SciVerse ScienceDirect
Disponible en www.sciencedirect.com

Revista Iberoamericana de Automática e Informática industrial 10 (2013) 228–239

www.elsevier.es/RIAI

Diseño de un Software de Intermediación de Comunicación para Sistemas Distribuidos de Tiempo Real Críticos en Java

Daniel Tejera, Alejandro Alonso*, Miguel A. de Miguel

Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, Ciudad Universitaria s/n, 28040, Madrid, España

Resumen

Las facilidades e independencia de plataforma de Java han generado un gran interés en la comunidad de tiempo real. Dicho interés se ha reflejado en la especificación RTSJ (*Real-Time Specification for Java*), que extiende y adapta el lenguaje Java para permitir el desarrollo de sistemas de tiempo real. Adicionalmente, se han desarrollado perfiles de RTSJ para garantizar la predecibilidad en sistemas de tiempo real críticos. Sin embargo, RTSJ y sus perfiles no proporcionan facilidades para sistemas distribuidos. El objetivo de este trabajo es afrontar dicha limitación definiendo un nuevo modelo de RMI (*Remote Method Invocation*) basado en los principales perfiles de RTSJ para sistemas de tiempo real crítico. Este trabajo presenta el diseño y la implementación de RMI-HRT (*RMI-Hard Real-Time*) que está enfocado a sistemas de tiempo real crítico con requisitos de alta integridad.

Palabras Clave: Sistemas críticos, sistemas de tiempo real, sistemas distribuidos, Java

1. Introducción

Los sistemas empotrados y distribuidos de tiempo real son cada vez más importantes para la sociedad. Su demanda aumenta y se depende de los servicios que proporcionan. Los sistemas de alta integridad constituyen un subconjunto de gran importancia. Se caracterizan por que un fallo en su funcionamiento puede causar la pérdidas de vidas humanas, daños en el medio ambiente o cuantiosas pérdidas económicas. La necesidad de satisfacer requisitos temporales estrictos, hace más complejo su desarrollo. El aumento de este tipo de aplicaciones implica la necesidad de tecnologías de desarrollo que permitan reducir su coste, que sean flexibles e independientes del hardware.

La tecnología que se emplea en aplicaciones de alta integridad como las de aviónica, espaciales o ferroviarias, se beneficia de varias décadas de innovación y enfoques de desarrollo de software avanzados, que han producido herramientas y métodos adecuados. Sin embargo, la situación actual presenta retos nuevos que requieren su mejora. La falta de flexibilidad y el coste de desarrollo de los sistemas de alta integridad, hace que los desarrolladores no estén plenamente satisfechos con las metodologías existentes. El coste de transportar una aplicación de una plataforma a otra es elevado. En muchos casos, el código resultante es dependiente de las características específicas del procesador y del comportamiento temporal y del uso de recursos en la plataforma de ejecución (Crespo et al., 2006). La evolución de las redes y paradigmas de comunicación, así como la necesidad de mayor potencia de cómputo y de tolerancia a fallos ha motivado la interconexión de dispositivos electrónicos. Los mecanismos de comunicación permiten la transferencia de datos con alta velocidad de transmisión. En este contexto, el concepto de sistema distribuido ha emergido, como sistemas donde sus componentes se ejecutan en varios nodos en paralelo y que interactúan entre ellos mediante redes de comunicaciones.

Un concepto interesante son los sistemas de tiempo real neutrales respecto a la plataforma de ejecución. Se caracterizan por la falta de conocimiento de esta plataforma durante su diseño. Esta propiedad es relevante, por que conviene que se ejecuten en la mayor variedad de arquitecturas, tienen una vida media mayor de diez años y la plataforma de ejecución puede variar. El lenguaje de programación Java es una buena base para el desarrollo de este tipo de sistemas.

La popularidad de Java, sus características y su independencia de la plataforma le han convertido en un lenguaje de interés para las comunidades de tiempo real y de sistemas empotrados. Esta situación ha motivado el desarrollo de RTSJ (Belliardi et al., 2006), que es una extensión del lenguaje para permitir el desarrollo de sistemas de tiempo real. Entre la extensiones que se han incluido, se encuentran las hebras de tiempo real, eventos

*Autor en correspondencia
Correos electrónicos: tejera@dit.upm.es (Daniel Tejera), aalonso@dit.upm.es (Alejandro Alonso), mmiguel@dit.upm.es (Miguel A. de Miguel)

http://dx.doi.org/10.1016/j.riai.2013.03.016

asíncronos y un modelo de memoria nuevo que permite evitar o limitar el efecto del recolector de memoria. Las ventajas potenciales de Java han motivado su interés en el desarrollo de sistemas de alta integridad.

El uso de Java en este tipo de sistemas requiere técnicas muy estrictas de desarrollo y prueba, así como el seguimiento de estándares relevantes. La versión actual de RTSJ incluye características que no están permitidas en el desarrollo de sistemas de alta integridad, como la creación dinámica de hebras, el uso de memoria dinámica o algunas características de la orientación a objetos. El enfoque empleado en otros lenguajes de programación ha consistido en la definición de un perfil restringido de los mismos que permita el análisis estático de sistemas de alta integridad que deban ser certificados. HRTJ (*Hard Real-Time Java*) define uno de estos perfiles, que fue desarrollado en el marco del proyecto HIJA (*High Integrity Java Applications*), que está basado en *Ravenscar-Java* (Kwon et al., 2005). Actualmente, se está creando una especificación similar en el contexto del proceso comunitario de Java (JSR-302), cuyo objetivo es definir el perfil SCJ (*Safety Critical Java*) (Locke et al., 2011) para crear aplicaciones de alta integridad con Java.

Aunque la mayoría de los sistemas actuales y futuros serán distribuidos, RTSJ no proporciona mecanismos para su desarrollo. Se ha creado un grupo de expertos en sistemas distribuidos de tiempo real en Java (JSR-50) para definir abstracciones apropiadas para solucionar este problema. Sin embargo, en este momento no hay una especificación formal de las mismas. El objetivo de este trabajo es desarrollar un software de intermediación de comunicaciones que sea adecuado para el desarrollo de sistemas distribuidos de tiempo real crítico con Java.

El objetivo principal de este trabajo es describir el desarrollo de RMI-HRT (Tejera et al., 2007) (Tejera, 2012), que es un software de intermediación de comunicaciones para el desarrollo sistemas distribuidos de tiempo real crítico. El objetivo subyacente es allanar el camino para la certificación de aplicaciones distribuidas con la tecnología Java. El modelo del software de intermediación (*middleware*) debe permitir el uso de herramientas de análisis para obtener los tiempos de respuesta, optimizar los recursos y servir de punto de partida para la certificación de las aplicaciones.

En el resto del documento se presenta los principales aspectos que caracterizan a RMI-HRT. Después del estado de la técnica, la sección 3 describe el modelo computacional y la sección 4 muestra el diseño del software de intermediación. La sección 5 aborda la serialización predecible y la sección 6 la validación en un caso industrial. Finalmente la sección 7 describe las conclusiones.

2. Estado de la Técnica

Un sistema de tiempo real es aquél en el que la corrección del sistema depende tanto del valor lógico de los resultados, como del instante de tiempo en el que se producen. Los sistemas de tiempo real se pueden clasificar en tres tipos fundamentales, dependiendo de las consecuencias que pueden producirse si se incumplen requisitos temporales:

- Críticos: Un incumplimiento puede provocar daños en seres humanos.
- Firmes: Una respuesta fuera de plazo es inútil, pero el sistema puede seguir funcionando aunque la calidad de servicio se vea degradada.
- Acríticos: una respuesta fuera de tiempo tiene una validez relativa, pero el sistema sigue operativo.

2.1. RTSJ

RTSJ es un especificación de lenguaje diseñada para desarrollar sistemas de tiempo real. Sus fundamentos son: no añadir restricciones al entorno de ejecución de Java, mantener la compatibilidad con los programas de Java, no añadir extensiones sintácticas, ni palabras reservadas adicionales y asegurar una ejecución predecible. Las características más relevantes de RTSJ son:

Objetos planificables y Planificadores: RTSJ define un modelo de concurrencia basado en un conjunto de hebras o manejadores de eventos asíncronos (conocidos como objetos planificables). RTSJ permite utilizar diferentes planificadores, pero el planificador por defecto está basado en prioridades con desalojo (28 prioridades como mínimo).

