Serialización en sistemas distribuidos de tiempo real crítico

Daniel Tejera, Alejandro Alonso, Miguel A. de Miguel
Departamento de Ingeniería de Sistemas Telemáticos,
ETSI Telecomunicación, Universidad Politécnica de Madrid,
Ciudad Universitaria s/n, E-28040, Madrid, Spain.
Contacto email: tejera@dit.upm.es

Resumen

RTSJ (*Real-Time Specification for Java*) extiende y adapta la tecnología Java para permitir el desarrollo de sistemas de tiempo real. Pero para soportar el desarrollo de sistemas distribuidos de tiempo real crítico es necesario rediseñar operaciones básicas tales como la serialización. Este trabajo describe un nuevo proceso de serialización centrado en los sistemas distribuidos de tiempo real crítico, el cual proporciona total predecibilidad en tiempo y espacio.

1. Introducción

RTSJ (*Real-Time Specification for Java*) [1] es una extensión del lenguaje la cual permite el desarrollo de sistemas de tiempo real (STR). Estas extensiones incluyen hebras (*threads*) de tiempo real, eventos asíncronos y un nuevo modelo de memoria.

RTSJ no fue pensado para sistemas distribuidos de tiempo real (SDTR). Esto es una limitación importante, pues la mayoría de los sistemas son o serán distribuidos. Un grupo de expertos (*The Distributed RTSJ DRTSJ Expert Group*) fue creado bajo proceso de la comunidad Java (JSR-50) [10] para definir un conjunto de abstracciones apropiadas para superar dicha limitación. Aunque hay algunas guias técnicas [2], [17] y [9], en este momento no hay una especificación formal. El último trabajo divulgado se basa en hebras distribuidas y RMI-RT (*Remote Method Invocation for Real-Time*) como dos abstracciones adecuadas para la distribución.

El proyecto europeo HIJA [8] (*High-Integrity Java Applications*), en el cual los autores han participado activamente, ha tenido como objetivo crear condiciones tecnológicas para permitir una arquitectura neutral para los STR. En el contexto de HIJA, se han propuesto tres perfiles para satisfacer las necesidades de diversos dominios de aplicación:

- *Perfil de tiempo real crítico* (HRTJ – *Hard Real Time profile*): Está enfocado a aplicaciones críticas, donde fallos pueden causar lesiones graves en seres humanos (por ejemplo: en aviónica)
- *Perfil de tiempo real acrítico* (*Flexible Soft Real-time Profile*): Está enfocado a aplicaciones críticas de negocio (por ejemplo: algunos sistemas telemáticos).
- *Perfil de tiempo real flexible* (*Flexible Mixed Real-Time Profile*): Está pensado para aplicaciones que requieren mucha flexibilidad (por ejemplo: en ambientes inteligente).

El dominio de aplicación de este trabajo es el definido por el perfil HRTJ. Dicho perfil está basado en el perfil de Ravenscar Ada [3] y las pautas del NRC [7]. Define un subconjunto de RTSJ para proveer un modelo de programación concurrente con una ejecución de aplicaciones predecible y confiable.

El proyecto HIJA también ha tenido como objetivo proporcionar soporte para aplicaciones distribuidas de tiempo real. En el área de

aplicaciones de tiempo real crítico, se ha desarrollado RMI-HRT (*Remote Method Invocation - Hard Real-Time*) [15], es una adaptación del modelo general de RMI [14] y es consistente con el perfil HRTJ y con la especificación del protocolo de RMI [14].

RMI y RMI-HRT se basan en la serialización de objetos [13] para pasar objetos como argumento en una invocación remota (desde un cliente a un servidor) o como valor de retorno (desde un servidor a un cliente). Por consiguiente RMI-HRT requiere la implementación de algún proceso de serialización consistente con el perfil HRTJ, el cual permita calcular estáticamente el peor tiempo de ejecución (WCET - *Worst Case Execution Time*) y el uso de memoria.

