CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Herramienta autoconfigurable para el desarrollo de aplicaciones distribuidas de tiempo real flexibles y dinámicas

Tesis que presenta:
Christian Iván Mejía Escobar

Para obtener el grado de:
Maestro en Ciencias

En la especialidad de:
Ingeniería Eléctrica

Opción Computación

Director de tesis:
Dr. José Guadalupe Rodríguez García

México, D.F. Noviembre 2007

Resumen

La evolución experimentada por los sistemas de computación, desde centralizados a distribuidos, ha motivado el surgimiento de aplicaciones que demandan mayor complejidad, seguridad, disponibilidad, escalabilidad, *etc*. Cada vez se hace más necesario la intervención de sistemas distribuidos, donde varias computadoras conectadas por medio de una red trabajan conjuntamente para satisfacer las necesidades anteriormente mencionadas.

El paradigma de *programación orientada a objetos* ha demostrado ser exitoso para el desarrollo de aplicaciones, por lo que es utilizado ampliamente en ambientes distribuidos. Un *middleware* como *Java RMI* permite el desarrollo y la ejecución de aplicaciones distribuidas orientadas a objetos de propósito general; sin embargo, existen aplicaciones especializadas, cuya ejecución debe considerar parámetros de planificación y restricciones de tiempo. Este tipo de aplicaciones se presentan en el dominio de tiempo real y su utilización es cada vez más común en sistemas distribuidos.

A pesar de que *Java* no fue diseñado para aplicaciones de tiempo real, posee ventajas como simplicidad y portabilidad, consideradas muy prometedoras por la comunidad de tiempo real. Por tal razón, se creó la especificación de tiempo real para *Java* (*RTSJ*), pero se enfoca intencionalmente para trabajar en un ambiente centralizado, ***i.e.*** no provee soporte para aplicaciones distribuidas.

En el presente trabajo de tesis proponemos un mecanismo desarrollado en lenguaje *Java* que permite propagar los parámetros de planificación y las restricciones de tiempo de aplicaciones distribuidas entre los nodos que conforman un sistema distribuido. Nuestra solución aprovecha la actual *RTSJ*, combinada con herramientas de *Java* tales como: *API* de *sockets*, el modelo de *Java RMI*, carga dinámica de *bytecodes*, y programación *multi-hilo*.

Abstract

The development on computing systems, from centralized to distributed systems, has made possible a new kind of applications, with more complexity and that require higher security, availability, scalability, *etc*. Distributed systems are more and more useful. They are characterized by having multiple computers connected through a network, working together and collaborating to meet the requirements mentioned above.

The paradigm of *object-oriented programming* has proved to be successful for the development of software and today it is widely used in distributed environments. A *middleware* like *Java RMI* enables the development and execution of object-oriented general purpose distributed applications. For special purpose applications, *e.g.* real time applications, with scheduling parameters and time constraints, *Java RMI* is not the must suitable tool, however such type of applications are increasingly common in distributed systems.

Java was not designed for real time applications, but it has some advantages such as portability and simplicity, which are considered very promising by the real time community. For this reason, the *Real-Time Specification for Java* (*RTSJ*) was created, but it is focused on centralized systems, *i.e.* there is not support for distributed applications.

In this thesis, we propose a mechanism developed in *Java* that allows the propagation of scheduling parameters and time constraints of distributed applications. This transfer of context is carried out among the nodes of distributed system. Our solution takes advantage of the current *RTSJ*, combined with *Java* tools such as *API sockets*, *Java RMI* model, *dynamic loading*, and multithreading programming.

Agradecimientos

A mis padres, Luis Enrique y Mariana de Jesús.

A mi hermana, Mónica Gabriela.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT)
por el respaldo financiero otorgado a través del proyecto número 45306.

Al Dr. José Guadalupe Rodríguez García (Director de tesis),
al Dr. Francisco Rodríguez Henríquez (Coordinador Académico),
y a todos los investigadores que conforman el
Departamento de Computación del CINVESTAV.

Índice general

Lista de figuras

Lista de tablas

Convenciones tipográficas

En el presente documento, para el texto regular hemos utilizado un tipo de fuente Times New Roman; sin embargo, ciertas partes del texto aparecen con un estilo tipográfico distinto. A continuación enunciamos estos casos:

Times New Roman (estilo cursiva) es utilizada para:

- La palabra o conjunto de palabras (español/inglés) que nombran un concepto, un programa, un dispositivo, un sistema, un estándar, *i.e.*, cualquier término técnico.
- Acrónimos (español/ inglés) y su correspondiente significado.
- Acrónimos de expresiones en latín como: *i.e.*, *e.g.*, *etc*.
- Frases textuales.

Times New Roman (estilo negrita) para:

- Títulos de capítulos y secciones.
- Títulos de tablas y figuras.
- Resaltar una idea importante.

`Courier New` es utilizada para:

- Nombres de variables, constantes, clases, métodos, código, y comandos.

Capítulo 1

Introducción

El aumento en la capacidad de procesamiento de las computadoras personales, el desarrollo de las redes de comunicaciones, el auge de *Internet*, el abaratamiento del hardware de cómputo, así como el continuo interés por compartir los recursos computacionales, son factores que han propiciado un cambio sustancial en el paradigma de computación, desde un enfoque centralizado a uno distribuido. Así, el término *sistema de computación*, inicialmente referido como una computadora trabajando independientemente, ahora encierra el trabajo cooperativo de varias computadoras.

Actualmente, la implementación de sistemas de computación difícilmente se limita al uso de una sola computadora. Tales sistemas demandan cualidades y capacidades como: complejidad, rendimiento, seguridad, disponibilidad, escalabilidad, *etc.*, características que pueden ser cubiertas por un *sistema distribuido*, donde intervienen varias computadoras que interactúan para alcanzar una meta en común.

En el ámbito de *Internet*, las *intranets*, así como en la *computación móvil* y *ubicua*, podemos encontrar sistemas distribuidos muy populares. El *World Wide Web* (o simplemente *Web*), el correo electrónico (*e-mail*), la transferencia de archivos (*FTP*, *File Transfer Protocol*), el sistema de nombres de dominio (*DNS*, *Domain Name System*), las telecomunicaciones (audio y video), los sistemas de reservación de boletos y las transacciones bancarias, son algunos de los ejemplos de este tipo de sistemas.

Los sistemas distribuidos brindan el soporte necesario para el desarrollo de aplicaciones distribuidas. La *programación orientada a objetos* (*POO*) ha demostrado ser uno de los modelos más exitosos en los últimos años para el desarrollo de aplicaciones debido a características y ventajas como: herencia, polimorfismo, reutilización, modularidad, y fácil mantenimiento [1].

Por tales razones, el modelo de programación basado en objetos ha sido utilizado para la implementación de *sistemas distribuidos orientados a objetos* hasta el punto de que la mayoría del software de sistemas distribuidos, actualmente está desarrollado con lenguajes orientados a objetos [2]. Las aplicaciones distribuidas están conformadas por *objetos* que pueden estar ubicados en diferentes computadoras y que cooperan entre sí para lograr toda la funcionalidad de una sola aplicación como un todo unificado. Estos objetos tienen el mismo significado que el de *objeto* en *POO* estándar y poseen todas las características de este paradigma de programación.

Las aplicaciones distribuidas necesitan de componentes denominados *middlewares* para superar los detalles de heterogeneidad, tanto de sistemas operativos como arquitecturas de hardware, algo característico de los ambientes distribuidos. Uno de los *middlewares* más populares es *Java RMI*[1] (*Java Remote Method Invocation*), basado en objetos y que facilita la comunicación a través de invocaciones de métodos y devolución de resultados mediante el envío y la recepción de mensajes [3].

Java RMI fue desarrollado por *Sun Microsystems* únicamente para el lenguaje de programación *Java* y, en la actualidad, tiene una creciente popularidad en *Internet*. Esta tecnología permite la implementación de aplicaciones distribuidas de propósito general; sin embargo, **¿Qué sucede con aplicaciones especializadas cuya ejecución debe ser realizada considerando parámetros de planificación[2] y restricciones de tiempo[3]?** Estos requerimientos caracterizan una *aplicación de tiempo real* y determinan su *contexto de ejecución*. Para este tipo de aplicaciones, *Java RMI* no ofrece soporte de ejecución ni de desarrollo.

En los últimos años, los desarrolladores de aplicaciones de tiempo real han dirigido su atención hacia *Java* no sólo porque ofrece las ventajas inherentes al paradigma de *POO*, sino que proporciona características sobresalientes como: simplicidad, portabilidad y soporte incorporado para programación *multi-hilo* (*multi-threading*), todas muy útiles desde la perspectiva de la ingeniería del software.

A pesar de su gran potencial, *Java* (y por ende *Java RMI*) no fue diseñado para desarrollo de aplicaciones de tiempo real y, por tanto, no es adecuado para este dominio, principalmente debido a dos inconvenientes: 1) bajo rendimiento por ser un lenguaje interpretado (*bytecodes*) y 2) falta de predictibilidad (indeterminismo) en tiempo de ejecución ocasionada por la gestión dinámica de memoria (recolector de basura).

La comunidad de *Java* ha trabajado para afrontar las limitaciones mencionadas y en 1999 comenzó a desarrollar **la *Especificación de Tiempo Real para Java* (*RTSJ, Real-Time Specification for Java*)**. Como resultado de tal especificación, se dispone de una máquina virtual de *Java* de tiempo real (*JVM-RT*) así como también de una *API* (paquete `javax.realtime`) para el desarrollo de aplicaciones de tiempo real. Sin embargo, ambas son herramientas **diseñadas exclusivamente para trabajar en sistemas centralizados**.

1.1 Motivación

En nuestros días las aplicaciones tienden hacia la distribución, de la misma manera que las aplicaciones de tiempo real tienden cada vez más a la descentralización. En el dominio de tiempo real, el uso de aplicaciones distribuidas se ha incrementado considerablemente. Podemos citar el caso de las aplicaciones para sistemas de automatización industrial, control de tráfico aéreo, aviónica, control en automóviles, telecomunicaciones, multimedia, *etc*.

1 Utilizamos *Java RMI* o *RMI* indistintamente
2 *e.g.* prioridad, planificador, política de planificación
3 *e.g.* plazo, periodo, tiempo de cómputo

Estas aplicaciones se caracterizan por tener asociadas diversas **restricciones de planificación y de tiempo** que **deben ser cumplidas en los nodos involucrados** del sistema distribuido. Así, la correcta ejecución no sólo depende de los resultados lógicos del procesamiento sino también de los instantes de tiempo en los cuales se producen dichos resultados.

La comunidad de tiempo real ha puesto especial atención en la tecnología *Java* como un ambiente de desarrollo y ejecución para sistemas de tiempo real, aprovechando ventajas como la portabilidad, concurrencia y simplicidad. *Java* es ampliamente utilizado en *Internet*, principalmente en los dominios de negocios y escritorio [44]. Gracias a *Java RMI*, es posible proveer la infraestructura necesaria para el desarrollo y la ejecución de aplicaciones distribuidas convencionales. Cuando tratamos con aplicaciones distribuidas de tiempo real, un *middleware* como ***Java RMI* no proporciona el soporte necesario,** pues al igual que *Java*, no fue diseñado con este propósito. Con el surgimiento de la *RTSJ*, una extensión de tiempo real para *Java*, es posible desarrollar aplicaciones de tiempo real pero que están confinadas en una sola computadora.

Hace algunos años comenzó el desarrollo de una especificación distribuida de la *RTSJ*, denominada *DRTSJ* (*Distributed Real-Time Specification for Java*), cuyo propósito es producir un *middleware de tiempo real* basado totalmente en la tecnología *Java*. Una de las aproximaciones es crear una versión de tiempo real de *Java RMI*, que trabaje sobre las implementaciones de la *RTSJ* y así proveer la plataforma necesaria para *sistemas distribuidos de tiempo real* [7]. Hasta el momento se han explorado diversas formas en las que *Java RMI* y la *RTSJ* podrían ser integradas y aunque se han establecido guías generales para evitar modificaciones sustanciales a la sintaxis del lenguaje, aún no es posible emitir una especificación.

La construcción de un *middleware de tiempo real* es un proyecto muy complejo que implica la colaboración de un sinnúmero de personas y requiere de tiempo incalculable. Entre sus principales funciones constan: asignar, controlar y planificar los recursos de procesador, memoria y red de comunicaciones, con el fin de asegurar la predictibilidad en entornos distribuidos [44]. Nuestro interés está enfocado en resolver una de las funcionalidades que debería satisfacer un *middleware Java* de tiempo real: **la propagación del contexto de ejecución de los hilos de una aplicación de tiempo real, basada en *Java*, entre los nodos del sistema distribuido**.

La solución estará basada en la *RTSJ* actual que, integrada con diversas herramientas de Java disponibles gratuitamente, proporcionarán la herramienta de desarrollo y de ejecución para las aplicaciones distribuidas de tiempo real. De tal forma, un desarrollador que desea implementar este tipo de aplicaciones en lenguaje *Java*, puede utilizar nuestra herramienta de desarrollo como esquema básico para su implementación particular.

1.2 Objetivo

La ejecución de aplicaciones de tiempo real en un entorno distribuido demanda la propagación de los parámetros de planificación y las restricciones de tiempo (contexto de ejecución) entre los nodos del sistema distribuido. En el presente trabajo de tesis proponemos **un mecanismo basado en *Java* que permita el paso del contexto de ejecución entre los nodos involucrados** y, de esta

manera, mantener el cumplimiento de las restricciones temporales de aplicaciones distribuidas, aprovechando herramientas de *Java* disponibles como:

- ***RTSJ***: la actual especificación de tiempo real para *Java*, que provee una *máquina virtual de Java de tiempo real* (*JVM-RT*) así como una *API*[4] de desarrollo para aplicaciones de tiempo real. A pesar de que es una herramienta diseñada para trabajar en una sola computadora, se convierte en el punto de partida para nuestra solución.

- ***API* de *sockets***: incluida en *Java* estándar para brindar soporte a la comunicación en red. Permitirá manejar la comunicación e intercambio de información entre diferentes instancias de la *JVM-RT*.

- **Modelo de *Java RMI***: es el modelo de programación que *Java* incluye para sistemas distribuidos. Permite llevar a cabo la *programación orientada a objetos* en entornos distribuidos. Aunque no ofrece soporte para aplicaciones de tiempo real, el principio sobre el cual trabaja y sus entidades fundamentales (*cliente*, *servidor*, e *interfaz remota*), serán la base para nuestra arquitectura de solución.

- **Programación *multi-hilo***: *Java* es un lenguaje de programación concurrente, así que podremos representar la concurrencia natural de los sistemas de tiempo real por medio de hilos independientes de ejecución. Además, la organización de sistemas distribuidos en términos de clientes y servidores, hace que un servidor atienda a varios clientes simultáneamente o, en caso de existir algún fallo en una conexión de cliente, no afecte a las demás.

- ***Reflectividad y carga dinámica***: son dos utilidades que ofrece *Java* para obtener información de una determinada clase y cargar los *bytecodes* de la misma en tiempo de ejecución. De esta manera pretendemos dar un enfoque dinámico a la solución, ya que el cliente puede enviar todos los componentes que el servidor necesita para la ejecución.

La integración de todas estas herramientas dará como resultado el mecanismo para paso de contextos. Además, proveerá una herramienta de desarrollo y una plataforma de ejecución para aplicaciones *Java* con requisitos de tiempo en un ambiente distribuido.

Nuestra solución debe cumplir con las siguientes características: **generalidad**, **flexibilidad**, **versatilidad** y **adaptabilidad**.

Puesto que hacemos énfasis en el diseño y la ejecución de aplicaciones, usamos el término aplicaciones distribuidas de tiempo real en lugar de sistemas distribuidos de tiempo real. Un *sistema distribuido de tiempo real*, tal como un sistema distribuido convencional, está compuesto por un conjunto de computadoras conectadas en red, donde se ejecutan aplicaciones con restricciones temporales.

4 *Interfaz de programación de aplicaciones*, colecciones de clases en las bibliotecas o paquetes de clases de *Java* [43]

1.3 Trabajo relacionado

La *RTSJ* está diseñada para plataformas centralizadas [4]. De tal forma, se trabaja en la creación de una especificación para la versión distribuida de *Java* denominada *Especificación de Tiempo Real Distribuida para Java* (*DRTSJ, Distributed Real-Time Specification for Java*) [5]. En [6] se resume el trabajo cumplido hasta la fecha por parte del grupo de expertos en cuanto al desarrollo de la *DRTSJ*. El objetivo del proyecto es minimizar las modificaciones a la sintaxis de *Java* y a su especificación para tiempo real.

Una de las aproximaciones es el desarrollo de la versión distribuida de *RMI* (*RT-RMI*, *Real-Time RMI*). En [7] se exponen las formas en las que se podría integrar y extender la *RTSJ* existente con la facilidad *RMI* de *Java* para proveer la base de la *DRTSJ*. Un avance importante se describe en [8] donde se presenta un trabajo en curso para proveer un *framework* que soporte la predictibilidad en la invocación de métodos remotos. La solución es definida como un paso inicial hacia una especificación ya que la completa adaptación del modelo de *Java RMI* con la sintaxis y semántica de la *RTSJ* aún no está resuelta.

La meta de la *DRTSJ* es convertir a *Java* en una plataforma de desarrollo conveniente para sistemas distribuidos de tiempo real. Sin embargo, **integrar la reciente *RTSJ* con mecanismos de comunicación es un problema abierto y materia de investigación** [8]. Nuestro trabajo tiene como propósito describir un mecanismo que permita propagar requisitos de planificación y de tiempo de una aplicación distribuida entre los nodos de un sistema distribuido. Esta funcionalidad puede ser integrada con otras capacidades para disponer de un *middleware Java de tiempo real*.

1.4 Organización de la tesis

Este documento se encuentra estructurado de la siguiente manera: el capítulo 2 contiene la base teórica que sustenta el trabajo desarrollado incluyendo los conceptos fundamentales. El capítulo 3 examina las herramientas *Java* que han sido utilizadas para la solución del problema. El capítulo 4 expone la arquitectura de la solución, destacando los componentes que la conforman así como sus interacciones. El capítulo 5 describe la implementación del mecanismo propuesto mientras que el capítulo 6 detalla las pruebas realizadas así como los resultados obtenidos. Finalmente, en el capítulo 7 enunciamos las conclusiones del trabajo.

Capítulo 2

Sistemas distribuidos

En este capítulo examinamos los conceptos fundamentales acerca de sistemas distribuidos, la infraestructura que soporta la solución propuesta. Comenzamos con una síntesis de la evolución experimentada por los sistemas de computación y la motivación que provocó el paso de sistemas centralizados a distribuidos. Presentamos diversas definiciones de un sistema distribuido, identificando los elementos comunes que permiten caracterizarlo, sus ventajas y desventajas, así como algunos ejemplos muy conocidos. Luego, revisamos la arquitectura general de un sistema distribuido, basada inicialmente en el modelo *cliente-servidor* y posteriormente en el paradigma de *orientación a objetos*. Por último, abordamos el propósito de un *middleware*, las implementaciones más populares hoy en día, destacando *Java RMI* como el modelo para el desarrollo de nuestra solución.

2.1 Evolución

En sus inicios, los sistemas de computación se limitaban a programas mono-usuarios residentes en grandes computadoras. Luego, los *mainframes* de tiempo compartido podían atender a varios usuarios desde terminales remotas para el acceso a recursos centralizados. Con la llegada del *microprocesador* aparecen las computadoras personales (*PCs*) que, junto a las redes de área local (*LAN*, *Local Area Network*), permitieron la descentralización de los recursos. Con base en una *arquitectura cliente-servidor*, las aplicaciones se dividieron en partes que podían residir en computadoras distintas. El *boom* de *Internet*, la reducción en tamaño y costo del hardware de cómputo, el aumento en la capacidad de las computadoras personales y el desarrollo de las tecnologías de red, dan como resultado *sistemas distribuidos* con *PCs* más poderosas conectadas por medio de redes de alta velocidad. Una aplicación distribuida puede ejecutar partes de su código en distintos procesadores que coordinan la ejecución mediante mensajes. En los últimos años, la computación distribuida se ha consolidado y es la base para el desarrollo de tecnologías como *grids*, *clusters*, la comunicación *peer to peer* y las redes de dispositivos móviles.

2.2 Motivación

La principal motivación detrás de la consecución del paradigma de computación distribuida, se debe a la necesidad de compartir recursos. En computación, el término *recurso* comprende dispositivos de hardware (*e.g.* procesador, disco duro, impresora) así como entidades de software (*e.g.* archivos, bases de datos, secuencias de imágenes, audio de un teléfono móvil) [2]. En la actualidad, compartir recursos por medio de una red de computadoras es una característica tan familiar que es aprovechada en diversos ámbitos tales como: los negocios, la investigación, el gobierno y el hogar [9].

2.3 Definición

Es difícil capturar en una única definición las diversas facetas que presentan los sistemas distribuidos [10]. Enunciamos a continuación varias de las definiciones encontradas en la literatura:

> *"Un sistema distribuido es una **colección de computadoras** autónomas que **trabajan conjuntamente** para dar la apariencia de un **sistema coherente único.**"* [11]

> *"Un sistema distribuido es aquel en el que los **componentes de hardware o software**, localizados en computadores conectadas mediante **red**, comunican y **coordinan** sus acciones sólo mediante **paso de mensajes.**"* [2]

> *"Un sistema distribuido es una **colección** de **dispositivos individuales** de computación que pueden **comunicarse** unos con otros."* [9]

> *"Un sistema distribuido es un sistema compuesto de **varias computadoras** que se **comunican** mediante una **red**, almacenando procesos que utilizan un conjunto de protocolos distribuidos común para soportar la ejecución coherente de actividades distribuidas.*" [10]

> *"Un sistema distribuido es un sistema de procesamiento de información que contiene **múltiples computadoras** independientes que **cooperan** unas con otras sobre una **red** de comunicaciones para alcanzar un **objetivo específico**.*" [12]

Tal gama de definiciones obliga a identificar ciertos elementos en común que, en conjunto, permitirán caracterizar de mejor manera un sistema distribuido, así:

- Varias computadoras o procesos, denominados ***nodos***.
- Red de comunicaciones o de computadoras, denominada simplemente **red**.
- Intercambio de **mensajes** (coordinación).
- **Meta común**, *e.g.* compartir un recurso, brindar un servicio, o ejecutar una aplicación.
- Apariencia de un **sistema único**.

Cada una de las características mencionadas, por sí solas, no definen un sistema distribuido, *e.g.* una red de computadoras no es un sistema distribuido ya que las computadoras en una red quizá nunca interactúen, o están limitadas a recibir o enviar mensajes de manera ocasional [10].

Una diferencia importante es que los **procesos de computadoras en un sistema distribuido cooperan para alcanzar un objetivo común**. Cabe recalcar que, desde el punto de vista de los usuarios, el conjunto de recursos disponibles en un sistema distribuido actúa como un único sistema virtual [13].

2.4 Ejemplos

Los enormes progresos tanto en tecnología de computadoras como en redes de comunicaciones han guiado hacia ambientes de computación distribuida con una multitud de servicios [14]. En este contexto, *Internet* constituye una gran infraestructura para la construcción de sistemas distribuidos.

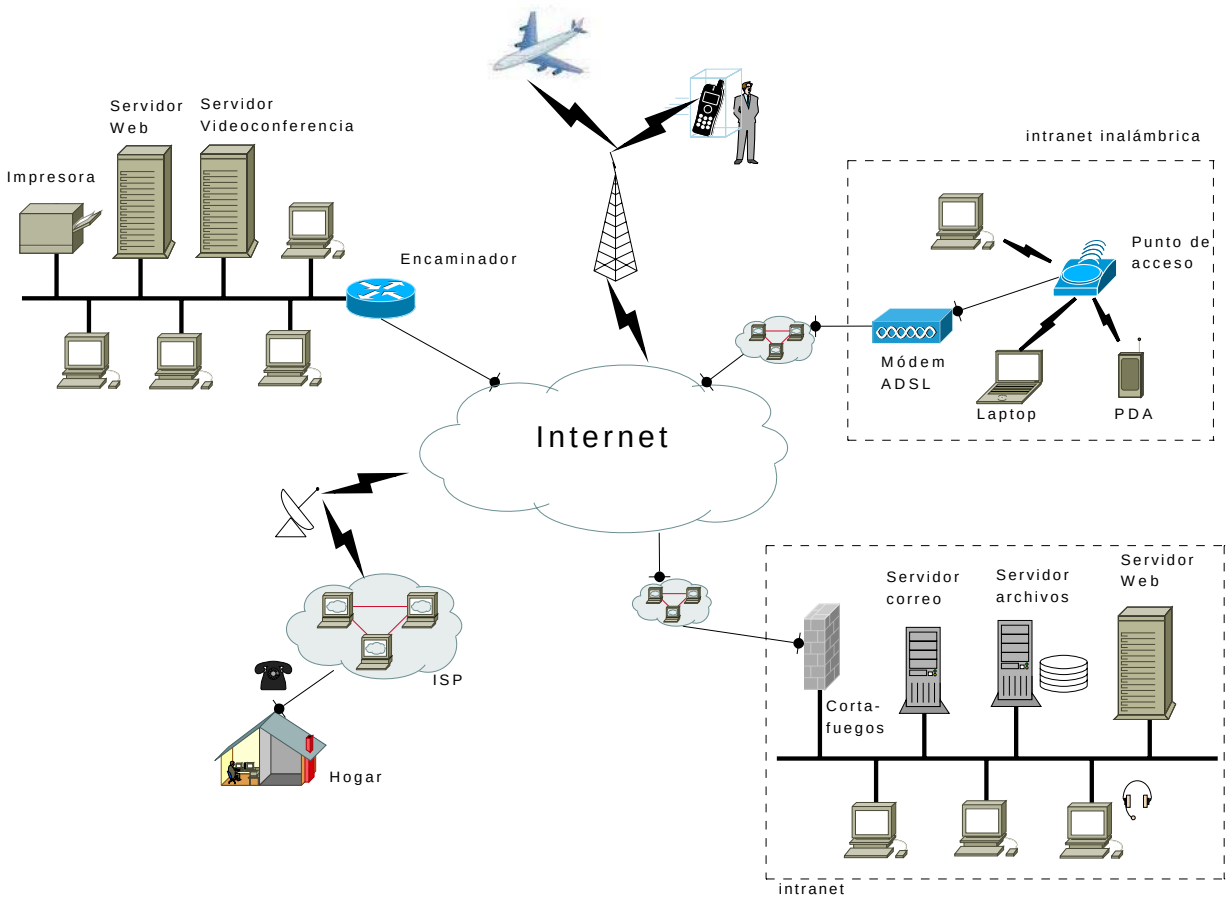


Figura 2.1: Sistemas distribuidos

En la figura 2.1 presentamos varios ejemplos de este tipo de sistemas, *e.g.* un servidor *Web* ubicado en la *intranet* de una organización (red privada con tecnología *Internet*) proporciona servicio tanto a usuarios que pueden estar dentro de la organización, fuera de ella o incluso desde el hogar a través de empresas que proveen acceso a *Internet* (*ISP*, *Internet Service Provider*). De

la misma forma, servicios como el correo electrónico, transferencia de archivos, multimedia e impresión, están disponibles para usuarios internos y externos. Los avances en cuanto a tecnología inalámbrica han permitido que dispositivos como computadoras portátiles (*Laptops*), *PDAs* (*Personal Digital Assistant*), teléfonos celulares y, hasta dispositivos integrados en aviones y automóviles, sean capaces de conectar a usuarios mientras ellos están en movimiento.

2.5 Ventajas y desventajas

El paso de sistemas centralizados a distribuidos ha sido el resultado de un proceso de evolución, guiado principalmente por la mentalidad de que compartir es mejor que duplicar. A través de este proceso, han surgido nuevas posibilidades además de poder compartir recursos tanto físicos como lógicos. Podemos mencionar el acceso a computadoras distantes geográficamente, la ejecución de código en computadoras remotas, la integración de varias computadoras con el fin de aumentar la capacidad de procesamiento y así poder resolver problemas complejos, *etc*.

La tabla 2.1 presenta una comparación entre sistemas distribuidos y sistemas centralizados, tomando en consideración diversos criterios que son importantes para decidir la utilización de una u otra tecnología. Hemos reunido la información encontrada en [10], [12], y [15]:

Criterio	***Sistemas centralizados***	***Sistemas distribuidos***
Acceso a recursos	Local	Global
Tecnología de los nodos	Homogénea	Heterogénea
Administración	Una organización (simple)	Varias (compleja)
Coste económico	Bajo	Alto
Modularidad	Un nodo	Varios nodos
Complejidad	Baja	Alta
Consistencia	Simple	Compleja
Escalabilidad	Restringida	Ilimitada
Seguridad	Alta	Baja
Rendimiento/Velocidad	Bajo	Alto
Riesgo de fallos	Bajo	Alto
Disponibilidad	Baja	Alta
Tolerancia a fallos	Baja	Alta

Tabla 2.1: Ventajas y desventajas en sistemas centralizados y sistemas distribuidos

Podemos apreciar que los sistemas distribuidos ofrecen una serie de ventajas muy importantes, sin embargo, no es menos cierto que éstas son contrarrestadas por algunas desventajas. Por tanto, **no siempre una solución distribuida es la más adecuada**, solamente cuando la naturaleza del

problema así lo determina. De tal forma, cada aplicación requiere de un análisis exhaustivo para concluir si su desarrollo, bajo el enfoque distribuido, resulta beneficioso.

2.6 Arquitectura general

Corresponde a la estructura lógica y física de los elementos que intervienen en un sistema distribuido así como la manera de interactuar entre ellos.

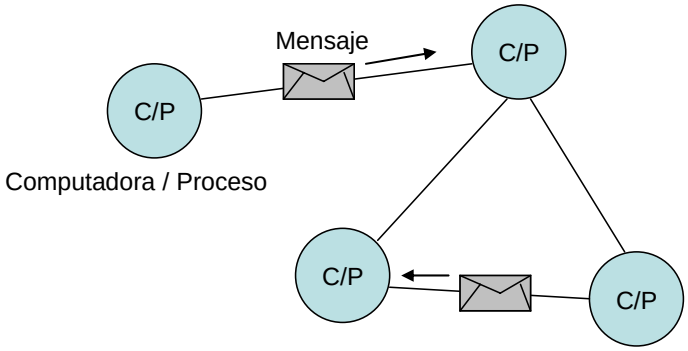


Figura 2.2: Estructura de un sistema distribuido

En la figura 2.2 se pueden establecer dos perspectivas [12]:

1. *Física*, relacionada con el hardware y en donde las computadoras constituyen los nodos del sistema distribuido conectadas por medio de una red.

2. *Lógica*, relacionada con el software y que puede ser interpretada como un conjunto de procesos[5] cooperantes.

En ambos casos el mecanismo de comunicación para lograr la interacción es a través del paso de mensajes. Es importante mencionar que la distribución lógica es independiente de la física ya que los procesos no necesariamente han de estar enlazados mediante una red sino que todos pueden encontrarse en una sola computadora.

Una vez identificados los elementos constitutivos de un sistema distribuido, es necesario organizarlos y establecer el rol que desempeñan. El modelo *cliente-servidor* ayuda a comprender y administrar la complejidad inherente de los sistemas distribuidos.

2.6.1 Modelo *cliente-servidor*

Compartir recursos es uno de los motivos principales para construir sistemas distribuidos. Por tanto, debemos distinguir entre quienes poseen tales recursos y quienes los requieren. El modelo

5 Un proceso es definido como un programa de ejecución

cliente-servidor define dos roles que pueden ser asumidos ya sea por los procesos o por las computadoras: el rol de usuario del servicio (*cliente*) y el rol de proveedor del servicio (*servidor*).

Por lo general, se comparten recursos como impresoras, discos duros, archivos, *etc*. Tales recursos deben ser administrados mediante servidores para que puedan ser accedidos por clientes, *e.g*. los servidores *Web* administran un conjunto de páginas *Web* que son accedidas por clientes mediante los denominados *navegadores* (*browsers*).

Sin embargo, en muchas ocasiones es más significativo que el servidor se encargue de realizar algún tipo de procesamiento más intenso como puede ser la realización de cálculos estadísticos o científicos, el tratamiento de imágenes, la ejecución de simulaciones, *etc*.

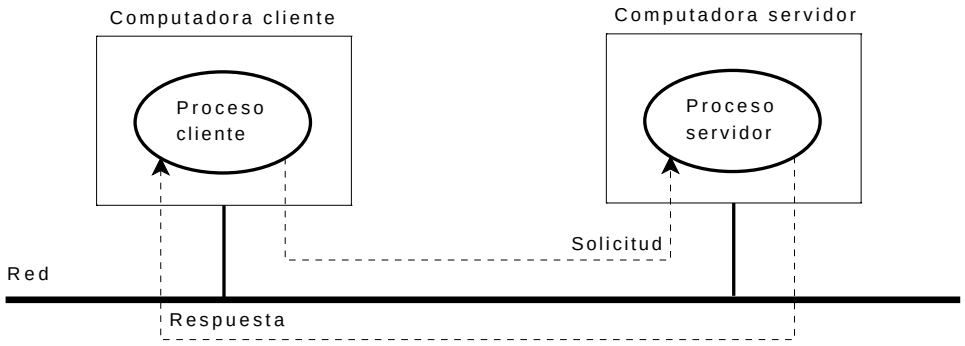


Figura 2.3: Modelo *cliente-servidor*

En la figura 2.3 se representa la interacción entre cliente y servidor. Cuando un cliente requiere de un servicio, envía un mensaje de solicitud al servidor, quien espera por solicitudes entrantes y una vez recibida, realiza el trabajo de procesamiento con el fin de devolver el resultado en un mensaje de respuesta para el cliente.

Una ventaja de este modelo es que ofrece la posibilidad de proporcionar servicios de manera simultánea a múltiples clientes (concurrencia). Así, un servidor no tendría que esperar a que termine de atender un cliente para atender otro; sin embargo, aunque existen dos partes (cliente y servidor), el servicio se encuentra centralizado en un único servidor, lo cual es un cuello de botella cuando el número de clientes crece o se produce alguna falla o desconexión.

Podemos mencionar el caso del servicio *DNS*, que traduce nombres de dominio de *Internet* en direcciones de red, ¿Cómo trabajaría este servicio si estuviese implementado con una sola tabla de direcciones almacenada en una computadora específica? En estas condiciones, el único servidor debería atender todas las solicitudes de resolución de direcciones, provocando incluso que nadie pueda usar el *Web* [11].

La figura 2.4 ilustra una de las técnicas para enfrentar esta limitación. Consiste en que los procesos o las computadoras asuman ambos roles, tanto cliente como servidor. En el ejemplo anterior, los servidores *DNS* pueden, a su vez, ser clientes de otros servidores *DNS* cada uno administrando una base de datos de direcciones. De manera semejante, los servidores *Web* y la mayoría del resto de los servicios de *Internet* son clientes del servicio *DNS* [2]. Así, un servicio

puede estar implementado por varios servidores interactuando unos con otros.

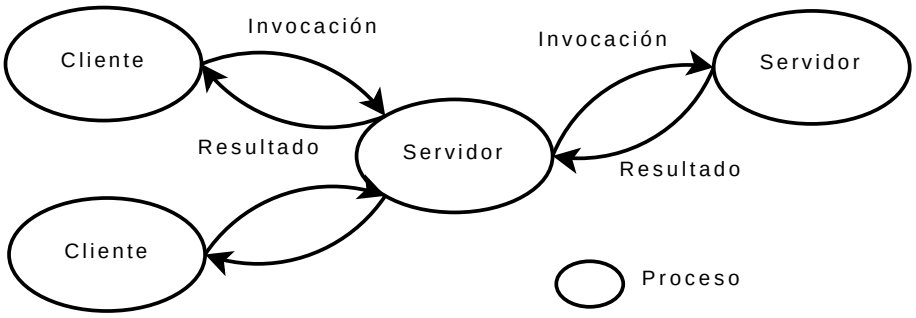


Figura 2.4: Modelo *cliente-servidor* con intercambio de roles

2.6.2 Modelo de objetos distribuidos

El interés por el paradigma de *orientación a objetos* (*OO*) ha crecido rápidamente en los últimos años [1]. Esta técnica ha probado ser conveniente para modelar la estructura y las interacciones en sistemas distribuidos cada vez más complejos. Esto es debido a las propiedades fundamentales de objetos tales como: abstracción, encapsulación, modularidad, herencia y polimorfismo [14].

En la tabla 2.2 (página siguiente) se muestra cómo determinados conceptos del paradigma *OO* pueden ser aprovechados en sistemas distribuidos (*SD*).

Concepto	*Paradigma de OO*	*Adaptabilidad a SD*
Objeto	Entidad física o conceptual del mundo real [16]. Define un estado y comportamiento.	Los *SD* pueden estar organizados en términos de objetos, unidades de distribución que interactúan. Los objetos se refieren a procesos o computadoras, indistintamente.
Abstracción	Permite identificar y separar los atributos (estado) y métodos (comportamiento) esenciales de un objeto.	Los objetos en un *SD* también consisten de un estado (datos administrados por el objeto) así como de un comportamiento, el cual corresponde al rol que desempeña, ***e.g.*** cliente, servidor, o ambos.
Encapsulamiento	Agrupación de los atributos y métodos en una entidad independiente, definiendo respectivamente, un estado interno y **una interfaz que provee algunos servicios al exterior**.	Los recursos en un *SD* son administrados por servidores y pueden ser encapsulados como objetos para ser accedidos por objetos clientes.
Ocultamiento	Es una consecuencia de la encapsulación, de tal forma que el estado del objeto puede ser modificado solamente por los métodos que conforman la interfaz, la cual permite interactuar con otros objetos.	La separación entre interfaz y su implementación es clave para comunicar objetos distribuidos. Es posible colocar una interfaz en una computadora mientras que el objeto en sí reside en otra. Además, la única forma de interactuar con un objeto es la interfaz, donde se exponen solamente los detalles necesarios para el resto del sistema.
Modularidad	Debido a que cada objeto es considerado como una entidad única y aislada, se convierten en módulos naturales.	La modularidad permite desarrollar componentes de aplicación (***e.g.*** clientes y servidores) que pueden ser distribuidos en varias computadoras.
Herencia	Es un mecanismo para compartir atributos y métodos con base en una relación jerárquica. Podemos ir de lo general a lo particular evitando redundancia.	La herencia en un entorno distribuido soporta la *re-utilización* de código [14]. Objetos de nuevas aplicaciones pueden aprovechar directamente las implementaciones de otros objetos en distintas ubicaciones.
Polimorfismo	Propiedad estrechamente ligada con la jerarquía de herencia. Significa que un mismo comportamiento puede tomar diferentes formas en función del objeto que se está tratando.	Ofrece capacidad de crecimiento, ***i.e.*** permite la *extensión* de un sistema ya que partes pueden ser añadidas para proporcionar nuevas y/o mejores funcionalidades. De manera similar, facilita la sustitución o intercambio de la implementación de un servicio (mantenimiento).
Comunicación	La comunicación (interacción) entre objetos se realiza por medio de invocaciones de métodos.	En un *SD* los procesos cooperan unos con otros a través de intercambio de mensajes. Así, las invocaciones de métodos pueden ser transformadas en mensajes.

Tabla 2.2: Adaptación del paradigma *OO* a sistemas distribuidos (*SD*)

La adaptabilidad del paradigma *OO* para entornos distribuidos ha determinado que la mayoría del software de sistemas distribuidos actual tienda a desarrollarse con lenguajes orientados a objetos [11], *e.g.* aplicaciones financieras, bursátiles, colaborativas, mensajería, bases de datos compartidas, reservación de boletos aéreos, *etc*.

2.7 Middleware

Los sistemas distribuidos toman en cuenta y tratan nuevos problemas que no existen en los sistemas centralizados, entre los cuales destaca la **heterogeneidad**, que se manifiesta en los siguientes aspectos [12]:

- *Aplicaciones:* los objetos que conforman una aplicación distribuida pueden estar implementados en **diferentes lenguajes de programación**.
- *Sistemas Operativos*: **distintos sistemas operativos** de las computadoras en un sistema distribuido.
- *Hardware*: **diferentes arquitecturas** de procesador y otros detalles técnicos que se presentan en cada computadora (*e.g.* representaciones de datos).
- *Red*: las diversas computadoras de un sistema distribuido se conectan por medio de **tecnologías de red diferentes**.

La heterogeneidad en cada uno de los niveles anteriores es resuelta por medio de componentes denominados *middlewares*. En la figura 2.5 se muestra la ubicación del *middleware* en un sistema distribuido:

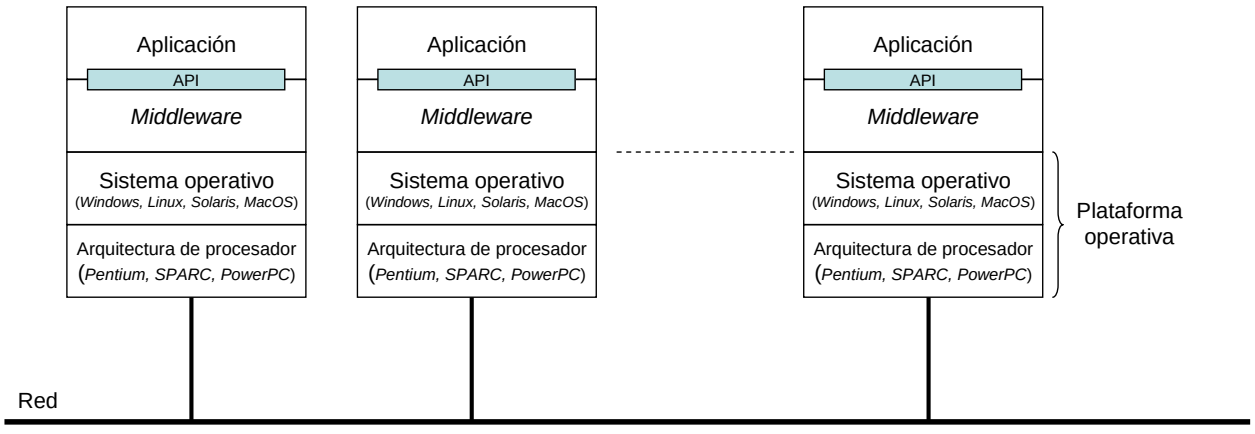


Figura 2.5: *Middleware* en un sistema distribuido

El *middleware* se ejecuta sobre diversas combinaciones de arquitectura de procesador y sistema operativo (*e.g. Pentium*/*Linux*, *PowerPC*/*MacOS*, *SPARC*/*Solaris*), y supera esta variedad

ofreciendo igual funcionalidad en todas las computadoras por medio de una *biblioteca* de programas o *API*, útil para el desarrollo de aplicaciones distribuidas.

La implementación de un *middleware* se adhiere a un estándar con el fin de lograr *compatibilidad*. El estándar es una descripción abstracta de un comportamiento deseado y que sirve como un modelo de acuerdo con el cual diferentes productos (implementaciones) pueden ser producidos [12]. Muchas tecnologías de *middlewares* están disponibles hoy y, en cuanto a aquellos que ofrecen soporte para el paradigma *OO*, tres de los estándares más populares para el desarrollo de sistemas distribuidos son:

- *CORBA, (Common Object Request Broker Architecture), de OMG (Object Management Group)* [17]
- ***Java RMI, (Java Remote Method Invocation)**, de Sun Microsystems* [18]
- *DCOM*, (*Distributed Component Object Model*), de *Microsoft Corporation* [19]

En general, estas tecnologías tienen una filosofía similar de funcionamiento, *i.e.* extienden el modelo de invocación de método en una sola computadora a invocaciones remotas entre objetos sobre distintas computadoras en un sistema distribuido. Desde el punto de vista del desarrollador, la interacción remota entre objetos es transparente, prácticamente idéntica a interacciones locales.

CORBA es resultado del esfuerzo de la *OMG* para definir un marco estándar para interoperabilidad de objetos distribuidos independientemente del lenguaje de programación. Tanto *CORBA* como *Java RMI* son sistemas abiertos, cuyas especificaciones y documentación se encuentran libremente publicadas. Los productos de ambas especificaciones pueden ser utilizados sobre diversas plataformas de sistemas operativos, desde *mainframes* a máquinas *UNIX*, *Linux*, o *Windows* a dispositivos de mano siempre que haya una implementación para la plataforma. La especificación de *DCOM* permite que los componentes de servidor sean escritos en diversos lenguajes de programación como *C++*, *Java*, *Object Pascal* (*Delphi*), *Visual Basic* e incluso *COBOL* [20]. *DCOM* es actualmente muy utilizado sobre la plataforma *Windows* y ya que es un software propietario, difícilmente puede ser considerado para un trabajo de investigación como el presente, donde pretendemos obtener una solución genérica y totalmente disponible.