Transferencia asíncrona de control: Es una extensión al mecanismo de interrupción de hebras de Java para permitir la entrega de control y el manejo de excepciones asíncronas.

Sincronización: El algoritmo por defecto para evitar una inversión de prioridad no acotada es el protocolo de herencia de prioridad, aunque el protocolo de techo de prioridad puede usarse bajo demanda.

Memoria: Se ha introducido un modelo nuevo de memoria para evitar la incertidumbre que produce el recolector de memoria. Define nuevos tipos de memoria:

- Memoria Inmortal: contiene objetos que nunca serán fragmentados o recolectados por el recolector de memoria y pueden ser compartidos por diferentes objetos planificables. Los objetos en memoria inmortal permanecerán hasta el final de la aplicación.
- Memoria Restringida (*scoped*): tiene un tiempo de vida limitado. Los objetos ubicados en memoria restringida se mantienen mientras que haya objetos planificables con referencias a los mismos. Cuando se eliminen todas las referencias a los objetos en memoria restringida, pueden ser eliminados.
- Se han definido otros tipos de memoria, que permiten acceder a la memoria física.

Tiempo y Temporizadores: Se han definido clases adicionales para representar tiempos relativos o absolutos con alta resolución. A cada objeto que representa un tiempo se le asocia un reloj de tiempo real.

2.2. Requisitos del software de intermediación

La primera actividad de este trabajo ha sido la identificación del conjunto de requisitos que debe satisfacer un software de intermediación para sistemas de tiempo real distribuidos de alta integridad o críticos. Estos requisitos se han compilado a partir del estado de la técnica, de las características de este tipo de sistemas y de la interacción con la industria. Han servido de base para la comparación y análisis de software de intermediación existente. A continuación se muestran los requisitos más importantes:

- Gestión de hebras: Debe ser posible analizar el tiempo de respuesta de las hebras. Además, se deben cumplir las restricciones de los sistemas de alta integridad sobre el modelo de concurrencia, por ejemplo, la hebra debe ser creada de forma estática y su patrón de activación debe ser periódico o esporádico.
- Gestión de memoria: Se debe evitar la creación dinámica de objetos y proporcionar mecanismos, para crear objetos temporales y mantener el estado estado de la aplicación entre diferentes fases. Además, debe ser posible determinar una cota máxima a la capacidad de memoria necesaria.
- Gestión de la comunicación: Se debe evitar la creación dinámica de conexiones, por lo que se deben establecer durante la inicialización del sistema. Los protocolos de comunicación deben permitir el cálculo del tiempo de transmisión de cada mensaje. Para ello, se debe determinar estáticamente el tamaño máximo de los mensajes a trasmitir.
- Parámetros de activación: Los parámetros de activación de las hebras o los manejadores de eventos, deben ser independientes de la implementación del software de intermediación y establecidos por el desarrollador.
- Serialización predecible: Se debe poder acotar el tiempo de cómputo de peor caso y el uso de otros recursos.
- Evitar características orientadas a objetos: Algunas de las funcionalidades de los lenguajes orientados a objetos, hacen que sea complicado determinar el tiempo de uso de procesador o de memoria.
- Mecanismos de recuperación de errores: Los sistemas deben incluir mecanismos de detección y recuperación de errores funcionales y temporales.
- Perfil de tiempo real crítico de Java: El software de intermediación debe utilizar un subconjunto de Java que permita lograr un alto nivel de predecibilidad y permitir el uso de técnicas de verificación.

2.3. Trabajos Relacionados

El interés del problema planteado ha motivado el desarrollo de software de intermediación como los siguientes: RT-CORBA (Raman et al., 2005) (OMG, 2002) (OMG, 1999), PolyORB (Vergnaud et al., 2004), RT-GLADE (Gutiérrez et al., 2001), DRTSJ (JSR-50), DREQUIEMI (Basanta et al., 2004) y RMI-QoS (Tejera et al., 2005). Estos trabajos fueron analizados teniendo en cuenta la lista de requisitos mostradas en la sección anterior. La conclusión es que ninguno de ellos satisface la mayoría de los requisitos enumerados, no siendo adecuados para el desarrollo de sistemas distribuidos de alta integridad.

Todos estos trabajos incluyen un manejo de hebras adecuado. La mayoría de ellos permite especificar parámetros de activación, como la prioridad y el plazo, y tienen una buena gestión de conexiones. La interoperabilidad con sistemas no críticos sólo es posible con RT-CORBA y PolyORB. La gestión de memoria presenta carencias en todas las soluciones. Los requisitos de control sobre las conexiones, mecanismos de recuperación de errores y conformidad con el perfil de tiempo real de Java, solo se proporcionan de forma parcial en DREQUIEMI y RMI-QoS.

La predecibilidad en el uso de recursos como la memoria y la red, es el requisito menos cubierto en estas sistemas. En casi todos los casos no es posible acotar con precisión la cantidad de memoria utilizada o el ancho de banda necesario para transmitir los mensajes. Además, la mayoría de las implementaciones usan algoritmos complejos o características de orientación a objetos, que dificultan el análisis de tiempos de respuesta.

2.4. Invocación remota a método

RMI es el modelo de objetos distribuidos de Java que permite invocar un método de un objeto que se ejecuta en otra maquina virtual. Su principal propósito es permitir el desarrollo de aplicaciones distribuidas con el mismo modelo de programación que en sistemas centralizados. RMI es una infraestructura poderosa para la construcción de aplicaciones distribuidas cliente/servidor. RMI emplea TCP/IP y serialización de objetos para serializar y deserializar los argumentos y el valor de retorno de los métodos remotos.

El servidor crea un conjunto de objetos remotos, los hace accesibles, y espera a peticiones de los clientes. Normalmente, el cliente recibe una o más referencias remotas a los objetos en el servidor, mediante un registro que almacena par de referencia remota - objeto remoto. Con las referencias el cliente llama a los métodos que se encuentran en el servidor. El cliente, el servidor y el registro forman la aplicación de objetos distribuidos. RMI es el software de intermediación que proporciona servicios para el desarrollo de aplicaciones de objetos distribuidos.

RMI de Java tiene limitaciones para sistemas de tiempo real:

- Emplea tecnologías tales como: reflexión, características de orientación de objetos y recursividad que hacen que sea muy difícil estimar el peor tiempo de cómputo y el uso de recursos.
- El modelo de memoria no es apropiado. Las hebras, las conexiones, los objetos remotos, etc. se crean dentro del montículo. Por lo tanto, la ejecución de las invocaciones remotas puede sufrir los efectos del recolector de memoria y los objetos se crean de forma dinámica.

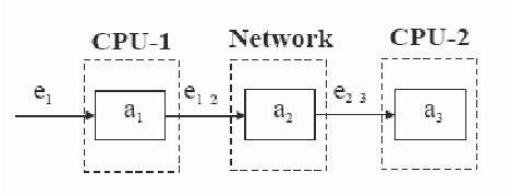

Figura 1: Modelo lineal

- En las actuales implementaciones de RMI las conexiones se generan de forma dinámica. Cuando un cliente quiere enviar una invocación, primero crea una nueva conexión con el servidor. Es difícil estimar el tiempo y los recursos necesarios para establecer la conexión.
- Las implementaciones de RMI (por ejemplo, GNU Classpath) crean hebras dinámicamente, lo que no es adecuado en sistemas de tiempo real crítico.
- Uso de algoritmos de serialización no predecibles.
- El uso de recursos es completamente transparente, y por lo tanto no es configurable. RMI no permite al programador especificar los parámetros de red.
- En RMI no se han teniendo en cuenta los parámetros de tiempo real. No se consideran los parámetros de planificación o de activación del cliente o del servidor.

3. Modelo computacional

El modelo de análisis para cada nodo del sistema distribuido está basado en el planificador de prioridades fijas con desalojo. La prioridad se asigna estáticamente a las hebras y siempre se ejecuta la hebra lista con mayor prioridad. El acceso a los recursos compartidos se basa en el protocolo del techo de prioridades. Es una técnica madura que permite calcular estáticamente el tiempo de respuesta de cada hebra y la planificabilidad de todo el sistema. Además, hay herramientas disponibles para realizar estos cálculos, como MAST (Drake et al., 2006).