Este trabajo es una actualización de lo hecho en [16]. La sección 2 introduce RMI-HRT y la sección 3 describe el proceso de serialización, donde la serialización predecible es comentada con mayor profundidad. La sección 4 proporciona un uso industrial. La sección 5 comenta brevemente algunos trabajos relacionados, mientras que la sección 6 da las conclusiones.

2. RMI-HRT

RMI-HRT rediseña RMI con la finalidad de proveer mecanismos para el desarrollo de aplicaciones distribuidas dentro del dominio definido por el perfil HRTJ. El objetivo primordial de RMI-HRT es proveer una plataforma distribuida y determinista, la cual permita limitar el peor tiempo de ejecución y calcular el uso de recursos (memoria y el ancho de banda). La implementación de RMI-HRT ha tomado como punto de partida la implementación de RMI de Classpath [5], pero ambas implementaciones tienen muchas diferencias tanto a nivel de clases e interfaces como a nivel de funcionalidad. Las principales características de RMI-HRT son las siguientes:

- *Perfil HRTJ*: puede ser ejecutado en cualquier máquina virtual que implemente RTSJ y las restricciones del perfil HRTJ.
- *Dos Fases*: la fase de inicialización, donde se crean todos los objetos que existirán durante toda la vida de la aplicación, y la fase de ejecución, donde la aplicación se lleva a cabo.
- *Memoria y Concurrencia*: utiliza la memoria inmortal para mantener un estado entre invocaciones y memoria restringida para eliminar los objetos temporales. Solo utiliza hebras de tiempo real y eventos asíncronos con prioridades fijas.
- *Protocolos de red*: con el objetivo de soportar diferentes protocolos de red, la gestión del protocolo de red subyacente es separada del núcleo de RMI-HRT. Se ha definido una interfaz de red que especifica las comunicaciones entres el núcleo y los módulos de red. La implementación incluye módulos para: TCP/IP, UPD/IP y UDP/AFDX.
- *Compatibilidad*: el protocolo subyacente de RMI ha sido respetado casi en su totalidad para facilitar futuras interacciones con las actuales implementaciones de RMI. La única diferencia es un campo para propagar parámetros de tiempo real.
- *Inter-operabilidad*: soporta clientes HRT (*Hard Real-Time*) y SRT (*Soft Real-Time*). La implementación selecciona automáticamente las clases apropiadas para manejar los diferentes tipos de invocaciones. Para manejar invocaciones provenientes de clientes SRT se ha implementado el protocolo RMI-UDP desarrollado por Thales-Avionics. Para manejar las invocaciones provenientes de clientes HRT se ha usado una adaptación del protocolo de RMI.
- *Configurable*: dispone de clases de configuración que permiten especificar los parámetros asociados al protocolo de red y los asociados a RMI-HRT.
- *Mecanismos ante fallos*: en RMI, el servidor no puede tomar acciones ante excepciones o otros tipos de fallos. RMI-HRT introduce un par de métodos que son invocados si durante la ejecución de un método remoto una excepción es lanzada o

un plazo ha sido superado. Del lado cliente, RMI-HRT utiliza temporizadores para detectar posibles problemas.

RMI-HRT utiliza la serialización de objetos para construir la petición y respuesta de una invocación remota. Pero para mantener la predecibilidad del código se ha rediseñado el proceso de serialización el cual será comentado en la próxima sección.

3. Serialización

Los procesos de serialización traducen un objeto dado a un *stream* lineal de octetos, que pueden ser enviados a través de un *socket*, almacenados en un archivo, o manipulados simplemente como un *stream* de datos. La desserialización permite construir un objeto a partir de un *stream* lineal de octetos. El objeto puede ser tan simple como un número entero o tan complejo como una estructura enlazada.

La especificación de Java para la serialización de objetos bosqueja un algoritmo de serialización, define una arquitectura de clases e interfaces, y un protocolo que especifica la estructura necesaria para representar objetos en un *stream*.

El proceso de serialización indicado en la especificación e implementado por Classpath [5] es invocado en tiempo de ejecución y corre dentro de la hebra que realiza la invocación. El núcleo del algoritmo sigue los pasos siguientes (el proceso de des-serialización es similar):

- Si el objeto es un *string* (*java.lang.String*), el octeto TC_STRING, la longitud del *string* y el propio *string* codificado en UTF-8 es escrito al *stream*.

