProcessingGroupParameters pgp
                                          ) throws java.rmi.RemoteException{
```

Figura 4.5: Detalle de la clase `NoHeapRealtimeUnicastRemoteObject`

`HeapRealtimeThread` o por un `RealtimeThread`, utilizando para ello (ver la figura 4.6) el parámetro `heap` del constructor.

```
package es.uc3m.it.drequiem.rtrmi.server;

public class AsyncRealtimeUnicastRemoteObject
                                      extends RealtimeUnicastRemoteObject {
public AsyncRealtimeUnicastRemoteObject(  int port,
                                          boolean heap,
                                          SchedulingParameters sp,
                                          ReleaseParameters rp,
                                          MemoryParameters mp,
                                          MemoryAreaPool map,
                                          ProcessingGroupParameters pgp
                                          )throws java.rmi.RemoteException
```

Figura 4.6: Detalle de la clase `AsyncRealtimeUnicastRemoteObject`

Por último, en el caso de que se invoque un objeto remoto tradicional de tipo `UnicastRemoteObject`, desde un cliente `RealtimeThread`, también se obtienen una serie de garantías mínimas sobre la ejecución realizada en el servidor. La prioridad a la que ejecutará el servidor será la misma que la del cliente y la memoria utilizada para satisfacer la petición del mismo tipo que la que había sido utilizada para crear el servidor, esto es, la `HeapMemory`.

4.1.2. Extensiones para el cliente

A la hora de definir el modelo para el cliente se ha querido mantener la compatibilidad hacia atrás y se ha buscado una solución en la cual no son necesarias modificaciones en el compilador de interfaces de RMI: el *rmic*. Este objetivo se ha logrado mediante la introducción de una clase estática que permite asignar y recuperar las propiedades del sustituto de tiempo real a partir del sustituto tradicional generado por el *rmic*. Esta clase, tal y como muestra la figura 4.7, es la `RealtimeRemoteStub`.

```
package es.uc3m.it.drequiem.rtrmi.server;
import javax.realtime.*;
import es.uc3m.it.drequiem.rtrmi.*;
import java.rmi.server.RemoteStub;
public class RealtimeUnicastRemoteStub {
 public static void setParameters(  RemoteStub stub,
                                    boolean async,
                                    SchedulingParameters sp,
                                    ReleaseParameters rp,
                                    MemoryParameters mp,
                                    ProcessingGroupParameters pgp,
                                    ConnectionPool cpool)

. . .
public static SchedulingParameters getSchedulingParameters(
                                    RemoteStub stub);
public static void setSchedulingParameters(
                                    RemoteStub stub,
                                    SchedulingParameters sp);
. . .
public static ConnectionPool getConnectionPool(
                                    RemoteStub stub);
public static void setConnectionPool(
                                    RemoteStub stub,
                                    ConnectionPool conn);

}
```

Figura 4.7: Detalle de la clase `RealtimeRemoteStub`

Esta clase decide sobre cuáles son los parámetros que serán propagados desde el cliente al servidor en cada invocación remota. En el caso de que al invocar a un objeto remoto no se hayan fijado cuáles son los parámetros de invocación que serán utilizados, éstos serán tomados del hilo que realiza la invocación sobre el sustituto. En este caso, si la invocación la realiza un hilo que es de tiempo real, se propagarán su prioridad y la relación que mantiene con el recolector de basura `-heap` o `noheap-`, siendo la invocación remota realizada síncrona. Pero en el caso contrario, en el que se haya fijado algún tipo de parámetro para el sustituto, éste prevalecerá sobre el del propio hilo.

Esta clase también permite decidir sobre el tipo de *connectionpool* que se utilizará para realizar la invocación remota. Mediante el constructor de la clase se puede fijar uno por defecto que mediante métodos de tipo `get` y `set` puede ser modificado dinámicamente.

El parámetro `async` permite decidir sobre si la invocación a los métodos de tipo `void method(...)` es asíncrona o no. En caso de serlo, internamente, esos métodos tendrán un comportamiento que se ha caracterizado como asíncrono en el modelo desarrollado en el capítulo anterior, retornando la invocación justo después de enviar los parámetros de la invocación al servidor, sin esperar ningún tipo de respuesta proveniente de éste.

Por último, es de resaltar que el mecanismo propuesto, aunque nuevo, tiene un claro antecedente en la jerarquía estándar de RMI. El método estático `toStub` de la clase `java.rmi.server.RemoteObject` permite obtener una referencia a un sustituto a partir de un objeto remoto cualesquiera. Y por tanto, no se puede decir que la idea de utilizar una clase estática para modificar características del sustituto sea algo totalmente novedoso.

4.1.3. Extensiones relacionadas con la gestión de recursos

Al igual que el propio RTSJ, DREQUIEMI identifica tanto a los recursos como al planificador distribuido como entidades de primer orden, definiendo clases que le proporcionan soporte. En total, tal y como se muestra en la figura 4.8, DREQUIEMI cuenta tanto con una serie de recursos como de gestores por defecto que permiten a las aplicaciones ganar un cierto control sobre el comportamiento interno del middleware.

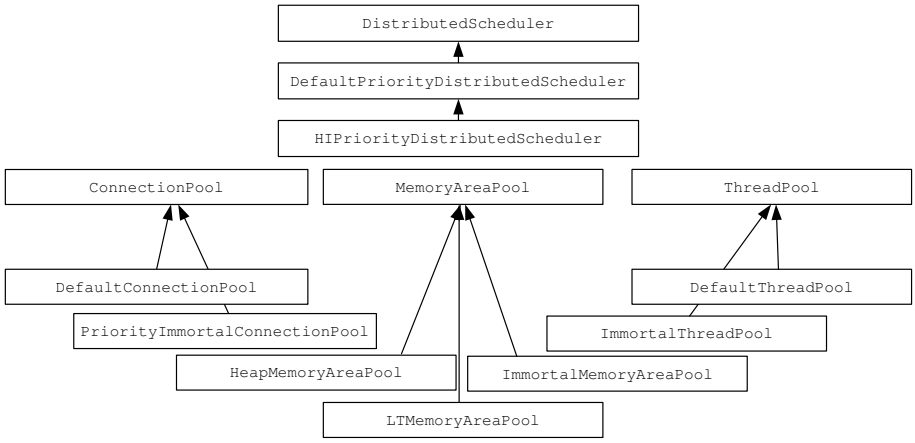


Figura 4.8: Clases relacionadas con la gestión de recursos

Recursos

Siguiendo el modelo propuesto en el capítulo anterior, se distinguen hasta tres familias de clases, una por cada tipo de recurso:

- `MemoryAreaPool`. Esta familia de clases se asocia a los *memorypools* del modelo descrito en el capítulo anterior y permite reservar un conjunto de bloques de memoria que serán más tarde utilizados durante la invocación remota. Al igual que en el modelo de *memoryareas* de RTSJ, existen tres tipos de especializaciones: `LTMemoryAreaPool` donde la memoria que se maneja son instancias de la `LTMemory`; `ImmortalMemoryAreaPool` donde la memoria manejada es de tipo `ImmortalMemory`; y `HeapMemoryAreaPool` donde se maneja memoria de tipo `HeapMemory`.

  La figura 4.9 muestra en detalle la clase `LTMemoryAreaPool`. Ésta consta de dos métodos, el `getMemoryArea` y el `returnMemoryArea` de la interfaz `MemoryAreaPool`, que permiten respectivamente conseguir y devolver instancias de

tipo `LTMemory`. El constructor permite fijar un número mínimo y otro máximo tanto para el número de instancias que puede proporcionarnos un `LTMemoryAreaPool` -`min`,`max`- como para el tamaño de memoria -`minSize`, `maxSize`- que es manejado por cada instancia de tipo `LTMemory`.

```
package es.uc3m.it.drequiem.rtrmi;
import javax.realtime.*;
public class LTMemoryAreaPool implements MemoryAreaPool {
  public LTMemoryAreaPool(int min, int max, int minSize, int maxSize)
  public MemoryArea getMemoryArea(Object policy)
  public void returnMemoryArea(MemoryArea ma)
}
```

Figura 4.9: Detalle de la clase `LTMemoryAreaPool`

- `ConnectionPool`. Esta familia de clases permite la realización de comunicaciones con un determinado nodo de la red y se corresponde con el *connectionpool* del modelo desarrollado en el capítulo anterior. Hasta el momento, en DREQUIEMI existen dos tipos: uno `DefaultConnectionPool` donde las conexiones necesarias son creadas de forma dinámica para cada invocación remota, y otro, `PriorityImmortalConnectionPool`, basado en la preserva de cierto número de conexiones y entidades concurrentes remotas que son reutilizadas tras la finalización de la invocación remota.

  La figura 4.10 muestra detalles de la clase `PriorityImmortalConnectionPool`. Ésta consta de un único método estático, `addConnection`, que a partir de una referencia a un objeto remoto permite establecer e inicializar una conexión con el nodo remoto donde reside dicho objeto remoto así como decidir sobre los parámetros de planificación -`sp`- que le son enviados durante el establecimiento de dicha conexión.

```
package es.uc3m.it.drequiem.rtrmi;
import javax.realtime.*;
import java.rmi.*;
public class PriorityImmortalConnectionPool implements ConnectionPool {
      public static void addConnection(RemoteRef ref, SchedulingParameters sp)
}
```

Figura 4.10: Detalle de la clase `PriorityImmortalConnectionPool`

- `ThreadPool`. Esta familia de clases, correspondiente al *threadpool* descrito en el capítulo anterior, permite la obtención de entidades concurrentes que pueden ser utilizadas durante el proceso de invocación remota para proporcionar un modelo de comunicaciones asíncrono. Al igual que en el caso de los elementos de conexión, son posibles dos configuraciones. Una en la que éstos son creados de forma dinámica durante cada invocación remota, asociada a la clase `DefaultThreadPool`. Y otra, asociada a la clase `ImmortalThreadPool`, donde es

posible realizar una reserva inicial de entidades concurrentes para luego evitar su creación de forma dinámica durante la ejecución de operaciones remotas.

La figura 4.11 muestra en detalle el `ImmortalThreadPool`. El constructor permite fijar un número mínimo de hilos así como un número máximo de hilos a ser creados en memoria inmortal durante la instanciación del *threadpool*. Los métodos `getThread` y `returnThread` permiten al middleware obtener las entidades concurrentes que serán utilizadas internamente por el middleware para atender peticiones entrantes asíncronas. Como se puede ver, no es necesario definir unos parámetros de planificación para estos hilos ya que el middleware de distribución los inicializa adecuadamente con información propagada por el cliente o definida por el objeto remoto de tiempo real.

```
package es.uc3m.it.drequiem.rtrmi;
import javax.realtime.*;
import java.rmi.*;
public class ImmortalThreadPool implements ThreadPool {
      public ImmortalThreadPool(int min, int max){ . . .}
      public Thread getThread(Object policy){ . . . }
      public void returnThread(Thread ma){ . . . }

}
```

Figura 4.11: Detalle de la clase `ImmortalThreadPool`

Gestores de recursos distribuidos

Por otro lado, DREQUIEMI introduce una nueva jerarquía de clases, con raíz común `DistributedScheduler`, que permiten un cierto grado de parameterización del comportamiento interno del middleware subyacente. Este planificador distribuido guarda relación con el `Scheduler` de RTSJ, en el sentido de que ambos permiten la realización de un control fino sobre los recursos internos. La gran diferencia entre ambos radica en el tipo de gestión realizada, mientras el de RTSJ es un gestor más general, el de DREQUIEMI está más orientando a proveer soluciones a la gestión predecible de las comunicaciones remotas, tomando el de RTSJ como base.

Hasta ahora, en DREQUIEMI se caracterizan dos gestores:

- `DefaultPriorityDistributedScheduler`. Este gestor es el por defecto y está enfocado a lo que es la obtención de una gestión de recursos pareja a la existente en RTSJ pero dentro del marco de RMI. Dentro de la clasificación en niveles de DRTSJ podríamos encuadrarlo en el nivel 1 ya que este tipo de planificador distribuido ofrece soporte predecible para la invocación remota, sin que su implementación requiera de cambios dentro de la máquina virtual de tiempo real ni de sincronización temporal entre los diferentes nodos de la red. Su principal limitación, presente también en el nivel 1 de DRTSJ, es la ausencia de soporte básico para el paradigma de hilo distribuido de tiempo real.

  La figura 4.12 muestra un detalle de su constructor en cual se pueden ver los parámetros sobre los que puede actuar el programador.

```
package es.uc3m.it.drequiem.rtrmi;
import javax.realtime.*;
public class DefaultPriorityDistributedScheduler
                        extends DistributedScheduler {
 public DefaultPriorityDistributedScheduler(
                                              SchedulingParameters sp,
                                              ReleaseParameters rp,
                                              MemoryParameters mp,
                                              ProcessingGroupParameters pgp,
                                              ThreadPool globalth,
                                              SchedulingParameters dgcsp,
                                              MemoryAreaPool dgcmemorypool,
                                              SchedulingParameters namingsp,
                                              MemoryAreaPool namingmemorypool,
                                              ConnectionPool connp )
```

Figura 4.12: Detalle de la clase `DistributedDistributedScheduler`

Los parámetros del constructor hacen referencia tanto a la infraestructura más básica como a cuestiones relacionadas con los servicios básicos presentes en el modelo de distribución RMI. Así `sp`, `rp`, `mp`, `pgp`, `globalth`, `connpool` sirven respectivamente para definir el comportamiento interno del hilo durante el procesado del protocolo de comunicaciones así como para definir un *threadpool* y un *connectionpool* globales de los cuales se pueda obtener respectivamente entidades concurrentes y conexiones. Por otro lado, `dgcdsp` y `dgcmemorypool` permiten controlar el comportamiento del servicio de recolección distribuida de basura existente en cada nodo y `namingsp` y `namingmemorypool` el de nombres.

- `HIPriorityDistributedScheduler`. Este gestor restringe al gestor por defecto impidiendo que se utilicen hilos que no sean instancias del `NoHeapRealtimeThread`. Si un hilo diferente del `NoHeapRealtimeThread` intenta interactuar con el middleware, éste le deniega sus servicios, lanzando una excepción. Además, los únicos recursos que puede utilizar son el `LTMemoryAreaPool`, el `ImmortalThreadPool` y el `ImmortalConnectionPool`. Se trata pues de un gestor especialmente pensado para el desarrollo de aplicaciones con unos requisitos de predictibilidad altos.

Tras esto ya se ha visto, una a una, las diferentes interfaces verticales definidas por DREQUIEMI que permiten al programador interactuar con el middleware de distribución. A continuación, se verá como DREQUIEMI extiende el protocolo de comunicaciones básico de RMI, JRMP, creando un protocolo de tipo RTJRMP así como cuáles son las interfaces que define para el servicio de recolección distribuida de basura.

4.2. Interfaces horizontales de comunicación

A nivel horizontal, DREQUIEMI, extiende dos de los protocolos de comunicación nodo a nodo definidos por RMI. El primero es el protocolo de comunicaciones JRMP para que permita la transmisión de datos, entre el cliente y el servidor, que

son utilizados por el `DefaultPriorityDistributedScheduler`. El segundo, es la definición de una nueva interfaz que es utilizada por el servicio de recolección distribuida de basura. DREQUIEMI no define ningún tipo de interfaz para el servicio de nombres sino que la parametrización de su funcionamiento se hace mediante el `DistributedScheduler`.

4.2.1. Protocolo de comunicaciones de tiempo real

Una de las grandes limitaciones del protocolo JRMP, a la hora de ser utilizado para la transmisión de datos entre los diferentes nodos de la red, es la falta de mecanismos que permitan la transmisión de información no funcional, entre los diferentes nodos de la red. Aunque la mayoría de los autores del estado del arte relativo a RTRMI no proponen extensiones al protocolo de comunicaciones JRMP sino que proponen su transmisión como parámetros adicionales de la invocación, DREQUIEMI, propone extensiones al protocolo JRMP de tal manera que dicha información viaja junto a los mensajes JRMP.

Las razones argumentadas para realizar tal cambio son dos. La primera es la de facilitar el procesado tanto del lado del cliente como del lado del servidor de información no funcional. Y la segunda es la de reducir la inversión de prioridad extremo a extremo experimentada durante el envío y la recepción de mensajes.

El protocolo RTJRMP, tal y como se muestra en la figura 4.13, se basa en una idea muy sencilla que es la de introducir información adicional en una cabecera que es intercambiada de forma síncrona cada vez que se negocia un tipo de conexión JRMP así como cada vez que se envían datos por dicha conexión.

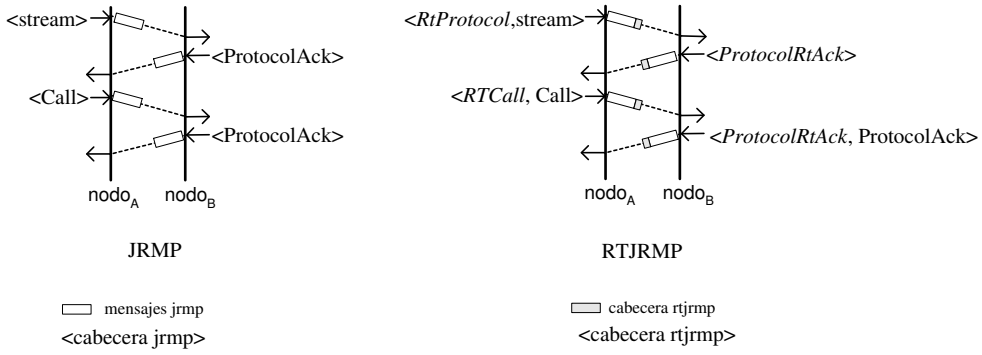


Figura 4.13: Diferencias entre el protocolo JRMP y el RTJRMP

A continuación, se van a ver más en detalle los cambios que se introducen tanto en la negociación de la conexión como durante el intercambio de datos entre el cliente y el servidor. Estos cambios han sido diseñados para satisfacer el modelo de gestión del `DefaultPriorityDistributedScheduler` y del `HIPriorityDistributedScheduler`. Otros modelos de planificador distribuido más potentes, como por ejemplo uno que soportase el paradigma del hilo distribuido deberían, en principio, de introducir nuevas extensiones en el protocolo que a continuación va a ser descrito.

Conexión de tiempo real: *RTProtocol*

El modelo de JRMP distingue hasta tres tipos de subprotocolos de comunicación: (1)*StreamProtocol* donde se crea un canal de comunicaciones que es reutilizado en sucesivas invocaciones, (2) *SingleOpProtocol* donde se crea un canal de comunicaciones para el envío de cada mensaje y (3) *MultiplexProtocol* donde se crea un canal de comunicaciones que permite enviar datos de forma multiplexada mediante la creación de subconexiones, permitiendo además en todos los subprotocolos el encapsulado de los mensajes en cabeceras *http*, lo que le permite atravesar cortafuegos. De éstos, el perfil para sistemas con recursos limitados elimina tanto el mecanismo de multiplexación como el de encapsulación dentro de flujos *http*. Y por último, DREQUIEMI define un nuevo tipo de subprotocolo, *RTProtocol*, que sirve de envoltorio a cualquiera de los tres subprotocolos descritos y que permite el intercambio de información no funcional durante la fase de establecimiento del canal. Este mensaje es contestado con un mensaje de tipo *RTAck* que permite que fluya la información en el sentido contrario, hacia la entidad que establece la conexión.

En el caso de que se esté utilizando el `DefaultPriorityDistributedScheduler` la información de planificación intercambiada será la prioridad de ejecución inicial a la que un determinado nodo estará esperando peticiones y la relación que es mantenida con el recolector de basura.

La figura 4.14 muestra, en negrita, los cambios que es necesario realizar en la gramática de JRMP descrita en la especificación RMI [1] para dar cabida a los dos tipos de mensajes.

Básicamente, en la gramática, se definen dos nuevos elementos: un nuevo tipo de protocolo, *RTProtocol*, y un nuevo tipo de respuesta, *ProtocolRTAck*, ambos de tiempo real. El primero de ellos posibilita que antes de enviar la información relativa al establecimiento de la conexión, ambos nodos intercambien información relativa al tipo de hebra o a la prioridad a la que serán procesadas las peticiones entrantes.

Básicamente la información que recibirá es el contexto de ejecución -*heap* o *noheap*- así como la prioridad base a la que ejecutará este hilo.

Como el número de prioridades existentes en un nodo y el rango que cubren, de forma similar a lo que ocurre en RTCORBA con el sistema operativo subyacente, puede variar de una máquina virtual de tiempo real a otra, es necesario llegar a un convenio sobre cómo es transmitida dicha información dentro de un flujo JRMP. En el caso de DREQUIEMI, lo que se hace es utilizar una correspondencia una-a-una entre las prioridades de cada nodo y las transmitidas finalmente por la red, fijando la prioridad máxima transmitida en cero. Y así, la prioridad más alta del planificador, `PriorityScheduler.MAXPRIORITY`, cuando se serializa para ser transmitida por la red toma el valor 0. Y de la misma manera y tal y como se muestra en la figura 4.15, el resto de valores se asignan de forma consecutiva utilizando para ello los valores negativos comprendidos en el rango $[0, -2^{16}]$. De tal manera que la prioridad finalmente transmitida es resultado de aplicar la siguiente conversión:

$$priority_{trans} = PriorityScheduler_{local}.MAXPRIORITY - priority_{local}$$

[1] Realmente el protocolo JRMP no aparece descrito en la especificación de RMI (ver [167]) como tal, sino que aparece denominado como RMI Wire Protocol.

```
Out:
      . . .

Protocol:
      SingleOpProtocol
      StreamProtocol
      MultiplexProtocol
      RTProtocol
      . . .
MultiplexProtocol:
      0x4d
RTProtocol:
      0x54 priority noheap SingleOpProtocol
      0x54 priority noheap StreamProtocol
      0x54 priority noheap MultiplexProtocol

priority:
      long
noheap:
      boolean
      . . .

In:
      ProtocolRTAck ProtocolAck
      ProtocolAck
      . . .


ProtocolRTAck:
      0x55 priority noheap
      . . .
```

Figura 4.14: Cambios introducidos por *RTProtocol* y *ProtocolRTAck* en la gramática de la especificación JRMP

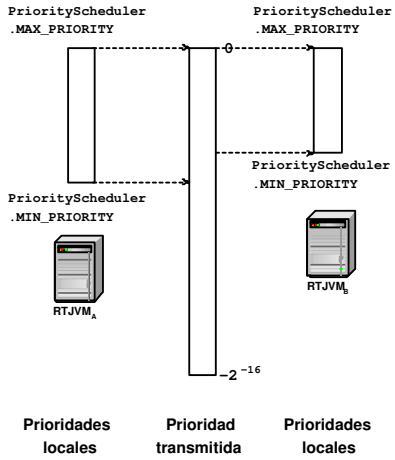


Figura 4.15: Serializado y deserializado de prioridades en DREQUIEMI

En el otro extremo, la entidad que recibe los datos provenientes de otro nodo tendrá que adaptar la prioridad que ha sido transmitida desde un nodo remoto a su esquema de prioridades locales. Para ello la operación que realizará es la siguiente:

$$priority_{local} = PriorityScheduler_{local}.MAXPRIORITY + priority_{trans}$$

Mensajes de tiempo real: *RTCall*

En el modelo de JRMP también se caracterizan una serie de mensajes que son intercambiados entre el cliente y el servidor dentro de cada uno de sus subprotocolos, definiendo para ello tanto mensajes de entrada como de salida. Los posibles mensajes de entrada definidos por la especificación RMI son tres: *Call*, encargado de realizar una invocación remota; *Ping*, utilizado para comprobar el estado de una conexión; y *DGCAck* utilizado por el recolector distribuido de basura. A este conjunto inicial de mensajes DREQUIEMI añade uno nuevo, *RTCall*, que sirve de envoltorio al resto. Al igual que el *RTProtocol*, este mensaje es contestado con un *RTAck*.

La figura 4.16 muestra los cambios que son necesarios, en negrita, en la gramática de JRMP para dar acomodo al mensaje *RTCall*.

En ella se puede ver que la información adicional que se le hace llegar al servidor incluye tanto elementos útiles a la hora de realizar una planificación como a la hora de gobernar el comportamiento interno del middleware. La información de planificación pasada, al igual que sucede durante el establecimiento de la conexión consiste en información sobre el tipo de invocación que va a ser realizada -*noheap*- y una prioridad normalizada -*priority*-. La información relativa al comportamiento permite elegir si la invocación es asíncrona o no -*async*- poseyéndose además un identificador único -*mid*- de la invocación remota generado por el cliente, que puede ser utilizado para multiplexar mensajes sobre un mismo canal, y el identificador del objeto remoto -*oid*- sobre el cual se va a realizar la invocación remota.

Aunque sea redundante, DREQUIEMI introduce el identificador del objeto remoto, disponible también en el mensaje JRMP *CallData*[2], para facilitar la implementación, posibilitando así el uso de técnicas de búsqueda rápida: $\Theta(1)$.

4.2.2. Extensiones para el recolector distribuido de basura de tiempo real

Aunque al igual que sucede con el servicio de nombres se podrían haber reutilizado las del mecanismo de recolección de RMI en DREQUIEMI, introduciendo para ello modificaciones en las interfaces actuales y dentro de su comportamiento interno, se ha querido independizar el recolector de basura tradicional del de tiempo real, en un intento de minimizar la inversión de prioridad extremo a extremo experimentada. Así pues en este nuevo recolector distribuido de basura, al contrario que en el de RMI estándar, no es necesario ningún tipo de garantía extra sobre la tasa de progreso de los relojes de los diferentes nodos del sistema, pudiendo cada uno de ellos evolucionar a diferente velocidad. Así, se ha definido una interfaz que internamente no hace uso del mensaje *DGCAck* para confirmar la correcta transmisión de una referencia a un objeto remoto, perdiéndose así garantías frente a tolerancia a fallos, garantizada por el actual RMI, pero ganándose en determinismo temporal y eficiencia en entornos donde no hay necesidad de soporte para la tolerancia a fallos.

[2]Este mensaje, el *CallData*, forma parte del mensaje *Call* y contiene la información necesaria para realizar la invocación remota en el servidor. Entre esa información están el identificador de objeto (*ObjectIdentifier*), la operación realizada (*Operation*), una comprobación de integridad (*Hash*) y los argumentos opcionales ($Arguments_{opt}$).

```
Message:
      RtCall
      Call
      Ping
      DgcAck
RtCall:
      0x55 priority noheap mid async oid Call
      0x55 priority noheap mid async oid Ping
      0x55 priority noheap mid async oid DgcAck
mid:
      long
async:
      boolean
oid:
      objNum unique time count
objNum:
      long
unique:
      int
time:
      long
count:
      short
```

Figura 4.16: Cambios introducidos por *RTCall* en la gramática de JRMP

Tal y como se ha propuesto en el modelo del capítulo anterior, y como también se refleja en la figura 4.17, el modelo es sencillo y está basado en un mecanismo de contaje que se apoya en la utilización de un objeto remoto con una interfaz bien definida. El método `referenced` incrementa en una unidad el contador de referencias externas al objeto remoto con identificador `oid` y `unreferenced` lo decrementa.