El modelo de análisis para una aplicación distribuida está basado en el modelo lineal (Gutiérrez et al., 1997). El sistema se compone de un conjunto de transacciones, como se muestra en la figura 1. Cada transacción está compuesta por un conjunto de acciones ordenadas y se caracteriza por un patrón de activación periódico o esporádico. Una acción puede ser una hebra ejecutado una porción de código o un mensaje enviado por el medio de comunicaciones. Una acción es activada por la previa (excepto por la primera donde el tiempo de activación es igual al tiempo de activación de la transacción) y activa a la siguiente.

El análisis del tiempo de respuesta se basa en los peores casos de tiempo de transmisión y ejecución. Aunque este modelo es adecuado para varios tipos de sistemas, el trabajo (Gutiérrez et al., 2000) extiende el enfoque para soportar transacciones que no son lineales, tales como situaciones donde una acción puede activar varias acciones.

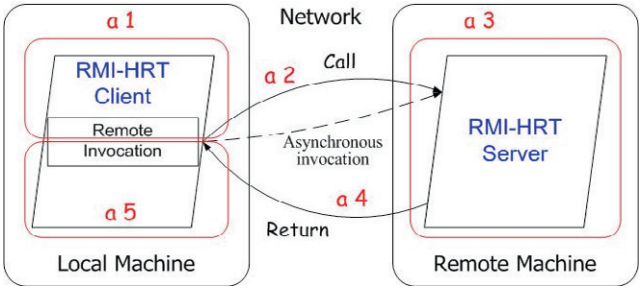

Figura 2: Modelo computacional

3.1. *Perfil de tiempo real critico de Java*

El modelo computacional y la implementación están basados en el perfil HRTJ (un subconjunto de RTSJ definido dentro del proyecto HIJA) y la especificación SCJ. Estos perfiles eliminan las características que implican una gran sobrecarga y semántica compleja, para proveer un comportamiento predecible.

Cada aplicación se ejecuta en dos fases: la fase de inicialización, donde se llevan a cabo todas las acciones que no son de tiempo real, y la fase de misión, donde se ejecuta la funcionalidad propia de la aplicación. Los objetos que son necesarios durante toda la vida de la aplicación se crean durante la fase de inicialización en memoria inmortal y nunca se eliminan. Cada objeto planificable tiene su propia memoria restringida, donde se crean los objetos necesarios durante la fase de misión. Entre activaciones de un objeto planificable, la memoria privada restringida se libera. No se permiten áreas de memoria restringida anidadas, ni que se compartan por varias hebras.

El sistema está compuesto por un conjunto de objetos concurrentes gestionados con un planificador basado en prioridades estáticas y con expulsión. Los objetos pueden ser hebras o manejadores de eventos, y deben tener parámetros de activación periódicos o esporádicos. Para prevenir una inversión de prioridades ilimitada, se utiliza el protocolo de techo de prioridad para todos los objetos con bloques o métodos sincronizados. Las transferencias asíncronas de control están prohibidas, ya que dificultan el análisis y la maquina virtual.

3.2. *Extensiones al perfil para sistemas distribuidos*

El modelo de comunicaciones para sistemas distribuidos está basado en invocaciones remotas. En el caso más simple, figura 2, el modelo computacional está compuesto por dos flujos de control (o dos hebras): el cliente y el servidor. El cliente llama a métodos que se ejecutan en un equipo remoto. Un servidor obtiene las invocaciones, ejecuta el método apropiado y retorna los resultados. Según se muestra en la figura 2, las invocaciones pueden ser síncronas o asíncronas. En el primer caso, el cliente invoca un método remoto y espera por la respuesta del servidor. En el segundo, el cliente no necesita esperar por la respuesta. En una aplicación más compleja, puede formarse una cadena de invocaciones donde el servidor ejecuta también el papel de cliente.

En este tipo de sistemas, es necesario emplear un protocolo de comunicaciones con comportamiento temporal predecible. Los mensajes deben cumplir sus plazos de repuesta, los cuáles se deben verificar por construcción o analíticamente. Hay varios tipos de redes que ofrecen estas garantías, con diferentes enfoques:

- Dirigidas por tiempo: Los mensajes se transmiten en instantes precisos de tiempo, determinados estáticamente y almacenados en tablas. La construcción de estos planes garantiza un ancho de banda determinado y acota el peor caso de entrega de un mensaje. El protocolo TTP (Kopetz et al., 1994) está dirigido por tiempo.
- Dirigidas por prioridades: Las prioridades se asignan a los mensajes y se usan para decidir cuál es el orden de envió de mensajes. Hay métodos analíticos para calcular el peor tiempo de transmisión de los mismos. La red CAN (CAN, 1991) se comporta según este principio.
- Enlaces virtuales: Estas redes aseguran a cada enlace un cierto ancho de banda y limitan la variabilidad (*jitter*) que pueden sufrir los mensajes. El protocolo de red AFDX (ARINC, 2004) permite definir enlaces virtuales con estas características.

Algunos aspectos del modelo de invocaciones remotas, como los servicios web, el servidor de nombres y el recolector de memoria distribuido, dificultan el análisis de tiempos de respuesta. Además, el número de invocaciones en cada activación tiene que ser conocido y acotado. Con estas restricciones las invocaciones remotas son analizables.

4. Diseño de RMI-HRT

4.1. Referencias remotas

En RMI, el concepto de variable de referencia de Java se extiende a las aplicaciones distribuidas como referencia remota. Una aplicación puede utilizar las referencias para invocar métodos en objetos locales o referencias remotas para invocar métodos en objetos remotos. La referencia remota contiene: la dirección IP, un puerto TCP, un identificador de objeto y otros parámetros necesarios para invocar métodos en un objeto remoto específico. La mayoría de los parámetros se refieren a donde se encuentra el objeto remoto. Cuando un cliente utiliza una referencia remota, los parámetros se utilizan para crear una nueva conexión con el servidor. A pesar de que la referencia contiene varios parámetros, es independiente de la conexión y el cliente. Por esta razón, puede ser compartida por varios clientes.

En RMI-HRT, cada referencia a un objeto remoto se asocia con todos los recursos necesarios para manejar las solicitudes invocadas. Cada referencia se asocia también con los parámetros de tiempo real del cliente, del servidor y de la red. Además, RMI-HRT introduce el concepto de creación de referencias. Es el proceso por el que el cliente y el servidor interactúan para asignar recursos a las referencias. Se trata de la creación de una nueva conexión y recursos tales como: hebras y áreas de memoria. Todas las referencias en el sistema tienen que ser creadas antes de que los clientes puedan llamar a los métodos remotos.

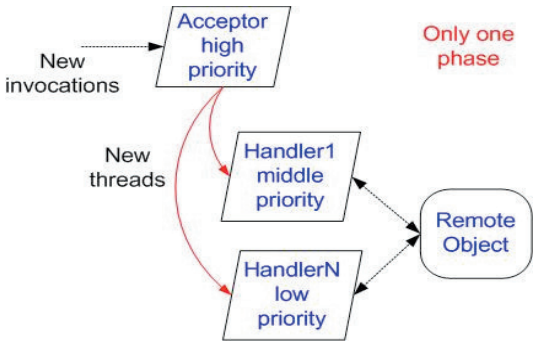

Figura 3: Modelo de hebras genérico

Este enfoque permite a la aplicación especificar todos los parámetros de los recursos. Una vez que los recursos están reservados, no se puede cambiarlos. Si un cliente quiere hacer invocaciones diferentes sobre el mismo objeto remoto con diferentes parámetros, es necesario tener diferentes referencias.

Las referencias ya no son genéricas, por que representan a un cliente, un servidor, los parámetros de tiempo real, las hebras y las áreas de memoria. Por tanto, no pueden ser compartidas. La referencia es el medio básico para llamar a métodos específicos de un determinado objeto remoto con atributos de tiempo real fijos.