- Si el objeto es un *array*, el octeto TC_ARRAY, el descriptor del *array*, la longitud del *array*, y cada uno de los elementos del *array* son escritos al *stream*.

- Si es un objeto regular, el octeto TC_OBJECT, el descriptor para la clase del objeto, y todos los campos del objeto son escritos al *stream*. La estrategia es buscar de forma ordenada todos los campos del objeto y transformarlos en una secuencia de octetos. Se comienza desda los campos de la primera super-clase no serializable hasta los campos de la clase en cuestión y dentro de cada clase los campos son ordenados de forma canónica. En la implementación de Classpath, un único método es invocado recursivamente para escribir cada uno de los campos del objeto.

La ventaja principal de este enfoque es que puede manejar cualquier tipo de clase serializable. Sin embargo, tiene un número de limitaciones que impiden su utilización en STR:

- No hay un mecanismo para acotar el largo del *string*, según el proceso típico el largo es conocido en tiempo de ejecución. Por lo tanto, no es posible calcular cuanta memoria será utilizada y cuanto tiempo consumirá la escritura del *string* al *stream*

- No hay mecanismos para determinar de antemano cual será el tamaño del *array*. Igual que antes dificulta obtener estáticamente el WCET y el uso de memoria.

- Para obtener información del objeto se crea un descriptor, es decir, un objeto del tipo `ObjectStreamClass`. Dicho objeto permite entre otras cosas ordenar los campos a ser escritos al *stream*. La creación y utilización de dicho descriptor consume memoria y tiempo de CPU.

- Los valores primitivos son escritos al *stream* sin mayores dificultades, pero las referencias a otros objetos son escritos al *stream* invocando recursivamente al mismo algoritmo. Acotar esta recursividad es una tarea compleja y prácticamente imposible cuando los objetos tienen estructuras enlazadas, y los tamaños de los objetos varían en tiempo de ejecución. Cuando un objeto hace referencia a otro objeto y dicho objeto hace referencia de forma directa o indirecta al primer objeto se crean bucles. Para limitar la recursividad y poder realizar análisis estáticos es necesario

prohibir dichos bucles (o introducir mecanismos que permitan acotarlos).

- La utilización de *reflection*, algoritmos recursivos y mecanismos de programación orientada a objetos hacen que el proceso de serialización sea demasiado complejo como para determinar el WCET y el uso de memoria.

- No es consistente con el perfil HRTJ, especialmente con el modelo de memoria.

Dadas estas limitaciones, no es factible utilizar las implementaciones disponibles en STR. Para superar este problema, un par de procesos de serialización fueron desarrollados:

- *Serialización de tiempo real:* Es una adaptación de la implementación de Classpath. Los cambios incorporan el modelo de memoria del perfil HRTJ.

- *Serialización predecible:* Es completamente un nuevo proceso de serialización que es consistente con el perfil HRTJ, y optimiza el WCET y el uso de memoria.

3.1. Serialización de tiempo real

La serialización de tiempo real fue pensada para interactuar con STR acríticos donde los requisitos temporales suelen ser más flexibles. El uso de una derivación de Classpath hace más fácil la inter-operabilidad con otras implementaciones. Es consistente con el perfil HRTJ y puede serializar cualquier tipo de clase. Todos los objetos temporales son creados dentro de una memoria restringida los cuales son eliminados una vez finalizada la invocación remota. Aunque puede ser posible limitar el WCET y uso de memoria, obtener valores optimos no ha sido el objetivo de este enfoque.

3.2. Serialización predecible

El objetivo de la serialización predecible es proporcionar medios a RMI-HRT para serialización objetos de forma predecible en tiempo y espacio. Los requisitos que fueron identificados para esta funcionalidad son:

- *Adecuado para aplicaciones HRT:* esto implica ajustarse al perfil HRTJ y a las técnicas de análisis estático.