CORBA soporta múltiples lenguajes de programación originando que su tecnología sea mucho más compleja. Además, la demanda de recursos para su instalación es una gran desventaja frente a la tecnología *Java*, la cual ofrece versiones de la plataforma, tanto para computadoras de alto rendimiento como para dispositivos con recursos limitados. Éstos últimos son requeridos cada vez más por las aplicaciones de tiempo real, así que hemos utilizado *Java* para el desarrollo de nuestra solución. Un *middleware* como *Java RMI* es la infraestructura que soporta el desarrollo y la ejecución de aplicaciones distribuidas de propósito general, *i.e.* aplicaciones convencionales como: bancarias, control de inventarios, comercio electrónico, contabilidad, nóminas de pago, agendas colaborativas, repositorios de información química, genética, atmosférica, *etc*. Nuestro trabajo está enfocado en aplicaciones donde no basta que los resultados sean correctos sino que se deben cumplir requisitos de planificación y de tiempo. Para este tipo de aplicaciones, ***Java RMI* no proporciona el soporte necesario**. A pesar de esta limitación, *Java RMI* provee un modelo útil para el desarrollo de nuestra solución, razón por la cual, en el siguiente capítulo se examina esta tecnología así como otras herramientas de *Java* convenientes para el presente trabajo.

Capítulo 3

Herramientas *Java*

En este capítulo exponemos de manera general las herramientas que ofrece *Java* y que servirán para el desarrollo de nuestra solución. El término *Java* típicamente es asociado únicamente con el lenguaje de programación; sin embargo, *Java* es mucho más que un lenguaje, es también la especificación de una *API*, una especificación de *máquina virtual* y otras utilidades, que conjuntamente definen un ambiente de programación y ejecución [21]. Primeramente, mencionamos el *API* de *sockets* como el mecanismo proporcionado por *Java* para la comunicación en red. Luego, tratamos la tecnología *Java RMI* cuyo modelo servirá como base para la arquitectura de nuestra solución. Enunciamos varias ventajas y limitaciones de este modelo, entre las cuales, hacemos énfasis en la falta de soporte para aplicaciones distribuidas con requisitos de tiempo real. Brevemente revisamos los conceptos fundamentales acerca de tiempo real, incluyendo características esenciales de este tipo de sistemas como concurrencia y planificación. Presentamos la *Especificación de Tiempo Real para Java* (*RTSJ*), sus principales clases e implementaciones.

3.1 *Sockets* en *Java*

Una red de computadoras es el medio de comunicación que utilizan los nodos de un sistema distribuido. Para establecer una comunicación, el emisor de un mensaje debe conocer la dirección del receptor. Típicamente deseamos comunicar procesos, cada uno ejecutándose en una computadora específica. En redes que utilizan el protocolo de *Internet* (*IP*), el direccionamiento está basado en la *dirección IP* de la computadora y un *número de puerto* asociado al proceso, identificado por un número, *e.g.* un servidor *Web* por lo general está asociado al puerto 80 [3].

Java incluye soporte directo para comunicaciones de red. Para tal efecto, el paquete `java.net` provee un *framework* para la creación y manejo de objetos que representan conexiones de red. Específicamente, la clase `Socket` define un canal de comunicación entre procesos sobre una red *IP*.

Una vez realizada la conexión entre dos procesos sobre la red, necesitamos una forma de enviar y recibir datos a través de la conexión establecida. El paquete `java.io` provee diversas clases para leer y escribir flujos de datos de diferentes formatos (enteros, flotantes, cadenas de

caracteres, *bytes*, *etc*.).

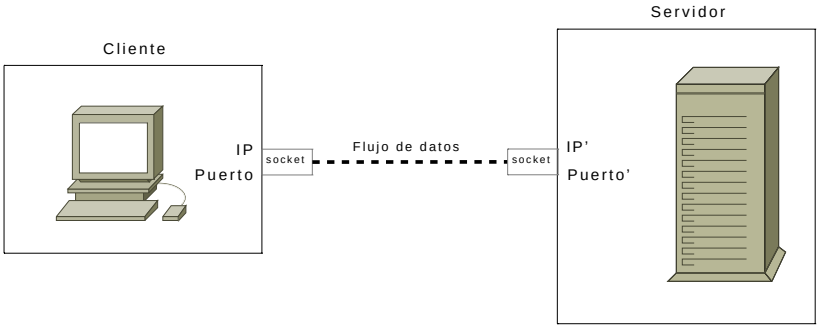


Figura 3.1: Conexión de red por medio de un *socket*

La figura 3.1 muestra la conexión de datos entre cliente y servidor. Nuevamente aquí, los términos cliente y servidor, se refieren a computadoras o procesos. El *socket* proporciona los puntos extremos de la comunicación asociando cada uno a un número de puerto y la dirección *IP* locales. El servidor "escucha" en el puerto conocido por el cliente. El cliente se conecta a este puerto y a la dirección *IP* del servidor por un *socket*. Una vez que la solicitud de conexión ha sido aceptada por el servidor, se produce el intercambio de información o flujo de datos a través de dos primitivas de comunicación: enviar y recibir. El cliente envía mensajes al servidor, quien realiza la recepción de los mismos con el fin de ejecutar algún tipo de operación. El resultado es devuelto al cliente utilizando el mismo *socket* de conexión.

3.2 *Java RMI* (*Java Remote Method Invocation*)

Cabe mencionar que el término *RMI* se refiere a la invocación de un método remoto de forma genérica [2]. *Java RMI* es una implementación de *Sun Microsystems* [18] y fue introducida en el *API* estándar de *Java* en el *JDK* (*Java Development Kit*) versión 1.1, como paquete o biblioteca de clases denominado `java.rmi`.

3.2.1 Propósito

Java RMI es un *middleware orientado a objetos*, y por tanto, proporciona una infraestructura para el desarrollo y la ejecución de aplicaciones distribuidas en lenguaje *Java*. Específicamente, **permite llevar a cabo la programación orientada a objetos para un ambiente distribuido**, considerando objetos distribuidos que residen en diferentes computadoras a lo largo de una red heterogénea, los cuales interaccionan a través de invocaciones de métodos.

Un objeto ejecutándose sobre una *máquina virtual de Java* (*JVM, Java Virtual Machine*) *en* una computadora puede invocar métodos en objetos que se ejecutan en otra instancia de la *JVM* (generalmente en otra computadora). De esta manera, se puede ejecutar código en una computadora remota desde una local y, si es el caso, se devuelve un resultado. Así se pensaría que los objetos se encuentran ejecutándose sobre la misma *JVM* [41].

La **invocación remota emplea la misma sintaxis que una invocación local**, puesto que los detalles de la comunicación de red entre objetos distribuidos remotos están ocultos y totalmente manejados por *Java RMI*.

3.2.2 Arquitectura

El diseño de *Java RMI* está basado en el siguiente principio: *"La especificación de comportamiento y la implementación de dicho comportamiento son conceptos separados"* [22]. Por tanto, la clave para comprender *RMI* es recordar que las ***interfaces*** **especifican comportamiento y las *clases* definen implementación**.

La especificación de un servicio remoto se encuentra en una *interfaz Java*. La implementación de un servicio remoto es definida en una *clase Java*. De esta forma, el código de ambas entidades permanece separado y puede ejecutarse en *JVMs* distintas. Como analizamos en el capítulo 2, las propiedades de encapsulación y ocultamiento de *OO* encajan perfectamente con las necesidades de un sistema distribuido, donde los clientes están interesados en un servicio publicado y los servidores están enfocados en proveer (implementar) el servicio.

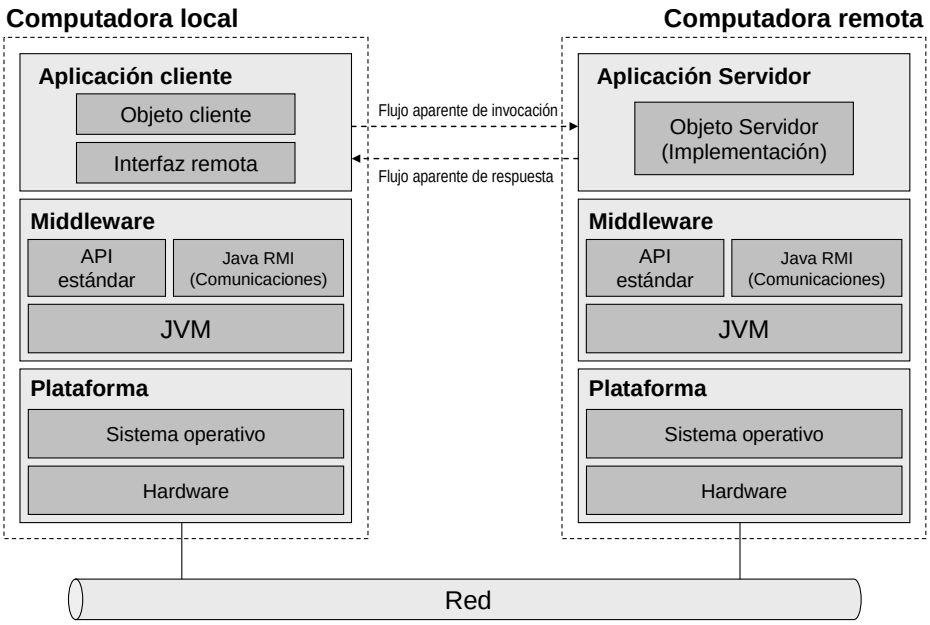


Figura 3.2: Modelo de capas para la comunicación con *Java RMI*

La figura 3.2 muestra los componentes del modelo de *Java RMI* que serán útiles para la arquitectura de nuestra solución. Podemos apreciar dos computadoras físicas separadas; generalmente, se tendrá esta situación y es la que vamos a considerar a lo largo del documento; sin embargo, puede darse el caso de que dos *JVMs* se ejecuten y comuniquen en la misma computadora física.

Una aplicación distribuida consiste de un conjunto de objetos que cooperan entre sí para lograr una determinada funcionalidad. Se puede considerar que una parte de la aplicación, denominada ***aplicación cliente***, se ejecuta en la computadora local mientras que otra parte de la misma aplicación, denominada ***aplicación servidor***, se ejecuta en la computadora remota. Este nivel corresponde al de *aplicación distribuida*. La capa de *middleware* corresponde a *Java RMI* en ambos casos. La *plataforma* puede ser muy diversa y en un sistema distribuido está constituida por distintas combinaciones de sistema operativo y arquitecturas de hardware.

Para la comunicación entre un objeto cliente y uno servidor, debe existir una ***interfaz remota*** que expone (sin implementarlos) los métodos que pueden ser invocados por el ***objeto cliente*** y que corresponden a los servicios ofrecidos por el ***objeto servidor***. La *interfaz remota* es utilizada dentro de la *JVM* local para acceder al *objeto servidor* ejecutándose sobre la *JVM* en la computadora remota. El *objeto cliente* realiza una invocación a un método en el *objeto remoto* casi exactamente como ellos invocan métodos locales. Por lo general, la implementación del método devuelve un valor que es enviado por la red al cliente. Gracias a *Java RMI*, los detalles de la comunicación (*e.g.* empaquetado de datos y paso de mensajes) pasan desapercibidos produciéndose los flujos aparentes de invocación y respuesta representados en la figura 3.2.

Aunque la tecnología *RMI* incorpora muchos elementos adicionales (*e.g.* servicio de registros y compiladores especiales), éstos han sido omitidos y nos hemos enfocado en los aspectos más representativos y que forman parte del desarrollo de nuestra solución.

3.2.3 Ventajas y limitaciones

Entre las ventajas que ofrece la tecnología *RMI* tenemos:

- *RMI* está diseñado para *Java* por lo que hereda todas las características de un lenguaje *OO* con el fin de aprovecharlas en un ambiente de computación distribuida.

- *RMI* se integra de manera natural dentro de la tecnología *Java* y la extiende añadiendo una biblioteca de clases (`java.rmi`) que permite comunicar múltiples *JVMs*.

- Utiliza un único lenguaje, por tanto, es una tecnología menos compleja si la comparamos con *CORBA*, la cual abarca extensos estándares y varios lenguajes de programación.

- La *portabilidad*, *i.e.* distribuir la aplicación o el código por toda la red (“*escribir una sola vez y ejecutar en cualquier lugar*”)[6].

Entre las limitaciones:

- Se puede decir que para tareas computacionales pesadas, es recomendable desarrollar aplicaciones con lenguajes de programación distintos a *Java*, mismos que puedan ser compilados obteniendo código nativo y así mejorar el rendimiento. La arquitectura de la

6 “*Write Once, Run Anywhere*”, es el eslogan de *Java* [22]

JVM es una desventaja en este caso ya que una interpretación adicional es añadida al procesamiento de las instrucciones.

- Un *middleware* como *Java RMI* permite el desarrollo y la ejecución de aplicaciones distribuidas de **propósito general**. Pero **¿Qué sucede con aplicaciones distribuidas especializadas?**, principalmente aquellas que tienen requisitos de tiempo real (muy utilizadas en los dominios de la automatización industrial, la aviónica, el militar, las telecomunicaciones, entre otros). Este tipo de aplicaciones no sólo deben obtener resultados lógicos correctos sino que además **deben cumplir las restricciones de planificación y de tiempo en los nodos que participan en un sistema distribuido**.

Precisamente esta última limitación es la que pretendemos enfrentar en el trabajo de tesis, aprovechando diversas herramientas de *Java*. Antes de revisar los esfuerzos realizados para dotar a *Java* de capacidades para tiempo real, a continuación presentamos una visión general acerca de dos temas muy relacionados entre sí y que son fundamentales para el desarrollo de nuestra solución. Por una parte, incluimos los conceptos necesarios sobre el dominio de tiempo real y, por otra parte, los aspectos teóricos y prácticos más relevantes sobre concurrencia.

3.3 Fundamentos de tiempo real

En computación es habitual considerar el término *tiempo real* como un sinónimo de respuesta instantánea o cómputo rápido; sin embargo, su connotación es mucho más amplia pues más que ser rápido, un **sistema de tiempo real debe ser predecible**, *i.e.* conocer con anterioridad los tiempos de respuesta que debe brindar a un evento determinado. De esta manera, es tan importante garantizar una respuesta en un milisegundo como en una hora o en un día determinado.

3.3.1 Definición de *sistema de tiempo real*

Los *sistemas de tiempo real* (*STR*) constituyen un tipo especial de sistemas de computación donde el tiempo es el parámetro más valioso a tratar y gestionar. Según [23]:

> *"Un sistema de computación de tiempo real es un sistema cuyo correcto funcionamiento no sólo depende de los resultados lógicos del procesamiento, sino también del instante físico en el cual estos resultados son producidos."*

Los *STR* se caracterizan por interactuar con su entorno, por lo que la respuesta ante estímulos externos (incluyendo el paso del tiempo) debe realizarse dentro de un intervalo de tiempo finito y especificado [24]. Un *STR* responde de manera predecible a estímulos externos muchas veces impredecibles.

3.3.2 Ejemplos

Los *STR* cada vez más se encuentran presentes en nuestro alrededor. Un *STR* se refiere a todo un sistema, *i.e.* incluye la aplicación de tiempo real, sistema operativo y controladores de propósito específico, dispositivos de E/S para interactuar con *sensores* y *actuadores* [16]. Usualmente se encuentran en el campo industrial, sin embargo, el auge de la electrónica de consumo y de los dispositivos móviles, ha causado la proliferación de los *sistemas embebidos o empotrados*.

A continuación mencionamos diversos ejemplos [25] [26]:

- Industria:
  - Sistemas de control y supervisión de procesos industriales
  - Control de plantas de manufactura
  - Control de robots industriales
  - Plantas nucleares
- Transporte:
  - Sistemas de control de tráfico aéreo
  - Sistemas de aviónica (electrónica en aeronaves para su control y comando)
  - Sistemas de control en automóviles
- Milicia:
  - Sistemas de guía y detección de misiles
  - Reconocimiento automático de blancos
  - Sistemas de radar y navegación
- Telecomunicaciones:
  - Sistemas de telefonía móvil y *Voz IP*
  - Sistemas multimedia y tele-conferencias
  - Tele-medicina (*e.g.* cirugía remota)
- Otras:
  - Reconocimiento de voz
  - Aplicaciones de marketing y bolsa de valores
  - Realidad virtual

Varios de los ejemplos citados son sistemas embebidos, en otras palabras, forman parte de un sistema de hardware y software más grande. La estrecha relación entre *STR* y sistemas embebidos hace que sean tratados conjuntamente. Muchos de estos sistemas son ahora, o serán en el futuro, interconectados por una red de comunicaciones [24].

3.3.3 Clasificación

Generalmente se distinguen dos tipos de *STR*, de acuerdo con el nivel de criticidad [16]:

- ***STR estrictos o críticos (hard)***: en los cuales un fallo puede producir graves daños, incluso consecuencias catastróficas, *e.g.* control de plantas nucleares, aviónica, control de tráfico, aplicaciones médicas, control de líneas de voltaje eléctrico.

- ***STR flexibles o acríticos (soft)***: la ocurrencia de fallos es tolerable y no es grave, *e.g.* sistema de procesamiento de transacciones en línea, sistemas de adquisición de datos, sistemas de obtención de cotizaciones de bolsa, sistemas multimedia.

Por lo general, los *STR* están compuestos por una mezcla de restricciones de tiempo críticas y acríticas.

3.3.4 Parámetros de tiempo y tareas

Una característica esencial de los *STR* es su interacción con el entorno. Ellos monitorean sensores y controlan actuadores de una amplia gama de dispositivos. Por tanto, son reactivos ante estímulos externos realizando una determinada acción o actividad compuesta de una o varias tareas. Una tarea es una entidad de ejecución independiente, que comienza con la lectura de los datos de entrada y termina con la producción de los resultados. Las tareas pueden ser [23][27]:

- *Periódicas*: son tareas que tienen un intervalo de tiempo constante entre activaciones sucesivas, *e.g.* procesamiento de señales de entrada de un sonar o radar.

- *Aperiódicas*: son tareas donde no se conocen los instantes de activación ni el intervalo de tiempo mínimo entre activaciones. Son ejecutadas en instantes no predecibles y en respuesta a eventos externos particulares, *e.g.* alarma debido a lecturas de alta temperatura y presión o la detección de un misil en vuelo.

- *Esporádicas*: los tiempos de activación no son conocidos pero se conoce que un intervalo de tiempo mínimo existe entre activaciones sucesivas, *e.g.* pulsaciones sobre un teclado o *mouse*, donde se establece un intervalo de tiempo mínimo relacionado con la capacidad de reacción del ser humano.

Como se ha mencionado, la ejecución de un sistema de tiempo real está conformada por un conjunto de tareas $\tau = \{T_1, T_2, T_3, \ldots, T_n\}$, donde n es la cardinalidad de τ. En la figura 3.3, se muestran los parámetros funcionales de una tarea $T_i = (C_i, D_i, P_i, S_i)$, con $i = 1,\ldots, n$:

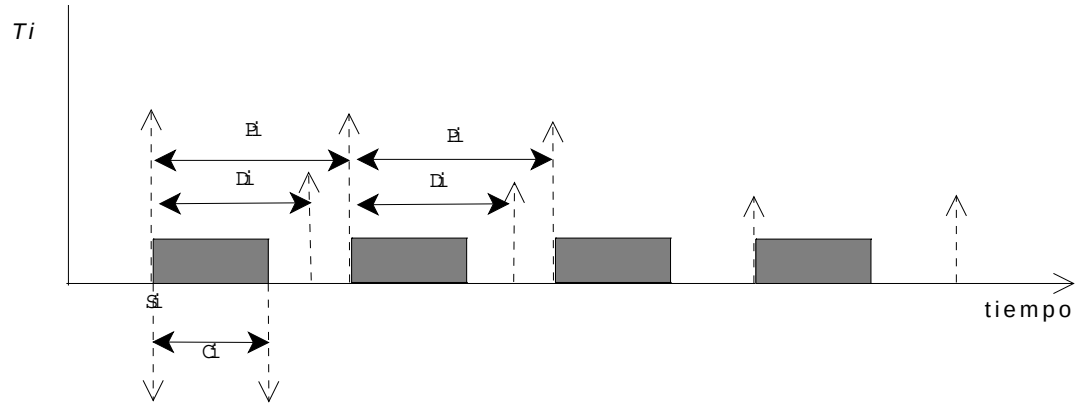


Figura 3.3: Parámetros de tiempo de una tarea

Donde [28]:

- *Tiempo de cómputo* (C_i): es el tiempo de procesador necesario para completar la ejecución de una tarea. Depende de la velocidad del procesador.

- *Plazo de respuesta o deadline* (D_i): es el intervalo de tiempo máximo que puede transcurrir entre el instante a partir del cual una tarea debe activarse y su finalización.

- *Periodo* (P_i): una tarea es ejecutada de forma regular cada intervalo de tiempo.

- *Tiempo de inicio* (S_i): corresponde al instante de tiempo de la primera activación.

Cabe señalar que, antes de implementar un *STR*, se deben realizar mediciones de tiempos para asegurar que $C_i \leq D_i$. Los tiempos pueden ser obtenidos mediante software (instrucción por instrucción) o instrumentos de medición electrónicos (***e.g.*** osciloscopios digitales).

3.3.5 Planificación[7]

Un *STR* responde de manera predecible a estímulos impredecibles y que, incluso, pueden producirse **al mismo tiempo**. Por tanto, el sistema debe resolver distintos problemas a la vez (concurrencia) tomando en cuenta que **en la computadora el tratamiento es secuencial** y un conjunto de tareas requieren el acceso a un recurso compartido (***e.g.*** el procesador).

La ejecución de varias tareas en un mismo procesador conlleva a la asignación del recurso procesador a cada una de las tareas. La *planificación* consiste en decidir el orden de ejecución de las tareas dando como resultado una secuencia de ejecución denominada *plan*. El componente encargado de establecer el plan de ejecución es el ***planificador***, quien utiliza un *algoritmo* o *política de planificación* para el ordenamiento de la ejecución de las tareas, ***i.e.***, es el criterio de acuerdo con el cual las tareas tienen acceso al procesador. Algunas de las políticas o algoritmos de planificación más utilizadas son [23][24]:

- *Fixed Priority Scheduling* (*FPS*): atención a las tareas basada en una simple prioridad numérica.

- *Rate Monotonic* (*RM*): asignación de prioridades crecientes según la frecuencia de las tareas.

- *Earliest Deadline First* (*EDF*): el criterio utilizado es el plazo de respuesta. Las tareas con plazos más cortos tienen mayor preferencia de ejecución.

- *Least Laxity* (*LL*): utiliza la laxitud (holgura) como criterio de decisión. La holgura es la diferencia entre el plazo de respuesta y el tiempo de ejecución de la tarea. Una tarea con la holgura más corta tiene mayor preferencia de ejecución.

7 Se denomina también: *ordenamiento*, *calendarización* o *scheduling*

En esta sección hemos revisado brevemente los elementos básicos acerca de los sistemas de tiempo real y que son los necesarios para el desarrollo de nuestra solución. El dominio de tiempo real es muy extenso y un estudio profundo del mismo no es el propósito del presente documento. Un análisis más detallado de este ámbito puede encontrarse en [23].

3.4 Concurrencia y planificación en *Java*

Hemos identificado la inherente concurrencia de los sistemas de tiempo real y embebidos, así como la necesidad de mecanismos de planificación. Ahora vamos a analizar cómo estas características pueden ser manejadas por *Java*.

3.4.1 Hilos

Java es un lenguaje de programación concurrente y no requiere de bibliotecas de programa auxiliares para implementar concurrencia, tal como sucede con *C* y *C++*. La programación concurrente permite que un único programa pueda realizar múltiples actividades o tareas. Las tareas separadas que se ejecutan simultáneamente, reciben el nombre de *hilos* (*threads*)[8]. Los hilos permiten al desarrollador de la aplicación manejar varias tareas a la vez. Esta técnica también se denomina ***programación multi-hilo***.

Un programa *Java* en ejecución es un proceso y, dentro de él, es posible definir varios hilos de ejecución. Cada hilo es una sección de código ejecutado independientemente de otros hilos que conforman el programa [21].

Los hilos de un programa Java se ejecutan todos dentro de la misma *JVM* mientras que una *JVM* será ejecutada como un único proceso para el sistema operativo.

El modelo de concurrencia de *Java* está basado en la clase `Thread` (paquete `java.lang`) que provee el mecanismo para la creación de hilos. La figura 3.4 [24] muestra la creación de un hilo de ejecución para la clase `MiHilo`. Esta clase debe extender[9] de `Thread` y hereda los métodos [21]:

- `void start()`: crea un nuevo hilo y ejecuta el método `run()`, definido en la clase que hereda de `Thread`.

- `void run()`: es el método que el hilo recientemente creado ejecutará. El desarrollador provee una implementación propia de este método con el código que desea que el nuevo hilo ejecute.

Una vez creado el objeto de la clase `MiHilo`, éste no comienza su ejecución hasta que el

8 Aunque el término *thread* es mejor conocido, utilizaremos *hilo* para ser congruentes con el idioma español

9 Extender una clase es la forma en que *Java* hereda métodos y atributos de una clase padre

método `start()` es invocado. El método `start()` después de tratar con algunos detalles de inicialización invoca el método `run()`, y el hilo es ahora ejecutable. Se debe notar que si el método `run()` es llamado explícitamente por la aplicación entonces el código es ejecutado de manera secuencial no concurrentemente [24].

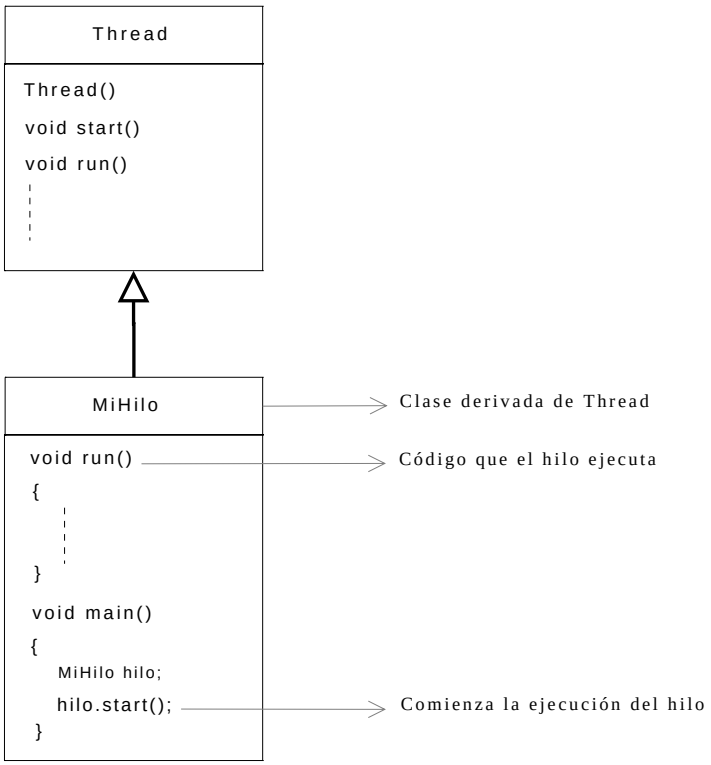


Figura 3.4: Creación de un hilo y sus métodos

3.4.2 Planificación en *Java*

La *JVM* permite a una aplicación tener múltiples hilos ejecutándose concurrentemente así que ellos compiten por el procesador. *Java* implementa un ***planificador basado en prioridad y con desalojo*** que monitorea todos los hilos en ejecución y decide cuál hilo debería estar ejecutándose en cualquier instante.

Se dice **basado en prioridad** puesto que cada hilo tiene asignado una prioridad numérica, un entero positivo en un rango que va de 1 a 10 y el valor por defecto es 5. La prioridad es dada a un hilo por medio de la clase `Thread` y puede ser cambiada únicamente por el desarrollador. Un hilo con más alta prioridad será seleccionado para su ejecución antes de uno con más baja prioridad.

Se dice **con desalojo** ya que cuando un hilo de más alta prioridad está listo, el planificador interrumpe (desaloja) el hilo de menor prioridad que está ejecutándose y el nuevo hilo de mayor

prioridad se convierte en el hilo en ejecución.

Aunque *Java* ofrece soporte directo para concurrencia y planificación, no fue diseñado para tiempo real; sin embargo, en los últimos años, los propulsores de *Java* han intentado extender su influencia hacia los dominios de los sistemas de tiempo real y embebidos.

3.5 *Java* para tiempo real

En nuestros días, los sistemas de tiempo real se han vuelto más comunes, *e.g.* sistemas de automatización industrial, transporte, telecomunicaciones. Los desarrolladores de aplicaciones para este tipo de sistemas generalmente utilizan lenguajes como: *ensamblador*, *C, C++*, *Ada*. Recientemente se ha producido un interés especial sobre *Java*, un lenguaje *OO* "puro" que proporciona ventajas propias como [27]:

- Simplicidad (aprendizaje rápido y mayor productividad, esencial para la industria del software)
- Portabilidad (independencia de la plataforma)
- Seguridad
- Popularidad
- Distribuido
- Multi-hilo

Estas características pueden ser muy útiles a la hora de diseñar y construir aplicaciones en el dominio de tiempo real. En el caso de portabilidad, las aplicaciones para sistemas embebidos y de tiempo real no son portables pues dependen de una plataforma operativa particular y de hardware específico.

A pesar de las enormes ventajas, *Java* estándar también posee serias limitaciones que lo hacen no apropiado para desarrollo de aplicaciones de tiempo real. Típicamente se han mencionado que los inconvenientes del lenguaje son:

- *Falta de rendimiento* (lentitud) debido a la interpretación de código (*bytecodes*), el cual es un código intermedio independiente de la máquina que es interpretado en tiempo de ejecución y, por tanto, disminuye el rendimiento.

- *Falta de predictibilidad* en tiempo de ejecución o comportamiento no determinista, debido a la gestión dinámica de memoria (*GC*, *Garbage Collector*) que podría interrumpir la ejecución por intervalos de tiempo impredecibles.

Cabe mencionar que el primer punto ha sido mejorado en pequeña proporción con el uso de la compilación *just-in-time*, incluida en las versiones más recientes de *Java*, donde se compilan segmentos de código en tiempo de ejecución en lugar de ser interpretados. Además, es posible la generación de programas *Java* a código nativo como se verá posteriormente con una de las herramientas analizadas en este trabajo.

Para el segundo punto no hay una solución definitiva y diferentes enfoques son explorados, *e.g.* en [27] y [29] se abordan trabajos relacionados con un *GC* de tiempo real. Otras limitaciones están relacionadas con la falta de habilidad del lenguaje para interrumpir segura y rápidamente una operación de entrada-salida o el manejo del tiempo con suficiente precisión.

Para hacer frente a tales desventajas, desde hace algunos años se realizan esfuerzos de investigación para lograr una extensión de tiempo real para *Java* estándar. El *Real-Time for Java Experts Group* (*RTJEG*), respaldado por *Sun*, comenzó a desarrollar la ***Especificación de Tiempo Real para Java*** *(RTSJ, Real-Time Specification for Java)* [4] en marzo de 1999 dentro del *Java Community Process* (*JCP*).

3.6 *RTSJ*

La versión 1.0.1 de la *RTSJ* fue emitida en junio de 2005 y se mantiene en desarrollo para lograr que *Java* sea efectivamente adecuado para tiempo real. Provee adiciones para que *Java* pueda ser utilizado para sistemas de tiempo real. En resumen, define [25]:

- Extensiones a la especificación de la *JVM*, para una *máquina virtual de Java de tiempo real* (plataforma de ejecución).

- Creación de una *API* de tiempo real (paquete `javax.realtime`) para desarrollar programas *Java* de tiempo real (plataforma de desarrollo).

Una de las ventajas principales y guías bajo las cuales se desarrolla la *RTSJ* es permitir que los programas puedan **combinar código *Java* de tiempo real y código estándar** (*Java* sin tiempo real), *i.e.* que puedan interaccionar en el mismo ambiente de ejecución.

3.6.1 *API* de la *RTSJ*

La *RTSJ* produce un paquete adicional (`javax.realtime`) que define varias clases e interfaces con el fin de permitir la programación para tiempo real. A continuación realizamos una visión general de las clases de la *RTSJ* que soportan las siguientes áreas: relojes y valores de tiempo, planificación e hilos de tiempo real [24].

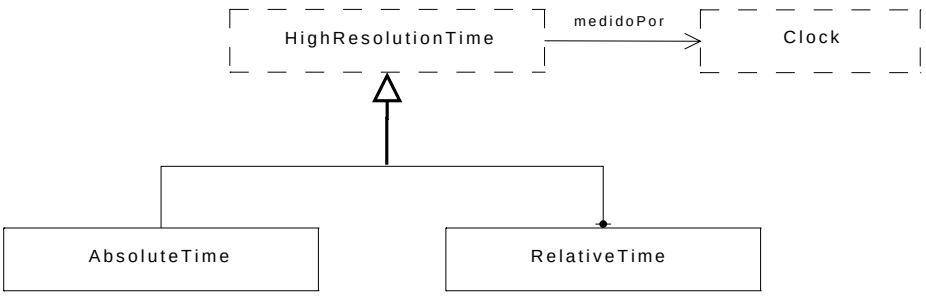


Figura 3.5: Clases *RTSJ* para manejo del tiempo

La figura 3.5 presenta las clases e interfaces de la *RTSJ* para manejo del tiempo. La clase abstracta `Clock` permite definir, por defecto, un reloj de tiempo real que avanza en sincronía con el mundo externo y que expresa el tiempo con una precisión de *nanosegundos*.

La clase `HighResolutionTime` permite representar el valor de tiempo dado en *mili* y *nanosegundos*. Ya que esta clase es abstracta, se utilizan sus clases derivadas. Por una parte, `AbsoluteTime` mide el lapso del tiempo transcurrido desde el 1 de enero de 1970 (00:00:00) hasta un punto específico, mientras que `RelativeTime` permite la medición de un intervalo de tiempo entre dos instantes determinados.

En secciones anteriores se mencionó la importancia de la planificación en sistemas de tiempo real. Aunque la *RTSJ* explícitamente soporta planificación basada en prioridad a través de la clase `PriorityScheduler` (figura 3.6), es posible que alguna implementación soporte otro tipo de planificador (*e.g. EDF*). Consecuentemente, `Scheduler` es una clase raíz con una clase derivada definida: `PriorityScheduler`.

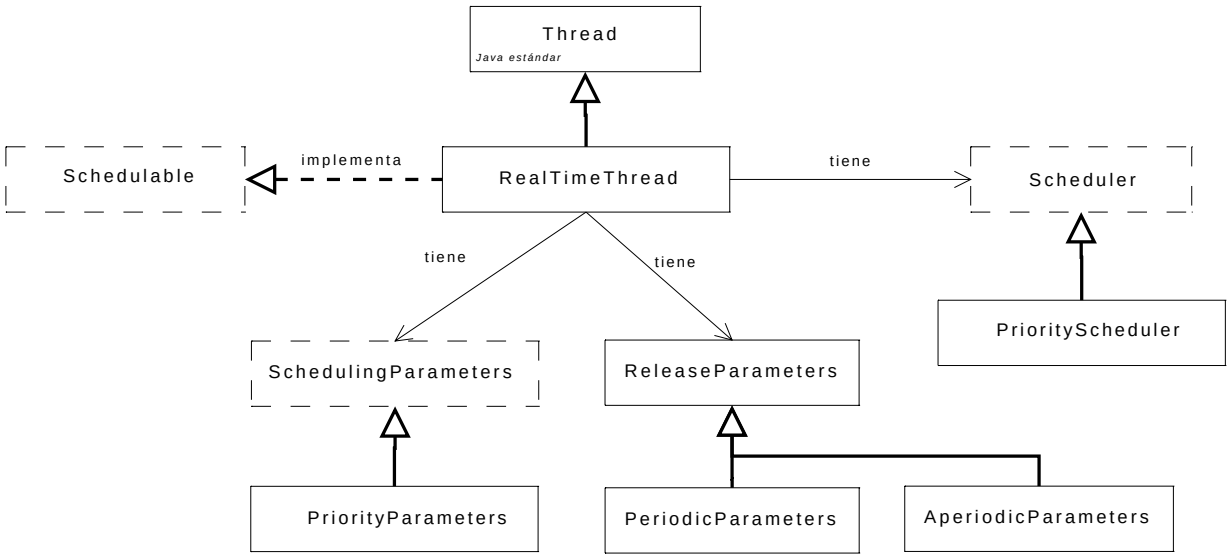


Figura 3.6: Clases y relaciones para hilo de tiempo real y planificación

En general, las aplicaciones están constituidas por múltiples hilos de ejecución (*threads*). Dentro de la *RTSJ* destaca la creación de un nuevo tipo de hilo, un **hilo de tiempo real** que es una extensión de la clase `Thread` de Java estándar, denominado `RealtimeThread`. Los hilos de tiempo real pueden tener asociados **parámetros de planificación** (`SchedulingParameters`) y **parámetros de activación** (`ReleaseParameters`).

La clase `SchedulingParameters` provee la clase raíz desde la cual un rango de posibles criterios de planificación puede ser expresado. La *RTSJ* define únicamente un criterio, basado en prioridad mediante la clase `PriorityParameters`. Este parámetro es utilizado por el planificador para determinar cuál hilo de tiempo real es elegible para ejecución.

La jerarquía de clases a partir de `ReleaseParameters` especifica los requerimientos de activación de un hilo de tiempo real, *i.e.* indican cuándo un hilo de tiempo real debe volverse ejecutable. Se identifican tipos de activación periódica (de manera regular) y aperiódica (al azar), mediante las clases `PeriodicParameters` y `AperiodicParameters`, respectivamente.

Todas las clases correspondientes a parámetros de activación tienen parámetros asociados. encapsulan un valor de `cost` y un `deadline`. El `cost` es la cantidad máxima de tiempo de procesador (tiempo de ejecución) necesitado para ejecutar el hilo de tiempo real cada vez que es activado. El `deadline` es el tiempo en el cual el hilo de tiempo real debe haber finalizado la ejecución actual. La clase `PeriodicParameters` incluye dos componentes: un valor de tiempo `start` para la primera activación y el intervalo de tiempo (`period`) entre activaciones. Estos datos **se corresponden con los parámetros de las tareas de tiempo real** (tiempo de cómputo, plazo de respuesta, periodo y tiempo de inicio), analizados en la sección 3.3.4.

3.6.2 Limitaciones de la *RTSJ*

La *RTSJ* es un gran paso para lograr que *Java* sea un ambiente de desarrollo y de ejecución apropiado para sistemas de tiempo real. Está diseñada para soportar aplicaciones de tiempo real tanto flexibles como estrictas [37], sin embargo, **su diseño considera únicamente** la ejecución de programas *Java* de tiempo real **sobre sistemas de un solo procesador**.

Por otra parte, la *RTSJ* asume para su funcionamiento un *sistema operativo de tiempo real*. Este tipo de sistemas operativos se caracteriza por la falta de disponibilidad, alto costo, dependencia de un hardware en concreto, escasa documentación y poco soporte para aplicaciones y para trabajo en red, entre otras desventajas.

A continuación presentamos el trabajo de evaluación de varias implementaciones de la RTSJ sobre un sistema operativo estándar como *Linux*, para determinar la mejor alternativa y su viabilidad para ofrecer un soporte conveniente aplicaciones de tiempo real. De esta manera, es posible generalizar el uso de dichas aplicaciones sobre una plataforma operativa muy popular. Aprovechamos que las versiones actuales del *kernel* de *Linux* proveen soporte para tareas que son consideradas de tiempo real. Para tal efecto, existen planificadores especiales incluidos en el *kernel* que manejan este tipo de tareas como se explicará en adelante.

3.7 Evaluación de implementaciones de la *RTSJ*

Existen algunas implementaciones bien conocidas de la *RTSJ*. Puesto que la especificación se mantiene en continuo desarrollo, las diferentes versiones pueden funcionar levemente diferente [30]. Nuestra evaluación cualitativa y cuantitativa determinará: a) la mejor alternativa para trabajar conjuntamente con *Linux*; b) identificar las causas y/o factores de impacto en el rendimiento; c) la viabilidad de utilizar *Linux* con *Java* de tiempo real para aplicaciones con restricciones de tiempo real flexible, estricto, o ambos, y así generalizar el uso de aplicaciones sobre un sistema operativo muy popular como *Linux*.

3.7.1 Selección de las herramientas

Hemos seleccionado:

- ***jRate*** de la Universidad de Washington en St. Louis, E.U.A. [31].
- ***TimeSys RI*** de la empresa estadounidense *TimeSys* [32].
- ***JamaicaVM*** de la empresa alemana *Aicas* [33].

Los factores que han determinado la selección de estas implementaciones son: **popularidad, disponibilidad, y compatibilidad** con el sistema operativo *Linux*. Esta última característica es fundamental para comparar el rendimiento sobre una misma plataforma operativa. A continuación mencionamos los aspectos más sobresalientes de las herramientas evaluadas:

jRate (*Java Real-Time Extension*) es una extensión de código abierto para el compilador *GNU* de *Java* denominado *GCJ*. Esta utilidad provee soporte para la mayoría de las características requeridas por la *RTSJ* [31]. Como producto de la compilación con *jRate* se genera un código ejecutable nativo del sistema y, por ende, no necesita una máquina virtual para su ejecución.

La empresa *TimeSys* es la encargada de desarrollar la implementación de referencia oficial de la *RTSJ* denominada ***TimeSys RI*** y satisface todas las características obligatorias de la especificación. Está basada en la plataforma *J2ME* (*Java 2 Micro Edition*) [34]. Al contrario de *jRate*, cuando se ejecutan las aplicaciones, el *bytecode Java* es interpretado, no existe compilación *ahead-of-time* o *just-in-time* [35].

JamaicaVM (*Jamaica Virtual Machine*) es una implementación de la *RTSJ* compatible con la plataforma estándar *J2SE* (*Java 2 Standard Edition*). Ofrece soporte completo para la especificación *RTSJ* y, al igual que la implementación anterior, ejecuta todo el código del lenguaje *Java* utilizando una máquina virtual.

Es importante revisar la arquitectura sobre la cual se soporta el desarrollo y la ejecución de una aplicación *Java*. Para una mejor comprensión de la estructura y el funcionamiento de las implementaciones seleccionadas, a continuación se presenta un diagrama comparativo de sus correspondientes modelos de capas con respecto al de la *JVM* clásica tomando en cuenta la plataforma *Linux x86* [34][36]:

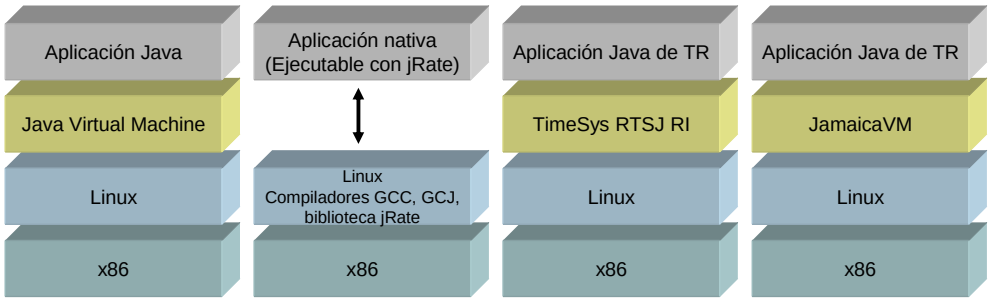


Figura 3.7: Arquitecturas de las implementaciones *RTSJ vs*. la arquitectura tradicional *JVM*

De acuerdo con la figura 3.7, la arquitectura de *jRate* determina que no hay interpretación de *bytecodes* sino que existe una ejecución directa de la aplicación por parte del sistema operativo. En los demás casos, los modelos de capas son similares a las de una máquina virtual de *Java* convencional. La existencia o falta de un nivel para interpretación es esencial para analizar los resultados obtenidos en las pruebas de rendimiento de las diferentes implementaciones de la *RTSJ*.

3.7.2 Infraestructura de pruebas

Con el fin de realizar un análisis confiable es necesario evaluar las herramientas sobre una misma infraestructura operativa. La comparación es posible pues las tres implementaciones son compatibles bajo *Linux* para plataformas *x86*. Sin embargo, no todas las distribuciones de *Linux* permiten instalar simultáneamente las implementaciones estudiadas.

