UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERíA EN INFORMÁTICA

PROYECTO FIN DE CARRERA

DISEÑO E IMPLEMENTACIÓN DE UN SERVIDOR HTTP Y MECANISMOS DE SERIALIZACIÓN EN J2ME

Autor: Guillermo Diez-Andino Sancho (gdandino@it.uc3m.es)
Tutora: Celeste Campo Vázquez (celeste@it.uc3m.es)
Fecha: 23 de septiembre de 2002

Agradecimientos

Han pasado ya unos cuantos meses desde que comencé este proyecto. La verdad es que no hubiese sido lo mismo sin el apoyo y los ánimos infundidos por aquéllos que me rodean.

Lo cierto es que yo no estaría aquí si no hubiese sido por mis padres, por la oportunidad que me han dado de estudiar, por todo el apoyo que he tenido siempre y los ánimos que nunca me han faltado, a vosotros os lo debo todo, gracias Obdu y Conchi.

A mis hermanos, Carlos y Yolanda, por aguantarme en los buenos y en los malos momentos, por saber cuándo necesitaba vuestra ayuda y no haberme fallado nunca.

A Celeste, por su atención y preocupación en todo momento, su apoyo incondicional y su paciencia durante estos meses.

Finalmente a Andrés al que nunca le ha faltado la sonrisa cuando le interrogaba sobre HTTP y a Carlos, que como siempre, no ha dudado en echarme una mano con el incomprensible Latex, muchas gracias.

Resumen

La tecnología de agentes móviles ha tenido un especial impacto en los últimos años gracias a su aplicación en la computación distribuida, ya que ofrece grandes ventajas como son la movilidad, autonomía y comunicación.

Con el nacimiento en 1999 de la plataforma J2ME, destinada a la programación de aplicaciones en pequeños dispositivos con ciertas limitaciones de memoria y cómputo, se han planteado diversas propuestas en cuanto a la aplicabilidad de los agentes en este tipo de dispositivos se refiere.

Java en su versión J2SE posee ciertas características que lo hacen un buen lenguaje de programación para soportar la movilidad de código: mecanismos de serialización y carga dinámica de clases. En J2ME sin embargo ambas características se han eliminado, por lo que se hace necesario encontrar alguna forma para solventar estos problemas. En este proyecto se ha diseñado e implementado un servidor HTTP y un mecanismo de serialización sobre J2ME, que nos permitirá soportar movilidad de código entre dispositivos limitados.

De este modo se habrá dado un paso hacia la total integración de las plataformas de agentes móviles en la nueva plataforma de Java J2ME.

Abstract

During the last years, mobile agent's technology has entailed a special impact thanks to its applicability on distributed computing due to the fact that it offers great advantages such as mobility, autonomy and communication.

With the birth in 1999 of the J2ME platform, targeted to the programming of small devices with certain constraints in memory and computing power, several proposes have been put forward with regards to the agent's applicability on this kind of devices.

The J2SE edition possess some characteristics that turn it into a good programming language to support code mobility: a serialization mechanism and dynamic class download. On the other hand, these characteristics have been eliminated from J2ME, making necessary to find a way to avoid this lack. In this project an HTTP server and a serialization mechanism have been designed and implemented with J2ME. This will allow us to support code mobility between limited devices.

In this way, a step beyond will have been done to achieve the complete integration between the mobile agent's and the new J2ME platform.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación del proyecto

La tecnología de agentes ha tenido un especial impacto en los últimos años gracias a su aplicación en la computación distribuida. La ventaja que proporciona respecto a los modelos clásicos cliente/servidor es que permite realizar las mismas operaciones pero con la ventaja de que sean asíncronas y no precisen conexiones permanentes para la ejecución de tareas.

El concepto de agente ha sido muy discutido y aunque existen varias definiciones, la más aceptada es la que describe a los agentes como una entidad software que actúa en nombre de otra entidad, por ejemplo, de una persona o de otro agente. Que es autónomo y se comporta según los objetivos que debe alcanzar. Reacciona ante eventos externos y puede comunicarse y colaborar con otros agentes.

Un agente móvil es un agente que puede migrar entre dos nodos de una red. Junto a este tipo de agentes, surgieron también los llamados agentes inteligentes, basados en aplicar conceptos de inteligencia artificial al paradigma de agentes. Un agente móvil puede ser inteligente y a la inversa, pero ambas características son independientes.

Una plataforma de agentes se define como la infraestructura, tanto hardware como software que precisan los agentes para ser desarrollados y usados. Uno de los principales esfuerzos en los desarrollos de plataformas de agentes fue la definición de lenguajes que permitieran la movilidad de un agente entre plataformas. Aunque se han desarrollado lenguajes específicos, ha sido Java el lenguaje en el que más plataformas de agentes se han desarrollado, debido a las siguientes ventajas:

- Independencia de la plataforma
- Ejecución segura

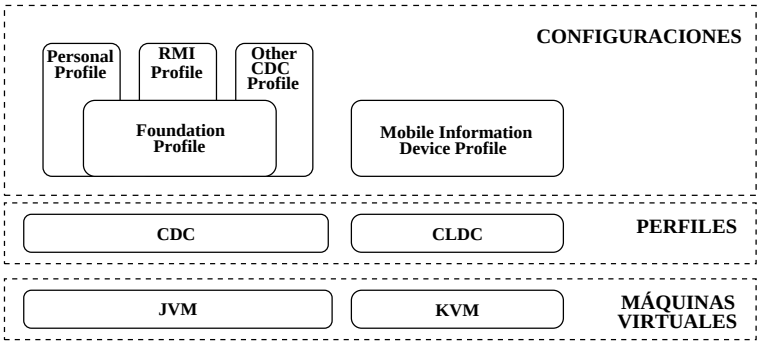

Figura 1.1: Arquitectura J2ME.

- Carga dinámica de clases
- Programación multithread
- Serialización de objetos
- Reflexión

En 1999 aparece Java 2 Micro Edition (J2ME) con el propósito de permitir que aplicaciones Java se ejecuten en dispositivos con capacidad de proceso limitada, como teléfonos móviles, pagers, palm pilots, set-top boxes, etc.

Una de las principales ventajas de J2ME es que es una arquitectura modular, figura 1.1, que se adapta a las limitaciones de los dispositivos en los que se quiere integrar. En concreto, para dispositivos limitados se emplea, como máquina virtual, la KVM (arquitecturas de 16/32 bits) y como configuración, la denominada Connected Limited Device Configuration (CLDC). Sobre esta configuración se ha estandarizado el perfil Mobile Information Device Profile (MIDP), que garantiza la portabilidad de aplicaciones entre los dispositivos que soporten este perfil, a estas aplicaciones que se ejecutan sobre MIDP se les denomina MIDlets, por analogía a los applets.

Debido a limitaciones de recursos y a la seguridad, como se verá en posteriores capítulos, gran parte de las características que convertían a Java en un buen lenguaje para desarrollar plataforma de agentes, han desaparecido en la versión J2ME, en concreto, aquéllas que nos permitían implementar movilidad de objetos (carga dinámica de clases, serialización y reflexión).

En este proyecto se plantea el reto de implementar las bases de una plataforma de agentes en dispositivos limitados CLDC/KVM, que permita a los agentes moverse entre estos dispositivos para realizar sus tareas, sin necesidad de involucrar a dispositivos no limitados. Esto es interesante por motivos de indisponibilidad de conexiones con la red fija o por motivos de coste.

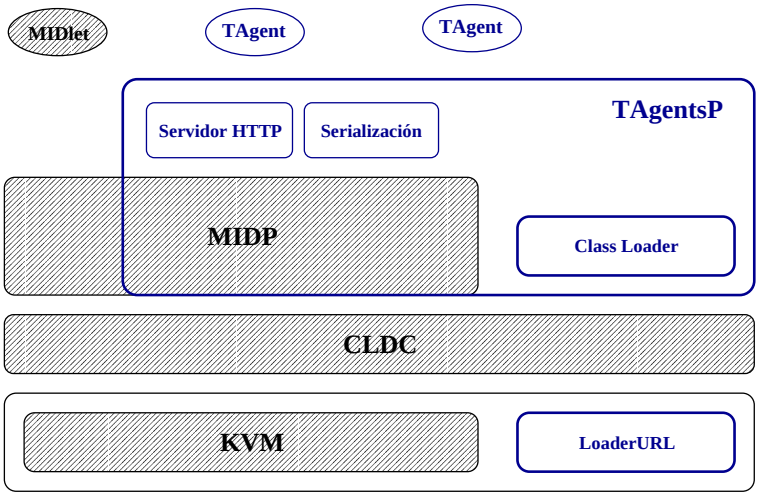

Figura 1.2: Arquitectura de nuestra propuesta.

Siguiendo la filosofía modular de J2ME, se pretende complementar el MIDP para construir un perfil sobre el CLDC que proporcione la funcionalidad básica de un plataforma de agentes móviles, a este perfil se le ha llamado TAgentsP (Travelling Agents Profile), por analogía a la plataforma de agentes Java, TAgents [15]. La funcionalidad básica se refiere a dotar a J2ME de aquéllas características que tenía Java como lenguaje de desarrollo de plataformas de agentes y que J2ME no posee, en concreto las que nos permiten movilidad:

- Carga dinámica de clases
- Serialización de objetos

1.2. Objetivos

La idea de este proyecto tiene su origen en la plataforma de agentes TAgents, mediante la cual se pueden gestionar diferentes agentes capaces de migrar a través de la red empleando para ello mecanismos tales como la reflexión, la serialización y la carga dinámica de clases. Partiendo de la plataforma TAgents, y con la finalidad de implementarla en dispositivos móviles, se detectó la ausencia de ciertas características (mencionadas en el apartado anterior).