- *Minimizar el uso de recursos:* los STR disponen de pocos recursos (principalmente los sistemas empotradas), por lo que, soluciones que minimicen el uso de memoria y CPU tienen ventajas sobre otras soluciones.

- *Determinar el uso de recursos:* los mecanismos propuestos deben permitir calcular el WCET y determinar la máxima memoria necesaria para ejecutar la aplicación.

- *Determinar el ancho de banda:* para determinar cuál es el ancho de banda necesario para realizar una determinada invocación remota es fundamental conocer de antemano cuántos octetos deben ser enviados y recibidos por la red. Por consiguiente, el proceso de serialización debe especificar cuántos octetos ocupa la serialización de un objeto dado. Una vez conocido el ancho de banda disponible y la cantidad de información a transmitir, es posible calcular los tiempos de transmisión.

- *Interoperabilidad de RMI:* se requiere la interacción con otros objetos remotos non-HRT. Por lo tanto, la serialización propuesta debe ser consistente con el protocolo subyacente definido en la especificación.

- *Soportar un amplio conjunto de clases:* se requiere soportar, por lo menos, las clases más usadas en STR.

Una forma de cumplir con todos estos requisitos es tener en tiempo de ejecución un código lo más simple posible. Un código principalmente secuencial, sin recursión, en lo posible sin bucles (solo bucles acotados de antemano) y sin mecanismos que introduzcan incertidumbre. Las aplicaciones HRT deben ser estáticamente analizables. Todas las clases y hebras

utilizadas en tiempo de ejecución son conocidas de antemano. El enfoque propuesto de serialización aprovecha este hecho para realizar tanto trabajo como sea posible en tiempo de compilación. Por lo tanto, el proceso de serialización ha sido divido en dos fases, una que se ejecuta en tiempo de compilación y otra en tiempo de ejecución. En la primera se lleva a cabo gran parte de la complejidad de la serialización, consecuentemente, se evita la generación de objetos temporales y se reduce el uso de la CPU en el tiempo de ejecución. En la segunda fase se ejecuta un código prácticamente secuencial, el cual contiente de forma ordenada casi toda la información de los objetos a serializar.

Una nueva versión del compilador de RMI-HRT (rmic) es el encargado de realizar la primera fase de la serialización. Por lo tanto, es el responsable de integrar el proceso de serialización predecible en RMI-HRT. El compilador sigue una estrategia "basada en clases". Por cada clase implicada en una invocación remota (por cada clase a serializar) genera una clase para la serialización y otra para la des-serialización (las clases creadas por el compilador son llamadas clases de serialización). Dichas clases están formadas principalmente por el método estático `writeObject` / `readObject`, el cual contiene un conjunto de métodos que escriben a / leen de un *stream* una secuencia de octetos que representan la serialización / des-serialización de un objeto determinado. En tiempo de ejecución, cuando se requiere serializar / des-serializar una instancia de una clase determinada se invoca el método `writeObject` / `readObject` de las clases de serialización para la clase en cuestión.

El compilador calcula cuántos octetos puede ocupar la representación de un objeto, ésto es factible dado que el conjunto de clases a ser usadas son conocidas de antemano. El *stub* y el *skeleton* (los intermediarios entre la aplicación y RMI-HRT) son creados según esos valores. Por consiguiente, RMI-HRT conoce por adelantado cuántos octetos requiere una petición y su respuesta. El núcleo de RMI-HRT utiliza internamente estos datos para crear *buffers* internos con un tamaño óptimo.

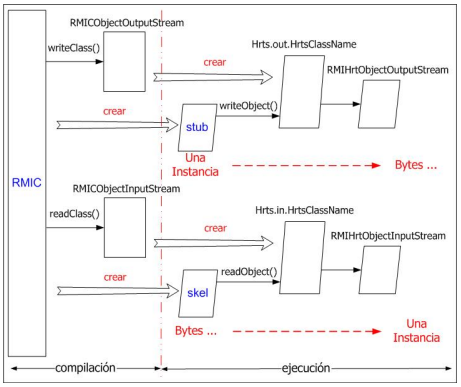

Figura 1: Clases de la serialización predecible

Dichos valores son también usados para determinar el tamaño de los paquetes que serán enviados por la red. Por lo tanto, esta información se puede utilizar para determinar el ancho de banda requerido y calcular el peor tiempo de transmisión. Esta información es necesaria para obtener el tiempo de respuesta de extremo a extremo de una transacción distribuida.