```
package es.uc3m.it.drequiem.rtrmi.server.dgc;
import java.rmi.server.dgc.*;
public interface RTDGCInterface extends java.rmi.Remote{
    public void referenced(java.rmi.server.ObjID objid)
                                        throws java.rmi.RemoteException;
    public void unReferenced(java.rmi.server.ObjID objid)
                                        throws java.rmi.RemoteException;
}
```

Figura 4.17: Interfaz remota del recolector de basura de tiempo real

Los parámetros de planificación `-dgcsp`, `dgcmemorypool-` así como el `MemoryAreaPool` utilizados durante la invocación de cada uno de estos métodos pueden ser fijados a través del `DefaultPriorityDistributedScheduler`. De no definirse ningún parámetro de planificación para el servicio de recolección distribuida de basura, se sigue un modelo propagado y se toman estos parámetros del hilo que realiza la invocación al servicio de recolección distribuida de basura en el cliente, tal y como se ha visto en el capítulo anterior.

4.3. Relación entre DREQUIEMI y otras aproximaciones a RTRMI

Tal y como ha visto durante el análisis del estado del arte y en el inicio de esta sección, aparte de las interfaces DREQUIEMI existen otros tres trabajos en la misma línea y que proveen nuevas clases para el desarrollo de aplicaciones de tiempo real distribuidas basadas en RTSJ y en RMI:

- DRTSJ, descrito en la sección 2.4.2.
- RTRMI-York, descrito en la sección 2.4.5.
- RTRMI-UPM, descrito en la sección 2.4.6.

El objetivo de esta sección es establecer equivalentes entre la funcionalidad provista por estas aproximaciones a RTRMI y DREQUIEMI. Para ello y en primer lugar se irá viendo cómo las diferentes características de DREQUIEMI están soportadas o no por las aproximaciones anteriormente descritas. Después, de forma más breve e inversamente, se verá si el soporte ofrecido por DREQUIEMI es suficiente para soportar la funcionalidad descrita por cada una de las anteriores aproximaciones.

4.3.1. Correspondencia entre DREQUIEMI y DRTSJ, RTRMI-York y RTRMI-UPM

En primera parte de la comparación se tratará de identificar las principales características del modelo DREQUIEMI en el resto de soluciones RTRMI descritas en el estado del arte. Característica a característica de DREQUIEMI, se irá viendo si las diferentes aproximaciones le proporcionan un buen soporte o no.

- *Sustituto de tiempo real*: `RealtimeRemoteStub`.

  Básicamente esta interfaz permite definir unos parámetros de planificación para el sustituto que serán utilizados cada vez que se realice la invocación remota desde él. Además, existe la posibilidad de que se defina si el comportamiento del método remoto realizado es asíncrono o no.

  En DRTSJ no existe la posibilidad de definir este tipo de parametrización ligada a la vida del sustituto. Si éste precisa que cada sustituto propague unos parámetros de planificación específicos habrá de cambiar los del propio hilo justo antes de invocar al objeto remoto. Tampoco existe la posibilidad de realizar asincronismo, aunque existen clases que proporcionan mecanismos alternativos basados en un mecanismos de eventos.

  RTRMI-York tampoco permite asociar parámetros de planificación al sustituto. Si se desea realizar dicha operación se ha de o bien modificar la implementación del sustituto, haciendo uso de versiones especiales del *rmic*, o bien se han de asignar parámetros de planificación al hilo cliente justo antes de realizar la invocación remota. El dotar al sistema de mecanismos de asincronía no se

encuentra dentro de los objetivos del modelo RTRMI-York y no se le proporciona ningún tipo de soporte. Por tanto, se puede decir que el grado de soporte ofrecido por RTRMI-York es medio.

RTRMI-UPM, fuertemente basado en RTRMI-York, tampoco permite asociar parámetros de planificación al sustituto sino que estos son generados por el rmic. Tampoco soporta asincronía en el cliente y por tanto al igual que en RTRMI-York el grado de cobertura de soporte ofrecido por RTRMI-UPM es medio.

- *Objetos remotos de tiempo real*: `RealtimeUnicastRemoteObject`.

  Esta clase y las que heredan de ella permiten definir una serie de parámetros de planificación que son utilizados a la hora de realizar la invocación remota en el cliente así como definir un comportamiento por defecto para el uso del montículo en el servidor y el establecimiento de un modelo de asincronía confirmado por el servidor. También permite especificar un puerto en el cual atender peticiones de conexiones provenientes del cliente.

  DRTSJ, en el nivel 1, no llega a especificar ningún tipo de objeto remoto de tiempo real. En el nivel 2 se definen las interfaces de un hilo distribuido de tiempo real, pero tampoco se define ningún tipo de objeto remoto de tiempo real. Y por tanto, se puede decir que no se está especificando tal comportamiento. También, pese a contar con modelo de asincronía, éste no es capaz de dar cabida al modelo de DREQUIEMI, pues DRTSJ no permite que exista un flujo de datos relativos a la aplicación entre el cliente y el servidor. Y por tanto el grado de soporte ofrecido por DRTSJ es bajo.

  RTRMI-York provee una clase, `UnicastRealtimeRemoteObject`, que se asemeja bastante a la del modelo propuesto. Se soporta la definición de unos parámetros de planificación y se permite asociar un objeto remoto a un puerto tcp/ip particular, pero no se permite decidir sobre si éste soporta asincronismo confirmado por el servidor o por el contrario, no lo hace.

  RTRMI-UPM no introduce detalles suficientes que nos permitan saber si dicha funcionalidad está o no presente en el modelo. Claramente, el asincronismo con confirmación del servidor no se encuentra presente pues la aproximación es totalmente síncrona. Otros parámetros, como por ejemplo el puerto de aceptación, no aparecen en el modelo de interfaces pero de alguna manera deberían de aparecer.

- *Planificadores distribuidos y recursos*: `DistributedScheduler`.

  Tanto esta interfaz y las diferentes clases que la soportan -`DefaultPriority-DistributedScheduler`-, como las encargadas de la reserva y la liberación de recursos -`ThreadPool`, `MemoryAreaPool` y `ConnectionPool`- permiten que el middleware de comunicaciones pueda ser parametrizado y configurado para diferentes tipos de aplicaciones y/o técnicas de gestión de recursos, sin necesidad de modificar lo que son las interfaces del objeto remoto o del sustituto.

DRTSJ se apoya directamente en el planificador de RTSJ y no proporciona abstracciones capaces de decidir sobre la gestión interna realizada por el middleware de distribución. Se puede decir que el concepto de planificador distribuido de DREQUIEMI no encuentra un equivalente en este modelo. También, pese a que RTSJ provee clases que permiten la reserva y la liberación de recursos (e.g. `LTMemory`), DRTSJ no llega a identificar nuevos recursos ligados a lo distribuido. Por tanto podemos decir que DRTSJ, aunque muy flexible en otros aspectos como es la abstracción de hilo distribuido, no lo es tanto a la hora de proveer reconfigurabilidad en el middleware.

RTRMI-York, inspirado en el nivel 1 de DRTSJ, tampoco ofrece ningún tipo de mecanismo que permita acercarse a lo que es la flexibilidad del planificador distribuido de DREQUIEMI. A la hora de especificar los recursos utilizados en la invocación remota, RTRMI-York caracteriza un *threadpool*, pero no caracteriza un *connectionpool* o un *memoryareapool*, que permanecen ocultos al programador.

RTRMI-UPM, ofrece una aproximación estática al planificador distribuido. Así, en vez de definir un planificador para cada uno de los posibles escenarios de aplicación, lo que se propone es una jerarquía de clases RTRMI -HRTRMI y QoSRMI- para cada uno de los escenarios a los que puede ser enfocado. Y por tanto, se puede decir que se ofrece un soporte parcial a lo que es el modelo del planificador distribuido de DREQUIEMI.

- *Protocolo de comunicaciones de tiempo real: RTJRMP*

  En lo que es la comunicación horizontal, DREQUIEMI, caracteriza un protocolo RTJRMP que incorpora de forma síncrona información relativa a la planificación y otros aspectos no funcionales. Básicamente esta información incluye una prioridad global, una relación inicial para con el montículo, un tipo de invocación síncrona o con algún grado de asincronísmo, un identificador único que permite hacer un uso multiplexado del canal de comunicaciones y también un identificador de objeto remoto para uso interno.

  DRTSJ aunque especifica, parcialmente, lo que es la comunicación vertical no llega a especificar ningún tipo de interfaz horizontal de comunicación. Esto, aunque necesario, aún no ha sido abordado.

  RTRMI-York identifica la necesidad de transmitir información relacionada con la planificación entre los diferentes nodos de red pero en vez de proponer extensiones a JRMP propone que estos parámetros sean enviados como parámetros adicionales de la invocación, obligando a que sean serializables. Tampoco propone ningún tipo de prioridad global que mantenga cierta coherencia entre los diferentes nodos de la red, similar a la provista por RTCORBA o por DREQUIEMI.

  RTRMI-UPM aplica la misma solución que RTRMI-York, todos los parámetros son considerados como parámetros adicionales de la invocación.

- *Interfaz de recolección distribuida de basura de tiempo real: RTDGC*

En lo que son el conjunto de interfaces horizontales de comunicación, DREQUIEMI incluye una para el recolector distribuido de basura de tiempo real. Ésta permite que los diferentes nodos puedan incrementar y decrementar referencias a un objeto remoto residente en el nodo RMI local, coordinando la recolección de basura local con la distribuida.

DRTSJ no soporta recolección distribuida de basura de tiempo real. Es una característica de RMI que no se contempla en el modelo.

RTRMI-York no soporta recolección de basura distribuida de tiempo real aunque sugiere que sería interesante su incorporación.

RTRMI-UPM prohíbe el empleo de recolección de basura distribuida en sus dos perfiles de tiempo real.

4.3.2. Correspondencia entre DRTSJ y DREQUIEMI

DRTSJ, en sus diferentes niveles, provee nuevas interfaces remotas y clases capaces de soportar hilos, eventos asíncronos y transferencia asíncrona de control. Al igual que antes, intentaremos establecer relaciones entre cada una de esas características y el modelo de DREQUIEMI, a fin de ver cuál es grado de cobertura que DREQUIEMI ofrece a DRTSJ.

- *Interfaces remotas*: `RealtimeRemote` y `NoHeapRealtimeRemote`.

  DREQUIEMI provee garantías similares de forma más dinámica extrayendo dicha información de lo que es el tipo de hilo que realiza la invocación en el sustituto o de forma un poco más estática a partir de los parámetros del objeto remoto o del sustituto.

- *Hilos distribuidos*: `DistributedRealtimeThread` y `DistributedNoHeapRealtimeThread`.

  Actualmente DREQUIEMI no ofrece ningún tipo de soporte a esta funcionalidad. Su incorporación dentro del modelo de distribución de DREQUIEMI requeriría cambios tanto en sus interfaces como en su implementación.

- *Eventos asíncronos*: `GlobalAsyncEvent` y jerarquía `GlobalAsyncronousHandler`.

  DREQUIEMI provee una funcionalidad similar mediante el empleo de invocaciones remotas asíncronas tanto en el cliente como en el servidor. Aún así, existen dos diferencias clave entre ambos. La primera es que mientras en DRTSJ el modelo de comunicación utilizado es el *publisher-subscriber*, en DREQUIEMI es el *producer-consumer*. Y la segunda es que DREQUIEMI, al contrario que DRTSJ, permite acompañar este tipo de comunicaciones con un flujo de datos de la aplicación.

- *Transferencia asíncrona*: `RemoteFirer`, `RemoteAsynchronouslyInterruptedException` y `DistributedInterruptible`.

Al igual que en el caso de los hilos distribuidos, el soporte actual de DREQUIEMI tendría que ser extendido para darle soporte a dicha funcionalidad.

En términos generales no se puede decir que DREQUIEMI de una cobertura total al modelo de DRTSJ. Por una lado DREQUIEMI provee una buena cobertura tanto para el modelo de asincronía como para el de interfaces, proporcionando incluso mayores grados de flexibilidad que el propio DRTSJ. Pero por otro lado, la abstracción de los hilos distribuidos y transferencia asíncrona, tal y como se presenta en DRTSJ, no es soportable en la actualidad por DREQUIEMI.

4.3.3. Correspondencia entre RTRMI-York y DREQUIEMI

La aproximación RTRMI de York, tal y como veremos a continuación, está bastante bien soportada por el modelo DREQUIEMI. Sus principales características -interfaces remotas, sustituto de tiempo real, objeto remoto de tiempo real y el servidor de tiempo real- encuentran buenos pares en el modelo DREQUIEMI.

- *Interfaces remotas*: `RealtimeRemote`.

  DREQUIEMI no proporciona ningún tipo de interfaz que diferencie un objeto remoto tradicional de uno de tiempo real, sino que esta decisión se realiza a la hora de instanciar el objeto remoto, siendo por tanto más flexible.

- *Sustituto de tiempo real*: `RealtimeRemoteStub`.

  DREQUIEMI posee también una clase `RealtimeRemoteStub` pero su funcionalidad es diferente. En DREQUIEMI la clase estática sirve para asociar al sustituto la información no funcional que será utilizada durante la invocación remota y además no implica cambios en el *rmic*, pudiendo utilizarse sin realizarle modificaciones. La funcionalidad de seleccionar qué parámetros serán enviados al servidor en DREQUIEMI es tarea del planificador distribuido que tampoco requiere de un *rmic* especializado. Por tanto, se puede decir que aunque de manera distinta, DREQUIEMI da cobertura a esta característica del RTRMI de la Universidad de York.

- *Objeto remoto de tiempo real*: `UnicastRealtimeRemoteObject`.

  El `RealtimeUnicastRemoteObject` de DREQUIEMI ofrece una funcionalidad similar a la de esta clase. Permite seleccionar el puerto y los parámetros de planificación pero no permite asociar un *threadpool*, pues en el modelo de DREQUIEMI existe un único *threadpool* global que es asociado al planificador distribuido. Además de estos parámetros, DREQUIEMI, ofrece la posibilidad de definir otros parámetros como son el `ReleaseParameters` o los `MemoryParameters`, lo que dota al sistema de una mayor flexibilidad a la hora de soportar diferentes políticas de planificación. Y por tanto, se podría decir que existe una buena cobertura de dicha característica.

- *Servidor de tiempo real*: `UnicastAcceptorRealtime`.

Esta clase permite controlar el grado de paralelismo máximo que va a ser soportado en el servidor, utilizando para ello un *threadpool*.

En DREQUIEMI, esta funcionalidad está cubierta por el planificador distribuido; el cual, haciendo uso de un *threadpool* y un *connectionpool* gestiona lo que es el grado de paralelismo alcanzable en el sistema distribuido.

Como conclusión más general, se podría decir que el modelo RTRMI-York encuentra un buen soporte dentro del modelo DREQUIEMI. En mayor o menor grado todas sus principales características encuentran buenos equivalentes en el modelo DREQUIEMI.

4.3.4. Correspondencia entre RTRMI-UPM y DREQUIEMI

Dado que las interfaces HRTRMI y QoSRMI son muy similares estudiaremos tan solo HRTRMI. Los resultados que se obtengan para ésta serán extrapolables a QoSRMI pues éste define interfaces paralelas a las de HRTRMI. Y así, tan sólo se hará referencia a QoSRMI cuando resulte necesario.

- *Interfaces remotas*: `HrtRemote`, `HrtRemoteRef` y `HrtServerRef`.

  DREQUIEMI retrasa todas las decisiones hechas por estas interfaces estáticas a fases de la ejecución del programa, lo que le dota de una mayor flexibilidad que la proporcionada por el esquema de clases estáticas de la Universidad Politécnica de Madrid.

- *Sustituto de tiempo real*: `UnicastHrtRemoteStub`.

  Conceptualmente esta clase es similar a `RemoteRealtimeStub` de la Universidad de York. Y por tanto y al igual que en el caso previo, DREQUIEMI le continúa dando un buen soporte.

- *Objeto remoto de tiempo real*: `UnicastHrtRemoteObject`

  Esta clase permite fijar ciertas características de planificación relativas a lo que es el objeto remoto de tiempo real. Una vez fijadas, éstas no pueden ser variadas. Esto en DREQUIEMI también es posible, mediante una serie de parámetros que pueden ser fijados cuando se crea el objeto remoto y que pueden ser modificados dinámicamente durante la ejecución del programa.

Al igual que sucedía con RTRMI-York, el modelo RTRMI-UPM parece ser totalmente soportable pues la funcionalidad que oferta es equivalente a la que se puede conseguir mediante otras interfaces de DREQUIEMI.

4.3.5. Síntesis

A modo de resumen final, con los resultados de esta comparación hemos construido el cuadro 4.1. En él se muestra de forma sintética lo que son las relaciones existentes entre las diferentes interfaces de tipo RTRMI y DREQUIEMI así como las relaciones inversas que se pueden establecer tomando como punto de partida

DREQUIEMI. Se han definido tres grados de relación: $\times$, $\trianglelefteq$ y $\surd$, significando respectivamente que no se encuentra una equivalencia, que existe cierta equivalencia y que hay una buena equivalencia. Las casillas en blanco significan que la equivalencia no ha sido analizada.

Tras haber analizado lo que son las relaciones existentes entre los diferentes modelos de interfaces definidos y DREQUIEMI, así como la relación inversa, se puede decir que el mayor aporte realizado por DREQUIEMI se concentra alrededor de las interfaces del `DistributedScheduler` y la caracterización de un protocolo horizontal RTJRMP y una interfaz RTDGC. El planificador distribuido, al igual que el centralizado de RTSJ, permite la incorporación de técnicas de gestión diversas sin que ello implique cambios en la implementación de los objetos remotos. Las dos interfaces de comunicación horizontal permiten la transmisión de información no funcional entre los diferentes nodos de la red.

Correspondencia entre: / y:		DRTSJ	RTRMI-York	RTRMI-UPM	DREQUIEMI
DREQUIEMI	Objetos remotos de tiempo real	$\times$	$\surd$	$\surd$	
	Planificador distribuido	$\times$	$\times$	$\trianglelefteq$	
	Invocaciones remotas asíncronas	$\trianglelefteq$	$\times$	$\times$	
	Protocolo rtjrmp	$\times$	$\trianglelefteq$	$\trianglelefteq$	
	Interfaz rtdgc	$\times$	$\times$	$\times$	
	Sustituto de tiempo real	$\times$	$\trianglelefteq$	$\trianglelefteq$	
DRTSJ	Interfaces remotas de tiempo real				$\surd$
	Hilos distribuidos				$\times$
	Eventos asíncronos				$\surd$
	Transferencia asíncrona				$\times$
RTRMI-York	Interfaces remotas de tiempo real				$\surd$
	Sustituto de tiempo real				$\surd$
	Objeto remoto de tiempo real				$\surd$
	Aceptor de tiempo real				$\surd$
RTRMI-UPM	Interfaces remotas de tiempo real				$\surd$
	Sustituto de tiempo real				$\surd$
	Objeto remoto de tiempo real				$\surd$

Cuadro 4.1: Relaciones directas e inversas entre DREQUIEMI y otras aproximaciones a RTRMI

Por contra, las interfaces para el sustituto de tiempo real y el objeto remoto de tiempo real son las que ofrecen un menor grado de aporte al estado del arte, siendo el soporte de un modelo de asincronismo entre el cliente y el servidor otra característica bastante novedosa.

4.4. Conclusiones y líneas futuras

En este capítulo, haciendo uso del modelo desarrollado en el capítulo anterior, hemos propuesto una extensión para RMI que permite el desarrollo de aplicaciones de tiempo real, denominada como DREQUIEMI. DREQUIEMI define, al igual que DRTSJ, RTRMI-York y RTRMI-UPM, interfaces de comunicación verticales, con el programador, y horizontales, entre los diferentes nodos de la red. Verticalmente se incluye el sustituto de tiempo real y el objeto remoto de tiempo real, proveyéndose además un nueva entidad, el planificador distribuido, encargado de gestionar los recursos de diferentes nodos de forma conjunta. Mientras que horizontalmente se han propuesto dos extensiones, una de ellas adaptando el protocolo JRMP a las necesidades de transmisión de información de tiempo real entre diferentes nodos de la red y otra definiendo una interfaz de recolección distribuida de basura de tiempo real. Por último, se han comparado estas interfaces con las definidas por el resto de aproximaciones a RTRMI, obteniéndose que DRTSJ encuentra un soporte parcial por parte de DREQUIEMI y que tanto los modelos de RTRMI-York como el de RTRMI-UPM son soportables por DREQUIEMI. En la comparación inversa destaca que la definición de mecanismos de comunicación horizontal, soportada en DREQUIEMI, no se encuentra en el resto de las aproximaciones.

La principal línea de trabajo que se ha identificado es la de integrar dentro del modelo propuesto tanto el paradigma de hilo distribuido de tiempo real como la transferencia asíncrona de control. Tras haber comparado las diferentes aproximaciones RTRMI con DREQUIEMI se han detectado una serie de limitaciones en las interfaces de DREQUIEMI que de ser solventadas permitirían realizar un mejor soporte de las características de DRTSJ por parte de DREQUIEMI. Casi todas derivan de la imposibilidad de soportar, de forma práctica, el modelo de hilo distribuido junto a un mecanismo distribuido de transferencia asíncrona de control. Y así, aunque el modelo de clases DREQUIEMI les podría dar cabida mediante nuevos planificadores distribuidos y nuevas extensiones en el protocolo de comunicaciones RTJRMP, el problema es más complejo pues en el caso general se requiere de una sincronización temporal entre los diferentes nodos de la red, así como llegar a un cierto acuerdo global sobre cómo se intercambia la información de planificación.

En el siguiente capítulo se complementará el modelo propuesto con una serie de extensiones, RTSJ++, que acercan el modelo de computación de RTSJ al de Java facilitando, en consecuencia, tanto la implementación de DREQUIEMI como, en general, el desarrollo de otras aplicaciones Java de tiempo real.

Capítulo 5

Extensiones para Java de tiempo real centralizado

RTSJ es una especificación bastante joven que aún ha de sufrir múltiples transformaciones antes de llegar a un grado de madurez que la convierta en un producto de gran estabilidad. De hecho, en esa línea, existe una iniciativa dentro de los *Java Community Processes*, conocida como RTSJ 1.1 [92], que propone mejoras al modelo computacional de RTSJ intentando arreglar diferentes limitaciones que se han observado en su versión anterior, la 1.0. En algunos puntos, como puede ser la regla bidireccional de asignación, está previsto relajar el modelo de RTSJ 1.0 y en otros como puede la revisión de algunas interfaces como las de eventos asíncronos o las de recolección de basura, se persigue arreglar ciertas limitaciones del modelo actual. En nuestro caso particular, partiendo de la experiencia previa acumulada en la implementación del prototipo de DREQUIEMI, se ha desarrollado una serie de mejoras a RTSJ, denominadas en su conjunto RTSJ++, que son de utilidad al desarrollador tanto a la hora de implementar el middleware de distribución como aplicaciones centralizadas de tiempo real.

Una gran parte de la polémica que rodea a RTSJ ha girado sobre la gestión de memoria, sobre si era necesario o no un sistema alternativo al de regiones. Se ha criticado el nuevo sistema de referencias de RTSJ, que impone restricciones como la regla de asignación y del padre único, no presentes en Java tradicional; el modelo de regiones que considera a la `ScopedMemory` como un mecanismo demasiado complejo de utilizar por el programador tradicional Java; y por último también se ha criticado el modelo de computación por incluir dos tipos de hilos `-NoHeapRealtimeThread` y `RealtimeThread-` que dan pie a la aparición de dos entornos de programación diferenciados.

Nuestra experiencia con el manejo de estas tres limitaciones ha dado lugar a tres extensiones: `AGCMemory`, `ExtendedPortal` y `RealtimeThread++` que conforman RTSJ++ y que de forma individualizada atacan a estos tres problemas, acercando el modelo de computacional de RTSJ al de Java tradicional. Así, la `AGCMemory` [17] [16] facilita la utilización del modelo de regiones en aplicaciones, mediante el soporte de cierto tipo de recolección de basura predecible dentro del modelo de regiones de RTSJ, reduciendo la necesidad de recurrir a regiones anidadas y aproximando

el modelo de regiones al del recolector de basura. Por otro lado, el `ExtendedPortal` [18] posibilita la realización de violaciones de la regla de asignación de forma segura e incorpora modelos de navegación sencilla dentro del árbol de regiones de RTSJ, lo que en aplicaciones no triviales, con muchas regiones anidadas, facilita el establecimiento de referencias prohibidas. Por último, el `RealtimeThread++` rompe el dualismo computacional existente en el actual RTSJ, posibilitando que un único hilo de tiempo real dinámicamente elija la relación de dependencia que desea mantener con el recolector de basura. En conjunto, siguiendo el espíritu de RTSJ 1.1, se puede decir que todas estas mejoras están encaminadas a proveer un RTSJ más versátil y sencillo de utilizar.

Desde el punto de vista de DREQUIEMI, cada una de estas tres extensiones facilita su implementación en un determinado aspecto. Así, la `AGCMemory` se puede entender como una optimización válida a la hora de reducir el consumo dinámico de memoria realizado tanto por el servidor como por el cliente durante el transcurso de una invocación remota durante la fase de serialización y deserialización de información. A la hora de implementar DREQUIEMI, el `ExtendedPortal` posibilita el almacenamiento de referencias a objetos remotos creados en memoria de tipo `ScopedMemory` dentro de memoria inmortal, así como su posterior recuperación. Y por último, la aproximación `RealtimeThread++` facilita la implementación de los *threadpools*, siendo capaz de reducir además el número de hebras necesarias para dar soporte a una determinada aplicación.

La organización del capítulo presenta cada una de las extensiones de forma individualizada motivándolas, presentando interfaces para ellas, dando ciertos detalles sobre los cambios que habría que realizar a bajo nivel para su implementación y ofreciendo también una serie de conclusiones y líneas futuras. La sección 5.1 presenta la `AGCMemory` desde esa triple perspectiva, la sección 5.2 presenta al `ExtendedPortal` y último la sección 5.3 al `RealtimeThread++`. Por último, cierra el capítulo la sección 5.4 con las conclusiones y líneas futuras.

5.1. Recolección de basura flotante en regiones

Son muchos los sistemas, entre ellos los de gran escala, que pueden beneficiarse de las características de los lenguajes de alto nivel de abstracción para ver reducidos tanto sus costes de desarrollo como de mantenimiento. En estos sistemas, las especiales características de lenguajes como Java -portabilidad, gestión automática de memoria, simplicidad y soporte para la red- pueden llegar a abaratar notablemente dichos costes. Pero sin embargo, cuando se introduce el término "tiempo real", esto ya cambia pues muchos de los mecanismos, en especial la gestión automática de memoria, a pesar de ser capaces de reducir los costes de desarrollo, también obstaculizan los plazos de las diferentes tareas de tiempo real. Por ello, RTSJ ofrece modelos alternativos de medio nivel, basados en el modelo de regiones, que a cambio de ser más predecibles requieren de una mayor colaboración por parte del programador.

Desgraciadamente, el problema de producir una solución para la gestión automática de memoria, en Java, carece de una solución perfecta. El candidato natural, el recolector de basura de tiempo real, es capaz de ofrecer cotas máximas de interfe-

rencia que pueden ser incluso planificadas, pero sin embargo el coste en términos de consumo adicional de memoria y de procesador puede disparar el coste final del sistema. En el otro extremo, dejar que la gestión de memoria recaiga directamente en el programador tampoco es una buena política en Java, pues su gran dinamismo nos forzaría a reimplementar muchas de las librerías actuales; un coste extra que restaría atractivo a su empleo. Por ello, las dos principales especificaciones para Java de tiempo real: RTCORE y RTSJ, ofrecen mecanismos alternativos basados en regiones.

En el caso de RTSJ, éstas tienen a la `ScopedMemory` como clase raíz de la que todas las regiones heredan. Este tipo de memoria permite realizar una reserva y liberación de memoria ligada a lo que es la vida de la región y adicionalmente, utilizando una técnica de contaje, es capaz de eliminar todos los objetos en ella almacenados, recuperando la memoria previamente reservada. Su gran limitación, abordada por la `AGCMemory`, es la incapacidad de eliminar colecciones parciales de objetos particulares, limitándose a realizar un todo o nada. Por el contrario, la `AGCMemory` permite eliminar colecciones parciales de objetos, de forma transparente al programador y con garantías de predictibilidad. Así, se puede entender que la `AGCMemory` lo que hace es recuperar parte de la gestión automática, provista por el recolector de basura, en el contexto de las regiones, mejorando el grado de portabilidad ofrecido por éstas.

Más concretamente, el tipo de basura que es capaz de detectar y de eliminar la `AGCMemory` es la flotante. Ésta es aquella más sencilla que se produce cuando durante la ejecución de un método Java se realiza una reserva de memoria temporal para crear objetos Java cuya vida no sobrepasa la del método invocado y que, por tanto, tras la finalización de éste se pueden eliminar.

En el caso específico de DREQUIEMI, este tipo de región mejorada permite reducir tanto el número como el tamaño de las regiones utilizadas durante una invocación remota para eliminar objetos temporales creados durante el envío y la recepción de datos entre el nodo cliente y el servidor.

A continuación veremos cómo las diferentes aproximaciones existentes en el estado del arte han tratado este problema -sección 5.1.1- para después ver un ejemplo -sección 5.1.2- que sirve de motivación para la `AGCMemory`, siguiendo con la caracterización de una interfaz de programador y de detalles de implementación -sección 5.1.3. Ya finalizando, aparecen las conclusiones y de líneas futuras -sección 5.1.4.

5.1.1. Punto de partida

La comunidad investigadora Java de tiempo real ha sido muy sensible a lo que es el modelo de regiones de RTSJ, dedicando muchos esfuerzos a comprender las diferentes implicaciones que el uso de este mecanismo tiene tanto en el programador como en el entorno de ejecución. Dos son los grandes problemas que presenta la `ScopedMemory`: el de la eficiencia computacional y el de la asignación de regiones a porciones de código. Las comprobaciones que han de ser realizadas durante la ejecución de la máquina virtual para garantizar que las reglas del padre único y de asignación no son violadas, suponen un coste extra que puede alcanzar una complejidad $\Theta(n)$ que puede repercutir significativamente en el rendimiento final del sistema. Por otro

lado, en RTSJ las regiones han de ser asociadas a porciones de código haciendo uso de instancias de objetos `java.lang.Runnable`, lo que supone un cierto conocimiento, en tiempo de generación del código, sobre la vida de cada uno de los objetos del sistema. Además, dado que cada región maneja una cantidad de memoria finita, es necesario que cada `ScopedMemory` sea dimensionada individualmente.

El tema de la eficiencia computacional ha sido ampliamente analizado en el estado del arte, siendo los trabajos de Corsaro [45] y Higuera-Toledano [76] unos de los más maduros. Corsaro propone el uso de un mecanismo de espejos que sea capaz de reducir la complejidad de verificación de la regla de asignación de $\Theta(n)$ a $\Theta(1)$ e Higuera-Toledano extendiendo dicha técnica propone técnicas capaces de verificar la regla del padre único con una complejidad acotable por una función $\Theta(1)$. Siguiendo estas mismas ideas, las barreras utilizadas internamente por la `AGCMemory` tratan también de mantener una complejidad acotable por una función $\Theta(1)$.

Otro trabajo altamente relacionado es el de Deters [50] donde se estudia la asignación automática de regiones a código Java mediante el uso de técnicas basadas en el análisis de escape. Utilizando esta técnica se consigue que el número de regiones que el programador ha de introducir de forma manual en su código se vea reducido notablemente. Siguiendo esta línea de pensamiento, la `AGCMemory` implementa un algoritmo de escape capaz de asignar conjuntos de objetos a regiones de forma dinámica durante la ejecución del código Java.

Por último, otro de los puntos de partida de la `AGCMemory` son los *stackableobjects* de RTCORE. En RTCORE mediante este modificador se puede indicar que un objeto en vez de residir en el montículo, lo haga en la pila local del hilo. La `AGCMemory` cubre esta misma funcionalidad permitiendo eliminar objetos creados en ella tras la invocación de un método Java.

Desde el punto de vista práctico, la `AGCMemory` puede entenderse como una combinación de diferentes propiedades de los trabajos descritos anteriormente acompañada de ciertos ajustes específicos. Como en el trabajo de Corsaro, los mecanismos de validación presentan complejidad $\Theta(1)$, pero a diferencia de éste, aparecen nuevas barreras ligadas a la invocación de los métodos Java tanto en el momento de iniciar la invocación como en el de finalizarla. Como en el trabajo de Deters se utilizan técnicas de análisis de escape, pero a diferencia de éste, éstas son ejecutadas dinámicamente con el código. Y por último, como en los *stackableobjects* se permite eliminar objetos tras la finalización de un método Java, pero en la `AGCMemory` y a diferencia de los *stackableobjects*, se permite que éstos sobrevivan a la ejecución del método.

5.1.2. Recolección de basura flotante

En primer lugar convendría comprender un poco más lo que se ha denominado como basura flotante. Una forma sencilla de entender este concepto nos la ofrece el método Java `System.out.println(1)`. Cada vez que se invoca este método se procede a crear una serie de objetos temporales que son utilizados para imprimir el número 1 que tras la invocación del método ya no son referenciados desde ningún otro objeto Java sino que son inalcanzables. O dicho de otra manera, que se convierten en basura flotante.

```
System.out.println(1); //Puede producir basura flotante
```

Y esto es problemático porque dependiendo de la implementación concreta la cantidad de basura flotante generada puede variar enormemente. Y así, esta operación de imprimir un número podría consumir 88 bytes, 0 bytes o incluso 1 Mb, dependiendo de la máquina virtual y las clases utilizadas. [1] Lo que desde el punto de vista práctico supone una reducción en lo que es el nivel de portabilidad de las aplicaciones desarrolladas pues éstas han de ser adaptadas a las peculiaridades de cada plataforma de ejecución.

Para ilustrar el problema que se produce desde el punto de vista del programador con la basura flotante hemos escogido un aplicación sencilla, un hilo de tiempo real que de forma periódica incrementa el valor de una variable interna `counter`. El código completo se puede consultar en la figura 5.1. En el constructor del hilo, en la línea 11, se asocia una instancia de tipo `LTMemory` a lo que es el método `run`, de tal manera que todos los objetos instanciados en el método `run` son creados dentro de esa región. Esto es, cada vez que se invoca al método `println`, todos los objetos creados durante su invocación son creados en dicha región.

Pero sin embargo, tal y como se ha codificado, el programa no funciona correctamente en todas las máquinas virtuales. Tal y como se muestra de forma gráfica en el perfil de consumo de memoria para dicha aplicación en el caso particular de jTime -figura 5.1-, tras la cuarta invocación, la totalidad de la memoria reservada es consumida provocando un *out-of-memory-error*. Ello es debido a que internamente se crean objetos temporales para imprimir el valor del entero que no son eliminados tras la ejecución del método `println`, lo que hace que inexorablemente se tienda a ocupar toda la memoria disponible en la `LTMemory`. Los objetos temporales serían eliminados cuando finalizase el método `run` pero sin embargo, dado que el hilo ejecuta un bucle infinito, nunca llegan a ser borrados.

En RTSJ, utilizando la técnica de regiones anidadas se pueden eliminar los objetos temporales creados durante la invocación a `println`. Básicamente para operar de dicha manera es necesario crear una nueva instancia de `LTMemory` -que en la figura 5.2 es denominada `lt-` que elimina los objetos temporales y un nuevo objeto *runnable* que sirve de cápsula al código que queremos imprimir por pantalla. El resultado, tal y como se muestra en el perfil de consumo de memoria de la aplicación, es que tras finalizar el método `enter(r)` se produce una recuperación de la memoria que había sido utilizada durante la invocación del método.

Pero sin embargo, esto no ha sido conseguido de forma transparente al programador. Ha sido necesario colocar una nueva región `-lt-` a la que se ha asignado cierto código `-impr-` y además ha sido necesario dimensionarla adecuadamente (con `150` bytes). Y esto es problemático por varios motivos pues en función de la máquina virtual y las librerías que se utilicen tanto el código que debería de ser asociado al objeto `runnable` como el tamaño de la memoria de la región auxiliar utilizada variarían notablemente.

Solventar esto con la `AGCMemory` es sencillo pues de forma automática y transpa-

[1] En el caso de jTime la operación de imprimir el número 1 consume 88 bytes.

```
01: import javax.realtime.*;
02: public class PeriodicCounter extends RealtimeThread{
03:    public PeriodicCounter(){
04:           super( null, //Schedulling Parameters
05:                  new PeriodicParameters(null,
06:                  new RelativeTime(1000,0),//T
07:                  new RelativeTime(50,0),  //C
08:                  new  RelativeTime(100,0),//D
09:                  null,null);
10:                  null,
11:                  new LTMemory(250,250),
12:                  null);
13:           start(); //starts thread
14:           }//@constructor

15:    int counter=1;

16:    public void run(){
17:           do{System.out.println(counter);
18:               counter++;}while(waitForNextPeriod());
19:           }//@run
20:    public static void main(String s[]){
21:           new PeriodicCounter();
22:           }
23:    }
```

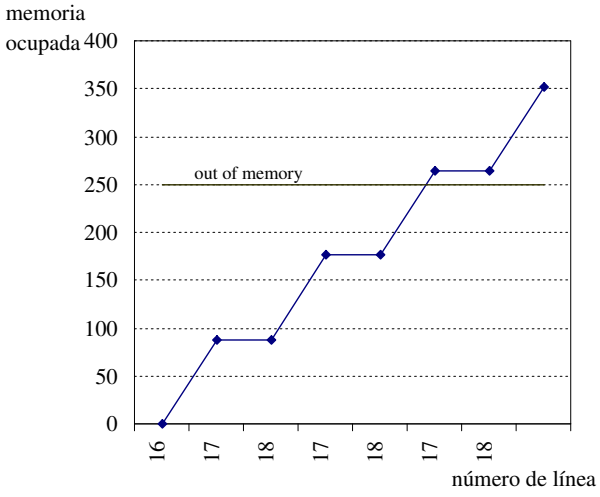


Figura 5.1: Código de la aplicación `PeriodicCounter` y perfil de consumo de memoria

rente al programador, este tipo de región se encarga de detectar si tras la ejecución de un método se puede eliminar la basura flotante o no. Tal y como se muestra en la línea 11 de la figura 5.3, si en el código de partida se asocia una `AGCMemory` en vez de una `LTMemory`, el perfil de consumo de memoria es el mismo que en el caso de que utilicemos anidamiento de regiones. Y lo que es más importante, no resulta necesario definir ningún tipo de región auxiliar ni ningún tipo de objeto *runnable*.

Las ventajas obtenidas son claras, el código generado es más sencillo pues ya no es necesario crear regiones auxiliares y es también más mantenible pues se reducen las dependencias para con el modelo de regiones.

```
01: import javax.realtime.*;
02: public class PeriodicCounter extends RealtimeThread{
03:    public PeriodicCounter(){
04:           super( null, //Schedulling Parameters
05:                  new PeriodicParameters(null,
06:                  new RelativeTime(1000,0), //T
07:                  new RelativeTime(50,0),   //C
08:                  new  RelativeTime(100,0), //D
09:                  null,null);
10:                  null,
11:                  new LTMemory(250,250),
12:                  null);
13:                   start();
14:             }
15:    int counter=1;
16:    public void run(){
17:                  do{ lt.enter(impr);
18:                  counter++;}while(waitForNextPeriod());
19:           }
20:    LTMemory lt=new LTMemory(150,150);
21:    Runnable impr=new Runnable(){
22:                  public void run(){
23:                  System.out.println(counter);};
24:    public static void main(String s){
25:                  new PeriodicCounter();
26:           }
27:    }
```

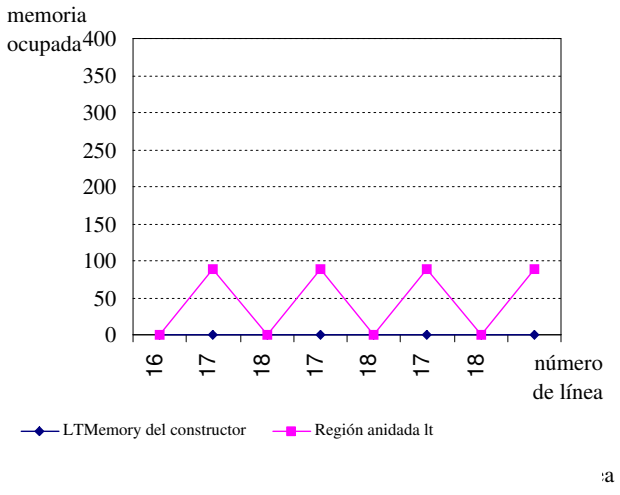


Figura 5.2: Recolectando basura flotante utilizando regiones anidadas

5.1.3. Modificaciones requeridas

A la hora de proveer soporte para la `AGCMemory` se deben de tener en mente dos grandes cuestiones. La primera de ellas está relacionada con el programador y consiste en la definición de unas interfaces que estén alineadas con RTSJ y que permitan su utilización. La segunda es la de especificar, de alguna manera, los cambios que

```
01: import javax.realtime.*;
02: public class PeriodicCounter extends RealtimeThread{
03:    public PeriodicCounter(){
04:           super( null, //Schedulling Parameters
05:                  new PeriodicParameters(null,
06:                  new RelativeTime(1000,0),//T
07:                  new RelativeTime(50,0),  //C
08:                  new  RelativeTime(100,0),//D
09:                  null,null);
10:                  null,
11:                  new AGCMemory(250,250),
12:                  null);
13:           start(); //starts thread
14:           }//@constructor

15:    int counter=1;