Con el objetivo de dotar de la funcionalidad necesaria para la migración de la plataforma de agentes TAgents sobre dispositivos móviles MIDP se pretende construir:

- **Mecanismo de intercambio dinámico de clases** entre dispositivos móviles. Se hace necesario encontrar algún modo en el que los dispositivos puedan buscar así como cargar las clases que necesitan en tiempo de ejecución. A este requisito se le podría añadir la restricción de que sean los propios dispositivos los servidores "dinámicos" de clases.

  Al lograr esto sin la necesidad de un servidor intermedio dos dispositivos móviles se podrían comunicar de manera directa, solicitándose y transmitiéndose los recursos necesarios. Con este fin se plantea la construcción de un **servidor HTTP en J2ME**. Este servidor podrá albergarse en cualquier dispositivo que cumpla los requerimientos del MIDP y deberá implementar la funcionalidad mínima de un servidor HTTP versión 1.0.

- **Mecanismo de serialización** mediante el cual las clases puedan transformarse en un flujo de bytes para su posterior transmisión y reconstrucción en el dispositivo destino. La serialización es necesaria para que los agentes puedan migrar sin perder su estado.

  Debido a la ausencia del mecanismo de reflexión en J2ME, por el cual las propias clases son capaces de inspeccionarse a sí mismas, averiguando sus métodos, atributos, parámetros, constructores, etc ... la serialización a llevar a cabo va a tener ciertas limitaciones, no pudiéndose conseguir una serialización totalmente automatizada, en la que el programador no tenga que intervenir en este proceso.

  Se puede decir que lo que se busca conseguir es el soporte básico para que mediante éste y el conocimiento del programador sobre las clases que implementa se disponga de un mecanismo genérico y extensible para transformar y recuperar objetos, en un flujo de bytes y a un flujo de bytes, respectivamente.

1.3. Contenido de la memoria

El contenido de esta memoria se encuentra dividido en los siguientes capítulos:

- **Capítulo 1: Introducción**

  En este capítulo se presenta la motivación por la cual ha surgido este proyecto, cuál es su origen así como los objetivos planteados en el desarrollo del mismo. Finalmente se expone un breve apartado describiendo el contenido de esta memoria.

- **Capítulo 2: Estado del arte**

  En el capítulo Estado del Arte se hace un repaso sobre el estado actual de las tecnologías inalámbricas existentes y en particular la nueva plataforma de Sun, J2ME. Se explica detalladamente en qué consiste J2ME, las partes que la componen y cuáles son las características que la diferencian respecto a otras plataformas.

- **Capítulo 3: Servidor HTTP en J2ME**

  Descripción detallada del desarrollo del servidor HTTP en J2ME. Inicialmente se comenta la aquitectura de comunicación de J2ME, para seguidamente mencionar las soluciones planteadas así como la considerada más adecuada. Se incluyen los diagramas de clases y de secuencia más relevantes así como un ejemplo de funcionamiento del servidor.

- **Capítulo 4: Serialización en J2ME**

  En este capítulo se detalla el proceso de serialización seguido en J2SE así como los conceptos básicos necesarios para el entendimiento de este proceso. Seguidamente se expone detalladamente la solución de serialización planteada en J2ME. Finalmente se incluye un sencillo ejemplo del resultado obtenido con la puesta en marcha de esta implementación.

- **Capítulo 5: Pruebas**

  Exposición resumida de las diferentes pruebas llevadas a cabo tanto para el servidor HTTP como para la serialización. Se incluyen los comentarios más relevantes obtenidos mediante la realización de estas pruebas, así como conclusiones prácticas obtenidas de las mismas.

- **Capítulo 6: Historia del proyecto**

En este capítulo se hace una planificación temporal del proyecto, indicando el tiempo aproximado dedicado a cada una de las tareas. Así mismo se comentan los problemas encontrados y las diferentes soluciones planteadas.

- **Capítulo 7: Conclusiones y trabajos futuros**

  Se realiza un repaso sobre los objetivos propuestos y su grado de satisfacción. Finalmente se comentan las líneas futuras de trabajo deseables tanto para el servidor HTTP como para la serialización.

Finalmente se incluyen en la memoria una serie de anexos de apoyo descritos a continuación:

- **Anexo A: Atributos de un MIDlet**

  Tabla con los posibles atributos de un MIDlet clasificados por su tipo.

- **Anexo B: Variables de compilación y configuración del entorno MIDP**

  Tablas con las diferentes variables de compilación y configuración del entorno MIDP necesarias para la creación de un nuevo perfil.

- **Anexo C: Instalación y configuración del Wireless Toolkit**

  Breve manual sobre el proceso de instalación del Wireless Toolkit, entorno de desarrollo empleado en este proyecto.

- **Anexo D: Manual del programador**

  Información considerada de utilidad para el programador. Incluye los pasos necesarios para la creación de una aplicación J2ME así como un resumen del manejo de las clases implementadas en este proyecto.

- **Anexo E: Distribución**

  Descripción del contenido del CD-ROM adjuntado junto a esta memoria.

- **Anexo F: Glosario de términos**

  Glosario de los términos empleados a lo largo de toda esta memoria.

Para concluir, se incluye un apartado con toda la **bibliografía** utilizada a lo largo del desarrollo de este proyecto.

Capítulo 2

Estado del arte

A lo largo de los siguientes apartados se describe en qué consiste la plataforma J2ME, qué partes la componen así como cuáles son sus aspectos más destacables.

2.1. Qué es J2ME

Con el fin de solucionar la creciente demanda de aplicaciones en pequeños dispositivos, Sun ha extendido el ámbito de la tecnología Java con la introducción de la plataforma *Java 2, Micro Edition* (J2ME). Esta nueva plataforma está orientada a pequeños dispositivos tales como teléfonos móviles, PDAs o paginadores, que disponen de capacidades de memoria limitadas, pantallas reducidas y limitaciones en la capacidad de cálculo.

Una de los mayores beneficios de la tecnología Java en este tipo de dispositivos es la distribución dinámica y segura de diferentes aplicaciones a través de cualquier tipo de red. Por el contrario, antiguamente estos dispositivos venían con un software empotrado al cual era prácticamente imposible añadir nuevas funcionalidades.

Sun ha agrupado su tecnología Java en tres ediciones, cada una destinada a un área tecnológica diferente:

- *Java 2 Enterprise Edition (J2EE)* orientada a empresas que necesitan proporcionar servicios a clientes y proveedores a través de soluciones e-commerce y e-business.

- *Java 2 Standard Edition (J2SE)* para el ya bien establecido mercado familiar del ordenador de sobremesa (aplicaciones de usuario, applets, etc ...).

- *Java 2 Micro Edition (J2ME)* que combina las necesidades de los fabricantes de dispositivos embebidos, los proveedores de servicios que desean distribuir información a sus clientes a través de estos dispositivos y en tercer lugar, a los creadores de contenidos para que estén disponibles así mismo en estos dispositivos.

Figura 2.1: Tecnologías Java.

Cada una de estas ediciones difiere respecto al resto en la máquina virtual que poseen y en los APIs que utilizan, pero siempre tienen en común el lenguaje de programación que emplean, Java.

2.2. Otras tecnologías Inalámbricas

En este apartado se comentan algunas de las tecnologías y servicios inalámbricos con el fin de esclarecer lo que no es J2ME, entre ellas se hablará de las más conocidas.

- **WAP**

  Acrónimo de *Wireless Application Protocol*, es una tecnología introducida en el mercado en 1995 que permite a los dispositivos inalámbricos recibir datos de Internet y mostrarlos en las pantalla de los mismos. Se piensa en WAP como una tecnología que da soporte a buscadores Web en móviles, pero en realidad no es una aplicación sino un protocolo.

- **I-MODE**

  Introducido al mercado por la compañía Japonesa NTT DoCoMo, esta tecnología compite con WAP, ya que de igual modo ofrece un mecanismo de acceso a Internet a través de los dispositivos móviles. I-Mode dispone de un lenguaje de etiquetas similar a HTML denominado compact HTML (cHTML). La mayoría de sus usuarios se encuentran en Japón y otros países asiáticos.

La principal diferencia entre estas tecnologías es que J2ME permite el desarrollo de aplicaciones genéricas que podrán ejecutarse en un determinado tipo de dispositivos, mientras que WAP e I-MODE simplemente proporcionan acceso a Internet desde un dispositivo móvil.

2.3. Arquitectura de J2ME

Con el fin de conseguir una mayor flexibilidad y adaptación a diversos tipos de dispositivos, J2ME se estructura en tres niveles:

- **Máquina Virtual**

  Adaptada a dispositivos con capacidades limitadas, esta máquina está ligada a una configuración. En la actualidad existen dos tipos de máquinas virtuales en J2ME: la **CVM** (C Virtual Machine) y la **KVM** (Kilo Virtual Machine).

  La CVM está orientada a dispositivos embebidos y electrónica de consumo (dispositivos como TV digital, electrodomésticos, set-top-box, etc ...), ligada a la configuración CDC, requiere mayores recursos que su hermana pequeña, la KVM.