3.3. Detalles de implementación

La Figura 1 representa las clases que participan en el proceso de serialización. *rmic* y las clases `RMICObjectOutputStream` y `RMICObjectInputStream` se utilizan en la primera fase de la serialización predecible. Las clases `RMICObjectOutputStream` y `RMICObjectInputStream` proporcionan métodos que obtienen información de las clases a serializar. En tiempo de compilación, *rmic* usa los métodos `writeClass` y `readClass` de las clases mencionadas para crear las clases de serialización. Los métodos `writeClass` y `readClass` incluyen como primer parámetro el tipo de la clase a serializar. Cuando la clase representa un *string* o un *array*, el segundo parámetro es un *array* de tamaños que define los tamaños máximos para cada dimensión.

El *stub*, el *skeleton*, las clases de serialización, y las clases `RMIHrtObjectOutputStream` y `RMIHrtObjectInputStream` forman la

segunda fase de la serialización predecible, y por ende se utilizan en tiempo de ejecución.

Las clases `RMIHrtObjectOutputStream` y `RMIHrtObjectInputStream` son clases generales que incluyen métodos generales tales como los usados para transformar valores primitivos en una secuencia de octetos, y son usadas por las clases de serialización. El *stub* y el *skeleton* usan los métodos `writeObject` y `readObject` de las clases de serialización al comenzar la segunda fase de la serialización. El `writeObject` incluye como parámetro el objeto a serializar, y `readObject` retorna la des-serialización de un objeto.

A groso modo, los métodos `writeObject` y `readObject` de las clases de serialización permiten escribir / leer al stream un encabezado, un descriptor, y una secuencia de octetos que representa a todos los campos serializables de una clase.

Tipos de objetos
La estructura de un objeto puede ser muy compleja. Por ejemplo, un objeto puede incluir referencias a otros objetos y formar una lista enlazada. Si no hay límites a la complejidad de este tipo de objetos, no es posible proporcionar un proceso de serialización que obtenega un WCET razonable. Por esta razón, ha sido necesario restringir los tipos de objetos que pueden ser serializados de forma predecible:

- Los objetos no pueden incluir referencias recursivas.
- Las clases deben tener el constructor por defecto y todos los campos serializables del tipo *string* o *array* (de cualquier dimensión o de cualquier tipo) deben ser inicializados en dicho constructor. Así *rmic* puede determinar su tamaño máximo.
- Las clases deben implementar la interfaz `HRTSerializable`.

Uso de memoria
Las clases de serialización creadas por *rmic* consumen una cantidad fija de memoria (el valor depende de cada clase), por lo tanto, una vez cargadas en memoria ocupan un valor fijo y conocido. El método `writeObject`, el cual implementa la serialización en tiempo de ejecución, no consume memoria. Por esta razón, la serialización de cualquier cantidad de instancias de una clase no consume más memoria que la calculada previamente. El método `readObject`, el cual implementa la des-serialización en tiempo de ejecución, crea nuevos objetos para reconstruir el objeto serializado previamente. La implementación ha sido desarrollada teniendo en cuenta que los objetos son creados en memoria restringida, y entre des-serializaciones (entre invocaciones) los objetos son liberados. Por consiguiente, la des-serialización de un objeto ocupa como máximo lo que ocuparía una instancia donde todos sus campos contienen sus valores máximos, es decir, las campos primitivos tienen cualquier valor, los *strings* y los *arrays* tienen las máximas dimensiones y las referencias apuntan a instancias cuyos campos también tienen sus valores máximos. Por las restricciones impuestas a los objetos, la cantidad de memoria consumida en la des-serialización está acotada y puede calcularse.