16:    public void run(){
17:           do{System.out.println(counter);
18:               counter++;}while(waitForNextPeriod());
19:           }//@run
20:    public static void main(String s[]){
21:           new PeriodicCounter();
22:           }
23:    }
```

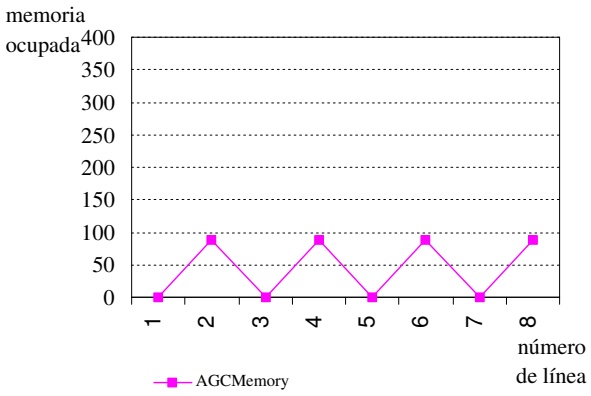


Figura 5.3: Recolección de basura flotante con la `AGCMemory`

son necesarios en el esquema de máquina virtual de tiempo real actual para ofrecer soporte al mecanismo de regiones.

Integración en el modelo de interfaces d RTSJ

La jerarquía de clases de RTSJ ofrece múltiples vías de integración para la `AGC-Memory`. En primer lugar podría ser una subclase de la `MemoryArea` pero el hecho de que presente características comunes con la `LTMemory` o la `VTMemory` nos ha llevado a colocarla a su misma altura, como subclase de la `ScopedMemory`. Tampoco se ha

querido que fuese una subclase de la `LTMemory` o de la `VTMemory`, en gran parte debido a la incapacidad de saber cuál de las dos era la más próxima conceptualmente. Al igual que la `LTMemory`, la `AGCMemory` puede presentar tiempos de creación de objetos acotables por una función lineal y de forma similar a lo que ocurre en la `VTMemory` permite la eliminación parcial de los objetos contenidos en ella. Por tanto, al final se le ha dado un nivel de protagonismo similar al de estas dos clases, poniéndola a su mismo nivel tal y como se muestra en la figura 5.4.

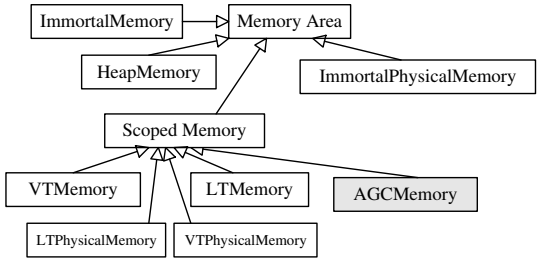


Figura 5.4: Insertando la `AGCMemory` dentro de la jerarquía de clases de RTSJ

La `AGCMemory` no define ningún tipo de método diferente, a excepción claro está del constructor, de los definidos en la clase abstracta `ScopedMemory`. Sus métodos `enter` y `executeInArea` no son diferentes a los de la `LTMemory` o los de la `VTMemory`, lo que desde el punto de vista del programador implica que los posibles costes de aprendizaje asociados a su utilización permanecerán bajos. Su constructor, de forma semejante al de la `LTMemory`, permite hacer una reserva inicial de memoria que más adelante será utilizada para el almacenamiento de objetos.

Funcionamiento de bajo nivel

Al igual que ocurre con el resto de *memoryareas* de RTSJ, existen múltiples formas de implementar el modelo propuesto. En el presente caso, se explorará una vía basada en la utilización de memoria no compartida, lo que debidamente combinado con barreras de ejecución de complejidad $\Theta(1)$ y un algoritmo de escape dinámico, nos permitirá eliminar la basura flotante generada durante la ejecución de un método Java tras su ejecución.

Restricciones impuestas por el algoritmo

Dos son las restricciones impuestas por el algoritmo utilizado:

- *La no compartición de la región.* Una `AGCMemory` tan sólo puede ser utilizada al mismo tiempo por un hilo de tiempo real. De esta manera, si un hilo intenta incorporarla en su *scopestack*, utilizando por ejemplo `enter`, el hilo recibirá una excepción `ScopedCycledException`, si ésta está siendo utilizada por otro hilo. Y por tanto, cuando sea necesaria la compartición de datos entre varios hilos, se deberá de recurrir a mecanismos ya presentes en RTSJ como son las colas de mensajes o los portales.

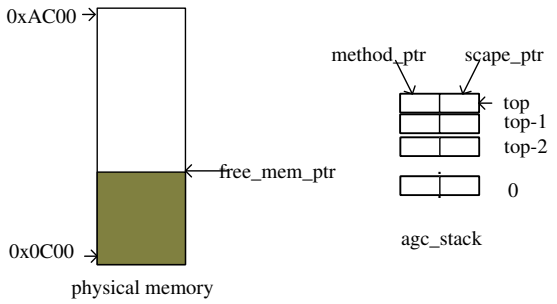


Figura 5.5: Estructuras de datos manejadas internamente por la `AGCMemory`

- *Capacidad de detección y de eliminación de basura flotante.* El algoritmo diseñado es capaz, una vez que ha finalizado un determinado método, de eliminar todos los objetos que habían sido creados durante su ejecución. La granularidad alcanzada no es total y se limita a eliminar la totalidad de objetos creados o ninguno de ellos, no siendo posible la eliminación de conjuntos de objetos particulares.

Estructuras de datos

Tal y como muestra la figura 5.5, cada instancia de `AGCMemory` consta de un bloque de memoria donde se crean los objetos y de una estructura de datos denominada `agc_stack` que es utilizada para detectar la basura flotante. El bloque de memoria se direcciona linealmente y cada vez que se crea un objeto el puntero `free_mem_ptr` se incrementa reservando la memoria necesaria para almacenar el estado de dicho objeto. De la misma forma, cuando los objetos se eliminan, el valor del puntero `free_mem_ptr` es decrementado. Cada entrada del `agc_stack` consta de dos campos: el `method_ptr` y el `scape_prt`. El primero almacena el valor de `free_mem_ptr` justo antes de que de inicio la invocación al método Java. El segundo, modificado por una barrera ejecutada de forma dinámica durante el método Java, permite averiguar tras la ejecución de éste si se puede recuperar la memoria consumida por el método.

Barreras encargadas de detectar y de eliminar la basura

Para ser capaz de detectar y de eliminar la basura flotante, la máquina virtual ha de ejecutar de forma dinámica cierto código que modifica los datos del `agc_stack` para conseguir detectar y eliminar la basura flotante. Los datos almacenados en el `agc_stack` son utilizados y modificados cada vez que da comienzo un método Java, durante su ejecución y cuando éste finaliza por tres tipos de barreras: la de *preinvocación*, la de *asignación* y la de *postinvocación*. La de preinvocación sirve para saber qué objetos son creados durante la invocación de un determinado método Java. La de asignación, cada vez que un objeto es referenciado intenta saber si lo es desde objetos externos o no. Y por último, la de postinvocación o bien elimina los objetos creados durante la invocación o bien delega esta tarea al método Java padre.

1. *Barrera de preinvocación.*

La barrera de preinvocación es ejecutada justo antes de que comience la ejecución de un método Java. Su función es la de inicializar la estructura de datos `agc_stack`, introduciendo una nueva entrada. El valor con que se inicializan cada uno de sus dos campos es el mismo: `free_mem_ptr`.

$$agc\_stack[top].scape\_ptr \Leftarrow free\_mem\_ptr$$

$$agc\_stack[top].method\_ptr \Leftarrow free\_mem\_ptr$$

Esto nos permite determinar qué objetos son creados en el método Java que se va a ejecutar y cuáles, por el contrario, pertenecen a métodos Java padres.

2. *Barrera de postinvocación*

La barrera de postinvocación es ejecutada justo tras la finalización de un método Java, antes de que tome el control la rutina que lo había invocado. Esta barrera realiza dos acciones, en primer lugar elimina una entrada del `agc_stack` y la segunda es decir sobre lo que se hace con los objetos creados durante la invocación de un método Java. Se observan dos opciones: (1) destruirlos o (2) delegar su destrucción al método desde el cual ha sido invocado el presente.

Para tomar esa decisión se realiza la siguiente comprobación:

$$agc\_stack[top].scape\_ptr \geq agc\_stack[top].method\_ptr$$

En el caso de que sea cierta, significa que tras la invocación de un método, todos los objetos creados durante él se han convertido en basura pudiendo ser eliminados. Y en este caso, se hace mediante la siguiente operación:

$$free\_mem\_ptr \Leftarrow agc\_stack[top].method\_ptr$$

Por el contrario, en el caso de que no se cumpla la condición, se propaga la responsabilidad de su destrucción al método Java que ha invocado al que estaba ejecutando la barrera.

Para ello, se realiza la siguiente operación:

$$agc\_stack[top{-}1].scape\_ptr \Leftarrow min\{agc\_stack[top{-}1].scape\_ptr, agc\_stack[top].scape\_ptr\}$$

Esta operación lo que hace es delegar la posibilidad de la destrucción de los objetos en el método Java padre. En este caso, será más tarde, durante barrera de post-invocación del método padre, cuando se decida si se borrarán los objetos del método invocado o no.

3. *Barrera de asignación*

Por último existe una última barrera, encargada de modificar el `scape_ptr` que es ejecutada antes de cada asignación a referencia. El propósito de ésta es detectar y anotar adecuadamente si alguno de los objetos creados durante la ejecución del método Java es referenciado desde objetos externos.

Para ello, dada una referencia a un objeto `-ref-` que está siendo tratada de ser asignada a un atributo de otro objeto remoto `-atrib-`, la barrera de asignación que se ejecuta cuando ambos objetos residen en el misma instancia de `AGCMemory` consiste en la realización de la siguiente comprobación:

$$(ref \geq attrib)$$

En el caso de que el resultado sea cierto, se ha de realizar la siguiente actualización en la estructura `agc_stack`:

$$agc\_stack[top].scape\_ptr \Leftarrow min\{agc\_stack[top].scape\_ptr, attrib\}$$

Esto es, en cada asignación se comprueba si un objeto es referenciado desde otro más antiguo. Y en caso de que sea cierto y que por tanto escape, se comprobará si el que ha escapado ha sido el que lo ha hecho a la posición más lejana de memoria o no, utilizando para ello la función mínimo.

Un detalle importante es que todas las barreras carecen de bucles, lo que nos permite acotar la sobrecarga computacional introducida por una función de complejidad $\Theta(1)$.

5.1.4. Conclusiones y líneas futuras

En esta sección se ha propuesto una extensión al modelo actual de regiones de RTSJ denominada `AGCMemory` cuyo principal aporte al modelo de computación de RTSJ es el de proveer un modelo de regiones más flexible, capaz de eliminar la basura flotante generada durante la invocación de los métodos Java. Se ha mostrado su utilidad mediante un sencillo caso de estudio donde se ha visto cómo el programador puede obtener beneficios tales como reducciones en el número de regiones que necesita introducir de forma explícita en el código de sus aplicaciones. Después, se ha analizado el marco tecnológico en el que se enmarca la extensión, proponiendo un encuadre dentro de la jerarquía actual de clases de RTSJ y caracterizando una serie de barreras adicionales, de complejidad añadida $\Theta(1)$, que permiten que la máquina virtual pueda recolectar basura flotante de forma dinámica.

Dos son las principales líneas de trabajo que surgen tras proponer la `AGCMemory`, la primera de ellas es la de buscar algoritmos que permitan la compartición de los objetos de una misma región entre varios hilos y la segunda consiste en determinar con un mayor grado de exactitud cuál es su dominio de aplicación. Una de las principales limitaciones introducidas por la `AGCMemory`, que permite realizar implementaciones de complejidad $\Theta(1)$, es la imposibilidad de ser compartida entre dos hilos. En esta misma línea, sería muy interesante el estudio de soluciones que fuesen capaces de eliminar esta restricción manteniendo, en la medida de lo posible, la cota de complejidad en $\Theta(1)$. Por último, mediante un sencillo caso de estudio se ha mostrado

que es una aproximación que resulta interesante pero no se ha llegado a determinar exactamente cuál es su dominio de aplicación. En este sentido, los resultados de Dibble [52], donde se muestra que alrededor del 50 % de los métodos Java pueden crear objetos durante su invocación, sugieren que el potencial de la `AGCMemory` es verdaderamente alto.

5.2. Modelo de referencias extendidas

Una de las principales complicaciones que presenta el modelo de regiones de RTSJ es que fuerza al programador a elaborar sus programas de una forma diferente a la que está acostumbrado en Java. El modelo de programación de RTSJ, imponiendo las reglas de programación conocidas como la del *padre único* y la de *asignación* sobre el modelo de computación, se convierte en una traba para el programador pues este modelo es más restrictivo que el presente en Java tradicional donde no existen tales reglas, impidiendo que gran parte de las aplicaciones Java actuales puedan ser ejecutadas en este modelo computacional.

Consciente de ello, la comunidad RTSJ, lo ha entendido como una carencia específica en lo que es el área de patrones de programación y por tanto gran parte de su trabajo ha ido en esa línea, en la de adaptar los diferentes patrones existentes en el estado del arte a dicho modelo. La característica común a todos ellos es que intentan amoldarse al modelo de regiones de RTSJ mediante, o bien paradigmas de programación orientados a objetos que toman en consideración a las regiones, o bien haciendo uso de patrones de programación específicos como por ejemplo el de copia.

Sin embargo, existe una cierta duda razonable sobre si serán suficientes o no pues hay aplicaciones donde su utilización no es tan sencilla. Un reto especial al que se debe enfrentarse este modelo de referencias es el ofrecido por las grandes aplicaciones legado como pueden ser los middleware de distribución RMI y CORBA. En los principales trabajos relativos a la implementación de RTRMI y RTCORBA se ha visto que su implementación con el actual modelo de regiones no resulta siempre sencilla. Y por tanto, algunos investigadores del área incluso han llegado a proponer extensiones al propio RTSJ [18] [31] mientras que otros, más conservadores, descansan en nuevos y complejos patrones de programación [140] que son compatibles con el actual RTSJ.

La extensión RTSJ++ propuesta, el `ExtendedPortal`, asume que no siempre es posible realizar dicho tipo de violaciones de forma sencilla y propone una fórmula que permite realizar violaciones de la regla de asignación de RTSJ de forma segura. Esta extensión se entiende como un complemento al modelo actual de referencias, útil en casos poco comunes donde resulta interesante realizar violaciones de la regla de asignación.

En el caso de DREQUIEMI, esta extensión ha sido utilizada exitosamente para almacenar referencias a objetos remotos que residen en `ScopedMemory` dentro de `ImmortalMemory`. Una operación de bastante utilidad en el middleware de distribución RTSJ que no resulta sencilla de conseguir con la especificación actual.

Para presentar la extensión, se esbozan un serie de secciones que nos aproximan a ella en varias direcciones. En primer lugar -sección 5.2.1- se profundiza más en

detalle en las diferentes propuestas existentes en el estado del arte para realizar violaciones de la regla de asignación para después -sección 5.2.2- exponer los principales problemas que presenta el mecanismo actual basado en portales, para continuar con otra -sección 5.2.3- donde se discuten las modificaciones que son requeridas para su implementación. Por último, -sección 5.2.4- aparecen las conclusiones y líneas futuras.

5.2.1. Punto de partida

De alguna manera algunos investigadores defienden el modelo de regiones de RTSJ, proponiendo una serie de patrones que permiten ampliar el abanico de aplicaciones que se pueden beneficiar de dicho modelo. Así algunos investigadores como Corsaro [46] han propuesto la revisión de los patrones más utilizados dentro de la ingeniería del software proponiendo nuevas implementaciones acordes con el modelo de regiones de RTSJ. Otros autores como Benowitz [22] y el propio Corsaro [46] también proponen soluciones basadas en mecanismos de copia que permiten realizar las copias de objetos de una forma más o menos automática entre regiones. Y por último, otros investigadores como Pizlo [140] proponen el uso de hilos auxiliares, denominados *wedge threads*, que permiten controlar la vida de las regiones.

Pero sin embargo en el modelo de referencias de RTSJ resulta necesario incorporar mecanismos que nos permitan violar la regla de asignación. Uno de los primeros en darse cuenta de ello ha sido el propio RTSJ [4], el cual ha propuesto una extensión -*portals*- que permite acceder a un objeto almacenado en una `ScopedMemory` mediante métodos especiales. Pero sin embargo, el portal, tal y como veremos en la siguiente sección, es difícil de utilizar.

Esto ha hecho que surja algún tipo de extensión alternativa, como por ejemplo los *pinning scopes* de Timesys [172] que flexibiliza la navegación dentro de la jerarquía de regiones evitando tener que recurrir al encadenamiento de portales a la hora de acceder a los objetos almacenados dentro de una región. Pero sin embargo, el mecanismo sigue manteniendo parte de la complejidad intrínseca de los portales pues tan sólo existe uno por región.

Otro trabajo relacionado es el realizado por Borg [31] sobre el modelo de las *weakreferences* de Java. En este trabajo se propone un nuevo tipo de referencia que permite realizar una navegación de alto nivel dentro de la jerarquía de regiones de RTSJ, incluyendo mecanismos de alto nivel para manipular el *scopestack*. Pero sin embargo, ciertas opciones de implementación como la imposibilidad de acceder a la referencia del objeto referenciado, le restan valor frente a los portales.

En esta línea hay que destacar también el trabajo realizado por Higuera-Toledano [76] en su revisión del modelo de referencias de RTSJ. En su propuesta, se unifican la primitivas de creación del *scopestack* y de su modificación en un único método, `enter`, que permite realizar una navegación sencilla dentro la jerarquía de regiones, evitando que se produzcan fallos en la regla del padre único.

Conceptualmente, el `ExtendedPortal` guarda cierta relación con cada uno de los trabajos anteriores. Al igual que en las técnicas de copia, la idea subyacente consiste en evitar los inconvenientes impuestos por las reglas de asignación y del padre único. Y al igual que en el caso de Borg y los *pinning scopes*, se pretende violar la regla

de asignación. Por último, al igual que Higuera-Toledano se utilizan primitivas de navegación de medio nivel altamente alineadas con el modelo RTSJ.

Pero sin embargo, el `ExtendedPortal` también realiza ciertos aportes en las aproximaciones ya existentes. Así, la extensión propuesta puede ser entendida como una generalización de los portales donde la asignación ya no aparece ligada a lo que es la estructura de la región. Y a diferencia de la aproximación de Borg, con un `ExtendedPortal` se puede obtener el valor almacenado en la referencia, siendo también posible utilizar una semántica de tipo *strong*. Y por último, a diferencia de Higuera-Toledano, el modelo de regiones propuesto es compatible con el actual modelo de referencias de RTSJ.

5.2.2. Limitaciones impuesta por el portal

En esta sección se verá cuáles son los principales problemas que implica la utilización de los portales del actual RTSJ. Para realizar los razonamientos de forma sencilla se parte de un escenario fijo -figura 5.6- con dos hilos que comparten una misma estructura de memoria. El hilo de la izquierda maneja una estructura de regiones con un bloque de memoria immortal -`I`- sobre el que aparece una región anidada -`sa`-, mientras que el de la derecha maneja un doble nivel de anidamiento de regiones -`sb` y `sc`. Las regiones `sb` y `sa` han sido creadas en memoria immortal mientras que `sc` se encuentra en `sb`. El dibujo también nos muestra el concepto de *portal*, una referencia existente en cada una de las regiones que puede ser utilizada para apuntar a un objeto en ella contenida. Y por último, también muestra aquellos tipos de referencias que son permitidas por RTSJ y aquellas que no lo son, viéndose el cómo se sigue una disciplina tipo *cactus stack* que impide el establecimiento de referencias no seguras (de `i` a `a`, `b` o `c`) y que permite aquellas (de `a`, `b` o `c` a `i`) que de forma natural son seguras.

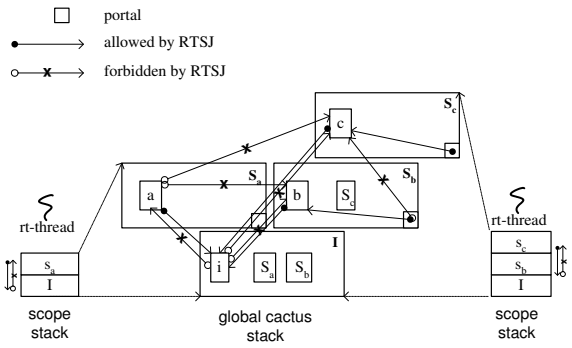


Figura 5.6: Aplicación ejemplo. Referencias prohibidas y permitidas en RTSJ.

Una vez descrito el escenario, es el momento de ejemplificar cuáles son los problemas que trae consigo el uso del portal, empezando por el problema del acceso. Supongamos que el hilo de la derecha quiere acceder a `c`. El primer paso que debe dar es el de crear un nuevo *scopestack*, haciendo uso de `executeInArea`. Después, tendrá que ir reconstruyendo el *scopestack* que le permitirá leer la referencia a `c`.

Para ello, primero tendría que introducir la región `sb` y después la `sc` haciendo uso dos veces del método `enter()`. Y para poder acceder a la referencia al objeto `sc` se tendría que utilizar el portal `sb`, haciendo uso de la siguiente expresión: `Object aux=sb.getPortal()`. Y por último, ya se podría acceder al objeto `c` haciendo uso del método `Object c= sc.getPortal()`. Lo que, en general, implica que el acceso a una referencia a un objeto cuyo nivel de anidamiento es n tiene un grado de complejidad $\Theta(n)$; requiriéndose `n` invocaciones al método `enter`, otras tantas a `getPortal` y una a `executeInArea`.

Otro problema del portal, como ya se ha mencionado y como la figura 5.7 muestra, es que tan sólo existe uno por región, lo que nos obliga a mantener tablas internas en cada una de las regiones para diferenciar cuál de los objetos contenidos es el que se pretende acceder. Y esto viene con el inconveniente adicional de que se añade complejidad adicional en el programador que se ha de encargar de su manipulación, proveyendo los mecanismos pertinentes para la gestión de la tabla.

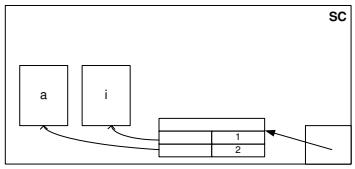


Figura 5.7: Utilizando una tabla para acceder a múltiples objetos

Y por último, estaría el problema de la semántica. Los portales se destruyen al mismo tiempo que la región que los define, lo que no nos permite afirmar que tengan algún tipo de semántica en concreto del estilo *strong* de las referencias tradicionales Java o *weak* de las débiles definidas en el paquete `java.lang.ref`. Para solventar esta limitación, una solución es la de utilizar hilos auxiliares (*wedge threads*), tal y como se muestra en la figura 5.8, que controlen la vida de las regiones permitiéndonos evitar la destrucción prematura de objetos. Pero esto trae consigo dos problemas: (1) resulta necesario el empleo de tablas auxiliares para acceder a dicho hilo auxiliar y (2) se malgastan recursos computacionales pues ese hilo consume memoria adicional y requiere de espacio adicional en la tablas internas de la máquina virtual y del sistema operativo.

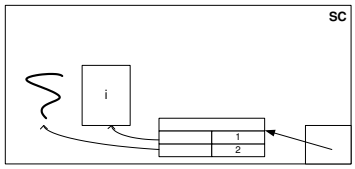


Figura 5.8: Utilizando una entidad concurrente auxiliar para mantener la vida de la región

Frente a todo este conjunto de problemas, el `ExtendedPortal`, lo que propone es ocultar tanto la complejidad en las operaciones de acceso como en las del mantenimiento de referencias prohibidas, definiendo para ello una interfaz que facilita

el almacenamiento de referencias prohibidas a objetos, permitiendo su acceso y la manipulación de una semántica. La idea básica en el modelo será la de que todas las tablas e hilos auxiliares necesarios en el caso de trabajar con portales se ocultarán detrás de la interfaz común `ExtendedPortal`.

5.2.3. Modificaciones requeridas

Para poder aprovechar las ventajas anteriormente descritas son necesarias dos tipos de modificaciones. Una en el sistema de interfaces, que ha de dar acomodo a este nuevo tipo de referencias y otras de más bajo nivel relacionadas con los detalles de implementación. En el caso que nos acompaña se comenzará proponiendo una interfaz Java para después caracterizar su comportamiento interno.

Interfaces

Tal y como recoge la figura 5.9, la extensión consta de una única clase. Esta clase tiene métodos que permiten parametrizar el comportamiento de la referencia así como otros -`getPortal` y `setPortal`- que permiten acceder y modificar la referencia almacenada. Además, existe un método especial -`enter`- que facilita la navegación dentro de la jerarquía de regiones de RTSJ. Y por último, hay dos métodos -`isStrong` y `setStrong`- que permiten gestionar el tipo de referencia con el que se trabaja: *weak* o *strong*.

```
package es.uc3m.it.drequiem.rtrmi;
public class ExtendedPortal{
 public ExtendedPortal(long depth, Object initial);
 public Object getPortal();
 public void setPortal(Object c);
 public void enter(Runnable r);
 public void setStrong(boolean b);
 public boolean isStrong();
}
```

Figura 5.9: Interfaz para el `ExtendedPortal`

A parte del constructor por defecto existe otro que permite configurar dos detalles de bajo nivel: la profundidad máxima de la estructura *scopestack* utilizada dentro del `ExtendedPortal` y un valor inicial para la referencia almacenada internamente. Este nivel máximo de profundidad permite hacer una reserva de bajo nivel de memoria que puede ser utilizada para evitar que las interfaces del resto de los métodos consuman memoria de forma dinámica.

El método `setPortal`, al igual que el de los portales de RTSJ, permite almacenar una referencia a un objeto dentro del `ExtendedPortal`. Pero a diferencia del tradicional, cualquier tipo de referencia puede ser almacenada en él pues la máquina virtual garantiza que esta referencia no es destruida antes de que desaparezca el objeto.

El método `getPortal`, al igual que en el caso de los portales, permite recuperar una referencia almacenada en una referencia extendida. Este proceso puede no ser siempre exitoso y así, cuando la región no está presente en el *scopestack* del hilo invocante, se prohíbe la lectura generando una excepción.

Para evitar este problema se puede utilizar el método `enter`. De forma similar al `enter` del *memoryarea* de RTSJ, este método modifica el *scopestack* del hilo invocado. Pero en este caso, en vez de introducir una única región, se introduce el *scopestack* de la referencia extendida por completo, lo que permite realizar un cierto acceso garantizado a la referencia.

Por último, existe un par de métodos `-setStrong` y `isStrong` - que permiten establecer y modificar dinámicamente la relación mantenida con el algoritmo de recolección de basura, permitiendo que se puedan establecer referencias de tipo *strong* o *weak*. La diferencia entre ambas es que mientras la de tipo *strong* evita que el algoritmo de gestión automática de memoria destruya el objeto referenciado, la de tipo *weak* no es capaz de ello. Por defecto, cuando es creada, una referencia es de tipo *strong*.

Detalles de bajo nivel: implementación

Para implementar [2] el modelo anteriormente descrito son necesarios ciertos mecanismos que nos permitan interactuar con lo que son los algoritmos de gestión automática de memoria ya definidos por RTSJ. Más en detalle, son necesarios dos tipos de mecanismos. Aquellos que nos permiten gestionar la vida de un objeto evitando que desaparezca y aquellos que nos permiten detectar que un objeto ha sido destruido. Los primeros son utilizados por las referencias de tipo *strong* mientras los segundos lo son por las *weak*. Además, la implementación de este mecanismo presenta el doble problema de que tiene que trabajar con un sistema de gestión automática de memoria dual, capaz de utilizar tanto el modelo de regiones de RTSJ como el del recolector de basura tradicional.

Empecemos describiendo la estructura de datos interna que hay en cada `ExtendedPortal`, para después ver el soporte específico requerido por cada operación.

Estructura de datos

Internamente, se crean dos tipos de estructuras de computación. La primera de ellas es una referencia que sirve para evitar que el objeto sea destruido y la segunda es un array de tamaño `length` utilizado para evitar que el algoritmo de gestión automática de memoria de las regiones destruya un objeto antes de tiempo.

Soporte para `set`

Así pues, cuando se almacena una referencia, que bien puede ser creada en `HeapMemory`, `ImmortalMemory` o en `ScopedMemory`, lo primero que se hace es almacenar dicha referencia a objeto en la referencia interna del `ExtendedPortal`. Tras ello se intenta evitar que el algoritmo de gestión automática de memoria la destruya

[2] En la actualidad, el `ExtendedPortal` ha sido implementado en la máquina virtual jTime dentro del contexto definido por DREQUIEMI, a fin de permitir el acceso a objetos remotos almacenados en regiones arbitrarias. Están soportadas aquellas características más básicas, provistas por los métodos `enter`, `getPortal` y `setPortal`, careciéndose de soporte específico para la semántica *weak* por no disponer de acceso al código fuente de jTime.

prematuramente. En el caso de que tengamos `HeapMemory` o `ImmortalMemory` esto es sencillo de conseguir pues al almacenarla en un atributo del objeto se evita su destrucción. Pero en el caso de que esté en `ScopedMemory` resulta necesario realizar pasos adicionales.

En este caso, tal y como se muestra en la figura 5.10, el evitar que la referencia sea destruida requiere que se guarde el contexto de creación del objeto referenciado y que se evite que sea prematuramente destruido. En el caso implementado, el contexto de creación se extrae directamente del *scopestack* del hilo invocante y se almacena -paso `(1)`- en el array interno del `ExtendedPortal`. Tras ello -en el paso `(2)`- se procede a evitar la destrucción del objeto referenciado incrementando para ello el contador interno de cada una de las `ScopedMemory` contenidas en el array interno del `ExtendedPortal`.

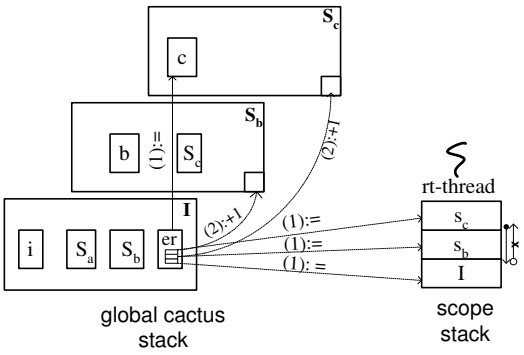


Figura 5.10: Almacenando una referencia en un `ExtendedPortal`

Soporte para `get`

El proceso de lectura, asociado al método `get`, presenta dos casos bien diferenciados. El caso en el que el objeto referenciado reside en `HeapMemory` o `ImmortalMemory` y en el cual reside en `ScopedMemory`. En el primer caso simplemente se devuelve la referencia, comprobándose adicionalmente, en el caso de que el hilo que accede sea de tipo `NoHeapRealtimeThread`, que el objeto cuya referencia es leída no reside en `HeapMemory`. Pero en el caso de caso de que resida en una `ScopedMemory` son necesarias comprobaciones adicionales.

Tal y como muestra la figura 5.11, resulta necesario saber si el hilo que pretende acceder al valor lo puede hacer o no. Para ello, se comprueba si contiene en su *scopestack* la región donde ha sido creado el objeto referenciado `-c` en el ejemplo de la figura-. Para ello se explora cada una de las regiones del *scopestack* del hilo que realiza la invocación a `get`, viendo si la contiene. En caso positivo, se le permite acceder al objeto y en caso contrario se lanza una excepción de tipo `IllegalAccessError`.

Soporte para `enter`

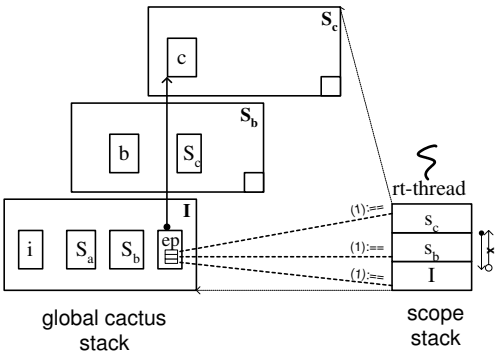


Figura 5.11: Acceso a una referencia almacenada en un `ExtendedPortal`

El que el acceso no esté siempre permitido nos lleva a proponer un tercer método `enter(Runnable r)` que posibilite que el programador pueda acceder, utilizando `getPortal`, a la referencia almacenada dentro del `ExtendedPortal`.

Internamente el pseudocódigo ejecutado por el método `enter` es el que sigue:

1. En el hilo invocante se crea un nuevo *scopestack* igual al de la referencia extendida (`ep`), incrementándose los contadores internos de cada región.

2. Tras ello, se invoca el método `run()` de `r`.

3. Por último, se restaura el *scopestack* previo a la invocación, procediendo a decrementar los contadores de todas las instancias de tipo `ScopedMemory` contenidas en la referencia extendida.

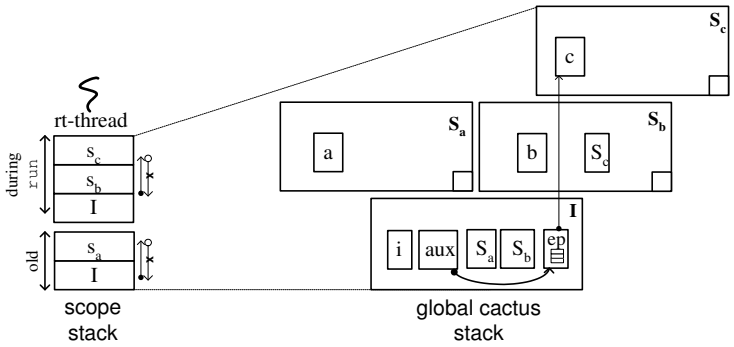
La figura 5.12 muestra cómo el hilo de la derecha puede acceder al objeto `c` para llamar a uno de sus métodos sin que se produzca ningún tipo de violación de la regla de asignación. Para ello es necesario, al igual que sucede con el mecanismo `enter` de un `MemoryArea`, el empleo de un objeto de tipo *runnable* que resida en el mismo tipo de región que la referencia extendida para que así se pueda hacer la siguiente asignación: `auxep=ep`. Tras haber hecho esta asignación, la ejecución por parte del hilo de `er.enter(r)`, le permitirá acceder de forma segura al valor del objeto `c` garantizando que no se producen violaciones de la regla de asignación.

Soporte para `setStrong` e `isStrong`

Ya tan sólo nos falta por ver un último caso, las implicaciones que tiene la conversión de una referencia de tipo *strong* a *weak* y viceversa.

En la conversión de referencia de *strong* a *weak* se dan dos pasos. El primero es eliminar el veto que impide la destrucción del objeto referenciado. Y el segundo es notificar al algoritmo de gestión automática de memoria que se han producido cambios en la semántica de una referencia.

Para eliminar el veto en la `HeapMemory` tan sólo es necesario eliminar la referencia que está almacenada en el atributo. Y en el caso de que estemos ante una `Scoped-`




```
public class Aux implements Runnable{
 private ExtendedPortal auxep=ep;
 public void run(){
       Object c= auxep.getPortal();//a.r ok
       c.hashCode();
 }
}
```

Figura 5.12: Forzando la regla de asignación con el método `enter`

`Memory`, es necesario además realizar un decremento de todas las regiones contenidas en el *scopestack*, tal y como muestra la figura 5.13.

El caso de la transformación de una referencia *weak* a *strong*, el proceso es el complementario al descrito con anterioridad. En primer lugar se ha de notificar a la máquina virtual de que el estado de la referencia ha mudado a *strong* para después proceder al incremento de los contadores de las regiones contenidas en el `Extended-Portal`.

5.2.4. Conclusiones y líneas futuras

En esta sección se ha propuesto un modelo de referencias extendido para RTSJ. Este modelo, denominado `ExtendedPortal`, permite realizar violaciones de la regla

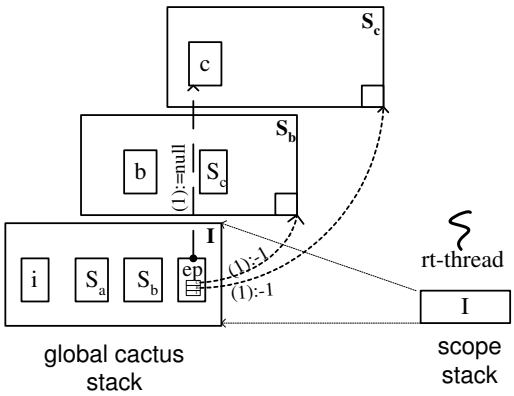


Figura 5.13: Transformando un `ExtendedPortal` *strong* en *weak*.

de asignación de forma segura, soportando dos semánticas: *strong* y *weak*.

Como resultado más destacable se puede resaltar que la extensión propuesta pone de manifiesto un importante hecho: que resulta posible tener un tipo de referencia general, capaz de interactuar con el sistema de referencias del actual RTSJ, donde al igual que en Java se subyugue el sistema de gestión automática de memoria al modelo de referencias, manteniendo ciertas garantías de seguridad en su acceso.

Hasta el momento se han identificado dos líneas de trabajo a explorar: una relacionada con la eficiencia y otra con el mecanismo empleado para realizar el acceso al portal. En el estado del arte aparecen ciertas técnicas capaces de reducir la complejidad necesaria para validar la regla de asignación de $\Theta(n)$ a $\Theta(1)$. En esa misma línea, sería interesante investigar cómo aplicar este tipo de técnicas dentro de la solución de tal manera que la complejidad de implementación también pasase de $\Theta(n)$ a $\Theta(1)$. La otra vía a explorar consistiría en la búsqueda de modelos alternativos al propuesto donde no sea necesario el empleo de patrones de programación que hagan uso del método `enter` a la hora de acceder a la referencia contenida en el `ExtendedPortal`, intentando acercar más el modelo del `ExtendedPortal` al tradicional de Java.

5.3. Modelo unificado para los hilos de tiempo real

Una de las grandes ventajas que presenta RTSJ frente a otros lenguajes de tiempo real es que la separación entre las características de tiempo real y lo que son las de propósito general es menor que la existente en otras aproximaciones. Especificaciones Java de tiempo real alternativas como RTCORE diferencian mucho más estos dos entornos de ejecución proponiendo el empleo de una jerarquía de clases diferente para las aplicaciones del CORE y otra para las de propósito general. Y en otros entornos, como por ejemplo el provisto por muchos de los sistemas operativos de tiempo real, esta separación es aún mucho mayor utilizándose algunas veces módulos especiales que son cargados en el núcleo del sistema operativo. Pero aún así, en RTSJ aún existe una cierta diferenciación entre las aplicaciones de tiempo real estricto y las existentes para sistemas de tiempo real más laxos, materializada en la existencia de un cierto dualismo computacional.

De facto, en RTSJ, existen dos modelos de programación, uno enfocado a la alta predictibilidad denominado con el término *noheap* donde no se puede utilizar el montículo Java y otro en principio menos predecible denominado *heap* donde resulta posible utilizar el recolector de basura. A nivel de entidades concurrentes esto se traduce en la existencia de dos entidades concurrentes: `RealtimeThread` y `NoHeapRealtimeThread`. La primera es capaz de utilizar el montículo de Java para crear y acceder a objetos pero a cambio padece las latencias del recolector de basura. La segunda, por el contrario, consigue librarse del recolector a cambio de no poder acceder al montículo Java. Y como consecuencia de ello, una aplicación RTSJ compleja tenderá a utilizar ambos tipos de hilos, no pudiendo restringirse a un tipo en particular.

Además, para que estas dos estructuras puedan acceder a los datos, RTSJ define un nuevo tipo de mecanismo de sincronización, no considerado previamente en Java tradicional: la cola de mensajes. Y esto, desde el punto de vista del programador,

también acarrea nuevas complicaciones, pues el programador ha de familiarizarse con nuevos mecanismos como son las colas de mensajes, no siendo posible la utilización del `synchronized` de Java de forma generalizada.

Intentando retornar al modelo tradicional de Java donde tan sólo existe una única entidad concurrente de tiempo real, la extensión `RealtimeThread++` lo que propone es un hilo genérico, capaz de mudar su comportamiento de forma dinámica entre lo que es un `RealtimeThread` y un `NoHeapRealtimeThread`. De tal manera que este nuevo tipo de hilo pueda decidir de forma dinámica, durante su ejecución, el tipo de relación que desea mantener con el recolector de basura.

La utilidad de esta extensión en el contexto marcado por DREQUIEMI es la de facilitar y optimizar su implementación. En el lado del servidor es necesario que de forma dinámica el hilo que está procesando una petición entrante decida si ésta ha de ser procesada por un hilo de tipo `NoHeapRealtimeThread` o uno de tipo `RealtimeThread`. La extensión propuesta facilita que esta decisión sea tomada de forma dinámica durante el procesado de la invocación remota entrante y evita tener que recurrir al empleo de dos hebras, una de cada tipo, optimizando por tanto la implementación en el lado del servidor.

El resto de la secciones se dedican a presentarlo desde diferentes ángulos. Se comienza -en la sección 5.3.1- viendo las diferentes posturas existentes en el estado del arte para abordar dicho dualismo computacional. Después -en la sección 5.3.2- veremos mediante un ejemplo sencillo cuáles son las ventajas que presenta la unificación del modelo de computación así como -en la sección 5.3.3- diferentes cuestiones relacionadas con su implementación como son los cambios que son necesarios tanto en las interfaces de programador como a la hora de realizar su implementación. Y ya para finalizar -en la sección 5.3.4- están las conclusiones y líneas futuras.

5.3.1. Punto de partida

El dualismo existente en el modelo de concurrencia de RTSJ es una característica que la comunidad investigadora ha ignorado, limitándose a asumirla, sin proponer ningún tipo de mejora encaminada a su eliminación. Wellings cuando revisa el modelo de eventos de RTSJ [183] es consciente de este dualismo pero no propone ningún tipo de extensión, aceptando el modelo de sincronización basado en colas. Se detecta el problema de las inversiones de prioridad que pueden sufrir los hilos de tipo `NoHeapRealtimeThread` cuando se sincronizan con los de tipo `RealtimeThread`, justificándose así el empleo de colas de mensajes y una estructura de *threadpool* dual. En plena sintonía con este trabajo, pero esta vez dentro del proyecto mackinac [24] y también cuando se habla de la implementación del modelo de asincronía, se propone la utilización de dos *threadpools*, uno *heap* y otro *noheap*, diferenciados y dimensionables mediante interfaces propietarias.

Otros autores, sobre todo los que se dedican a la construcción de soluciones para sistemas de alta integridad, tampoco abordan el dualismo, sino que más bien lo evitan aprovechando las ventajas brindadas por los escenarios que manejan. Así, tanto ravenscar-Java [83] como las propuestas de alta integridad para sistemas distribuidos [170], obligan a una utilización exclusiva del `NoHeapRealtimeThread`, evitando

también el empleo de colas de mensajes.

Todo esto, unido al hecho de que en la futura revisión de la especificación no se haya propuesto ningún tipo de mejora en este aspecto, hace que el problema del dualismo sea algo novedoso. Aunque en la siguiente versión de la especificación (la 1.1 [92]) aparecen ciertas propuestas para mejorar el sistema de recolección de basura y de referencias, no hay ningún tipo de mejora encaminada a romper con el dualismo computacional actual de RTSJ.

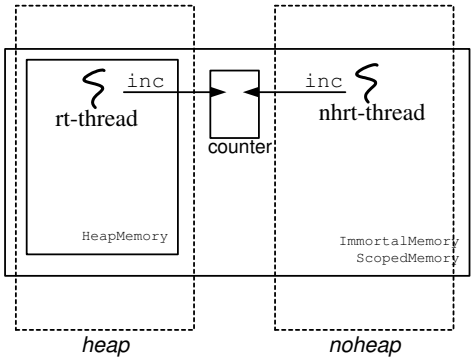
En esta línea los `RealtimeThreads++` se pueden entender como un intento de reunificar los dos tipos de hilo de RTSJ bajo una única forma generalizada de entidad concurrente. De forma contraria a RTSJ, esta nueva forma rompe con la dualidad existente, proponiendo un hilo que es capaz de mudar durante su ejecución las relaciones que mantiene con el recolector de basura. Como efecto de este cambio, tal y como se verá en el resto de la sección, se consiguen ventajas, no presentes en el actual RTSJ, que derivan de poder utilizar el `synchronized` de Java de forma segura para sincronizar tareas `RealtimeThread` y `NoHeapRealtimeThread`, evitando la propagación de la inversión de prioridad provocada por el recolector de basura desde el entorno *heap* al *noheap*.

5.3.2. Sincronización con hilos de tiempo real tradicionales y generalizados

Antes de comenzar con la definición de la extensión propiamente dicha veremos con un sencillo ejemplo cuál es el aporte que el `RealtimeThread++` hace al modelo actual de RTSJ. Para ello comenzaremos viendo un ejemplo de sincronización no válida en RTSJ, para después ver cómo el propio RTSJ puede solventar el problema mediante la incorporación de colas de mensajes y una nueva entidad concurrente. Esto nos dará pie a realizar una crítica que nos servirá de motivación para la extensión `RealtimeThread++`.

Empecemos comprendiendo cuál es el problema que entraña el empleo de la palabra reservada `synchronized` entre un `RealtimeThread` y un `NoHeapRealtimeThread` de RTSJ. Para ello utilizaremos el ejemplo de la figura 5.14. En esta figura se muestran dos hilos intentando invocar concurrentemente al método `incr` del objeto compartido de tipo `Counter`. Internamente, el método, tal y como su nombre sugiere, se dedica a incrementar el valor del atributo `count`, acción tras la cual se procede a liberar el cerrojo que había sido previamente cerrado. La peculiaridad que presenta el ejemplo es que mientras un hilo está ejecutando en entorno *heap*, el otro lo está haciendo en *noheap*.

Pues bien, el problema que presenta este código es el siguiente. Imaginemos que mientras el hilo `rt` está modificando la variable `count` dentro del bloque sincronizado mientras que el hilo `nhrt` se encuentra esperando a que `rt` salga del bloque sincronizado. Cuando en este escenario salta el recolector de basura se produce un problema de inversión de prioridad. De esta manera, si por ejemplo el algoritmo de recolección utilizado es del tipo más dañino, el *stop-the-world*, la tarea `nhrt` que se encontraba bloqueada a espera de la tarea `rt`, sufrirá por la no progresión de `rt` el proceso de recolección de basura. Es decir, que no podría ejecutarse hasta que hubiese finalizado