Con respecto a lo anterior, realizamos pruebas con las distribuciones: *Fedora*, *SuSE*, *Ubuntu* y ***Red Hat,*** **resultando ésta última la única que permitió instalar a la vez las 3 implementaciones de la *RTSJ***. La tabla 3.1 describe las características técnicas de la computadora sobre la cual se llevaron a cabo las pruebas:

Característica	***Descripción***
Procesador:	Intel Core 2
Velocidad:	1.66 Ghz
Memoria RAM:	1 GB
Sistema Operativo:	Linux RedHat 9.0 kernel 2.4.20-8

Tabla 3.1: Plataforma de pruebas

3.7.3 Pruebas

Es nuestro objetivo conocer el comportamiento y rendimiento de las herramientas seleccionadas cuando ejecutan una aplicación de tiempo real. Para tal efecto, los parámetros que van a ser analizados son:

- Variación en la activación periódica de una tarea
- Tiempo de ejecución de una tarea
- Pérdida de plazos de ejecución (*deadlines*)
- Uso de recursos computacionales (memoria y procesador)

Para realizar las mediciones se han utilizado programas en código *Java* de tiempo real que servirán como programas de prueba y que se deben ejecutar con cada una de las implementaciones de la *RTSJ*. Enseguida se describen los programas realizados en cada caso así como los resultados obtenidos a partir de la ejecución.

3.7.3.1 Cálculo de la variación en la activación de una tarea

Las aplicaciones de tiempo real usualmente están conformadas por tareas periódicas, *i.e.* tareas que son ejecutadas de forma regular cada intervalo de tiempo. La precisión con la cual se realiza la ejecución periódica es importante y crítica para un sistema de tiempo real. La variación en la activación se ve reflejada directamente en la respuesta que se genera. Mientras mayor es la variación en la activación, mayor es la variación en el tiempo con la que se producirá la respuesta. Una variación significativa, en la activación y la respuesta, es uno de los principales problemas que se deben evitar en aplicaciones de tiempo real.

Una aplicación *Java* de tiempo real está conformada por uno o varios hilos de ejecución con parámetros de tiempo. Los hilos son implementados por medio de la clase `RealtimeThread`. Para asociar parámetros de tiempo al hilo, en primer lugar, deben ser convertidos a un tipo de dato de tiempo usando la clase `RelativeTime`. Enseguida son asociados al hilo mediante la clase `PeriodicParameters`. La prueba consiste en la ejecución de 5 hilos de tiempo real, cada uno con un valor de periodo y plazo, tal como se muestra en la tabla 3.2:

Hilo de tiempo real	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de activaciones**
1	100	100	500
2	300	300	500
3	500	500	500
4	700	700	500
5	900	900	500

Tabla 3.2: Parámetros de prueba

Se producen 500 activaciones y medimos el tiempo que transcurre entre cada par de activaciones sucesivas, el cual debe ser muy aproximado al periodo establecido para cada hilo. La figura 3.8 muestra una sección del programa de prueba, específicamente parte del método `run()` cuyo código se ejecutará en cada activación del hilo.

```
public void run() {
  reloj.getTime(tiempoInicial);
  for (int i = 0; i < limite; ++i) {
    hiloTR.waitForNextPeriod();
    reloj.getTime(tiempoFinal);
    periodo = tiempoFinal.subtract(tiempoInicial);
    reloj.getTime(tiempoInicial);
  }
}
```

Figura 3.8: Sección del código de prueba para la variación del periodo

El método `run()` consiste de un ciclo `for()` que realiza un determinado número de iteraciones fijado por la variable `limite`. Además, contiene instrucciones para el cálculo del tiempo entre cada activación (periodo). Para tal propósito, utilizamos una instancia de la clase `Clock`, denominada `reloj`. El tiempo entre cada activación se calcula como la diferencia entre `tiempoFinal` y `tiempoInicial`, ambas instancias de la clase `AbsoluteTime`. Los instantes de tiempo inicial y final se obtienen con el método `getTime()`. La diferencia entre los dos valores es calculada por el método `substract()` y es de tipo `RelativeTime`. Este procedimiento es ejecutado en 500 ocasiones para cada hilo. El método `waitForNextPeriod()`es utilizado por los hilos periódicos para bloquear su ejecución hasta el comienzo del próximo periodo.

Procedemos a ejecutar el programa con cada una de las herramientas y en las siguientes gráficas se muestran los valores obtenidos. Así podemos observar el comportamiento o variación que experimenta el tiempo de activación durante la ejecución del hilo de tiempo real:

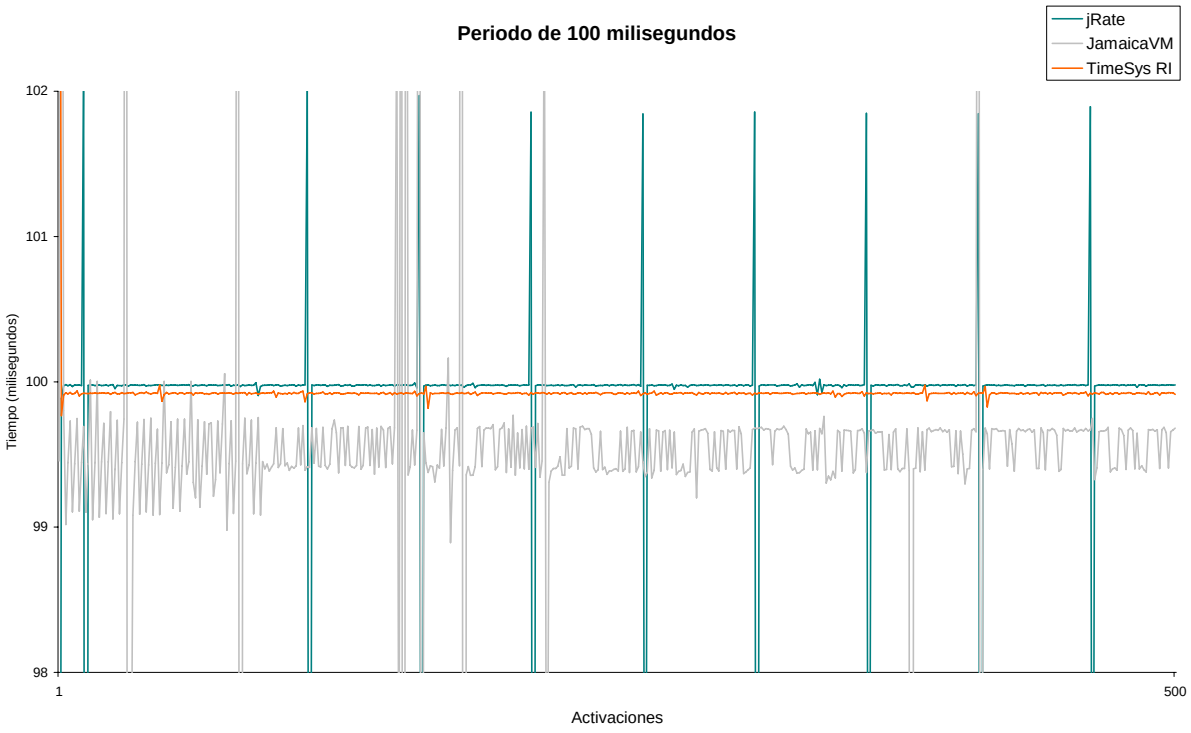


Figura 3.9: Gráfica de ejecución con periodo de 100 milisegundos

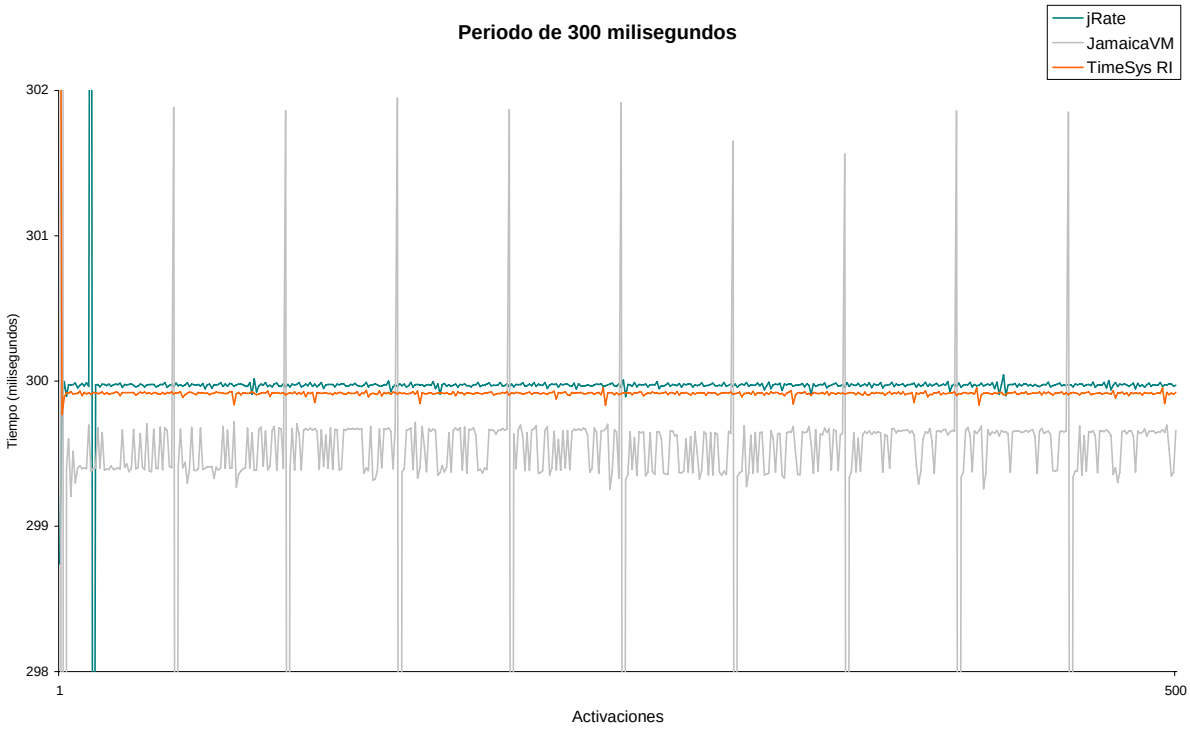


Figura 3.10: Gráfica de ejecución con periodo de 300 milisegundos

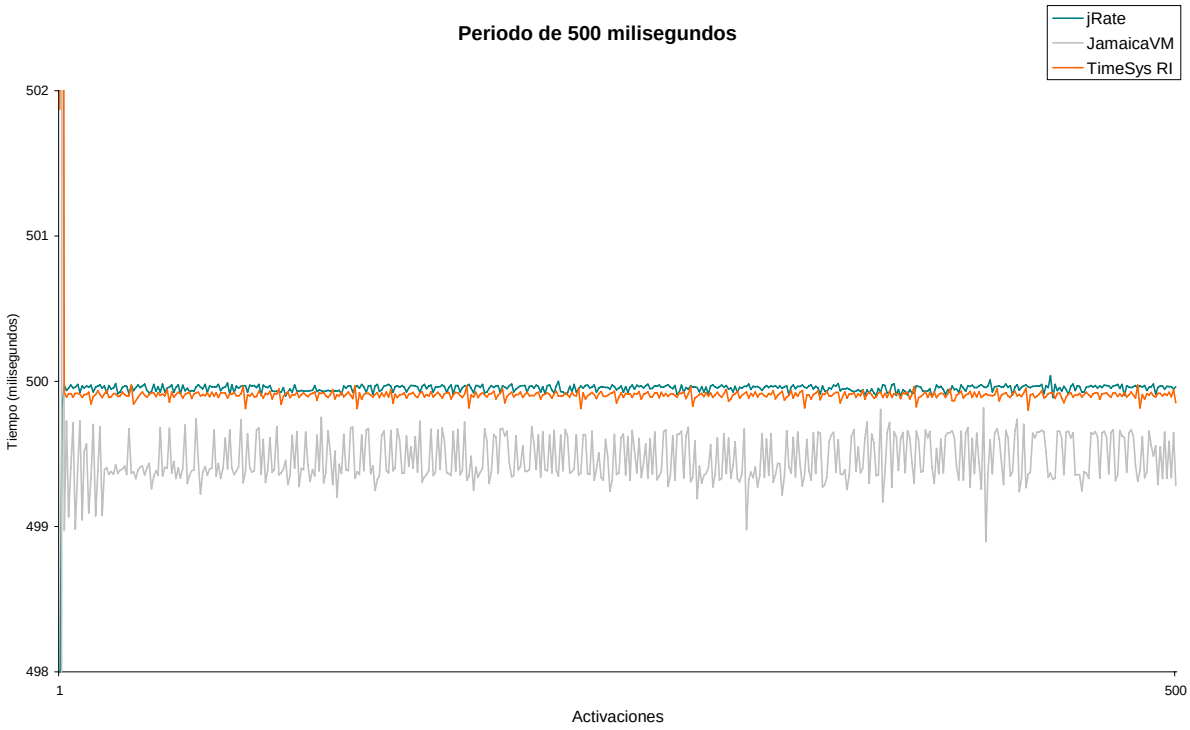


Figura 3.11: Gráfica de ejecución con periodo de 500 milisegundos

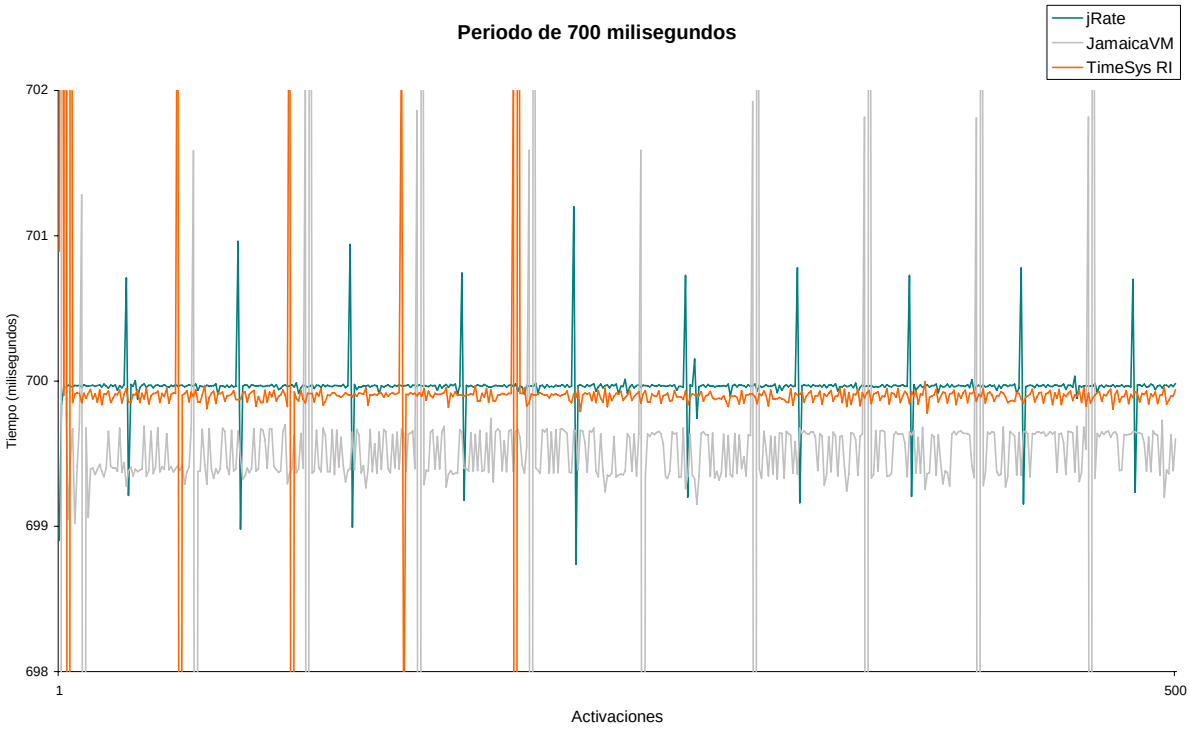


Figura 3.12: Gráfica de ejecución con periodo de 700 milisegundos

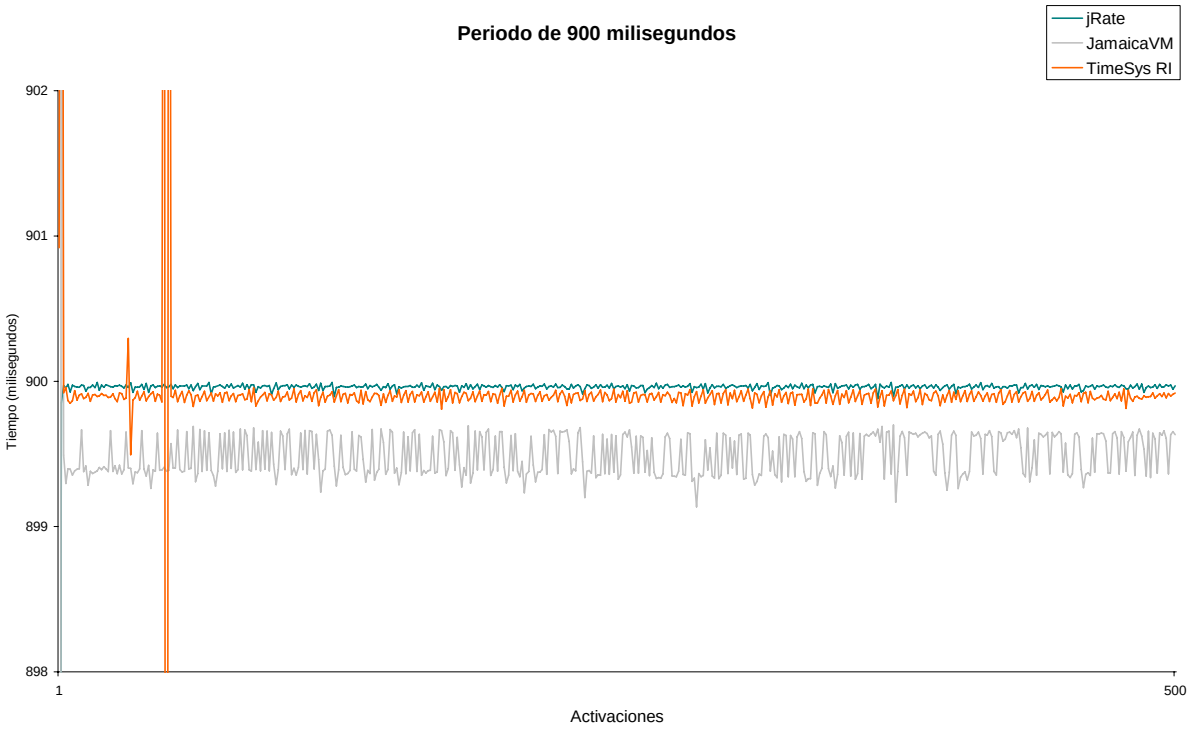


Figura 3.13: Gráfica de ejecución con periodo de 900 milisegundos

Para calcular la variación en cada activación, obtenemos la diferencia entre el tiempo calculado y el valor del periodo especificado. Así podemos conocer la desviación estándar (*S*) en milisegundos (*ms*) con respecto al periodo (*T*) que experimenta cada hilo, tal como se muestra en la tabla 3.3. Además presentamos el valor medio del periodo (*Tp*).

	T = 100 *ms*		*T* = 300 *ms*		*T* = 500 *ms*		*T* = 700 *ms*		*T* = 900 *ms*	
	Tp	*S*	*Tp*	*S*	*Tp*	*S*	*Tp*	*S*	*Tp*	*S*
jRate	99.83	1.38	299.97	0.51	499.95	0.11	699.96	0.18	899.94	0.37
Jamaica VM	99.22	5.41	298.84	13.68	498.48	22.27	697.99	31.33	897.71	40.16
Time SysRI	99.94	0.36	299.92	0.22	499.93	0.45	699.85	2.79	899.92	1.08

Tabla 3.3: Valor promedio de periodo y desviación estándar

Las siguientes figuras representan gráficamente los datos anteriores:

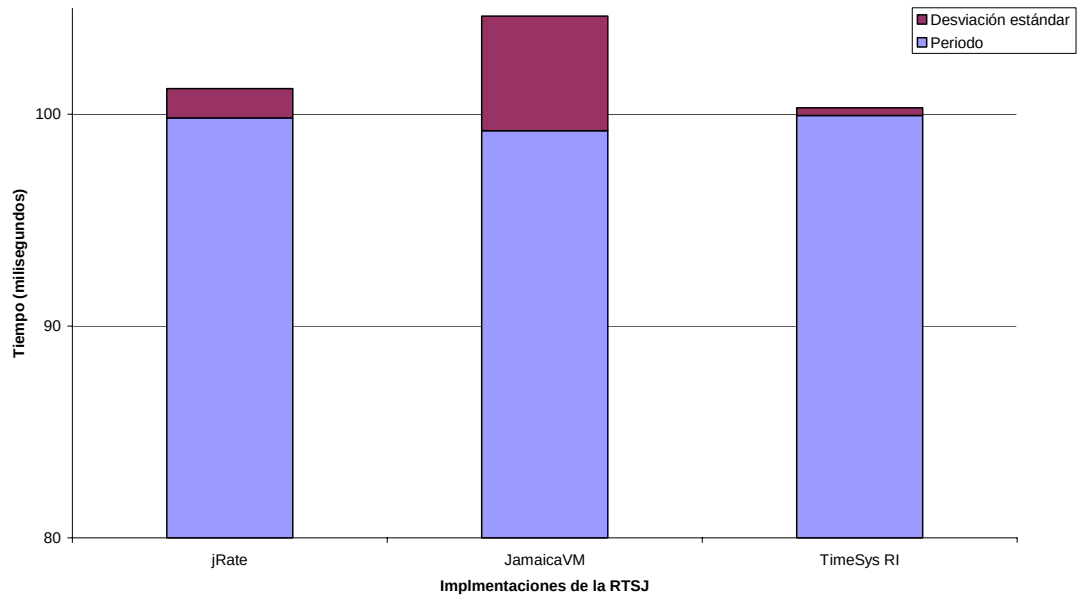


Figura 3.14: Periodo promedio y desviación estándar para 100 milisegundos

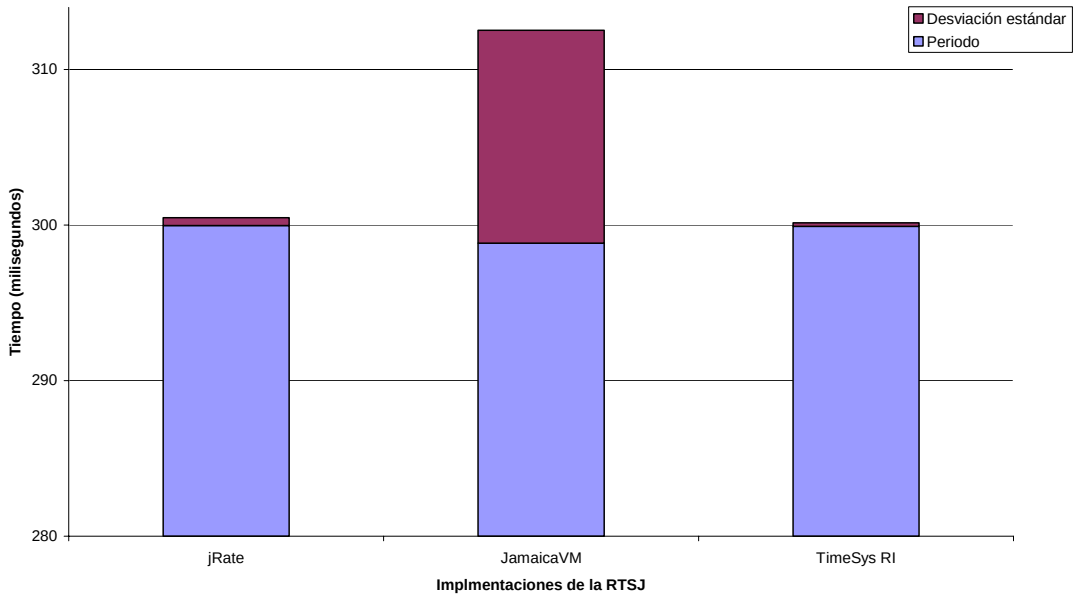


Figura 3.15: Periodo promedio y desviación estándar para 300 milisegundos

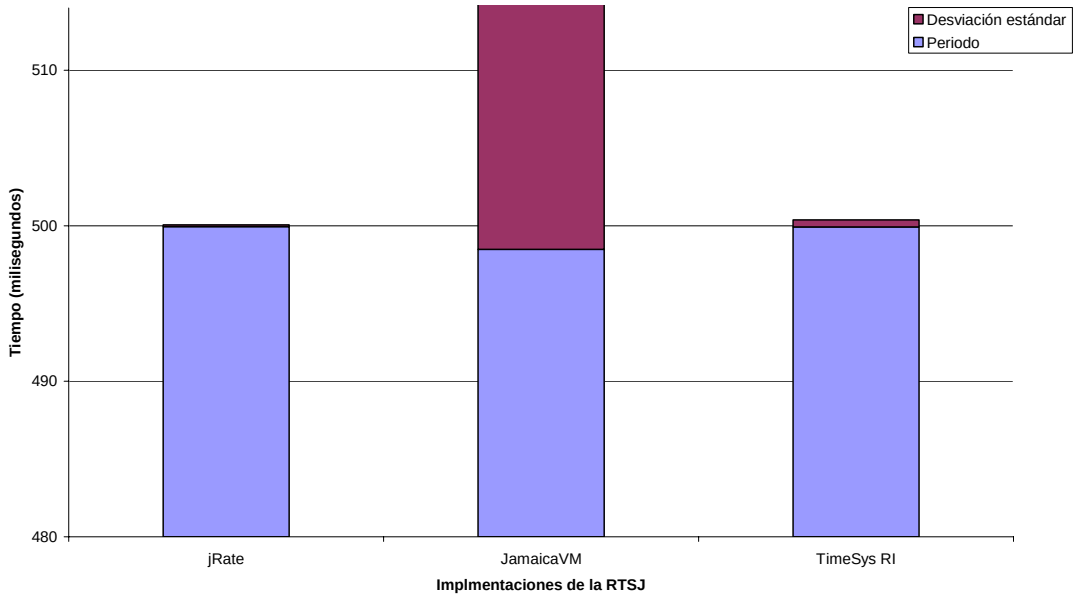


Figura 3.16: Periodo promedio y desviación estándar para 500 milisegundos

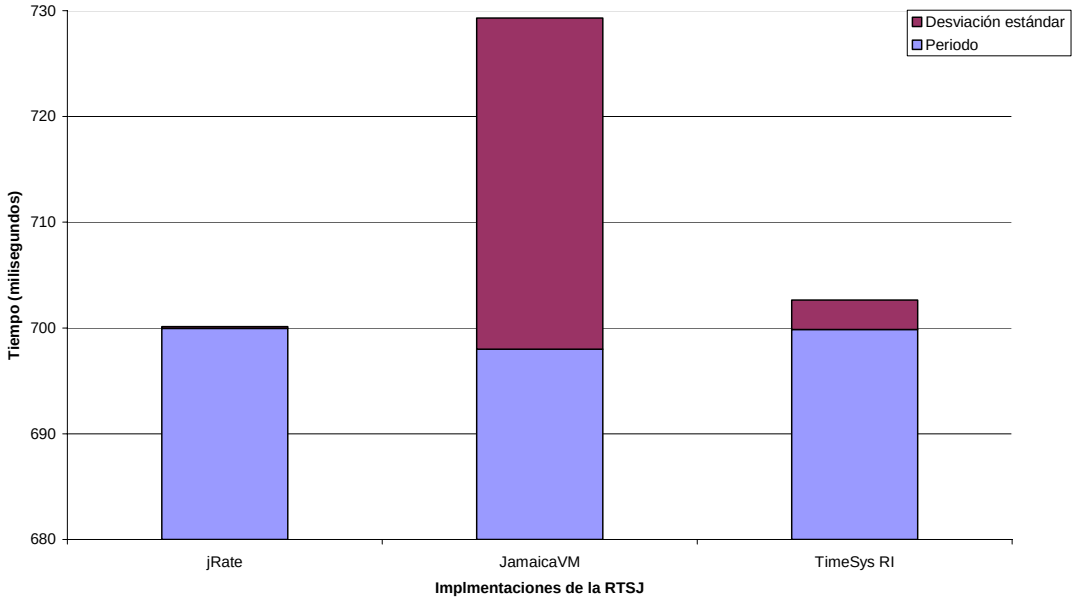


Figura 3.17: Periodo promedio y desviación estándar para 700 milisegundos

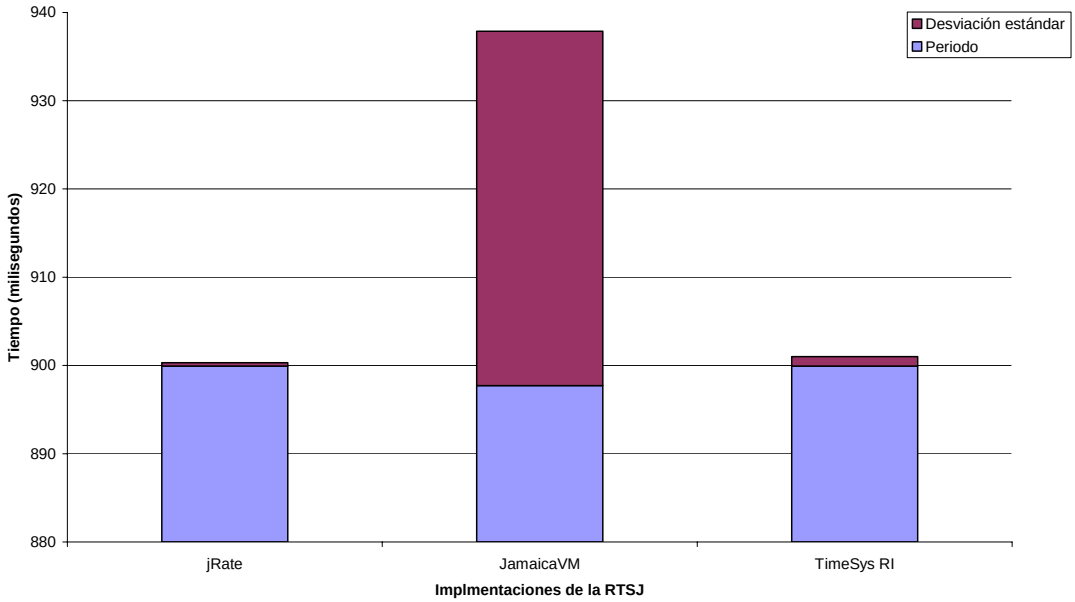


Figura 3.18: Periodo promedio y desviación estándar para 900 milisegundos

3.7.3.2 Cálculo del tiempo de ejecución de una tarea

Es el tiempo utilizado por el procesador para la ejecución de una tarea (hilo) y depende de la velocidad del procesador.

```
public void run() {
  while (waitForNextPeriod() && (n < 100)) {
    tiempoInicio = reloj.getTime();
      for (int i = 0; i < iteraciones; ++i) {
        for (int j = 0; j < 1000000; ++j) { }
      }
    tiempoFin = reloj.getTime();
    tiempoEjec = tiempoFin.subtract(tiempoInicio);
    n++;
  }
}
```

Figura 3.19: Sección del código de prueba para el tiempo de ejecución

La figura 3.19 muestra el código que ejecuta el hilo en cada activación, definido en el método `run()`, y que servirá para el cálculo del tiempo de ejecución. El experimento consiste en 10 pruebas (rondas), y en cada una se solicita la ejecución de un hilo de tiempo real periódico con un valor de carga especificado desde la línea de comando, el cual será asignado a la variable `iteraciones`. Este valor comienza en 10 y en cada ronda se incrementa en 10, llegando a un valor final de 100. La tabla 3.4 representa las 10 rondas (R), donde se obtienen 100 valores de tiempo de ejecución (t), cada uno calculado en cada activación del hilo (A).

	R_1		R_2	…….		R_{10}
$A_{1\,(10)}$	$: t_{1\,(10)}$	$A_{1\,(20)}$	$: t_{1\,(20)}$	…….	$A_{1\,(100)}$	$: t_{1\,(100)}$
$A_{2\,(10)}$	$: t_{2\,(10)}$	$A_{2\,(20)}$	$: t_{2\,(20)}$	…….	$A_{2\,(100)}$	$: t_{2\,(100)}$
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
$A_{100\,(10)}$	$: t_{100\,(10)}$	$A_{100\,(20)}$	$: t_{100\,(20)}$	…….	$A_{100\,(100)}$	$: t_{100\,(100)}$
	$t_{p\,(10)}$		$t_{p\,(20)}$			$t_{p\,(100)}$

R_i : ronda de ejecución i
$A_{i(j)}$: número de activación i con valor de carga j
$t_{i(j)}$: tiempo de ejecución calculado en la activación i con el valor de carga j
$t_{p(j)}$: tiempo promedio de ejecución con el valor de carga j

Tabla 3.4: Esquema de pruebas

Dentro del método `run()` existen 3 ciclos. El primero es un ciclo `while()` que permite realizar las 100 activaciones del hilo por medio del contador `n`, que empieza con un valor de cero y se va incrementando en 1. El método `waitForNextPeriod()` sirve para efectuar las activaciones periódicas del hilo, ya que cada activación espera hasta el próximo periodo. El siguiente ciclo es una instrucción `for()` cuyo límite denominado `iteraciones` controla la carga de procesamiento en cada prueba. El ciclo `for()` más interno sirve para simular la carga de procesamiento.

Para la medición del tiempo de ejecución, calculamos la diferencia entre las instancias de tiempo real absoluto (clase `AbsoluteTime`) denominadas `tiempoInicio` y `tiempoFin`, correspondientes al tiempo de inicio del método `run()` y al tiempo de finalización, respectivamente. Así obtenemos un dato de tiempo real relativo (clase `RelativeTime`) que se almacena en `tiempoEjec` y representa al tiempo de ejecución ($t_{i(j)}$) ocupado en una activación $A_{i(j)}$. Puesto que se realizan 100 activaciones, procedemos a calcular un valor promedio (t_p) para cada valor de carga. La figura 3.20 muestra los valores de t_p para cada una de las herramientas:

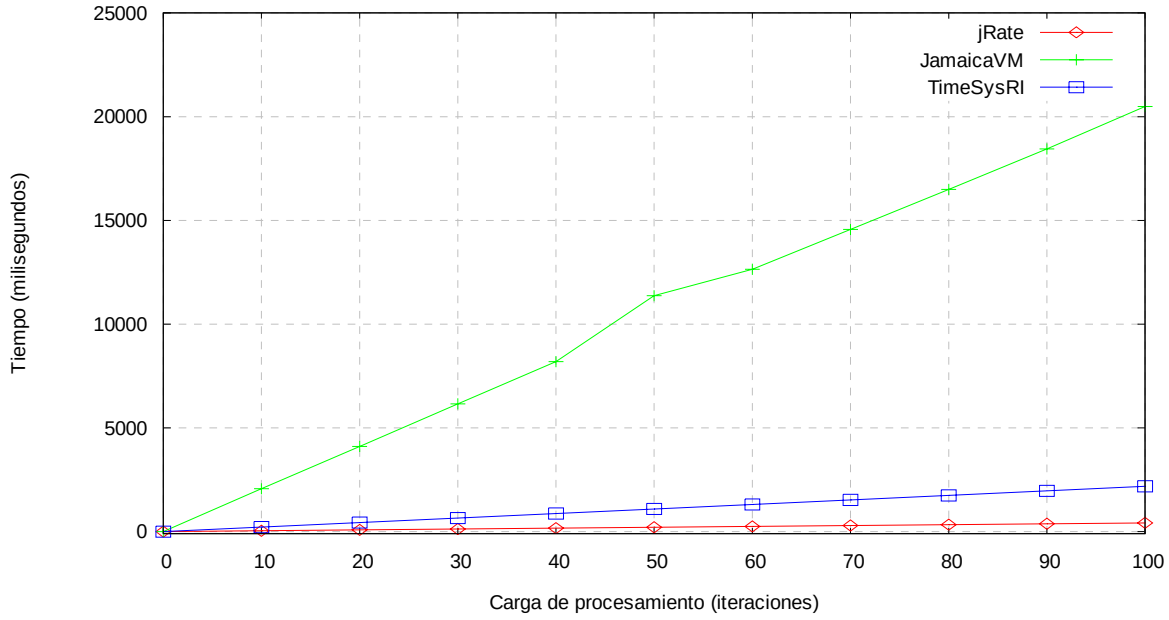


Figura 3.20: Tiempo de ejecución de una tarea

3.7.3.3 Número de plazos de ejecución perdidos

Un plazo de ejecución (*deadline*) es el tiempo en el cual debe completarse la ejecución de una determinada tarea. Para que la ejecución del hilo no se detenga, cuando sobrepasa un plazo, utilizamos un *manejador de plazos perdidos*, que se encarga de re-planificar la ejecución del hilo. Este componente puede ser implementado mediante una clase que hereda de la clase

`AsyncEventHandler`, perteneciente a la *RTSJ*. Tomamos como base el manejador implementado en [24].

En la figura 3.21 se muestra el código relevante, tanto de la clase `MissHdlr` que implementa el manejador de plazos perdidos así como el código del método `run()`, que ejecutará el hilo de tiempo real durante la prueba.

```
class MissHdlr extends AsyncEventHandler {
  static int nuevoLimite = 1000000;
  public void asociarHilo(RealtimeThread hiloTR) {
    this.hiloTR = hiloTR;
  }
  public void handleAsyncEvent() {
    ++plazosPerdidos;
    establecerLimite();
    hiloTR.schedulePeriodic();
  }
  public void establecerLimite() {nuevolimite -= 10000;}
  public static int obtenerLimite() {return nuevoLimite;}
  public int obtenerPlazosPerdidos() {return plazosPerdidos;}
}
...
public void run() {
  ...
  limite = MissHdlr.obtenerLimite();
  for (int i = 0; i < iteraciones; ++i) {
    for (int j = 0; j < limite; ++j) { }
  }
  ...
}
```

Figura 3.21: Sección del código del manejador de plazos perdidos

Realizamos la ejecución de 3 hilos de tiempo real, cada uno con los parámetros indicados en la tabla 3.5.

Hilo de tiempo real	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de activaciones**
1	200	100	100
2	300	150	100
3	400	200	100

Tabla 3.5: Datos de prueba

Los valores de plazo han sido establecidos como la mitad del periodo con el fin de producir plazos perdidos de manera intencional. La carga de procesamiento es manejada de modo similar que en el experimento anterior, *i.e*. desde 10 hasta 100; sin embargo, en esta ocasión se producen pérdidas de plazo. El hilo de tiempo real es asociado a una instancia del manejador de plazos por medio del método `asociarHilo()`, y cuando ocurre una pérdida de plazo, el método `handleAsyncEvent()` es activado, haciendo que el contador `plazosPerdidos` sea incrementado.

El método `schedulePeriodic()` permite que el hilo en ejecución sea re-planificado y se mantenga en ejecución a pesar de la ocurrencia de un plazo perdido. Además, el valor de la carga es disminuido con el método `establecerLimite()`, el cual reduce el límite del ciclo `for()` más interno, dentro del método `run()`. Previamente el valor del límite se obtiene con el método `obtenerLimite()`. Este procedimiento se realiza con el fin de evitar más plazos perdidos. Una vez terminada la ejecución del hilo periódico, el método `obtenerPlazosPerdidos()` da como resultado el total de plazos perdidos. La cantidad de plazos perdidos por cada herramienta es representada en las figuras 3.22, 3.23, y 3.24, correspondientes a cada una de las 3 pruebas.

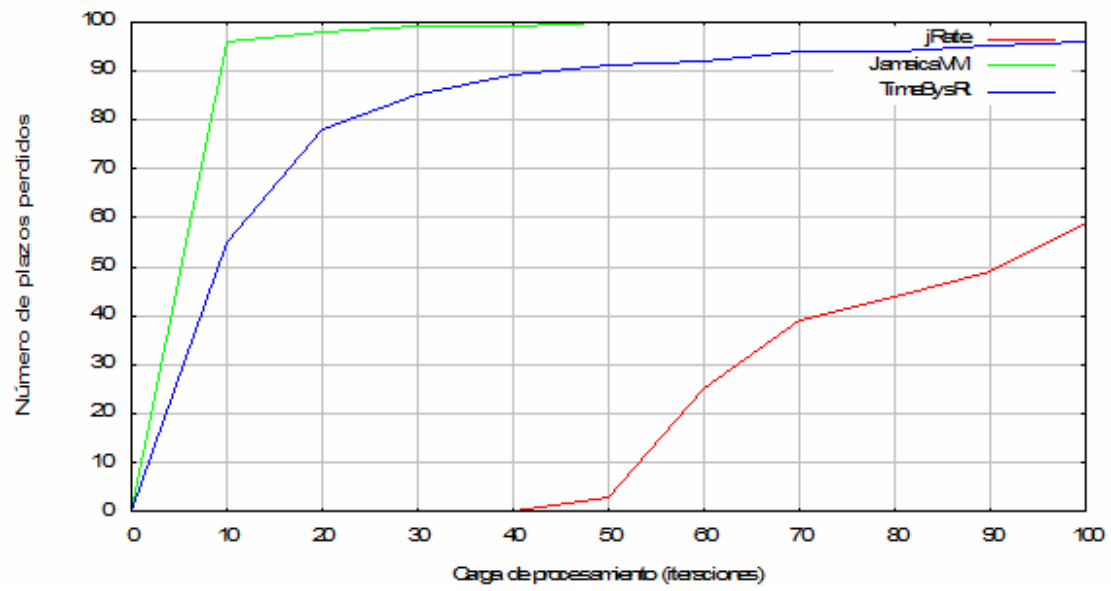


Figura 3.22: Plazos perdidos para el periodo de 200 milisegundos

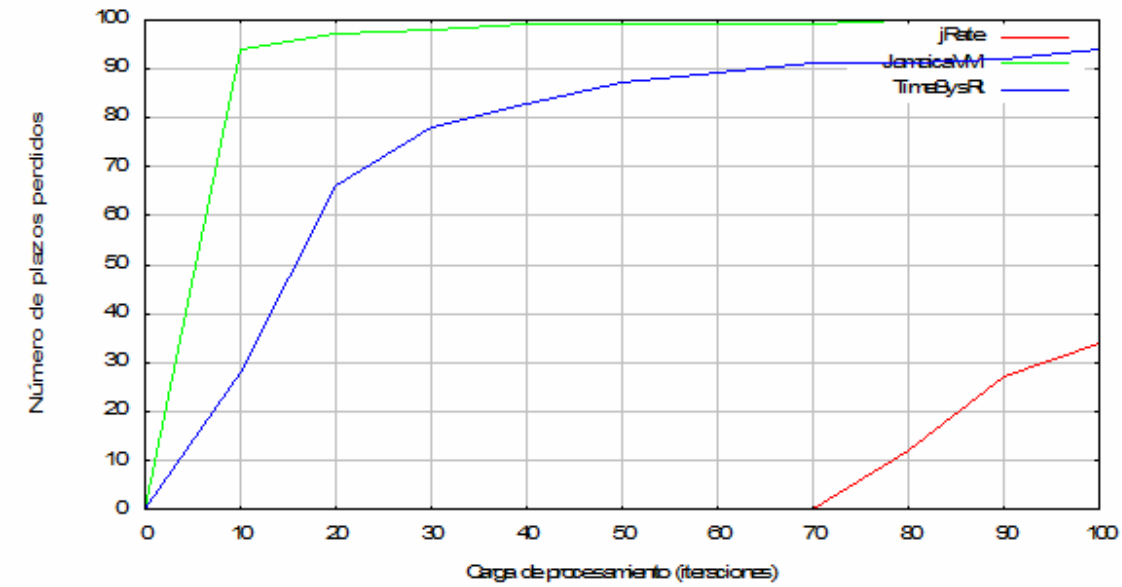


Figura 3.23: Plazos perdidos para el periodo de 300 milisegundos

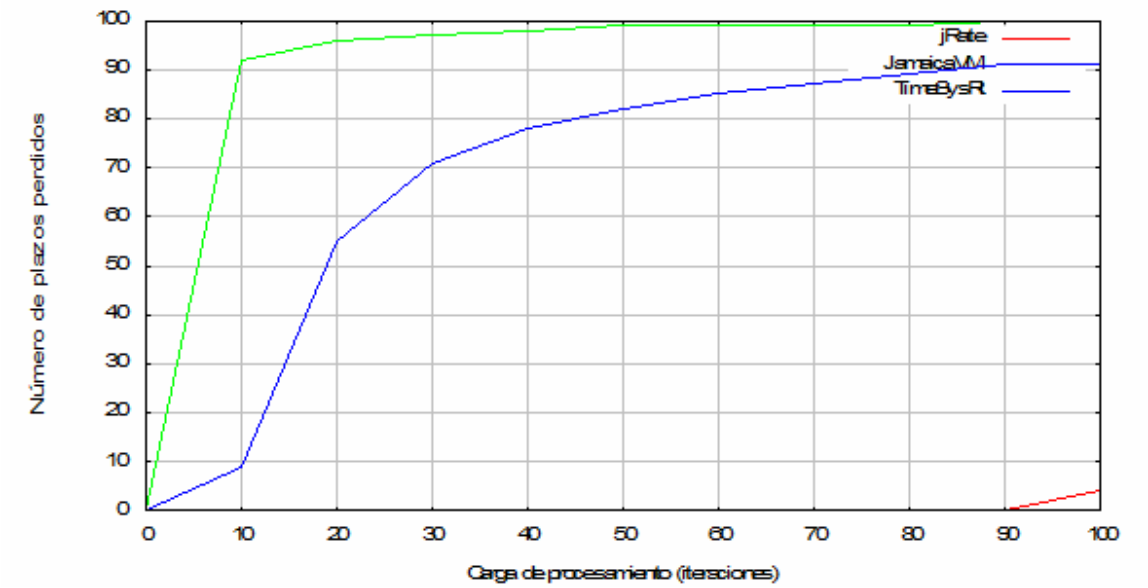


Figura 3.24: Plazos perdidos para el periodo de 400 milisegundos

3.7.3.4 Uso de memoria y procesador

La medición de recursos permitirá establecer el grado de eficiencia de las herramientas y la carga que producen al sistema. El comando `top` de *Linux* despliega información sobre la actividad del sistema en tiempo real y permitirá determinar el porcentaje del tiempo de procesador total y la porción de la memoria física ocupada por la tarea en tiempo de ejecución.