4.2. Modelo de hebras en RMI-HRT

El modelo utilizado por RMI contiene sólo una fase en la que un servidor siempre espera nuevas conexiones. Cuando se crea una conexión nueva, se crean las hebras necesarias para manejar las invocaciones que llegan a través de ella. La asignación de recursos y la ejecución sucede al mismo tiempo. De acuerdo con la especificación de RMI, las implementaciones pueden utilizar un mecanismo arbitrario de hebras para el manejo de invocaciones ya que los requisitos temporales no se consideran. RMI-HRT define dos fases y un mecanismo de hebras específico para garantizar un comportamiento temporal adecuado, que es compatible con el modelo computacional.

Un enfoque común es incluir a dos hebras (llamadas *Acceptor* y *Handler*) para ejecutar las invocaciones en el lado servidor, como se muestra en la figura 3. El *Acceptor* acepta nuevas peticiones y crea una nueva conexión a nivel de red y del software de intermediación. Cuando recibe nuevas invocaciones, las entrega al *Handler*, que realiza la ejecución del método. En algunos casos, el *Handler* se crea a petición del *Acceptor*. En otros casos, el *Handler* se toma de un conjunto de hebras creadas previamente.

Con este enfoque, todas las invocaciones nuevas se tratan con los mismos parámetros de planificación. Por lo general, el *Acceptor* tiene la máxima prioridad para evitar la inversión de prioridades (invocaciones de prioridad alta deben desalojar a invocaciones de baja prioridad en curso). Sin embargo, este

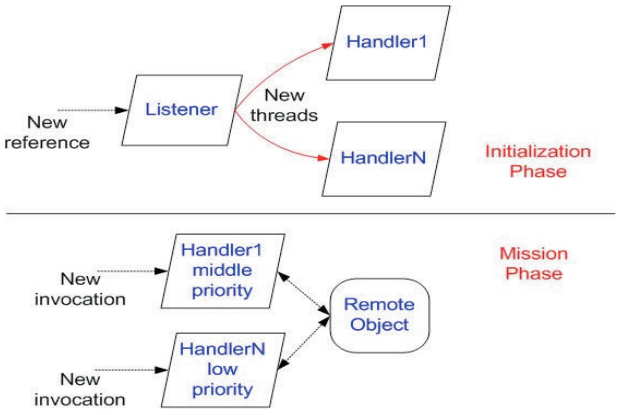

Figura 4: Modelo de hebras en RMI-HRT

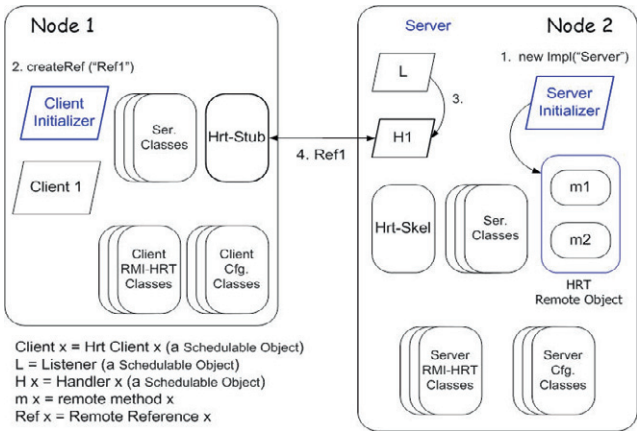

Figura 5: Fase de Inicialización

enfoque tiene la desventaja de desalojar a una invocación de prioridad media en proceso (un *Handler*) para aceptar una nueva invocación que podría tener una prioridad baja. RMI-HRT también se basa en dos hebras: (*Listener* y *Handler*). Como se muestra en la figura 4, el *Listener* es el encargado de crear las referencias remotas durante la fase de inicialización. Esta operación supone la creación de una nueva conexión y un nuevo *Handler*. En la fase de misión, el *Handler* es el encargado de aceptar y ejecutar las invocaciones remotas con los parámetros asociados de planificación.

En RMI-HRT, la inversión de prioridades se evita por que las invocaciones de prioridad media no serán desalojadas por una invocación de baja prioridad. En otras palabras, el *Handler* de más alta prioridad se ejecuta en primer lugar y sin interrupciones.

RMI-HRT no requiere la creación de hebras en el lado del cliente para manejar las llamadas. La hebra que invoca el método remoto, ejecuta el código que envía la solicitud y espera una respuesta. RMI-HRT incluye también llamada asincrónas, en las que se modifica el flujo de control habitual y se permite al cliente llevar a cabo otras tareas mientras el servidor procesa la llamada remota. RMI-HRT permite este tipo de operaciones únicamente cuando el método remoto no devuelve un valor de retorno.

4.3. Fase de inicialización

En la fase de inicialización se crean y se reservan los recursos, de acuerdo a los parámetros que describen la configuración del sistema. Por lo tanto, el cliente y el servidor interactúan para reservar recursos tales como: conexiones, hebras y áreas de memoria. La figura 5 describe las tareas llevadas a cabo durante la fase de inicialización:

1. Cuando la aplicación del servidor se inicia, se ejecuta la hebra que se encarga de llevar a cabo la inicialización, para crear los objetos utilizados por la aplicaciones, entre ellos, los objetos remotos. Cuando un objeto remoto se crea y exporta, RMI-HRT lleva a cabo varias tareas:
   - Las clases de configuración y todas las clases de RMI-HRT (las utilizadas por el servidor) se cargan en memoria (esta tarea se ejecuta sólo una vez, cuando se crea el primer objeto remoto). Las clases de serialización se cargan en la memoria inmortal.
   - El objeto remoto se asocia a un identificador, que se usa en RMI-HRT para obtener la configuración de los parámetros de tiempo real y los parámetros de red correspondientes.
   - RMI-HRT carga la clase del esqueleto en la memoria. Una instancia del esqueleto se utiliza para invocar el proceso de serialización y el método remoto correspondiente.
   - Con los parámetros de red, RMI-HRT configura el protocolo de red subyacente para aceptar nuevas conexiones.
   - Un objeto planificable (*Listener*) se crea para recibir y manejar el proceso de creación de referencias.
2. En el lado del cliente, se crea un objeto planificable asociado a una memoria restringida, y una referencia remota para llevar a cabo la aplicación del cliente. Cuando se crea una referencia remota se ejecutan las siguientes actividades
   - La configuración y todas las clases de RMI-HRT se cargan en memoria (esta tarea se ejecuta sólo una vez, durante la creación de la primera referencia remota). Además, las clases de serialización se cargan en la memoria inmortal.
   - RMI-HRT carga el suplente (*stub*) y se crea una instancia. La referencia utilizada para la aplicación del cliente es una referencia a un suplente.
   - Con los parámetros de red, se establece una conexión con el servidor. Los parámetros no funcionales se intercambian entre el cliente y el servidor.
   - El *Listener* se encarga de asociar un nuevo objeto planificable (*Handler*) a la nueva conexión, que se

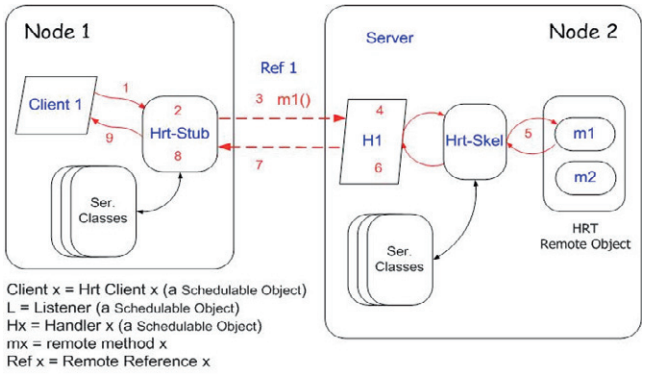

Figura 6: Fase de misión

encargará de tratar las llamadas recibidas a través de la conexión (referencia). El *Listener* también crea la memoria restringida utilizada por el *Handler*.

- A partir de este momento los recursos han sido asignados y el cliente puede utilizar la referencia para invocar métodos remotos.