```
public class Counter {
private long count=0;
private Object lock=new Object();
 public void inc(){
    synchronized(lock){ //GC inv. propagation
    count++;
    }
 }
}
```

Figura 5.14: Propagación de la inversión de prioridad del recolector basura en RTSJ

por completo la ejecución del algoritmo de recolección de basura.

Este fenómeno, que se denomina como propagación de inversión de prioridad debida al recolector de basura entre el entorno *heap* y el *noheap*, no es propiedad exclusiva del `synchronized`. En un principio cualquier mecanismo que permita que un hilo `NoHeapRealtimeThread` se bloquee a la espera de las ordenes de un `RealtimeThread` presenta el mismo tipo de problema.

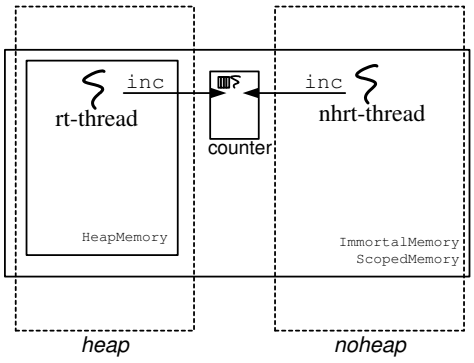
RTSJ, consciente de este problema, propone un mecanismo de sincronización de bajo nivel: la cola de mensajes. Ésta permite realizar una sincronización no bloqueante entre dos hilos.

Tomando como base el ejemplo anteriormente desarrollado, la figura 5.15 muestra el código adicional que permite, utilizando una cola de mensajes auxiliar, proteger la variable `counter`. La solución consiste en utilizar un nuevo tipo de hilo, de tipo `NoHeapRealtimeThread`, encargado de manejar el atributo `count`. La comunicación entre los hilos se hace mediante una cola de mensajes `-wfq-` en la que se deposita un objeto cada vez que se quiere incrementar el contador y que es retirado cada vez que se incrementa el contador. Este proceso de depositar y de retirar es atómico[3], garantizándose que el hilo `nhrt` no sufre las inversiones del recolector de basura pues la cola de mensajes sirve como elemento desincronizador.

Pero desde el punto de vista lógico, la principal desventaja del modelo es que se pierden las ventajas proporcionadas por el sincronismo. En este caso implica que el hilo que deposita el mensaje no tiene garantías sobre cuándo es realmente modificado el valor del atributo `count`.

Un detalle a destacar es que utilizando la cola de mensajes como mecanismo de

[3]Lo garantiza la implementación de la cola de mensajes.



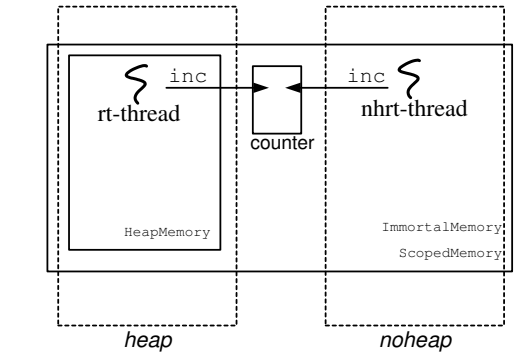

```
public class Counter{
private long count=0;
private Object lock=new Object();
private WaitFreeReadQueue wfq=new
WaitFreeReadQueue(5,false);
public Counter(){
    th.start();
}
private NoHeapRealtimeThread th=new
NoHeapRealtimeThread(){
  public void run(){
    do{
        wfq.waitForData();
        wfq.read();
        count++;
    }while(true);
}};
public void inc(){
    wfq.write(lock);//OK
    }
}
}
```

Figura 5.15: Utilizando colas de mensajes en RTSJ para evitar la propagación de la inversión de prioridad del recolector

sincronización no resulta posible sincronizar los dos hilos sin que llegue a interferir la recolección de basura. Si fuese posible, significaría que el hilo `nhrt` debería de esperar a que el hilo `rt` finalizase de utilizar el contador, lo que implicaría que el hilo `nhrt` sufriría la inversión del recolector de basura. Y en este caso estaríamos otra vez con problemas de propagación de inversiones de prioridad.

Frente a esta situación, lo que la extensión `RealtimeThread++` ofrece es la posibilidad de mover el hilo entre los entornos *heap* y *noheap*, evitándose el problema de la propagación de la inversión de prioridad del recolector de basura. Tal y como se muestra en la figura 5.16, la posibilidad de moverse al entorno *noheap*, ofertada por el nuevo método `enterNoheap()` del `RealtimeThread++`, permite utilizar esta capacidad para mover el entorno de ejecución del hilo al entorno *noheap* donde no hay recolector de basura. Y una vez eliminada la dependencia con el recolector de basura, ya resulta posible utilizar el `synchronized` de Java para realizar la sincronización de tiempo real de forma predecible.

Pero tal y como se pone de manifiesto en el ejemplo, resulta necesario realizar modificaciones en el conjunto de interfaces de RTSJ para acomodar esta nueva fun-




```
public class Counter {
private long count=0;
private private Object lock=new Object();
private Runnable r=new Runnable(){
 public void run(){
    synchronized(lock){ //GC OK
    count++;
    }
 };
 public void inc(){
 RealtimeThread.enterNoheap(r); //Ok
 }
}
```

Figura 5.16: Utilizando el `synchronized` en la aproximación `RealtimeThread++`

cionalidad. Y además, a nivel máquina virtual son también necesarios cambios.

5.3.3. Modificaciones requeridas

Al igual que se hizo en el resto de extensiones, en primer lugar se propondrá una extensión, en este caso a la interfaz actual del `javax.realtime.RealtimeThread`, para después de esto explorar los cambios a dar en el modelo de máquina virtual actual a la hora de soportarla.

Interfaces

La figura 5.17 nos muestra un conjunto de nuevos métodos que se han definido para permitir modificar y consultar cuál es la relación que se quiere mantener con el recolector de basura. La extensión `RealtimeThread++` los añade como nuevos métodos de la actual clase `RealtimeThread`.

Básicamente hay dos métodos: el `enterHeap` y el `enterNoheap`, que permiten el movimiento entre lo que son los entornos de ejecución. Y además existe otro adicional, `isRunningInHeap`, que permite consultar cuál es el estado de ejecución actual del hilo. Por defecto, cuando son creados, un hilo `RealtimeThread` se inicializará en

el entorno *heap*, mientras que uno de tipo `NoHeapRealtimeThread` lo hará en uno *noheap*.