Para obtener información del programa en ejecución utilizamos el siguiente comando:

```
$ top -p <pid>
```

Donde `pid` es el identificador del proceso (valor asignado por *Linux*).

En la figura 3.25 se muestran los porcentajes de uso del procesador y de la memoria física por parte de cada una de las herramientas. Estos resultados corresponden a la prueba descrita en la sección 3.7.3.1, donde la carga de ejecución no varía.

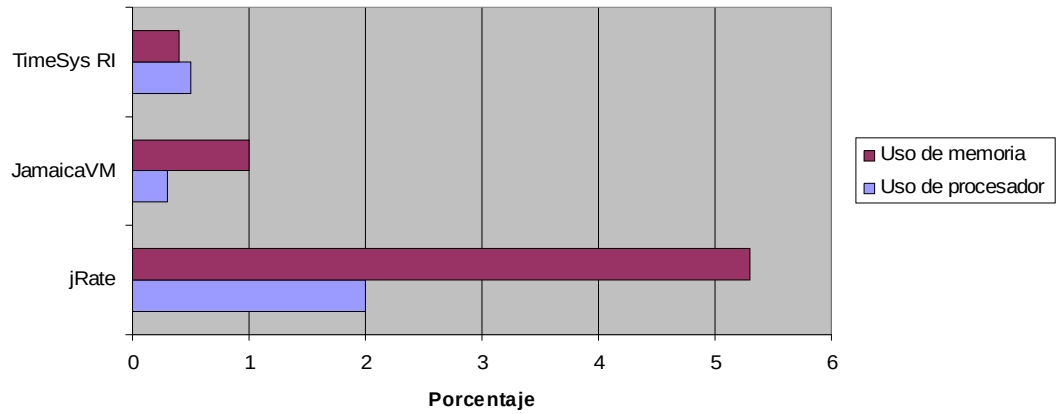


Figura 3.25. Uso de recursos de memoria y procesador

3.7.4 Análisis de resultados

3.7.4.1 Variación en el tiempo de activación de una tarea

Los valores de variación en la activación de una tarea periódica permiten determinar el nivel de estabilidad en el comportamiento de la implementación de la *RTSJ*. Un comportamiento inestable implica que no será predecible y, por tanto, no adecuada. Cabe mencionar que en todos los casos, **el valor promedio no supera el periodo establecido**, lo cual es un buen indicativo, pero el rendimiento de *jRate* y *TimeSys* es más satisfactorio, porque hay menos variación que en el caso de *JamaicaVM*.

En las figuras 3.9-3.13, representamos el comportamiento del periodo para 100, 300, 500, 700, y 900 milisegundos, respectivamente. Cada una de las líneas corresponde a la ejecución de un hilo con una de las implementaciones de la *RTSJ*.

En general, *jRate* se aproxima más al periodo establecido aunque experimenta variaciones considerables para 100 y 700 milisegundos. Para los demás periodos, el comportamiento es muy estable. Aunque *TimeSys RI* no se aproxima tanto como *jRate*, no experimenta variaciones tan

acentuadas, con excepción del periodo de 700 milisegundos, y en menor proporción para 900 milisegundos. *TimesSys RI* presenta una estabilidad bastante aceptable. *JamaicaVM* presenta el comportamiento menos satisfactorio. Además de que los valores del periodo son los más distantes de lo establecido, experimenta variaciones notorias en todos los periodos.

En las figuras 3.14-3.18, representamos los valores promedio del periodo junto con la desviación estándar, para cada uno de los periodos mencionados. Con respecto al periodo promedio, *TimeSys RI* y *jRate* tienen valores similares, muy cercanos al periodo establecido; mientras que, el valor promedio obtenido por *JamaicaVM*, es muy distante en comparación con los anteriores.

Podemos apreciar que *jRate* presenta una menor variación, seguida muy de cerca de *TimeSys RI*. Por último, *JamaicaVM* resulta con valores altos de desviación respecto a las otras herramientas. El desempeño de *jRate* y *TimeSys RI* es mucho más satisfactorio que el de *JamaicaVM*.

3.7.4.2 Tiempo de ejecución de una tarea

La figura 3.20 representa el tiempo de ejecución utilizado por cada herramienta con respecto a la carga de procesamiento. Hasta cierto punto resulta extraño que el tiempo consumido para ejecutar la misma tarea varíe de manera tan considerable. Los resultados nos dan una idea clara acerca del rendimiento de cada herramienta. Mientras mayor es la carga de procesamiento, el tiempo de ejecución es mucho mayor con *JamaicaVM*, y muy distante con respecto a las otras herramientas.

Las diferencias en cuanto a rendimiento se atribuyen al hecho de que si **la ejecución se produce de forma directa o por intermedio de una máquina virtual**. ***jRate* no utiliza máquina virtual**, el programa ejecutable de código nativo es generado a partir de la compilación. Así, la ejecución es directa por parte del sistema operativo y el rendimiento (velocidad) es mayor.

TimeSys RI utiliza una máquina virtual de tiempo real, lo cual significa que existe interpretación de código y, por tal motivo, el rendimiento es menos eficiente. Por último, *JamaicaVM* también es una máquina virtual de tiempo real, sin embargo, su rendimiento es mucho más lento que *TimeSysRI*. La explicación está relacionada con el manejo de prioridades, analizado en la sección 3.7.5.

3.7.4.3 Plazos de ejecución perdidos

Las figuras 3.22-3.24, representan el número de plazos perdidos por cada una de las herramientas durante la ejecución de un hilo de tiempo real con un periodo de 200, 300, y 400 milisegundos, respectivamente. Los resultados obtenidos son consecuentes con los resultados del experimento anterior, *i.e.* mientras mayor es el tiempo de ejecución utilizado, mayor es el número de plazos perdidos.

El bajo rendimiento de *JamaicaVM* provoca una gran cantidad de plazos perdidos, mientras que *jRate* y *TimeSys RI*, en ese orden, lograron tiempos de ejecución menores, razón por la cual, producen menos plazos perdidos. Cabe mencionar que, conforme aumenta el periodo y el plazo (asignamos un plazo igual a la mitad del periodo), observamos que se producen menores plazos perdidos por cada una de las herramientas.

3.7.4.4 Uso de memoria y procesador

La figura 3.25 presenta un cuadro comparativo del porcentaje de memoria y de procesador utilizados por cada instancia de las herramientas. Los valores para *jRate* indican una mayor demanda de recursos. Cabe recordar que cuando una aplicación utiliza demasiados recursos de la computadora, se refleja directamente en el desempeño de la misma. Este fenómeno puede ser desventajoso cuando existan otro tipo de tareas ejecutándose al mismo tiempo.

Por otra parte, *TimeSys RI* y *JamaicaVM* producen valores menores de consumo de memoria y procesador, por debajo del 1%; sin embargo, en el primer caso, estos valores están prácticamente equilibrados, aproximadamente en 0.5 %, lo cual permite concluir que *TimeSys RI* tiene un mayor grado de eficiencia.

Es importante señalar que los resultados están en función de las características técnicas de la computadora utilizada así como también si se ejecutan una o varias tareas adicionales.

3.7.5 Manejo de prioridades

Durante la ejecución de las pruebas, hemos detectado aspectos importantes relacionados con el valor de prioridad que *Linux* otorga a cada una de las tareas en ejecución. Tales particularidades son detalladas a continuación.

3.7.5.1 Planificación a nivel de la *RTSJ*

La *RTSJ* determina que todas las implementaciones deben proveer un planificador basado en prioridad fija con no menos de 28 prioridades [37]. Sin embargo, cada implementación puede ofrecer su propio rango de prioridades.

En la figura 3.26 aparece el código incluido en los programas de prueba que permite verificar el planificador utilizado así como los valores de prioridad máxima y mínima de cada planificador. **Un mayor valor de prioridad indica preferencia de ejecución sobre un valor menor de prioridad**.

El planificador de prioridad de la *RTSJ* es una instancia de la clase `PriorityScheduler` y usa la clase `PriorityParameters` para determinar la elegibilidad de ejecución de los hilos de tiempo real. Los valores de prioridad son los retornados por los métodos `getMinPriority()` y `getMaxPriority()`.

```
// Establecer prioridad para un hilo
   int pri = PriorityScheduler.instance().getMinPriority();
   PriorityParameters prip = new PriorityParameters(pri);

// Nombre del planificador estándar y prioridad del hilo
   this.getScheduler().getPolicyName();
   this.getPriority();

// Valores mínimo y máximo de prioridades del planificador estándar
   PriorityScheduler.instance().getMinPriority();
   PriorityScheduler.instance().getMaxPriority();
```

Figura 3.26: Código para obtener el planificador y sus valores de prioridad

La tabla 3.6 muestra la política de planificación que incluye cada una de las herramientas así como los valores de prioridad máxima y mínima correspondientes:

	Política de Planificación	***Prioridad***	
		Mínima	***Máxima***
jRate	PRIORITY_FIFO_SCHED	1	99
JamaicaVM	Fixed Priority	11	38
TimeSys RI	Fixed Priority	11	90

Tabla 3.6: Política y valores de prioridad máxima y mínima

Aunque el nombre de la política de planificación es diferente para *jRate*, verificamos que todas las herramientas implementan el planificador basado en prioridad. *TimeSys RI* ofrece un mayor rango de prioridades mientras que *JamaicaVM* cumple con el estándar exigido de al menos 28. En ambos casos, se respeta que *Java* estándar maneja prioridades entre 1 y 10. En el caso de *jRate*, los valores corresponden al rango permitido por *Linux*. Enseguida comparamos el valor de prioridad asignado al hilo de ejecución en el código fuente frente al valor presentado por el comando *top*.

	Prioridad	
	Asignada en el programa	***Comando top***
jRate	$1 \leq$ prioridad ≤ 99	-6
JamaicaVM	$11 \leq$ prioridad ≤ 38	24
TimeSys RI	$11 \leq$ prioridad ≤ 90	$-2 \leq$ prioridad ≤ -81

Tabla 3.7: Valores de prioridad asignados en el código fuente *vs.* los asignados por *Linux*

En la tabla 3.7 observamos que los valores de prioridad especificados en el código fuente no equivalen a los mostrados por el comando *top*. **La planificación final del hilo es realizada por el sistema operativo**. De tal forma, **es importante determinar si existe algún tipo de correspondencia entre estos valores**. Un mapeo de prioridades resulta muy útil para el

desarrollador ya que podrá tener control sobre la planificación de las tareas y sus prioridades desde el ambiente de programación. El siguiente punto nos permite clarificar esta idea.

3.7.5.2 Planificación a nivel de *Linux*

El comando `strace` de *Linux* efectúa un seguimiento de cada uno de los programas en ejecución. Este comando permite visualizar las llamadas al sistema realizadas durante la ejecución de un programa sobre *Linux*.

```
1173310973.315639 brk(0)                  = 0x804ddb4
1173310973.315739 brk(0x804edb4)          = 0x804edb4
1173310973.315798 brk(0x804f000)          = 0x804f000
1173310973.315972 sched_get_priority_min(0x1) = 1
1173310973.316029 sched_get_priority_max(0x1) = 99
1173310973.316089 getpid()                = 13072
1173310973.316145 sched_setscheduler(0x3310, 0x1, 0xbffff018) = 0
1173310973.317270 open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
1173310973.317423 fstat64(3, {st_dev=makedev(3, 5), st_ino=468671, st_mode=S_IFREG|(
1173310973.317579 mmap2(NULL, 2097152, PROT_READ, MAP_PRIVATE, 3, 0) = 0x406d2000
1173310973.317684 mmap2(NULL, 4096, PROT_READ, MAP_PRIVATE, 3, 0x37e) = 0x405a3000
1173310973.317741 close(3)                = 0
```

Figura 3.27: Extracto del archivo de salida generado por `strace` para *jRate*

```
1173894686.732789 brk(0)                  = 0x829b8b4
1173894686.732836 brk(0x829c000)          = 0x829c000
1173894686.732924 set_thread_area({entry_number:-1 -> 6, base_addr:0x829ab20, limit:1048575, seg_
1173894686.733056 getpid()                = 13665
1173894686.733123 rt_sigaction(SIGRTMIN, {0x401a5e10, [], SA_RESTORER, 0x4008f4a8}, NULL, 8) = 0
1173894686.733215 rt_sigaction(SIGRT_1, {0x401a5e60, [], SA_RESTORER, 0x4008f4a8}, NULL, 8) = 0
1173894686.733287 rt_sigaction(SIGRT_2, {0x401a67f0, [], SA_RESTORER, 0x4008f4a8}, NULL, 8) = 0
1173894686.733366 rt_sigprocmask(SIG_BLOCK, [RTMIN], NULL, 8) = 0
1173894686.733453 _sysctl({{CTL_KERN, KERN_VERSION}, 2, 0xbfffde7c, 31, (nil), 0}) = 0
1173894686.733621 getpid()                = 13665
1173894686.733681 sched_setscheduler(0x3561, 0x1, 0xbfffe16c) = 0
1173894686.733860 syscall_434(0x1, 0xffffffffe, 0xbfffe0e4, 0x8262740, 0x8276ca0, 0xbfffe0b0, 0xff:
1173894686.733978 syscall_434(0x1, 0xffffffffe, 0xbfffe0e4, 0x8276ca0, 0x8276ca0, 0xbfffe0b0, 0xff:
1173894686.734090 syscall_434(0x1, 0xffffffffe, 0xbfffe178, 0x8276cd0, 0xbffffa98, 0xbfffe0fc, 0xf:
1173894686.734179 uname({sysname="Linux", nodename="localhost.localdomain", release="2.4.20-8", ve
1173894686.734335 uname({sysname="Linux", nodename="localhost.localdomain", release="2.4.20-8", ve
```

Figura 3.28: Extracto del archivo de salida generado por `strace` para *TimeSys RI*

```
[pid 13790] 1173895993.212774 getcwd( <unfinished ...>
[pid 13780] 1173895993.212838 <... nanosleep resumed> NULL) = 0
[pid 13790] 1173895993.212892 <... getcwd resumed> "/opt/Test/periodicThread", 4096) = 25
[pid 13780] 1173895993.212950 nanosleep({0, 20000000},  <unfinished ...>
[pid 13790] 1173895993.213085 uname({sysname="Linux", nodename="localhost.localdomain", release="2
[pid 13790] 1173895993.213273 uname({sysname="Linux", nodename="localhost.localdomain", release="2
[pid 13790] 1173895993.213436 uname({sysname="Linux", nodename="localhost.localdomain", release="2
[pid 13790] 1173895993.225156 sched_setscheduler(0x35dd, 0, 0x40419f00) = 0
[pid 13790] 1173895993.225320 futex(0x8a22210, FUTEX_WAKE, 1, {1, 0}) = 1
[pid 13789] 1173895993.225397 <... futex resumed> ) = 0
[pid 13790] 1173895993.225496 futex(0x8a32620, FUTEX_WAIT, 0, NULL <unfinished ...>
[pid 13789] 1173895993.225568 futex(0x8a32620, FUTEX_WAKE, 1, {1, 0) <unfinished ...>
[pid 13790] 1173895993.225616 <... futex resumed> ) = -1 EAGAIN (Resource temporarily unavailable)
[pid 13789] 1173895993.225665 <... futex resumed> ) = 0
```

Figura 3.29: Extracto del archivo de salida generado por `strace` para *JamaicaVM*

Las figuras 3.27-3.29 muestran extractos de las salidas producidas por este comando. Hemos resaltado en color los renglones que más interesan. En cada línea resaltada se produce una llamada al sistema para seleccionar un *planificador* (*scheduler*) que controle la ejecución del programa. Se trata de la función del sistema `sched_setscheduler()`, misma que establece la *política de planificación* así como los parámetros asociados al proceso. Esta función está definida de la siguiente manera:

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)
```

Donde:

`pid` es la identificación del proceso;

***`policy`* define el algoritmo de planificación o planificador seleccionado;**

`param` representa una estructura de datos necesarios para el planificador. Tal estructura se define de la siguiente manera (archivo `sched.h`):

```
struct sched_param {
      int sched_priority;
};
```

Específicamente, para la versión 2.4 del *kernel* de *Linux*, este último parámetro equivale a la prioridad asignada al proceso.

En la figura 3.29, correspondiente a la ejecución con *JamaicaVM*, observamos que la llamada a la función `sched_setscheduler()` tiene como segundo parámetro un valor igual a *0*; mientras que para *jRate* y *TimeSys* (figuras 3.27 y 3.28), el segundo parámetro tiene un valor en hexadecimal igual a *0x1* que equivale a *1* en entero. Estos valores se corresponden con las definiciones dentro del código fuente de *Linux*, en el archivo de cabecera `sched.h` :

```
/*
 * Scheduling policies
 */
#define SCHED_OTHER           0
#define SCHED_FIFO            1
#define SCHED_RR              2
```

De esa forma, *Linux* admite tres políticas de planificación: **dos para aplicaciones en tiempo real denominadas**:

- *SCHED_FIFO* (*first in first out* en tiempo real)
- *SCHED_RR* (*round robin en* tiempo real)

Y una para otros procesos o procesos normales:

- *SCHED_OTHER* (*time sharing*)

El planificador predeterminado y utilizado por la mayoría de procesos es *SCHED_OTHER*, el cual se basa en tiempo compartido. En contraste, las tareas manejadas por *SCHED_FIFO* en tiempo real tienen la prioridad más alta y sólo pueden ser desalojadas por otra tarea del mismo

tipo que se encuentre lista. Las tareas planificadas por *SCHED_RR* en tiempo real son iguales, excepto que pueden ser desalojadas por el reloj. Si varias tareas con este planificador están listas, cada una se ejecuta durante su *cuanto* de tiempo (número de tics de reloj durante los cuales puede ejecutarse el proceso como máximo) [15].

Con esta información, verificamos que *JamaicaVM* utiliza el planificador estándar de *Linux* denominado *SCHED_OTHER*, por lo que la ejecución del programa es considerado como un proceso normal. En cambio, *jRate* y *TimeSys RI* utilizan el planificador especial *SCHED_FIFO*, destinado para procesos de tiempo real y cuya prioridad toma un valor negativo. Una tarea ejecutada con este planificador siempre prevalecerá sobre cualquier otra tarea normal y no será interrumpida. Este hecho explica que la ejecución del programa de prueba con *TimeSys RI* sea mucho más rápida que con *JamaicaVM*, aunque las dos herramientas utilicen una máquina virtual. Por tanto, establecemos que existen dos niveles de planificación en la ejecución de una aplicación *Java* de tiempo real:

- **Uno de alto nivel, manejado por la *RTSJ*.**
- **Otro de bajo nivel, manejado por el sistema operativo**.

De acuerdo con la tabla 3.7, en el caso de *jRate* no existe un mapeo de prioridades. El valor de prioridad asignado en el programa no influye en el valor que asigna el planificador del sistema operativo ya que siempre utiliza el valor de -6; sin embargo, este valor nos indica que el proceso es tratado como uno de tiempo real y tiene preferencia de ejecución sobre los demás. Para *JamaicaVM*, tampoco hay un mapeo de prioridades. El planificador es el estándar de *Linux* y generalmente asigna un valor de prioridad igual a 24 sin importar el valor que se especifique en el programa. Solamente **en el caso de *TimeSys RI* existe un mapeo de prioridades entre el planificador de la *RTSJ* y el del sistema operativo**.

La figura 3.30 ilustra el mapeo identificado. Se definen 3 rangos de prioridades: el primero corresponde a *Java* estándar con valores entre 1 y 10. A partir del valor 11, se encuentra el rango de prioridades de tiempo real utilizadas por *TimeSysRI*, con un valor máximo de 90. Si se especifica un valor mayor en el programa ocurrirá un error. Finalmente, en el lado negativo tenemos las prioridades que *Linux* obtiene a partir de las definidas por *TimeSys RI*, demostrando así que existe una correspondencia.

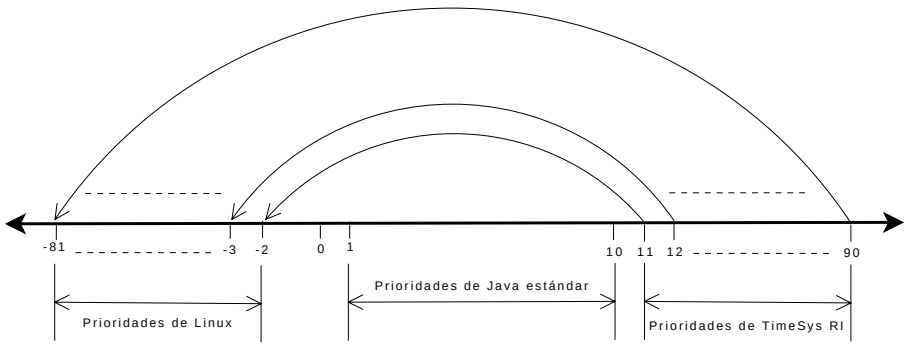


Figura 3.30: Mapeo entre las prioridades de *TimeSys RI* y *Linux*

3.7.6. Resumen de desempeño y conclusiones

En la tabla 3.8 presentamos un compendio de los parámetros evaluados y una calificación que determina si una herramienta es adecuada para sistemas de tiempo real de tipo flexible o estricto:

	Parámetros									
	Periodo y desviación estándar		Tiempo de ejecución		Plazos perdidos		Uso de recursos		Mapeo de Prioridades	
	TRE	TRF	TRE	TRF	TRE	TRF	TRE	TRF	TRE	TRF
jRate	x	√	x	√	x	√	√	√	x	x
JamaicaVM	x	x	x	x	x	x	√	√	x	x
TimeSys RI	x	√	x	√	x	√	√	√	x	√

TRE: tiempo real estricto
TRF: tiempo real flexible

Tabla 3.8. Resumen de desempeño

En general, *jRate* es la implementación con mejor desempeño durante las pruebas, seguida por *TimeSys RI*. Hemos comprobado que *JamaicaVM* junto con *Linux* estándar no es una combinación adecuada para tiempo real. Esta herramienta requiere definitivamente un sistema operativo específico de tiempo real.

Por otra parte, el manejo de prioridades y planificadores es determinante para el comportamiento de las herramientas. Hemos identificado dos niveles de planificación: 1) de alto nivel, manejado por la *RTSJ*; y 2) de bajo nivel, a cargo de *Linux.*

La existencia de una correspondencia entre ambas es importante para que un desarrollador pueda tener control sobre las prioridades manejadas por el sistema operativo, ciertamente una gran ventaja para aplicaciones que tienen requisitos de tiempo. En este sentido, *jRate* es superada por la implementación de *TimeSys*, cuyo mapeo de prioridades ha sido ilustrado.

En [35] se menciona que un sistema operativo puede ser considerado adecuado para tiempo real flexible si al menos: el sistema tiene planificación basada en prioridad y si los procesos de tiempo real tienen la más alta prioridad. Hemos comprobado que ***Linux* conjuntamente con *TimeSys RI* cumplen dichos criterios así que su combinación es viable para ser utilizada con aplicaciones de tiempo real flexibles**, en las cuales se pueden incumplir restricciones de tiempo ocasionalmente.

Por último, *jRate* es una herramienta que no utiliza una máquina virtual y el código fuente debe ser compilado. De esta forma, la portabilidad característica de *Java* no es aprovechada. ***TimeSys RI* utiliza la filosofía de la máquina virtual de *Java* estándar y, por ende, permite la portabilidad de aplicaciones**.

De acuerdo con los resultados obtenidos y las facilidades que brinda cada una de las

herramientas, determinamos que la combinación de *TimeSys RI* y *Linux* es adecuada para aplicaciones de tiempo real flexibles (*soft*), y por ende, será la opción seleccionada para el desarrollo de nuestra solución.

La integración de las herramientas analizadas en este capítulo como: *sockets*, el modelo de *Java RMI*, programación concurrente, y la combinación de *Timesys RI* y *Linux*, conformarán la plataforma para el desarrollo y la ejecución de aplicaciones distribuidas de tiempo real sobre un sistema distribuido. Precisamente, **el mayor reto del trabajo de tesis fue acoplar dichas herramientas para que trabajen conjuntamente** y alcancen la funcionalidad requerida.

Capítulo 4

Diseño de la solución

En este capítulo describimos la secuencia de pasos que permitirá crear la propuesta de solución, para lo cual, comenzamos planteando de manera concisa el problema abordado. A continuación definimos la arquitectura de solución donde analizamos los componentes que intervienen así como la interacción entre ellos. Incluimos una serie de diagramas que representan el diseño conceptual y funcional de la solución para, posteriormente, implementarla en lenguaje de programación *Java*.

4.1 Descripción del problema

En el dominio de tiempo real, el uso de aplicaciones distribuidas se ha incrementado considerablemente. Muchas aplicaciones de tiempo real, *e.g.*, en el campo de la automatización industrial, militar, aviónica, y telecomunicaciones, deben ejecutarse en un ambiente distribuido, para lo requieren de componentes denominados *middlewares*.

Un *middleware* convencional como *Java RMI* permite la comunicación y la ejecución de aplicaciones distribuidas de **propósito general**; sin embargo, no ofrece el soporte necesario para aplicaciones que tienen requisitos de tiempo real tales como: parámetros de planificación y parámetros de tiempo.

Actualmente, la comunidad *Java* realiza esfuerzos de investigación para crear una especificación que extienda el modelo de *Java RMI* con el fin de obtener una versión de tiempo real de este *middleware*. Aunque existen algunos trabajos de investigación donde se exponen diversas aproximaciones, hasta el momento no se ha emitido una especificación.

En el presente trabajo de tesis proponemos un mecanismo que aproveche la actual ***RTSJ*** que, a pesar de que **está diseñada para trabajar en sistemas con un único procesador**, puede ser integrada con otras herramientas tales como: *sockets*, el modelo de *Java RMI*, programación *multi-hilo* (concurrencia), para propagar los requerimientos de tiempo real entre los diversos nodos de un sistema distribuido y, de esta manera, permitir la ejecución de aplicaciones distribuidas de tiempo real. Se debe considerar que una aplicación, cuyo comportamiento involucra más de un nodo de procesamiento, **debe propagar el contexto de ejecución**

(parámetros de planificación y de tiempo) **de cada uno de los hilos** que conforman la aplicación a los nodos involucrados en el sistema distribuido.

4.2 Arquitectura de solución

La arquitectura del sistema deberá permitir establecer la estructura en términos de componentes y sus interacciones. La mayoría de los sistemas distribuidos están basados sobre una arquitectura *cliente-servidor*, razón por la cual, utilizamos este esquema para identificar los componentes del sistema.

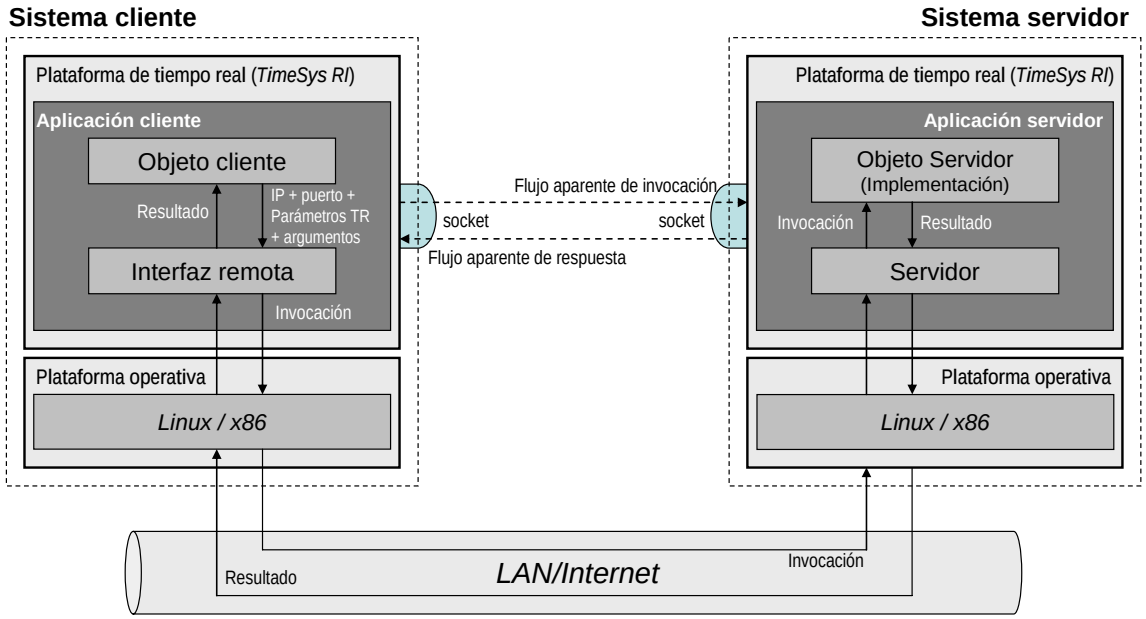


Figura 4.1: Arquitectura de la solución

La figura 4.1 presenta la arquitectura de la solución, utilizando como base el modelo *cliente-servidor*. Procuramos una división de responsabilidades entre clientes y servidores, estableciendo un *sistema cliente* y un *sistema servidor*. Cada uno puede encontrarse en una computadora distinta, en la misma computadora o, incluso, los sistemas pueden intercambiar roles. **El mecanismo de solución funciona para cualquiera de los casos mencionados**.

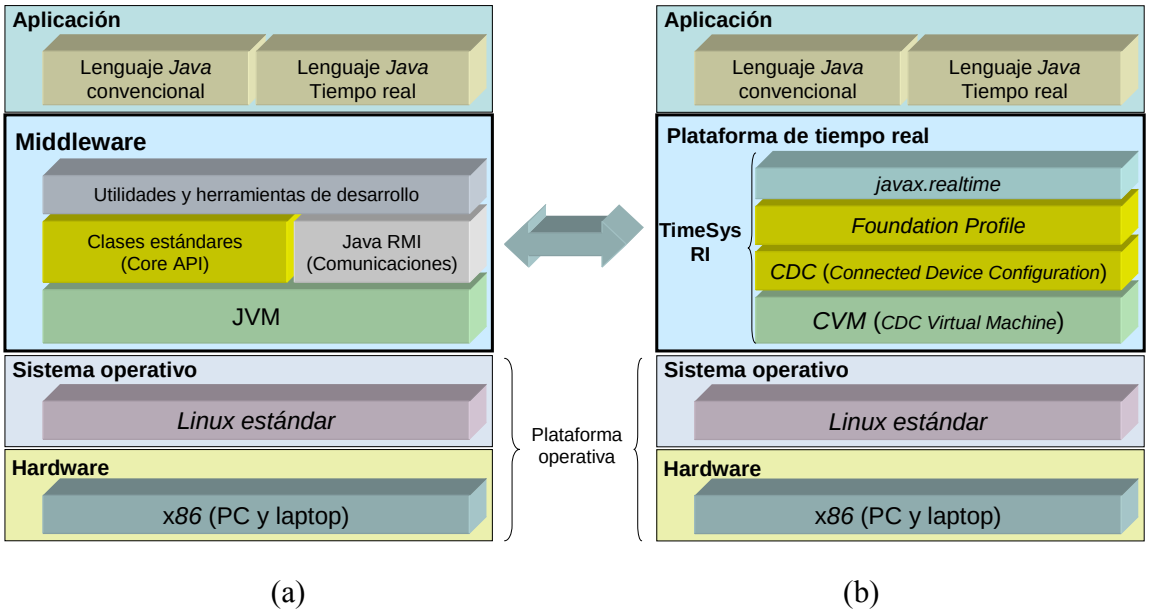
Tanto el sistema cliente como el servidor comparten dos componentes en común: la ***plataforma operativa*** y la ***plataforma de tiempo real***. Por otro lado, la aplicación distribuida se divide en la ***aplicación cliente*** y la ***aplicación servidor***.

4.2.1 Plataforma operativa y plataforma de tiempo real

La conjunción de estos dos componentes provee la infraestructura que permite la ejecución de una aplicación distribuida de tiempo real. Gracias a las pruebas desarrolladas en el capítulo

anterior, establecimos que la combinación de la máquina virtual de tiempo real *TimeSys RI* y el sistema operativo *Linux* es adecuada para sistemas de tiempo real flexibles y, por tal razón, es la plataforma de ejecución seleccionada para la solución propuesta.

Por tanto, es importante señalar que la solución no utiliza un *middleware*. Las aplicaciones cliente y servidor se ejecutan cada una sobre una *máquina virtual de Java de tiempo real* (*JVM-RT*) separada, mismas que se comunican por medio de *sockets*.



(a) (b)

Figura 4.2: Modelo de capas de un *SD* convencional *vs.* la solución propuesta

En la figura 4.2 se puede comparar un modelo de capas de un sistema distribuido convencional (a) frente al modelo utilizado en la solución (b). El nivel de *middleware* correspondiente a *Java RMI*, es reemplazado por *TimeSys RI*, la implementación de la *RTSJ*. Uno de los aspectos claves de la solución ha sido **sustituir el *middleware* por la máquina virtual de *Java* de tiempo real (*JVM-RT*) y la funcionalidad de *sockets***. Así logramos el soporte necesario para la ejecución de la aplicación distribuida de tiempo real.

TimeSys RI está basada en la plataforma *J2ME CDC* (*Java 2 Micro Edition Connected Device Configuration*), diseñada para dispositivos móviles y empotrados. No incluye la biblioteca de clases para *RMI* de acuerdo con los lineamientos de la *RTSJ*, *i.e.* únicamente para trabajar en un solo procesador. **A pesar de múltiples intentos realizados para anexar el paquete `java.rmi`, no fue posible lograr tal funcionalidad**. La figura 4.2 también muestra los componentes que conforman *TimeSys RI*:

- *CVM* (*CDC Virtual Machine*): una versión optimizada de máquina virtual de *Java* pero que soporta todas las características y funcionalidades de una *JVM* estándar. Contiene las *API*s estándares, mismas que se presentan en dos conjuntos: *CDC* y *Foundation Profile*.

- *CDC* (*Connected Device Configuration*): es un subconjunto de *APIs* de la *J2SE* (*Java 2 Standard Edition*) e incorpora el perfil básico denominado *Foundation Profile*.

- *Foundation Profile*: un conjunto de *APIs* que proporcionan el soporte para trabajo en red así como para funciones de entrada/salida.

- `javax.realtime`: *API* para tiempo real definida por la especificación *RTSJ*.

Cada uno de los nodos que conforman el sistema distribuido de tiempo real deben estar constituidos por las capas del modelo expuesto anteriormente. Por supuesto, todos los nodos deben estar conectados mediante una red con el protocolo *TCP/IP*.

Para resolver el problema de comunicación en red, hemos utilizado la *API* de *sockets* que ofrece *Java*. De esta forma, es posible conseguir la comunicación entre máquina virtuales de tiempo real que se ejecutan en computadoras separadas. Así, conseguimos propagar entre ellas los parámetros de tiempo real tal como se explica enseguida.

4.2.2 Aplicación distribuida

Constituida por:

- La ***aplicación cliente*** puede contener uno o varios objetos (***objetos clientes***) y necesita de una ***interfaz remota*** para conocer los métodos del objeto remoto que puede invocar.

- La ***aplicación servidor*** contiene un módulo del mismo nombre (***servidor***) que espera solicitudes de conexión, recibe la invocación de parte del cliente y la transmite al ***objeto servidor*** quien realmente invoca el método especificado y, si es el caso, devuelve un resultado al cliente.

Con base en la arquitectura descrita, detallamos el funcionamiento global del sistema. El propósito es que un objeto (denominado cliente) que se ejecuta sobre una *JVM-RT* pueda invocar un método de un objeto (denominado servidor) que se ejecuta sobre una *JVM-RT* separada (generalmente en otra computadora física).

Este proceso difiere de la invocación de método remoto convencional en que la ejecución debe efectuarse bajo un determinado contexto que incluye parámetros de planificación y de tiempo, mismos que deben ser cumplidos. Para tal efecto:

- El ***objeto cliente*** ejecutándose sobre una *JVM-RT* (*TimeSys RI*) realiza una invocación de un método implementado en un ***objeto servidor***, este objeto se ejecuta en una *JVM-RT*. La invocación incluye el ***número de puerto***, la ***dirección IP*** del servidor, los parámetros que conforman el ***contexto de ejecución***. Pueden existir argumentos opcionales para la ejecución del método remoto, de acuerdo con las características propias de cada aplicación.

- La invocación, en cuanto a sintaxis, es similar a una invocación local, ***i.e.*** tal como si el método del *objeto servidor* se ejecutase en la máquina virtual local. Sin embargo, lo que realmente sucede es que la invocación es recibida por la ***interfaz remota***, quien representa al *objeto servidor*, y por tanto, contiene la declaración del método remoto.

- La *interfaz remota* se desempeña como un **intermediario** durante todo el proceso. Recibe los datos de conexión (*dirección IP* y *puerto* en el *servidor*), los parámetros de tiempo real y, si es el caso, argumentos opcionales. Crea el *socket* de comunicación y transmite el mensaje (invocación) que incluye todos los datos mencionados.

- El componente denominado ***servidor***, ejecutándose sobre la *JVM-RT*, se encuentra a la espera de conexiones entrantes. Una vez aceptada la conexión, lee el mensaje enviado por el *objeto cliente*.

- El *servidor* es quien verdaderamente realiza la invocación del método en el *objeto servidor*. Cabe mencionar que los parámetros son pasados como tipos de datos primitivos (***e.g.*** enteros y cadenas de caracteres) y, una vez que son recibidos por el *servidor*, éste se encarga de convertirlos en tipos de datos definidos por la *RTSJ* para que puedan ser tratados por la *JVM-RT*.

- Finalmente, si el método del *objeto servidor* devuelve un resultado, debe ser enviado al cliente siguiendo el mismo camino descrito pero en sentido contrario.

Una vez que hemos determinado los objetos del dominio del problema y su interacción, procedemos a representarlos como entidades lógicas y relaciones estructurales.

4.3 Diagramas *UML*

Un aspecto importante dentro del diseño es la generación de diagramas utilizando *UML* (*Unified Modeling Language*) [38], una notación estándar que ha emergido para describir modelos orientados a objetos [16].

4.3.1 Descomposición del sistema

Hemos dirigido nuestra atención al sistema de software (aplicación) de la solución. Un sistema que, a su vez, puede estar estructurado por sistemas menores que contienen objetos funcionalmente dependientes unos con otros. Estos objetos que están fuertemente acoplados forman parte del mismo sistema y los que están débilmente acoplados están en diferentes sistemas.

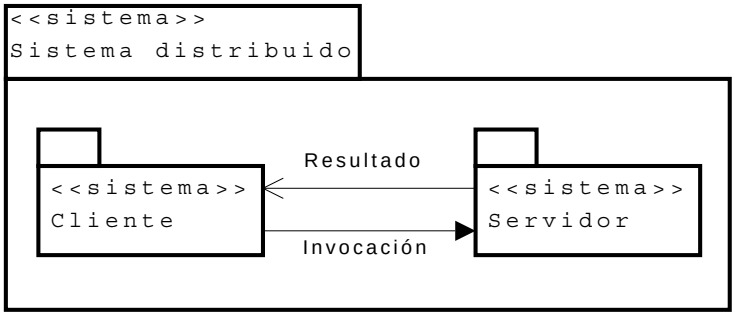


Figura 4.3: Descomposición del sistema

La figura 4.3 muestra la descomposición del sistema total, dividiendo la estructura de la aplicación distribuida en sistemas más pequeños que pueden ejecutarse sobre nodos separados en un ambiente distribuido. Puesto que para el caso de aplicaciones distribuidas se hace énfasis en la división de responsabilidades entre cliente y servidor, éstos constituyen, por sí solos, sistemas que se comunican por medio de mensajes.

Vamos a utilizar los términos *aplicación distribuida*, *aplicación cliente*, y *aplicación servidor*, en lugar de *sistema distribuido*, *sistema cliente*, y *sistema servidor*, debido a que nuestro trabajo enfatiza el diseño de aplicaciones.

4.3.2 Diagrama conceptual

A su vez, cada uno de los sistemas cliente y servidor es descompuesto incluyendo los objetos identificados en la arquitectura de la solución, los cuales son representados como *clases*. Aunque aún no se definen atributos y métodos, incluimos otros detalles como relaciones entre clases y su cardinalidad.

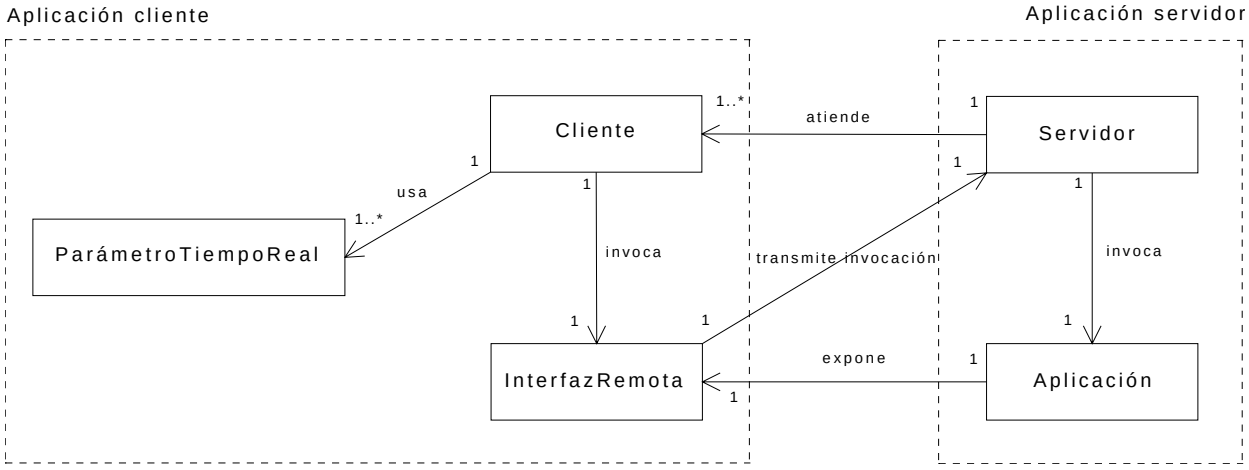


Figura 4.4: Diagrama conceptual

La asociación entre dos clases define una relación, representada por una línea dirigida y sobre la cual se encuentra el nombre de la relación y la cardinalidad en los extremos. La cardinalidad de una relación indica el número de instancias de una clase que están relacionadas a una instancia de la otra clase. En la figura 4.4 se aprecian cardinalidades de *exactamente uno* (1), *cero o más* (0..*), y de *uno o más* (1..*).

4.3.3. Diagrama de interacción

Permite describir cómo los objetos cooperantes interactúan dinámicamente por medio del envío y la recepción de mensajes. Los objetos son instancias activas de las clases especificadas anteriormente.

En la figura 4.5 se representa la secuencia de acciones que los objetos realizan entre sí.

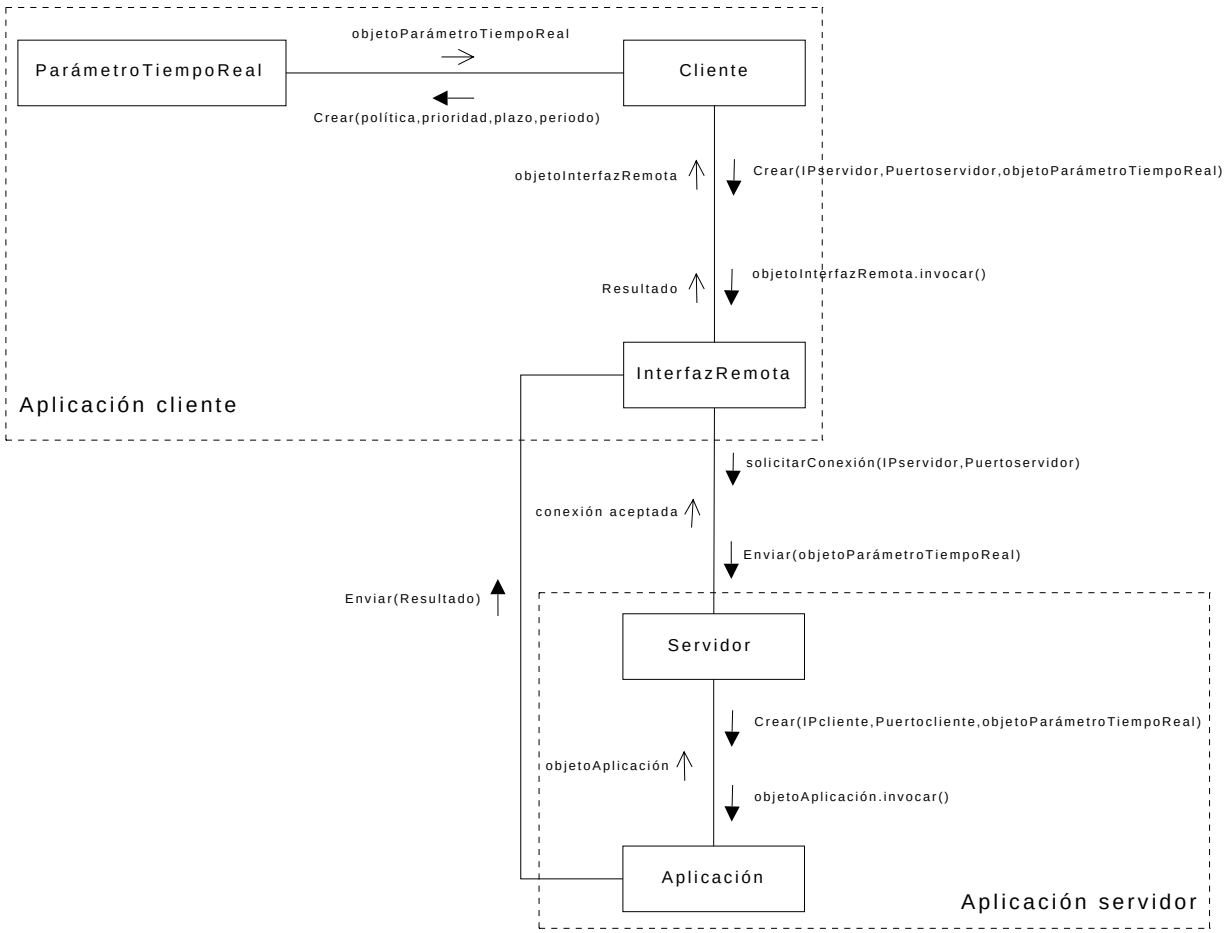


Figura 4.5: Diagrama de interacción

Hasta el momento hemos identificado únicamente los componentes de la aplicación distribuida, sin embargo, debido a que tratamos con aplicaciones distribuidas de tiempo real, debemos considerar algunas necesidades especiales, tales como: la concurrencia, los hilos de ejecución, la comunicación en red y los tipos de datos adecuados para los parámetros de tiempo real. Estas necesidades son consideradas en los siguientes diagramas *UML*.

4.3.4 Diagrama de paquetes

Como mencionamos, hemos definido una parte de la solución, relacionada con las clases que conforman la aplicación distribuida. Estas clases deben trabajar conjuntamente con las clases que provee *Java* estándar tanto para la programación *multi-hilo* como para la comunicación en red. Además debemos integrar las clases de la *RTSJ* para satisfacer los requerimientos de tiempo real.

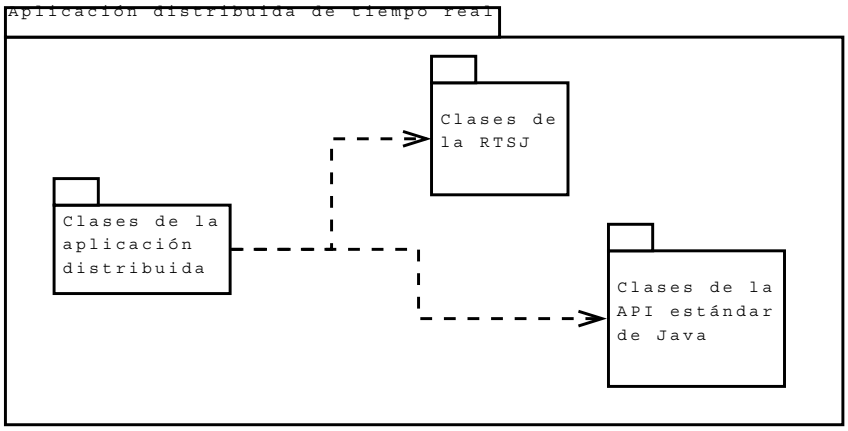


Figura 4.6: Diagrama de paquetes

La figura 4.6 muestra la solución total a nivel lógico (aplicación distribuida de tiempo real) conformada por tres paquetes de clases:

- ***Clases de la aplicación distribuida***, que constituyen el esqueleto para estructurar la aplicación (ver figura 4.4).

- ***Clases de la API estándar de Java*** para el trabajo en red (*sockets*), manejo de entrada/salida (flujos o *streams*) y programación con hilos (*threads*).

- ***Clases de la RTSJ*** (paquete `javax.realtime`).

La línea entrecortada representa una relación de dependencia entre paquetes. En nuestro caso, el paquete de la aplicación distribuida depende de los paquetes de la *API* estándar de *Java* y de la *RTSJ* para obtener las funcionalidades de comunicación, concurrencia y tiempo real.

4.3.5 Diagrama de clases

Una vez que se han determinado todos los componentes de la solución, en la figura 4.7 presentamos el diagrama *UML* de clases, incluyendo los atributos de las clases, las relaciones entre clases y las operaciones de cada clase.

> **Nota:** *Los acentos en los nombres de clases, atributos, y operaciones, han sido omitidos ya que son las entidades que serán implementadas con el lenguaje de programación, el cual no acepta este tipo de caracteres.*

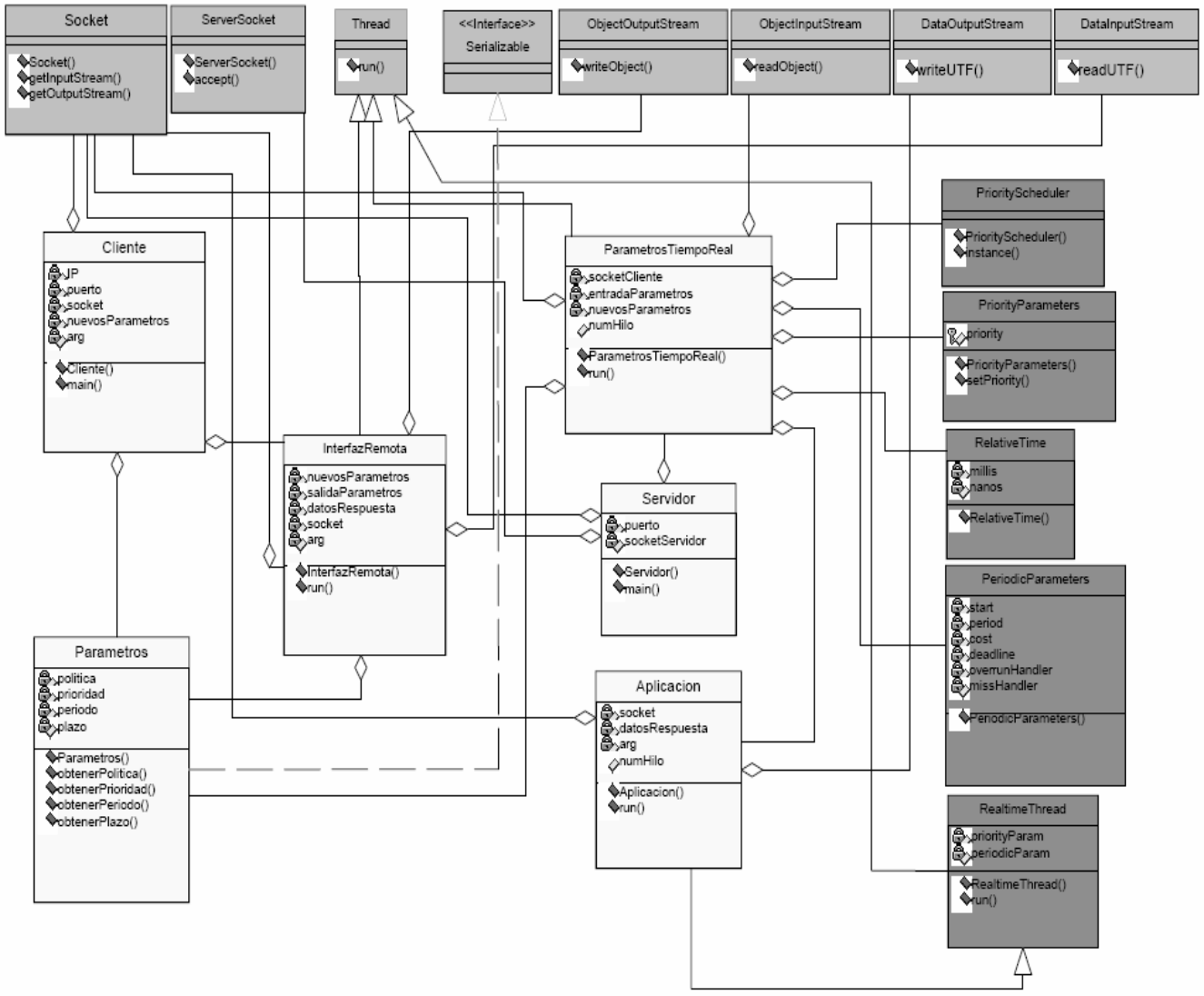


Figura 4.7: Diagrama de clases

4.4 Uso de las clases

Los parámetros que corresponden a la planificación son: la política de planificación y la prioridad de la tarea; mientras que los parámetros relacionados con la activación del hilo de tiempo real son restricciones de tiempo: periodo y plazo de ejecución. La clase `Parametros` encapsula dichos parámetros que definen el contexto de ejecución de la aplicación distribuida. **En el caso de necesitar parámetros adicionales, simplemente deben ser añadidos dentro de esta clase**.

Cabe mencionar que en la *aplicación cliente*, los parámetros de tiempo real corresponden a datos de tipo primitivo como enteros (*int*) y cadenas de caracteres (*strings*). Una vez que son transmitidos y recibidos por el servidor, serán convertidos al tipo de objeto de tiempo real adecuado para la *JVM-RT*. La transmisión de los parámetros es realizada conjuntamente con la invocación del método remoto por parte del cliente. Por tal motivo, la clase `Parametros` implementa la interfaz `Serializable` (paquete `java.io`) con el fin de que sus instancias sean serializables. La *serialización* en *Java* se refiere al proceso de convertir los objetos en secuencias de bytes para que puedan ser transmitidos en un mensaje por la red. La serialización es útil también para solucionar las diferentes representaciones de datos originadas por diferentes arquitecturas de computadoras [2].

La clase `Cliente` puede recibir los parámetros de tiempo real ya sea desde la línea de comandos o desde el código fuente. En ambos casos se crea una instancia de la clase `Parametros`. De la misma forma, el cliente debe conocer la *dirección IP* y el *número de puerto* utilizados por el proceso servidor en la computadora destino. Con estos datos, la clase `Socket` (paquete `java.net`) puede establecer la comunicación con el servidor.

Adicionalmente, la clase `Cliente` utiliza la clase `InterfazRemota` como **intermediario en la invocación del método remoto**. Para tal propósito, crea una instancia de `InterfazRemota` pasándole el *socket* de comunicación, el objeto que contiene los parámetros de tiempo real.

A su vez, `InterfazRemota` es la encargada de serializar el objeto de tipo `Parametros` y transmitirlo por la red al servidor. Para serializar el objeto `Parametros`, se crea el flujo de datos de salida por medio de la clase `ObjectOutputStream` (paquete `java.io`) y se invoca su método `writeObject()`, pasándole el objeto de tipo `Parametros`. Mediante la clase `DataInputStream` (paquete `java.io`), se crea el flujo de datos de entrada para recibir el resultado de la invocación de método remoto.

Es importante destacar que la clase `InterfazRemota` hereda de la clase `Thread`, (paquete `java.lang`); por tanto, una instancia de esta clase se convierte en un hilo de ejecución independiente y el código que ejecutará debe ser implementado en su método `run()`. Este código servirá para acceder al *socket* de comunicación con el servidor para enviar los parámetros de tiempo real y, posteriormente, recibir el resultado de la invocación. **Por tanto, existe un hilo de ejecución por separado para cada conexión de cliente**.

Hemos aprovechado la ***programación de hilos*** (*threads*) para implementar una solución que es ***multi-hilo***, un estilo *ad-hoc* para este tipo de aplicaciones; no sólo porque cada instancia del *objeto cliente* y del *servidor* se encuentran simultáneamente en ejecución, por medio de su método `main()`, sino porque cada una de las conexiones entre cliente y servidor son manejadas por un hilo exclusivo de ejecución permitiendo así que se puedan procesar varias de ellas en **forma concurrente**.

En la *aplicación servidor*, la clase `Servidor` se ejecuta sobre una instancia de la *JVM-RT* (generalmente en otra computadora). Esta clase tiene como propósito esperar por peticiones de conexión de los clientes. La clase `ServerSocket` está diseñada para crear un conector en el *puerto* especificado y que escuchará las peticiones de conexión por parte de los clientes. El método `accept()` toma una petición y se crea una instancia de la clase `ParametrosTiempoReal`, la cual está encargada de las siguientes funciones:

- Extiende de la clase `Thread` para crear un hilo por separado para cada cliente.

- El método `run()` lee los bytes del mensaje desde el flujo de entrada y reconstruye (deserialización) el objeto de tipo `Parametros` enviado desde el cliente. Para tal fin, utiliza la clase `Parametros` y el método `readObject()` de la clase `ObjectInputStream`.

- Realiza la **conversión de datos primitivos** a tipos de **datos de tiempo real**. Los parámetros de política de planificación y prioridad son convertidos a través de las clases de la *RTSJ*: `Scheduler` y `PriorityParameters`, respectivamente; mientras que los parámetros de plazo y periodo son convertidos mediante: `RelativeTime`, y `PeriodicParameters`, respectivamente.

- Por último, se crea una nueva instancia de la clase `Aplicacion` pasándole como argumentos: el *socket* de comunicación y los parámetros de tiempo real en el formato apropiado.

La clase `Aplicacion` hereda de la clase `RealtimeThread` (perteneciente a la *RTSJ*) y permite definir hilos de ejecución que tienen asociados parámetros de planificación y de tiempo. En el método heredado `run()`, el desarrollador debe implementar el código que ejecutará el hilo de tiempo real.

La ejecución del hilo de tiempo real comienza con la llamada al método `start()` que enseguida llama al método `run()`. Aquí es donde realmente se efectúa la invocación solicitada por el cliente. Si el método `run()` devuelve un resultado, se utiliza la clase `Socket` para acceder al *socket* de comunicación y la clase `DataOutputStream` para crear el flujo de datos de salida y escribir el resultado con el método `writeUTF()`. Este método trabaja con el *Formato de Transferencia Universal* (*UTF*), un sistema de codificación independiente de la computadora.

Capítulo 5

Implementación

Después de haber establecido los componentes que conforman la arquitectura del sistema, es el turno de la implementación. Iniciamos con una breve mención acerca del ambiente de implementación que soporta la ejecución de una aplicación distribuida de tiempo real. Después, realizamos la implementación de cada una de las clases que conforman la estructura de la solución, tanto la *aplicación cliente* como la *aplicación servidor*. Finalmente, detallamos el uso y adecuación de un software de visualización que se desempeñará como un supervisor de la ejecución de una aplicación distribuida de tiempo real.

5.1 Ambiente de implementación

La solución propuesta consiste de código tanto en lenguaje Java estándar como lenguaje Java para tiempo real. *TimeSys RI* está basada en la plataforma optimizada *J2ME*, y soporta la ejecución de código de tiempo real y código no tiempo real. La implementación de las clases correspondientes a la aplicación distribuida (identificadas en el diagrama de clases de la figura 4.7), conforman un conjunto de clases *Java* que interactuará con la *API* estándar de Java (*sockets*, *threads*, *Input/Output*) para lograr la funcionalidad deseada.

Es importante mencionar que hemos utilizado la técnica de programación *multi-hilo*, la cual es parte integral de la plataforma *Java* [21] y que, en contraste con la programación secuencial clásica, permite que la solución basada en el esquema *cliente-servidor* pueda realizar varias tareas independientes y de manera simultánea, características demandadas por los entornos distribuidos.

La implementación es realizada sobre la infraestructura de un sistema distribuido, *i.e.* la plataforma operativa y la de tiempo real, identificadas en la arquitectura del sistema.

La figura 5.1 muestra la infraestructura necesaria para permitir la ejecución de aplicaciones distribuidas de tiempo real, donde cada uno de los nodos debe satisfacer el modelo de capas que incluye: plataforma operativa (*Linux x86*), plataforma de tiempo real (*TimeSys RI*), y el nivel de aplicación, del cual se trata a continuación.

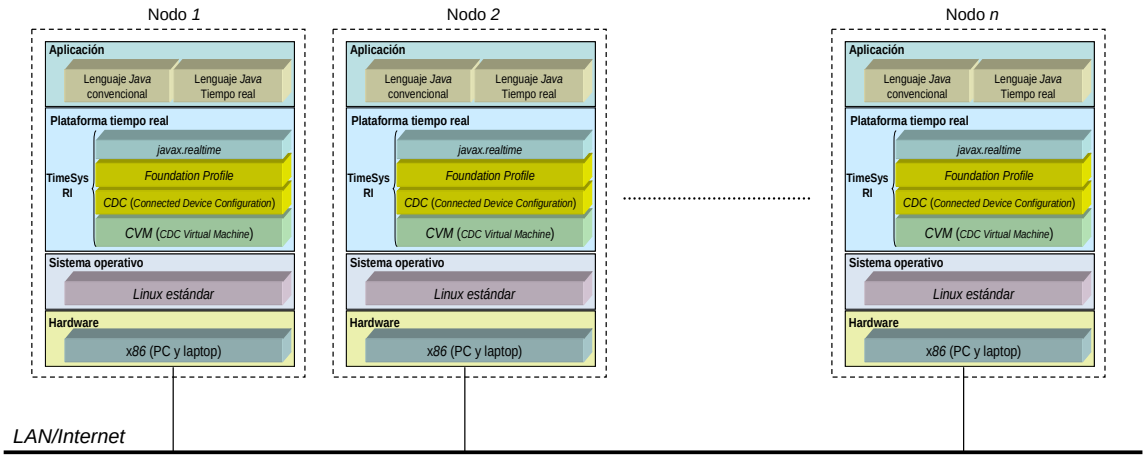


Figura 5.1: Infraestructura del *SD* de tiempo real

5.2 Implementación de clases

En esta etapa del desarrollo, los componentes de programación definidos en la arquitectura de la solución son implementados en lenguaje *Java* como clases de objetos. Partiendo de la división en *aplicación cliente* y *servidor*, los componentes pertenecientes a cada uno de estos sistemas son implementados como archivos de *clase*.

5.2.1 Implementación de la *aplicación cliente*

La *aplicación cliente*, a su vez, está conformado por las siguientes clases: `Parametros`, `Cliente` , `InterfazRemota`.