4.4. *Fase de misión*

En la fase de la misión, los recursos ya se han reservado, por lo tanto, los clientes llevan a cabo las invocaciones remotas y el servidor las procesa. La figura 6 describe las actividades llevadas a cabo durante esta fase:

1. El cliente inicia la invocación remota llamando a un método auxiliar del suplente, con la misma signatura que el método remoto.
2. El suplente serializa los argumentos del método mediante las clases apropiadas.
3. La solicitud se transmite por la red. Se envían uno o más mensajes, dependiendo del tamaño de la solicitud y la implementación del propio protocolo. Si la solicitud representa una invocación asíncrona, el suplente devuelve el control al cliente. En caso contrario, espera la respuesta.
4. En el lado del servidor, cuando se recibe una nueva invocación, se activa el *Handler*. Éste llama al método correspondiente del esqueleto, para extraer los argumentos de la solicitud usando las clases de serialización e invocar el método remoto.
5. Se ejecuta el método remoto.
6. El esqueleto invoca las clases de serialización para serializar el valor de retorno. RMI-HRT compone la respuesta con un encabezado fijo y el valor de retorno. Para detectar posibles errores, la respuesta se almacena en un tampón (*buffer*) intermedio antes de enviar el mensaje.
7. Si la invocación es síncrona, el *Handler* envía la respuesta al cliente y queda a la espera de nuevas invocaciones.
8. En una invocación síncrona, el suplente llama a las clases de serialización para obtener el valor de retorno, y devuelve el control a la aplicación.

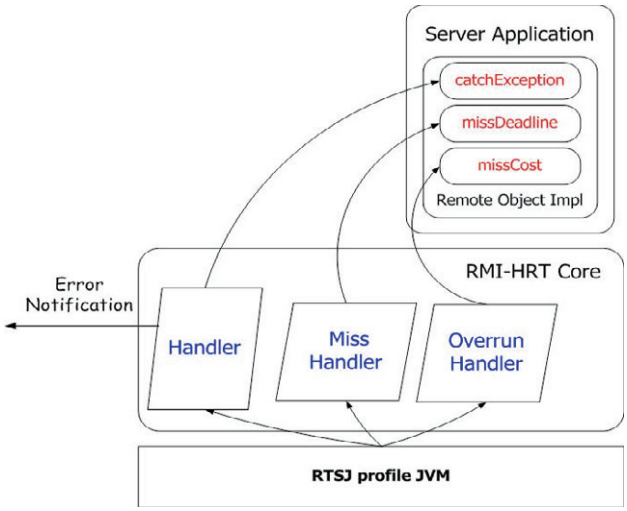

Figura 7: Gestión de errores

4.5. *Gestión de errores*

RMI-HRT incluye mecanismos para que la aplicación adopte medidas en caso de posibles errores. RMI-HRT gestiona las excepciones y controla los tiempos de ejecución. Se lanza una excepción, cuando no se cumple el plazo, o cuando se supera el tiempo de ejecución. RMI-HRT permite a la aplicación llevar a cabo acciones correctivas o conducir el sistema a un estado seguro.

La figura 7 muestra la interacción entre aplicaciones y el RMI-HRT. Si una excepción se produce cuando un método remoto se está ejecutando en el servidor, el *Handler* envía una respuesta al cliente notificando la excepción, llama al método *catchException*, y restablece todos las variables y los tampones internos. Al final de la operación, el *Handler* puede recibir nuevas peticiones del cliente. Sin embargo, la aplicación es quien debe determinar si eliminar el objeto remoto o parar un *Handler* determinado (una referencia determinada).

Cuando el *Handler* no cumple su plazo o su máximo tiempo de ejecución (coste en RTSJ) durante la ejecución del método remoto, se invoca a *missDeadline* o a *missCost*. Estos métodos permiten a la aplicación las acciones necesarias para tratar estos eventos. Aunque RTSJ no permite dejar un hilo activo, la aplicación puede tomar medidas para reducir el consumo de CPU. El controlador ejecuta el método *catchException*, mientras que *missDeadline* y *missCost* las ejecutan manejadores de eventos asociados con el *Handler*: *MissHandler* y *OverrunHandler* respectivamente.

En el lado cliente, las excepciones se manejan como en RMI, por lo tanto, la aplicación cliente puede capturarlas y manejarlas. Se emplean temporizadores para detectar un comportamiento temporal anómalo. Por ejemplo, cuando se produce un error inesperado en el servidor, el temporizador desbloquea el cliente. El valor del temporizador es el tiempo de respuesta de extremo a extremo de la invocación, calculado analíticamente.

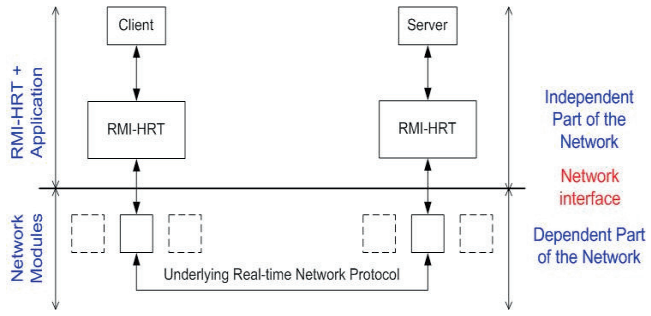

Figura 8: Módulos de red

4.6. *Interfaz de red*

La figura 8 muestra cómo RMI-HRT aísla la parte dependiente de la red en módulos para soportar diferentes protocolos e independizar el software de intermediación de la red. La configuración del sistema es responsable de definir qué módulo será utilizado para cada referencia remota.

RMI-HRT supone que los módulos de red encapsulan protocolos de tiempo real, donde el tiempo de transmisión está acotado y puede ser analizado. Además, RMI-HRT supone que los módulos de red tienen un comportamiento orientado a conexión y se garantiza la entrega de mensajes. Por lo tanto, RMI-HRT no tiene que encargarse de funciones tales como: la retransmisión de mensajes, algoritmos de gestión de congestión, etc.

Se ha definido una interfaz de red para especificar las comunicaciones entre el núcleo de RMI-HRT y el módulo de red. En la figura 9, podemos ver los cinco grupos de primitivas.

1. El primer grupo tiene como objetivo crear conexiones entre el cliente y el servidor.
2. RMI-HRT supone que el nombre de computador está asociado con los parámetros de red (en muchas redes su nombre está asociado con una dirección IP). Por lo tanto, el módulo de red debe proporcionar a RMI-HRT este nombre.
3. El tercer grupo introduce el concepto de flujo (*stream*), que es una secuencia de bytes. Es una abstracción que permite enviar o recibir mensajes a través de una red.

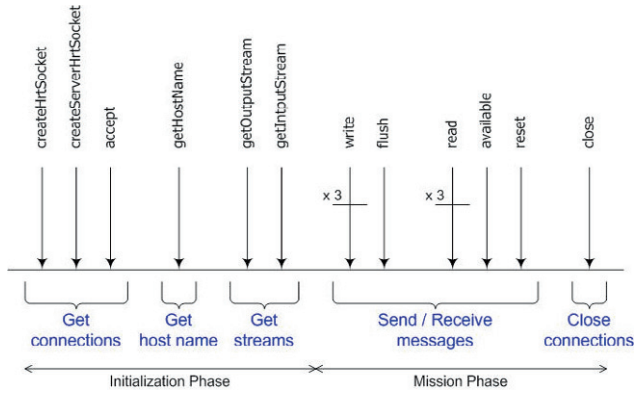

Figura 9: Interfaz de Red

4. El cuarto grupo es el encargado de proporcionar un mecanismo simple para leer y escribir bytes en el flujo.
5. Finalmente, el último grupo incluye una primitiva para cerrar las conexiones.

RMI-HRT gestiona estas primitivas sin saber qué protocolo de red se está empleando. Los tres primeros grupos de primitivas se invocan durante la fase de inicialización para crear las conexiones y los flujos asociados. En la fase de la misión, RMI-HRT emplea primitivas como *read* o *write* para intercambiar mensajes. Al final de la aplicación o cuando se elimina un objeto remoto, se ejecuta la primitiva *close* para liberar la conexión y sus recursos.

El servidor utiliza las primitivas *accept* y *createServerHrtSocket* para esperar a conexiones nuevas. La primitiva *createServerHrtSocket* da la orden al módulo de red para prepararse para recibir nuevas conexiones y la primitiva *accept* da la orden de esperar y aceptar nuevas conexiones. El cliente tiene que usar la primitiva *createHrtSocket* para crear una conexión con el servidor. Las primitivas *createHrtSocket* y *accept* requieren dos parámetros importantes: el máximo tamaño de la solicitud y el máximo tamaño de la respuesta. Estos valores se usan para establecer tampones internos.