```
package javax.realtime;
public class RealtimeThread extends Thread{
 public static void enterHeap(Runnable r);
 public static void enterNoHeap(Runnable r);
 public static boolean isRunningInHeap();
              .      .      .
}
```

Figura 5.17: Métodos introducidos en la clase `RealtimeThread` por la extensión `RealtimeThread++`

El pseudocódigo que se ejecutará durante el método `enterNoheap(Runnable r)` es el siguiente:

1. Cambiar a entorno de ejecución *noheap*.
2. Ejecutar el método `r.run()`.
3. Restaurar el entorno de ejecución previo.

Tanto el nombre del método como su comportamiento es muy parecido al de los métodos que permiten manejar el *scopestack*. Al igual que en el caso del método `MemoryArea.enter(Runnable r)` se modifica el estado interno relacionado con la gestión de memoria y también, al igual que en este caso, se ejecuta el método `r.run()`, restaurándose al final de su ejecución el estado previo.

La principal diferencia ente ambos radica en la parte de la gestión automática de memoria que es modificada. Así, si mientras el método `enter` de un *memoryarea* modifica la región donde son creados los objetos, el `enterHeap` modifica la relación mantenida con el algoritmo de recolección de basura, permitiendo o prohibiendo la utilización de la `HeapMemory`.

De forma muy parecida al método `enter` del `MemoryArea`, el pseudocódigo que se ejecutará durante el método `enterNoheap(Runnable r)` es el siguiente:

1. Cambiar a entorno de ejecución *heap*.
2. Ejecutar el método `r.run()`.
3. Restaurar el entorno de ejecución previo.

En ninguno de los dos casos se modifica el *scopestack* del hilo que los invoca. Esto es, los objetos serán creados en la región que se encuentre en la cima de esta estructura y por tanto es responsabilidad del programador que éste se encuentre configurado adecuadamente. Es decir, será el programador el que habrá de evitar que se utilice la `HeapMemory` durante la ejecución del método `enterNoheap(Runnable r)`.

Por último, el método `isRunningInHeap` permite averiguar cuál es el estado actual del hilo que se está ejecutando. En un principio, este nuevo método se ha definido

porque la introducción de métodos que permitan modificar el estado actual de ejecución del hilo de tiempo real impide saber el estado de ejecución de un hilo a partir de su tipo. En RTSJ tradicional se sabe que un hilo `RealtimeThread` se ejecuta en entorno *heap* y además que uno `NoHeapRealtimeThread` lo hace en entorno *noheap*. En RTSJ++ esta regla ya no es válida pues pueden haber sido modificados por el método `enterHeap` o por el `enterNoheap`. Es por ello que es necesario un método complementario al de modificación que nos permita realizar la consulta.

Estos métodos implican interacciones con el mecanismo de gestión automática de memoria, requiriéndose cambios en la implementación de la máquina virtual actual.

Cambios requeridos en la máquina virtual

A la hora de analizar cuáles son los cambios requeridos en una máquina virtual RTSJ para soportar dichos mecanismos se parte de que se dispone de una máquina virtual completa, capaz de utilizar el recolector de basura sin interferir en la ejecución de un `NoHeapRealtimeThread`. Y en consecuencia existen, tal y como sucede en las implementaciones completas de RTSJ -Jamaica o jTime-, además de una serie de barreras de ejecución encargadas de verificar que no se violan las reglas de asignación ni del padre único, otra barrera de lectura, propia del `NoHeapRealtimeThread`, que le impide acceder a objetos almacenados en `HeapMemory`.

El hecho de que se parta de una máquina que sabe ejecutar tanto en el entorno *heap* como en el *noheap* reduce el problema de la implementación a decidir qué se hace cuando se cambia de un entorno de ejecución a otro. Esto es, a una secuencia de operaciones que son ejecutadas cuando se invoca al método `enterHeap` y al `enterNoheap`. En el resto de los casos, la máquina virtual habrá de tratar al hilo o bien bajo las restricciones aplicables al entorno *heap* o bien bajo las aplicables al *noheap*, escenarios en los cuales ya conoce cuáles son las condiciones de ejecución.

Veamos pues los cambios que se deben de realizar para que los hilos puedan cambiar su entorno de ejecución.

Básicamente para garantizar que un hilo se ejecuta correctamente en entorno *noheap* es necesario verificar dos condiciones:

1. Que no se poseen referencias a objetos almacenados en el montículo Java.

2. Que durante su ejecución no se accede a referencias a objetos almacenados en `HeapMemory`.

En RTSJ tradicional, la primera condición se verifica en el instante en el cual el hilo de tiempo real es creado y la segunda dinámicamente durante su ejecución mediante barreras que se ejecutan cada vez que se intenta acceder a referencias a objetos. En RTSJ++ este proceso realizado durante la creación del objeto ha de ser realizado cada vez que el hilo cambie a estado *noheap*. Esto es, al retornar del método `enterHeap` cuando el estado previo era *noheap* y al iniciar el método `enterNoheap` cuando estado previo era *heap*.

Así, en RTSJ++ para modificar el estado de ejecución de *heap* a *noheap* resulta necesario dar los siguiente pasos:

1. Activar la barrera de lectura de referencias a objetos del montículo. Esta barrera se ejecuta con los *bytecodes* que permiten acceder a los atributos de objetos tanto estáticos como de objeto.

2. Verificar que el objeto `r` no ha sido creado en `HeapMemory`. En caso negativo, se lanzará una excepción, impidiendo la ejecución del método `run`.

La segunda condición verifica que las variables locales con las que comienza el método se encuentran en entorno *noheap*, permitiendo manejar una barrera dinámica que impida el acceso a variables globales si éstas residen en `HeapMemory`.

El otro tipo de mecanismo que se necesita es el que nos permite realizar el camino inverso desde el *noheap* al *heap*. En RTSJ++ este proceso ha de ser realizado cada vez que el hilo cambia de estado *noheap* a *heap*. Esto es, al retornar del método `enterNoheap` cuando el estado previo era *heap* y al iniciar el método `enterHeap` cuando el estado previo era *heap*.

Para realizar este camino tan solo es necesario dar un único paso:

1. Desactivar la barrera de lectura.

Nótese que ya no es necesario comprobar que el objeto reside en *heap*, tal y como se hacía en el caso donde el hilo se movía desde el entorno *heap* al *noheap*. Ello es debido a que en este entorno siempre se puede acceder a un objeto almacenado en `HeapMemory`.

Por último, existen múltiples formas de implementar el método `isRunningInHeap` pero quizás la más sencilla sea la de utilizar una variable interna `inheap`. Esta variable interna se inicializará a valor `true`, cuando el hilo creado sea de tipo `RealtimeThread` y se inicializará a `false` cuando sea un `RealtimeThread`. Durante la ejecución de `enterNoHeap` su valor se debería de cambiar a `true` y durante el `enterHeap` a `false`. Finalmente, nótese que esta variable, al ser no compartida no sufre de problemas relacionados con condiciones de carrera.

5.3.4. Conclusiones y líneas futuras

En esta sección se ha propuesto un modelo de hilo generalizado dentro del contexto de las extensiones RTSJ++, mostrando cuáles son algunos de los puntos flacos presentes en el modelo actual de sincronización de Java de tiempo real y presentando una alternativa basada en el empleo de nuevos métodos que permiten modificar la relación establecida con el recolector de basura de forma más dinámica. Y tras ello, se han propuesto extensiones en la jerarquía actual de clases de RTSJ, identificando una serie de cambios en la infraestructura de la máquina virtual que nos permiten darle soporte.

Los resultados nos muestran que la propuesta es interesante para el programador. Las nuevas posibilidades de sincronización que esta aproximación ofrece así como su interfaz, sencilla de utilizar desde el punto del programador RTSJ, son sus puntos más fuertes. Además, el poder utilizar el `synchronized` de Java reduce el coste de aprendizaje extra derivado de las colas de mensajes, permitiéndonos realizar una sincronización síncrona entre los diferentes hilos de tiempo real sin que se produzcan interferencias del recolector de basura.

A cambio de este grado de flexibilidad adicional la aproximación propuesta requiere realizar cambios dentro de la propia máquina virtual que implican al propio recolector de basura. Ello es debido a que es necesario que al modelo de computación de RTSJ se le añadan nuevas barreras en tiempo de ejecución que permitan modificar la relación mantenida con el recolector de basura.

A la hora de mejorar esta aproximación dos son las líneas de trabajo que se han identificado. La primera de ellas consiste en la integración de estos mecanismos con los protocolos de sincronización de RTSJ de la clase `javax.realtime.MonitorControl` de tal manera que permitan definir la política de planificación de cada cerrojo, intentando reducir la necesidad de tener que recurrir de forma explícita a los métodos `enterHeap` y `enterNoheap`. La segunda es la de obtener implementaciones del modelo donde el coste del cambio de contexto realizado sea altamente eficiente.

5.4. Conclusiones generales y líneas de actuación futura

De forma general la gran novedad que introduce RTSJ++, si se compara con RTSJ, es la incorporación de abstracciones que faciliten la utilización de sistemas de gestión automática de memoria avanzados. Las extensiones propuestas en este capítulo así parece hacérnoslo creer pues desde el punto de vista del programador tanto la `AGCMemory` como el `ExtendedPortal` como el `RealtimeThread++` ofrecen mejoras sustanciales. El primero mejorando el soporte de gestión de memoria ofrecido por el modelo de regiones actual, el segundo permitiendo una violación segura de la regla de asignación y el tercero reconciliando el modelo de programación dual de RTSJ bajo la forma común de una única entidad concurrente.

Una de las cuestiones comunes que deberían de ser exploradas más en detalle son aquellos temas relacionados con la eficiencia de las diferentes extensiones desarrolladas. Las ventajas de cada una de las implementaciones han quedado claras, pero el coste computacional extra de las aproximaciones no ha sido completamente esclarecido. Aunque en el caso del `ExtendedPortal` y del `RealtimeThread++` ha quedado claro que los costes puede ser que no tengan un gran impacto en el sistema final, en otras extensiones como por ejemplo la `AGCMemory` el cuantificar este coste es de suma importancia. Por lo que, una vía común de futuras mejoras consistiría en la implementación eficiente de algoritmos para cada una de las tres extensiones.

Por último, una de las posibles vías de mejora de RTSJ++ sería el empleo de protocolos de comunicaciones de tiempo real. En el estado del arte existen múltiples protocolos de comunicación que permiten obtener ciertas garantías sobre el tiempo de respuesta de la red, pero sin embargo, cómo hacer que estos mecanismos puedan ser utilizados directamente desde RTSJ a la hora de construir aplicaciones de tiempo real es aún un aspecto que no sido tratado por la especificación. Y esa será la línea de trabajo hacia la cual se enfocarán las nuevas propuestas para RTSJ++.

Capítulo 6

Evaluación empírica

En casi cualquier middleware de distribución de tiempo real una fase importante es la de validación empírica. En ella se suele determinar no sólo la validez de las técnicas propuestas de forma teórica sino también el rango de aplicabilidad de éstas, pudiendo llegar a identificar bandas de utilización prácticas para cada uno de los algoritmos propuestos. A esto se ha de sumar el hecho de que el conocer estos límites nos permite desarrollar nuevas técnicas capaces de atajar deficiencias no previamente cubiertas bajo la forma de optimizaciones específicas o extensiones. Siguiendo este mismo espíritu, en este capítulo nos disponemos a realizar la evaluación empírica de un prototipo software que se ha desarrollado de DREQUIEMI.

En el caso de DREQUIEMI esta evaluación empírica se torna altamente interesante pues el gran número de capas software empleado en su implementación -sistema operativo, máquina virtual y middleware de distribución- arroja cierta duda razonable no sólo sobre los tiempos mínimos de respuesta que será capaz de ofrecer este tipo de solución RTRMI sino que además crea dudas sobre si el coste en términos de procesador y de memoria de este tipo de sistemas será adecuado para su empleo en sistemas embebidos. Éste es quizás su mayor aporte, ya que la comunidad Java de tiempo real carece de valores orientativos sobre el coste real de lo que son las soluciones RTRMI, poseyendo tan sólo algunos valores orientativos sobre lo que sería el coste de las soluciones RTCORBA proporcionados por el proyecto RTZen descrito en la sección 2.4.8. En esta línea, los resultados obtenidos para DREQUIEMI podrían ser tomados como una primera estimación de lo que sería el nivel 1 de integración propuesto por DRTSJ.

El resto del capítulo presenta tanto el escenario de prueba y las herramientas de medida utilizadas en esta evaluación empírica así como los resultados de ésta. Para ello se comienza relatando cuál es el estado actual del prototipo -sección 6.1-, las configuraciones hardware y software utilizadas -sección 6.2- durante la experimentación práctica así como el conjunto de herramientas utilizadas a la hora de desarrollar y validar el prototipo -sección 6.3. A continuación se procede a mostrar los resultados de las pruebas desarrolladas, comentando los resultados obtenidos. Se comienza con una serie de experimentos orientados a cuantificar -sección 6.4- la inversión de prioridad sufrida por una tarea de tiempo real de mayor prioridad cuando compite con otras de menor prioridad durante el acceso a un objeto remoto compartido.

Después -sección 6.5- se evalúa cuál es la influencia que el empleo de un mecanismo de regiones tiene sobre el coste total de la invocación remota, presentando tanto patrones de comportamiento temporal como la dependencia para con el tamaño de memoria ocupada o la cantidad de memoria disponible en el sistema. Y por último, se procede a analizar lo que es la dependencia existente entre el consumo de memoria -sección 6.6- y de procesador -sección 6.7- para con el flujo de datos intercambiados entre los nodos cliente y servidor. Cierra el capítulo la sección 6.8 donde se sintetizan los logros más interesantes alcanzados en esta evaluación empírica y donde se señalan posibles líneas de trabajo futuro.

6.1. Estado actual del prototipo

El prototipo ha sido construido tratando de optimizar el esfuerzo necesario en su implementación. Por ello, tras haber estudiado y parcialmente probado diferentes piezas software utilizables a la hora de construir Java de tiempo real distribuido, se ha optado por escoger el tándem jTime-RMIOP.

Otras soluciones como por ejemplo el uso de la jerarquía de clases RMI del proyecto GNU -utilizables con jRate, OVM y Jamaica- fueron descartadas por diferentes motivos. En el caso de jRate y OVM el bajo soporte que tienen del estándar RTSJ, inferior al de jTime, nos lo ha desaconsejado. Y en el caso de Jamaica problemas de índole práctica a la hora de realizar aplicaciones de prueba con las clases del GNU classpath también nos desanimaron. Este interés en Jtime y RMIOP se vio acrecentado al comprobar que el código fuente de RMIOP tiene una calidad muy alta, lo que sin duda ha facilitado la comprensión de su funcionamiento interno. La mayor limitación encontrada ha sido, dado que no se disponía del código fuente de jTime, la incapacidad práctica de implementar el soporte para RTSJ++, a excepción de una versión parcial del `ExtendedPortal`.

De forma incremental se han ido incorporando diferentes técnicas en su núcleo. La primera de ellas ha sido la de un gestor de memoria predecible basado en el empleo de las regiones de RTSJ, bajo la forma de un *memorypool* especializado. Tras comprobar y evaluar su funcionamiento, se procedió a implementar el esquema de prioridades globales de DREQUIEMI, similar al de prioridades RTCORBA y que aparece descrito en el capítulo 3 de esta tesis. Tras ello, se implementó un *connectionpool* así como el protocolo de comunicaciones RTJRMP.

Tres limitaciones de índole práctica que tiene el prototipo desarrollado son: (1) que ha de poseer una fase de inicialización donde se reserven e inicialicen ciertas partes internas de prototipo como son las tablas utilizadas por el mecanismo de invocación remota en el servidor, (2) la carencia práctica de un servicio de nombres, lo que nos obligado ha hacer uso del sistema de ficheros local para intercambiar referencias a objetos remotos entre diferentes máquinas y (3) la imposibilidad de utilizar en el servidor hilos de tipo `NoHeapRealtimeThread` y `RealtimeThread` al mismo tiempo, debiendo de recurrirse a unos o a otros.

Como línea futura a corto-medio plazo se prevé completar el prototipo con un modelo de asincronismo tanto confirmado por el servidor como sin confirmar así como con un servicio de nombres en tiempo real. Y además, se pretende dotar a la abs-

tracción de gestión de prioridades de un mayor grado de flexibilidad permitiendo que el sustituto pueda influir sobre el comportamiento del servidor, tal y como describe en el sistema de interfaces de DREQUIEMI. Y por último, si finalmente se consigue el acceso al código fuente de jTime, se pretende implementar la `AGCMemory` y el `RealtimeThread++`.

Por último, en una estancia realizada durante el año 2006 en la Universidad de Aveiro se ha estado trabajando en el soporte de mecanismos de gestión de tiempo real basados en los del mecanismo FTT [138], dentro del modelo de comunicaciones RTRMI de DREQUIEMI bajo la forma de un nuevo servicio de sincronización global. El laboratorio de tiempo real (LSE) de esta universidad tiene una larga trayectoria desarrollando un modelo computación denominado FTT que ha sido aplicado a diferentes tecnologías -CAN, Ethernet, switched Ethernet y más recientemente CORBA- generando soluciones específicas para cada uno esos entornos. Durante esta estancia se ha estado trabajando de forma conjunta sobre cómo utilizar el modelo FTT en el modelo DREQUIEMI, definiendo para ello un nuevo servicio de planificación centralizado capaz de ejecutarse sobre el software DREQUIEMI. Para más detalles ver [15].

El software desarrollado ha sido probado en varios sistemas operativos de tiempo real gracias, en parte, al alto grado de portabilidad ofrecido por el modelo de máquina virtual. Sin embargo, debido a que las clases RMI utilizadas tienen una fuerte dependencia con la máquina virtual utilizada, la única máquina virtual sobre la que llega a ejecutarse es jTtime, no siendo posible ejecutarlo con otras máquinas virtuales Java de tiempo real. Actualmente, la versión de jTime sobre la que ejecuta es la 1.0 (build 1.0fcs-ar[1.0.0547],native threads) aunque se ha empezado a trabajar en la migración a la versión 1.1b. A nivel sistema operativo se ha comprobado que funciona adecuadamente en sistemas Linux de no tiempo real -debian Woody, Knoppix y Redhat- así como en otros de tiempo real -TIMESYS [175] y RTAI [51].

6.2. Escenarios de prueba

Tanto a la hora de desarrollar las extensiones como a la hora de probarlas se ha puesto de manifiesto la existencia dos tipos de escenarios, el centralizado y el distribuido, sobre los cuales se podían realizar mediciones que sirviesen para corroborar tanto el correcto comportamiento del middleware como que permitiesen obtener valores significativos sobre su comportamiento interno.

El escenario centralizado -ver figura 6.1- es aquel donde múltiples máquinas virtuales ejecutan sobre el mismo procesador y es un buen escenario tanto para desarrollar el prototipo como para comenzar a probarlo. Trabajando en este tipo de escenario se puede establecer una estricta precedencia y no solapamiento entre la ejecución del cliente y la del servidor, lo que permite obtener un control más fino sobre la ejecución que el disponible en un entorno distribuido. Además, el hecho de que tanto el cliente como el servidor se estén ejecutando sobre un mismo procesador permite compartir una misma base temporal que puede ser utilizada para calcular tiempos de respuesta entre un cliente y un servidor o el consumo de procesador realizado. Y por último, el hecho de que la memoria consumida durante la invocación remota no guarde dependencia directa con el comportamiento centralizado o distri-

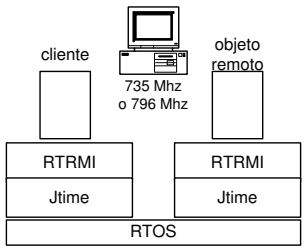


Figura 6.1: Escenario de medidas centralizado

Característica	valor
modelo	AMD Atholon XP 2400++
velocidad	796.104 Mhz
caché	512 KB
RAM	430236 Kb
TSYS	2.4.7-TIMESYS-3.1.214
	egcs-2.91.66
OS	Knoppix 2.4.22-xfs
	egcs-2.95.4
distribución	Knoppix *kernel 2.4.22*

Cuadro 6.1: Principales características del ordenador portátil

buido de la máquina virtual, permite realizar mediciones de consumo de memoria en entornos centralizados fácilmente extrapolables a distribuidos. Sin embargo, y ésta es quizás su mayor limitación, el escenario centralizado es bastante malo a la hora de cuantificar cuáles son las inversiones de prioridad que son sufridas por los clientes de alta prioridad cuando hay interferentes de menor nivel de prioridad compitiendo por el acceso a un mismo objeto remoto.

Durante el desarrollo e implementación del sistema se han utilizado dos tipos de infraestructura hardware: un portátil a 2.1 Ghz y cinco ordenadores fijos de la gama pentium a 733 Mhz. El cuadro 6.1 y el cuadro 6.2 aportan más detalles, extraídos a través de los ficheros `/proc/cpuinfo` y `/proc/meminfo`, sobre cada uno de ellos.

El hecho de que en entornos distribuidos se disponga de cierto paralelismo nos impide extrapolar directamente los resultados obtenidos en el escenario centralizado a uno distribuido. Y por ello se ha construido un escenario distribuido complementario. Este escenario aunque no resulta bueno para verificar el correcto funcionamiento del middleware, sí que lo es para la obtención de medidas válidas sobre el rendimiento del sistema y medidas relacionadas con la inversión de prioridad ocasionada por las zonas de código común del middleware de distribución.

La figura 6.2 muestra el escenario distribuido que se ha utilizado para realizar las mediciones. Hay un portátil, descrito en el cuadro 6.1, y cinco ordenadores fijos, descritos en el cuadro 6.2. Cada uno de ellos posee una dirección IP perteneciente a la subred 192.168.8.0. Físicamente esta subred está soportada por una red de tipo *switched ethernet* a 100 Mbps. Las tarjetas de red de los ordenadores fijos son 3COM-

Característica	valor
modelo	Celeron *Coopermine*
velocidad	735.014 Mhz
caché	128 KB
RAM	120800 kB
TSYS	2.4.7-TIMESYS-3.1.214
	egcs-2.91.66
RTAI	2.4.21-adeos
	egcs-3.2.2
distribución	Redhat 9 *Shrike*

Cuadro 6.2: Principales características de los ordenadores fijos

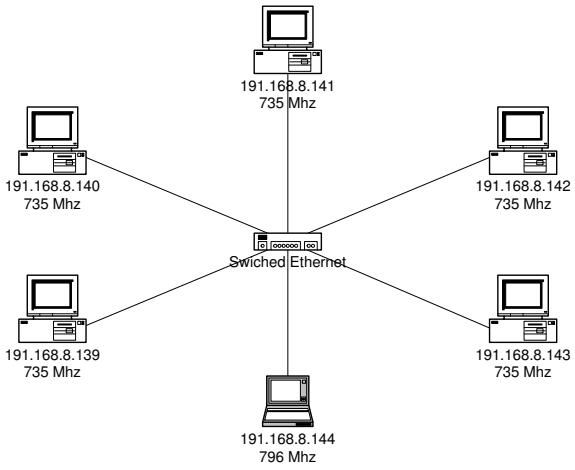


Figura 6.2: Escenario distribuido de medidas

590 a 100 Mbps. A fin de evitar interferencias externas, la red se encuentra aislada. El hecho de que no haya compartición real del medio físico, de forma contraria a lo que sucede en ethernet clásico, evita las colisiones existentes en el medio de transmisión proveyéndonos de una red de tiempo real de bajo coste.

6.3. Aplicaciones auxiliares

Durante el desarrollo y la validación del prototipo se han creado una serie de herramientas que han permitido depurar el funcionamiento interno del middleware así como realizar una serie de mediciones sobre el consumo de recursos. Dado que pueden ser interesantes para el desarrollo de otros middlewares de distribución, a continuación, las describimos brevemente enumerando sus principales características y mostrando, cuando sea adecuado, ejemplos de su funcionamiento.

6.3.1. DRQTracer

Esta herramienta resulta básica a la hora de trazar el consumo de memoria y las latencias introducidas por el middleware de distribución. Permite, haciendo internamente uso del reloj del sistema y de los métodos de un `MemoryArea`, salvar en el instante en que se invoca su método estático `photo(çheckpoint")` el estado del sistema. Este estado comprende un identificador, pasado como parámetro; el instante temporal en que esto se produce, almacenado en un `AbsoluteTime`; y la memoria consumida en el montículo Java, en la memoria immortal y en la *memoryarea* que se encuentra en la cima del *scopestack* del hilo invocante.

Una característica muy importante de esta herramienta de medida es el de que no introduce interferencias en el consumo de memoria. Todos sus métodos han sido diseñados para que la memoria se reserve en la fase de inicialización de la clase, mediante un constructor estático de la clase, o en la fase de ejecución usando una región auxiliar para eliminar los objetos creados durante el proceso de volcado por pantalla. Lo que de forma práctica permite trazar el consumo de memoria de un método remoto de forma exacta, introduciendo ciertos puntos de traza en partes bien conocidas del código Java, sin que la herramienta introduzca interferencias.

De forma contraria, su utilización si que interfiere en las medidas temporales obtenidas. Los diferentes pasos que son realizados a la hora de acceder al reloj del sistema así como al estado del hilo que invoca al método `photo` falsean las medidas, introduciendo un coste fijo y determinístico y otro variable.

La figura muestra este efecto de forma experimental, tratando de ver cuál es el comportamiento de la infraestructura de prueba. En ella se muestra el coste de 32000 invocaciones consecutivas al método `photo` en tres escenarios: (1) el proporcionado por un sistema operativo tradicional, (2) el del sistema operativo de tiempo real RTAI y (3) el de TIMESYS. El caso del sistema operativo de tiempo real TIMESYS se ha medido tanto con la infraestructura portátil, descrita en la tabla 6.1, como en la fija, descrita en la tabla 6.2, mientras que el del sistema operativo tradicional ha sido medido tan sólo en la infraestructura portátil. En todos los casos para obtener las muestras se ha realizado una ejecución a la prioridad máxima del sistema y ejecutando con el usuario `root`, lo que reduce la interferencia interna causada por otras tareas del sistema de menor prioridad.

No se observan grandes diferencias en cuanto a lo que es el coste medio de la operación `photo(null)`. En el portátil el coste medio por medida de traza es de 9.96 μs si se está ejecutando con un sistema que no es de tiempo real, mientras que es un poco más ineficiente -10.02 μs- en el caso de que se utilice TIMESYS. En infraestructura fija estos costes son de 10.77 μs para RTAI y de 10.79 μs para TIMESYS.

En cuanto a la variación máxima que puede darse cuando se realizan estas medidas se ha observado que existen ciertas variaciones significativas entre el mejor y el peor de los casos. Así, en las muestras obtenidas para el portátil el valor máximo medido es de 37 μs en el caso de que se tenga un sistema operativo de no tiempo real mientras que este valor desciende hasta 31 μs en el caso de que se utilice TIMESYS. De forma similar, el tiempo máximo necesario para realizar una medición es de 76 μs si se utiliza RTAI y de 57 μs en el caso de utilizarse TIMESYS.

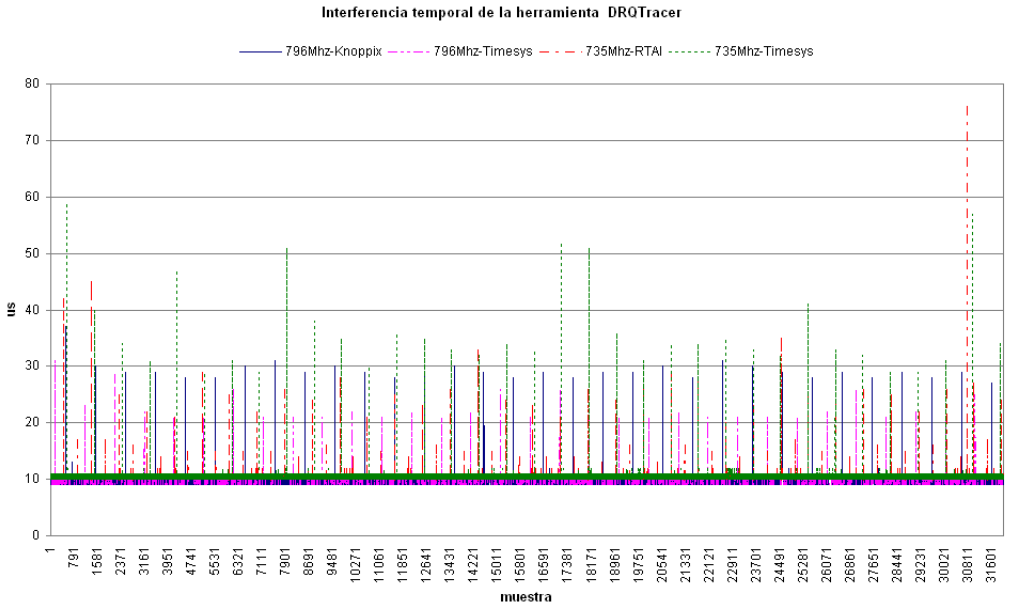


Figura 6.3: Coste temporal introducido por la herramienta de medición

El error de esta herramienta de mediciones es el que posee nuestra plataforma de experimentación y ha de ser tenido en cuenta a la hora de realizar mediciones.

6.3.2. SharedRemoteObject

Este objeto remoto ha sido diseñado con el objetivo de ser utilizado a la hora de realizar mediciones de la inversión de prioridad que es introducida por la infraestructura de comunicaciones. Este objeto remoto posee un método remoto, `void dowork(int work)`, que consume cierta cantidad de procesador de forma activa, haciendo uso de un par de bucles anidados. El parámetro `work` controla el número máximo de ejecuciones que realiza uno de los bucles. Esto permite que un cliente de forma dinámica pueda variar el coste total de la invocación remota aumentando el consumo de procesador realizado en el nodo servidor.

Veamos de forma empírica cuál es la relación existente entre el parámetro `work` y el coste de la invocación remota síncrona al objeto remoto de tipo `SharedRemoteObject` de forma experimental. La figura 6.4 nos muestra cuál es el tiempo necesario para realizar una invocación al método remoto `dowork(work)` en ausencia de otras tareas intentando acceder al objeto remoto, mostrándonos la relación existente entre el parámetro `work` y el coste temporal de la invocación remota en tres configuraciones. Dos de los experimentos han sido realizados en un entorno centralizado, primero en el portátil de 796 Mhz y después en el ordenador fijo de 733 Mhz, mientras que el tercero ha sido realizado en un entorno distribuido con dos ordenadores a 733 Mhz conectados con una red switched ethernet de 100 Mbps. En todos los casos, el sistema operativo subyacente utilizado ha sido TIMESYS y la prioridad utilizada para

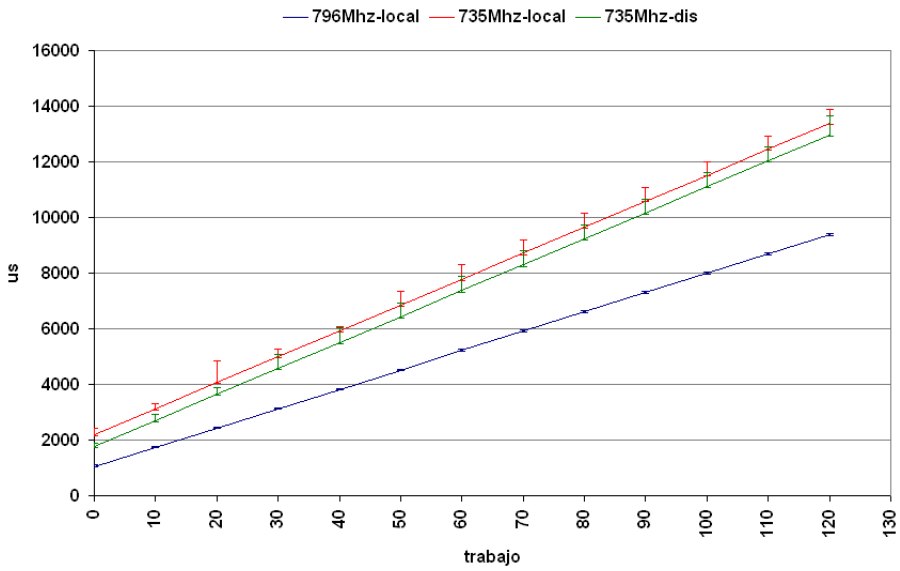


Figura 6.4: Coste de la invocación a `doWork` en diferentes escenarios

ejecutar el programa ha sido la máxima del sistema, a fin de evitar las interferencias generadas por otras tareas. Para cada punto representado se muestra el valor mínimo, máximo y medio alcanzado en una secuencia de 10000 invocaciones remotas consecutivas a dicho método remoto. Los valores medios obtenidos aparecen unidos por una línea que interpola aquellos valores que no son múltiplos de 10.

Los resultados nos muestran varios detalles interesantes tanto sobre la variabilidad de los resultados obtenidos así como sobre el rendimiento medio de las diferentes infraestructuras utilizadas en la evaluación empírica.

El primero de ellos es que aún en ausencia de tareas de mayor prioridad existe cierta variabilidad en lo que es el coste de la invocación remota debida a la infraestructura (sistema operativo, máquina virtual y conexión de red) utilizada. En el caso del ordenador portátil esta variación se sitúa por debajo de los 80 μs pudiendo venir derivada en gran parte del propio proceso de medida, que introducía un error de 60 μs. En el caso centralizado con ordenador fijo, es mayor y alcanzando los 700 μs. Y por último, en el caso de tener un sistema distribuido este valor puede llegar a situarse alrededor de los 600 μs.

El segundo es el relativo al rendimiento de las infraestructuras utilizadas. El menor coste se obtiene con la infraestructura más rápida que es la del portátil, mientras que el peor lugar lo ocupa la infraestructura fija en modo centralizado. Y por último, el escenario distribuido mejora el tiempo de respuesta del centralizado debido mayormente a que ciertos procesos como son el envío y la recepción de datos pueden ser realizados en paralelo reduciéndose así el coste total de la invocación remota. De forma práctica, en el caso de trabajo 0, este coste se reduce de 2133 μs a 1769 μs, ganándose alrededor de 400 μs en el tiempo de respuesta del sistema distribuido frente al del centralizado. Ganancia que se mantiene independientemente

del trabajo exigido al servidor.

6.3.3. DRQTestResourceConsumption

De forma general, el objetivo de esta herramienta es el de cuantificar los recursos que son consumidos durante la realización de la invocación remota a un determinado objeto remoto. Para ello se mide cuánta memoria es necesaria para realizar una invocación remota así como las latencias introducidas por la invocación remota.

Dado que es imposible evaluar todos los diferentes tipos de métodos remotos existentes se ha escogido un subconjunto representativo. Básicamente el estudio aparece centrado en aquellos objetos remotos que no realizan ningún tipo de operación durante su ejecución, donde el coste computacional de la invocación remota recae en la infraestructura de ejecución encargada de transferir los datos entre los diferentes nodos del sistema.

En total, se realizan pruebas en tres tipos de familias de métodos remotos:

- `void doNothing(X)`
- `X doNothing()`
- `X echo(X)`

La primera de ellas intenta ver cómo influye el envío de datos, bien sean primitivos u objetos, desde el cliente al servidor en el coste total de la invocación remota. La segunda hace lo mismo pero centrándose en el flujo de retorno desde el servidor al cliente. Y por último, la tercera familia representa a aquellos métodos remotos donde se envía un dato que tras ser procesado en el servidor es, más tarde, devuelto al cliente.

Y para estudiar cada una de esas familias se utilizan 34 parámetros diferentes. La tabla 6.3 nos muestra los datos que se han seleccionado para sustituir a la `X` en cada una de las tres familias de métodos remotos estudiadas. En primer lugar aparecen los 7 datos primitivos Java, a continuación sus equivalentes objetos, tras ello aparece un objeto remoto de tiempo real y por último aparecen los datos estructurados que representan a aquellos escenarios donde existen flujos de datos pesados. Para estos flujos pesados se han escogido tres subcasos de estudio: la cadena de caracteres, el array de objetos y el vector. En el caso de la cadena de caracteres los datos almacenados son caracteres de 16 bits mientras que en el caso del array y del vector son objetos de tipo `java.lang.Double`. Lo que hace que en total se estén contemplando 102 casos de prueba: 34 variantes en tres familias distintas.

6.3.4. DRQJitterTracer

Esta herramienta calcula el coste de la invocación remota almacenando el instante temporal en el que comienza y finaliza cada una de las invocaciones remotas. Los parámetros de configuración con los que cuenta son los siguientes:

- $prio_{client}$. Esto es la prioridad a la que ejecuta tanto el cliente como el servidor.

Tipo	Tamaño	Descripción
void	-	Tipo vacío
byte	1	Entero de 8-bits complemento a dos
short	2	Entero de 16-bits complemento a dos
int	4	Entero de 32-bits complemento a dos
long	8	Entero de 64-bits complemento a dos
float	4	Real de 32-bits formato IEEE 754-1985
double	8	Real de 64-bits formato IEEE 754-1985
char	2	Carácter unicode de 16 bits
boolean	1	Dato boolean de tamaño 1-bit
null	8	Referencia a objeto
Byte	16	Objeto tipo `java.lang.Byte`
Short	16	Objeto tipo `java.lang.Short`
Integer	16	Objeto tipo `java.lang.Integer`
Long	20	Objeto tipo `java.lang.Long`
Float	16	Objeto tipo `java.lang.Float`
Double	20	Objeto tipo `java.lang.Double`
Character	16	Objeto tipo `java.lang.Character`
Boolean	16	Objeto tipo `java.lang.Double`
RemoteObject	48 [1]	Instancia de la clase `RealtimeUnicastRemoteObject`
String()	40	Instancia de la clase `java.lang.String` vacía
String(10)	60	Instancia de la clase `java.lang.String` con 10 caracteres