  La KVM tiene sus orígenes en Spotless (máquina virtual para el sistema operativo PalmOS). Diseñada desde cero para dispositivos con poca memoria, capacidad de proceso limitada, limitaciones de batería y conexiones a red intermitentes, esta máquina va unida a la configuración CLDC.

- **Configuración**

  Una configuración es el conjunto mínimo de clases disponible en una categoría de dispositivos. Las categorías se establecen según requisitos similares de memoria y procesamiento. Cabe mencionar que cada configuración está íntimamente relacionada a una máquina virtual, como se ha mencionado anteriormente.

  En la actualidad existen dos configuraciones en J2ME:

- **CDC** *Connected Device Configuration*

  Esta configuración está orientada a dispositivos con 512 KB de ROM, 256 KB de RAM, conexión a red fija, soporte completo de la especificación de la JVM e interfaz de usuario relativamente limitado.

  Iniciativas anteriores a esta configuración son: Personal Java, JavaTV y JavaPhone.

- **CLDC** *Connected, Limited Device Configuration*

  Los requisitos de esta configuración son sensiblemente inferiores a la CDC. La memoria requerida varía de 160 KB a 512 KB y los procesadores requeridos son de 16 o 32 bits. Otro requerimiento es la conectividad a algún tipo de red, a menudo inalámbrica, con conexión intermitente y ancho de banda limitado.

- **Perfil**

  El perfil es un conjunto de clases Java que complementan una configuración para un conjunto específico de dispositivos (segmento vertical).

  Los perfiles permiten la portabilidad de aplicaciones J2ME entre dispositivos diferentes.

  Actualmente existe una implementación sobre CLDC, la MIDP y una especificación, la del PDA Profile.

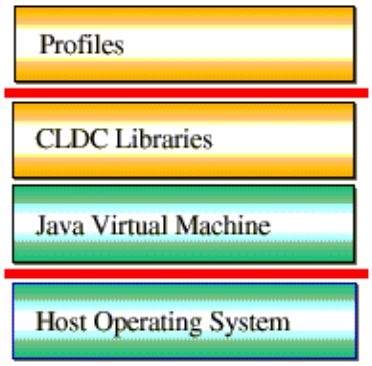

Figura 2.2: Arquitectura J2ME.

A continuación se explica más detalladamente la KVM, el CLDC y el MIDP.

2.3.1. Máquina Virtual K (KVM)

La K Virtual Machine es una máquina virtual Java altamente portable y compacta pensada y diseñada para funcionar en dispositivos con recursos limitados y con conexión a red intermitente.

El objetivo de la tecnología KVM es crear la JVM más pequeña y completa posible que mantenga los aspectos más importantes del lenguaje de programación Java y que funcione en dispositivos con procesadores de 16 o 32 bits y con unos pocos cientos de KiloBytes (originalmente 128 Kbytes de memoria).

2.3.2. Configuración CLDC

Este nivel es menos visible a los usuarios, pero es muy importante para los implementadores de perfiles. Define el conjunto mínimo de características de la máquina virtual y las librerías de clases disponibles en una determinada categoría de dispositivos. Representaría una franja horizontal de dispositivos con un mínimo denominador común, que los desarrolladores van a poder asumir que tienen todos los dispositivos con esa configuración.

Las configuraciones son especificadas a través de la iniciativa *Java Community Process* y sus implementaciones testeadas a través del test de compatibilidad TCK (*compatibility test kit*).

2.3.3. Perfil MIDP

Este nivel es el más visible a los usuarios y desarrolladores de aplicaciones. Las aplicaciones se desarrollan sobre un determinado perfil que a su vez está implementado sobre una determinada configuración.

El perfil define un conjunto de APIs y características comunes para una franja vertical de dispositivos. Las clases de un perfil permiten el acceso a funcionalidades específicas de los dispositivos como el interfaz gráfico, funcionalidades de red, almacenamiento persistente, etc ...

Las aplicaciones desarrolladas sobre un determinado perfil van a ser portables a cualquier dispositivo que soporte ese perfil. Cabe destacar que un dispositivo puede soportar varios perfiles y que sobre una configuración pueden existir diversos perfiles.

2.4. Requerimientos Hardware

A partir de ahora se va a considerar J2ME como la máquina virtual KVM, la configuración CLDC y el perfil MIDP.

Todos los dispositivos que deseen soportar J2ME deben poseer unas características hardware mínimas, ya que de otra manera sería imposible ejecutar la máquina virtual de J2ME y en concreto la **KVM**.

Estos requerimientos son los siguientes:

- **160 KB a 512 KB de memoria** disponible para la plataforma Java
- **Procesador** de 16 o 32 bits con velocidades a partir de 25 Mhz
- **Bajo consumo**, a menudo estos dispositivos funcionan con baterías
- **Conectividad** a algún tipo de red, con frecuencia mediante conexiones inalámbricas intermitentes y con limitaciones de ancho de banda (generalmente 9600 bps o menos)
- **128 Kb de memoria no volátil** disponible para la KVM y el CLDC.
- **32 Kb de memoria volátil** disponible para la ejecución de Java y la disposición de objetos en memoria

El perfil **MIDP** impone adicionalmente otros requisitos:

- **Pantalla**
  - **Tamaño** de 96x 54
  - **Profundidad de color** de 1 bit
- **Dispositivo de entrada** que deberá ser un teclado o bien una pantalla táctil
- **Memoria** de varios tipos
  - **128 KB de memoria no volátil** para los componentes del MIDP
  - **8 KB de memoria no volátil** para el almacenamiento persistente de las aplicaciones
  - **32 KB de memoria volátil** para la ejecución de Java (por ejemplo la pila de Java)
- **Conectividad** bidireccional, inalámbrica con ancho de banda limitado

2.5. J2ME vs J2SE

Java 2 Micro Edition ha sido creado para adaptarse a las características de los nuevos dispositivos inalámbricos tales como teléfonos móviles y PDAs. Consecuentemente existirán diferencias con la versión estándar de Java destinada a PCs. Algunas de las principales diferencias son las siguientes:

1. **Tipos de datos**

   J2ME soporta un subconjunto de los tipos de datos de J2SE. Los tipos **float** y **double** no están soportados por dos razones: la mayoría de dispositivos CLDC no disponen del hardware de unidad de coma flotante y en segundo lugar, es una operación muy costosa de llevar a cabo en modo software.

2. **Preverificación (on-line y off-line)**

   Al contrario que en J2SE en donde la verificación del código se realiza en tiempo de ejecución, en J2ME parte de la verificación se realiza off-line, es decir, fuera del dispositivo. Esto tiene la finalidad de reducir la carga de la máquina, llevando a cabo el resto de la verificación on-line.

3. **Descriptor y manifiesto**

   Al empaquetar clases en J2ME es necesaria la inclusión de unos archivos especiales que contienen información de las aplicaciones que incluyen, estos archivos de denominan manifiesto y descriptor

4. **Nueva librería gráfica**

   J2ME define un nuevo conjunto de clases para la creación de interfaces gráficas. Estas clases están adaptadas a dispositivos con memorias muy limitadas y pantallas de tamaño reducido. La librería gráfica AWT desaparece en esta nueva plataforma.

5. **Desaparición del main**

   Al contrario que las aplicaciones de la edición estándar de Java, en J2ME no existe una función main como punto de entrada para la ejecución del programa.

6. **Ausencia del recolector de basura**

   Con el objetivo de reducir la utilización de memoria, desaparece el recolector de basura en J2ME siendo necesario eliminar de forma explícita todos aquellos elementos que no vayan a ser utilizados más.

7. **Limitación en el tratamiento de errores**

   Aunque J2ME soporta el tratamiento de excepciones, el conjunto de excepciones incluidas en las librerías de la configuración se ha reducido sustancialmente. Esto se debe a dos razones principalmente:

   - En los dispositivos embebidos la recuperación de errores es algo generalmente específico de cada uno de ellos. Algunos dispositivos tratan de recuperarse de estas situaciones, mientras otros simplemente se resetean al detectar un error. El programador de aplicaciones no puede tener en cuenta este tipo de situaciones, ya que es característico de cada dispositivo en concreto.
   - La implementación de las capacidades de manejo de excepciones de acuerdo a la especificación del lenguaje Java es relativamente costoso, y supondría una carga significativa en la implementación dadas las estrictas limitaciones de memoria de los dispositivos CLDC.

8. **Otras características han sido eliminadas** de la máquina virtual Java que soporta el CLDC debido a la carencia de ciertas librerías o con el objetivo de mantener el modelo de seguridad establecido por el CLDC. Entre las características eliminadas cabe destacar:

   - **Interfaz Nativo Java (JNI)**
     El modelo de seguridad proporcionado por el CLDC asume que el conjunto de funciones nativas debe ser cerrado (Sandbox Model). Así mismo supone un incremento notable en los requerimientos hardware.
   - **Cargadores de clases definidos por el usuario**
     Al igual que con JNI el hecho de poder reconfigurar o crearse nuevos cargadores es parte de las restricciones del modelo de seguridad establecido.
   - **Reflexión**
     Con el objetivo de mantener la seguridad así como reducir el tamaño de la máquina virtual la reflexión no está soportada.
   - **Grupos de hilos e hilos demonio**
     Aunque el CLDC implementa los multihilos no existe soporte para grupo de hilos e hilos demonios. Las operaciones como arranque y parada de los threads se pueden aplicar de manera individual. En caso de que los desarrolladores de aplicaciones quieran utilizar estas características deberán implementarlas.