Peor tiempo de ejecución
Para determinar el WCET sólo hace falta analizar los métodos `writeObject` y `readObject` de las clases de serialización. Dichos métodos presentan una estructura secuencial, sin recursión, y sin bucles no acotados, donde la acciones más complejas son leer un campo de un objeto, crear un objeto, y escribir un campo de un objeto. Por lo tanto, con poco esfuerzo es posible obtener el WCET.

4. Validación en un caso industrial

Thales Avionics ha sido el socio industrial a cargo de comprobar y validar las herramientas desarrolladas dentro del proyecto europeo HIJA [4]. RMI-HRT y la serialización predecible han sido utilizados para soportar parte de las comunicaciones de la implementación de un FMS (*Flight Management System*). El gestor del vuelo necesita transmitir y actualizar datos del vuelo en todo el sistema distribuido, para eso se ha definido la clase `AircraftData`. Una versión simplificada

contiene cinco campos de tipo *double*, que hacen referencia a la velocidad, al peso, la latitud, la longitud y la altitud del avión. El código mostrado abajo incluye parte de las clases de serialización. Contiene un conjunto de métodos que escriben el descriptor de la clase `AircraftData` al *stream* (en el ejemplo, se puede llegar a utilizar hasta 137 octetos del `stream`). Como se puede apreciar se usan algunos métodos de la clase `RMIHrtObjectOutputStream`, para obtener y escribir al *stream* cada uno de los campos de la clase.

```
// Descriptor para la clase AircraftData:
out.writeByte(114);          // TCCLASSDESC
out.writeUTF(AircraftData);// CLASS NAME
out.writeLong(...);          // UID
out.writeByte(2);            // FLAGS
out.writeShort(5);           // FIELD COUNT
out.writeByte('D');          // TYPE CODE 0
out.writeUTF("alt");         // FIELD NAME0
out.writeByte('D');          // TYPE CODE 1
out.writeUTF("latitude");    // FIELD NAME1
out.writeByte('D');          // TYPE CODE 2
out.writeUTF("longitude"); // FIELD NAME2
out.writeByte('D');          // TYPE CODE 3
out.writeUTF("tas");         // FIELD NAME3
out.writeByte('D');          // TYPE CODE 4
out.writeUTF("weight");      // FIELD NAME4
out.writeByte(120);          // TCENDBLOCKDATA
out.writeByte(112);          // TCNULL
// El descriptor consume: 137 bytes
// Escribiendo campos de AircraftData:
out.writeDouble(out.getDoubleField(obj, cl, alt));
out.writeDouble(out.getDoubleField(obj, cl, latitude));
out.writeDouble(out.getDoubleField(obj, cl, longitude));
out.writeDouble(out.getDoubleField(obj, cl, tas));
out.writeDouble(out.getDoubleField(obj, cl, weight));
// La solicitud consume 487 bytes (RMI-HRT).
// La respuesta consume 6 bytes (RMI-HRT).
```

RMI-HRT proporciona mensajes sobre el consumo de memoria. En este ejemplo, el consumo de memoria inmortal es 0, y de memoria restringida es 46272 octetos. Serializar y des-serializar la versión completa del `AircraftData` consume cerca de 3 milisegundos en un Pentium IV de 3.20 GHz.

5. Trabajos relacionados

Real-Time CORBA [11] ha sido creado para SDTR; sin embargo, es silencioso respecto a la serialización de objetos. Hay varios ORBs que incluyen procesos de serialización. No obstante, la mayoría están centrados en STR acríticos. Además, la información de cómo realizan la serialización no es pública.

El último informe sobre DRTSJ reconoce que algunas clases de RTSJ tales como `HighResolutionTime` y `PriorityParameters` deben ser marcadas como `Serializable` para facilitar el comportamiento distribuido. También identifica que el protocolo de RMI incluye la serialización de objetos. Sin embargo, una serialización de tiempo real con una gestión de memoria adecuada son dejados como un valor agregado de cada implementación.