String(25)	92	Instancia de la clase `java.lang.String` con 25 caracteres
String(50)	140	Instancia de la clase `java.lang.String` con 50 caracteres
String(100)	240	Instancia de la clase `java.lang.String` con 100 caracteres
Object[0]	16	Array de 0 objetos
Object[10]	256	Array de 10 objetos `Double`
Object[25]	616	Array de 25 objetos `Double`
Object[50]	1216	Array de 50 objetos `Double`
Object[100]	2416	Array de 100 objetos `Double`
Vector(0)	44	`java.util.Vector` vacío
Vector(10)	280	`java.util.Vector` con 10 objetos `Double`
Vector(25)	664	`java.util.Vector` con 25 objetos `Double`
Vector(50)	1235	`java.util.Vector` con 50 objetos `Double`
Vector(100)	2444	`java.util.Vector` con 100 objetos `Double`

Cuadro 6.3: Tipos de datos utilizados por `DRQTestResourceConsumption`

- $prio_{remote}$. Esto es la prioridad que se utiliza en el servidor para empezar a atender la invocación remota hasta que se consiguen procesar los datos necesarios para ejecutar el servidor a la prioridad del cliente.
- *ini*. Esto es el trabajo inicial que se pretende que realice el servidor.
- *inc*. Esto es el incremento de trabajo hecho entre cada una de las muestras.
- *end*. Esto es el valor máximo de trabajo que determina la finalización del experimento.
- *samples*. Esto es el número de muestras tomadas para el cálculo de cada valor.

Haciendo uso de estos valores, en una fase de inicialización, la aplicación establece una conexión con un objeto de tipo `SharedRemoteObject` fijando su prioridad a $prio_{remote}$. Tras ello, modifica la suya propia fijándola a $prio_{client}$. Tras ello, ya en la de misión, el programa utilizando un bucle comienza a medir los tiempos consumidos por cada una de las invocaciones remotas al método `doWork` con los valores de dentro del intervalo $[ini, end]$ separados *inc* unidades entre sí. Para cada valor son tomadas *samples* muestras consecutivas que a posteriori, tras la finalización del experimento, son volcadas por la salida estándar. A modo de ejemplo, la gráfica de la figura 6.4 ha sido obtenida con los siguientes parámetros: 79, 79, 0, 10, 120, 10000.

6.3.5. DRQWorkTracer

Esta herramienta es similar a la anterior pero en vez de medir variaciones en el coste mide el coste medio de la invocación, siendo los parámetros manejados los siguientes:

- $prio_{client}$. Esto es la prioridad a la que ejecuta tanto el cliente como el servidor.
- $prio_{remote}$. Esto es la prioridad que es utilizada en el servidor para empezar a atender la invocación remota hasta que se consigue cambiar la prioridad del servidor por la del cliente.
- *ini*. Esto es el trabajo inicial que se pretende que realice el servidor.
- *inc*. Esto es el incremento de trabajo realizado entre cada una de las muestras.
- *end*. Esto es el valor máximo de trabajo para el cual se realizan mediciones.
- *samples*. Esto es el número de muestras utilizadas para el cálculo de cada valor.

La principal diferencia entre esta herramienta y la anterior radica en que el tiempo tan solo es tomado dos veces en cada muestra: al principio y al fin. Esto elimina, si se toman muchas muestras, la interferencia introducida por la herramienta de medida, perdiéndose, a cambio, la información fina sobre costes máximos o mínimos de invocaciones remotas aisladas. Al igual que en el caso anterior, los resultados son volcados tanto por la salida estándar como a fichero.

6.3.6. DRQForeverTracer

Esta pequeña utilidad ha sido desarrollada para crear clientes que acceden a un objeto remoto de tipo `SharedRemoteObject` con un cierto patrón de ráfaga. Los parámetros que admite son los siguientes:

- $prio_{client}$. Esto es la prioridad a la que ejecuta tanto el cliente como el servidor.
- $prio_{remote}$. Esto es la prioridad que es utilizada en el servidor para empezar a atender la invocación remota hasta que se consiguen procesar los datos necesarios para ejecutar el servidor a una prioridad igual a la del cliente.
- *work*. Esto es la cantidad de trabajo que se le pide que realice al método remoto `doWork` del `SharedRemoteObject`.
- *samples*. Este es el número de invocaciones remotas consecutivas realizadas que componen una ráfaga.
- *sleep*. Esto son los milisegundos que descansa el cliente desde que termina de introducir una ráfaga hasta que inicia la siguiente.

La aplicación muestra el tiempo medio consumido en la realización de una ráfaga interferencia tras la realización de ésta.

6.4. Reducción de la inversión de prioridad extremo a extremo mediante el empleo de prioridades

A lo largo de esta sección se procede a cuantificar cuáles son los beneficios que en términos de reducción de inversión de prioridad extremo a extremo aporta el modelo de prioridades extremo a extremo propuesto para DREQUIEMI. Para ello, en todos los experimentos se parte de la existencia de un hilo que está intentando hacer invocaciones remotas a la prioridad máxima del sistema y lo que se intenta medir es cómo la existencia de otros hilos de menor prioridad que también hacen uso del mismo objeto remoto llegan o no a influir en los tiempos de respuesta experimentados por el de mayor prioridad.

Se parte del conocimiento del coste de la invocación remota en ausencia de competencia, empíricamente obtenido para diferentes escenarios y cuyo valor se puede consultar en la figura 6.4. Y a partir de este escenario ideal se introducen variaciones como puede ser la existencia de otros hilos intentando acceder al sistema o el uso de bandas de prioridades compartidas en el proceso interno de aceptación del middleware.

En todos estos casos se calcula cuál es el efecto que introduce ese tipo de operaciones en la nueva curva del tiempo de respuesta de la tarea de máxima prioridad.

Por último, se estudia el comportamiento de un objeto remoto tradicional RMI donde no existe tal tipo de funcionalidad, comprobando que las tareas de baja prioridad introducen altas inversiones de prioridad en las de mayor.

6.4.1. Interferencia introducida por tareas de baja prioridad

La primera evaluación que se quería realizar es la de la interferencia que sufre una tarea de tiempo real que intenta acceder a un objeto remoto haciendo uso de la prioridad máxima del sistema cuando existen otras tareas de menor nivel de prioridad residentes en otros nodos que también están intentando realizar tal operación. Idealmente, la tarea de mayor prioridad no debería de sufrir ningún tipo de inversión de prioridad, pero debido a que la expulsividad de la plataforma de ejecución no es ideal, tal y como veremos experimentalmente, sí que sufre cierta interferencia no nula.

Para evaluar este efecto se realizaron cuatro experimentos:

- En el primero no existe ningún tipo de tarea interferente. El cliente es un `DRQ-JitterTracer` residente en 192.168.8.141 que tiene prioridad de ejecución 79 y que establece una conexión con prioridad de aceptación 79 con un `Shared-RemoteObject` que reside en 192.168.8.143. Este objeto remoto funciona con un modelo de prioridades propagadas desde el cliente, con lo que todas las peticiones de 192.168.8.142 a `doWork` se ejecutarán a prioridad 79.

- En el segundo hay un interferente de baja prioridad. Éste ejecuta a prioridad 78, preestableciendo una conexión con prioridad de aceptación 78 con el `SharedRemoteObject` residente en 192.168.8.143. A fin de maximizar la interferencia causada por las tareas interferentes, se invoca a `doWork(0)` 10000 veces consecutivas, descansando 0 ms antes de generar la siguiente ráfaga de interferencia.

- En el tercero se elige una situación en la cual el objeto remoto invocado está saturado por clientes de baja prioridad. A fin de maximizar el efecto se habilita otro cliente que reside en la máquina 192.168.8.143 que también introduce una interferencia de 10000 invocaciones consecutivas con 0 ms de descanso entre ráfaga y ráfaga pero cuya prioridad de atención ya no es de 78 sino 77. Además, en 192.168.8.140 se ha puesto otro cliente a interferir a prioridad 76 que es atendido haciendo uso de una prioridad de atención 76. En este escenario la carga introducida por los 3 clientes interferentes y el de máxima prioridad es máxima, en el sentido de que el cliente de prioridad 76 no es capaz de realizar ninguna invocación remota. Los clientes interferentes residentes en 192.168.8.143 y en 192.168.8.141 son capaces de consumir todo el tiempo disponible en el `SharedRemoteObject`, saturando el servidor con peticiones remotas. Esta no progresión del cliente de prioridad 76 es lo que nos garantiza que la interferencia creada en el servidor por los clientes de prioridades 78 y 77 es suficiente para saturarlo.

- En el cuarto se introduce un elevado número de clientes intentando acceder al objeto remoto aleatoriamente. Esto se consigue colocando cinco clientes en 192.168.8.140 y otros cinco en 192.168.142. Cada uno de ellos ejecuta a prioridad 77 y tiene una conexión preestablecida al mismo nivel de atención con el servidor. El objetivo de este escenario es que el patrón de interferencias sea

más aleatorio que el previo, encontrándose el servidor también saturado de peticiones remotas.

En todos los casos se realiza la misma medición. Partiendo de cero y con incrementos de 2, el cliente de mayor prioridad (79) se dedica a trazar el coste de la invocación remota al método `doWork()` para diferentes valores, obteniéndose valores tanto para el coste medio, el mínimo como el máximo de 10000 invocaciones consecutivas.

Con el valor medio obtenido se ha construido la gráfica de la figura 6.5. En ella se representa el coste medio de la invocación para valores de trabajo iguales o inferiores a 16 pues son aquellos donde el nivel de interferencia se hace más acusado.

De las tres configuraciones estudiadas la que se muestra más eficiente a la hora de introducir interferencia es la que consta de dos hilos interferentes. Este valor de interferencia alcanza un valor máximo absoluto en el origen donde toma un valor medio de 50 μs. Asintóticamente, cuando crece el trabajo, las cuatro gráficas convergen a 0. Lo que resulta lógico pues el coste medio de la interferencia introducida tiende a ser cero cuando aumenta la cantidad de trabajo realizado por la tarea de mayor prioridad.

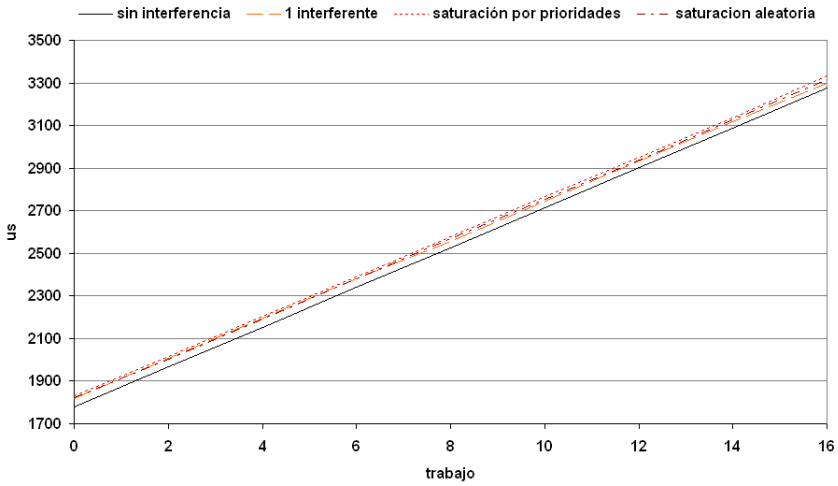


Figura 6.5: Evolución del coste medio de la invocación a `doWork` bajo la interferencia de tareas de menor prioridad

Si a ese valor medio le añadimos el mínimo y el máximo -ver figura 6.6[2]- y los comparamos, veremos que hay una fuerte dependencia con el escenario experimental donde se observen los resultados. La menor variación se obtiene en ausencia de otras tareas interferentes, donde se rondan los 600 μs de variación, y los máximos cuando trabajamos con escenarios saturados, donde nos aproximamos a los 1400 μs. Esto concuerda con lo que parece lógico, a mayor número de tareas activas, más sencillo es que se produzca el peor de los casos de ejecución.

[2]En esta figura el valor medio está representado por la columna gruesa, mostrándose el valor

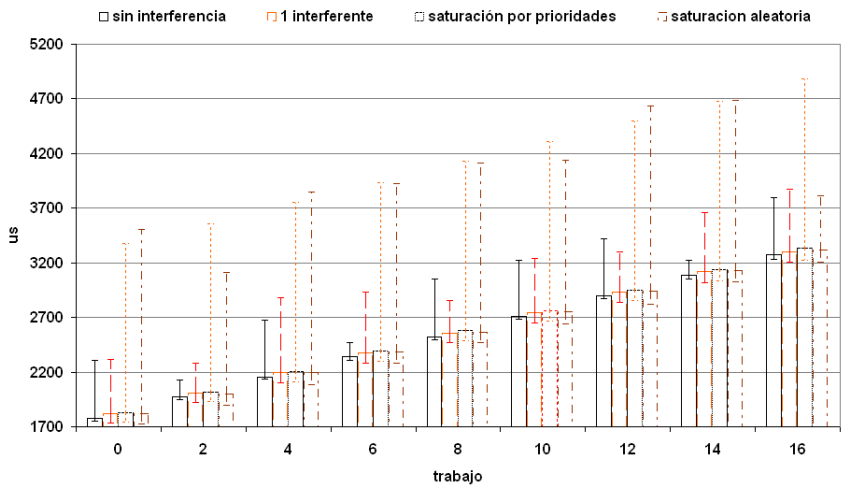


Figura 6.6: Coste máximo, medio y mínimo de la invocación a `doWork` bajo la interferencia de tareas de baja prioridad

En la figura también se puede ver como los valores mínimos observados en cada uno de los escenarios son bastante próximos, no existiendo grandes diferencias entre ellos. En éstos, la variación máxima entre el valor mínimo y el medio es de 60 μs.

6.4.2. Interferencia introducida cuando se comparte una prioridad de procesado inicial

Otro caso también interesante es aquel en el que existe una prioridad de aceptación a la cual se procesan las diferentes peticiones del sistema hasta que se consigue determinar cuál es la prioridad a la cual se tiene que atender la petición remota. En este tipo de sistemas la porción de código compartido es mayor produciéndose en consecuencia unos mayores incrementos en la inversión de prioridad que es experimentada por la tarea de mayor prioridad.

Para evaluar este fenómeno se utilizan los cuatro escenarios descritos en el experimento anterior. La única diferencia es que en todos los casos la prioridad de aceptación en vez de ser la misma que la del hilo cliente que pretende realizar la invocación remota ésta va a ser siempre 78. El resto de condiciones de la prueba se mantienen y las entidades concurrentes utilizadas son las mismas. Los hilos y la forma de generar la interferencia no sufren variaciones.

Las figuras 6.7 y 6.8 muestran respectivamente la evolución del valor medio así como los valores máximos, medios y mínimos de la invocación remota al método `doWork`.

Una de las primeras cosas que se observa es que el coste mínimo de la invocación se ve alterado aún en el caso de que no exista ningún tipo de tarea interferente. En

máximo y el mínimo con barras de error.

este caso los dos cambios de contexto realizados en el servidor en cada invocación remota ocasionan un incremento en el coste medio de la invocación remota de 19 μs, 8 μs por cada uno de los dos cambios realizados.

Tal y como resulta lógico, la interferencia media que un hilo de tiempo real sufre cuando comparte una misma banda de prioridad para el procesado inicial de la comunicación con otras tareas que están intentando acceder al middleware de comunicaciones es mayor que la existente en el caso previo, donde no había tal compartición. Y así, de valores de 50 μs obtenidos en el experimento anterior se pasa a valores medios cercanos a los 90 μs.

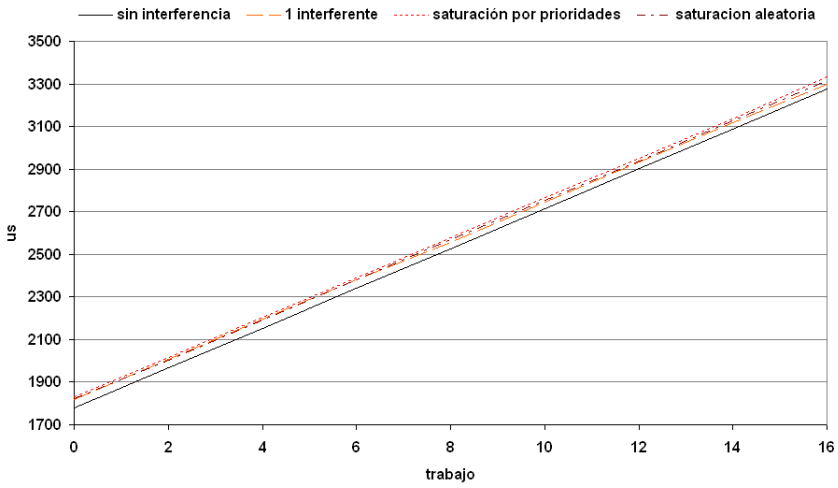


Figura 6.7: Evolución del coste medio de la invocación remota a `doWork` cuando se comparte la prioridad de aceptación

Si en vez de considerar el caso medio se estudia el coste en el peor y el mejor de los casos, al igual que sucedía en el caso anterior, se observan dos comportamientos diferentes. El caso en el que no hay o se tiene una baja interferencia, donde en el peor de los casos la variación observada se mantiene alrededor de los 600 μs, y el caso donde hay una interferencia alta y donde esos valores alcanzan los 1900 μs. También e igualmente a lo que sucedía en el experimento anterior, los valores mínimos se encuentran bastante cercanos a los medios (60 μs en el peor de los casos).

6.4.3. Interferencia causada por RMI tradicional

Se ha realizado también un experimento donde se midió cual es la interferencia causada en un cliente de alta prioridad por el uso de un servidor RMI tradicional, en vez de uno RTRMI, comprobándose experimentalmente que los costes de la invocación remota se disparan. Este experimento consiste en dejar de utilizar el sistema de prioridades propagadas de DREQUIEMI y pasar a utilizar el de RMI tradicional donde todos los hilos del servidor ejecutan a la misma prioridad: la máxima.

Se tienen en cuenta tres escenarios:

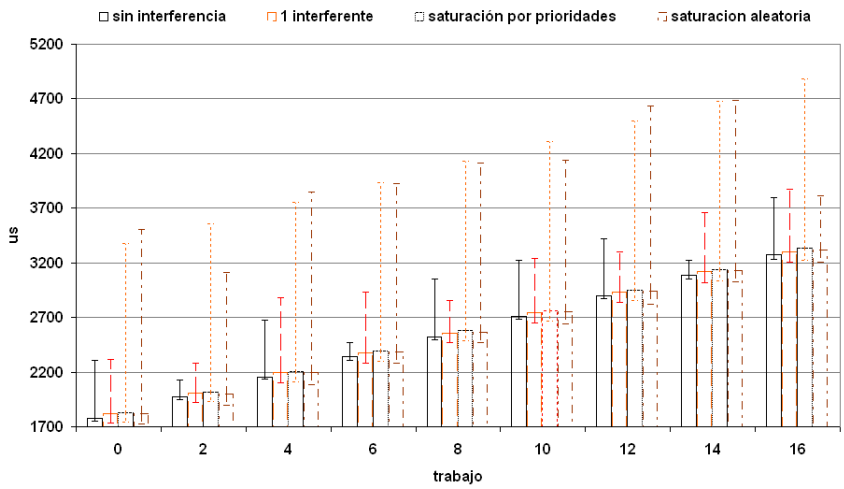


Figura 6.8: Coste máximo, medio y mínimo de la invocación remota a `doWork` cuando se comparte una misma prioridad de aceptación

- Sin ninguna tarea que produzca interferencia. `SharedRemoteObject` reside en 192.168.8.142 mientras que `DRQJitterTracer` lo hace en 192.168.8.141. La prioridad utilizada por éste es de 79 durante la ejecución de la invocación remota.

- Con una tarea que produzca interferencia. Además de las dos tareas descritas se añade un interferente, `DRQForeverTracer`, que ejecuta a prioridad 78 y que tiene establecida de antemano una conexión con el servidor que es atendida a esa misma prioridad.

- Con dos tareas en diferentes máquinas produciendo interferencia. Además de las tareas descritas anteriormente, se añade un nuevo interferente, `DRQForeverTracer`, que ejecuta a prioridad 78 y que tiene preestablecida una conexión con el servidor que se procesa a esa misma prioridad. Este interferente invoca 10000 veces a `doWork(0)` y tras descansar 0 ms vuelve a realizar otras 10000 invocaciones. Esta tarea interferente reside en 192.168.8.141.

Las figuras 6.9 y 6.10 nos muestra los resultados del experimento. En ellas se recoge tanto la tendencia en media del coste de la invocación remota para el hilo de mayor prioridad así como el valor medio, el mínimo y el máximo.

Los resultados medios muestran que el tiempo de respuesta de un hilo de un cliente de tiempo real, cuando intenta acceder a un servidor RMI tradicional, aumentan notablemente cuando existen otras tareas que también lo intentan. A diferencia del resto de los casos estudiados donde los incrementos en el coste de la invocación remota eran relativamente bajos, en este caso son muy significativos, pasándose de valores próximos a la centena de microsegundos (en ausencia de tareas interferentes) a las milésimas de segundo (cuando hay un número relativamente bajo de interferentes).

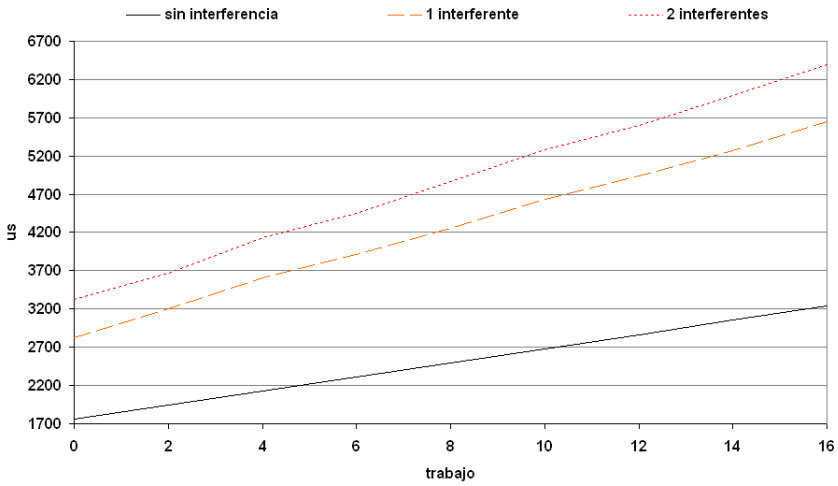


Figura 6.9: Evolución del coste medio de la invocación remota a `doWork` cuando existe un servidor RMI tradicional atendiendo peticiones

Así, el tiempo medio necesario para realizar una invocación remota a un objeto `SharedRemoteObject` cuando el trabajo exigido al servidor es 0 es de 1765 μs en ausencia de otras tareas interferentes, de 2821 μs cuando existe un único interferente de baja prioridad y de 3221 μs cuando existen dos.

La tendencia en cuanto al comportamiento observado es inversa a la experimentada hasta ahora. Si en los casos anteriores la inversión de prioridad disminuía cuando aumentaba el trabajo que debía de realizar el servidor, aquí aumenta. Ello es debido a que al aumentar la demanda de trabajo del servidor, también aumenta la interferencia introducida, lo que acaba implicando que se produzca un incremento en el coste total de la invocación remota.

Si analizamos (ver figura 6.10) los casos medios, máximos y mínimos vemos también que el coste se dispara en presencia de otras hebras interferentes, tal y como parece lógico. La interferencia máxima sufrida en el peor de los casos, que antes no superaba los 2 ms en casos con mucha sobrecarga llega ahora hasta los 120 ms con tan sólo dos hilos interferentes.

Claramente, el que el servidor compartido no tenga ningún tipo de gestión de las prioridades en el servidor influye negativamente en la tarea de tiempo con prioridad más alta pues ve como las de menor prioridad son capaces de robarle un mayor número de ciclos de procesador en el servidor, aumentando la inversión de prioridad experimentada.

6.4.4. Interferencia en entornos monoprocesador

Aunque se han realizado experimentos en entornos centralizados, no se ha obtenido ningún tipo de resultado significativo relacionado con la interferencia que una tarea de menor prioridad puede estar causando en una de mayor prioridad. El hecho

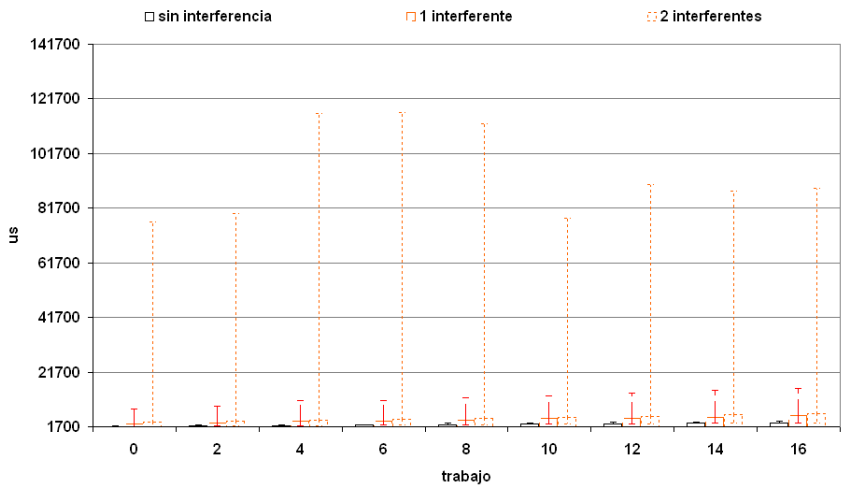


Figura 6.10: Coste máximo, medio y mínimo de la invocación remota a `doWork` cuando existe un servidor RMI tradicional

de que el procesador utilizado por cada máquina virtual sea el mismo bloquea el progreso de los clientes de baja prioridad, impidiendo que éstos puedan evolucionar e interferir en el hilo de mayor prioridad. Lo que nos lleva a la obtención de resultados experimentales donde la interferencia medida es de 0 μs.

6.4.5. Reflexión

En esta sección se ha realizado una cierta evaluación sobre la calidad del esquema de prioridades de DREQUIEMI. Los resultados obtenidos nos muestran un buen comportamiento de las tareas de tiempo real de mayor prioridad cuando se ven sometidas a interferencias de menor prioridad, corroborándose que utilizar un esquema de prioridades en el servidor puede llegar a ser altamente beneficioso. El coste mínimo de la invocación es de 1.7 ms en un sistema distribuido dotado de procesadores a 730 Mhz y las inversiones de prioridad experimentadas son, en media, inferiores a los 100 μs. Los resultados también nos muestran que el no tener ningún mecanismo que controle la prioridad a la que ejecuta el servidor puede implicar tiempos de respuesta mucho más altos próximos a los 120 ms cuando en vez de utilizar un servidor de tipo RTRMI se utiliza uno tradicional RMI.

Los peores resultados son quizás los relativos a la variación máxima experimentada en los tiempos de respuesta pues el haber utilizado una red switched ethernet, un sistema operativo de tiempo real, la máquina virtual jTime y el software DREQUIEMI, nos ha introducido incrementos adicionales en los tiempos de respuesta en el peor de los casos próximos a los 2 ms. Y este es quizás el punto del experimento que más se debería de intentar mejorar, buscando infraestructuras de ejecución más predecibles.

6.5. Reducción de la inversión de prioridad mediante el uso de regiones en el servidor

Otra evaluación realizada ha sido la de cuantificar cuál es la reducción de prioridad experimentada cuando en vez de utilizar el modelo de recolección de basura tradicional se utiliza uno alternativo basado en regiones. DREQUIEMI ha sido diseñado de tal manera que resulta posible utilizar el modelo basado en montículo Java y el de regiones, seleccionando para ello el `MemoryAreaPool` que es asignado a cada objeto remoto de tiempo real. Lo que nos permite realizar comparaciones entre los dos mecanismos de forma sencilla.

6.5.1. Caracterización temporal del coste de la invocación remota

El primer experimento realizado[3] ha estado orientado a la obtención del impacto que tienen las dos técnicas de gestión de memoria en el coste total de la invocación remota. Para ello se dispone de un único objeto remoto con tres tipos de métodos remotos diferentes y se han realizado múltiples invocaciones remotas a cada uno de ellos. En un primer escenario se ha utilizado un `HeapMemoryAreaPool` y en el segundo un `LTMemoryAreaPool`. Los tiempos de respuesta obtenidos para cada uno de los escenarios se muestran en la figura 6.11 y han sido medidos en el cliente.

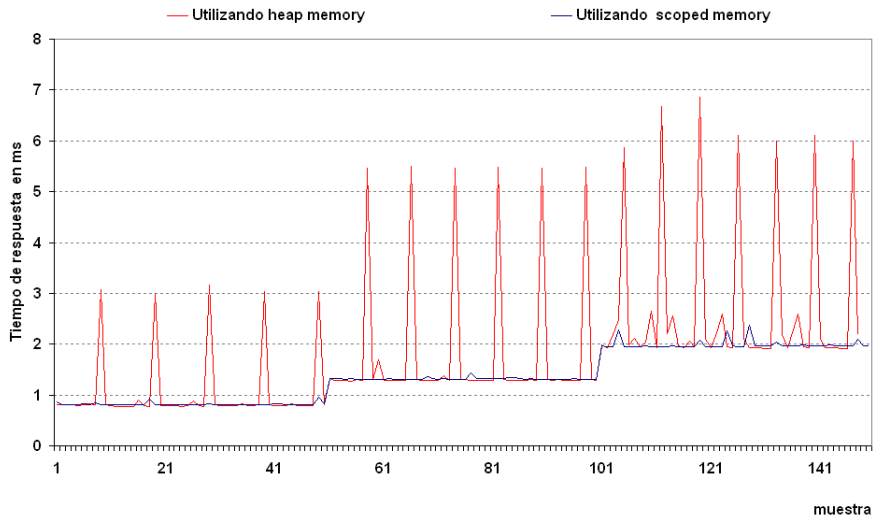


Figura 6.11: Influencia del recolector de basura y la técnica de regiones en el coste total de la invocación remota

El comportamiento del recolector de basura y de la técnica de regiones es bastante distinto; mientras el primero ofrece un comportamiento en picos, con coste desigual dependiendo de si este proceso ha lanzado el de recolección de basura o no,

[3]Todas las pruebas han sido realizadas en un entorno centralizado (servidor y cliente ejecutando en la misma máquina virtual) proporcionado por el portátil descrito en la tabla 6.1.

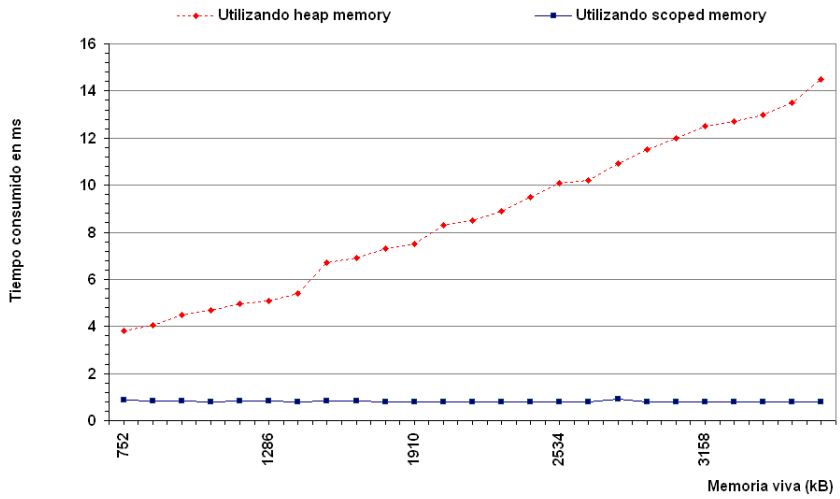


Figura 6.12: Comportamiento del coste de la invocación remota ante aumentos de la memoria viva

el segundo ofrece un comportamiento homogéneo en todas las invocaciones remotas. Los picos que son introducidos por el recolector de basura son debidos a que durante la invocación remota algunas veces resulta necesario reciclar el montículo Java añadiéndose este coste al de algunas de las invocaciones remotas. En el caso del `LTMemoryAreaPool` dado que cada bloque de memoria aportado por el middleware es reciclado tras la invocación remota, el coste de liberación del bloque se añade al de invocación, provocando la aparición de un comportamiento plano.

6.5.2. Efecto introducido por el aumento del tamaño de la memoria viva

Este comportamiento del recolector de basura se hace más acusado al aumentar el tamaño de la memoria ocupada. Así, al aumentar el tamaño de memoria ocupada, el coste de recolectar ese tipo de estructura crece más o menos linealmente con el tamaño de la memoria ocupada, provocando que en el peor de los casos el peor tiempo de repuesta del servidor crezca también de forma lineal.

La figura 6.12 da buena cuenta de ello. En ella se muestra el peor de los tiempos necesario para realizar una invocación remota a `doNothing()` cuando aumenta el tamaño de la memoria ocupada. Para aumentar el tamaño de memoria desde 752 kb a 4000 kb se ha utilizado un `java.util.vector` almacenado en un atributo estático de una clase auxiliar, y se le han ido añadiendo objetos consiguiendo así aumentar el tamaño de la memoria ocupada. Para cada uno de los valores calculados se muestra el valor de pico alcanzado tanto por el recolector de basura como por la técnica de regiones.

Tal y como se puede ver en la figura, la técnica de regiones no ve aumentados sus

costes. La técnica de regiones, al estar basada en un algoritmo de contaje, no ve aumentado su coste, mientras que la de recolección de basura, basada en la exploración de memoria ocupada para determinar cuál es la que está libre, ve como aumentada la porción de memoria que ha de explorar. Lo que provoca que en este último caso exista una dependencia lineal entre la cantidad de memoria ocupada y el coste total de la invocación remota.

6.5.3. Variación en el tamaño del montículo

Por último, se ha realizado otro experimento en el cual se varía el tamaño del montículo utilizado, manteniendo el tamaño de la memoria ocupada sin variar. Así, para cada uno de los valores se ha trazado el peor de los tiempos de respuesta con las dos técnicas. Los resultados obtenidos se muestran en la figura 6.13.

Los valores explorados se mueven dentro del rango de los 40 kb a los 1000 kb. Para variar el tamaño de estos valores se ha utilizado la opción `-Xms` de la máquina virtual. Esta opción permite, cuando es creada la máquina virtual, fijar el tamaño máximo del montículo Java.

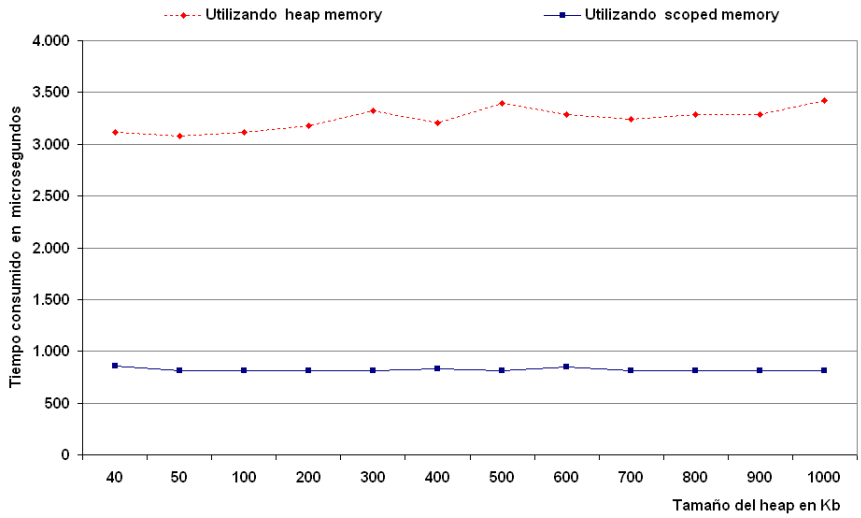


Figura 6.13: Influencia del aumento del tamaño del montículo Java en el tiempo máximo de respuesta remoto

Ni el recolector de basura ni la técnicas de regiones ven alterados sus costes máximos cuando aumenta el coste de la invocación remota. La técnica de regiones sigue proporcionando unos tiempos de respuesta menores que el peor de los tiempos del recolector de basura y su coste se mantiene más o menos igual cuando se varía el tamaño del montículo Java utilizado.

6.5.4. Reflexión

Los resultados obtenidos nos muestran que con la técnica de regiones pueden ser obtenidas reducciones en el coste de la invocación remota altamente significativas si se comparan con las que nos pueden ofrecer las técnicas basadas en recolectores de basura tradicionales. Los resultados también nos muestran que la técnica de regiones es capaz de distribuir el coste de la invocación remota, incluso ante incrementos significativos de la memoria ocupada.

Con este experimento queda demostrado que la inclusión de un mecanismo de recolección de basura basado en regiones dentro del middleware de distribución Java de tiempo real resulta interesante para el desarrollo de aplicaciones distribuidas.

Sin embargo, y ahí habría campo para trabajo futuro, los resultados obtenidos no nos permiten extrapolar ningún tipo de conclusión sobre los tiempos de respuesta de las regiones frente a los de las técnicas de recolección de basura de tiempo real. El hecho de que la comparación se haga con un algoritmo de recolección de basura que no es de tiempo real nos impide determinar cuál es la técnica que ofrece unos mejores tiempos de respuesta a las aplicaciones pues es bien sabido que los recolectores incrementales, bien utilizados, son capaces de reducir las latencias introducidas por la recolección de basura significativamente. Los resultados obtenidos a través de la experiencia realizada tan sólo nos permiten afirmar que las regiones son un mecanismo eficaz a la hora de obtener reducciones en el coste total de la invocación remota.

6.6. Análisis del consumo de memoria realizado durante la invocación remota

Otra de las evaluaciones a las que se ha sometido al prototipo de DREQUIEMI consiste en cuantificar cuál es el coste que tiene la invocación remota en términos de consumo de memoria. Durante la invocación remota hay una cierta demanda dinámica de memoria tanto en el cliente como en el servidor que es utilizada por el middleware de distribución para enviar y recibir datos entre los diferentes nodos de la red. Y dentro de este contexto, el objetivo de esta sección es el de investigar cuál es la cantidad de memoria que tanto el nodo cliente como el servidor consumen dinámicamente durante una invocación remota.

Este coste se torna especialmente interesante en el caso de que se desee utilizar la técnica regiones para eliminar la totalidad de los objetos creados durante la invocación remota. Así, en el cliente, los valores que se obtengan serían los que habría que utilizar a la hora de configurar el tamaño de la `LTMemory` auxiliar dentro de cuyo contexto se ejecutaría el proceso de la invocación remota en el nodo cliente. Y en el servidor estos valores serían los que habrían de ser utilizados a la hora de configurar el tamaño de cada uno de los bloques del `LTMemoryAreaPool` que se puede asociar al objeto remoto de tiempo real.

Para evaluar este consumo se ha utilizado la herramienta `DRQTestSizeParameters`. Agrupando y analizando el flujo de datos que nos proporciona hemos trazado el coste, en términos de memoria consumida de forma dinámica, en varios puntos de

la invocación remota. Lo que nos permite razonar sobre cuestiones sencillas como son el consumo absoluto de memoria realizado en el cliente y en el servidor, así como de otras más elaboradas como pueden ser la asimetría que se produce en el consumo de memoria en diferentes puntos de la ejecución de la invocación remota o la eficiencia de transmisión en bytes que presenta el mecanismo de comunicaciones.

6.6.1. Memoria total consumida durante el proceso de invocación remota

La primera medida que se ha realizado está orientada a determinar cuál es la cantidad de memoria que se consume a la hora de realizar la invocación remota tanto en el nodo cliente como en el servidor. Como este valor es dependiente del tipo de parámetros intercambiados entre cliente y servidor se han realizado los cálculos en los 102 escenarios proporcionados por las tres familias de métodos remotos que maneja `DRQTestResourceComsuption` (para más información volver a la sección 6.3.3).

Con los resultados obtenidos se han construido las dos gráficas de la figura 6.14. La primera gráfica contiene el consumo de memoria realizado en un nodo cliente cuando desea invocar a un método remoto mientras que la segunda contiene la misma información pero para un nodo actuando de servidor.

Tanto en el servidor como en el cliente se observa que existe un coste mínimo, en términos de memoria, para lo que es el proceso de invocación remota. En el prototipo de DREQUIEMI este coste mínimo se produce, tal y como parece lógico, cuando el método remoto invocado es `void doNothing()`. En este caso, se consumen 3600 bytes en el cliente y 5080 en el servidor. Esta memoria se utiliza tanto en el cliente como en el servidor para procesar tanto el protocolo RTJRMP como para serializar y deserializar los datos transmitidos desde el cliente al servidor. En el resto de los casos, tal y como se aprecia en las gráficas, este consumo es mucho mayor.

Tal y como parece lógico, el coste en términos de memoria, cuando en vez de trasmitir un dato primitivo se trasmite un tipo objeto equivalente, crece. El coste en caso de utilizar los objetos equivalentes a los datos primitivos supone un incremento en el consumo de memoria dinámica que en el peor de los casos es un 30 % mayor en el servidor y un 40 % en el cliente. Así, si en vez de invocar `long echo(long)` usamos su equivalente objeto `Long echo(Long)`, el consumo de memoria pasará de 3480 a 4904 bytes en el cliente y de 5192 a 6760 bytes en el servidor.

Un caso altamente interesante se produce cuando se envía una referencia a un objeto remoto. En este caso al coste ordinario de enviar los datos del cliente al servidor se añade uno extraordinario; el de realizar la comunicación con el algoritmo de recolección de basura remoto. Esto provoca que el coste en términos de memoria consumida se dispare, alcanzándose los 25000 bytes para el método remoto `X echo(X)`.

Por último, en los tipos de datos estructurados la tendencia observada también resulta lógica. Al aumentar la carga de datos, el consumo de memoria crece en todas y cada una de las tres familias de métodos remotos. El coste máximo observado, que se produce cuando se envían 100 objetos de tipo `Double` haciendo uso de un

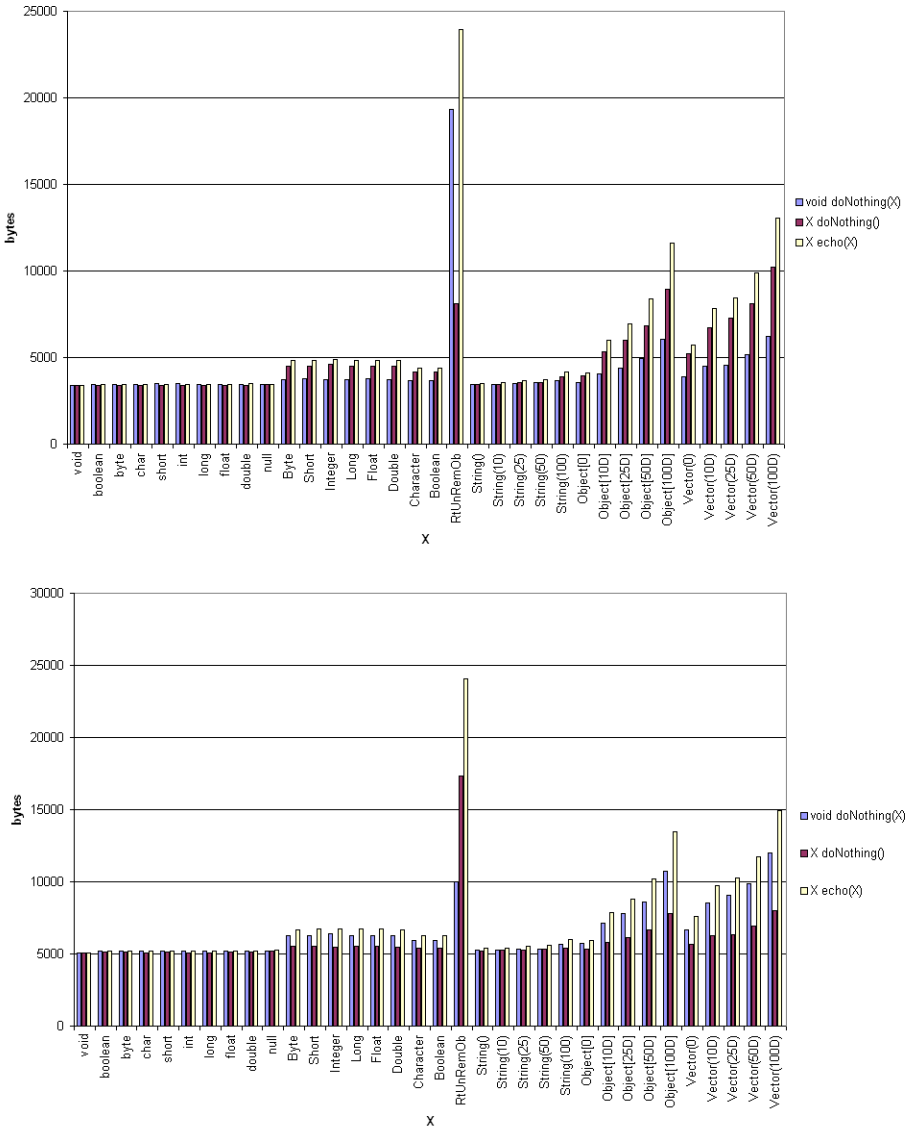


Figura 6.14: Consumo total de memoria durante la invocación remota realizado en el cliente y en el servidor

`java.lang.Vector`, es de 13060 bytes en el cliente y de 14916 bytes en el servidor.

6.6.2. Memoria necesaria para iniciar la invocación remota

Otro análisis que se torna interesante consiste en saber cuál es la cantidad de memoria que se consume en el cliente antes de recibir los resultados provenientes del servidor así como cuál es la cantidad de memoria que se consume en el servidor antes de que de comienzo la ejecución de la lógica del método remoto. Estos valores son buenos indicadores de lo que podría ser el coste optimista, en términos de memoria,

de la invocación remota asíncrona tanto con confirmación del servidor como sin ella.

La figura 6.15 muestra el consumo de memoria realizado, tanto en el cliente como en el servidor, justo al inicio de la invocación remota para cada una de las tres familias de métodos remotos estudiadas.

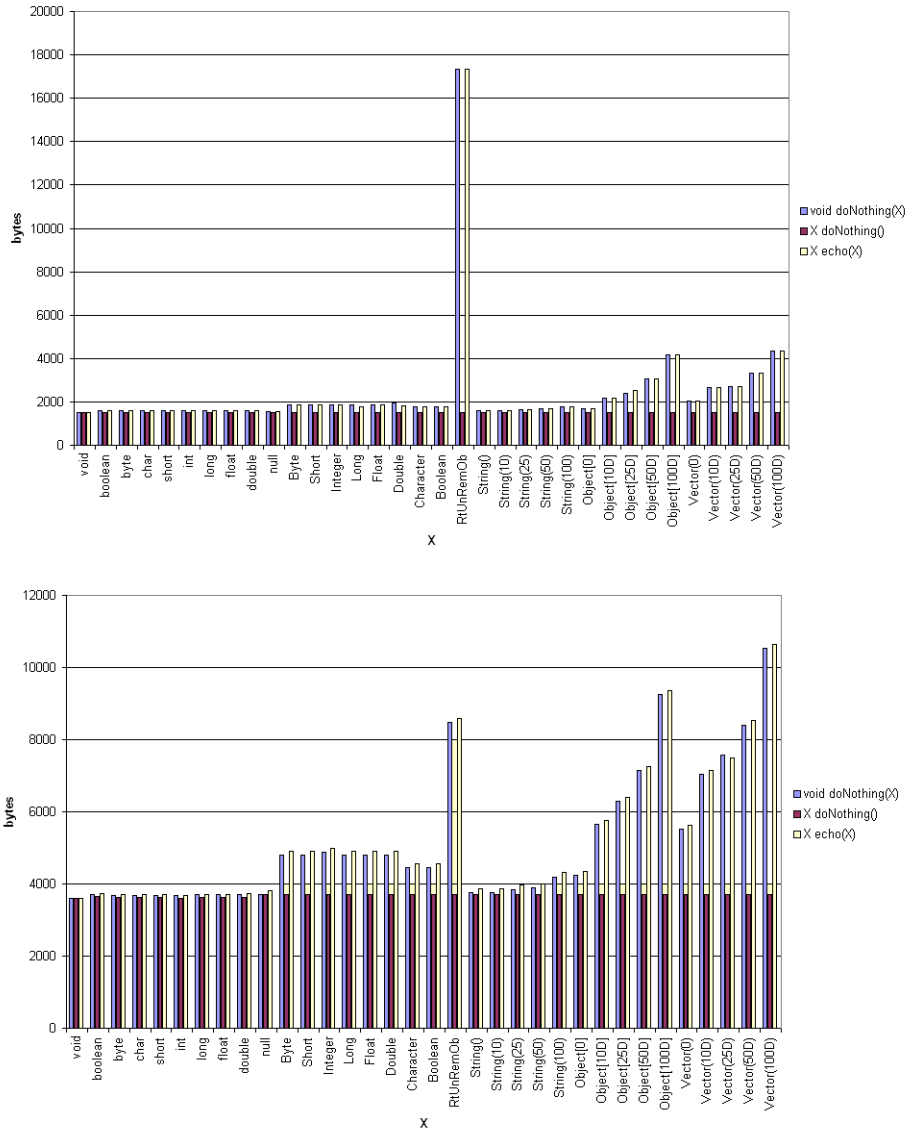


Figura 6.15: Memoria necesaria para iniciar la invocación remota

Las tendencias observadas son similares a las que se obtuvieron en el caso en el cual se observaba el consumo de memoria total realizado durante la invocación remota.

Observando los resultados se puede ver que la familia `void doNothing(X)` y la `X echo(X)` tienen un comportamiento similar y que `X doNothing()` presenta un coste constante, independiente del tipo de dato que los diferentes nodos hayan intercam-

biado. Teniendo el tipo de escenario utilizado en mente esto resulta lógico pues si sólo se mide la memoria que resulta necesaria para comenzar la invocación remota el método remoto `X doNothing()` equivale al `void doNothing()` y el `X echo(X)` al `void doNothing(X)`.

Pero quizás el resultado más interesante sea el que muestra que la cantidad de recursos necesarios para realizar la invocación remota puede sufrir fuertes reducciones si se utilizan las técnicas basadas en asincronismo. Así, en el cliente el coste mínimo para la invocación remota pasa de los 3392 a los 1524 bytes y en el servidor de los 5080 a los 3600 bytes. Porcentualmente esto significa que se pueden obtener reducciones en el consumo de memoria de un 55 % en el servidor y de un 29 % en el cliente cuando en vez de utilizarse métodos de comunicación síncronos se utilizan los asíncronos.

6.6.3. Asimetrías en el consumo de memoria durante la invocación remota

Sabiendo cuales son los costes totales y parciales de memoria medidos en varios puntos clave de la invocación remota es posible realizar ciertos cálculos sobre las diferentes asimetrías existentes en el consumo de memoria durante la invocación remota.

En nuestro caso se han estudiado tres asimetrías:

- (1) la servidor-cliente,