- **Finalización**
  Las librerías CLDC no incluyen el método *Object.finalize()* con lo que la finalización de instancias de clases no está soportada.

En conclusión se puede decir que estas características han sido eliminadas principalmente (sin tener en cuenta la carencia de soporte de coma flotante debido a la ausencia del soporte hardware necesario) a las estrictas limitaciones de memoria en primer lugar y a los problemas de seguridad que surgen debido a la ausencia de un modelo de seguridad completo como el de la edición J2SE.

2.6. MIDlet

Un MIDlet es la aplicación Java desarrollada de acuerdo a la especificación MIDP. Estas aplicaciones además de ser construidas en base a las restricciones de los APIs del CLDC y del MIDP deben pasar por un proceso de preverificación previamente a su distribución.

Al igual que los applets y otras aplicaciones Java, las clases de los MIDlets son almacenadas en ficheros de bytecode con la extensión *.class*.

2.7. MIDletSuite

Se conoce por MIDletSuite a la colección de MIDlets empaquetados conjuntamente en un archivo JAR.

Los MIDlets incluyen dentro del archivo jar los siguientes elementos:

- Las **clases Java** de cada uno de los MIDlets que incluyen
- **Clases compartidas**
- **Los archivos de recursos** utilizados por los MIDlets
- **Manifiesto** que describe los contenidos del archivo JAR y especifica los atributos utilizados por el software encargado de llevar a cabo la instalación de los MIDlets
- **Descriptor** de las aplicaciones (archivo JAD) que contiene un conjunto predefinido de atributos empleados para la recuperación e instalación de los MIDlets. Todos los atributos incluidos en el descriptor son accesibles desde los MIDlets. Cabe mencionar que el programador puede añadir sus propios atributos al MIDlet siempre que éstos no empiecen por MIDlet-.

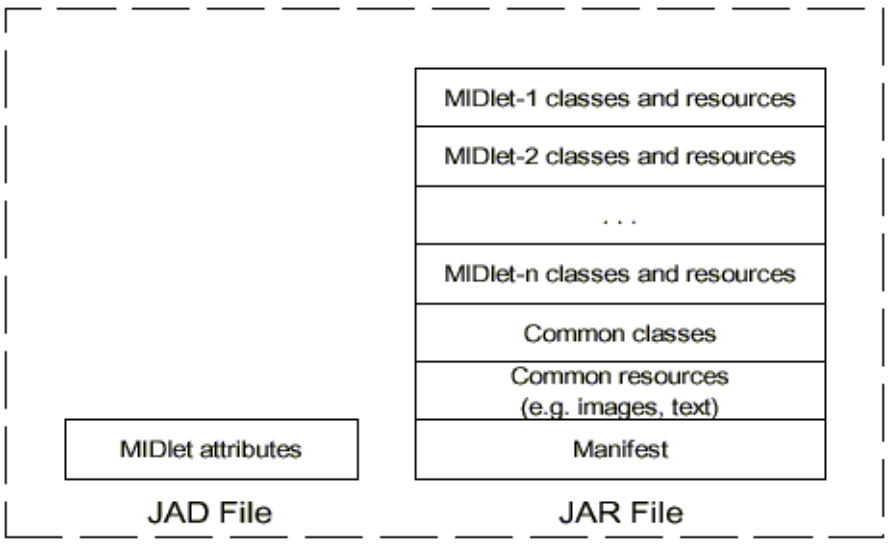

Figura 2.3: MIDletSuite.

La idea subyacente del MIDletSuite es la de permitir a todos los MIDlets que contenga compartir los recursos limitados de los dispositivos móviles, siendo inaccesible al resto. Para añadir un nuevo MIDlet sería necesario volver a generar el .JAR y descargarse de nuevo la aplicación.

2.8. Características especiales

A continuación se describen aquéllas características especiales que diferencian a J2ME respecto al resto de las plataformas Java.

2.8.1. Almacenamiento CLDC

El almacenamiento y recuperación de datos de forma persistente es una característica propia de prácticamente todos los lenguajes de programación. J2ME MIDP define una simple base de datos de registros denominada Record Management System (RMS) con el objetivo de poder almacenar información una vez que el MIDlet finalice.

La unidad de almacenamiento básica dentro de RMS es el record que es almacenado en una base de datos especial denominada record store. Un record, registro, como su nombre indica es un simple array de bytes, mientras que un record store es un fichero binario que contiene una colección de registros.

El API empleado para la creación y manejo de estos registros es descrito en el paquete *javax.microedition.rms*; este paquete define dos clases, tres interfaces y cinco excepciones que permitirán manejar de manera flexible este modo de almacenamiento.

2.8.2. Comunicación CLDC

El funcionamiento de red en J2ME tiene que ser muy flexible para soportar una gran variedad de dispositivos, para ello se introduce un nuevo concepto, denominado Marco Genérico de Conexión (MGC), consistente en abstraer el funcionamiento de red y el de los ficheros de entrada/salida.

La idea consiste en crear todas las conexiones utilizando un método estático de una clase del sistema llamada *Connector*. Si todo funciona correctamente este método devolverá un objeto que implementa uno de los interfaces genéricos de conexión. Existe una jerarquía de estos interfaces siendo el interfaz *Connector* su raíz.

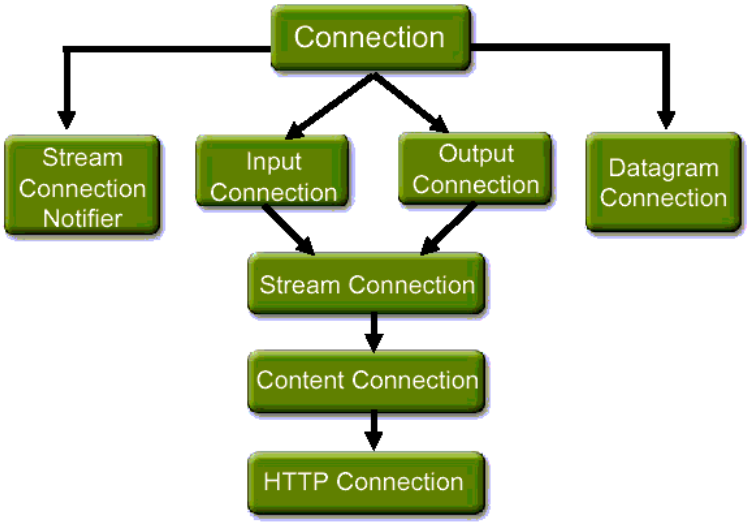

Figura 2.4: Jerarquía de clases del Marco Genérico de Conexión.

Cualquier llamada a este método tendría la siguiente forma:
Connector.open("<protocol>:<address>;<parameters>");

El objetivo principal de esta abstracción consiste en aislar en la medida de lo posible las diferencias que puedan existir entre las llamadas de cualquier protocolo que se pueda implementar en el perfil.

El MGC incluido en el CLDC no especifica los protocolos soportados ni incluye implementaciones de protocolos concretos (en realidad se incluye

alguna implementación a modo de ejemplo, aunque no forma parte de la especificación), si no que es labor del perfil las decisiones de diseño e implementación.

2.8.3. Internacionalización CLDC

Todos los dispositivos CLDC soportan por defecto la codificación IO-LATIN1. El CLDC incluye un soporte limitado para la conversión de caracteres Unicode a y desde una secuencia de bytes. Para ello se emplean los objetos denominados *Readers* y *Writers* a cuyos constructores se les pasará el tipo de codificación a utilizar[1].

Otro aspecto relevante en cuanto a la internacionalización se refiere es que el CLDC no proporciona ninguna característica de *localización*, con lo que las soluciones relativas a formatos de fechas, horas, monedas, etc ... está fuera del ámbito del CLDC.

2.8.4. Interfaz de usuario en MIDP

Con el fin de permitir flexibilidad y control sobre el interfaz de usuario, el perfil divide los APIs en dos niveles: de bajo y alto nivel.

El API de alto nivel está diseñado para aplicaciones en las que se desea un alto grado de portabilidad entre diferentes dispositivos. Para lograr esta portabilidad es necesario llegar a un compromiso entre la cantidad de dispositivos que pueden soportar este interfaz y el control que los programadores tienen sobre éste.

Por otro lado, el API de bajo nivel tiene un nivel de abstracción mucho más pequeño y requiere ciertas adaptaciones para lograr su portabilidad. Este API está diseñado para aplicaciones que necesitan control sobre los elementos gráficos concretos o eventos de bajo nivel como pueden ser pulsaciones de teclas. Mediante este API se posee acceso a funcionalidades específicas de los dispositivos. La estructura de clases se puede ver en la figura 2.5

A medida que se utilizan más funcionalidades y clases del API de bajo nivel la portabilidad va a ser más complicada, ya que con este API se accede a funcionalidades concretas que puede no tengan algunos dispositivos.

2.8.5. Seguridad

La seguridad es uno de los aspectos más cruciales de la plataforma J2ME ya que existe un compromiso entre seguridad/memoria. La cantidad de código

[1]Si no se especifica ninguna codificación se empleará la codificación por defecto descrita por la propiedad del sistema *microedition.encoding*.