```
import java.io.Serializable;
public class Parametros implements Serializable
{
  private String politica;
  private int prioridad;
  private int periodo;
  private int plazo;
  private static final long serialVersionUID = 1;
  public Parametros(String politica, int prioridad, int periodo, int plazo)
  {
    this.politica = politica;
    this.prioridad = prioridad;
    this.periodo = periodo;
    this.plazo = plazo;
  }
  public String obtenerPolitica(){return this.politica;}
  public int obtenerPrioridad(){return this.prioridad;}
  public int obtenerPeriodo(){return this.periodo;}
  public int obtenerPlazo(){return this.plazo;}
}
```

Figura 5.2: Código fuente de la clase `Parametros`

La figura 5.2 muestra el código de la clase `Parametros` que permite encapsular los parámetros de planificación (política y prioridad) y las restricciones de tiempo (periodo y plazo) que controlarán la ejecución de la aplicación. Esta clase define una estructura de datos que puede ser modificada o extendida de acuerdo con las necesidades del desarrollador. Se puede apreciar que los tipos de datos de dichos parámetros son tipos de datos básicos o primitivos, mismos que en el servidor serán convertidos al formato de tiempo real.

La clase `Parametros` implementa la interfaz `Serializable` (paquete `java.io`), con el propósito de que un objeto de esta clase pueda ser transmitido en un mensaje por la red. Cuando se pasa un objeto por la red, tanto la computadora que envía como la que recibe, deben contar con el archivo de clase a la cual pertenece el objeto.

Puede suceder que existan distintas versiones de la clase haciendo que la reconstrucción del objeto en el otro lado no sea posible. Para resolver el problema, se utiliza el atributo privado `serialVersionUID` cuyo valor debe ser distinto para cada versión compilada y así *Java* detecta si las versiones en ambos lados son distintas.

Con respecto a los métodos de la clase, el *constructor* permite asignar valores a los atributos cuando se instancia la clase. Los restantes métodos sirven para acceder a dichos valores.

```
import java.net.Socket;
public class Cliente
{
 private String IP;
 private int puerto;
 private Parametros nuevosParametros;
 private Socket socket;
 public static void main(String[] args)
 {
   if(args.length != 6) {
     System.out.println("Uso: Cliente <IP> <puerto> <politica> <prioridad> <periodo> <plazo> ");
      System.exit(1);
   }
   String args0 = args[0];                    /* IP servidor */
   int args1    = Integer.parseInt(args[1]); /* puerto */
   String args2 = args[2];                    /* politica */
   int args3    = Integer.parseInt(args[3]); /* prioridad */
   int args4    = Integer.parseInt(args[4]); /* periodo */
   int args5    = Integer.parseInt(args[5]); /* plazo */
   Cliente cliente = new Cliente(args[0], args1, args[2], args3, args4, args5);
 }

 public Cliente(String IP,int puerto,String politica,int prioridad,int periodo,int plazo)
 {
   this.IP = IP;
   this.puerto = puerto;
   this.nuevosParametros = new Parametros(politica, prioridad, periodo, plazo);
   try
   {
     socket = new Socket(IP, puerto);
     InterfazRemota hiloTiempoReal = new InterfazRemota(socket, nuevosParametros);
     hiloTiempoReal.start();
   } catch (Exception e){e.printStackTrace();}
 }
}
```

Figura 5.3 Código fuente de la clase `Cliente`

En la figura 5.3 se muestra el código de la clase `Cliente`, en cuyo método `main()`, recibe los datos de *dirección IP* y *número de puerto* para crear el *socket* de conexión con el servidor. Además, recibe los datos que representan los parámetros de tiempo real para crear un objeto del tipo `Parametros`.

En este caso, los datos mencionados son proporcionados desde teclado como argumentos de la *aplicación cliente*, sin embargo, también pueden ser instanciados dentro del código fuente de la clase.

El método constructor de la clase sirve para establecer el *socket* de comunicación con el servidor. Luego, una instancia de la clase `InterfazRemota` actúa como intermediario puesto que este objeto será el encargado de realizar la invocación del método remoto en el *objeto servidor*. Debido a que la clase `InterfazRemota` hereda de la clase `Thread`, la invocación se realiza con el método `start()`, el cual crea un nuevo hilo de ejecución y después invoca el método `run()` definido en `InterfazRemota`.

Con respecto a lo anterior, para crear un nuevo hilo de ejecución es necesario heredar de la clase `Thread` e implementar el método `run()`. Este método puede ser considerado como el método `main()` del nuevo hilo creado. Un nuevo hilo comienza su ejecución con el método `run()` en la misma forma que un programa comienza su ejecución con el método `main()` [21].

La sentencia `catch` permite, de forma genérica, manejar posibles condiciones de excepción lanzadas por el código dentro del bloque definido por la sentencia `try`. El método `printStackTrace()` muestra en pantalla la pila de ejecución, *i.e.* la información sobre la secuencia de ejecución hasta el momento donde ocurrió el problema.

Es probable que el método remoto necesite para su ejecución uno o varios argumentos, el desarrollador debe modificar el código ligeramente para añadir estos argumentos. Si la ejecución del método remoto necesita *e.g.* de un contador, puede ser agregado como un atributo privado de la clase `Cliente` como:

```
 private int n;
```

En la línea de comando se debe especificar un argumento adicional. Por tanto, en el código deben aparecer las siguientes líneas:

```
if(args.length != 7)
 System.out.println("Uso:Cliente<IP><puerto><politica><prioridad><periodo><plazo><numero>");
```

Ahora el constructor de la clase se define de la siguiente manera:

```
public Cliente(String IP,int puerto,String politica,int prioridad,int periodo, int plazo, int
numero)
```

Por último, creamos una instancia de `InterfazRemota`:

```
InterfazRemota hiloTiempoReal = new InterfazRemota(socket, nuevosParametros, n);
```

De esta forma, la invocación remota se realizará considerando el argumento adicional de tipo entero.

El código de la clase `InterfazRemota`, mostrado en la figura 5.4, administra la comunicación con el servidor. Se encarga de enviar los parámetros de tiempo real y de recibir el resultado (en caso de existir) de la invocación. Para tales efectos, importa las clases `Socket`, `ObjectOutputStream`, y `DataInputStream`, respectivamente.

```
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class InterfazRemota extends Thread
{
  private Parametros nuevosParametros;
  private DataInputStream datosRespuesta;
  private ObjectOutputStream salidaParametros;
  private Socket socket;

  public InterfazRemota(Socket socket, Parametros nuevosParametros)
  {
    this.socket = socket;
    this.nuevosParametros = nuevosParametros;
    try
    {
      datosRespuesta = new DataInputStream(socket.getInputStream());
      salidaParametros = new ObjectOutputStream(socket.getOutputStream());
    } catch (Exception e){e.printStackTrace();}
  }
  public void run()
  {
    System.out.println("Enviando parametros de planificacion ...");
    try
    {
      salidaParametros.writeObject(nuevosParametros);
      String respuesta;
      while(true)
      {
        respuesta = datosRespuesta.readUTF();
        System.out.println("Respuesta: " + respuesta);
      }
    } catch (Exception e){e.printStackTrace();}
  }
}
```

Figura 5.4: Código fuente de la clase `InterfazRemota`

En cuanto a sus métodos, el constructor de la clase crea una instancia tanto para el flujo de entrada como para el de salida de datos. Puesto que la clase `InterfazRemota` hereda de la clase `Thread`, debe implementar el método `run()`, cuyo código permite enviar por el flujo de salida el objeto de la clase `Parametros`. Este objeto es serializado con el método `writeObject()` y mediante el método `readUTF()`se recibe el resultado por el flujo de entrada. Aquí suponemos que la respuesta es un tipo de dato primitivo como un *int* o *string*; sin embargo, el desarrollador puede adecuar esta parte del código según sus necesidades.

5.2.2 Implementación de la *aplicación servidor*

Una vez analizado la *aplicación cliente*, nos ocupamos de la *aplicación servidor*, ubicada generalmente en otra computadora aunque bien podría estar sobre otra instancia de la *JVM-RT*. La *aplicación servidor* está compuesta de las siguientes clases: `Servidor`, `ParametrosTiempoReal` y `Aplicacion`.

```
import java.net.ServerSocket;
import java.net.Socket;

public class Servidor
{
  private int puerto;
  public Servidor(int puerto)
  {
    this.puerto = puerto;
    try
     {
       ServerSocket socketServer = new ServerSocket(puerto);
       System.out.println("Servidor esperando en el puerto " + puerto + "...");
       while(true)
       {
         Socket cliente = socketServer.accept();
         ParametrosTiempoReal nuevoCliente = new ParametrosTiempoReal(cliente);
         nuevoCliente.start();
       }
     } catch(Exception e) {e.printStackTrace();}
  }
  public static void main(String[] args)
  {
   if(args.length != 1) {
     System.out.println("Uso: Servidor <puerto>");
     System.exit(1);
   }
   int args0 = Integer.parseInt(args[0]);
   Servidor servidor = new Servidor(args0);
 }
}
```

Figura 5.5: Código fuente de la clase `Servidor`

En la figura 5.5 aparece el código de la clase `Servidor`. Utiliza el constructor para atender conexiones entrantes de clientes por medio de la clase `ServerSocket`. El *socket* servidor es creado en un puerto especificado por teclado en el momento de ejecutar una instancia de la clase `Servidor`. Una alternativa es especificar este atributo privado directamente desde el código fuente.

El método `main()` comienza la ejecución del servidor que escucha en el puerto especificado. Adicionalmente, se utiliza la clase `Socket` para establecer la conexión con el cliente. Por medio de la estructura `while`, el servidor atiende indefinidamente y acepta conexiones solicitadas por los clientes a través del método `accept()`.

La clase `Servidor` está destinada únicamente para atender las conexiones solicitadas; y una vez efectuada una conexión, se crea una instancia de la clase `ParametrosTiempoReal` (figura 5.6), quien será la encargada de procesar la conexión establecida. Puesto que dicha clase

hereda de la clase `Thread`, se crea un hilo de ejecución exclusivo por cada cliente, dando la posibilidad de que se puedan manejar varias conexiones simultáneamente.

```
import java.net.Socket;
import java.io.ObjectInputStream;
import javax.realtime.*;
public class ParametrosTiempoReal extends Thread
{
  private Socket cliente;
  private ObjectInputStream entradaParametros;
  private Parametros nuevosParametros;
  static int numHilo = 0;
  public ParametrosTiempoReal(Socket cliente)
  {
    super();
    this.cliente = cliente;
    try
    {
      entradaParametros = new ObjectInputStream(cliente.getInputStream());
    } catch(Exception e) {e.printStackTrace();}
  }
  public void run()
  {
    try
    {
      nuevosParametros = (Parametros) entradaParametros.readObject();
      numHilo++;
      System.out.println("--- Parametros recibidos (Hilo " + numHilo + ") ---");
      System.out.println("Politica: " + nuevosParametros.obtenerPolitica());
      System.out.println("Prioridad: " + nuevosParametros.obtenerPrioridad());
      System.out.println("Periodo: " + nuevosParametros.obtenerPeriodo());
      System.out.println("Plazo: " + nuevosParametros.obtenerPlazo());
     int pri = riorityScheduler.instance().getMinPriority()+nuevosParametros.obtenerPrioridad();
      PriorityParameters prip = new PriorityParameters(pri);
      RelativeTime per = new RelativeTime(nuevosParametros.obtenerPeriodo(),0);
      RelativeTime dead = new RelativeTime(nuevosParametros.obtenerPlazo(),0);
     PeriodicParameters perp = new PeriodicParameters(new RelativeTime(0,0),per,null,dead,null,null);
      Aplicacion nuevoClienteTiempoReal = new Aplicacion(cliente, numHilo, prip, perp);
      nuevoClienteTiempoReal.start();
      try
      {
        nuevoClienteTiempoReal.join();
      } catch(Exception e) {e.printStackTrace();}
    } catch(Exception e) {e.printStackTrace();}
  }
}
```

Figura 5.6: Código fuente de la clase `ParametrosTiempoReal`

La clase `ParametrosTiempoReal` **es el puente de unión entre el ambiente *Java* estándar y *Java* de tiempo real**. Por esta razón, se importan las clases del paquete `javax.realtime`. En el constructor de la clase, se utilizan las clases `Socket` y `ObjectInputStream` para acceder al flujo de entrada del *socket* cliente. Esta clase hereda de `Thread` y dentro del método `run()`, lee el objeto de la clase `Parametros`, transmitido desde el cliente. Dicho objeto es deserializado con el método `readObject()` a su tipo original (`Parametros`).

Los parámetros recibidos: política, prioridad, periodo y plazo, son obtenidos por medio de los métodos públicos de la clase `Parametros` y enseguida son transformados al formato de tiempo

real correspondiente, utilizando los constructores de las clases de la *RTSJ*: `Scheduler`, `PriorityParameters`, `RelativeTime`, y `PeriodicParameters`.

Una vez realizada la transformación de tipo de datos, se crea una instancia de la clase `Aplicacion` (figura 5.7), que hereda de la clase `RealtimeThread`, convirtiéndose así en un hilo de tiempo real que se ejecuta por separado cuando se invoca el método `start()`. Este método invoca el método `run()`, que define el código que implementa el desarrollador de acuerdo con el propósito de la aplicación. Finalmente, el método `join()` permite que termine la ejecución del hilo de tiempo real.

```
import java.net.Socket;
import java.io.DataOutputStream;
import javax.realtime.*;

class Aplicacion extends RealtimeThread
{
  private Socket socket;
  private DataOutputStream datosRespuesta;
  private int numHilo;
  public Aplicacion(Socket socket,int numHilo,SchedulingParameters prip,PeriodicParameters perp)
  {
    super(prip, perp);
    this.socket = socket;
    this.numHilo = numHilo;
    try
    {
      datosRespuesta = new DataOutputStream(socket.getOutputStream());
    } catch (Exception e){e.printStackTrace();}
  }
  public void run()
  {
    RealtimeThread me = currentRealtimeThread();
    try
    { /* instrucciones que ejecuta el
         hilo de tiempo real    */
      datosRespuesta.writeUTF("Respuesta: ");
    } catch (Exception e){e.printStackTrace();}
  }
}
```

Figura 5.7: Código fuente de la clase `Aplicacion`

En la clase `Aplicacion` ocurre todo el procesamiento de tiempo real de la aplicación. En el caso de que el código produzca un resultado, se utilizan las clases `Socket` y `DataOutputStream` para acceder al *socket* de comunicación con el cliente y enviar la respuesta por el flujo de salida.

La clase `Aplicacion` hereda de la clase `RealtimeThread`, por lo que se convierte en un hilo cuya ejecución tiene asociados los parámetros de tiempo real especificados por el cliente. Dichos parámetros proveen las restricciones de planificación y de tiempo bajo las cuales se ejecutará el código implementado en el método `run()`. De esta manera, la invocación de método remoto realizada por el *objeto cliente* es ejecutada en el servidor utilizando un hilo de tiempo real que satisface los parámetros especificados por el cliente.

5.3 Sistema de supervisión

Una vez puesta en práctica la propuesta de solución, surgió la necesidad de integrarla con un componente que realice la supervisión del comportamiento de una aplicación distribuida en tiempo de ejecución, *i.e.* una herramienta empleada especialmente en sistemas de líneas de producción y, en general, en todos los sistemas de tiempo real.

Aunque no fue un componente considerado desde el inicio del proyecto, su utilización se debe principalmente a dos razones:

- En el momento de efectuar pruebas de rendimiento, se debe esperar el término de la ejecución de la aplicación distribuida para posteriormente graficar y analizar los resultados, lo cual puede tomar una cantidad considerable de tiempo.

- La visualización en línea de los parámetros de tiempo real, tales como: el periodo y su variación, tiempos de latencia, plazos no cumplidos, en el mismo instante en que son producidos, permite tener una idea más precisa del comportamiento de la aplicación y ofrece la posibilidad de tomar decisiones preventivas y/o correctivas.

El desarrollo desde cero de una herramienta que satisfaga estas necesidades no es una tarea fácil, razón por la cual se ha aprovechado un software existente, *LiveGraph* [40], creado con propósitos de investigación por la Universidad de Monash (Australia); además, disponible de manera gratuita y de código abierto.

5.3.1 Características de *LiveGraph*

Las siguientes características hacen de *LiveGraph* particularmente útil para aplicaciones que requieren visualización en tiempo real de grandes cantidades de datos en forma de representaciones gráficas: [40]

- *LiveGraph* es un *framework* que ofrece su propia *API* para la visualización y registro de datos.

- Actualiza automáticamente las gráficas de datos mientras están siendo procesados por una aplicación.

- Posee una interfaz gráfica que permite seleccionar y comparar un gran número de series de datos simultáneamente. Esta característica es útil para la visualización de varios hilos de ejecución.

- *Framework* basado en *Java* que puede ser ejecutado sobre cualquier sistema de computación y capaz de ser integrado con cualquier tipo de aplicación.

- *LiveGraph* lee archivos en formato de estilo *CSV* (*Comma Separated Values*) para aplicaciones desarrolladas en *Java*.

5.3.2 Instalación y utilización

LiveGraph se encuentra disponible para su descarga en el sitio web [40], en un archivo con formato *ZIP*. Una vez descargado, se descomprime y el software está listo para ser utilizado. Como único requisito, es necesario disponer de la plataforma *Java* 6 o posterior, disponible para descarga en el sitio de *Sun Microsystems* [22].

Después de descomprimir, se generan diversos archivos, entre los cuales se encuentra uno denominado `LiveGraph.1.1.Complete.jar`. En sistemas *Windows* y en varios sistemas *Linux*, es posible comenzar la ejecución del programa haciendo un doble *clic*. En el caso de una interfaz de comandos, se digita lo siguiente en el directorio correspondiente:

```
$ java –jar LiveGraph.jar
```

Una vez que el programa está en ejecución, la interfaz gráfica de *LiveGraph* luce como la figura 5.8 y está constituida por varias ventanas.

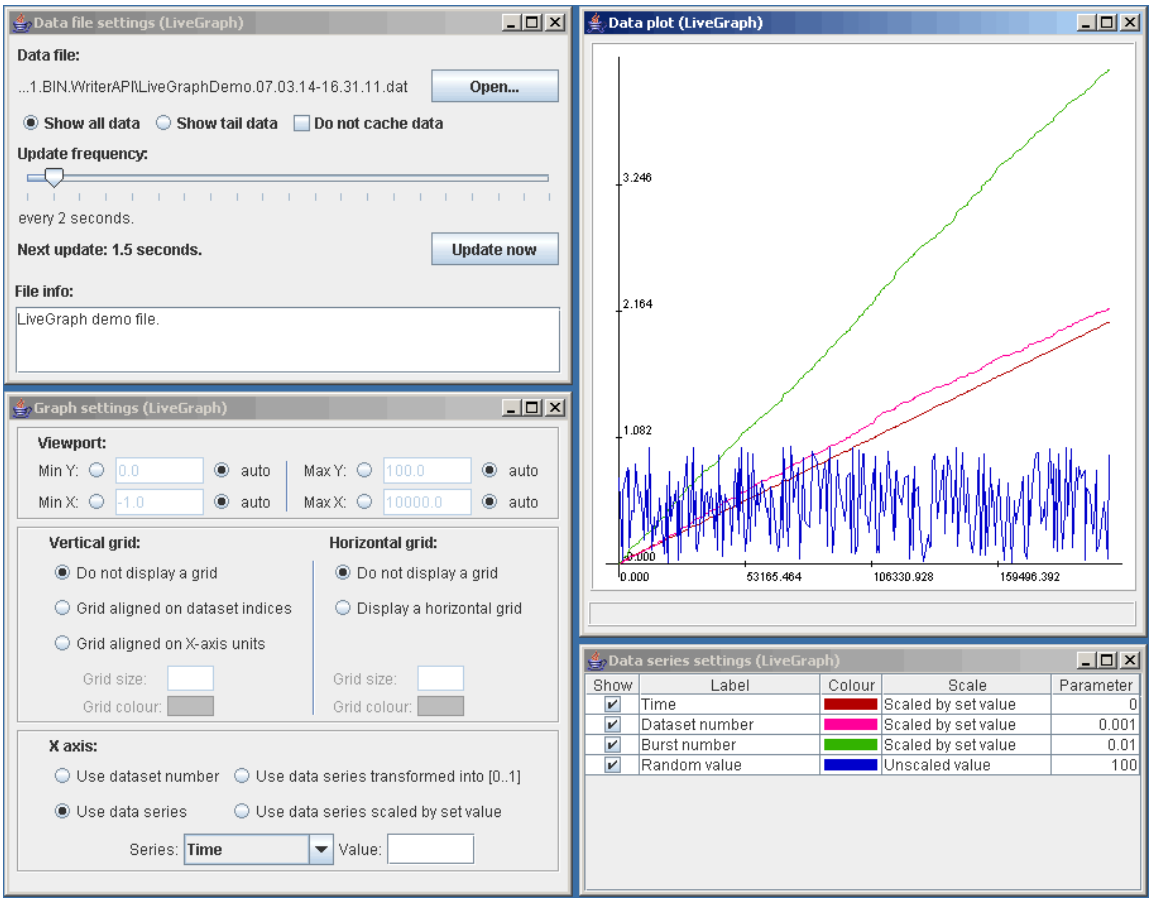


Figura 5.8: Interfaz gráfica de *LiveGraph*

La ventana *Data file settings* incluye opciones relacionadas con el archivo desde el cual *LiveGraph* leerá los datos. Aquí es posible seleccionar el archivo, mismo que debe estar

conforme al formato *CSV*; además, se puede especificar la frecuencia con la cual se realizará la lectura de datos para actualizar la gráfica en pantalla.

La ventana *Graph settings* permite definir opciones como valores mínimos y máximos de los ejes de la gráfica así como establecer si debe aparecer una rejilla sobre la gráfica.

La ventana *Data series settings* es utilizada para controlar la visualización de cada columna (serie de datos) del archivo de datos.

Dentro de la ventana *Data plot* se muestra la representación gráfica. Aquí se despliegan los datos contenidos en el archivo de datos seleccionado. Únicamente las series de datos escogidas en la ventana anterior serán visualizadas. A medida que se mueve el cursor del *mouse* sobre la gráfica, se desplegarán las coordenadas de datos en la parte inferior de la ventana.

5.3.3 Integración de *LiveGraph* con la solución

La figura 5.9 muestra la arquitectura del sistema con la inclusión del **nodo *Supervisor*** en una red con protocolo *TCP*/*IP*:

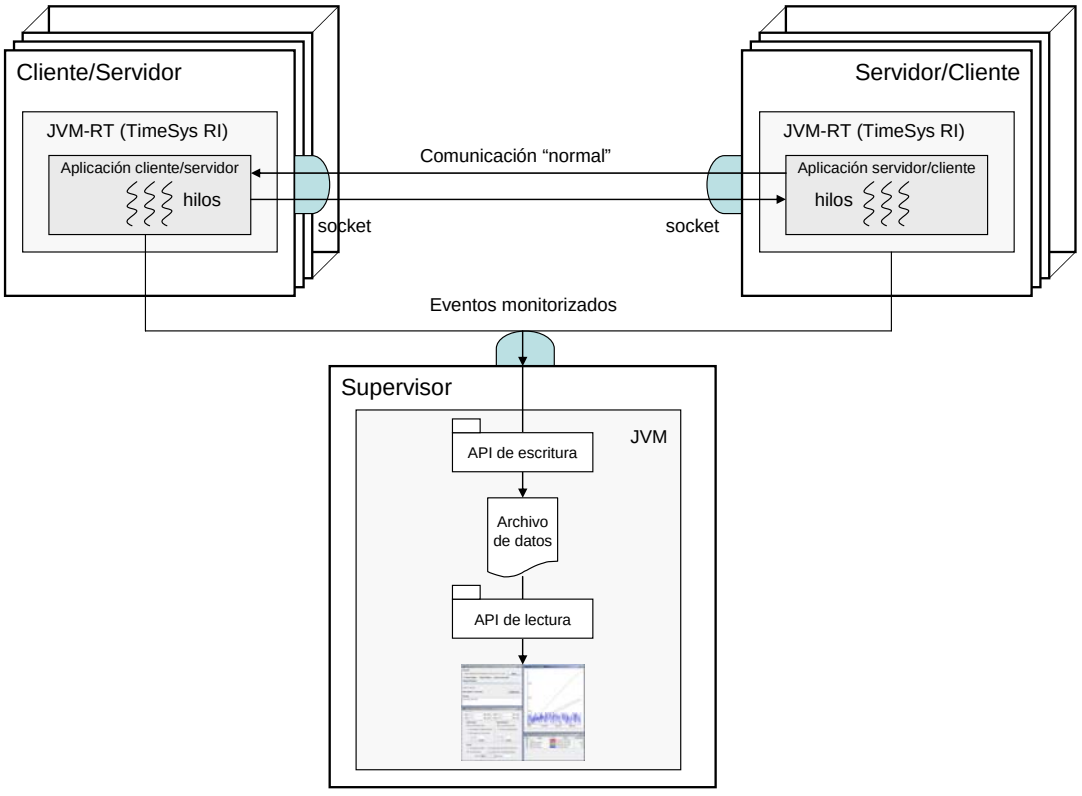


Figura 5.9: Arquitectura de solución con *Supervisor*

El sistema distribuido de la figura 5.9 permite la ejecución y supervisión de una aplicación distribuida con requisitos de tiempo real. Está conformado por tres nodos, dos de los cuales

pueden actuar indistintamente como cliente o servidor, mientras un tercer nodo se encargará de supervisar y desplegar en pantalla el curso de la ejecución.

La aplicación cliente y servidor de la aplicación distribuida se ejecutan sobre una *JVM-RT* (*Timesys RI*) en computadoras separadas. Estas se convierten en clientes de la computadora que aloja el sistema *Supervisor*, el cual ejecuta el software *LiveGraph* sobre una *JVM* convencional, no es necesario que sea de tiempo real. Es importante señalar que se pueden agregar más nodos al sistema distribuido ya que **el software de visualización fue adaptado para atender varios clientes de manera simultánea**.

Entre los nodos *cliente* y *servidor* se produce un flujo de parámetros por medio de invocaciones de métodos remotos. Estos parámetros de tiempo real controlan la ejecución de la aplicación distribuida. Mientras progresa la ejecución, se van generando eventos como la creación de hilos de ejecución, activaciones periódicas, cumplimiento o pérdida de plazos, *etc*.

En el momento en que se produce cada evento, éste es enviado al sistema *Supervisor*, quien ejecuta una aplicación en modo servidor para recibir los datos por el *socket* de comunicación. Se utilizan funciones de la *API* de escritura así como de lectura para registrar la información en el archivo de datos y presentarla gráficamente.

LiveGraph mantiene abierto un flujo de entrada al archivo para ser inspeccionado en intervalos de tiempo configurables por el usuario. Automáticamente se re-examina el archivo de datos en el intervalo de tiempo especificado y se actualiza la gráfica sobre la pantalla.

Un aspecto importante es la definición y el formato del archivo de datos, similar al de un archivo de texto basado en valores separados por comas. El archivo se compone de encabezados y columnas de datos que son detectados automáticamente por el software de visualización.

5.3.4 Adecuaciones realizadas a *LiveGraph*

Puesto que no es un software hecho a la medida, ha sido necesario efectuar ciertas adecuaciones con el fin de que pueda interactuar con la solución desarrollada. De esta forma, permite:

- Presentar al usuario los datos de tiempo real de manera intuitiva por medio de la interfaz gráfica.
- Observar el comportamiento de la aplicación mientras la ejecución está en progreso.
- Observar los eventos relacionados con los parámetros de tiempo real y almacenar información acerca de ellos.
- Emitir mensajes preventivos.
- Producir mínima carga al sistema.

Para conseguir la visualización inmediata de los parámetros de tiempo real, la aplicación distribuida debe interactuar con *LiveGraph*. Aprovechamos que la herramienta está basada en lenguaje *Java* y que ofrece una *API* (biblioteca de clases) que puede ser utilizada por las aplicaciones.

La supervisión se realiza por medio de un código que es específico de la aplicación de estudio pero puede servir de modelo para otras instancias de una aplicación distribuida de tiempo real en *Java*, lo cual se convierte en una contribución adicional del trabajo de tesis. En esta sección se incluyen las clases utilizadas para supervisar la ejecución de una aplicación distribuida de tiempo real. Vamos a tomar como caso de estudio la graficación de los datos correspondientes al periodo de cada uno de los hilos de ejecución que conforman la aplicación distribuida.

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.DataInputStream;
import org.LiveGraph.dataFile.write.DataStreamWriter;
import org.LiveGraph.dataFile.write.DataStreamWriterFactory;
public class GraficarPeriodo
{
  public static final String DEMO_DIR = System.getProperty("user.dir");
  public void exec()
  {
    DataStreamWriter out = DataStreamWriterFactory.createDataWriter(DEMO_DIR, "DatosPeriodo");
    out.setSeparator(";");
    out.writeFileInfo("Archivo de datos.");
    String nombreColumna;
    int MAX_HILOS = 50;
    for(int col=1;col<MAX_HILOS+1;col++)
    {
      nombreColumna = "Period" + col;
      out.addDataSeries(nombreColumna);
    }
    try
    {
      int i = 0;
      ServerSocket socketServidor = new ServerSocket(2222);
      System.out.println("Supervisor escuchando en puerto 2222 ...");
      while(true)
      {
        Socket cliente = socketServidor.accept();
        ++i;
        GraficarThread newThread = new GraficarThread(cliente, out, i);
        newThread.start();
      }
     } catch(Exception e){e.printStackTrace();}
     if (out.hadIOException())
     {
       out.getIOException().printStackTrace();
       out.resetIOException();
     }
     out.close();
     System.out.println("Supervisor finalizado.");
  }
  public static void main(String[] args)
  {
    (new GraficarPeriodo()).exec();
  }
}
```

Figua 5.10: Código fuente de la clase `GraficarPeriodo`

La figura 5.10 muestra el código de la clase `GraficarPeriodo`, basado en uno de los ejemplos que se incluyen con el software de *LiveGraph* y al cual hemos hecho diversas adecuaciones para que se comunique con la solución. El código puede ser modificado y extendido de acuerdo con los requerimientos particulares del desarrollador.