6. Conclusión

Este artículo introduce un nuevo proceso de serialización enfocado a SDTR. El nuevo proceso mejora el comportamiento determinista de las implementaciones actuales. Por construcción, el nuevo enfoque garantiza que el código obtenido tiene las siguientes características: i) puede ser ejecutado en cualquier máquina virtual con el perfil HRTJ (por consiguiente, incluye un modelo de concurrencia y una gestión de memoria adecuada), ii) la cantidad de memoria consumida está limitada y puede calcularse previamente, iii) el número de octetos enviados al *stream* está limitado y puede calcularse de antemano, iv) el peor tiempo de ejecución está acotado y puede ser calculado fácilmente de antemano. Aunque quedan algunas mejoras futuras, la serialización predecible proporciona mecanismos para construir aplicaciones distribuidas de tiempo real crítico.

7. Agradecimientos

Este trabajo ha sido apoyado por la Comunidad de Madrid, el Fondo Social Europeo y el proyecto Europeo HIJA, N° IST-511718. Los autores quieren agradecer el apoyo recibido.

Referencias

[1] Bollella, G., *The Real-Time Specification for Java*, G. et. al, 2000. Available from:

http://www.rtsj.org

[2] Borg, A., *A real-time RMI framework for the RTSJ*, y Wellings, A.,In Proceedings of the 15th Euromicro Conference on Real Time Systems. Euromicro, IEEE, July 2003.

[3] Burns, A., *The Ravenscar tasking profile for high integrity real-time programs*, Dobbing, B., Romanski G., In proceedings of Ada-Europe, 1411:263-275,June 1998.

[4] Erik YuShing Hu, *Safety Critical Applications and Hard Real-Time Profile for Java: A Case Study in Avionics*, Eric Jenn, Nicolas Valot, Alejandro Alonso, JTRES'06 Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems. Paris, France, 11-13 October 2006.

[5] GNU Classpath, *GNU Classpath: Essential Libraries for Java*, Available from: http://www.gnu.org/software/classpath/.

[6] Gutiérrez García, J. J., *Schedulability Analysis of Distributed Hard Real-Time Systems with Multiple-Event Synchronization*, Palencia Gutiérrez, J.C., y González Harbour, M., In Proceedings of 12th Euromicro Conference on Real-Time Systems, pages 15-24, Stockholm (Sweden), June 2000. IEEE Computer Society Press.

[7] Hetcht H., *Review guidelines for software languages for use in nuclear power plant systems*, Hecht M., S. Graff et at., U.S. Nuclear Regulatory Commission, (NUREG/CR-6463), 1997.

[8] *High Integrity Java*, 2004, Available from: http://www.hija.info

[9] Jonathan S. Anderson, E, *Distributed Real-Time Specification for Java. A Status Report (Digest)*, Douglas Jensen, JTRES'06 Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems, Paris, France, 11-13 October 2006.

[10] Jsr-50, *Jsr-50: Distributed real-time specification*, Available from: http://www.jcp.org/en/jsr/detail?id=050

[11] Object Management Group, *Dynamic scheduling real-time CORBA 2.0 (joint revised submission)*, Object Management Group orbos/2001-04-01 ed., 2001.

[12] Palencia Gutiérrez, J. C. *On the Schedulability Analysis for Distributed Hard Real-Time Systems*, Gutiérrez García, J. J. y González Harbour, M, In Proceedings of the 9th Euromicro Workshop on Real Time Systems, 1997.

[13] Sun Microsystems, *Java Object Serialization Specification*, Available from: http://java.sun.com/j2se/1.5.0/docs/\guide/serialization/

[14] Sun Microsystems, *Java Remote Method Invocation Specification*, Available from: http://java.sun.com/j2se/1.5.0/docs/\guide/rmi/

[15] Tejera, D., *Two Alternative RMI Models for Real-Time Distributed Applications*, Tolosa, R., de Miguel, M.A., and Alonso, A., IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), Pages 390-397, Seattle, Washington, USA, May 2005.

[16] Tejera, D., *Predictable Serialization in Java*, de Miguel, M.A., and Alonso, A., Tenth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007) Pages 102-109,Santorini, Grecia, Mayo 2007.

[17] Wellings, A., *A framework for integrating the real-time specification for java and java's remote method invocation*, Clark, R., Jensen, D,. y Wells, D., In Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pages 13-22, 2002.