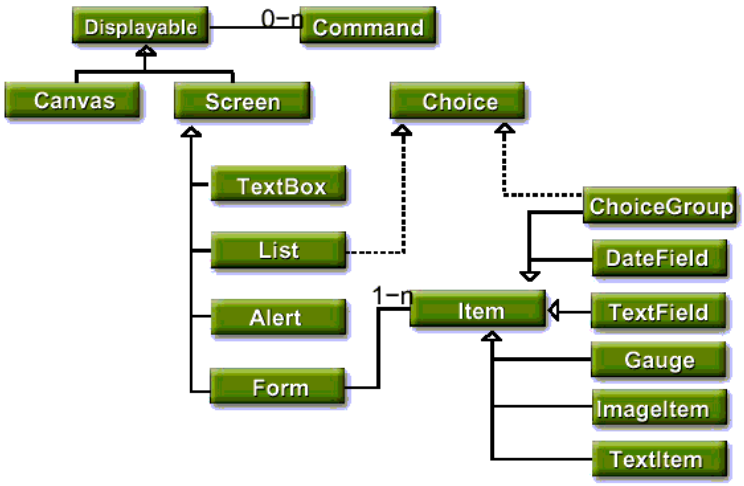

Figura 2.5: Jerarquía de clases de Interfaz de Usuario.

y más concretamente de memoria destinada a seguridad en la plataforma J2SE excede con creces la disponible en una máquina virtual para el CLDC.

La especificación CLDC, encargada de definir las pautas del modelo de seguridad disponible se centra en dos áreas principalmente:

- Seguridad a *bajo nivel* en la máquina virtual.
- Seguridad a *nivel de aplicación*.

La seguridad a bajo nivel de la máquina virtual es la encargada de asegurar que ninguna aplicación que se ejecute pueda "dañar" el dispositivo sobre el cual corre. Esto significa, por ejemplo, que no haya acceso a zonas de memoria prohibidas o fuera de la pila del programa.

En una implementación estándar de la máquina virtual esto se consigue mediante el verificador de clases, que asegurará que los bytecodes y otros elementos almacenados en los ficheros de clases Java no contengan este tipo de referencias no válidas.

Con el fin de reducir el consumo de memoria empleado por el verificador de J2SE (un mínimo de 50 KB para el código binario y de 30 a 100 KB de RAM en tiempo de ejecución) se ha optado por dividir el proceso de verificación en el CLDC en off-line y on-line, también conocidos como **pre-verificación** y **verificación**.

La pre-verificación se realiza generalmente en la máquina de desarrollo y analiza los ficheros de clases añadiéndoles un atributo especial conocido como

stack map attribute que facilitará la verificación en tiempo de ejecución. Cabe mencionar que el tamaño de los ficheros class se incrementa en un 5 por ciento aproximadamente.

El proceso de verificación tiene lugar en el propio dispositivo cuando se va a ejecutar la aplicación, el algoritmo de verificación realiza una serie de comprobaciones para garantizar completamente la seguridad y para ello emplea el atributo generado en el proceso de pre-verificación.

Esta nueva forma de verificación de las clases no sólamente optimiza el proceso de verificación en eficiencia, si no que reduce la cantidad de memoria necesaria en tiempo de ejecución a unos 100 KB de código binario y 100 bytes de memoria RAM.

En cualquier plataforma de desarrollo Java la verificación sólamente garantiza que la aplicación es válida por lo que continúan existiendo una serie de peligros potenciales que pasan desapercividos al proceso de verificación (acceso a sistemas de ficheros externos, impresoras, dispositivos de red, etc ...).

Para proporcionar control de acceso a estos recuros externos J2SE define el *security manager* que mediante una serie de políticas controlará los accesos a estos recursos. Al igual que ocurre con la seguridad a bajo nivel este proceso es demasiado costoso para realizarlo en el CLDC, por lo que se ha buscado una solución más simple conocida como *modelo sandbox*.

Un sandbox significa que las aplicaciones se protegen las unas de las otras ejecutándose en un entorno cerrado, más concretamente significa que:

- Los ficheros de clases java han sido verificados y se garantiza que son aplicaciones Java correctas.
- Solamente un conjunto limitado y predefinido de APIs está disponible para los programadores.
- No se permiten cargadores de clase definidos por los usuarios.
- El conjunto de funciones nativas accesibles por la máquina virtual es cerrado, es decir, el programador no puede descargar nuevas librerías que contengan funcionalidad nativa.

Puede que los perfiles definan soluciones adicionales de seguridad.

2.9. Ciclo de desarrollo de Aplicaciones

Las fases del ciclo de desarrollo para las aplicaciones J2ME[2] son las siguientes:

- **Edición** de los archivos fuentes empleando para ello cualquier editor de texto[3]
- **Compilación** de los ficheros fuentes. Para esto se utiliza el compilador de Java 2 Standard Edition. El compilador también se encargará de comprobar que la aplicación no está utilizando ningún paquete, clase o método que no forme parte de los APIs del CLDC o del MIDP.
- **Construcción**
  - **Verificación del Byte-code**
  - **Empaquetamiento** en un MIDlet Suite
- **Testeo en el emulador** ejecutando la aplicación en el dispositivo seleccionado.
- **Ejecución en el dispositivo final**

En el Wireless Toolkit[4] cabe la posibilidad de probar la aplicación con varios simuladores simultáneamente. El propósito de esto es comprobar el resultado y la apariencia de la ejecución de la aplicación en diferentes entornos.

2.10. AMS

Conocido como Application Management Software o JAM (Java Application Management) es la parte software del dispositio encargada de manejar y dirigir los MIDlet a través de sus diferentes estados, sirviendo de interfaz entre el sistema operativo y la máquina virtual del dispositivo.

El AMS asume que las aplicaciones se encuentran disponibles para ser descargadas en format JAR y que pueden obtenerse utilizando un protocolo de red (generalmente HTTP) o de almacenamiento definido por el MGC.

El AMS es el encargado de leer el contenido del fichero JAR y su descriptor asociado para a continuación lanzar la máquina virtual con la clase principal del MIDlet como parámetro.

[2]Se han incluido las fases soportadas por el Wireless Toolkit, kit de desarrollo empleado en la realización de este proyecto.

[3]El Wireless Toolkit no incluye el editor de texto.

[4]Entorno de desarrollo empleado en este proyecto, permite compilar, ejecutar y empaquetar las aplicaciones J2ME desarrolladas.

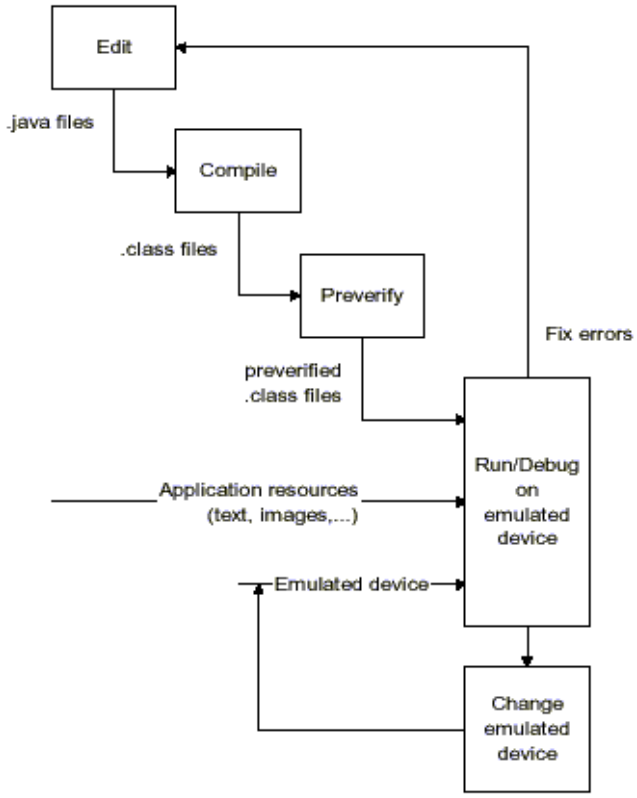

Figura 2.6: Ciclo de desarrollo de Aplicaciones.

2.11. Ciclo de vida de un programa J2ME

Un programa J2ME es una aplicación de MIDP que hereda de la clase MIDlet.

Un MIDlet tiene 3 estados que son activo, pasivo y destruido, éstos se corresponden con los siguientes métodos:

- *startApp()*
- *destroyApp()*
- *pauseApp()*

El administrador del entorno de ejecución de la aplicaciones es el que llama directamente a estos métodos, ya que un MIDlet no cuenta con el método 'main'.

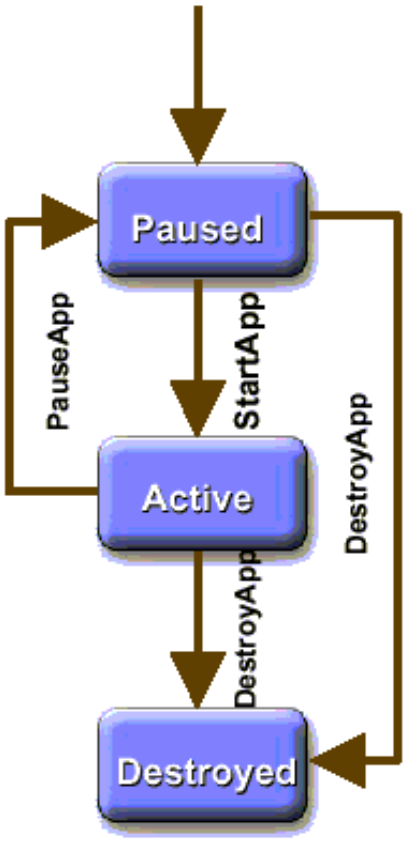

Figura 2.7: Ciclo de vida de un MIDlet.