Esta clase implementa un servidor en espera de conexiones de clientes. En este caso, la aplicación servidor se comporta como cliente y envía los datos de periodo al *Supervisor*. Los datos son almacenados en un archivo de texto que se ubicará en la ruta especificada por un atributo de tipo *string* denominado *DEMO_DIR*.

La clase contiene dos métodos: el método `main()` donde se crea un objeto de la clase para comenzar la aplicación y un método `exec()` dentro del cual se crea un objeto para registrar los datos recibidos, *i.e.* es el archivo de datos denominado `DatosPeriodo`. Esta operación se realiza por medio de la clase `DataStreamWriter`, la cual es parte de la *API* de *LiveGraph* (paquete *org.LiveGraph.dataFile.write*).

Luego, se define el símbolo que actuará como separador de los valores con el método `setSeparator()` y además se agrega un título inicial dentro del archivo con el método `writeFileInfo()`.

Una de las más importantes modificaciones realizadas al código original del ejemplo es la inserción de una estructura `for()` con el propósito de crear varias columnas mediante el método `addDataSeries()` que corresponderán a cada una de las series de datos; en este caso, una columna para cada hilo de ejecución.

Hemos definido una constante denominada `MAX_HILOS` para establecer el número límite de hilos que pueden ser graficados simultáneamente; sin embargo, este valor puede ser modificado a conveniencia de una aplicación en particular. Lógicamente, mientras más alto es el valor, mayor será el tamaño del archivo de datos.

Enseguida se crea el *socket* de servidor utilizando el constructor de la clase *ServerSocket* especificando como argumento el número de puerto. La estructura de control `while()` atiende indefinidamente conexiones de clientes y una vez aceptada a través del método `accept()`, se incrementa una variable contador que servirá para enumerar cada uno de los clientes.

Esta clase sirve únicamente para atender las conexiones de clientes. Por otro lado, quien se encarga de registrar los datos pertenecientes a un hilo determinado es una instancia de la clase `GraficarThread` (figura 5.11). Puesto que esta clase hereda de `Thread`, se crea un hilo de ejecución exclusivo por cada cliente, dando la posibilidad de que se puedan manejar varias conexiones a la vez.

Así conseguimos un **supervisor de tipo *multi-hilo*** que supervisa todos los hilos de la aplicación distribuida. Por último, añadimos una rutina para el manejo de posibles excepciones de entrada/salida.

```
import java.net.Socket;
import java.io.DataInputStream;
import org.LiveGraph.dataFile.write.DataStreamWriter;
class GraficarThread extends Thread
{
  private Socket client;
  private DataInputStream dataInput;
  private DataStreamWriter out;
  private int i;
  public GraficarThread(Socket socket, DataStreamWriter dataStreamWriter, int i)
  {
    super();
    client = socket;
    out = dataStreamWriter;
    this.i = i;
  }
  public void run()
  {
    float period;
    try
    {
      dataInput = new DataInputStream(client.getInputStream());
      while(true)
      {
        period = dataInput.readFloat();
        System.out.println("Period of Thread " + i + ": " + period);
        int MAX_HILOS = 50;
        for(int col=1; col<MAX_HILOS+1; col++)
        {
          if(col != i)
            out.setDataValue("null");
          if(col == i)
            out.setDataValue(period);
        }
        out.writeDataSet();
      }
    } catch(Exception e) {e.printStackTrace();}
  }
}
```

Figura 5.11: Código fuente de la clase `GraficarThread`

La clase `GraficarThread` permite manejar varios hilos creando un flujo de entrada de datos para cada hilo de la aplicación. El constructor de la clase asigna a los aributos privados, los valores del *socket* cliente, el archivo donde se registrarán los datos, y el identificador del hilo de ejecución. En el método `run()`, se utiliza la clase `Socket` y `DataInputStream` para acceder al flujo de entrada del *socket* cliente y leer los datos mediante el método `readFloat()` dentro de un ciclo `while`. En este caso, el valor de periodo es tratado como un tipo de dato *float*. Luego de recibir el valor de periodo resta escribir tal valor en el archivo de datos. De acuerdo con la identificación del hilo, el valor será escrito en la columna respectiva mientras que en las demás columnas se escribirá la cadena *null*. De esta manera se va registrando información en el archivo de datos, mismo que será examinado regularmente por *LiveGraph* para generar la correspondiente gráfica en pantalla. Las implementaciones mostradas en las figuras 5.10 y 5.11, sirven para la graficación de los datos de periodo de cada uno de los hilos de ejecución que conforman la aplicación distribuida. De la misma forma, creamos las clases `GraficarPlazos`, `GraficarLatencia`, y `GraficarVariacion`, para la graficación de los datos de plazos de ejecución perdidos, latencia de red, y variación del periodo, respectivamente.

La figura 5.12 muestra el diagrama final de clases considerando todos los elementos descritos, *i.e.* el paquete del software de visualización, las clases para la graficación de los parámetros de tiempo real, las clases para la carga dinámica y, la clase del planificador adicional.

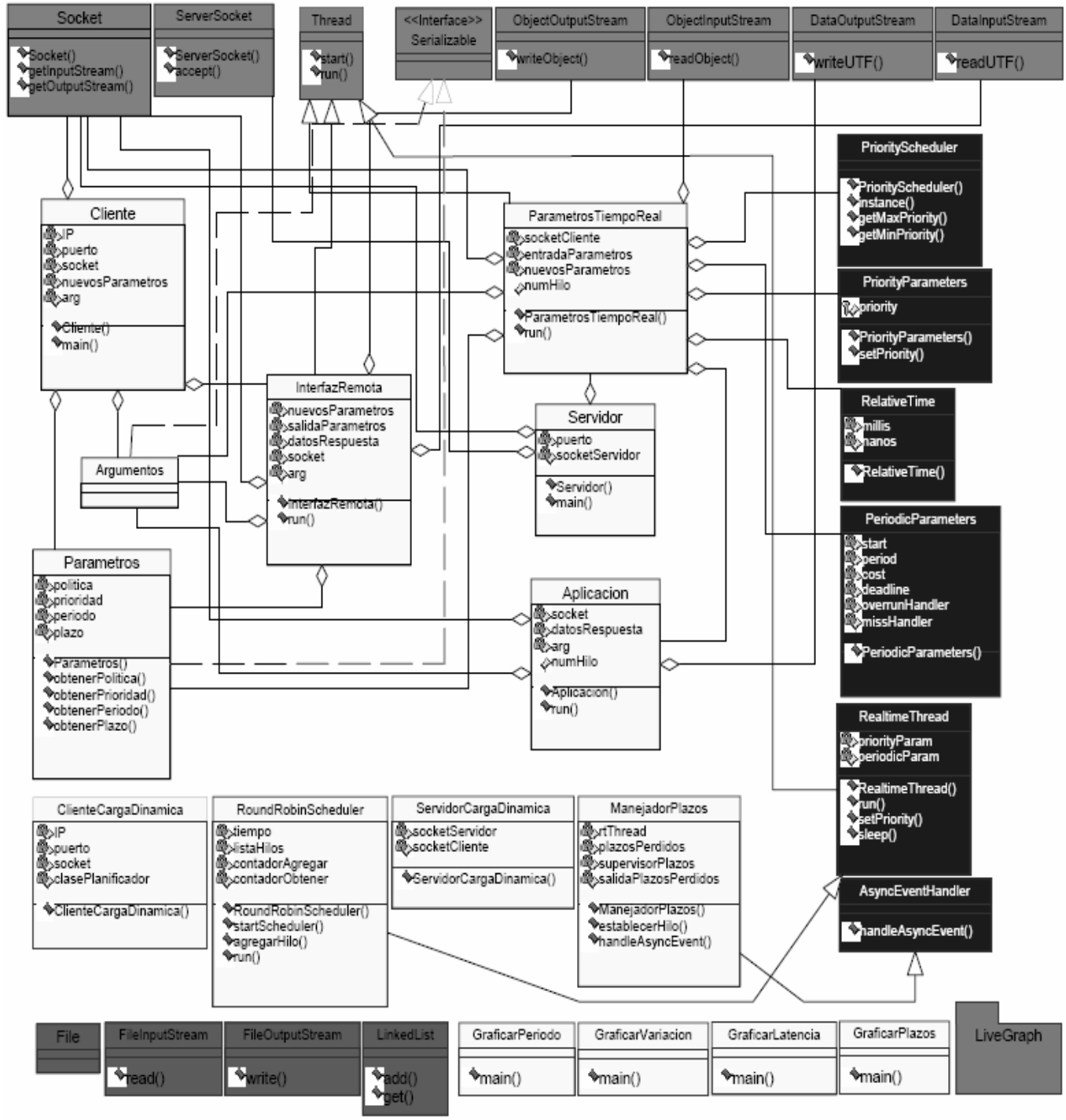


Figura 5.12: Diagrama final de clases

Capítulo 6

Pruebas y análisis de los resultados

En este capítulo presentamos la verificación del funcionamiento y la evaluación de la solución desarrollada. Primeramente, describimos la plataforma donde se llevaron a cabo las pruebas. Posteriormente, abordamos el problema de la sincronización en ambientes distribuidos, un aspecto esencial por resolver antes de efectuar las pruebas. A continuación detallamos las aplicaciones distribuidas que han permitido cuantificar diversos parámetros que son característicos de un entorno de tiempo real. Simultáneamente, los resultados obtenidos son interpretados. Por último, destacamos dos ventajas muy importantes de la solución propuesta: **adaptabilidad** y **versatilidad**. Para tal efecto, nuestra solución permite la implementación de un planificador *round robin* basado en prioridades. Describimos la forma en la cual puede ser incluido en una aplicación distribuida de tiempo real para controlar su ejecución y, por otra parte, exponemos la implementación de tres ejemplos de aplicaciones distribuidas con base en el esquema de clases propuesto.

6.1 Ambiente de ejecución de pruebas

Definimos la infraestructura necesaria de hardware y de software para poner en práctica la solución desarrollada, *i.e.* el sistema distribuido sobre el cual se realizaron las pruebas. La figura 6.1 muestra un esquema de los nodos que conforman el sistema distribuido y la red de comunicación. Cada uno de los nodos aparece con una etiqueta que determina su función ya sea *cliente*, *servidor*, o *supervisor*.

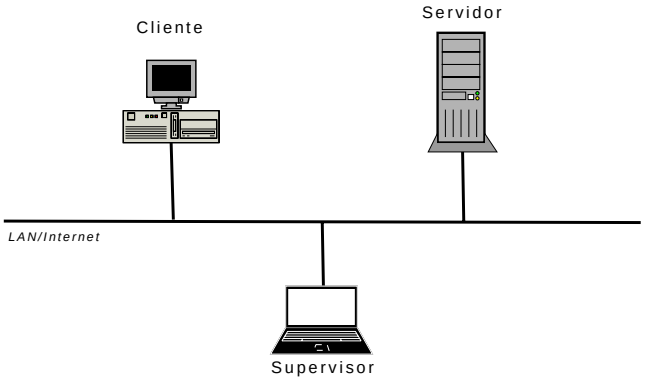


Figura 6.1: Plataforma de pruebas

Con respecto al hardware, la tabla 6.1 resume las principales características de las computadoras utilizadas:

Característica	*Cliente*	*Servidor*	*Supervisor*
Procesador	Intel Pentium 4	Intel Pentium D	Intel Core 2
Velocidad	3 GHz	2.80 GHz	1.66 GHz
Memoria RAM	1 GB	1 GB	1 GB
Sistema operativo	Linux Ubuntu (kernel 2.6.15-23-386)	Linux Ubuntu (kernel 2.6.15-23-386)	Windows Vista

Tabla 6.1: Características técnicas de las computadoras utilizadas

Las computadoras o nodos del sistema se comunican por medio del protocolo *TCP/IP* sobre una red de área local con tecnología *ethernet*.

Por otra parte, la figura 6.2 muestra el software utilizado y está representado como un modelo de capas omitiendo la capa de hardware (ya explicada en los párrafos anteriores). Se aprecian los niveles correspondientes al sistema operativo, a la máquina virtual de *Java* y al nivel de aplicación.

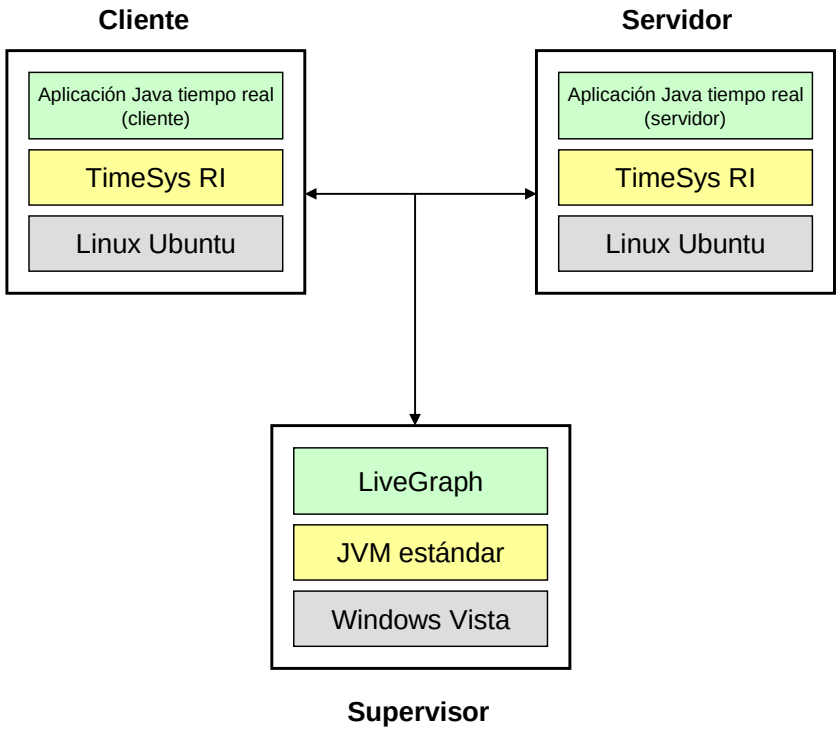


Figura 6.2: Software del sistema distribuido

6.2 Sincronización

Un problema muy importante en sistemas distribuidos y más aún en los de tiempo real, consiste en **mantener sincronizados los relojes físicos** de cada una de las computadoras que intervienen en el sistema. Cuando se considera un solo procesador, el tiempo no es ambiguo; sin embargo, en un sistema distribuido, cada computadora tiene su propio reloj físico, el cual es un dispositivo electrónico que cuenta el tiempo a diferentes frecuencias, debido principalmente a condiciones físicas y químicas, y por lo tanto divergen [2].

Los relojes de computadora pueden sincronizarse con fuentes externas de tiempo de gran precisión. Por ejemplo, el *Tiempo Universal Coordinado* (*UTC*, *Universal Time Coordinated*) es un estándar internacional de cronometraje utilizado por una tecnología denominada ***Protocolo de Tiempo de Red*** (*NTP, Network Time Protocol*). *NTP* está organizado bajo un modelo *cliente-servidor* jerárquico y provee un servicio de tiempo así como un protocolo estándar para distribuir la información del tiempo sobre *Internet* [10].

La arquitectura básica de *NTP* consiste de tres niveles:

1. En el nivel más alto de la jerarquía se encuentra un pequeño número de computadoras conocidas como los *relojes de referencia* y que reciben el tiempo generalmente desde satélites.

2. Un nivel más abajo se encuentran los servidores disponibles para *Internet*, mismos que permiten a sus clientes sincronizar sus relojes de computadora.

3. Los clientes que usan la referencia estándar de tiempo *UTC*.

Entre las características más importantes de este servicio se pueden mencionar:

- El servicio *NTP* es proporcionado por una red de servidores distribuidos en *Internet*.

- Es fiable puesto que existen servidores y recorridos redundantes.

- Los servidores pueden re-configurarse para continuar proporcionando el servicio si uno de ellos llega a ser inalcanzable o se producen fallos.

- El servicio permite que los clientes sean capaces de re-sincronizar con la suficiente frecuencia para compensar las tasas de deriva de las computadoras.

La figura 6.3 representa el sistema distribuido descrito en la parte inicial de este capítulo con la inclusión del servicio *NTP* para la sincronización de los relojes físicos de las computadoras.

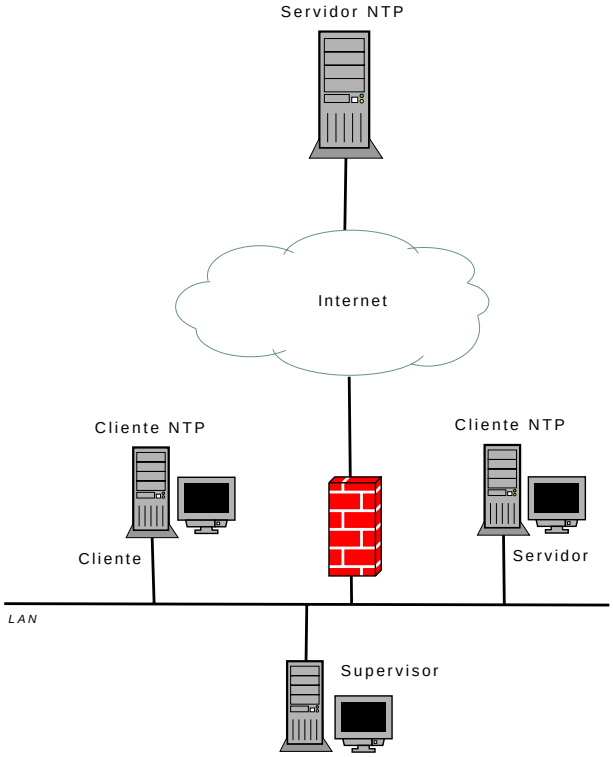


Figura 6.3: Plataforma de pruebas con servicio *NTP*

NTP se ha convertido en un protocolo estándar dentro de *Internet*, por lo que viene incluido en la mayoría de distribuciones del sistema operativo *Linux*. En el caso de *Ubuntu*, es posible instalar y configurar el cliente *NTP* para que se conecte periódicamente a servidores *NTP* (relojes de referencia) disponibles en *Internet*.

6.3 Pruebas

Para validar la solución propuesta, además de verificar que las invocaciones de método remotas son efectuadas entre las aplicaciones cliente y servidor, es necesario medir ciertos parámetros que son característicos las aplicaciones de tiempo real. Entre los parámetros de tiempo más comunes que se presentan en el dominio de tiempo real son: el *periodo*, el *plazo* de ejecución (*deadline*) y, debido a que en este caso se trata de un ambiente distribuido, es particularmente útil cuantificar la *latencia de red*. Estos parámetros se encuentran asociados a cada una de las tareas (hilos de ejecución) que componen una aplicación distribuida de tiempo real.

6.3.1 Cálculo de la latencia de red

Un aspecto fundamental en un sistema distribuido de tiempo real es el rendimiento de la red de

comunicaciones. Conocer a *priori* dicho rendimiento y específicamente el valor de su latencia, es clave para que un desarrollador pueda ajustar los requerimientos de tiempo de una aplicación distribuida a un ambiente de ejecución particular.

6.3.1.1 Consideraciones preliminares

Para medir la latencia se deben considerar factores como la velocidad y el medio de transmisión, condiciones de tráfico, *etc*. En nuestro caso, la red de área local utilizada define una velocidad de transmisión de 100 *Mbps* (*Megabits por segundo*); sin embargo, debido al nivel de tráfico y la limitación impuesta por las tarjetas de red involucradas (10 *Mbps*), en el momento de realizar las pruebas llegamos a establecer una velocidad de transmisión de datos de 6 *Mbps*. Para obtener esta velocidad realizamos la transferencia de varios archivos entre un cliente y un servidor ubicados en la misma red, por medio del programa *SSH* (*Secure Shell*) *File Transfer*. El programa proporciona el dato de velocidad de descarga para cada una de las transferencias, así que el promedio de estos valores, nos provee el valor de velocidad media de transmisión.

Examinamos la *latencia de red* como el tiempo transcurrido desde que se envían los parámetros de tiempo real por parte del cliente hasta que son recibidos por el servidor. En otras palabras, equivale al tiempo que tarda un mensaje en salir de un nodo de la red (*cliente*) y llegar a otro (*servidor*). Analizamos dos escenarios:

1. Tanto la *aplicación cliente* como la *aplicación servidor* se ejecutan sobre la misma computadora pero en instancias separadas de una *JVM-RT* (*TimeSys RI*).

2. La *aplicación cliente* como la *aplicación servidor* se ejecutan en diferentes computadoras conectadas por una red.

Comparar ambas situaciones permitirá establecer la influencia que representa la presencia de la red de comunicaciones durante la ejecución de la aplicación.

6.3.1.2 Ejecución de pruebas de latencia

Una vez **sincronizados los relojes de las computadoras cliente y servidor**, se procede a ejecutar una aplicación distribuida que considere la latencia de red entre los nodos cliente y servidor. Primeramente debemos iniciar el software de *LiveGraph* en el nodo *supervisor*, para monitorizar la ejecución. Enseguida ejecutamos *la aplicación servidor* y después la *aplicación cliente*. Cada una de estas aplicaciones está compuesta por varias clases, tal como se describe en la sección 5.2. De tal forma, en ambos casos ejecutamos la clase que contiene el método `main()`, *i.e.* la clase `Servidor` y la clase `Cliente`, respectivamente.

Una instancia de la clase `Cliente` solicita al servidor, mediante la línea de comandos, la ejecución de un hilo de tiempo real con sus propios parámetros de periodo y plazo. En total, realizamos 100 solicitudes de ejecución de este tipo. Las solicitudes son realizadas sucesivamente con el fin de que el servidor las atienda de manera simultánea, *i.e.*, no esperamos a que termine la ejecución de un hilo para empezar la ejecución de otro. La tabla 6.2 muestra los parámetros de

cada uno de los hilos de tiempo real.

	Hilo de tiempo real												
	1	**2**	**……**	**10**	**11**	**12**	**…….**	**20**	**……..**	**91**	**92**	**…..**	**100**
Periodo	100	200	…….	1000	100	200	……..	1000	……..	100	200	……	1000
Plazo	50	100	……	500	50	100	……..	500	……..	50	100	……	500

Tabla 6.2: Valores de parámetros para los hilos de tiempo real

Los valores de periodo y plazo están dados en milisegundos y se repiten cada 10 hilos. Para cada uno de los 100 hilos, el valor de plazo es la mitad de su correspondiente periodo. La prioridad es la misma para todos los hilos y tiene un valor de 11, *i.e.* el valor mínimo posible, con el fin de que no exista preferencia en la ejecución de un determinado hilo. El número de activaciones es 100 para todos los casos.

Cada hilo ejecuta el código definido en el método `run()` de la clase `Aplicacion` y consiste de un ciclo que consume tiempo de procesamiento, donde no se producen pérdidas de plazo. Para medir la latencia de red, hemos añadido en la clase `Parametros`, un atributo de tipo entero largo (*long*) para los *milisegundos* y un atributo de tipo entero (*int*) para los *nanosegundos*. Así, cuando los parámetros de tiempo real especificados por el cliente son enviados al servidor, dichos atributos son incluidos en el mensaje.

En el momento en el que se envían los parámetros desde la clase `InterfazRemota`, mediante una instancia del reloj de tiempo real (clase `Clock`) y un objeto de la clase `AbsoluteTime`, podemos obtener en formato de tiempo real, el instante en el que se produce el envío. Debido a que este dato no puede ser serializado, pues tiene un formato de tiempo real, es dividido en datos primitivos: un entero largo para el valor de milisegundos y un valor de entero para nanosegundos. De esta forma, pueden ser adjuntados al objeto de tipo `Parametros` y ser enviados al servidor.

En la *aplicación servidor*, la clase `Servidor` atiende al cliente creando un objeto de la clase `ParametrosTiempoReal`. Esta clase hereda de `Thread` así que se convierte en un hilo de ejecución por separado, el cual tiene la función de enviar los datos de latencia al *supervisor* por medio de un flujo de datos y un *socket* de comunicación.

La clase `ParametrosTiempoReal` recibe los parámetros de tiempo real enviados por el cliente accediendo al flujo de datos de entrada. En realidad los parámetros que recibe no son de un tipo de dato de tiempo real sino que son de tipos básicos (*long* e *int*). Esta clase se encarga de la conversión a tiempo real utilizando instancias de las clases de la *RTSJ*: `PriorityScheduler`, `PriorityParameters`, `RelativeTime` y `PeriodicParameters`. Adicionalmente, el dato del instante de envío, dividido en mili y nanosegundos, es reconstruido como un objeto de tiempo real por medio de la clase `AbsoluteTime`.

En el momento en que los parámetros enviados por el cliente son leídos desde el flujo de

entrada, tomamos la medida de este instante de tiempo. Puesto que ya tenemos los valores de tiempo correspondientes al envío y a la recepción de datos en formato de tiempo real, realizamos el cálculo de la diferencia de tiempo entre ambos a través del método `subtract()` y así obtenemos el valor de latencia de red. Este dato debe ser dividido en milisegundos (*long*) y nanosegundos (*int*) para poder ser enviado al nodo *supervisor*. Para cada uno de los hilos se efectúa el proceso descrito y, de esta manera, el *supervisor* puede mostrar en pantalla los datos de latencia de red para cada uno de ellos.

Cabe mencionar que, a medida que se van activando los hilos se añade más carga tanto para el servidor como para el tráfico de red. Es importante recalcar que para una correcta interpretación de los resultados, los relojes de cada una de las computadoras deben estar sincronizados.

6.3.1.3 Análisis de resultados de latencia

La figura 6.4 representa los resultados obtenidos para el primer escenario. En este caso, las pruebas descritas en la sección 6.3.1.2 son realizadas en la misma computadora. La *aplicación cliente* y la *aplicación servidor* se ejecutan en diferentes instancias de la *JVM-RT* pero en la misma computadora, razón por la cual, analizamos una latencia local.

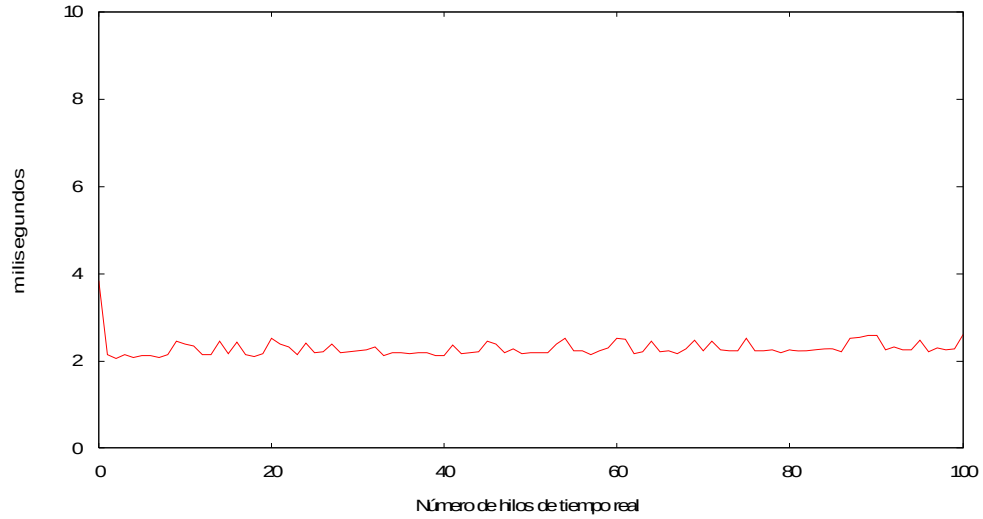


Figura 6.4: Latencia en la misma computadora

El primer hilo produce un valor de latencia cercano a los 4 milisegundos; posteriormente los valores van oscilando de tal forma que los valores máximos van creciendo pero no sobrepasan los 3 milisegundos. La tendencia creciente se debe a la presencia de mayor cantidad de hilos de tiempo real en ejecución simultánea.

El hecho de que el primer valor de la latencia sea mayor que el resto indica que el proceso de apertura y aceptación del *socket* de conexión entre cliente y servidor consume un tiempo adicional. Para las demás solicitudes del cliente, el *socket* permanece abierto.

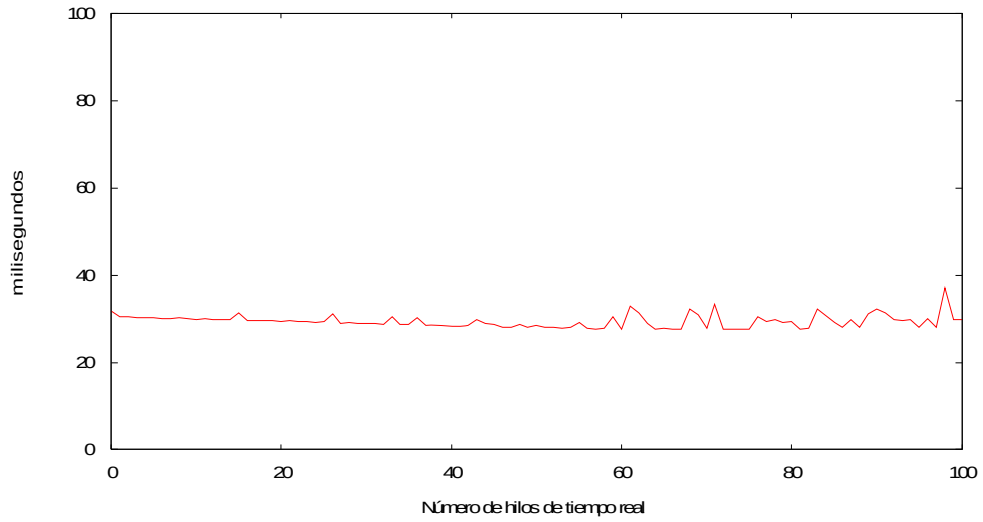


Figura 6.5: Latencia de red

En la figura 6.5, se muestran los resultados para el segundo escenario. En este caso, las pruebas descritas en la sección 6.3.1.2 son realizadas utilizando dos computadoras separadas físicamente. En una de ellas se ejecuta la *aplicación cliente* mientras que en la otra se ejecuta la *aplicación servidor*. Por tanto, analizamos la *latencia de red* entre cliente y servidor. El valor de la latencia, en promedio, es 10 veces mayor que en el caso anterior, debido lógicamente a la influencia del medio físico de comunicación. Se puede observar que, conforme aumenta el número de hilos, *i.e.* mayor tráfico, la latencia de red es mayor. Aproximadamente a partir del hilo número 60, el comportamiento en el valor de la latencia de red fluctúa considerablemente.

6.3.2 Cumplimiento de periodos

Es habitual que en aplicaciones de tiempo real se presenten tareas periódicas, especialmente en: sistemas de control y supervisión de procesos industriales, procesamiento por sensores, radar de seguimiento de vuelos, sistemas donde se producen datos en tasas fijas, monitores de temperatura en un reactor nuclear, *etc*.

En una tarea periódica, la solicitud de activación ocurre repetidamente a intervalos de tiempo regulares. La longitud de dicho intervalo de tiempo es denominado *periodo* de la tarea. Por ende, es recomendable comprobar el grado de precisión en el cumplimiento de los periodos asociados a cada una de las tareas de una aplicación.

6.3.2.1 Ejecución de pruebas

Para el cálculo del periodo no es necesario añadir código adicional en las clases de la *aplicación cliente* pues la activación de los hilos de ejecución ocurre en el servidor. La prueba consiste en la ejecución de 5 hilos de tiempo real con diferentes valores de periodo.

La tabla 6.3 contiene los parámetros de tiempo real para cada uno de los hilos, donde podemos apreciar que los valores de plazo así como el número de activaciones son constantes para todos los casos, 300 y 100, respectivamente.

Hilo de tiempo real	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de activaciones**
1	600	300	100
2	700	300	100
3	800	300	100
4	900	300	100
5	1000	300	100

Tabla 6.3: Parámetros de prueba

En cada activación de un hilo, se ejecuta el código que debe ser implementado en el método `run()` de la clase `Aplicacion` (ver sección 4.4). En esta ocasión, el código no es solamente un ciclo que consume tiempo de procesamiento sino que efectúa el cálculo del periodo como veremos a continuación.

La medición del periodo es realizada por medio de las clases `Clock`, `AbsoluteTime`, y `RelativeTime`. Una instancia del reloj de tiempo real (clase `Clock`) nos sirve para medir el instante en el que comienza la ejecución del método (objeto de la clase `AbsoluteTime`) y la diferencia entre cada par de activaciones sucesivas nos proporciona el valor del periodo como un objeto de tipo `RelativeTime`.

El valor del periodo debe ser dividido en sus componentes de milisegundos (*long*) y nanosegundos (*int*) para poder ser enviado al nodo *supervisor*. Para tal propósito, la clase `Aplicacion` maneja un *socket* de comunicación con este nodo y un flujo de datos de salida por donde los datos de periodo son enviados para su posterior visualización.

6.3.2.2 Análisis de resultados del periodo

Las gráficas de las figuras 6.6-6.10, presentan el comportamiento del periodo en cada una de las activaciones de los hilos de tiempo real con periodos establecidos de 600, 700, 800, 900, y 1000 milisegundos, respectivamente. De esta forma podemos comparar el comportamiento entre los diferentes hilos de ejecución.

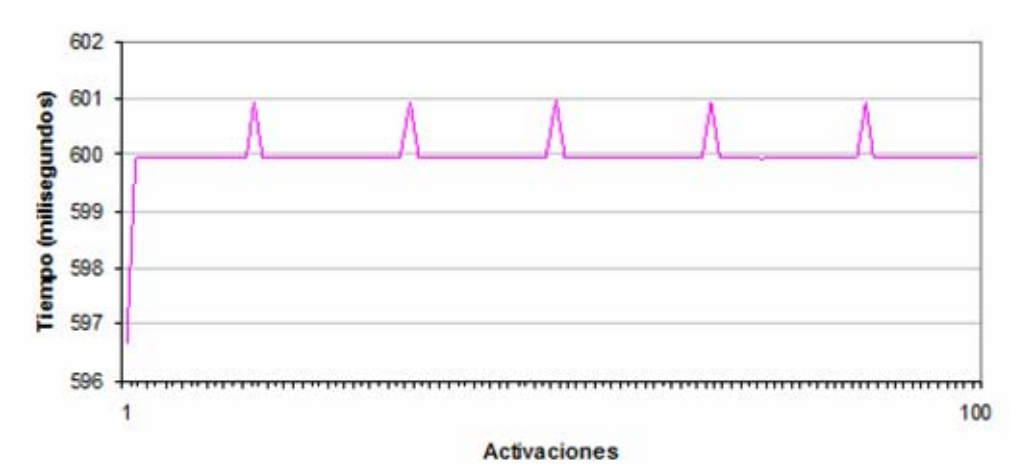


Figura 6.6: Ejecución del hilo de tiempo real con periodo de 600 milisegundos

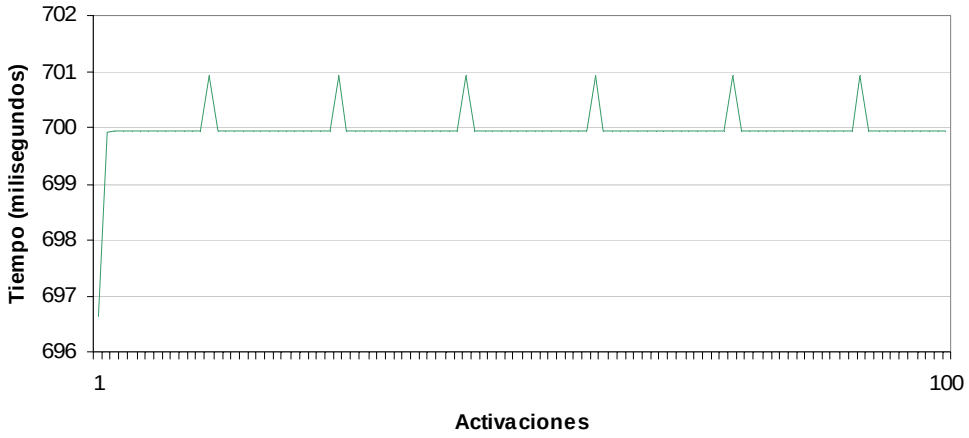


Figura 6.7: Ejecución del hilo de tiempo real con periodo de 700 milisegundos

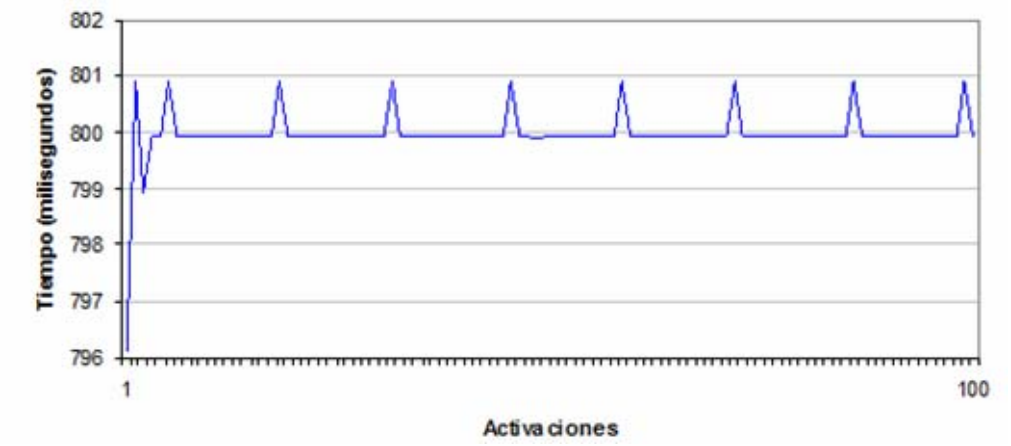


Figura 6.8: Ejecución del hilo de tiempo real con periodo de 800 milisegundos

Figura 6.9: Ejecución del hilo de tiempo real con periodo de 900 milisegundos

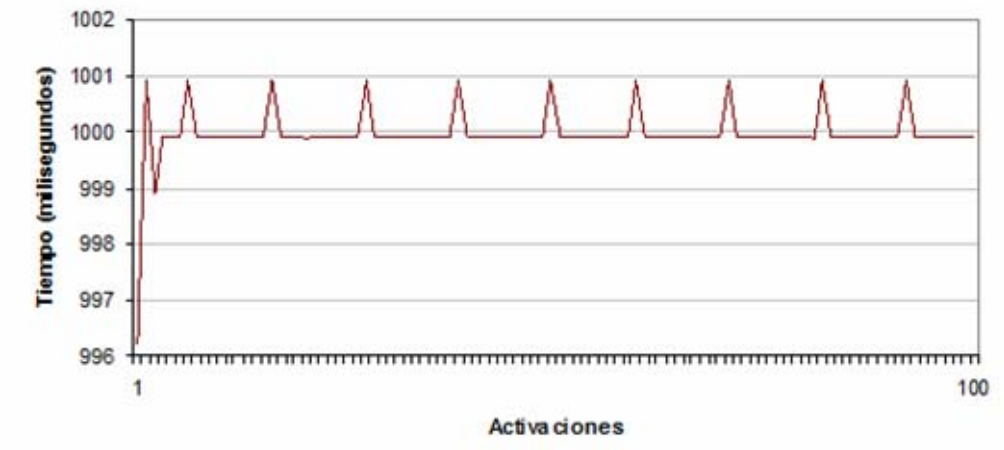


Figura 6.10: Ejecución del hilo de tiempo real con periodo de 1000 milisegundos

Cada uno de los 5 hilos de tiempo real periódicos es activado en 100 ocasiones. El comportamiento de los periodos es similar en los siguientes aspectos:

- En cada caso se presentan variaciones en intervalos de tiempo prácticamente regulares.
- Dichas variaciones son escasas con respecto al número total de activaciones y no exceden el valor de 1 milisegundo, con excepción de la primera activación, donde la diferencia es aproximadamente 4 milisegundos.
- La cantidad de variaciones aumenta mientras mayor es el periodo.
- La mayor parte de las activaciones se produce con un valor de periodo menor y muy cercano al establecido.

Por tanto, podemos decir que el cumplimiento del periodo por parte de la *JVM-RT* (*TimeSys*

RI) es bastante aceptable y se ve favorecida por el hecho de que la ejecución se produce en una sola computadora, sin intervención de la red de comunicaciones.