- (2) la del servidor y
- (3) la del cliente.

Cada asimetría se obtiene como cociente entre la cantidad de memoria consumida en dos fases de la invocación remota. Así pues, la asimetría servidor-cliente se calcula como cociente entre la memoria total consumida por el servidor para realizar la invocación remota y la total realizada por el cliente. De la misma manera, la del servidor es ratio entre la memoria consumida antes de que de comienzo la ejecución de la lógica de usuario del método remoto y la que se consume tras la finalización de éste. Y por último, en el cliente el ratio es entre la memoria consumida hasta el instante en el que se envían los datos al servidor y la que se consume de forma dinámica a la hora de recibir la respuesta proveniente del servidor. Por último, dado que no existen consumos negativos el rango de valores en los que se mueve el valor de la asimetría siempre se encuentra en el intervalo $[0, \infty)$.

Con los resultados obtenidos para cada una de las tres familias de métodos remotos se ha construido la figura 6.16.

Los datos que se obtienen para la asimetría servidor-cliente resultan lógicos.

Cuando el flujo de datos es poco significativo frente al coste del procesado del protocolo RTJRMP, cosa que sucede cuando se intercambia un único dato primitivo entre el cliente y el servidor, el ratio de cada una de las tres familias es el mismo: 1.5. Lo que significa que en estos casos la demanda de memoria realizada de forma dinámica para atender la petición remota en el servidor es un 50 % mayor que la realizada en el cliente.

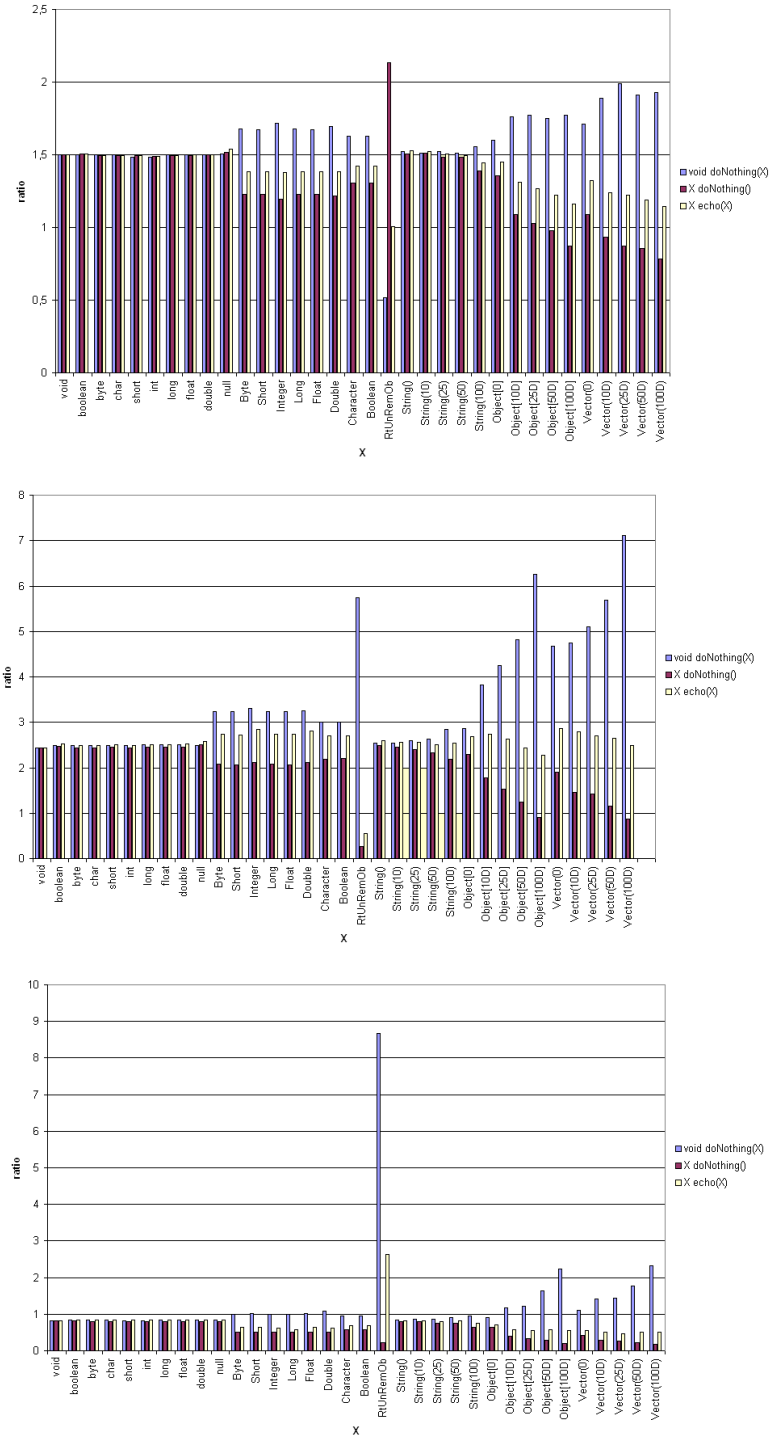


Figura 6.16: Asimetrías en el consumo de memoria servidor-cliente, en el servidor y en el cliente

Pero cuando se envían los objetos equivalentes a los datos primitivos, la influencia de la carga de datos de usuario es más significativa, provocando que cada familia de métodos remotos tenga un comportamiento diferente. En el caso de la asimetría servidor-cliente cada una de las tres familias presenta un comportamiento diferente. En la del método remoto `void doNothing(X)` el ratio servidor-cliente llega a alcanzar valores cercanos a 1,7. Y de forma contraria, cuando la utilizada es `X doNothing()`, el ratio disminuye hasta situarse en 0,7. Y ya por último, en el caso de que estemos trabajando con la familia `X echo(X)`, el ratio tiende a equilibrarse alrededor de 0,8.

El caso del envío de una referencia a un objeto remoto también merece ser estudiado con detalle. En este caso y en el método `void doNothing(X)` el consumo de memoria realizado en el cliente es mayor que el realizado en el servidor debido a que al coste de envío de la referencia remota hay que añadir el de comunicarse con el algoritmo de recolección de basura remoto. De igual forma, en el método remoto `X doNothing()` es el servidor el que tiene que comunicarse con el nodo remoto donde reside el objeto cuya referencia es intercambiada, asumiendo el coste de la recolección de basura, lo que dispara el ratio a valores superiores a 2. Y por último, cuando hay un doble trasiego de la referencia remota `-X echo(X)-`, el coste de la recolección de basura acaba enmascarando al del procesado del protocolo de comunicaciones, haciendo que el ratio se sitúe en valores cercanos a 1,0 .

Por último, cuando el flujo de datos intercambiado es suficientemente alto, el coste de procesado del protocolo de comunicaciones se ve enmascarado por el de serialización y deserialización de los datos de la aplicación. Esto provoca que cada uno de los comportamientos de las tres familias se acentúe más. Así, en el `void doNothing(X)` el consumo de memoria será mayor en el cliente, lo que hará, si el flujo de datos intercambiados es lo suficientemente alto, que el ratio de asimetría crezca con la carga intercambiada convergiendo a 2. En el caso del método `X doNothing()` el consumo de procesador subirá en el cliente pues es éste el que tendrá que asumir los costes de deserialización tendiendo el ratio a 0.7. Y por último, en el caso del método `X echo(X)` la tendencia observada es a equilibrarse siempre entorno a valores cercanos a la unidad.

En el caso de que se analicen los resultados obtenidos para la asimetría en el servidor y en el cliente los resultados obtenidos son similares a los que se han encontrado en el caso servidor-cliente. Existen dos bandas de acción, una donde lo que más pesa es el procesado del protocolo RTJRMP y otra donde predomina más la carga de datos del usuario. Lo que provoca que los índices de asimetría obtenidos alcancen valores máximos de 7,2 y mínimos de 0,25 en el caso de tratar con la asimetría en el servidor y de 8,7 y 0,2 en el caso de que se observe la asimetría en el cliente. De igual manera, al aumentar la carga de uno de los flujos en detrimento del otro, mediante las familias de métodos remotos `void doNothing(X)` o `X doNothing(void)`, el índice de asimetría tiende respectivamente a aumentar o a disminuir.

6.6.4. Eficiencia en el consumo de memoria durante la invocación remota

Otro tipo de evaluación que resulta interesante realizar es la de la eficiencia de transmisión en bytes del mecanismo de comunicaciones RMI. Esta eficiencia se puede definir como el número de bytes que son consumidos de forma dinámica tanto en el cliente como en el servidor para intercambiar un byte de información haciendo uso de un método remoto.

La figura 6.17 muestra el coste unitario en bytes para cada una de las tres familias de métodos remotos que se han venido estudiando. Cada uno de los valores mostrados se calcula como la cantidad total de memoria consumida de forma dinámica en el cliente y en el servidor dividida entre el tamaño de los datos de usuario intercambiados entre ambos. En el caso de la familia `X echo(X)` se suman los que son enviados por el cliente con los que son devueltos por el servidor. Todas las medidas realizadas están expresadas en bytes.

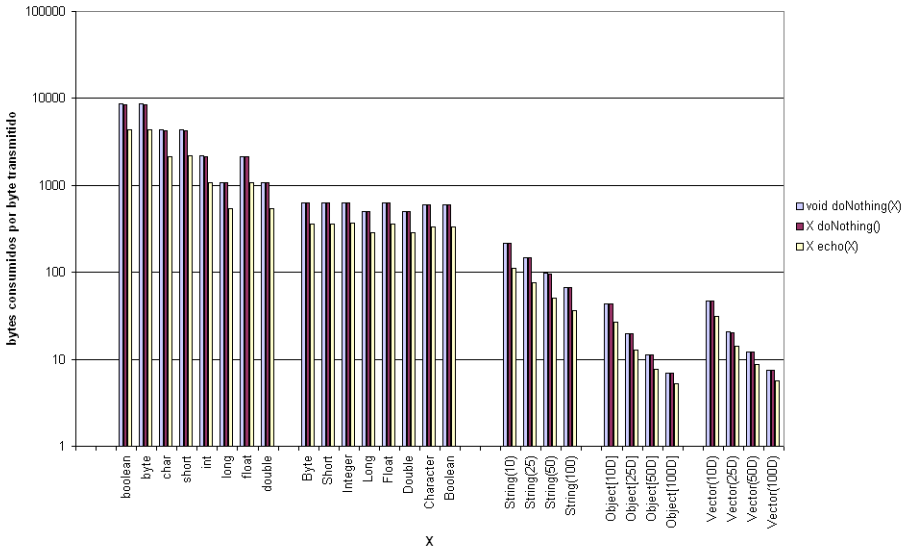


Figura 6.17: Coste unitario del envío de datos entre cliente y servidor

Los resultados nos muestran, tal y como resulta lógico, una alta ineficiencia a la hora de transmitir un único dato primitivo que es capaz de mejorar cuando el tipo de dato intercambiado se torna más complejo. Así, para un único dato primitivo se consumen cerca de 10000 bytes por byte transmitido, lo que desciende hasta ratios inferiores a 10 cuando son transmitidos 100 objetos tipo `Double` encapsulados dentro de un `java.util.Vector`.

El comportamiento de cada una de las tres familias de métodos remotos estudiadas resulta lógico. Los resultados obtenidos para `void doNothing(X)` y `X doNothing()` son los mismos pues en ambos casos la carga de datos intercambiada es la misma, no afectando el sentido (cliente-servidor o servidor-cliente) en el cual viajan los datos a la eficiencia. Y la eficiencia obtenida en el caso del método remoto `X echo(X)` es

siempre mejor que la que se puede obtener con las otras dos familias de métodos remotos pues en este caso el flujo de datos de usuario intercambiado se duplica.

6.6.5. Reflexión

En esta sección se ha estudiado el consumo de memoria durante el proceso de invocación remota, obteniéndose tanto costes totales como parciales para el consumo de memoria realizado en el conjunto de la invocación remota. Los resultados obtenidos nos han permitido analizar no sólo el coste total de la invocación remota en el cliente y en el servidor sino también hacer evaluaciones de las asimetrías que se producen en el consumo de memoria en diferentes etapas de la invocación remota, llegándose incluso a la obtención de estimadores de la mejora que en términos de consumo de memoria podría introducir el empleo de métodos asíncronos.

Este estudio se puede mejorar ampliando tanto el rango de mediciones que han sido realizadas como mediante la realización de nuevos experimentos. Así, aparte del estudio realizado se podrían incluir otras medidas orientadas a estudiar más en detalle las correlaciones existentes entre el tamaño del dato intercambiado entre el nodo cliente y el nodo servidor y la cantidad de memoria consumida de forma dinámica. También resultaría interesante el comprobar cuan extrapolables son los resultados obtenidos en el caso de utilizar un único parámetro de entrada a familias de métodos remotos donde existen múltiples parámetros de invocación, intentando establecer relaciones con los resultados ya obtenidos en el caso del contenedor `java.lang.Vector()` y el tipo de datos `java.lang.Double`.

6.7. Análisis del coste temporal de la invocación remota

Complementando el análisis del consumo de memoria, se ha realizado un análisis de las latencias que el middleware de distribución DREQUIEMI introduce en el proceso de invocación remota. El objetivo de este estudio no es tan sólo determinar el coste o la latencia mínima que es introducida por el middleware de comunicaciones en las comunicaciones extremo a extremo. El objetivo es, al igual que se hizo en el caso de la memoria, establecer relaciones más amplias donde se vea cuál es la influencia que el intercambio de un determinado tipo de dato tiene en el coste total de la invocación remota, tratando además de realizar estimaciones de lo que puede ser la asimetría cliente-servidor vs. servidor-cliente o la eficiencia que se puede obtener a la hora de intercambiar datos de usuario entre un cliente y un servidor.

Para ello, el entorno de las medidas utilizado es el entorno centralizado portátil descrito en la tabla 6.1. Esta elección tiene como principal ventaja que tanto el nodo cliente como el servidor comparten la misma escala temporal, disponiéndose de más información temporal que la existente en un entorno distribuido. A cambio de ello y como contrapartida, los resultados obtenidos tenderán a ser más pesimistas pues se perderán las ganancias del paralelismo ofertadas por el escenario distribuido.

6.7.1. Tiempos de respuesta del middleware de distribución DREQUIEMI

En primer lugar se han obtenido experimentalmente los tres tiempos de respuesta de DREQUIEMI más significativos:

- (1) el de respuesta extremo a extremo definido como el tiempo transcurrido entre el inicio y la finalización de la invocación remota en el cliente,
- (2) el de respuesta cliente-servidor definido como el tiempo transcurrido entre que el cliente inicia la invocación al sustituto y la lógica del método remoto comienza su ejecución y
- (3) el tiempo necesario en el cliente para enviar los datos al servidor.

Estos tres resultados se muestran respectivamente en las figuras 6.18, 6.19 y 6.20 y han sido medidos en ausencia de mecanismos de recolección de basura tanto en el cliente como en el servidor, utilizando exclusivamente la `ImmortalMemory`. En todos los casos las conexiones han sido establecidas de antemano, evitándose así el coste asociado a su creación. Y a fin de eliminar las interferencias introducidas por el middleware de infraestructura, cada muestra ha sido calculada como el valor mínimo de 10 ejecuciones consecutivas, lo que ayuda a eliminar el ruido de fondo introducido por la plataforma de medidas.

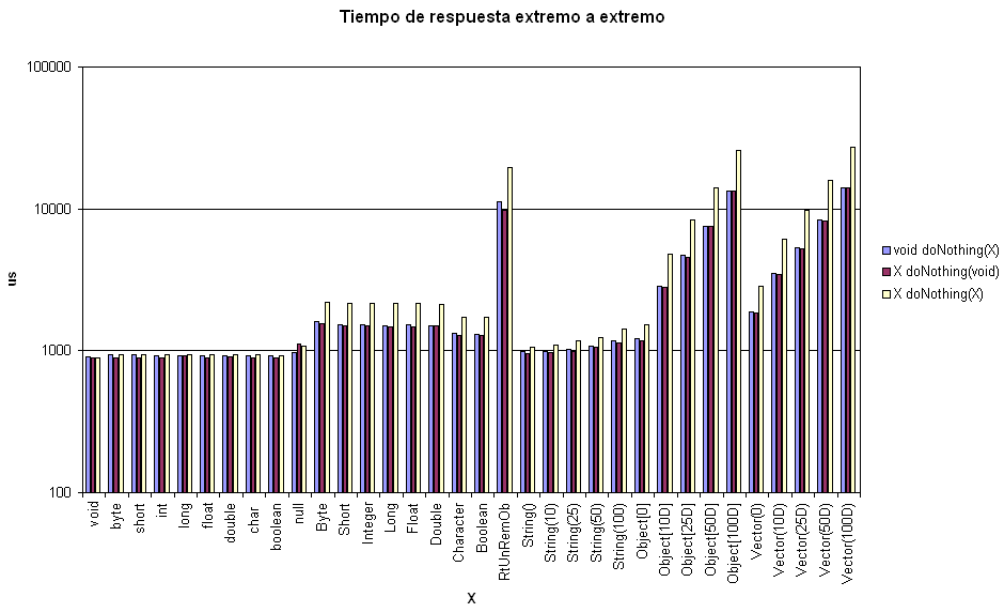


Figura 6.18: Tiempo de respuesta extremo a extremo

Los resultados obtenidos resultan lógicos, cumpliéndose que el tiempo de respuesta extremo a extremo obtenido para cada uno de los métodos de las tres familias de métodos remotos es menor que el tiempo de respuesta cliente-servidor y éste a su vez

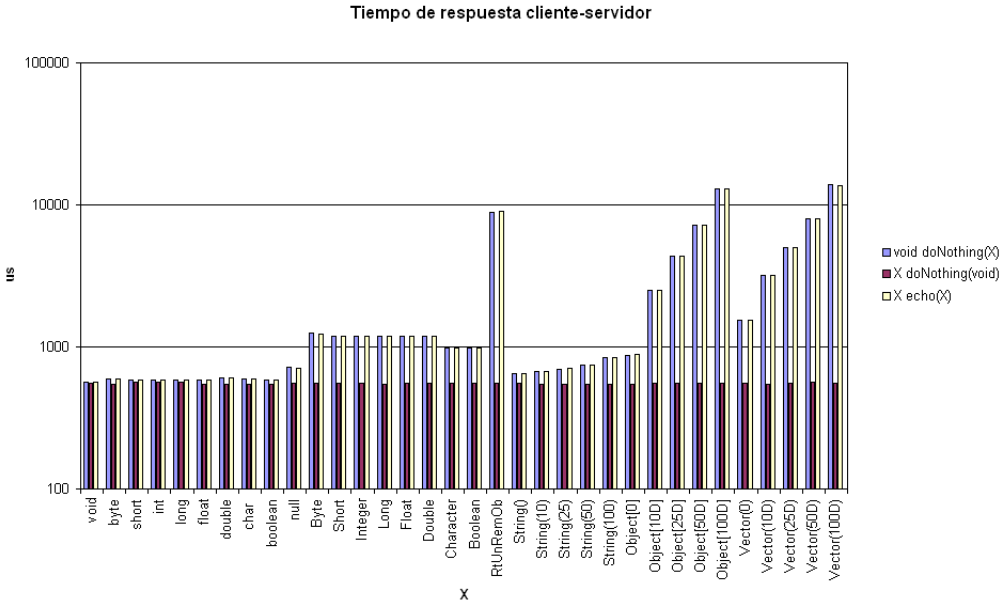


Figura 6.19: Tiempo de respuesta cliente-servidor

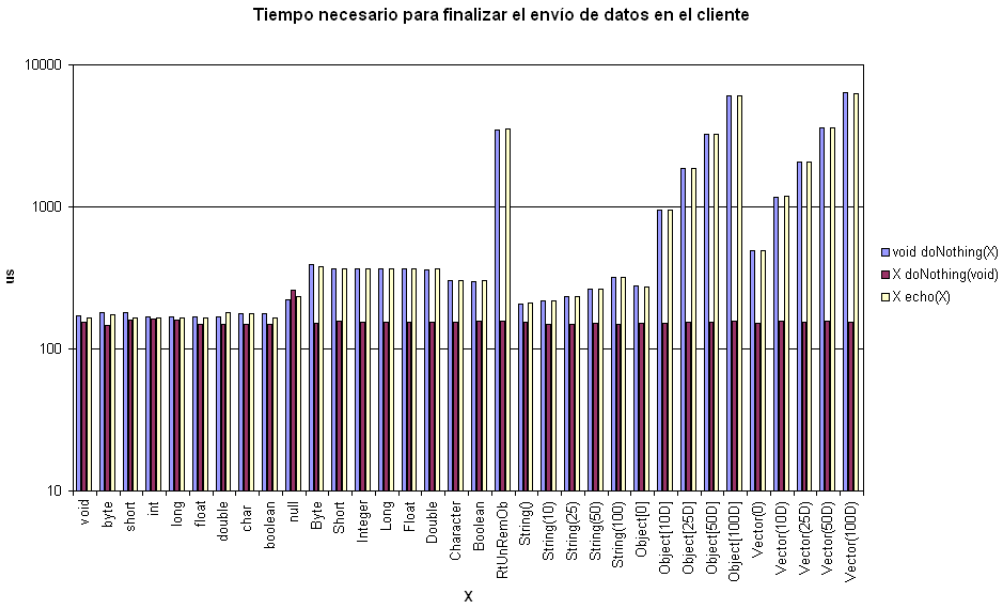


Figura 6.20: Tiempo de depósito en el cliente

es menor que el tiempo que resulta necesario para realizar el envío de datos desde el cliente al servidor.

En todos los casos se ha observado que existe un tiempo de respuesta mínimo que se corresponde con el caso en el cual no existe ningún tipo de trasiego de datos de aplicación entre los diferentes nodos de la red y donde las latencias son mayormente causadas por el procesado del protocolo RTJRMP. Este tiempo mínimo es de 906 μs para la latencia extremo a extremo, de 563 μs para la latencia cliente-servidor y de 173 μs para depositar la información que se desea hacer llegar al servidor dentro de las colas de mensajes del middleware de infraestructura.

Las tendencias observadas cuando se varían los parámetros utilizados en la invocación remota guardan ciertas similitudes con los resultados obtenidos en el análisis realizado en el caso del consumo de memoria. Así, al aumentar el flujo de datos que son enviados desde el cliente al servidor aumenta también el coste de la invocación remota. Esto se observa sobre todo en el caso en el que hay un gran trasiego de datos estructurados entre ambos. En este caso el coste se dispara de forma más o menos lineal con la cantidad de parámetros transmitidos. Así, en el peor de los casos en el que tanto el cliente como el servidor intercambian un `java.util.Vector` con 100 objetos de tipo `Double` haciendo uso de un método `echo`, el tiempo de respuesta extremo a extremo llega a superar los 27 ms. Lo que hace que el rango de tiempo de respuesta explorado por el conjunto de métodos remotos varíe desde valores ligeramente inferiores a 1 ms hasta otros superiores a los 27 ms.

También, al igual que sucedía en el caso del análisis del consumo de memoria, se observa que el coste de la transmisión de una referencia a un objeto remoto entre los diferentes nodos de la red es alto. Así, en el caso más sencillo, cuando la referencia es retornada como resultado de una invocación remota o es recibida por el servidor, el coste mínimo total de esta operación ronda los 10 ms. Esto llevado al terreno práctico recomienda, sobre todo en aplicaciones que requieran unos tiempos de respuesta extremadamente bajos, reducir al mínimo el trasiego de referencias a objetos remotos.

También se observa que existe un incremento en el coste temporal de la invocación remota que se produce cuando en vez de utilizar datos primitivos se utilizan sus equivalentes objeto. Los resultados obtenidos nos indican que el utilizar datos estructurados en la invocación remota supone un incremento en el coste total de alrededor del 60 % en el caso de la familia `void doNothing(X)`, del 70 % en la `X doNothing(void)` y del 128 % en el caso de que se utilice la `X echo(X)`. Lo que en términos prácticos nos lleva a concluir que se debe de intentar potenciar el empleo de datos primitivos cuando sea posible, ya que el tiempo de respuesta extremo a extremo mejora sustancialmente.

6.7.2. Asimetrías en las latencias introducidas por la invocación remota

Al igual que se hizo en el caso del análisis de la memoria, se ha procedido a evaluar la asimetría existente en las latencias introducidas por DREQUIEMI justo antes y después de que comience la ejecución de un método remoto. Para ello se ha calculado el tiempo transcurrido desde que el cliente comienza la invocación remota hasta que la lógica del método remoto comienza su ejecución así como el restante utilizado para retornar los resultados al cliente. Y con esos dos valores se ha calculado el ratio entre

ambos valores (el primero entre el segundo) representándose los resultados obtenidos en la figura 6.21. Valores cercanos a 1 significan que el coste de la invocación remota se reparte por igual entre el camino de ida y el de vuelta; valores inferiores a la unidad que el coste del camino de vuelta es mayor que el de ida y valores mayores que la unidad que el camino de ida es el mayor.

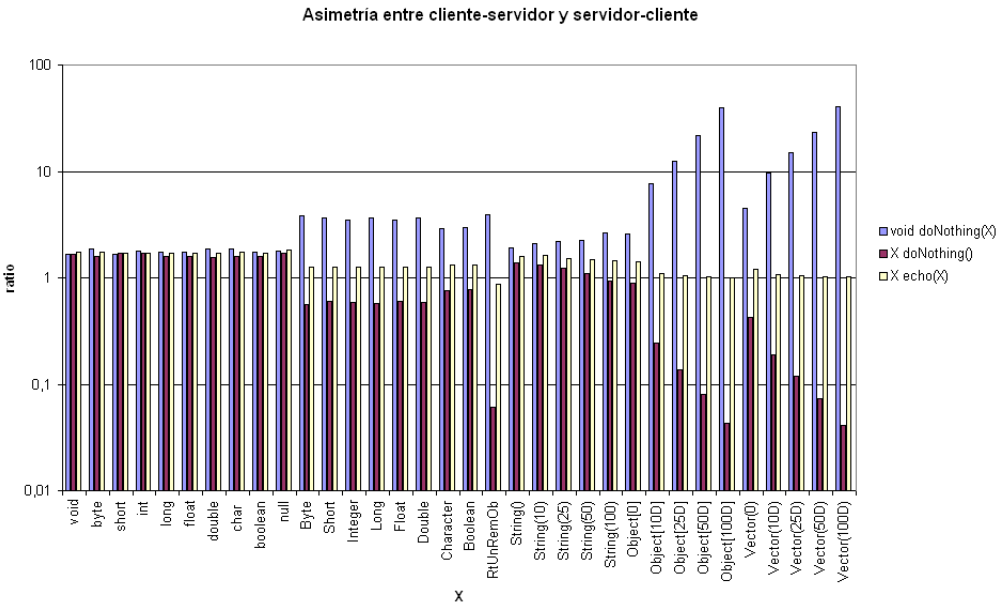


Figura 6.21: Ratio entre la latencia cliente-servidor y la servidor-cliente

A primera vista, las tendencias marcadas por las tres familias de métodos remotos resultan lógicas. En la `void doNothing(X)`, al aumentar el flujo de datos enviados aumenta, tal y como parece lógico, la asimetría a favor del flujo de ida apareciendo valores cercanos a 40. De forma complementaria cuando se aumenta el flujo de vuelta y se mantiene el de ida (ver la familia `X doNothing()`) desciende, llegándose a valores cercanos a 0,04. Y por último, cuando se trata con flujos simétricos (ver la familia `X echo(X)`) la interferencia del protocolo de comunicaciones tiende a desaparecer, apareciendo ratios cercanos a 1.

Para flujos de datos no muy pesados donde el envío de datos no introduce un gran coste adicional en la invocación remota -tipos primitivos-, el coste total tiende a valores cercanos a 1,7. Lo que significa que el coste de procesado del protocolo RTJRMP es un 70 % mayor en el tramo de ida, desde el cliente al servidor, que en el retorno, desde el servidor al cliente.

6.7.3. Estimación de las ventajas ofrecidas por el asincronismo en el cliente

Aunque el asincronismo no se encuentra soportado en el prototipo de DREQUIEMI resulta posible estimar empíricamente cuáles son las ventajas que su utilización

conlleva. Así, se ha calculado cuáles son las reducciones que en el tiempo de respuesta ofrece el asincronismo al programador a partir de los datos disponibles. Para ello se han utilizado los valores obtenidos en el caso de la respuesta extremo a extremo y los obtenidos para realizar el depósito de datos en el cliente. Los primeros estiman el coste de una invocación remota síncrona y los segundos la una asíncrona no confirmada por el servidor. Con los resultados obtenidos se ha construido la tabla 6.4 donde se reflejan las reducciones que tanto en forma porcentual como en absoluta son ofertadas por el asincronismo no confirmado por el servidor en un entorno de ejecución monoprocesador.

Así, si en vez de utilizar un método síncrono se emplea uno asíncrono, el tiempo que el cliente permanece bloqueado en el sustituto a la espera de la respuesta proveniente del servidor se puede ver reducido hasta en un 80 %. Esta reducción es menor en el caso de que aumente el tamaño del flujo intercambiado entre el cliente y el servidor. Y así, las reducciones que se pueden obtener están alrededor del 76 % en el caso de que se utilicen los objetos equivalentes a los datos primitivos y del 55 % en el caso de que se utilicen los flujos de datos más pesados.

El caso de transmisión de una referencia a objeto remoto entre dos nodos merece también un análisis en detalle. En este caso la reducción obtenida, del 21 % (1281 μs) es relativamente baja debido a que el cliente ha de esperar a que se realice la comunicación con el servicio de basura remoto correspondiente antes de desbloquear su ejecución.

Pero el resultado más importante es que estos valores justifican la inclusión de mecanismos de asincronismo dentro del propio middleware de distribución pues el bloqueo máximo experimentado por el cliente puede verse notablemente mermado. Así, en el mejor de los casos, que ocurre cuando no se envía ningún tipo de dato, se pasaría de un bloqueo de 906 μs a uno de 171 μs siendo la reducción porcentual alcanzada de un 81,12 %.

6.7.4. Impacto del establecimiento de conexiones en el coste de la invocación remota

Partiendo de los resultados para conexiones preestablecidas se ha querido determinar cuales son las ventajas, en términos de reducción del tiempo de respuesta, que implica la utilización de un esquema de tal tipo frente a otro modelo más dinámico donde en cada invocación remota es establecida una conexión.

Para ello se han vuelto a realizar las mediciones en un escenario en el que se establece una conexión antes de enviar los datos al servidor. Lo que ocasiona un incremento absoluto en el coste de la invocación remota de 2383 μs por invocación remota. En el prototipo de DREQUIEMI, este tiempo se utiliza para el establecimiento de la conexión tcp/ip, el de la conexión JRMP y la creación de la entidad concurrente en el servidor encargada de escuchar peticiones entrantes.

Tal y como puede ver de forma gráfica en la figura 6.22, el impacto porcentual es mayor en aquellos métodos remotos más rápidos que intercambian un menor número de datos. En estos casos el coste extra puede llegar al 270 %. Y en aquellos que realizan un alto intercambio de datos entre los diferentes nodos de la red esta sobrecarga

X	%	μs
void	81,12	735
byte	80,66	751
short	80,70	757
int	82,03	762
long	81,91	756
float	81,73	756
double	81,85	758
char	80,78	744
boolean	80,84	743
null	76,86	741
Byte	75,56	1206
Short	76,19	1168
Integer	76,17	1167
Long	75,92	1148
Float	76,04	1159
Double	76,04	1149
Character	77,13	1022
Boolean	77,20	1016
RtUnRemObj	21	1281
String()	78,94	776
String(10)	78,21	779
String(25)	77,19	792
String(50)	75,87	824
String(100)	72,57	847
Object[0]	77,28	939
Object[10D]	66,48	1898
Object[25D]	59,93	2801
Object[50D]	57,03	4289
Object[100D]	54,47	7299
Vector(0)	73,63	1385
Vector(10D)	66,51	2334
Vector(25D)	60,93	3239
Vector(50D)	56,78	4731
Vector(100D)	55,16	7807

Cuadro 6.4: Reducciones máximas porcentuales y absolutas en el tiempo de respuesta ofertables por el asincronismo no confirmado por el servidor a la familia de métodos remotos `void doNothing(X)` en un entorno monoprocesador

porcentual se ve reducida a valores cercanos al 18 %. Así pues, se puede concluir que en aplicaciones donde el tiempo de respuesta deba de mantenerse relativamente bajo, el empleo de conexiones preestablecidas puede ser una buena solución; mientras que en aplicaciones con tiempos de respuesta más laxos, en nuestro caso superiores a los 100 ms, se pueden establecer conexiones de forma dinámica sin que ello incremente

significativamente el coste total del proceso de la invocación remota.

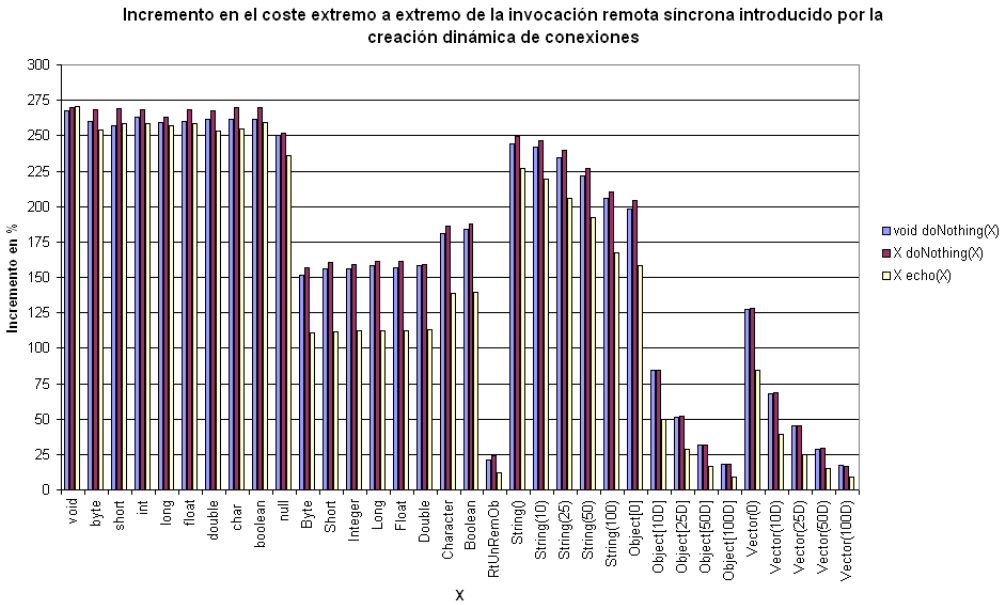


Figura 6.22: Coste extra originado por el establecimiento de conexiones dinámicas durante la invocación remota en un entorno monoprocesador

6.7.5. Sobrecarga introducida por el empleo de regiones en el servidor

Todas las medidas realizadas hasta ahora han sido obtenidas en el caso donde tanto el objeto remoto como el sustituto utilizaban `ImmortalMemory`. En esta sección se estima cuál es el incremento que introduce la utilización de un modelo de regiones en la parte del servidor en el coste total de la invocación remota.

Para ello se ha vuelto a calcular el coste total de la invocación remota en el caso de que en vez de utilizar un `ImmortalMemoryAreaPool` se utilice un `LTMemoryAreaPool` en el servidor mientras que en el cliente se continúa utilizando memoria de tipo `ImmortalMemory`. Y calculando la diferencia entre los valores obtenidos en esta experiencia y los ya obtenidos en el caso de la invocación remota extremo a extremo se ha procedido a estimar la sobrecarga introducida por el mecanismo de gestión automática de memoria basado en regiones.

Los resultados nos muestran que el utilizar el modelo de regiones en los casos estudiados no tiene un gran impacto en el coste total de la invocación remota. Porcentualmente, las mayores sobrecargas se producen cuando el flujo de datos intercambiado entre un cliente y un servidor es bajo, alcanzándose en el peor de los casos incrementos del 16 % cuando se intercambian datos primitivos. Al aumentar el flujo de datos intercambiado, utilizando por ejemplo `vector(100D)`, este coste disminuye hasta alcanzar cotas residuales del 0,1 %. Lo que significa que cuando el flujo

de datos intercambiado entre un cliente y un servidor es lo suficiente complejo, el coste asociado al envío y a la recepción de datos puede llegar a enmascarar al de su destrucción.

6.7.6. Eficiencia temporal en el intercambio de datos

En el último experimento realizado se intenta, al igual que se hizo en el caso del análisis de la memoria consumida, determinar cuál es el coste de la invocación remota por unidad de información intercambiada entre un cliente y un servidor. Pero en vez de evaluar el coste espacial se evalúa el temporal. Esto es, el tiempo necesario para intercambiar un byte de información entre un cliente y un servidor. Al igual que se hizo en el resto de las pruebas, se ha hecho la evaluación para las tres familias de métodos remotos, construyéndose con los resultados obtenidos la figura 6.23.

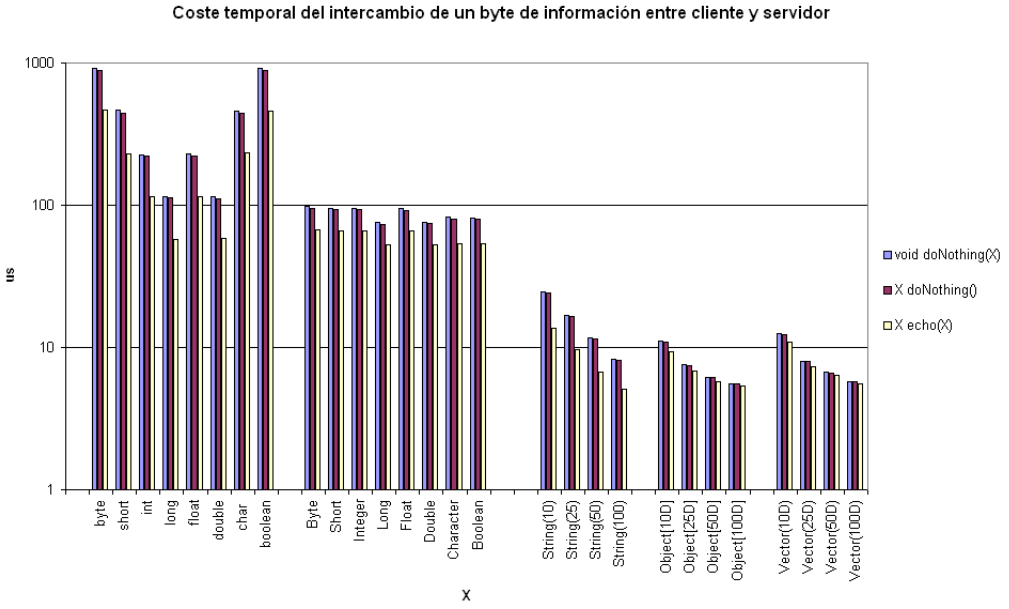


Figura 6.23: Coste temporal asociado al intercambio de un byte de información entre un cliente y un servidor

Los resultados son parecidos a los obtenidos en el caso del estudio de la eficiencia en el empleo de memoria durante el proceso de invocación remota, mostrándonos que la eficiencia aumenta cuando se envían múltiples datos entre el cliente y el servidor en una misma invocación remota. Así, cuando el flujo de datos intercambiado es mínimo, un byte, el coste se dispara hasta los 917 μs por unidad de información intercambiada. Y al hacer que aumente, el coste disminuye, llegando a valores cercanos a los 5,6 μs por byte de información transmitido. Lo que confirma las tendencias observadas en el caso de la memoria, a mayor número de datos intercambiados, mayor eficiencia en su transmisión.

6.7.7. Reflexión

De forma paralela a lo hecho en el estudio sobre el consumo de memoria durante la invocación remota en DREQUIEMI, en esta sección se ha evaluado la latencia temporal introducida por el middleware de distribución. Para ello se han tomado las mismas familias de métodos remotos que habían sido utilizadas en el análisis de memoria y se han realizado pruebas similares.

Los resultados obtenidos justifican varias de las decisiones de diseño que se habían tomado en el modelo de DREQUIEMI. Así, la inclusión de mecanismos que permitan realizar invocaciones asíncronas, gestionar las conexiones o emplear regiones encuentran aquí una buena justificación. El asincronismo, capaz de reducir las latencias que experimenta el cliente hasta en un 80 %; el uso de conexiones preestablecidas, capaces de eliminar la sobrecarga asociada al establecimiento de una conexión (que puede ver incrementado su coste en hasta un 270 %) y el empleo de regiones que introduzcan una baja sobrecarga en la invocación remota (alrededor del 16 %), se nos muestran como técnicas capaces de influir significativamente en el coste total de la invocación remota.

A medio plazo el presente experimento se podría mejorar mediante la obtención de resultados para entornos distribuidos, repitiendo los experimentos realizados en un sistema monoprocesador en uno distribuido. Esto nos permitiría obtener correlaciones entre el tiempo de respuesta de un sistema centralizado y otro distribuido equivalente. Nuestros experimentos preliminares con `SharedRemoteObject` nos mostraban ganancias de 400 μs en el tiempo de respuesta mínimo extremo a extremo pero desconocemos si estos resultados son generalizables o no a otros conjuntos de datos.

Como principal línea futura a explorar se puede apuntar la de ser capaces de medir el consumo de procesador tanto en el cliente como en el servidor. Las mediciones realizadas, al contrario de las obtenidas cuando se evaluaba el consumo de memoria, no nos permiten calcular la cantidad de procesador que consume cada una de las hebras del sistema, sino que nos proporcionan una información más pobre. La realización de estas mediciones nos permitiría conseguir más información sobre el comportamiento interno del middleware de distribución así como estudiar diferentes relaciones cruzadas existentes entre el consumo de memoria y de procesador dentro del contexto de la invocación remota.

6.8. Conclusiones y líneas futuras

Este capítulo ha tenido por objeto el comprobar empíricamente que las técnicas descritas a lo largo de esta tesis dentro del contexto de DREQUIEMI pueden llegar a afectar al tiempo de respuesta de las aplicaciones distribuidas significativamente. Los resultados nos muestran que el empleo de sistemas de gestión de procesador basados en prioridades pueden reducir notablemente la inversión de prioridad experimentada por las tareas de mayor prioridad cuando hay otras de menor nivel de prioridad intentando acceder a un objeto remoto de tiempo real. Y también nos muestran que la técnica de regiones asociada al `LTMemoryAreaPool` es capaz de eliminar eficientemen-

te (introduciendo incrementos en el coste total de la invocación remota que varían entre el 16 % y el 0,1 %) los objetos creados durante el proceso de invocación remota en el servidor. El estudio realizado sobre el consumo de recursos nos ha mostrado que existe un coste mínimo y fijo en términos de memoria y latencia que en el prototipo desarrollado y en un procesador 796 Mhz ronda los 8 kb y el 1 ms. Destaca también la fuerte influencia que el utilizar conexiones ya establecidas, capaces de reducir en el mejor de los casos a la cuarta parte el coste total de la invocación remota, puede llegar a tener en el coste total de la invocación remota. Por último, los resultados obtenidos para el asincronismo, capaces de reducir hasta en un 81 % el tiempo que permanece bloqueado un cliente, justifican su inclusión en el modelo computacional desarrollado.

Tal y como ya se había mencionado estos valores constituyen una cota superior de los que se deberían de encontrar en plataformas altamente eficientes. Entornos donde las librerías de serialización estén más optimizadas y donde se utilicen técnicas de compilación de bytecodes deberían de llevarnos a la obtención de tiempos de respuesta y consumo de memoria inferiores a los obtenidos en este capítulo.

Una de las posibles líneas futuras a abordar a corto plazo es la de comparar RTRMI-RTSJ con RTCORBA-RTSJ. Para ello se compararán tanto el prototipo RTZen como el de DREQUIEMI en los diferentes casos de estudio presentados en este capítulo.

Capítulo 7

Conclusiones y líneas futuras

Ante el imparable crecimiento, tanto en número como en complejidad, de los sistemas distribuidos de tiempo real esta tesis ha abordado este reto intentando producir un middleware de distribución de tiempo real para el modelo proporcionado por RMI. Para ello, tomando como punto de partida modelos de distribución ya pre-existentes de propósito general como RMI, aplicando las técnicas presentes en otras soluciones de tiempo real como RTCORBA y solventando algunos problemas específicos relacionados en su mayor parte con la gestión automática de memoria, se ha construido un modelo de computación para RTRMI denominado DREQUIEMI. Este modelo permite realizar un cierto control sobre diferentes recursos involucrados en el proceso de comunicación remota tales como son el procesador, la memoria y las comunicaciones de bajo nivel, facilitando la obtención de cotas máximas sobre los tiempos de respuesta extremo a extremo de las diferentes aplicaciones distribuidas Java. Complementando este modelo se han propuesto tres extensiones a RTSJ. La primera de ellas que permite la recolección de basura flotante dentro de una región. La segunda de ellas que permite el establecimiento de referencias normalmente prohibidas por las reglas del padre único y de asignación. Y por último, una tercera que unifica el modelo de planificación de Java de tiempo real en torno a una única entidad concurrente.

Desconocemos si las tecnologías Java de tiempo real distribuido, y por extensión también las centralizadas, finalmente serán acogidas de buen grado por la comunidad de desarrolladores de tiempo real. Pero sin embargo creemos viable tecnológicamente una solución de tipo RTRMI. Los resultados de esta tesis nos muestran que resulta posible definir un modelo de gestión de recursos interno para RMI, de tal manera que se pueda saber el cómo las tareas hacen uso de los recursos internamente durante una invocación remota, así como cuáles son las interferencias que pueden introducir tanto el recolector de basura como el servicios de nombres. También nos enseñan que es posible definir interfaces para RMI que permitan realizar el control de bajo nivel sobre los diferentes recursos del sistema involucrados, de tal manera que éstas se encuentren altamente alineadas con las actualmente presentes en RMI y en RTSJ. Y por último, también nos muestran, esta vez empíricamente, que la gestión realizada en recursos clave como son el procesador, la memoria y la comunicación de bajo nivel puede, al igual que en otros sistemas distribuidos de tiempo real, repercutir significa-

tivamente en los tiempos de respuesta experimentados por las diferentes aplicaciones distribuidas Java.

La sección 7.1 muestra las contribuciones especificas realizadas por esta tesis al conjunto de las tecnologías Java de tiempo real y la sección 7.2 nos muestra aquellas tareas que habiendo sido identificadas, aún precisan ser estudiadas en mayor profundidad.

7.1. Principales contribuciones

En grandes líneas, las contribuciones realizadas por esta tesis se pueden encuadrar en cuatro grupos:

1. *Estudio del estado del arte relativo a las tecnologías Java de tiempo real*

   Este estudio se ha centrado en identificar cuál es estado actual de Java tanto en sistemas centralizados como en distribuidos. El principal resultado que se ha obtenido es la existencia de una gran carencia, en la actualidad, de soluciones que nos permitan desarrollar sistemas distribuidos de tiempo real empleando como base las tecnologías Java. Lo que justifica la realización de esta tesis.

2. *Desarrollo de un modelo de distribución de tiempo real basado en RMI*

   Como primer paso encaminado a la obtención de una solución se ha construido un modelo de gestión de recursos que considera aspectos generales de RMI. Este modelo aunque es parcialmente independiente de la tecnología de objetos remotos distribuida que se utilice para su implementación está pensado para ser aplicado a RMI, mejorando al actual RMI con la posibilidad de que se realicen invocaciones remotas asíncronas y con la caracterización del comportamiento interno de las invocaciones remotas y los servicios de recolección distribuida de basura y de nombres.

3. *Desarrollo de un sistema de extensiones para RTRMI*

   Tomando como punto de partida ese modelo se ha construido un sistema de interfaces para RMI de tiempo real denominado DREQUIEMI. Estas interfaces permiten al programador desarrollar sistemas de tiempo real haciendo uso del paradigma de distribución RMI y del lenguaje de programación Java de tiempo real RTSJ. El principal aporte realizado por este tipo de interfaces es el de caracterizar un modelo con tres niveles de clases capaz de ofrecer un alto grado de reconfigurabilidad. Este modelo incorpora además un protocolo de comunicaciones de tiempo real de tipo JRMP.

4. *Desarrollo de extensiones a Java de tiempo real centralizado*

   En el propio proceso de definición del modelo de computación para RTRMI se han identificado una serie de mejoras que pueden ser introducidas en RTSJ. Estas extensiones facilitan el desarrollo de sistemas de tiempo real tanto centralizados como distribuidos y de forma conjunta intentan mitigar ciertas debilidades observadas en el modelo de regiones de RTSJ. Tres han sido las extensiones propuestas. La `AGCMemory`, capaz de eliminar la basura flotante; el