El método de esta clase permite al gestor software de la aplicación crear, iniciar, pasar al estado de pausa y destruir el MIDlet.

Un MIDlet es el conjunto de clases diseñadas para ser ejecutadas y controladas por la aplicación vía la interfaz. Se pueden tener varios MIDlets a la vez, y seleccionar cual está activo en un tiempo concreto llamando al método *startApp()* y ponerlo en pausa con la llamada a otro método, el *pauseApp()*.

En el estado **pasivo** el MIDlet está inicializado y en resposo. En este estado no debería utilizar ningún recurso compartido. Al construir un MIDlet con el método **new** se realiza una llamada al constructor sin parámetros del MIDlet. En teoría no debe haber ningún problema, ya que pocas operaciones se suelen realizar en este estado. Si por alguna razón se lanza una excepción el MIDlet entraría inmediatamente en el estado *destroyed* y sería descartado.

Durante la ejecución normal del un MIDlet, éste se encuentra en el estado **activo**. Para pasar a este estado ha sido necesario realizar una llamada del

tipo *MIDlet.startApp()*.

Por último un MIDlet se encuentra en el estado **destruido** cuando su ejecución ha finalizado, se han liberado todos sus recursos. Cabe mencionar que a este estado sólo se puede entrar una vez.

Un ejemplo típico del proceso de ejecución de un MIDlet es el siguiente:

- El AMS crea una instancia del MIDlet. En el MIDlet se produce una llamada al constructor por defecto (sin argumentos). En estos instantes el MIDlet se encuentra en estado **pausa**.
- El AMS ha decidio ejecutar el MIDlet, con lo que llama al método *MIDlet.startApp*. El MIDlet entra en estado **activo**, adquiere los recursos que necesita y comienza a ejecutar su funcionalidad.
- Se produce algún cambio y el AMS no necesita que el MIDlet esté activo, le envía la señal para que pase a estado **pasivo** mediante la llamada *MIDlet.pauseApp*. Es elección del MIDlet liberar algunos de los recursos que mantiene reservados.
- El usuario cierra la aplicación, el MIDlet ya no es necesario con lo que el AMS le envía la señal de destruirse. Se realiza una llamada al método *MIDlet.destroyApp*. El MIDlet guarda el estado y se limpia de la memoria.

2.12. Instalación de un MIDlet

Una vez creado un MIDlet llega la fase de distribución. Existen diversas maneras de descargar las aplicaciones a los dispositivos, entre las cuales destacan la del *cable* que se conecta al PC y en segundo lugar, la descarga conocida como *OTA* (On The Air). Cualquiera de estas modalidades requiere unos pasos adicionales que se describen a continuación.

- **Búsqueda de la aplicación**

  Los dispositivos MIDP proporcionan mecanismos que permiten a los usuarios buscar o ser notificados de los MIDlets suites que pueden ser descargados en sus dispositivos.

  En algunos casos, la búsqueda de aplicaciones se realiza a través de un navegador residente en el dispositivo (ej. WAP). En otras ocasiones, son aplicaciones residentes escritas específicamente para identificar MIDlets suites que se pueda descargar el usuario. Esta funcionalidad se conoce como *Discovery Application o DA*.

Figura 2.8: OTA.

Cuando se utiliza el DA, se presenta un enlace al usuario que lleva al descriptor de la aplicación. El usuario selecciona el enlace, momento en el que comienza el proceso de instalación.

- **Descarga y verificación del descriptor**

  Una vez seleccionado el MIDlet se procede a la descarga del descriptor en el dispositivo. El AMS (Application Management Software) examinará el nombre, tamaño y versión de CLDC/MIDP para verificar que el entorno de ejecución sea el correcto.

  El servidor debe indicar que el tipo de datos que se transfiere (en este caso el descriptor) tiene el tipo MIME *text/vnd.sun.j2me.app-descriptor* y que la extensión del fichero a descargar es .jad.

  Con la finalidad de ayudar al usuario a seleccionar los MIDlets suites apropiados para su dispositivo, el DA debería ser capaz de informar al servidor sobre las capacidades del dispositivo.

- **Descarga y ejecución del MIDlet suite**

  Verificado el descriptor, el AMS comenzará a descargar el archivo JAR de la dirección especificada en el atributo MIDlet-Jar-URL. Una vez descargada la aplicación se contrastará el descriptor con la información del manifiesto, para que en caso de que los atributos correspondan el

MIDlet suite se cargue y esté listo para su ejecución, en caso contrario será rechazado.

Todo este proceso puede verse de manera resumida en la figura 2.9.

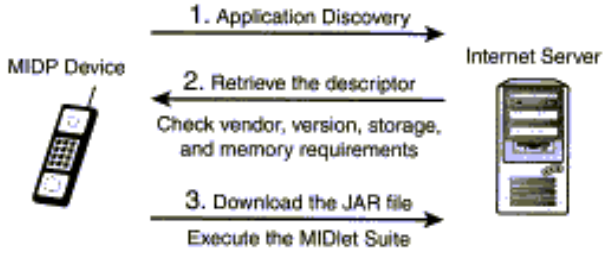

Figura 2.9: Instalación de un MIDlet.

2.12.1. OTA

A continuación se muestran las cabeceras de petición/respuesta intercambiadas entre un dispositivo que desea descargarse una aplicación vía OTA a través de un servidor que soporta este tipo de descargas.

En primer lugar el dispositivo solicita la descarga del descriptor de la aplicación, que como ya se ha mencionado anteriormente será verificado para asegurarse de la compatibilidad aplicación/dispositivo.

```
GET http://host.foo.bar/app-dir/game.jad HTTP/1.1
Host: host.foo.bar
Accept: text/vnd.sun.j2me.app-descriptor
User-Agent: CoolPhone/1.4 Profile/MIDP-1.0 Configuration/CLDC-1.0
Accept-Language: en-US, fi, fr
Accept-Charset: utf-8
```

El servidor emite su respuesta, establecerá una cookie para mantener el estado e informará del tamaño y características del archivo solicitado.

```
HTTP/1.1 200 OK
Server: CoolServer/1.3.12
Set-Cookie: Name=''abc''; Domain=''.foo.bar''; Path=''/app-dir''; \
JSESSIONID=''123''; VERSION=''1''
Content-Length: 2345
Content-Type: text/vnd.sun.j2me.app-descriptor; charset=utf-8
```

Código	Descripción
900	Éxito
901	Memoria insuficiente en el dispositivo
903	Pérdida del servicio
905	No concuerdan los atributos
906	Descriptor no válido
907	Jar no válido

Cuadro 2.1: Códigos HTTP OTA.

Una vez verificado el descriptor de la aplicación, y en el supuesto de que todo haya funcionado correctamente se solicita la descarga del jar del MIDlet suite.

```
GET http://host.foo.bar/app-dir/game.jar HTTP/1.1
Host: host.foo.bar
Cookie: Name=''abc''; Domain=''.foo.bar''; Path=''/app-dir''; \
JSESSIONID=''123''; VERSION=''1''
Accept: application/java, application/java-archive
```

La respuesta del servidor es la siguiente:

```
HTTP/1.1 200 OK
Server: CoolServer/1.3.12
Content-Length: 25432
Content-Type: application/java-archive
Cookie: Name=''abc''; Domain=''.foo.bar''; Path=''/app-dir''; \
JSESSIONID=''123''; VERSION=''1''
```

Con el fin de controlar los errores producidos por las diferentes situaciones que puedan surgir se han incluido una nueva serie de códigos HTTP y mensajes descriptivos del resultado de las operaciones OTA. Algunos de estos códigos pueden ser los mostrados en la tabla 2.12.1.

Capítulo 3

Servidor HTTP en J2ME

3.1. Introducción

El protocolo HTTP viene siendo ampliamente utilizado desde el nacimiento de la web, donde es indispensable para la creación de aplicaciones que hacen uso de ésta.

Este protocolo, básicamente, especifica el modo de comunicación entre un cliente y un servidor. El cliente solicita un documento, imagen o fichero del espacio de direcciones del servidor, y éste se lo sirve.

La arquitectura cliente/servidor se basa en el paso de mensajes de un dispositivo o máquina (cliente) que realiza las peticiones de servicio a otra en la que residen los datos y programas de aplicación (servidor).

Esta arquitectura apareció en los años 80, en contraposición a arquitecturas basadas en mainframes (*Mainframe architectures*) o en compartición de ficheros (*File sharing architecture*). Entre sus ventajas se encuentran la usabilidad, flexibilidad, interoperabilidad y escalabilidad.

Una de las aplicaciones más ampliamente utilizadas y que funciona sobre HTTP es la de un servidor Web. En esta aplicación existirá un cliente que mediante peticiones HTTP (GET, POST, HEAD) solicitará un determinado recurso o información sobre el mismo, al servidor HTTP.