6.3.3 Plazos de ejecución perdidos

Un parámetro característico de una aplicación de tiempo real es el *plazo* de ejecución (*deadline*) de cada una de las tareas que conforman la aplicación. Un *plazo* de ejecución es el tiempo en el cual una tarea debe ser completada. Específicamente, es el tiempo en el cual el hilo de tiempo real debe haber finalizado su ejecución actual.

En un *sistema de tiempo real estricto*, la pérdida o incumplimiento de un plazo puede ser catastrófico; sin embargo, el presente trabajo de tesis se enfoca en *sistemas de tiempo real flexible*, que pueden ejecutarse sobre una red de área local e incluso sobre *Internet*. Este tipo de sistemas puede tolerar la pérdida de algunos plazos y; por ende, se tiene como objetivo que el número de plazos perdidos sea el menor posible.

6.3.3.1 Ejecución de pruebas

En el servidor es necesario incorporar un nuevo componente para detectar la pérdida de plazos e informar al *supervisor* de tales eventos. Incluimos en la *aplicación servidor* un *manejador de plazos perdidos* basado en el ejemplo presentado en [24], y al cual hemos añadido código para la comunicación con el *supervisor*.

El manejador de plazos perdidos así como el código que ejecuta cada hilo de tiempo real durante esta prueba, son similares a los expuestos en la sección 3.7.3.3, donde existe un doble ciclo que consume tiempo de procesador y se producen pérdidas de plazos. La clase `MissHdlr` representa el manejador y hereda de `AsyncEventHandler`, una clase perteneciente a la *RTSJ* que permite definir un *manejador de eventos asíncrono*. Una instancia de `MissHdlr` debe ser asociada al hilo de tiempo real a través del último argumento del constructor de la clase `PeriodicParameters`. Este argumento especifica el objeto que manejará los plazos perdidos.

Hilo de tiempo real	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de activaciones**	**Prioridad (1ª prueba)**	**Prioridad (2ª prueba)**	**Prioridad (3ª prueba)**
1	700	300	100	11	50	90
2	800	300	100	11	50	90
3	900	300	100	11	50	90
4	1000	300	100	11	50	90

Tabla 6.4 Parámetros de los hilos de tiempo real

Para el experimento realizamos 3 pruebas. Cada prueba consiste en la ejecución simultánea de 4 hilos (clientes) con parámetros de tiempo real mostrados en la tabla 6.4. Únicamente el valor de prioridad de cada hilo es modificado en cada prueba. Los valores son: 11, para la primera prueba,

50 para la segunda y 90 para la tercera.

La asignación de prioridades está de acuerdo con el rango establecido por la *JVM-RT* (*TimeSys RI*), con valores entre 11 y 90, donde un valor de prioridad mayor tiene preferencia de ejecución sobre un valor de prioridad menor (ver sección 3.6.3.5). Por tanto, hemos seleccionado el valor mínimo, uno intermedio y el valor máximo, con el propósito de analizar la influencia que tiene el valor de la prioridad en el cumplimiento de plazos.

6.3.3.2 Análisis de resultados de plazos perdidos

Debemos recordar que el planificador por defecto de la *JVM-RT* está basado en prioridad y, de acuerdo con este valor, se produce la ejecución de los hilos de tiempo real en un orden determinado. Una tarea (hilo de ejecución) con mayor prioridad tendrá mayor preferencia de ejecución que una de menor prioridad.

Precisamente este hecho se puede apreciar en la figura 6.11 donde se muestran los resultados de las pruebas. La mayor pérdida de plazos se produce en los hilos con una prioridad mínima (11) o intermedia (50) mientras que en el caso de la prioridad máxima (90), la pérdida de plazos es nula.

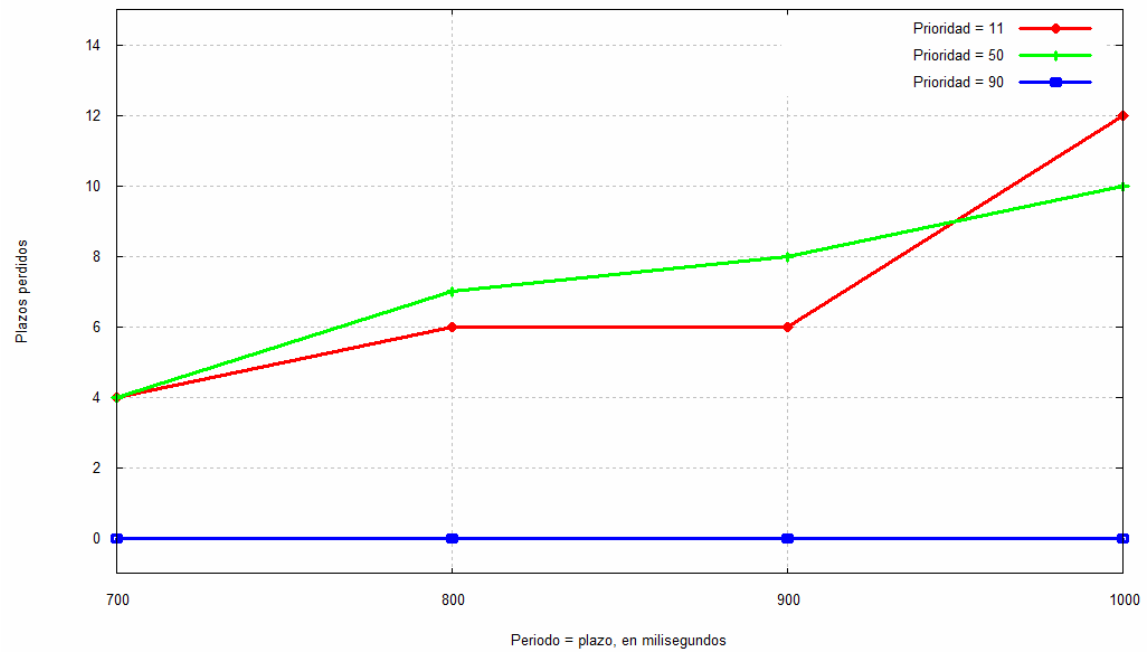


Figura 6.11: Plazos de ejecución perdidos

Tal resultado nos indica la gran influencia que tiene el valor de prioridad en la planificación de los hilos que constituyen una aplicación de tiempo real. En la siguiente sección aprovechamos la ventaja que ofrece el manejo de prioridades con el fin de implementar una política de planificación particular.

6.4 Adaptabilidad

Hasta el momento hemos analizado el comportamiento de las aplicaciones distribuidas de tiempo real bajo el control del único planificador provisto por la *RTSJ*. Aunque el propósito de esta especificación es soportar múltiples planificadores, **solamente define el planificador basado en prioridades** `PriorityScheduler`, cuya política de planificación determina que la ejecución de los hilos de tiempo real, por parte del procesador, es ordenada de acuerdo con una prioridad numérica. La prioridad es asignada por el desarrollador cuando los hilos son creados.

Sin embargo, **las aplicaciones de tiempo real exigen mayor flexibilidad en la planificación**. Algunas aplicaciones pueden necesitar ser planificadas con una cierta política mientras que otras pueden requerir diferentes políticas. En el presente trabajo de tesis exponemos la utilización de un planificador *round robin*, implementado en [21] pero que ha sido adaptado para trabajar con el esquema de prioridades del planificador base `PriorityScheduler`.

De tal forma, veremos que **nuestra solución permite la implementación y utilización de planificadores alternativos**, desarrollados de acuerdo con las necesidades de una aplicación en particular, siempre y cuando aproveche el esquema de prioridades subyacente. A continuación explicamos el procedimiento para incluir un planificador propio con el fin de controlar la ejecución de una aplicación distribuida de tiempo real.

6.4.1 Implementación de un planificador adicional

El principio sobre el cual construimos el planificador es el resultado obtenido en la sección 6.3.3 (pérdida de plazos). En dicho experimento observamos que un hilo de tiempo real con la máxima prioridad no perdió plazos. Este resultado es aprovechado para implementar un planificador que actuará en modo *round robin* y que permitirá que todos los hilos se ejecuten con la máxima prioridad y por una cantidad limitada de tiempo.

Las prioridades, tanto máxima como mínima, corresponden a los valores establecidos por la *JVM-RT*, *i.e.* 90 y 11, respectivamente. El proceso consiste en cambiar las prioridades de cada uno de los hilos de tiempo real entre mínima y máxima, todo ésto en tiempo de ejecución. Cabe mencionar que el nuevo planificador es un hilo de tiempo real adicional que debe ejecutarse en máxima prioridad con el fin de controlar a los demás hilos en ejecución. El código del planificador se presenta a continuación (página siguiente):

```
import javax.realtime.*;
import java.util.LinkedList;

public class RoundRobinScheduler extends RealtimeThread {
  private long tiempo;
  private static LinkedList hilos;
  private static RoundRobinScheduler planificador = null;
  static int contadorAgregar = 0;
  static int contadorObtener = 0;
  public RoundRobinScheduler(RealtimeThread rt)
  {
      hilos = new LinkedList();
      agregarHilo(rt);
      tiempo = 100;
      start();
  }
  public static RoundRobinScheduler startScheduler(RealtimeThread rt)
  {
    if(planificador == null)
            planificador = new RoundRobinScheduler(rt);
    agregarHilo(rt);
    return planificador;
  }
  public static void agregarHilo(RealtimeThread rt)
  {
    hilos.add(contadorAgregar, rt);
    rt.setPriority(PriorityScheduler.instance().getMinPriority());
    ++contadorAgregar;
  }
  public void run()
  {
    RealtimeThread actual;
    setPriority(PriorityScheduler.instance().getMaxPriority());
    while(true)
    {
      actual = (RealtimeThread) hilos.get(contadorObtener);
      if(actual == null)
              return;
      actual.setPriority(PriorityScheduler.instance().getMaxPriority());
      if(contadorObtener < (contadorAgregar-1))
         contadorObtener++;
      else
         contadorAgregar = 0;
      try
      {
        RealtimeThread.sleep(tiempo);
      }catch (InterruptedException ie){};
      actual.setPriority(PriorityScheduler.instance().getMinPriority());
    }
  }
}
```

Figura 6.12: Código del planificador *round robin* basado en prioridad

El planificador ha sido implementado en una clase denominada `RoundRobinScheduler` para ser congruente con la recomendación de la *RTSJ*, la cual establece que el nombre debe ser descriptivo de la política utilizada (*e.g.* `EDFScheduler`, `LLFScheduler`, *etc.*). Además, hereda de la clase `RealtimeThread` con el fin de que sea un hilo de tiempo real en ejecución.

El atributo `tiempo` es el intervalo durante el cual se ejecutan los restantes hilos que

conforman la aplicación. La clase `LinkedList` (paquete `java.util`) crea una lista denominada `hilos` que contendrá todos los hilos de ejecución de la aplicación y que serán controlados por el planificador. Un atributo de tipo estático denominado `planificador` será la instancia del planificador en ejecución. Dos variables (`contadorAgregar` y `contadorObtener`) actuarán como índices para controlar el acceso secuencial a la lista.

En cuanto a los métodos, el constructor de la clase se encarga de recibir el primer hilo de tiempo real que solicita ejecutarse bajo el control del planificador; la lista es creada e inmediatamente el hilo es agregado. Luego, la cantidad de tiempo (en milisegundos) es asignada y enseguida comienza la ejecución del planificador con el método `start()` que llamará al método `run()`.

Una atención especial merece el método `startScheduler()` puesto que es el puente de comunicación con la aplicación distribuida de tiempo real. Cada hilo de la aplicación solicita ejecutarse bajo el nuevo planificador a través de la invocación de dicho método. De esta forma, cualquier aplicación que implemente su propio planificador debe incluir este método para que sea la interfaz con el mecanismo de solución.

Una función importante de `startScheduler()`es controlar la ejecución de una sola instancia del planificador. Cada vez que se crea y agrega un nuevo hilo, se verifica el valor del atributo estático. Si es `null`, es el primer hilo que solicita el planificador y se crea la instancia; en caso contrario, el planificador ya se encuentra en ejecución y esa misma instancia es devuelta.

El método `agregarHilo()` inserta un nuevo hilo al final de la lista por medio del método `add()` perteneciente a la clase estándar `LinkedList`; luego, con el método `setPriority()` establecemos la prioridad del hilo en el valor mínimo y el contador es incrementado para la siguiente posición en la lista.

El método `run()` efectúa la planificación deseada. Cuando comienza la ejecución del planificador, su prioridad es máxima. Mediante la instrucción `while()`, el planificador se ejecutará indefinidamente, obteniendo uno por uno los hilos almacenados en la lista. Cada vez que se obtiene un hilo, su prioridad es asignada al valor máximo, asegurando su ejecución durante el tiempo que el planificador permanezca “dormido”. El método `sleep()` realiza esta labor por el número especificado de milisegundos. Transcurrido el tiempo establecido, el planificador “despierta” y se convierte en el hilo actual en ejecución, asignando el valor mínimo de prioridad al hilo previamente en ejecución.

6.4.2 Utilización del planificador implementado

Con esta aproximación ha sido posible implementar un tipo de planificador distinto al estándar. Mediante el ajuste de la prioridad de cada uno de los hilos en tiempo de ejecución, podemos afectar la planificación predeterminada y hacer que se comporte como lo requiere la aplicación. Por tanto, **una aplicación es capaz de implementar su propia política de planificación basada en el esquema de prioridades de la *RTSJ***.

Para la utilización del planificador alterno, pueden presentarse dos escenarios:

1. El archivo de clase correspondiente al planificador existe tanto en el cliente como en el servidor.

2. La clase debe ser enviada desde el cliente hacia el servidor conjuntamente con los parámetros de tiempo real.

La figura 6.13 muestra la sección de código que hemos añadido en la clase `ParametrosTiempoReal` para verificar la existencia del archivo de clase en el servidor.

```
clasePlanificador = new File(nuevosParametros.obtenerPolitica() + ".class");

...
...
...

if(!nuevosParametros.obtenerPolitica().equals("PriorityScheduler"))
{
  Class clase = Class.forName(nuevosParametros.obtenerPolitica());
  Method metodo = null;
  Method[] metodos = clase.getMethods();
  for(int i=0; i< metodos.length; ++i)
        if(metodos[i].getName().equals("startScheduler"))
            metodo = metodos[i];
  metodo.invoke(clasePlanificador, new Object[]{nuevoClienteTiempoReal});
}
```

Figura 6.13: Sección del código de la clase `ParametrosTiempoReal`

Una vez que los parámetros de tiempo real han sido recibidos por el servidor, el campo correspondiente a la política de planificación es obtenido con el método `obtenerPolitica()` y es comparado con el planificador por defecto `PriorityScheduler`. En caso de no ser igual, se debe encontrar la clase respectiva y ejecutar el método `startScheduler()`.

6.4.3 Uso de la *reflectividad* y *carga dinámica*

Para la búsqueda del archivo de clase correspondiente al planificador y la ejecución del método `startScheduler()`, aprovechamos las utilidades ofrecidas por *Java* denominadas ***reflectividad*** y ***carga dinámica***, que permiten obtener información de una determinada clase y cargar los *bytecodes* de la misma en tiempo de ejecución, respectivamente [42].

Para tal efecto, *Java* proporciona la clase `Class` y mediante su método `forName()` verificamos si el archivo *.class* está disponible para la *JVM-RT*. Con el método `getMethods()` se devuelve un arreglo de objetos que representan los métodos de la clase. Recorremos dicho arreglo para buscar el método que inicializará el planificador (`startScheduler()`) y su ejecución es a través del método `invoke()`.

En el caso de que la política de planificación no se encuentre definida en la *JVM-RT* del servidor, es necesario enviar el código de la clase (*bytecodes*) del planificador solicitado.

```
        if(existePolitica == 0)
        {
          System.out.println("Enviando planificador...");
          String politica = nuevosParametros.obtenerPolitica();
          new ClienteCargaDinamica(politica);
        }
```

Figura 6.14: Sección del código fuente de la clase `InterfazRemota`

La figura 6.14 muestra la sección de código que agregamos dentro de la clase `InterfazRemota` (localizada en el cliente) y que se encarga de enviar el archivo *.class* al servidor. Previamente, el servidor responde al cliente si dispone o no del archivo. Tal respuesta se almacena en la variable `existePolitica`.

Hemos incorporado dos clases a la solución original con el fin de permitir el envío, recepción y carga de la clase del planificador en tiempo de ejecución. Una de ellas, la clase `ClienteCargaDinamica` se encarga de enviar el archivo identificado por el argumento `politica`.

```
...
FileOutputStream salidaArchivo = (FileOutputStream) socket.getOutputStream();
FileInputStream entradaArchivo = new FileInputStream(clasePlanificador);
byte[] buffer = new byte[1024];
int longitud;
while((longitud = entradaArchivo.read(buffer))>0)
{
  salidaArchivo.write(buffer, 0, longitud);
}
...
```

Figura 6.15: Sección del código de la clase `ClienteCargaDinamica`

La figura 6.15 contiene la sección principal de la clase `ClienteCargaDinamica` que incluye el código necesario para transmitir los *bytecodes* del archivo de clase de la política solicitada por el cliente. Básicamente, se utiliza un flujo de lectura y escritura a un archivo que es enviado por el *socket* de comunicación como un arreglo de *bytes*.

La figura 6.16 incluye el código de la clase `ServidorCargaDinamica`, localizada en el servidor, que se encarga de recibir y almacenar el archivo de clase enviado por el cliente.

Una vez que el archivo de clase del planificador se encuentra en el servidor, es cargado en tiempo de ejecución. De esta forma, la *JVM-RT* del servidor puede utilizar esta clase para controlar la ejecución de los hilos de una aplicación distribuida. Con este mecanismo proporcionamos un **enfoque dinámico a la solución** propuesta, puesto que un desarrollador puede definir una política de planificación propia mediante un archivo de clase, el cual será utilizado por la aplicación en tiempo de ejecución.

```
...
servidor = new ServerSocket(2222);
cliente = servidor.accept();
InputStream entradaArchivo = cliente.getInputStream();
FileOutputStream salidaArchivo = new FileOutputStream(clasePlanificador);
byte[] buffer = new byte[1024];
int longitud;
while((longitud = entradaArchivo.read(buffer))>0)
{
  salidaArchivo.write(buffer, 0, longitud);
}
...
```

Figura 6.16: Sección del código de la clase `ServidorCargaDinamica`

6.4.4 Ejecución utilizando el planificador implementado

La aplicación de prueba es similar a la presentada en el experimento de plazos perdidos (sección 6.3.3). Esta vez el servidor atiende 5 clientes de manera simultánea. Para confrontar el desempeño del planificador estándar y el implementado, hemos realizado 4 rondas de ejecución: las tres primeras con el planificador `PriorityScheduler` pero con niveles de prioridad bajo (15), intermedio (40) y alto (89). La última ronda es realizada bajo el control del planificador propio `RoundRobinScheduler`. Los parámetros de tiempo real utilizados por cada cliente se muestran en las tablas 6.5 y 6.6.

Hilo de tiempo real	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de activaciones**	**Prioridad (1ª prueba)**	**Prioridad (2ª prueba)**	**Prioridad (3ª prueba)**
1	600	300	100	15	40	89
2	700	300	100	15	40	89
3	800	300	100	15	40	89
4	900	300	100	15	40	89
5	1000	300	100	15	40	89

Tabla 6.5: Parámetros de prueba bajo el planificador `PriorityScheduler`

Hilo de tiempo real	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de Activaciones**	**Prioridad (4ª prueba)**
1	600	300	100	Automática*
2	700	300	100	Automática*
3	800	300	100	Automática*
4	900	300	100	Automática*
5	1000	300	100	Automática*

*El planificador se encarga de asignar dinámicamente la prioridad

Tabla 6.6: Parámetros de prueba bajo el planificador `RoundRobinScheduler`

Para mostrar la utilidad de la *carga dinámica*, el archivo de clase del planificador `RoundRobinScheduler` sólo se encuentra almacenado en el cliente. Una vez que el primer cliente solicita la ejecución bajo este planificador, los *bytecodes* son transmitidos y se produce la carga dinámica del planificador. Para la ejecución de los restantes hilos de tiempo real, el planificador ya forma parte del servidor.

6.4.5 Análisis de resultados con el nuevo planificador

En la figura 6.17 representamos el número de plazos perdidos cuando se ejecuta una aplicación distribuida de tiempo real utilizando el planificador estándar `PriorityScheduler` así como el planificador `RoundRobinScheduler`.

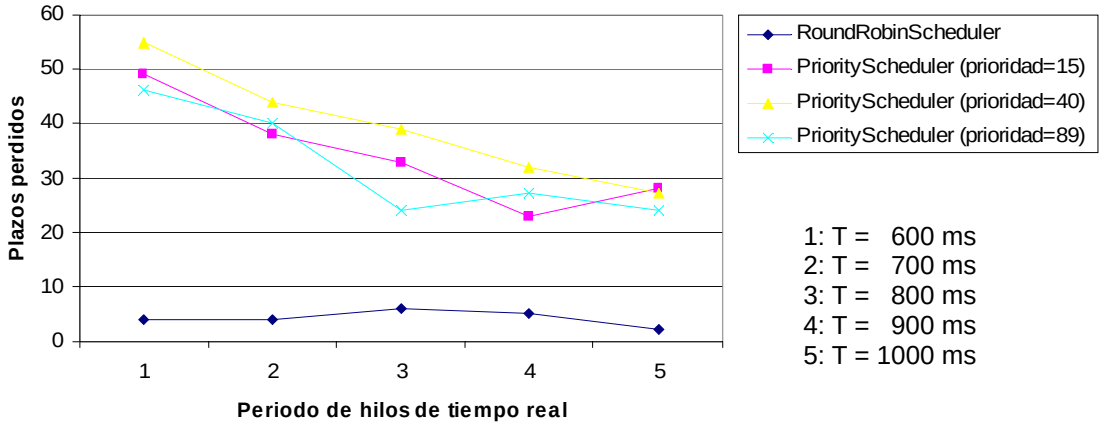


Figura 6.17. Plazos de ejecución perdidos por cada planificador

En cualquiera de los tres casos que corresponden a la ejecución de la aplicación bajo el planificador `PriorityScheduler`, la pérdida de plazos supera a la obtenida por el planificador `RoundRobinScheduler`. Este resultado puede ser aprovechado en aplicaciones que involucran tareas con diferente nivel de criticidad, *i.e.* una tarea puede ser considerada más crítica que otra. Mediante la asignación de una prioridad mayor es posible garantizar su preferencia de ejecución sobre las demás tareas.

De esta manera hemos mostrado cómo incorporar un planificador definido por una aplicación distribuida en particular. Dicho planificador utiliza la asignación dinámica de prioridades para cada una de las tareas, consiguiendo una menor pérdida de plazos.

6.5 Versatilidad

En esta sección vamos a desarrollar tres aplicaciones distribuidas con parámetros de tiempo real basadas en el conjunto de clases que conforman la solución propuesta. Aunque las dos primeras aplicaciones pueden no ser consideradas como casos de estudio en el dominio de tiempo real, nos permiten simular el procesamiento que se lleva a cabo en sistemas de este tipo, donde se

requieren cálculos que son útiles *e.g.* para el monitoreo de las condiciones físicas o químicas en el ambiente (temperatura, presión, humedad), para sensores de flujos o caudales en tuberías, procesamiento de señales, *etc*. Cada uno de los ejemplos permite ilustrar la utilización de nuestra herramienta de desarrollo y mostrar una forma distinta de implementar la *aplicación servidor*, mediante:

1. La implementación del método `run()` de la clase `Aplicacion`.

2. El método `run()` de la clase `Aplicacion` como interfaz para acceder a una clase adicional que implementa el servicio.

3. La utilización de una clase adicional, tanto en el cliente como en el servidor. Esta clase define los datos y las operaciones que implementan los servicios.

6.5.1 Ejemplo 1: *sucesión de Fibonacci*

Implementamos una aplicación distribuida que permite a uno o varios clientes, solicitar al servidor la generación de una cantidad arbitraria de términos de la *sucesión de Fibonacci*. Tal solicitud es realizada bajo ciertos parámetros de tiempo real (política, prioridad, periodo, y plazo). Además, el cliente debe especificar la cantidad de términos que desea. Este dato se convierte en un argumento necesario para el método remoto, por lo que es añadido a los parámetros de tiempo real en la línea de comandos.

En la sección 5.2.1 explicamos la manera de considerar un argumento extra. Puesto que se trata de un único argumento de tipo entero, no es necesario crear un objeto, simplemente añadimos este dato a los parámetros que necesita el cliente. El servidor recibe los parámetros (junto con el argumento) en la clase `ParametrosTiempoReal` y son pasados a la clase `Aplicacion`, en la cual hemos incluido un dato de tipo entero en su constructor.

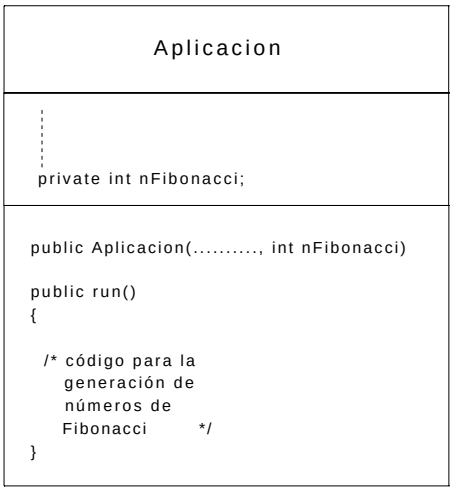


Figura 6.18: Clase `Aplicacion` para la *sucesión de Fibonacci*

La figura 6.18 muestra la clase `Aplicacion` donde se incluye un atributo privado de tipo entero denominado `nFibonacci` y que representa la cantidad de números que el servidor debe generar. En el método `run()` implementamos el código que calcula secuencialmente los números de *Fibonacci*. Cada número de la sucesión es la suma de los dos inmediatamente anteriores y el cálculo de cada término es realizado periódicamente de acuerdo con los parámetros de tiempo real. Cada uno de los términos generados es enviado al cliente y mostrado en pantalla hasta que se completa la secuencia solicitada.

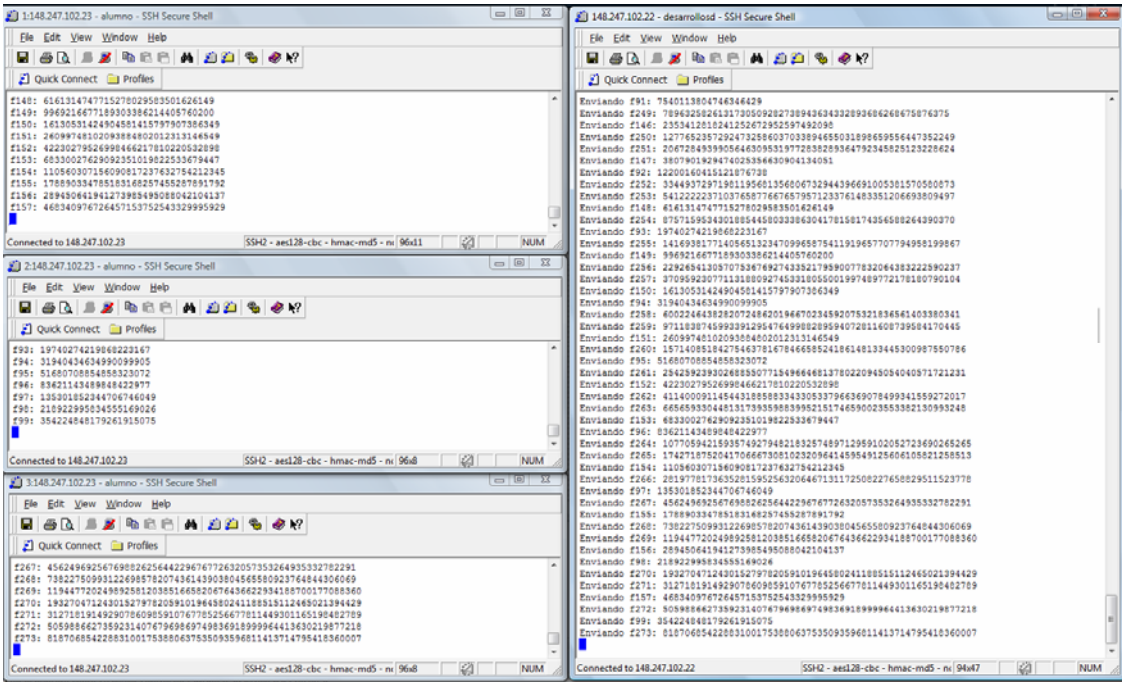
Sobre la plataforma de pruebas descrita en la sección 6.1, vamos a realizar la ejecución de la aplicación distribuida. Se tienen 3 clientes que solicitan simultáneamente la generación de números de *Fibonacci*, de acuerdo con los parámetros que se muestran en la tabla 6.7.

Cliente	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Número de Activaciones**	**Política**	**Prioridad**
1	2000	200	300	PriorityScheduler	25
2	3000	200	400	PriorityScheduler	25
3	1000	200	500	PriorityScheduler	25

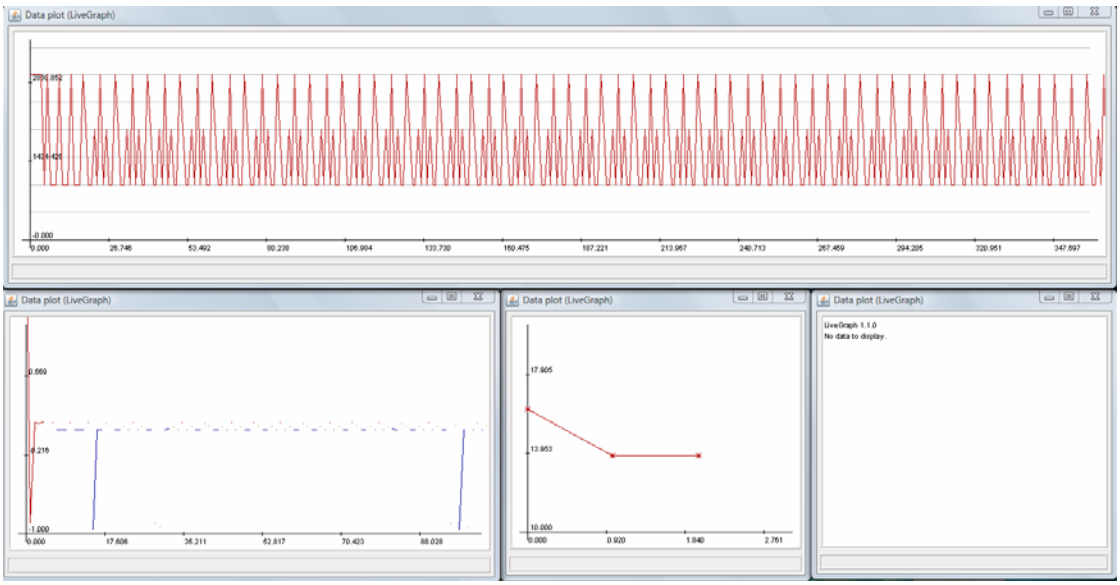
Tabla 6.7: Parámetros de tiempo real para generar números de *Fibonacci*

El número de activaciones equivale a la cantidad de números de *Fibonacci* solicitados por cada uno de los clientes. La figura 6.19 muestra las pantallas de salida de la ejecución (a) así como las pantallas que genera el software supervisor (b). En la figura 6.19 (a), cada una de las ventanas de la parte izquierda de la figura, corresponde a una terminal de conexión remota con el cliente, desde la cual se solicita al servidor la generación de cierta cantidad de números de *Fibonacci*. Esta cantidad está determinada por el número de activaciones de cada cliente. En estas terminales podemos apreciar cómo se reciben los números de *Fibonacci* generados por el servidor. Cabe mencionar que el servidor produce los resultados de manera simultánea para todos los clientes. Precisamente en la parte derecha de la figura, se encuentra la terminal de conexión remota con el servidor, donde se observan los resultados de la ejecución.

En la figura 6.19 (b), se muestran las ventanas que pertenecen al software de supervisión. En la parte superior podemos visualizar el valor del periodo de cada uno de los hilos con respecto al transcurso del tiempo, desde que comenzó la ejecución de la aplicación distribuida. En la parte inferior, la primera ventana (izquierda) presenta los datos de variación del periodo, *i.e.* la diferencia entre el valor del periodo calculado y el valor establecido. La segunda ventana (central) muestra los datos de latencia de red. Aquí se confirman los resultados experimentales de la sección 6.3.1, donde concluimos que el valor de latencia para el primer cliente es mucho más alto que para los demás, debido al tiempo que consume el proceso de apertura del *socket* de conexión. La tercera ventana (derecha) está destinada para representar los plazos perdidos; sin embargo, durante la ejecución no ocurrieron dichas pérdidas, puesto que el tiempo de cómputo para cada término de la sucesión, está por debajo del plazo establecido. Cabe recordar que la visualización de estos resultados se produce al mismo tiempo que la ejecución de la aplicación distribuida. Los datos son enviados al supervisor y son almacenados en un archivo de datos, el cual es accedido cada segundo con el fin de actualizar la gráfica.



(a)



(b)

Figura 6.19: Ejemplo de ejecución de la *sucesión de Fibonacci*

El objetivo de esta aplicación fue explicar la implementación de un servicio por medio del método `run()`dentro de la clase `Aplicacion`. Además, hemos considerado el paso de un argumento requerido para la ejecución del método. Puesto que cada número de la sucesión es calculado en intervalos de tiempo regulares, la aplicación se asemeja a la ejecución de una tarea periódica en aplicaciones de tiempo real.

6.5.2 Ejemplo 2: generación de números primos

En muchas aplicaciones puede ser necesaria la implementación de una clase extra donde definimos el código que deseamos ejecutar. En este caso, el método `run()` de la clase `Aplicacion` puede **actuar como una interfaz** para crear una instancia de la clase que finalmente implementa el servicio.

Para ilustrar esta funcionalidad, vamos a considerar el cálculo de números primos mediante la *criba de Eratóstenes* [45], un algoritmo que sirve para hallar todos los números primos menores que un entero dado.

La figura 6.20 muestra la clase `Primos`, creada para implementar el algoritmo mencionado. La interacción entre la clase `Aplicación` y la nueva clase se efectúa por medio del método `run()`.

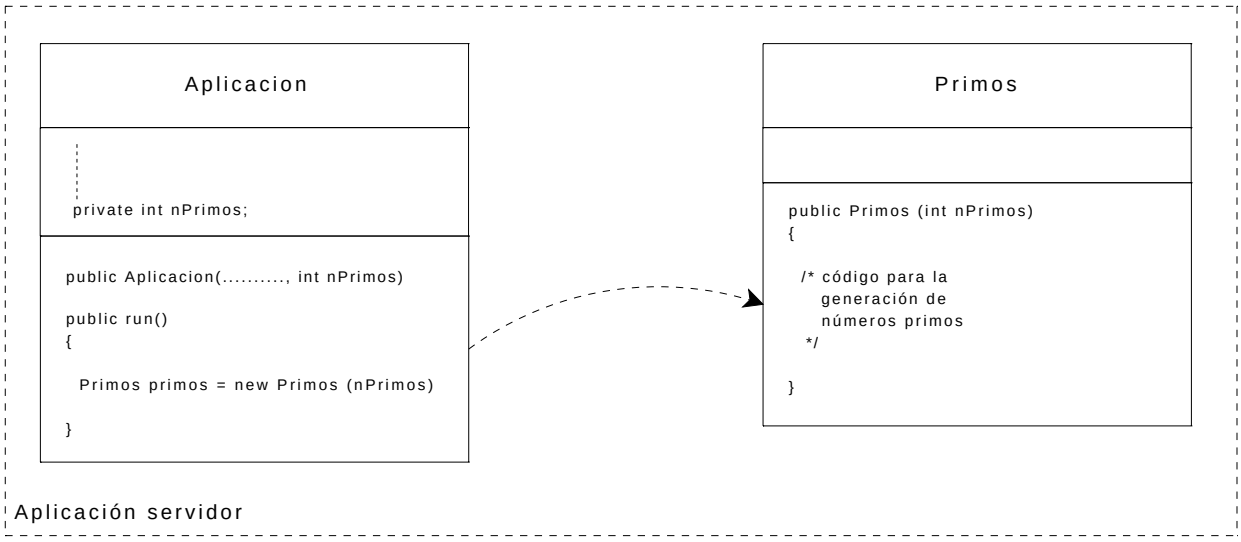


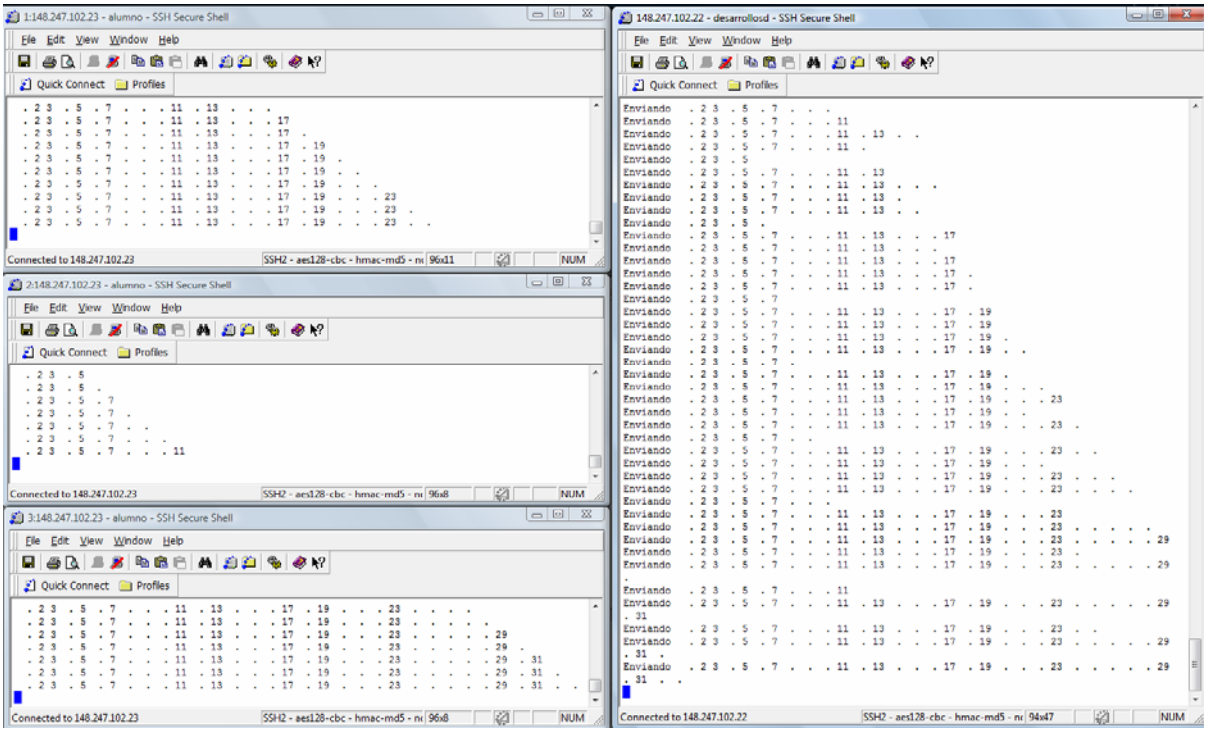
Figura 6.20: Clases `Aplicacion` y `Primos` para generación de números primos

De igual manera que en el ejemplo anterior, el cliente debe especificar un argumento de tipo entero que limitará la cantidad de números primos generados. Una vez que este argumento es recibido por la clase `Aplicacion` a través de la variable `nPrimos`, el método `run()` se encarga de crear una instancia de la clase `Primos`, cuyo constructor realiza el cálculo sucesivo de números primos, considerando el límite establecido por el cliente.

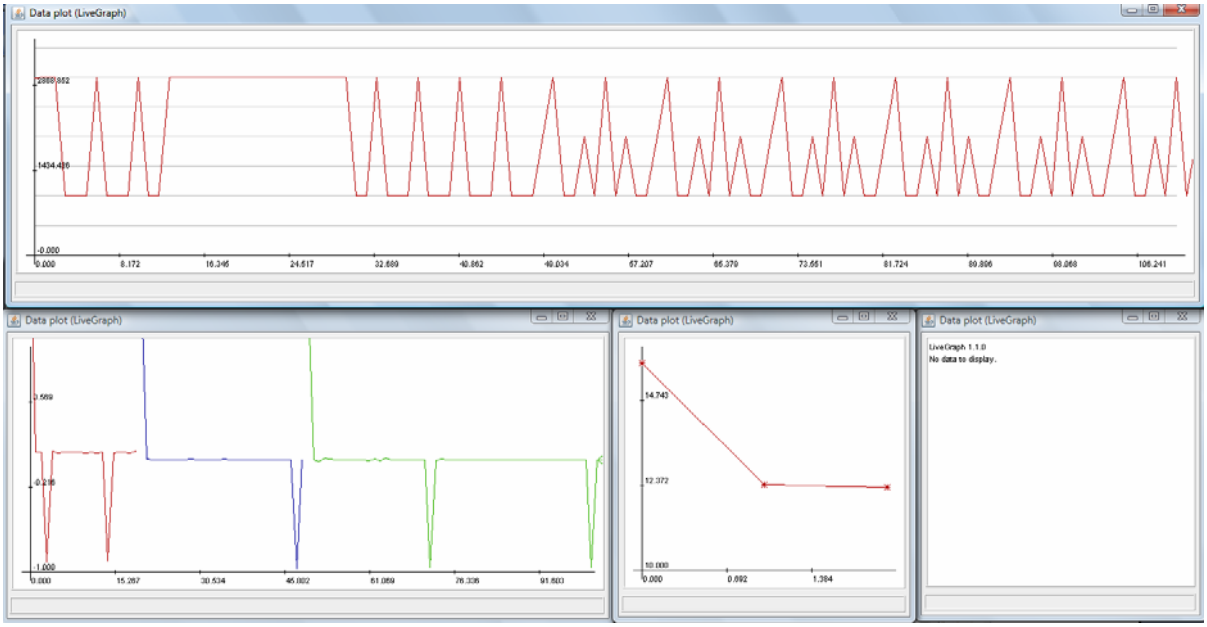
Cliente	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Cantidad de números primos**	**Política**	**Prioridad**
1	2000	200	300	PriorityScheduler	25
2	3000	200	400	PriorityScheduler	25
3	1000	200	500	PriorityScheduler	25

Tabla 6.8: Parámetros de tiempo real para generar números primos

Como en el experimento anterior, realizamos la ejecución de la aplicación considerando 3 clientes con parámetros de tiempo real, mostrados en la tabla 6.8. En este caso, el número de activaciones corresponde a la cantidad deseada de número primos. Cada uno de los números primos generado periódicamente se convierte en una respuesta enviada al cliente.



(a)



(b)

Figura 6.21: Ejemplo de ejecución de la generación de números primos

La figura 6.21 muestra la ejecución de la aplicación. El procedimiento y la disposición de las ventanas son semejantes al ejemplo anterior. El servidor atiende simultáneamente a 3 clientes y genera los resultados respectivos para cada cliente. Los resultados son visualizados en su correspondiente terminal. Mediante este ejemplo mostramos cómo **la solución desarrollada puede ser adaptada a problemas que requieren de la implementación de clases específicas** y no únicamente la ejecución de un método remoto. Estas clases pueden ser añadidas al esquema de solución base.

6.5.3 Ejemplo 3: simulación de comparación de imágenes

La aplicación distribuida que vamos a implementar en este ejemplo se aproxima más a lo que podríamos encontrar en la práctica y tiene relación con el procesamiento y la comparación de imágenes en una determinada escala de color. Una imagen digital puede ser representada como una matriz, cuyos elementos corresponden al valor de un *píxel* y representan un tono en la escala de colores. La cantidad de tonos está limitada por el rango de la matriz así que mientras mayor es el rango, mayor es la cantidad de tonos de color que podemos definir. Para el ejemplo, consideramos matrices con igual número de filas y columnas. La idea es que uno o varios clientes, con bajo poder de procesamiento, soliciten a un servidor de alto rendimiento, la comparación de dos matrices para verificar si las imágenes son iguales; en caso contrario, identificar los elementos que no coinciden. El servidor presenta en pantalla un reporte con las posiciones en donde difieren los elementos de las matrices.