`ExtendedPortal`, capaz de establecer referencias que usualmente se encuentran prohibidas por la regla de asignación y el `RealtimeThread++` que simplifica el modelo actual de entidades concurrentes de RTSJ.

5. *Obtención de resultados empíricos sobre el comportamiento de RTRMI*

   Por último se han obtenido resultados empíricos que corroboran de forma práctica que:

   - el empleo de esquemas de prioridades extremo a extremo es capaz de reducir la inversión de prioridad que experimentan los clientes de mayor prioridad de forma significativa.
   - el empleo de regiones puede reducir drásticamente el tiempo de respuesta extremo a extremo de una aplicación distribuida, eliminando la dependencia para con el recolector de basura de forma eficiente.
   - el control del establecimiento de la conexión puede reducir notablemente el tiempo de repuesta de algunas aplicaciones distribuidas y que por tanto es una característica que ha de poder ser controlada por el programador.
   - la introducción de algún tipo de mecanismo de asincronismo en la invocación remota permite reducir el consumo dinámico de memoria en el cliente así como el tiempo que éste permanece bloqueado notablemente, convirtiéndolo en otro mecanismo de interés para aproximaciones de tipo RTRMI.

De entre todas la propuestas en esta tesis para Java de tiempo real distribuido, una de las más de mayor grado de originalidad es quizás la creación de un modelo de gestión de memoria para el objeto remoto basado en el concepto de *memorypool*. El resto de abstracciones que dan soporte a la predictibilidad extremo a extremo ya se encontraban de alguna manera presentes en el modelo de RTCORBA, capaz de controlar tanto el establecimiento de las conexiones como la gestión del procesador en cada uno de los nodos del sistema. La separación en *contexto de creación* y *de invocación* aporta una solución al problema de cómo integrar el modelo de regiones de RTSJ dentro del modelo computacional de RMI, permitiendo esquivar la interferencia del recolector de basura durante la invocación remota.

Las tres extensiones propuestas para Java de tiempo real centralizado también gozan de un cierto grado de innovación pues facilitan el desarrollo de aplicaciones de tiempo real ofreciendo un modelo de regiones más flexible. De las tres descritas, quizás una de las más de mayor alcance sea la que propone reunificar el modelo de hilo de tiempo real en una única entidad concurrente capaz de establecer de forma dinámica la relación deseada con el recolector de basura. En el caso del modelo de referencia extendida esta funcionalidad se puede obtener, aunque a costa de utilizar un mayor número de tablas e hilos auxiliares, haciendo uso del mecanismo de portales del actual RTSJ. Y por último, en el caso del modelo de regiones con capacidad de recolección de basura flotante, el uso de la técnica de regiones anidadas es otra alternativa también válida.

7.2. Líneas futuras

Tras haber realizado esta tesis se han detectado una serie de líneas futuras de trabajo cuya exploración resultaría altamente interesante. En algunos casos tan sólo se ha identificado su necesidad mientras que en otras se ha dado ya algún paso encaminado a su consecución. A continuación, se enumerarán, una a una, describiendo a grandes trazos los objetivos perseguidos en cada una de ellas.

1. *Implementación de modelos para la invocación remota asíncrona*

   Durante el último tramo de esta tesis cuando se realizaba una evaluación empírica del modelo DREQUIEMI se han realizado estimaciones de cuál podría ser el impacto del asincronismo sobre la invocación remota, llegándose a la conclusión de que las potenciales reducciones que el cliente podría experimentar en el consumo tanto de memoria como en el tiempo que permanece bloqueado podrían ser elevadas. Pero sin embargo, no se llegó a validar empíricamente, sino que se estimó de forma experimental. En esta línea, el trabajo a realizar consistiría en implementar eficientemente el modelo de asincronismo propuesto, realizando mediciones más exactas que corroborasen la validez de las estimaciones hechas.

2. *Desarrollo de herramientas que ayuden a simplificar el desarrollo de sistemas distribuidos de tiempo real con DREQUIEMI*

   El modelo de gestión de recursos para sistemas distribuidos desarrollado impone al programador RMI la tarea de configurar adecuadamente el middleware de distribución. Así, el programador de DREQUIEMI ha de tener, al igual que el programador RTSJ tradicional, cierto conocimiento sobre las características temporales de la aplicación distribuida, de tal manera que sea capaz de dimensionar y configurar adecuadamente los recursos (procesador, memoria y conexión) que ésta utiliza. Este tipo de tarea se vería simplificada enormemente si existiesen herramientas capaces de realizar estas tareas de forma más automática a partir de la especificación de requisitos del sistema distribuido, utilizando para ello modelos como el de planificación de UML (Unified Modelling Language) [161]. Por tanto, otra línea de trabajo iría en esa dirección, en la de proveer herramientas que de forma automática permitiesen desplegar una aplicación distribuida en DREQUIEMI.

3. *Adaptación de otras abstracciones de orden superior: el servicio de descubrimiento y componentes de tiempo real*

   Moviéndonos en las abstracciones de mayor nivel nos encontramos con otras dos líneas a explorar. La primera de ellas iría encaminada hacia la obtención, dentro del marco de computación ofrecido por DREQUIEMI, de nuevas tecnologías de tiempo real como podrían ser una posible tecnología de tiempo real basada en la aplicación de las técnicas de tiempo real existentes en la actualidad al modelo de componentes de la tecnología *Enterprise Java Beans*(EBJ),

dando lugar a una especie de RT-EJB. La segunda estaría más orientada a la provisión de mecanismos de descubrimiento cuyo comportamiento fuese predecible, transformado por ejemplo la actual tecnología JINI en una especie de RT-JINI. En esta última línea de trabajo y dentro del grupo de tiempo real, ya se han realizado algunos esfuerzos (ver [63] y [55]).

4. *Definición de un servicio de sincronización de tiempo real basado en el paradigma FTT*

   Por último, durante la estancia realizada por el doctorando en la Universidad de Aveiro (Portugal) en el año 2006, se ha identificado otra nueva línea de trabajo: la introducción del conjunto de técnicas FTT [138] en el modelo de distribución de DREQUIEMI. Desde el punto de vista de DREQUIEMI la inclusión de este tipo de técnicas enriquece el modelo desarrollado con técnicas asíncronas de tipo *publisher-subscriber* que previamente no habían sido consideradas. Algunos de los resultados obtenidos de la exploración de esta línea pueden ser consultados en [15].

5. *Incorporación de algoritmos de planificación distribuida en DREQUIEMI*

   El trabajo realizado en DREQUIEMI se ha orientado sobretodo a la definición y a la validación empírica de técnicas que fuesen capaces de influir en los tiempos de respuesta de las aplicaciones distribuidas, dejando más de lado cómo derivar a partir de los requisitos de una aplicación distribuida un esquema de planificación capaz de satisfacerlos adecuadamente y de una forma óptima. En este sentido, sería muy interesante incorporar diferentes técnicas descritas en el estado del arte que abordan este problema, definiendo para ello nuevos planificadores distribuidos en la jerarquía actual de DREQUIEMI.

Bibliografía

[1] *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 2001.

[2] *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2001.

[3] *Real-Time Core Extensions*, 2001. Available on-line at http://www.j-consortium.org.

[4] *The Real-Time Specification for Java*. Adisson-Wesley, 2001.

[5] *Real-Time Systems and Programming Languages*. Addison-Wesley, 2001.

[6] High integrity java, 2005. Available on-line at http://www.hija.info.

[7] *Mackinac white paper*, 2005. Available on-line at http://research.sun.com/projects/mackinac/mackinacwhitepaper.pdf.

[8] Douglas C. Shmidt Alexander B. Arulanthu, Carlos O'Ryan and Michael Kircher. Applying c++, patterns, and componets to develop an idl compiler for corba ami callbacks. *C++ Report*, 12(3), 2000.

[9] James H. Anderson, Rohit Jain, and Srikanth Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 111–122, 1997.

[10] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.

[11] Jonathan S. Anderson and E. Douglas Jensen.

[12] Apogee. Aphelion, 2004. Available at http://www.apogee.com/aphelion.html.

[13] David F. Bacon, Perry Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *OTM Workshops*, pages 466–478, 2003.

[14] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and ada. *Real-Time Systems*, 1(1):7–25, 1989.

[15] Pablo Basanta-Val, Luis Almeida, Marisol García-Vals, and Iria Estévez Ayres. Towards a synchronous schedulling service on top of an unicast distributed real-time java. 2007. Submitted to RTAS 07: IEEE Real-time Application Symposium.

[16] Pablo Basanta-Val, Marisol García-Valls, and Iria Estévez-Ayres. Agcmemory: A new real-time java region type for automatic floating garbage recycling. *ACM SIGBED*, 2(3), July 2005.

[17] Pablo Basanta-Val, Marisol García-Valls, and Iria Estévez-Ayres. Enhancing the region model of real-time java por large-scale systems. In *2nd Workshop on High Performance, Fault Adaptative, Large Scale Embedded Real-Time Systems*, May 2005.

[18] Pablo Basanta-Val, Marisol García-Valls, and Iria Estévez-Ayres. Extended-portal: violating the assignment rule and enforcing the single parent one. In *4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, October 2006.

[19] Pablo Basanta-Val, Marisol García-Valls, and Iria Estévez-Ayres. No heap remote objects: Leaving out garbage collection at the server side. In *OTM Workshops*, pages 359–370, 2004.

[20] Pablo Basanta-Val, Marisol Garcia-Valls, and Iria Estevez-Ayres. Towards the integration of scoped memory in distributed real-time java. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 382–389, Washington, DC, USA, 2005. IEEE Computer Society.

[21] William S. Beebee and Martin C. Rinard. An implementation of scoped memory for real-time java. In *EMSOFT*, pages 289–305, 2001.

[22] Edward G. Benowitz and Albert F. Niessner. A patterns catalog for rtsj software designs. In *OTM Workshops*, pages 497–507, 2003.

[23] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[24] B. Guider R. Lizzi C. Parain F. Bollella, G. Delsart. Mackinac: Macking hotspot real-time. In *Eighth IEEE International Symposium on,*, pages 45–54, 2005.

[25] Greg Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[26] Gregory Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with non-heap memory in the real time specification for java. In *OOPSLA Companion*, pages 361–369, 2003.

[27] Gregory Bollella, Krystal Loh, Graham McKendry, and Thomas Wozenilek. Experiences and benchmarking with jtime. In *OTM Workshops*, pages 534–549, 2003.

[28] Gregory Bollella and Kirk Reinholtz. Scoped memory. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 23–25, 2002.

[29] A. Borg. A real-time rmi framework for the rtsj, 2003. Available from: http://www.cs.york.ac.uk/ftpdir/reports/.

[30] Andrew Borg and Andy J. Wellings. A real-time rmi framework for the rtsj. In *ECRTS*, pages 238–246, 2003.

[31] Andrew Borg and Andy J. Wellings. Reference objects for rtsj memory areas. In *OTM Workshops*, pages 397–410, 2003.

[32] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusett Institute of Technology, 2004.

[33] Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin C. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI*, pages 324–337, 2003.

[34] N. Brown and C. Kindel. Distributed component object model protocol dcom/1.0, 1998.

[35] Kevin Bryan, Lisa Cingiser DiPippo, Victor Fay Wolfe, Matthew Murphy, Jiangyin Zhang, Douglas Niehaus, David Fleeman, David W. Juedes, Chang Liu, Lonnie R. Welch, and Christopher D. Gill. Integrated corba scheduling and resource management for distributed real-time embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 375–384, 2005.

[36] Alan Burns, Brian Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Ada-Europe*, pages 263–275, 1998.

[37] Alan Burns and Andy J. Wellings. Processing group parameters in the real-time specification for java. In *OTM Workshops*, pages 360–370, 2003.

[38] Juan López Campos, J. Javier Gutiérrez, and Michael González Harbour. The chance for ada to support distribution and real-time in embedded systems. In *Ada-Europe*, pages 91–105, 2004.

[39] Byung-Kyu Choi, Sangig Rho, and Riccardo Bettati. Dynamic resource discovery for applications survivability in distributed real-time systems. In *IPDPS*, page 122, 2003.

[40] Byung-Kyu Choi, Sangig Rho, and Riccardo Bettati. Fast software component migration for applications survivability in distributed real-time systems. In *ISORC*, pages 269–276, 2004.

[41] Portable Applications Standards Committee. *POSIX Realtime and Embedded application Support*. IEEE Standard for Information Technology, 2003.

[42] Daniel E. Cooke. Nasa's exploration agenda and capability engineering. *Computer*, 29(1):63–73, January 2006.

[43] G. Cooper, L. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thurasignham, S. Wohlever, and V. Wolfe. Real-time corba development at mitre, nrad, tripacific and uri, 1997.

[44] Angelo Corsaro. Jrate. Web, 2004. Available at http://jrate.sourceforge.net/.

[45] Angelo Corsaro and Ron Cytron. Efficient memory-reference checks for real-time java. In *LCTES*, pages 51–58, 2003.

[46] Angelo Corsaro and Corrado Santoro. Design patterns for rtsj application development. In *OTM Workshops*, pages 394–405, 2004.

[47] Angelo Corsaro and Douglas C. Schmidt. The design and performance of the jrate real-time java implementation. In *CoopIS/DOA/ODBASE*, pages 900–921, 2002.

[48] Box D. *Essential COM.* Addison-Wesley, 1997.

[49] Miguel A. de Miguel. Solutions to make java-rmi time predictable. In *ISORC*, pages 379–386, 2001.

[50] Morgan Deters and Ron Cytron. Automated discovery of scoped memory regions for real-time java. In *MSP/ISMM*, pages 132–142, 2002.

[51] DIAPM. Rtai. Web, 2006. Available at http://www.rtai.org.

[52] Peter Dibble. Non allocating methods. Technical report, 2005. http://www.rtsj.org/docs/allocatingMethods/allocatingMethods1.html.

[53] Petter C. Dibble. *Real-Time Java Platform Programming.* Java Series, 2002.

[54] Daniel Dvorak, Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Project golden gate: Towards real-time java in space missions. In *ISORC*, pages 15–22, 2004.

[55] Iria Estévez-Ayres, Marisol García-Valls, and Pablo Basanta-Val. Static composition of service-based real-time applications. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 11–15, May 2005.

[56] Holmes D. et al. The ovm project. Web, 2004. Available at http://www.ovmj.org/.

[57] D. Fauth, J. Gossels, D. Hartman, B. Johnson, R. Kumar, N. Lesser, D. Lounsbury, D. Mackey, C. Shue, T. Smith, J. Steiner, and W. Tuvell. Osf distributed computing environment overview. Technical report, Open Software Foundation, Cambridge, MA, 1990.

[58] Victor Fay-Wolfe, Lisa C. DiPippo, Gregory Copper, Russell Johnston, Peter Kortmann, and Bhavani Thuraisingham. Real-time corba. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1073–1089, 2000.

[59] Shahrooz Feizabadi, William S. Beebee, Binoy Ravindran, Peng Li, and Martin C. Rinard. Utilitiy accrual scheduling with real-time java. In *OTM Workshops*, pages 550–563, 2003.

[60] David Flanagan. *Java Foundation Classes*. O'Reilly, 1999.

[61] ATM Forum. Atm user-network interface specification version 3.1, 1994.

[62] Ian T. Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, 2002.

[63] Marisol García-Valls, Iria Estévez-Ayres, Pablo Basanta-Val, and Carlos Delgado-Kloos. Cosert: A framework for composing service-based real-time applications. In *Business Process Management Workshops 2005*, pages 329–341, October 2005.

[64] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. pages 205–219, 1989.

[65] Richard-Foy M. Gauthier L. Expresso: Real-time java for safety and mission critical embedded systems. Technical report, IRISIA, 2003. Available on-line at: http://www.irisa.fr/rntl-expresso/.

[66] David Gay and Bjarne Steensgaard. Stack allocating objects in java. Technical report, Microsoft Technical Report, November 1998.

[67] Victor Giddings. Recommendations for a corba language mapping for rtsj, 2005. Available on-line at www.omg.org/news/meetings/workshops/RT_2005/04-3_Giddings.pdf.

[68] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time corba scheduling service. *Real-Time Syst.*, 20(2):117–154, 2001.

[69] Urs Gleim. Jarts: A portable implementation of real-time core extensions for java. In *Java Virtual Machine Research and Technology Symposium*, pages 139–149, 2002.

[70] Aniruddha Gokhale and Douglas C. Schmidt. Optimizing a corba iiop protocol engine for minimal footprint multimedia systems. *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 17(9), 1999.

[71] Aniruddha S. Gokhale and Balachandran Natarajan. Grit: A corba-based grid middleware architecture. In *HICSS*, page 319, 2003.

[72] J. C. Palencia Gutiérrez and Michael González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symposium*, pages 26–, 1998.

[73] Johannes Helander. Deeply embedded xml communication: towards an interoperable and seamless world. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 62–67, New York, NY, USA, 2005. ACM Press.

[74] Johannes Helander and Stefan Sigurdsson. Self-tuning planned actions time to make real-time soap real. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 80–89, Washington, DC, USA, 2005. IEEE Computer Society.

[75] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD thesis, Lund Institute of Technology, 1998.

[76] M. Teresa Higuera-Toledano. Towards an understanding of the behavior of the single parent rule in the rtsj scoped memory model. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 470–479, 2005.

[77] M. Teresa Higuera-Toledano and Valérie Issarny. Java embedded real-time systems: An overview of existing solutions. In *ISORC*, pages 392–391, 2000.

[78] M. Teresa Higuera-Toledano, Valérie Issarny, Michel Banâtre, Gilbert Cabillic, Jean-Philippe Lesot, and Frédéric Parain. Region-based memory management for real-time java. In *ISORC*, pages 387–394, 2001.

[79] M. Teresa Higuera-Toledano, Valérie Issarny, Michel Banâtre, and Frédéric Parain. Memory management for real-time java: An efficient solution using hardware support. *Real-Time Systems*, 26(1):63–87, 2004.

[80] Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers. Communicating Java Threads. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, pages 48–76, University of Twente, Netherlands, 1997. IOS Press, Netherlands.

[81] Gerald H. Hilderink, Andry W. P. Bakkers, and Jan F. Broenink. A distributed real-time java system based on csp. In *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 400, Washington, DC, USA, 2000. IEEE Computer Society.

[82] Hoare. *Communicating Sequential Process.* Prentice Hall Internatioal Series in Computer Science, 1985.

[83] Erik Yu-Shing Hu, Andy J. Wellings, and Guillem Bernat. Gain time reclaiming in high performance real-time java systems. In *ISORC*, pages 249–256, 2003.

[84] IETF. A high-level framework for network-based resource sharing. RFC 707, 1975.

[85] IETF. Specification of guaranteed quality of service. RFC 2212, 1997.

[86] ITU-T/ISO. Reference model for open distributed processing, parts 1,2,3 itu-t x.901-x.904—iso/iec is 10746-(1,2,3)., 1995.

[87] Nader Mohamed Jameela Al-Jaroodi. Object-reuse for more predictable real-time java behavior. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 398–401, Washington, DC, USA, 2005. IEEE Computer Society.

[88] Guy Steele Jamew Gosling, Bill Joy and Gilad Bracha. *The Java language Specification Second Edition*. Java Series. Adddison-Wesley,Boston, Mass.., 2004.

[89] E. Douglas Jensen. A proposed initial approach to distributed real-time java. In *ISORC*, pages 2–6, 2000.

[90] Sixto Ortiz Jr. The battle over real-time java. *Computer*, 32(6):13–15, 1999.

[91] JSR-1. Real-time specification for java. Java Community Process, June 2000. Available at http://www.jcp.org/en/jsr/detail?id=1.

[92] JSR-282. Rtsj version 1.1. Java Community Process, October 2005. Available at http://www.jcp.org/en/jsr/detail?id=282.

[93] JSR-50. Distributed real-time specification, 2000. Available on-line at http://www.jcp.org/en/jsr/detail?id=50.

[94] JSR-66. J2me optional package specification 1.0: Final release. Java Community Process, June 2002. Available at http://www.jcp.org/aboutJava/communityprocess/final/jsr066/.

[95] Dall S. K and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 1(26):127–140, 1978.

[96] Yvon Kermarrec. Corba vs. ada 95 dsa: a programmer's view. In *SIGAda*, pages 39–46, 1999.

[97] Raymond Klefstad, Arvind S. Krishna, and Douglas C. Schmidt. Design and performance of a modular portable object adapter for distributed, real-time, and embedded corba applications. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 549–567, London, UK, 2002. Springer-Verlag.

[98] Raymond Klefstad, Sumita Rao, and Douglas C. Schmidt. Design and performance of a dynamically configurable, messaging protocols framework for real-time corba. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 320.1, Washington, DC, USA, 2003. IEEE Computer Society.

[99] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan. Towards highly configurable real-time object request brokers. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 437–447, 2002.

[100] Arvind S. Krishna, Raymond Klefstad, Douglas C. Schmidt, and Angelo Corsaro. Towards predictable real-time java object request brokers. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 49, Washington, DC, USA, 2003. IEEE Computer Society.

[101] Arvind S. Krishna, Douglas C. Schmidt, and Raymond Klefstad. Enhancing real-time corba via real-time java features. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 66–73, Washington, DC, USA, 2004. IEEE Computer Society.

[102] Arvind S. Krishna, Douglas C. Schmidt, Krishna Raman, and Raymond Klefstad. Enhancing real-time corba predictability and performance. In *CoopIS/DOA/ODBASE*, pages 1092–1109, 2003.

[103] Yamuna Krishnamurthy, Irfan Pyarali, Christopher D. Gill, Louis Mgeta, Yuanfang Zhang, Stephen Torri, and Douglas C. Schmidt. The design and implementation of real-time corba 2.0: Dynamic scheduling in tao. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 121–129, 2004.

[104] Ravi Krishnan. Future of embedded systems technology. Technical report, BCC, Inc., June 2005.

[105] Dawid Kurzyniec and Vaidy S. Sunderam. Semantic aspects of asynchronous rmi: The rmix approach. In *IPDPS*, 2004.

[106] Stefan Lankes, Andreas Jabs, and Thomas Bemmerl. Design and performance of a can-based connection-oriented protocol for real-time corba. *J. Syst. Softw.*, 77(1):37–45, 2005.

[107] Stefan Lankes, Andreas Jabs, and Michael Reke. A time-triggered ethernet protocol for real-time corba. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 215–, 2002.

[108] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[109] Sheng Liang. *Java Native Interface Specification: Programmer's Guide and Specification*. Java Series. Adddison-Wesley,Boston, Mass., 1996.

[110] J. Lindblad. Reducing memory fragmentation. *Embedded Systems Engeneering*, 2004.

[111] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification. Second Edition.* Java Series. Adddison-Wesley,Boston, Mass.., 1999.

[112] Marcus Ruark Lisa Carnahan. Requirements for the real-time extensions for the java platform. Technical report, NIST, september 1999.

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

[114] Qusay H. Mahmoud, editor. *Middleware for Communications.* Wiley, 2004.

[115] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *RTSS*, pages 62–71, 2005.

[116] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *ECRTS*, pages 79–86, 2004.

[117] Akihiko Miyoshi, Takuro Kitayama, and Hideyuki Tokuda. Implementation and evaluation of real-time java threads. In *IEEE Real-Time Systems Symposium*, pages 166–175, 1997.

[118] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, Massachusetss Institute of Technology, 1983.

[119] Bruce Nelson. Remote procedure call. Technical report, Xerox Palo Alto Research Center, 1981.

[120] Eric Newcomer. *Understanding Web Services.* Addison-Wesley, 2002.

[121] Kelvin Nilsen. Distinctions between perc api and nist requirements document. Technical report, 1999.

[122] Kelvin Nilsen. Making effective use of the real-time specification for java. Available on-line at http://research.aonix.com/jsc/rtsj.issues.9-04.pdf, 2004.

[123] Kelvin Nilsen. *Draft Guidelines for Scalable Java Development of Real-Time Systems.* Aonix, 2005. Availale on-line at: http://research.aonix.com/jsc/rtjava.guidelines.3-26-05.pdf.

[124] Kelvin D. Nilsen. Invited note: Java for real-time. *Real-Time Systems*, 11(2):197–205, 1996.

[125] Kelvin D. Nilsen. Adding real-time capabilities to java. *Commun. ACM*, 41(6):49–56, 1998.

[126] Kelvin D. Nilsen and Andrew Klein. Issues in the design and implementation of efficient interfaces between hard and soft real-time java components. In *OTM Workshops*, pages 451–465, 2003.

[127] J. Duane Northcutt. *Mechanisms for reliable distributed real-time operating systems: The Alpha Kernel*. Academic Press Professional, Inc., San Diego, CA, USA, 1987.

[128] Inc. Objective Interface Systems. Jcp rtsj and real-time corba synthesis: Request for proposal. orbos/2002-01-16, 2001.

[129] Inc. Objective Interface Systems. J-consortium rtcore and real-time corba synthesis: Request for proposal. orbos/2002-01-16, 2002.

[130] Inc. Objective Interface Systems. Jcp rtsj and real-time corba synthesis: Initial submision. realtime/2002-06-02, 2002. Available on-line at http://www.omg.org/docs/realtime/02-06-02.pdf.

[131] Inc. Objective Interface Systems. J-consortium rtcore and real-time corba synthesis: Revised submision. realtime/2003-05-04, 2003. Available on-line at www.omg.org/docs/realtime/03-05-04.pdf.

[132] OMG. Common Object Request Broker Architecture (CORBA/IIOP). CORBA v.3.1, 2004.

[133] OMG. Real Time Corba Specification Version 1.2. formal/05-01-04, 2005.

[134] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 372–395, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[135] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 67–76, New York, NY, USA, 2003. ACM Press.

[136] Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In *ECOOP*, pages 378–404, 2003.

[137] Alessandro Paseti. *Software Frameworks and Embedded Control Systems*, volume 2331 of *Lectures Notes in Computer Science*. Springer, 2002.

[138] Paulo Pedreiras and Luis Almeida. The flexible time-triggered (ftt) paradigm: An approach to qos in distributed real-time systems. In *17th International Parallel And Distributed Processing Symposium*, page 123, April 2003.

[139] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[140] F. Pizlo, J. M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: Design patterns and semantics. In *ISORC*, pages 101–110, 2004.

[141] Daniel Port and Monica McArthur. A study of productivity and efficiency for object-oriented methods and languages. In *APSEC'99: Porceedings of the Sixth Asia Pacific Software Engineering Conference*, page 128, Washington, DC, USA, 1999. IEEE Computer Society.

[142] Irfan Pyarali. *Patterns for Providing Real-Time Guarantees in DOC Middleware*. PhD thesis, Washington University, St. Louis, MO 63130, December 2001. Available at: http://www.zen.uci.edu.

[143] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[144] Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares, and Raymond Klefstad. Patterns and tools for achieving predictability and performance with real-time java. In *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 247–253, Washington, DC, USA, 2005. IEEE Computer Society.

[145] Sangig Rho. *A Distributed Hard Real-Time Java for Hign Mobility Components*. PhD thesis, Texas, December 2004.

[146] Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[147] M. Rinard. Flex compiler infraestructure. Web, 2004. Available at http://www.flex-compiler.lcs.mit.edu/Harpoon/.

[148] Lankes S. and Bemmerl T. Design and implementation of a sci-based real-time corba. In *ISORC '01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 23, Washington, DC, USA, 2001. IEEE Computer Society.

[149] Richard E. Schantz and Douglas C. Schmidt. Research advances in middleware for distributed systems. In *Communication Systems: The State of the Art (IFIP World Computer Congress)*, pages 1–36, 2002.

[150] Douglas C. Schmidt, Aniruddha Gokhale, Richard E. Schantz, and Joseph P. Loyall. Middleware r&d challenges for distributed real-time and embedded systems. *SIGBED Review*, 1(1), April 2004.

[151] Douglas C. Schmidt and Fred Kuhns. An overview of the real-time corba specification. *IEEE Computer*, 33(6):56–63, 2000.

[152] Martin Schoberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Universidad Técnica de Viena, 2005. Available at http://www.jopdesign.com/thesis/.

[153] Martin Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[154] Martin Schoeberl. Restrictions of java for embedded real-time systems. In *ISORC*, pages 93–100, 2004.

[155] Lui Sha, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore P. Baker, Alan Burns, Giorgio C. Buttazzo, Marco Caccamo, John P. Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[156] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[157] Lui Sha, Ragunathan Rajkumar, Sang Hyuk Son, and Chun-Hyon Chang. A real-time locking protocol. *IEEE Trans. Computers*, 40(7):793–800, 1991.

[158] D. Sharp. Reducing avionics software cost through component based product line development. In *Software Technology Conference*, April 1998.

[159] David C. Sharp, Edward Pla, and Kenn R. Luecke. Evaluating mission critical large-scale embedded system performance in real-time java. In *RTSS*, pages 362–365, 2003.

[160] Hojjat Jafarpour Raymond Klefstad Shruti Gorappa, Jan A. Colmenares. Tool-based configuration of real-time corba middleware for embedded systems. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 342–349, Washington, DC, USA, 2005. IEEE Computer Society.

[161] Keng Siau and Terry A. Halpin, editors. *Unified Modeling Language: Systems Analysis,Design and Development Issues*. Idea Group, 2001.

[162] Siebert. The jamaica vm. Web, 2004. Available at http://www.aicas.com.

[163] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. In *RTCSA*, pages 96–102, 1999.

[164] Jon Siegel. A preview of corba 3. *IEEE Computer*, 32(5):114–116, 1999.

[165] John A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, 2004.

[166] D. B. Stewart and Pradeep Khosla. Real-time scheduling of sensor-based control systems. In *IEEE Workshop on Real-Time Operating Systems and Software (RTOS '91)*, pages 144 – 150, May 1991.

[167] Sun. Java remote method invocation. RMI v.1.5, 2004. Available on-line at http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf.

[168] Sun. Enterprise java beans. EJB v.2.1, 2005.

[169] Daniel Tejera, Ruth Tolosa, Miguel A. de Miguel, and Alejandro Alonso. Dos modelos alternativos de rmi para aplicaciones distribuidas de tiempo real. In *Actas del primer congreso español de informática*, 2005.

[170] Daniel Tejera, Ruth Tolosa, Miguel A. de Miguel, and Alejandro Alonso. Two alternative rmi models for real-time distributed applications. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 390–397, Washington, DC, USA, 2005. IEEE Computer Society.

[171] Krupp P. Schafer A. Thuraisingham, B. and V. Wolfe. On real-time extensions to the common object request broker architecture. 1994.

[172] Timesys. *Timesys JTIME RTSJ 1.0 Extensions User Guide*. Timesys, 2002. Available at http://www.timesys.com.

[173] Timesys. Jtime virtual machine. Web, 2004. Available at http://www.timesys.com.

[174] Timesys. Ibm websphere real-time. Web, 2006. Available at http://www-306.ibm.com/sotware/webservers/realtime/.

[175] Timesys. Timesysos. Web, 2006. Available at http://www.timesys.com.

[176] Ken Tindell, Alan Burns, and Andy J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, 1995.

[177] Bhavanai Thurasingham Victor Fay-Wolfe, John K. Black and Peter Krupp. Real-time method invocations in distributed environments., 1995. Technical Report 95-244, University of Rhode Island, Department of Computer Science and Statistics.

[178] Steve Vinoski. An overview of middleware. In *Ada-Europe*, pages 35–51, 2004.

[179] W3C. Soap version 1.2 part 1: Messaging framework. SOAP v.1.2 recomendation, 2003.

[180] Nanbor Wang. *Composing Systemic Aspects Into Component-Oriented DOC Middleware*. PhD thesis, Washington University, St. Louis, MO 63130, May 2004. Available at: http://www.zen.uci.edu/publications/.

[181] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao. Real-time component-based systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 428–437, 2005.

[182] Andy J. Wellings, Gregory Bollella, Peter C. Dibble, and David Holmes. Cost enforcement and deadline monitoring in the real-time specification for java. In *ISORC*, pages 78–85, 2004.

[183] Andy J. Wellings and Alan Burns. Asynchronous event handling and real-time threads in the real-time specification for java. In *IEEE Real Time Technology and Applications Symposium*, pages 81–89, 2002.

[184] Andy J. Wellings, Roy Clark, E. Douglas Jensen, and Douglas Wells. The distributed real-time specification for java: A status report. In *Embedded Systems Conference*, pages 13–22, 2002.

[185] Andy J. Wellings, Roy Clark, E. Douglas Jensen, and Douglas Wells. A framework for integrating the real-time specification for java and java's remote method invocation. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 13–22, 2002.

[186] James P. White and David A. Hemphill. *Java 2 Micro Edittion*. Manning, 2002.

[187] Paul R. Wilson. Uniprocessor garbage collection techniques. In *IWMM*, pages 1–42, 1992.

[188] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, pages 1–116, 1995.

[189] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russell Johnston. Real-time corba. In *IEEE Real Time Technology and Applications Symposium*, pages 148–157, 1997.

[190] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java system. In *COOTS*, 1996.

[191] Ann Wollrath, Geoff Wyant, and Jim Waldo. Simple activation for distributed objects. In *COOTS*, 1995.

[192] A. Zerzelidis and Andy J. Wellings. Requirements for a real-time .net framework. *SIGPLAN Notices*, 40(2):41–50, 2005.