Con la llegada de las tecnologías móviles pueden darse diversas situaciones en las que la creación de un servidor HTTP sea necesario.

La plataforma J2ME dispone del soporte necesario para la creación de todo tipo de clientes HTTP. Estos clientes HTTP están pensados en un principio para la utilización y comunicación con aplicaciones residentes en PCs, de modo que los dispositivos móviles puedan aprovechar todas los servicios ofrecidos por estas máquinas y tengan acceso a todo tipo de información almacenada en ellas (páginas Web, descarga de aplicaciones, etc ...).

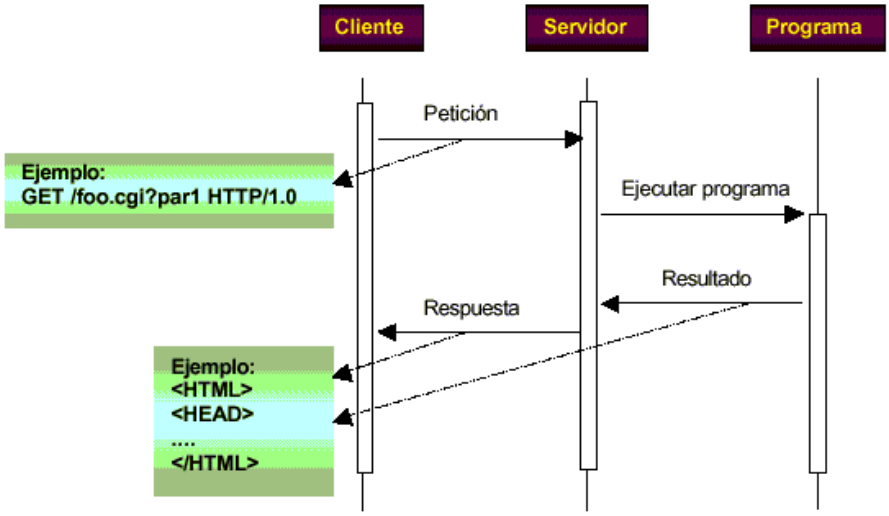

Figura 3.1: Secuencia de llamadas entre un cliente y un servidor HTTP.

Lo que se pretende conseguir es que estos dispositivos no sólo puedan utilizar estos servicios, si no que puedan ser también ellos mismos los que los proporcionan, dando acceso tanto a dispositivos móviles como a los que no lo son sin la necesidad de un servidor intermedio.

3.2. Objetivos

El principal objetivo a conseguir es construir un servidor HTTP sobre MIDP de manera que diferentes aplicaciones sobre MIDP puedan comunicarse directamente a través de éste. El servidor será el encargado de proporcionar los objetos solicitados por las diferentes aplicaciones de modo que no sea necesario un servidor intermedio.

Una posible utilización podría ser en la carga dinámica de clases. Supongamos que se dispone de un agente en un terminal móvil. Este agente va a migrar a otro dispositivo que no dispone de las clases necesarias para que se ejecute, gracias a un servidor de clases (en el propio dispositivo origen o bien en un servidor de clases), se podrían solicitar al servidor HTTP las clases que el dispositivo destino requiere, de manera que pueda ejecutar al agente que va a migrar.

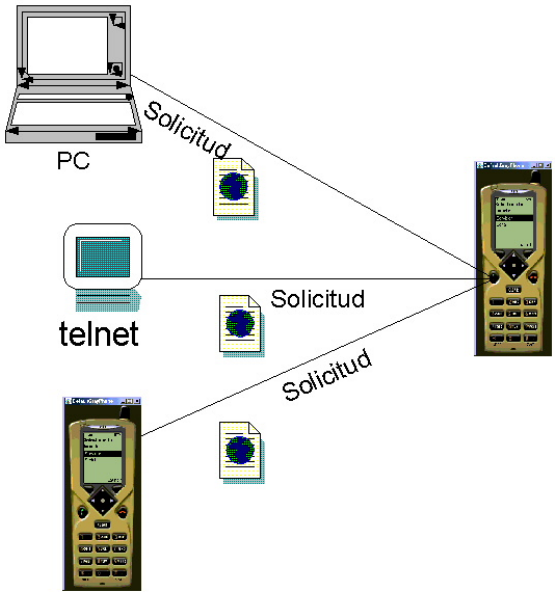

Figura 3.2: Diferentes clientes HTTP.

3.3. Marco de Conexión Genérico en CLDC

El funcionamiento de red en J2ME tiene que ser muy flexible para soportar una gran variedad de dispositivos, para ello se introduce un nuevo concepto, marco de conexión genérico, consistente en abstraer el funcionamiento de red y los ficheros de entrada/salida.

El Marco de Comunicación Genérica (MCG) proporciona las bases para todas las comunicaciones de red dentro de la arquitectura J2ME. Dentro del nivel de la configuración el MCG proporciona una serie de interfaces básicos que tendrán que ser implementados en un nivel superior.

Cabe mencionar que no se proporcionan implementaciones de protocolos específicos, si no que son los fabricantes de perfiles concretos los que deben proporcionar e implementar los interfaces.

El marco genérico de conexión reside en el paquete **javax.microedition.io** y consta de:

- **Una clase Connector (Connector)** encargada de crear cualquier tipo de conexión. Su método *open* devuelve un objeto que implementa uno de los interfaces genéricos de conexión. Existen un total de siete interfaces que forman la jerarquía del diagrama de clases del MGC, siendo el interfaz *Connection* el raíz de todos.

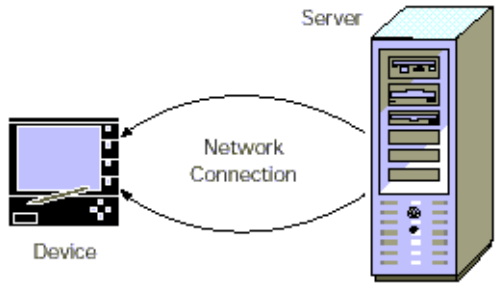

Figura 3.3: Comunicación tradicional de dispositivos móviles.

Figura 3.4: Nueva comunicación a través de dispositivos móviles.

- **Una excepción (ConnectionNotFoundException)** que indica que el objetivo de la conexión no se encuentra.

- **Siete interfaces** que deberán ser implementados por los desarrolladores de un perfil concreto.

Como se puede observar en la figura el interfaz *Connection* es el tipo más básico de conexión, este incluye el método *open* y *close*.

Los interfaces *InputConnection* y *OutputConnection* representan respectivamente dispositivos en los que escribir o leer. Estos interfaces se combinan en uno nuevo denomiando *StreamConnection*.

El interfaz *StreamConnectionNotifier* espera a que se establezca una conexión, para seguidamente devolver un *StreamConnection*. Por último decir que mediante *DatagramConnection* se definen las características que un datagrama puede tener.

Al igual que en el resto de los esquemas de direccionamiento de E/S del CLDC, la sintaxis del direccionamiento para los datagramas no se define en la especificación CLDC, si no que tiene lugar a nivel de perfil. La razón de esto es que estas clases se pueden utilizar para implementar diferentes protocolos

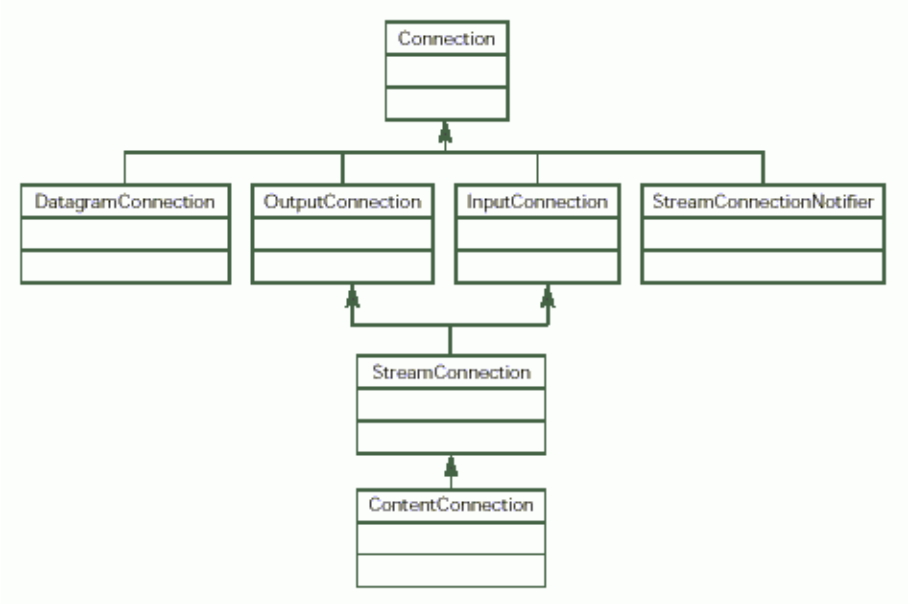

Figura 3.5: Jerarquía de Interfaces del MGC.

de datagrama. Por ejemplo redes IP y WDP, protocolos de infrarojos, etc ... y todos estos protocolos utilizan diferentes mecanismos de direccionamiento.

3.4. Conectividad en MIDP

La capacidad de movilidad y conectividad de los dispositivos móviles quedaría mermada si no se dispusiese de un mecanismo para poder acceder a datos remotos y redes corporativas, en resumen, Internet.