Las primitivas *getOutputStream* y *getInputStream* permiten obtener un flujo de salida y de entrada de una conexión específica. Las primitivas *write* y *flush* operan sobre un flujo de salida, y las primitivas *read*, *available* y *reset* operan sobre un flujo de entrada. RMI-HRT supone que la escritura no bloquea la hebra de tiempo real y los valores se almacenan en un tampón interno. Por medio de la primitiva *flush*, RMI-HRT solicita al módulo de red que envíe los datos. El modelo requiere una implementación predecible de la primitiva *flush*, con un tiempo de transmisión limitado.

La primitiva *read* obtiene una secuencia de bytes desde el flujo de entrada. Cuando el flujo está vacío, la hebra de tiempo real que realiza la llamada se bloquea hasta que el flujo tenga datos. En el lado servidor, este mecanismo es adecuado ya que debe esperar de forma indefinida a nuevas solicitudes. Sin embargo, el cliente debe esperar un período corto de tiempo por la respuesta. La primitiva *available* permite a RMI-HRT saber cuántos bytes están disponibles en el flujo de entrada.

4.7. *HRT-JRMP*

La especificación de RMI define un protocolo de conexión conocido como JRMP (Java Remote Method Protocol) que establece un formato estándar para el intercambio de peticiones y respuestas. El protocolo es flexible, ya que permite enviar mensajes con diferentes protocolos y multiplexar varias conexiones en una. Estas propiedades permiten ampliar las situaciones en las que se puede emplear RMI, pero complica el análisis de tiempos de respuesta. Por otro lado, este protocolo tiene algunas características que reducen su flexibilidad. Por ejemplo, JRMP no permite el intercambio de parámetros no funcionales entre el cliente y el servidor y se desarrolló principalmente para el protocolo TCP / IP.

Aunque algunos software de intermediación utilizan el protocolo JRMP y añaden los parámetros no funcionales como ar-

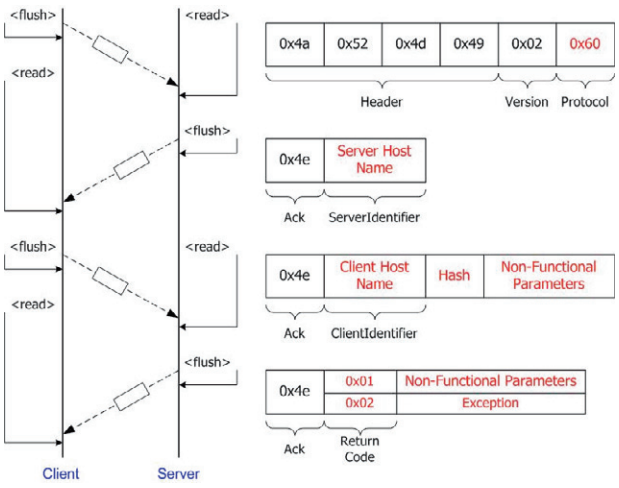

Figura 10: HRT-JRMP: Fase de inicialización

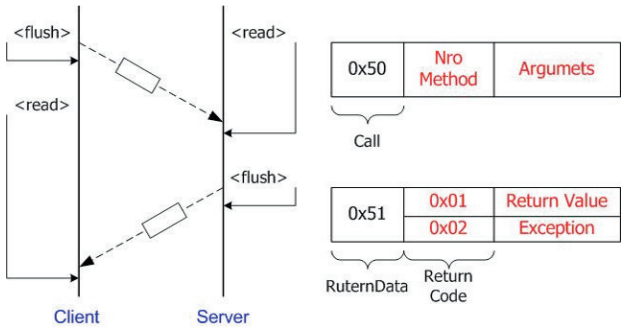

Figura 11: HRT-JRMP: Fase de misión

gumento, este trabajo propone HRT-JRMP (Hard Real Time - JRMP), un subconjunto de JRMP adaptado a RMI-HRT, en el que las fases de inicialización y misión están claramente definidas. Además, el HRT-JRMP es independiente del protocolo de red subyacente.

De acuerdo con JRMP, cuando la aplicación requiere un flujo simple los mensajes envían con el protocolo *StreamProtocol*. Por lo tanto, *HrtStreamProtocol* es una extensión del protocolo *StreamProtocol*.

La figura 10 muestra los mensajes intercambiados en la fase de inicialización para establecer una conexión. El cliente inicia la negociación enviando un mensaje; el servidor debe reconocer la negociación mediante el envío de un *ServerIdentifier* (el nombre del nodo del servidor). El cliente debe responder con un *ClientIdentifier*, un *hash* (de la interfaz del suplente) y un conjunto de parámetros no funcionales. Con el *hash*, el servidor puede verificar que el suplente del cliente se corresponde con el esqueleto. Si no se corresponde, el servidor envía un mensaje con un código de error y la excepción correspondiente.

Este mecanismo permite al cliente propagar sus requisitos y al servidor confirmarlos o actualizarlos. La implementación sabe que todos los parámetros están disponibles en archivos de configuración, por lo que se envían referencias a la configuración (nombre de la referencia) como parámetros no funcionales.

La figura 11 muestra la fase de misión. El cliente puede enviar solicitudes basadas en la versión 1.1 de RMI. El número de campos se ha reducido a tres: un valor que representa un mensaje de llamada, un número que representa el método que se invoca (asignado por `rmic-hrt`), y la lista de argumentos. No son necesarios parámetros para identificar el objeto remoto, dado que cada referencia, conexión y *Handler* hacen referencia al mismo. El *hash* también fue eliminado ya que se envían sólo una vez en la fase de inicialización.

La respuesta tiene tres campos: un valor que representa un mensaje de retorno, un código de retorno (un retorno normal o excepcional) y el valor de retorno o la excepción. Con este nuevo protocolo, la tasa entre el tamaño de los datos (en bytes) y el número de bytes enviados a través de la red en la fase de la misión se incrementa en un porcentaje significativo.

4.8. Gestión de memoria

La figura 12 muestra los componentes y los tipos de memorias que interactúan durante la ejecución del método remoto en el cliente. RMI-HRT supone que la hebra del cliente entra en su memoria restringida (CSM) antes de invocar a un método remoto. Por lo que todos los objetos temporales se almacenan en dicha memoria y se eliminarán en el momento que la hebra del cliente libere la memoria. La aplicación cliente puede utilizar objetos en la memoria inmortal (COI) o crear nuevos objetos, tanto en la memoria inmortal (COI) como en la memoria de restringida (COS).

La solicitud de la invocación remota no consume memoria restringida, la cabecera y los datos se almacenan en un tampón dentro de la memoria inmortal (B1). RMI-HRT reutiliza objetos en memoria inmortal. A pesar de que la respuesta también se almacena en otro tampón en la memoria inmortal (B2), el proceso de deserialización reconstruye el valor de retorno (*ReturnValue*) dentro de la memoria restringida (CSM). Además, algunos módulos de red necesitan crear objetos temporales (*TemporaryObjects*). La memoria restringida tiene que ser lo suficientemente grande para contener todos los objetos temporales creados por la aplicación cliente, el valor de retorno y los objetos creados por el módulo de red. Si el cliente quiere mantener el valor de retorno o de un estado entre diferentes invocaciones tiene que utilizar memoria inmortal.

En el servidor (figura 13), *TriggerHandler* es una hebra para abordar una carencia de la maquina virtual usada: el *Handler* no se puede activar cuando llega un mensaje. Esta hebra emplea una memoria restringida (TSM) para esperar a una nueva solicitud y sale de la memoria cuando la invocación ha sido procesada por el *Handler*. Como consecuencia, todos los objetos creados (*TemporaryObjects*) por el módulo de red se eliminan entre invocaciones. Esto no es necesario si el módulo de red reutiliza objetos en la memoria inmortal.