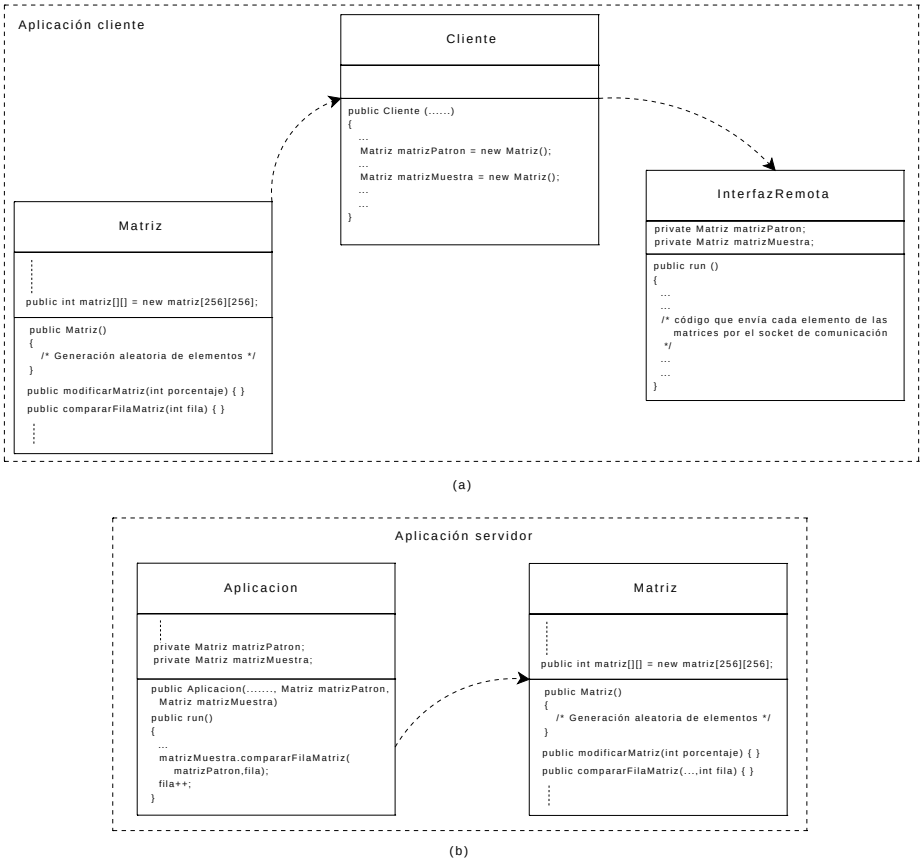


Figura 6.22: Clases para la comparación de matrices

El cliente debe enviar dos matrices, para lo cual, creamos una clase `Matriz` que se añade al esquema de clases básico de la solución. Para simular el proceso, el cliente genera de manera aleatoria los elementos que permitirán crear una instancia de la clase `Matriz`, denominada *matriz patrón*. Una rutina que modifica los elementos de la matriz patrón, de acuerdo con un porcentaje especificado por el cliente, nos permite generar otra instancia de la clase `Matriz`, a la que denominamos *matriz muestra*.

La figura 6.22 muestra la clase adicional `Matriz` tanto en la *aplicación cliente* (a) como en la *aplicación servidor* (b). Luego de que las matrices patrón y muestra han sido generadas, el cliente debe enviarlas al servidor junto con los parámetros de tiempo real correspondientes a la política de planificación, prioridad, periodo y plazo de ejecución. La clase `InterfazRemota` se encarga del envío mientras que la recepción es realizada por la clase `ParametrosTiempoReal`, quien reconstruye las matrices mediante instancias de la clase `Matriz`.

Una vez que las matrices se encuentran en el servidor, son pasadas a la clase `Aplicacion` y, dentro del método `run()`, se realiza la invocación del método perteneciente a `Matriz` y que efectúa la comparación de matrices fila por fila. Cada comparación es realizada periódicamente y el resultado es un reporte mostrado en pantalla que contiene la ubicación en que los elementos de las matrices difieren.

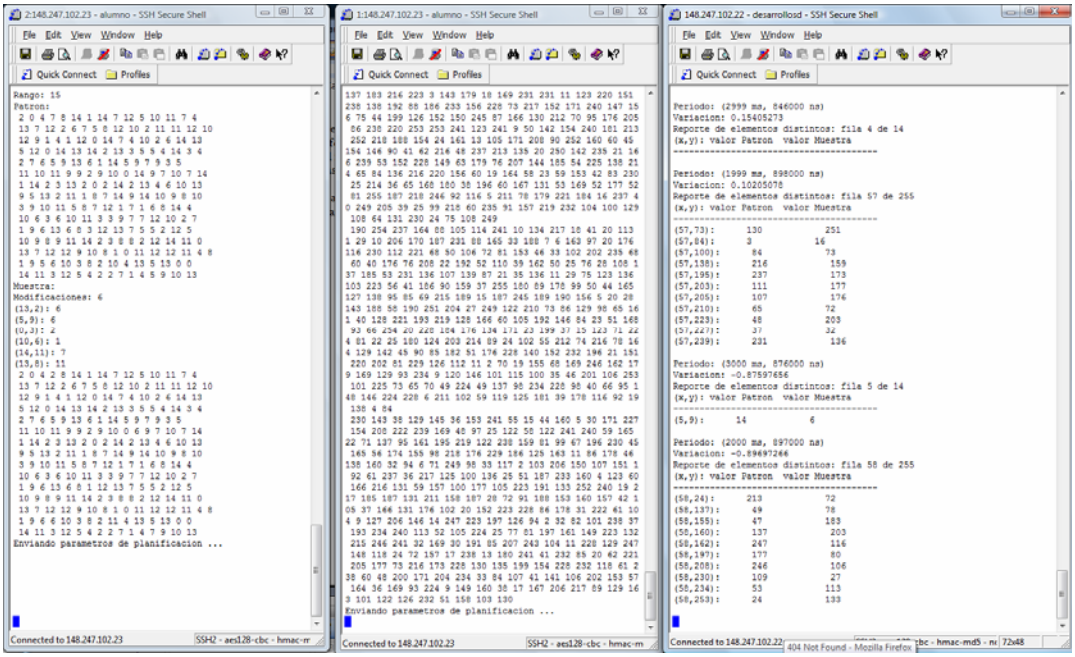
La aplicación distribuida de tiempo real considera la ejecución simultánea de 2 clientes con los parámetros mostrados en la tabla 6.9.

Cliente	**Periodo** (*milisegundos*)	**Plazo** (*milisegundos*)	**Rango matriz**	**Política**	**Prioridad**	**Modificaciones**
1	2000	200	256	PriorityScheduler	25	3276
2	10000	200	15	RoundRobinScheduler	Automática	6

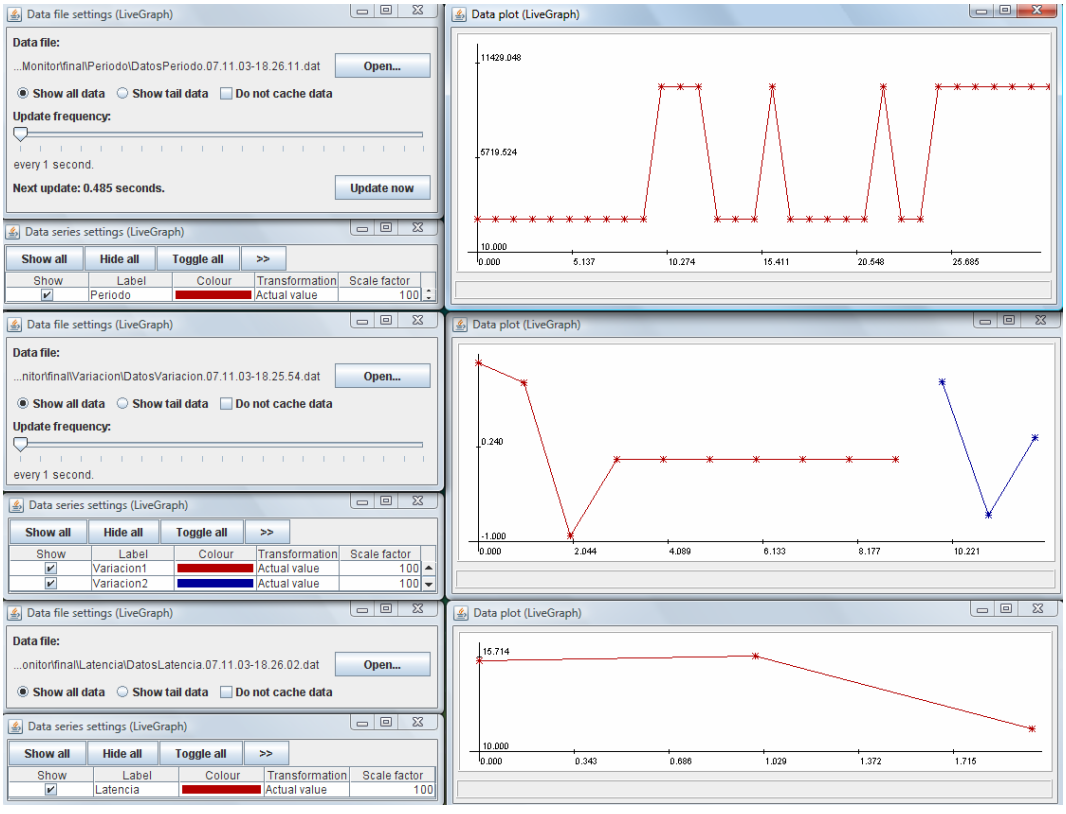
Tabla 6.9: Parámetros de tiempo real para comparación de matrices

Realizamos un ejemplo de ejecución de la aplicación distribuida de tiempo real para la comparación de matrices. Hemos diseñado la aplicación para que el rango de la matriz no sea estático sino que pueda ser especificado por el cliente.

En la figura 6.23 (a), la ventana izquierda pertenece a uno de los clientes, el cual genera una matriz de rango igual a 15, y a partir de ella, se genera otra matriz del mismo rango, con 6 elementos modificados. En la ventana central, el otro cliente ha generado una matriz con valor de rango igual a 256 y 3276 elementos modificados de los 65536 (256^2) que hay en total. Por esta razón, aparece una gran cantidad de números en la ventana. Cada cliente envía su par de matrices al servidor para su comparación. La ventana derecha corresponde al servidor, donde podemos apreciar los reportes de comparación que son producidos de manera periódica.



(a)



(b)

Figura 6.23: Ejemplo de ejecución de comparación de matrices

La figura 6.23 (b) muestra las ventanas del software de supervisión. En la parte izquierda aparecen las ventanas donde se pueden seleccionar los archivos de datos que serán visualizados. En la parte derecha de la figura, se presentan las gráficas del periodo (superior), variación del periodo (diferencia entre el periodo calculado y el establecido) en la gráfica de la parte media, y latencia de red (gráfica inferior).

En este ejemplo hemos creado una clase extra, en la cual, uno de los métodos contiene la implementación del servicio. Para fines de la aplicación, denominamos a esta clase como `Matriz`, sin embargo, es posible definir una o más clases adicionales de acuerdo con las necesidades de una aplicación en particular. Puede suceder que la clase o clases adicionales no existan en el servidor; en tal caso, **el desarrollador puede aprovechar la carga dinámica de *bytecodes***, tal como fue utilizada para el planificador personalizado (sección 6.4.3). Con este procedimiento, **el cliente puede proveer todos los elementos (clases) necesarios al servidor** para que pueda realizar la ejecución.

Por medio de las aplicaciones descritas, hemos demostrado la implementación de aplicaciones con base en nuestra herramienta de desarrollo. En todos los ejemplos, los servicios implementados en el método `run()` de la clase `Aplicacion` o en métodos pertenecientes a clases adicionales, pueden ser accedidos por uno o varios clientes simultáneamente y bajo los parámetros de tiempo real especificados.

La solución desarrollada tiene la capacidad para adaptarse a diferentes escenarios. En el primer ejemplo, el servicio está implementado en el método `run()` de la clase `Aplicacion`, la cual hereda de la clase `RealtimeThread` y, por tanto, es un hilo de ejecución. Podemos realizar la ejecución simultánea de varios hilos de tiempo real desde un mismo cliente. Cada hilo puede ejecutarse con sus respectivos parámetros de tiempo real.

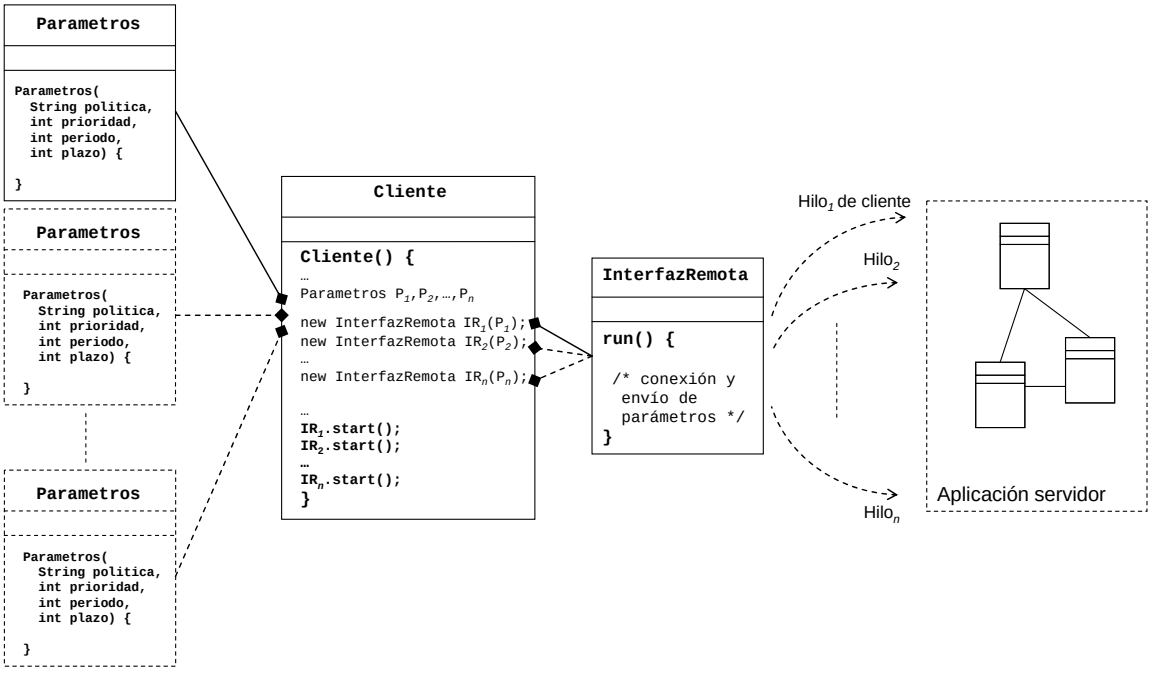


Figura 6.24: Esquema para cliente *multi-hilo*

La figura 6.24 ilustra la funcionalidad anterior. En el código de la clase `Cliente`, podemos crear varias instancias de la clase `Parametros` (P_1, P_2,...,P_n) con el fin de solicitar varios hilos de ejecución, cada uno con sus propios parámetros de política, prioridad, periodo, y plazo. De igual forma, creamos el mismo número de instancias de la clase `InterfazRemota`, (IR_1, IR_2,...,IR_n), para enviar los parámetros de tiempo real al servidor, quien se encarga de ejecutar simultáneamente todos los hilos.

En el segundo y tercer ejemplos, utilizamos el método `run()` de la clase `Aplicacion` como una **interfaz** para crear una instancia de una clase adicional. Con este esquema, podemos definir varios métodos y, por ende, varios servicios que pueden ser accedidos por los clientes. La alternativa más clara es crear varias clases que desempeñen la función de `Aplicacion`, cada una representada en el cliente por su respectiva clase `InterfazRemota`, tal como se muestra en la figura 6.25.

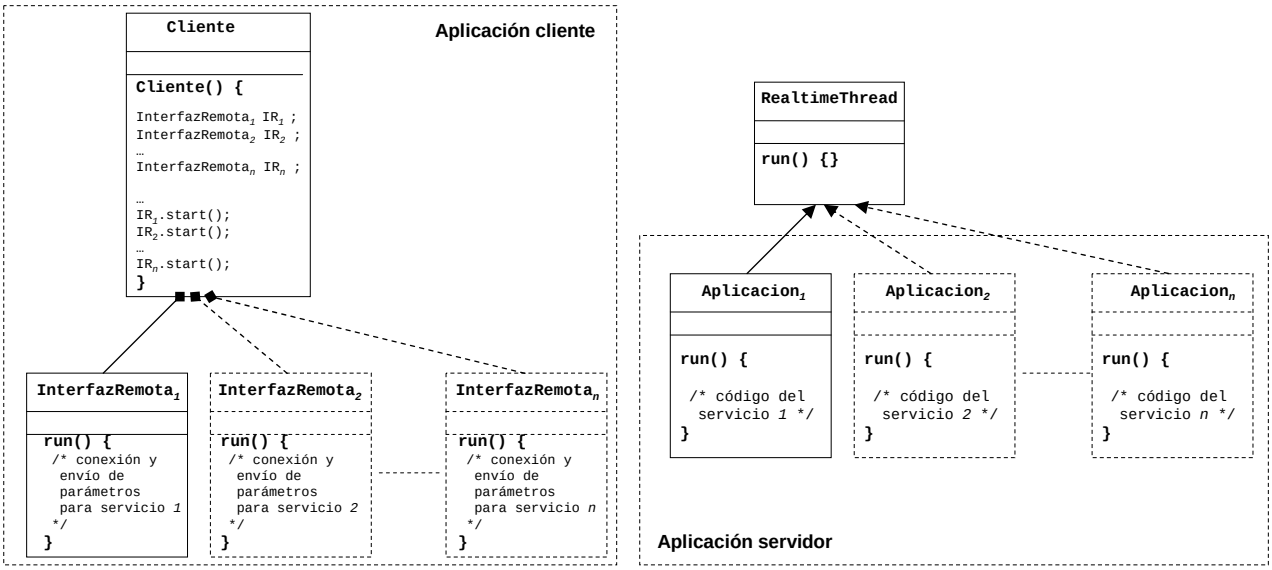


Figura 6.25: Esquema para varios servicios

Sin embargo, este esquema puede resultar algo inconveniente, principalmente porque añaden muchas clases a la estructura original. Por cada clase `Aplicacion`, se requiere de una clase `InterfazRemota` asociada.

La figura 6.26 muestra una alternativa más elegante, que consiste en crear una sola clase adicional denominada `Servicio`, donde se definen todos los servicios deseados, cada uno implementado en un método de la clase. El método `run()` de la clase `Aplicacion` se comporta como una interfaz, desde la cual se invoca cualquiera de los métodos, previamente creando una instancia de la clase `Servicio`. Con este objeto podemos invocar cualquiera de los métodos.

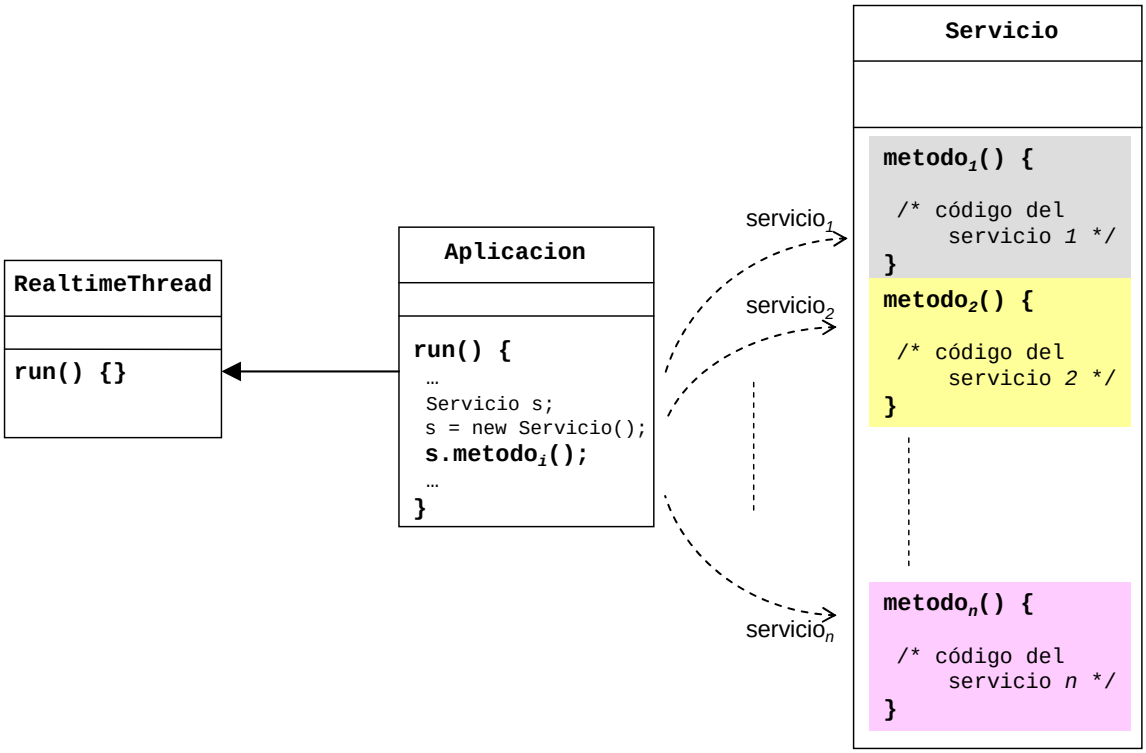


Figura 6.26: Esquema para varios servicios en una sola clase

Capítulo 7

Conclusiones y comentarios finales

El rápido avance tecnológico, especialmente relacionado con el hardware de cómputo y las redes de comunicaciones, así como el desarrollo de grandes volúmenes de información, han motivado el aparecimiento de aplicaciones más complejas y que requieren de un mejor desempeño, incremento de la disponibilidad, mayor seguridad, continua escalabilidad, *etc*. Tales necesidades difícilmente pueden ser cubiertas por una sola computadora, razón por la cual, los *sistemas distribuidos* constituyen un paradigma de computación que ha sido útil para satisfacer los requisitos demandados por las aplicaciones modernas.

En los últimos años, los sistemas distribuidos se han convertido en una plataforma de mucho interés para aplicaciones de propósito especial, *e.g.* aquellas que para su ejecución es necesario cumplir con ciertos parámetros de planificación y restricciones de tiempo. Este tipo de aplicaciones se presentan en el dominio de tiempo real y su utilización es cada vez más común en ambientes distribuidos.

La *programación orientada a objetos* es un paradigma muy prometedor para el dominio de tiempo real. El lenguaje *Java* es uno de los mayores exponentes de esta técnica de programación, sin embargo, no fue diseñado para el desarrollo de aplicaciones de tiempo real. Por tal razón, la comunidad de *Java* recientemente ha creado una especificación de tiempo real (*RTSJ*) para proveer una plataforma de desarrollo y de ejecución. Actualmente existen diversas implementaciones a partir de tal especificación, pero su diseño está dirigido hacia sistemas centralizados.

Por tanto, nuestro trabajo de tesis ha propuesto una herramienta de desarrollo así como una plataforma de ejecución para aplicaciones *Java* con requisitos temporales en un sistema distribuido, aprovechando la actual *RTSJ*. Como epílogo del presente documento, en esta sección exponemos los comentarios finales acerca del trabajo realizado, cubriendo los siguientes aspectos:

- **Conclusiones**, que sintetizan los resultados alcanzados, confrontándolos con los objetivos planteados para verificar su cumplimiento.
- **Contribuciones**, *i.e.* las aportaciones y los nuevos conocimientos.
- **Trabajo futuro**, donde se discuten las posibles mejoras del producto obtenido, nuevas capacidades y perspectivas de investigación en cuanto al tema abordado.

7.1 Conclusiones

1. El resultado principal del trabajo de tesis comprende:

   i) **Una arquitectura de clases *Java*** que sirve como patrón de diseño y herramienta para el desarrollo de aplicaciones distribuidas de tiempo real; y

   ii) **Una plataforma de ejecución**, integrada por diversas herramientas *Java* tales como:

   - ***JVM-RT***: ambiente de ejecución para *Java* de tiempo real, únicamente para una sola computadora.

   - ***API* de *sockets***: mecanismo que proporciona *Java* para la comunicación en red. Permite comunicar las *JVM-RTs* en diferentes computadoras.

   - ***Reflectividad* y *carga dinámica***: utilidades para la búsqueda y carga de clases en tiempo de ejecución, a través de *bytecodes*.

   - ***NTP***: el servicio de tiempo de *Internet* utilizado para sincronizar los relojes de las computadoras del sistema distribuido.

   - ***LiveGraph***: el software de visualización que permite supervisar la ejecución de la aplicación distribuida de tiempo real.

   Estas herramientas trabajan en conjunto sobre *Linux* y colaboran para permitir la ejecución de aplicaciones de tiempo real flexible en un sistema distribuido. Precisamente, **el mayor reto del proyecto fue acoplar dichas herramientas** para que alcancen la funcionalidad requerida. Cabe recordar que las herramientas *Java* están disponibles gratuitamente.

2. Las clases *Java* que conforman el esquema de solución en combinación con las herramientas descritas (*JVM-RT*, *API de sockets*, *NTP*, *LiveGraph*, *reflectividad* y *carga dinámica*), proveen un ambiente de desarrollo y de ejecución para aplicaciones de tiempo real flexibles sobre un sistema distribuido.

3. La estructura de clases *Java* de la solución asemeja la filosofía impuesta por el modelo de *Java RMI*, el cual ha demostrado ser exitoso para aplicaciones distribuidas convencionales y que, de manera general, establece los componentes *cliente*, *servidor*, e *interfaz remota*, para permitir la invocación de métodos remotos.

4. En nuestra solución, la técnica de **programación *multi-hilo*** ha sido clave para resolver los siguientes problemas:

   a) El soporte para la concurrencia del esquema *cliente-servidor*, con lo cual, un servidor puede atender varias conexiones de cliente en forma simultánea. De esta manera, es posible separar tareas, *e.g.* atender una conexión, esperar por otra, o procesar cada conexión. Esta separación de responsabilidades permite asegurar

que si un problema ocurre con una determinada conexión, ésta no afecte o interrumpa las demás.

b) Bajo el mismo concepto, es posible manejar la concurrencia inherente de las aplicaciones de tiempo real, donde varias actividades deben ser tratadas a la vez.

c) Gracias a esta técnica, hemos podido seguir el principio establecido por *Java RMI*, ya que la clase `InterfazRemota` hereda de `Thread`, con lo cual contiene un método `run()`, que representará al método `run()` de tiempo real en la clase `Aplicacion`.

5. La estrategia de solución está basada en la **propagación del contexto de ejecución** de los hilos de una aplicación entre los diversos nodos que conforman el sistema distribuido. El contexto de ejecución está constituido por parámetros de planificación y restricciones de tiempo que deben ser considerados por los nodos involucrados para mantener el cumplimiento de las restricciones temporales de las aplicaciones distribuidas.

6. Aunque la solución tiene una naturaleza *cliente-servidor*, no impide que un nodo del sistema distribuido pueda actuar como cliente y servidor al mismo tiempo. El esquema de clases desarrollado permite el intercambio de tales roles, algo que es requerido por las aplicaciones modernas.

7. Los resultados experimentales indican que nuestra implementación puede ser utilizada en s*istemas de tiempo real flexibles*, mismos que pueden ser implementados en una red de área local e incluso sobre *Internet*. Para *sistemas de tiempo real estrictos* en lenguaje *Java*, se realizan investigaciones con el fin de diseñar e implementar un *middleware Java* específico para tiempo real.

8. Hemos demostrado que la herramienta desarrollada es **genérica**, **flexible**, **versátil** y **adaptable**:

   a) Es genérica ya que se constituye como una base para la implementación de una serie de aplicaciones distribuidas de tiempo real flexibles en lenguaje *Java*.

   b) Es flexible puesto que no está sujeta al paso de parámetros fijos ni una estructura estricta de clases.

   c) Es versátil ya que puede realizar diversas funciones, como las que se mostraron en las aplicaciones de ejemplo. Mediante la *carga dinámica* de clases, el cliente puede proporcionar todos los elementos que el servidor necesita para realizar la ejecución.

   d) Es adaptable a planificadores particulares de una aplicación, siempre y cuando aproveche el esquema de prioridades de la *JVM-RT*.

7.2 Contribuciones

1. En síntesis, nuestra solución proporciona:

    a) Una ***plataforma de desarrollo***, *i.e.* un conjunto de clases *Java* (clases del *API* estándar, clases de la *RTSJ*, y clases propias) que define la estructura funcional sobre la cual una aplicación distribuida de tiempo real puede ser organizada e implementada.

    b) Un ***ambiente de ejecución***, conformado por varias herramientas *Java* disponibles gratuitamente que trabajan en conjunto sobre *Linux*.

2. La solución desarrollada permite su **re-utilización, tanto a nivel de diseño (arquitectura) como a nivel de implementación (código)**. De esta forma, un desarrollador no tiene que implementar una aplicación distribuida de tiempo real desde cero. Puede personalizar y extender nuestro esquema de clases de acuerdo con los requerimientos particulares, ahorrando tiempo y esfuerzo.

3. La capa de *middleware* en los sistemas distribuidos tradicionales, en nuestro caso, es sustituida por la *JVM-RT*, en combinación con la *API de sockets*, *reflectividad* y *carga dinámica*.

4. Con respecto a las herramientas utilizadas, destaca **el aprovechamiento de la *RTSJ* a pesar de que fue diseñada únicamente para sistemas centralizados** (un solo procesador). Aquí se centra la dificultad del proyecto.

5. Hemos realizado la evaluación de algunas de las implementaciones más populares de la *RTSJ* sobre *Linux*. Los resultados obtenidos demostraron que es viable utilizar la combinación de *TimeSys RI* / *Linux* como plataforma para aplicaciones de tiempo real flexible sobre redes de área local o incluso *Internet*. Utilizar un sistema operativo de propósito general con capacidades para tiempo real como *Linux* ofrece mayores facilidades que un sistema operativo de tiempo real específico.

6. Los parámetros de ordenamiento considerados en la solución son: la política de planificación y la prioridad; mientras que las restricciones de tiempo son: el plazo de ejecución y el período. Sin embargo, **la solución propuesta es extensible, *i.e.* permite la inclusión de parámetros adicionales** de acuerdo con las necesidades de una aplicación en particular.

7. Una actividad adicional, no considerada desde el inicio del proyecto, fue **agregar un supervisor de desempeño** que permita visualizar el comportamiento de una aplicación distribuida de tiempo real durante su ejecución. Para tal fin, hemos integrado y adaptado un software de visualización desarrollado en el ambiente académico que va actualizando el gráfico de los datos mientras son procesados por la aplicación.

8. Hemos descrito la forma en que **se puede adaptar un planificador particular de una aplicación**. Tal planificador trabaja con el esquema de prioridades provisto por la *JVM-*

RT y, mediante un intercambio dinámico de prioridad, entre máxima y mínima, es posible controlar la ejecución de todos los hilos de tiempo real que conforman la aplicación distribuida.

9. La posibilidad de incorporar un planificador personalizado mediante la carga de clases y el manejo de prioridades de los hilos de tiempo real en tiempo de ejecución, convierten a nuestra solución en un **sistema de planificación dinámica**. Adicionalmente, los resultados experimentales demostraron un mejor rendimiento del planificador permitiendo una menor pérdida de plazos.

9. Las pruebas realizadas permitieron establecer dos niveles de planificación de los hilos que conforman una aplicación distribuida de tiempo real:

    a) Nivel de la *JVM-RT*, donde podemos fijar valores de prioridad para cada uno de los hilos por medio de código *Java* de tiempo real;

    b) Nivel del sistema operativo, donde se realiza la planificación final del hilo.

    Una aplicación de tiempo real requiere que exista una correspondencia entre las prioridades de ambos niveles. De esta forma, el desarrollador puede controlar la ejecución de los hilos de la aplicación distribuida de tiempo real desde el ambiente de programación.

7.3 Trabajo futuro

1. Por el momento, la *JVM-RT* de *TimeSys* sólo proporciona soporte para *Linux*. Si en un futuro cercano aparece una versión compatible con otros sistemas operativos ó si otra *JVM-RT* resulta serlo, es probable que nuestra implementación pueda funcionar sobre plataformas operativas heterogéneas.

2. Puesto que la solución desarrollada se ejecuta sobre una plataforma *Java* basada en la *J2ME* (una versión optimizada del estándar), un proyecto interesante de emprender es extender la utilización de nuestra herramienta de desarrollo y ambiente de ejecución hacia dispositivos móviles y empotrados (*PDAs*, teléfonos celulares, reproductores multimedia, cámaras digitales, *etc.*).

3. La solución propuesta toma en cuenta determinados parámetros de planificación y restricciones de tiempo. Sin embargo, **existe la facilidad de agregar nuevos parámetros si una aplicación en particular lo demanda**, *e.g.* es posible incluir en la clase `Parametros` un atributo que represente el tiempo máximo para una invocación remota.

4. El principal trabajo de investigación relacionado con el tema abordado es el diseño y construcción de *middlewares* de tiempo real basados en su totalidad en la tecnología *Java*. Los esfuerzos de la comunidad de *Java* están encaminados en la creación de una versión de tiempo real de *Java RMI*; *i.e.* extender este modelo para considerar aspectos de tiempo real. En el presente trabajo presentamos una alternativa que utiliza la actual *RTSJ* pero

que no resuelve un factor de gran incertidumbre como la latencia de red.

5. Para limitar el tiempo de respuesta en redes de comunicación han surgido varias líneas de investigación, principalmente, a nivel de protocolos especiales y reservación de anchos de banda, dando como resultado las denominadas *redes de tiempo real*, basadas en protocolos como: *MIL-STD-1553B*, *FIP*, *Token-Bus*, *DCR-Ethernet*, *Profibus*, y *CAN*. Estas redes son de propósito especial y proveen una estructura capaz de exhibir comportamiento de tiempo real, mediante entregas de mensajes de acuerdo con prioridades y a pesar de fallos, líneas de conexión dedicadas, latencias de red constantes, y evitan que los nodos produzcan cargas que exceden la capacidad del medio de enlace [10].

6. Es posible realizar una investigación para conocer si la solución desarrollada puede complementarse con protocolos de comunicación como los mencionados anteriormente, permitiendo así, acotar los tiempos de entrega de los mensajes que intercambian las aplicaciones.

7. La solución expuesta es un paso inicial para la ejecución de aplicaciones *Java* distribuidas de tiempo real sobre ambientes de área local e *Internet*. Mejores resultados pueden ser obtenidos si las redes de área local son dedicadas, ***i.e.*** que funcionen específicamente para la aplicación distribuida de tiempo real.

8. Hemos implementado la carga dinámica de la clase correspondiente a un planificador personalizado. Si se logra que esta funcionalidad pueda ser generalizada para que cualquiera de los componentes de la aplicación distribuida de tiempo real, solicite o envíe los elementos necesarios (archivos de clase en *bytecodes*) para ejecutar alguna tarea específica entonces podemos catalogar a nuestra herramienta de desarrollo como un verdadero *framework* [39] para aplicaciones distribuidas de tiempo real flexibles.

9. De forma similar a lo que sucede con *Java RMI*, donde un compilador especial genera archivos de clase que representan a entidades remotas, es factible plantear el desarrollo de un componente que pueda ser añadido a nuestra herramienta y que permita la generación automática de clases con el fin de distribuir aplicaciones. El desarrollador simplemente definiría el código del método `run()` en la clase `Aplicacion` o realizaría la implementación explícita de la clase adicional que contiene el o los servicios. A partir de esta única implementación, un programa podría leer los caracteres del archivo de clase e identificar los elementos sintácticos necesarios, produciendo como salida los restantes archivos de clase del esquema de solución. Finalmente, el desarrollador sólo tendría que realizar ajustes mínimos, específicos de su aplicación particular.

10. Hemos realizado el trabajo experimental utilizando una computadora que actúa como cliente y otra computadora como servidor, ***i.e.*** existe una relación uno a uno. Los resultados nos indican que la herramienta desarrollada puede ser aprovechada como base para construir un mecanismo uno a varios (más de un servidor). Podemos citar el ejemplo de la comparación de matrices, donde partes de la matriz podrían ser distribuidas a varios servidores, y los resultados parciales serían recolectados por el cliente.

11. Nuestra solución contempla la ejecución simultánea de varios hilos en el servidor. Estos

hilos pertenecen a un mismo proceso y son ejecutados por un único procesador. Sin embargo, debemos considerar la existencia de computadoras que disponen de más de un procesador, las cuales son cada vez más populares y accesibles en el mercado. Por tanto, se puede estudiar la posibilidad de adaptar el servidor para que cada uno de los hilos pueda ser transformado rápida y eficientemente a su correspondiente proceso, con el propósito de que se puedan ejecutar en diferentes procesadores de una sola computadora.

REFERENCIAS

[1] Jacobson, Ivar. ***Object-Oriented Software Engineering : A Use Case Driven Approach***. Addison-Wesley. 1992.

[2] Coulouris, George; Dollimore, Jean; Kindberg, Tim. ***Sistemas Distribuidos: Conceptos y Diseño***. 3a ed. Madrid: Pearson Educación S.A. 2001.

[3] Farley, Jim. ***JAVA Distributed Computing***.1a ed. O'Reilly & Associates, Inc., USA. Enero, 1998.

[4] Java Community Process. ***JSR 1: Real-Time Specification for Java***. Java Specification Request (JSR). *http://jcp.org/en/jsr/detail?id=1*

[5] Java Community Process. ***JSR 50: Distributed Real-Time Specification***. Java Specification Request (JSR). *http://jcp.org/en/jsr/detail?id=50*

[6] Jensen, D.; Anderson, J. ***"Distributed Real-Time Specification for Java: A Status Report (digest)"***. ACM International Conference Proceeding Series. Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems. Vol. 177, pp. 3-9. París. Octubre 11-13, 2006.

[7] Wellings, A.; Clark, R.; Jensen, D.; Wells, D. ***"A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation"***. Object-Oriented Real-Time Distributed Computing, 2002. Proceedings. Fifth IEEE International Symposium. pp. 13-22. Washington, DC. 29 Abril–1 Mayo, 2002.

[8] Borg, Andrew and Wellings, Andy. ***"A Real-Time RMI Framework for the RTSJ"***. IEEE Computer Society. Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03). pp. 238 – 246. Julio 2-4, 2003.

[9] Attiya, Hagit and Welch, Jennifer. ***Distributed Computing: Fundamentals, Simulations and Advanced Topics***. 2a ed. John Wiley & Sons, Inc. New Jersey. 2004.

[10] Veríssimo, Paulo and Rodrigues, Luís. ***Distributed Systems for System Architects***. Kluwer Academic Publishers. 2001.

[11] Tanenbaum, Andrew and Van Steen Maarten. ***Distributed Systems: Principles and Paradigms***. Prentice-Hall, Inc. 2002.

[12] Puder, Arno; Römer Kay and Pilhofer, Frank. ***Distributed Systems Architecture: A Middleware Approach***. Elsevier, 2006.

[13] Kawsar, F.; Shaikot, S.; Saikat, S.; Razzaque, A.; Mottalib, M. ***"An Efficient Dynamic Scheduling Algorithm in Distributed System"***. Proceedings of the 5 th International Conference on Computer and Information Technology (ICCIT 2002). pp. 97-100. Dahka, Bangladesh. Diciembre, 2002.

[14] Grunder Holger and Geihs Kurt. "***Reuse and inheritance in distributed object systems"***. Trends in Distributed Systems CORBA and Beyond. Springer-Verlag. Lecture Notes in Computer Science, Vol. 1161, pp. 191-200. 1996. *http://citeseer.ist.psu.edu/grunder96reuse.html*

[15] Tanenbaum, Andrew. ***Sistemas Operativos Modernos***. Pearson Educación. 2a edición. 2003.

[16] Gomaa, Hassan. ***Designing Concurrent, Distributed, and Real-Time Applications with UML***. Addison-Wesley. Julio, 2000.

[17] Object Management Group (OMG). ***Common Object Request Broker Architecture (CORBA): Core Specification***. Versión 3.0.3. Marzo, 2004. *http://www.omg.org/cgi-bin/doc?formal/04-03-12*

[18] Sun Microsystems, Inc. ***Java Remote Method Invocation Architecture and Functional Specification.*** Versión 1.3.0. Diciembre, 1999. *ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf*

[19] Microsoft Corporation. ***The Component Object Model Specification.*** *http://www.microsoft.com/com/resources/specs.asp*

[20] Raj, Gopalan Suresh. "***A Detailed Comparison of CORBA, DCOM and Java/RMI"***. 2003. *http://my.execpc.com/~gopalan/misc/compare.html*

[21] Oaks, Scott and Wong, Henry. ***JAVA Threads***. 1a ed. O'Reilly & Associates, Inc., Sebastopol, CA. 1997.

[22] Sun Microsystems, Inc. ***Remote Method Invocation***. The Java Tutorials. *http://java.sun.com/developer/onlineTraining/rmi/RMI.html*

[23] Kopetz, Hermann. ***Real-Time Systems: Design Principles for Distributed Embedded Applications***. 1a ed. Kluwer Academic Publishers, Boston. 1997.

[24] Wellings, Andy. ***Concurrent and Real-Time Programming in Java***. 1a ed. John Wiley & Sons. Septiembre, 2004.

[25] Bollella, Greg (IBM) and Gosling, James (Sun Microsystems). ***"The Real-Time Specification for Java"***. IEEE Computer Society. Vol. 33, Issue 6, pp. 47-54. Junio, 2000.

[26] Corsaro, A. and Schmidt, D. ***"The design and performance of the jRate Real-Time Java implementation".*** In International Symposium on Distributed Objects and

Applications (DOA). Octubre, 2002.
http://citeseer.ist.psu.edu/corsaro02design.htm

[27] Nilsen, Kelvin. ***"Issues in the Design and Implementation of Real-Time Java"***. *Java* Developer's Journal, Julio, 1996. *http://citeseer.ist.psu.edu/nilsen96issues.html*

[28] Crespo, Alfons and Alonso, Alejandro. ***"Una panorámica de los sistemas de tiempo real"***. Revista Iberoamericana de Automática e Informática Industrial (RIAI), Vol. 3, Nº. 2. pp. 7-18. 2006.

[29] Kim, Taehyoun; Chang, Naehyuck; Kim, Namyun; Shin, Heonshik. ***"Scheduling Garbage Collector for Embedded Real-Time Systems"***. In Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers and Tools for Embedded Systems, pp. 55–64, Mayo, 1999. *http://citeseer.ist.psu.edu/523601.html*

[30] Concurrent and Real-Time Programming in Java.
http://www.cs.york.ac.uk/rts/CRTJbook.html

[31] Project jRate. ***jRate***.
http://jrate.sourceforge.net/

[32] TimeSys. ***Real-Time Specification for Java Reference Implementation (RTSJ RI)***.
http://www.timesys.com/java/

[33] Aicas. ***JamaicaVM from Aicas***. *http://www.aicas.com/jamaica.html*

[34] Corsaro, A. and Schmidt, D. ***"Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems"***. Real-Time and Embedded Technology and Applications Symposium. Proceedings. Eighth IEEE. Septiembre 24-27, 2002.

[35] M. Amersdorfer. ***"Jiotto – A Java Framework Implementing the Giotto Semantics".*** Master's thesis, Universit¨at Salzburg, 2004.

[36] Pereira, C. E.; Ataide, F. H.; Kunz, G. O.; Freitas, E. P.; Silva, E. T.; Carvalho, F. C. ***"Performance Evaluation of Java Architectures in Embedded Real-Time Systems"***. 10th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2005. Vol. 1, pp. 841-848. Septiembre 19-22, 2005.

[37] Real-Time Specification for *Java*.
http://www.rtsj.org/

[38] Booch, Grady; Rumbaugh, James; and Jacobson, Ivar. ***The Unified Modeling Language Reference Manual***. Addison-Wesley, Reading, Mass. 1999.

[39] Shin, Hyun-Jeong; Choi, Il-Woo; Kim, Soo-Dong; Rhew, Sung-Yul. ***"A Design of Object-Oriented Framework Repository"***. IEEE International Conference on Systems, Man, and Cybernetics. Vol. 3, San Diego, CA. Octubre 11-14, 1998.

[40] LiveGraph. ***Framework for Real-Time Data Visualization, Analysis and Logging***. *http://www.live-graph.org/*

[41] Harold, Elliote R. ***Java Network Programming.*** 3a ed. O'Reilly. Octubre, 2004.

[42] Eckel, Bruce. ***Piensa en Java***. 2a ed. Pearson Educación, S.A. Madrid, 2002.

[43] Deitel, Harvey M. and Deitel, Paul J. ***Cómo programar en Java***. 5a ed. Pearson Educación, México, 2004.

[44] Krishna, A.S.; Schmidt, D.C.; Klefstad, R. ***"Enhancing Real-Time CORBA via Real-Time Java Features"***. Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04), pp. 66-73, 2004.

[45] Bertot, Yves. ***"Filters on CoInductive Streams, an Application to Eratosthenes' Sieve"***. Springer-Verlag, Berlin Heidelberg, Vol. 3461, pp. 102–115, 2005.