Cada vez que el *TriggerHandler* activa al *Handler*, también se emplea memoria restringida (HSM) para manejar la nueva solicitud. Esta memoria puede ser liberada después de enviar la respuesta al cliente. Como resultado, todos los objetos temporales generados por el proceso de deserialización se almacenan en la memoria restringida del *Handler*. La implementación

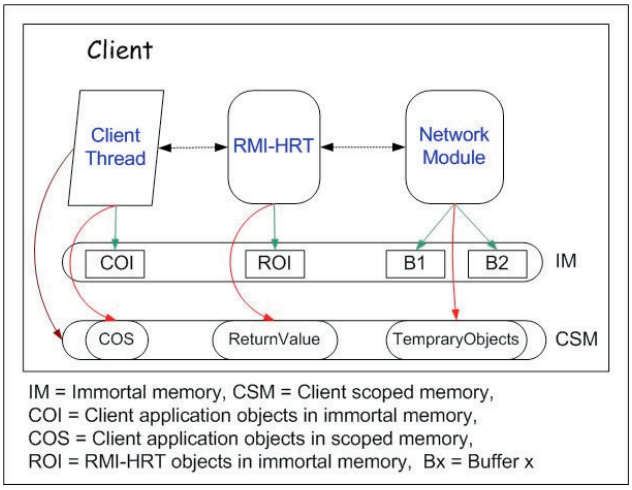

Figura 12: Gestión de memoria en el cliente

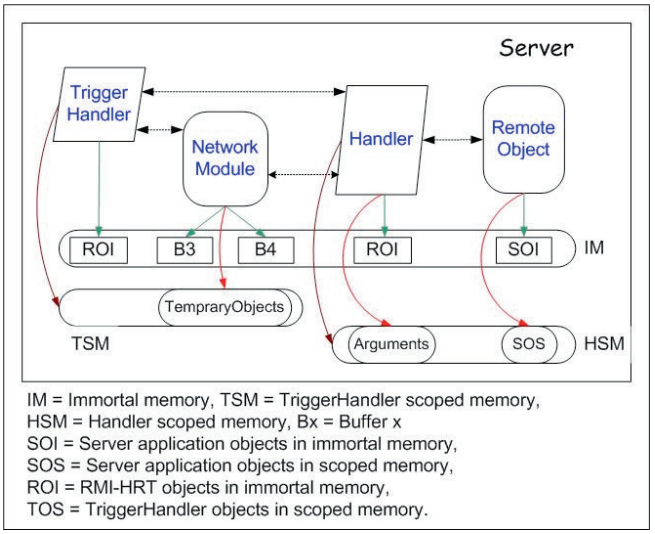

Figura 13: Gestión de memoria en el servidor

del objeto remoto se debe desarrollar con estas consideraciones para evitar la generación de asignaciones de memoria ilegales. Además, la aplicación tiene que evitar entrar en otras áreas de memorias restringidas. A pesar de estas limitaciones, la aplicación puede crear todos los objetos necesarios (SOI y SOS). Por lo tanto, la memoria restringida tiene que ser lo suficientemente grande para contener todos los argumentos de la invocación y los objetos creados por la ejecución del método remoto. La solicitud y la respuesta se guardan en tampones ubicados en memoria inmortal (B3, B4). Para mantener un estado entre las invocaciones, la aplicación tiene que utilizar objetos en memoria inmortal (SOI).

En resumen, el modelo de memoria propuesto incluye las siguientes características:

- Los objetos planificables, las áreas de memoria, los tampones, y todos los objetos necesarios durante la ejecución de un método remoto se crean en la memoria inmortal.
- Los objetos temporales (ya sean generados por el software de intermediación o por el método remoto) se crean en la memoria restringida.
- El método remoto puede mantener un estado entre invocaciones utilizando memoria inmortal.
- La implementación de RMI-HRT respeta las reglas de asignación de memoria impuestas por RTSJ y su perfil HRTJ.
- Con este modelo, es posible calcular la cantidad de memoria necesaria para invocar un método remoto.

5. Validación

Con objeto de validar el diseño e implementación de RMI-HRT, se ha llevado a cabo un exigente proceso de validación. Se han empleado varios métodos para satisfacer todos los requisitos establecidos, como la revisión de la documentación e inspección de código, la ejecución de pruebas específicas en una plataforma determinada. La mayor parte del esfuerzo se ha dedicado a validar el diseño, el prototipo y el compilador respecto: al comportamiento funcional, al uso de memoria, al uso de la CPU (tiempo de respuesta de extremo a extremo y en cada uno de los bloques funcionales) y al uso de la red (consumo real tiene que ser conforme a lo estimado) (Tejera, 2012).

El diseño y el prototipo han sido validados en una aplicación industrial en el marco del proyecto HIJA (Hu et al., 2006). RMI-HRT se ha utilizado de forma satisfactoria en las comunicaciones de un sistema de gestión de vuelo (FMS). Un FMS lleva a cabo una amplia variedad de tareas durante el vuelo, por ejemplo: proporciona las predicciones de tiempo de vuelo, el kilometraje, la velocidad, y elimina muchas de las operaciones de rutina realizadas normalmente por los pilotos. El software de intermediación se ha desarrollado y probado usando la máquina virtual JamaicaVM.

El prototipo de FMS implementaba cinco funciones: gestor de guiado, gestor de piloto automático, gestor de la predicción, gestor de plan de vuelo y gestor de bases de datos. Un simulador de vuelo se empleó para validar el comportamiento del prototipo. Estos elementos estaban conectados mediante RMI-HRT. Los periodos de la predicción, guiado, piloto automático y simulador eran de 20s, 150ms, 50ms y 50ms, respectivamente. Los resultados fueron satisfactorios y conformes a los requisitos especificados. Los mecanismos de serialización permitieron reducir la carga y acotar los tiempos de uso de CPU y de red. El uso de llamadas asíncronas permitió eliminar comunicaciones no necesarias.

6. Conclusión

La evolución de los sistemas empotrados requiere de técnicas para el desarrollo de aplicaciones distribuidas con requisi-

tos temporales y de alta integridad. RMI-HRT es un software de intermediación adecuado para satisfacer los requisitos del desarrollo de este tipo de sistemas. El modelo de invocación remota de método proporciona un alto nivel de abstracción en el desarrollo de aplicaciones distribuidas, ya que permite desarrollarlas con los mismos mecanismos que las centralizadas.

El diseño e implementación de RMI-HRT se ha realizado de forma que tenga un comportamiento predecible. La ejecución de la aplicación en dos fases permite separar la asignación de recursos de su uso. En la fase de inicialización, se asignan de todos los recursos necesarios para la siguiente fase y se llevan a cabo todas las operaciones que no son completamente deterministas (como la creación de conexiones). En la fase de misión, las invocaciones remotas se realizan con los recursos asignados previamente. Las referencias remotas se asocian a un conjunto específico de recursos (objetos planificables, memoria y red). La serialización de los datos es una funcionalidad fundamental en RMI-HRT. Para garantizar su operación coherente con el resto del sistema, se ha desarrollado un algoritmo de serialización que es predecible.

RMI-HRT es conforme al perfil de tiempo real crítico de Java. El sistema incluye todas las características conforme al perfil HRTJ y a los principales conceptos de la especificación SCJ actual. Al inicio de este trabajo se identificaron un conjunto de requisitos necesarios para permitir el uso de RMI-HRT en sistemas críticos. Las pruebas de validación y verificación han mostrado que estos requisitos se han satisfecho.

En este tipo de sistemas, la gestión de errores es una funcionalidad importante. RMI-HRT supervisa el comportamiento funcional y temporal. La aplicación ejecuta el código de tratamiento de errores cuando éstos se producen.

En relación a la interacción y comunicación, se proporciona la posibilidad de realizar invocaciones síncronas y asíncronas. La aplicación puede elegir si el cliente tiene que esperar por una respuesta o no. El modo asíncrono requiere menos memoria, CPU y ancho de banda. Con objeto de reducir la sobrecarga y mejorar la predicibilidad durante la invocación remota, se ha definido un protocolo de conexión (HRT-JRMP). Además, el sistema se ha diseñado para proporcionar independencia del protocolo de red. Se ha aislado la parte dependiente de la red, mediante la definición de una interfaz clara que permite al software de intermediación operar sin conocer detalles del protocolo subyacente.

El compilador desarrollado permite determinar estáticamente el tamaño máximo de los mensajes de la invocación y de la respuesta para cada uno de los métodos, considerando el peor de los casos. Estos valores son necesarios para calcular el ancho de banda necesario para cada invocación.

English Summary

Design of a Communication Middleware for Distributed Real-Time Safety Systems in Java

Abstract

Distributed real-time embedded systems are becoming increasingly important to society. More demands will be made on them and greater reliance will be placed on the delivery of their services. A relevant subset of them is high-integrity or hard real-time systems, where failure can cause loss of life, environmental harm, or significant financial loss.

Additionally, the evolution of communication networks and paradigms as well as the necessity of demanding processing power and fault tolerance, motivated the interconnection between electronic devices; many of the communications have the possibility of transferring data at a high speed. The concept of distributed systems emerged as systems where different parts are executed on several nodes that interact with each other via a communication network.

Java's popularity, facilities and platform independence have made it an interesting language for the real-time and embedded community. This was the motivation for the development of RTSJ (Real-Time Specification for Java), which is a language extension intended to allow the development of real-time systems. The use of Java in the development of high-integrity systems requires strict development and testing techniques.

However, neither RTSJ nor its profiles provide facilities to develop distributed real-time applications. This is an important issue, as most of the current and future systems will be distributed. The Distributed RTSJ (DRTSJ) Expert Group was created under the Java community process (JSR-50) in order to define appropriate abstractions to overcome this problem. Currently there is no formal specification.

The aim of this work is to describe the design and develop a communication middleware that is suitable for the development of distributed hard real-time systems in Java, based on the integration between the RMI (Remote Method Invocation) model and the HRTJ profile. It has been designed and implemented keeping in mind the main requirements such as the predictability and reliability in the timing behavior and the resource usage.

The design starts with the definition of a computational model which identifies among other things: the communication model, most appropriate underlying network protocols, the analysis model, and a subset of Java for hard real-time systems. In the design, the remote references are the basic means for building distributed applications which are associated with all non-functional parameters and resources needed to implement synchronous or asynchronous remote invocations with real-time attributes.

The proposed middleware separates the resource allocation from the execution itself by defining two phases and a specific threading mechanism that guarantees a suitable timing behavior. It also includes mechanisms to monitor the functional and the timing behavior. It provides independence from network protocol defining a network interface and modules. The JRMP protocol was modified to include two phases, non-functional parameters, and message size optimizations. Although serialization is one of the fundamental operations to ensure proper data transmission, current implementations are not suitable for hard real-time systems and there are no alternatives. This work proposes a predictable serialization that introduces a new com-

piler to generate optimized code according to the computational model. The proposed solution has the advantage of allowing us to schedule the communications and to adjust the memory usage at compilation time.

In order to validate the design and the implementation a demanding validation process was carried out with emphasis in the functional behavior, the memory usage, the processor usage (the end-to-end response time and the response time in each functional block) and the network usage (real consumption according to the calculated consumption).

Keywords:
Safety Systems, Real-Time Systems, Distributed Systems, Java.

Agradecimientos

Los autores agradecen a los miembros del grupo ”Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos”de la Universidad Politécnica de Madrid por sus útiles apreciaciones y a los miembros del consorcio del proyecto HIJA por su colaboración en el desarrollo de este trabajo.

Este trabajo ha sido financiado parcialmente por el proyecto HIJA de la Comisión Europea (IST-511718, Framework 6, priority 2.3.2.5, Embedded Systems) y por el Ministerio Español de Educación y Ciencia, por medio del proyecto HI-PARTES (Sistemas empotrados particionados de alta integridad, TIN2011-28567-C03-01) del Plan Nacional de I+D+I.

Referencias

ARINC 664. “Aircraft Data Network, 2004, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network, Draft 3”. September.

Basanta, P., García-Valls, M., Estevez-Ayres, I., 2004, “No Heap Remote Objects: Leaving Out the Garbage Collector at the Server Side”. En Proceedings of the Second International Workshop on Java Technologies for Real-Time and Embedded Systems JTRES04, LNCS 3292, p. 25-29 October.

Belliardi, R., Brosgol, B., Dibble, P., Holmes, D., Wellings, A., 2006, The Real-Time Specification for Java. Version 1.0.2, http://www.rtsj.org

CAN, Robert Bosch GmbH, Stuttgart. “CAN Specification Version 2.0”. Germany, 1991.

Crespo, A., Alonso, A., 2006, “Una Panorámica de los Sistemas de Tiempo Real”, Revista Iberoamericana de Automática e Informática Industrial, Volumen 3, Número 2, pp.7-18 - Abril.

Drake, J.M., González-Harbour, M., Gutiérrez, J.J., Medina, J., Palencia, J.C., 2006, “En Busca de la Integración de Herramientas de Tiempo Real a Través de un Modelo Abierto”, Revista iberoamericana de automática e informática industrial (RIAI), ISSN-e 1697-7912, Vol. 3, Nº. 2, pp. 28-39.

Gutiérrez, J., Harbour, M.G., 1997, “On the Schedulability Analysis for Distributed Hard Real-Time Systems”. En Proceedings 9th Euromicro Workshop on Real Time Systems, June.

Gutiérrez, J., Palencia, J.C., Harbour, M.G., 2000, “Schedulability Analysis of Distributed Hard Real-Time Systems with Multiple- Event Synchronization”. En Proceedings of 12th Euromicro Conference on Real-Time Systems, Stockholm (Sweden), IEEE Computer Society Press, pp. 15-24, June.

Gutiérrez J.J., Harbour, M.G. 2001, “Towards a real-time distributed systems annex in Ada”. ACM Ada Letters, Vol. XXI(1), p. 62-66.

Hu, E.Y., Jenn, E., Valot, E., Alonso, A., 2006, “Safety critical applications and hard real-time profile for Java: a case study in avionics”. E Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, ACM International Conference Proceeding Series; ISBN:1-59593-544-4, vol. 177, pp. 125-134, DOI: 10.1145/1167999.1168021

JSR-50: “Distributed Real-Time Specification”. http://www.jcp.org/en/jsr/detail?id=050.

Kopetz, H., Grünsteidl, G., 1994, “TTP-A Protocol for Fault-Tolerant Real-Time Systems”. IEEE Computer, pp 14-23, Enero.

Kwon, J., Wellings, A., King, S., 2005, “Ravenscar-Java: a high-integrity profile for real-time java: Research articles”. Concurr. Comput.: Pract. Exper. 17(5-6): 681-713.

Locke, D., Andersen, B. S., Brosgol, B., Fulton, M., Henties, T., Hunt, J. J., Nielsen, J. O., Nilsen, K., Schoeberl, M., Tokar, J., Vitek, J. and Wellings, A., “Safety-critical java technology specification, public draft.”2011. http://www.jcp.org/en/jsr/detail?id=302

OMG, Object Management Group, 1999, “The Common Object Request Broker: Architecture and Specification: Revision 2.3.1”. www.omg.org/cgi-bin/doc?formal/99-10-07/.

OMG, Object Management Group, 2002, Real-Time CORBA Specification”. August.

Oracle, ”Java Remote Method Invocation Specification”. http://docs.oracle.com/javase/1.4.2/docs/guide/rmi/spec/rmi-title.html

Raman, K., Zhang, Y., Panahi, M., Colmenares, J.A., Klefstad, R., Trevor, R., 2005, “RTZen: highly predictable, real-time java middleware for distributed and embedded systems”, en Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware, Ed. Springer-Verlag, p. 225-248.

Tejera, D., Tolosa, R., de Miguel, M.A., Alonso, A, 2005 “Two alternative RMI models for real-time distributed applications”. En Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society, Washington, DC, USA, p. 390-397.

Tejera, D., Alonso, A., de Miguel, M.A., 2007, “RMI-HRT: Remote Method Invocation - Hard Real Time”, En Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, Vienna, Austria, ACM International Conference Proceeding Series, ISBN: 978-59593-813-8, pp: 113 - 120.

Tejera, D., 2012, Communication Middleware for Distributed Hard Real-Time Systems in Java, PhD thesis.Universidad Poltécnica de Madrid, 2012

Vergnaud, T., Hugues, J., Pautet, L., Kordon, F., 2004, “Polyorb: A schizophrenic middleware to build versatile reliable distributed applications”, En Reliable Software Technologies - Ada-Europe, Lecture Notes in Computer Science, Vol. 3063, p. 106-119.