J2ME soporta las siguientes formas de comunicación, aunque cada fabricante sólamente implementa algunas de ellas:

- HTTP Connector.open("http://www.foo.com");

- Sockets Connector.open("socket://129.144.111.222:9000");

- Datagramas Connector.open("datagram://129.144.111.333");

- Puerto Serie Connector.open("comm:0;baudrate=9600");

- Fichero Connector.open("file:/foo.dat");

3.5. Utilización de sockets en MIDP

A continuación se describe la problemática existente a la hora de utilizar los sockets en el MIDP. Se plantearán varias soluciones destacando una que será la desarrollada en apartados posteriores.

3.5.1. Problemática

De todas las implementaciones MIDP existentes, MotoSDK es la única que soporta los tres protocolos de red (HTTP, UDP, TCP/IP), por el contrario el J2ME Wireless Toolkit de Sun sólo soporta comunicación en modo cliente HTTP, lo cual supone una barrera para la creación de un servidor HTTP.

Por defecto no se pueden utilizar los sockets en la implementación MIDP 1.0, aunque si bien están implementados a nivel CLDC, no se tiene acceso a ellos desde el perfil, únicamente se emplean a través de determinadas clases que hacen uso de ellos, aunque nunca directamente.

Debido a esto, es necesario encontrar una solución que pase por crear un nuevo perfil que sí tenga acceso a ellos o bien activar alguna característica de la configuración que sí nos proporcione el acceso deseado.

3.5.2. Soluciones planteadas

Entre las diversas soluciones planteadas a la hora de utilizar los sockets bajo la implementación de Sun se pueden destacar las siguientes:

- Existe una característica no documentada que permite ejecutar un programa que utilice UDP o un socket, estableciendo para ello la variable de entorno *com.sun.midp.io.enable_extra_protocols* a true. Esta variable se encuentra disponible en el fichero *internal.config* del directorio bin del Wireless Toolkit. Esto se debe a que los sockets se encuentran implementados a nivel CLDC, aunque no se tiene acceso directo a ellos desde el perfil MIDP.

  Activando esta variable en el Wireless Toolkit debería existir acceso a los sockets, tanto activos como pasivos. No obstante, esta solución es algo relativa, ya que en los entornos de ejecución en donde no se puedan realizar este tipo de modificaciones (por ejemplo la máquina virtual para Palm OS) la utilización de sockets seguiría sin estar disponible.

  Esta solución se puede utilizar de manera temporal y a efectos de desarrollo, ya que sería necesario buscar una nueva implementación del perfil MIDP que sí soporte los sockets.

- Una segunda solución a este problema consiste en ejecutar el MIDlet con el parámetro -D indicando que se desea tener accesibles los protocolos adicionales (en caso de que existan) a nivel de perfil. Al igual que con la solución anterior, va a existir la problemática de aquéllos entornos de ejecución en donde no sea posible pasar parámetros al ejecutar los MIDlets.

- Otra posibilidad consiste en engañar a la clase *Connection* de manera que crea que el socket es un protocolo válido. Esto se puede hacer creando una clase *Protocol.class* y situándola en *com/sun/midp/io/j2me/socket*, que es donde la clase Connector buscará los protocolos soportados.

  Esta clase *Protocol* debería heredar de *com.sun.cldc.io.j2me.socket.Protocol*.

  Relacionada con esta solución está la variable *javax.microedition.io.Connector.protocolpath* del fichero *system.config* con la que se puede especificar la ruta en dónde se encuentran las clases con los protocolos disponibles en el sistema.

- Por último existe la posibilidad de recompilar el perfil con las opciones de soporte de sockets activadas de manera que el nuevo perfil soporte la utilización de sockets. Este cambio habría que realizarlo para todas y cada una de las diferentes máquinas virtuales existentes (Palm OS, etc ...).

Cada uno de estas soluciones planteadas lleva asociada una complejidad diferente. De este modo se ha considerado la solución de activación de los sockets mediante la variable *com.sun.midp.io.enable_extra_protocols* la más sencilla y viable durante la fase de desarrollo.

Es de destacar que en la versión 2.0 del MIDP los sockets se van a encontrar disponibles por defecto, no siendo necesario llevar a cabo ningún tipo de modificación en el perfil o de configuración mediante parámetros.

3.6. Activación de sockets

Seguidamente se exponen los pasos a seguir para hacer que los sockets estén disponibles a nivel MIDP. Para ello se comenzará por las opciones de configuración para continuar con la compilación y su utilización.

3.6.1. Configuración del entorno MIDP

Es posible la personalización del MIDP, empleando para ello los ficheros de configuración *system.config* e *internal.config*. Cabe mencionar que en versiones anteriores al MIDP 1.0.3, la configuración se realizaba a través de variables de entorno, pero es a partir de esta distribución cuándo todos las opciones de configuración del perfil se van a realizar mediante estos dos ficheros.

Algunas de estas opciones de configuración permiten equilibrar el consumo de memoria frente a la funcionalidad disponible activando o desactivando determinadas características del sistema.

Los cambios realizados en la configuración pueden ser temporales o permanentes. Si el cambio se desea que no sea permanente habrá que utilizar la ejecución mediante la línea de comando con la opción -D<parametro>=<valor>. Si el cambio va a ser permanente habrá que modificar los ficheros *internal.config* y *sistem.config* ubicados en el directorio *build/share/lib*.

3.6.2. Opciones de Configuración

A continuación se muestran las principales opciones de configuración del entorno MIDP[1].

- **Variables internas**
  - microedition.configuration
  - microedition.profiles
  - microedition.locale
  - microedition.platform
- **Variables sistema**
  - system.jam_space
  - system.display.double_buffered
  - system.display.screen_depth
  - system.i18n.lang
  - system.i18n.encoding

[1]Se ha incluido un anexo con la descripción de cada una de estas variables, tanto internas como externas.

- system.display.slow_time_interval
- system.display.debug_screen
- system.display.visual_type
- system.throttle_per_milli
- com.sun.midp.io.enable_extra_protocols
- com.sun.midp.midlet.scheduler
- com.sun.midp.midletsuite.installer
- com.sun.midp.lcdui.eventHandler
- com.sun.midp.lcdui.skin
- com.sun.midp.lcdui.inputHandler
- com.sun.midp.io.http.proxy
- com.sun.midp.io.http.max_persistent_connections
- com.sun.midp.io.http.force_non_persistent

3.6.3. Compilación del MIDP

A continuación se describe el procedimiento a seguir para la creación de un MIDP personalizado.

1. Seleccionar las opciones de configuración del MIDP deseadas[2].
2. Seleccionar las opciones de configuración del entorno de generación del MIDP. Las opciones a elegir son las siguientes:
   - ALT_BOOTDIR
   - KVM_DIR
   - ROMIZING
   - DEBUG
   - ENABLE_DEBUGGER
   - INCLUDE_ALL_CLASSES
   - INCLUDEDEBUGCODE
   - ENABLEPROFILING

[2]Es dentro de los archivos system.config e internal.config en dónde se va a especificar el comportamiento deseado.

- INCLUDE_I18N
- INCLUDE_HTTPS

3. Ejecución del script de compilación del MIDP para un entorno en concreto (Linux, Win32, Solaris, ...). Para esto hay que situarse en el directorio *build* de la plataforma deseada y ejecutar el programa *gnumake* con la opción *all* (para el caso de que se quiera reconstruir el MIDP completo) o con cualquiera del resto de opciones disponibles para llevar a cabo otras acciones (creación de un paquete con ejemplos de aplicaciones, generación de documentación, etc ...).

De manera opcional cabe la posibilidad de añadir nuevas clases al perfil. Las clases que se deseen añadir pueden ser de dos tipos atendiendo a si son *genéricas* para cualquier plataforma o si son *concretas* para una determinada plataforma.

En función de esta característica, hay que modificar previamente a la creación del perfil una de las dos siguientes variables que se encuentran en el directorio **build/share/makefiles/Defs.gmk**. Si las clases son para una plataforma concreta habrá que añadirlas en la variable PLATFORM_INCLUDE_CLASSES, de lo contrario bastará con modificar la variable MIDP_INCLUDE_CLASSES.

Esto se puede apreciar con el siguiente ejemplo:

```
PLATFORM_INCLUDE_CLASSES += classes/com/MyCompany/productY/foo.java
MIDP_INCLUDE_CLASSES += classes/com/MyCompany/midp/bar.java
```

En el primer caso se está añadiendo la clase concreta *foo* a una determinada plataforma, sin embargo, en el segundo caso se desea que la clase *bar* esté disponible para todas las plataformas.

3.6.4. Utilización de sockets

Previamente a la utilización de los sockets es necesario comprobar si están disponibles en una determinada implementación del perfil. Para ello basta utilizar la función *Configuration.getProperty(com.sun.midp.io.enable_extra_protocols)*, que en caso de soportar protocolos adicionales devolverá el valor true. Si la variable no está establecida (es decir, no se encuentra inicializada ni a true ni a false) el valor devuelto será null.

Una vez comprobada la disponibilidad de los sockets bastará realizar una llamada *Connector.open(uri)* empleando como protocolo el *socket*

o *serversocket*. Un ejemplo sería la siguiente llamada que crea un socket pasivo en el puerto 80 de la máquina en la que se ejecuta *Connector.open(serversocket://:80)*.

Como parámetros adicionales a especificar en el método open, están el modo de acceso (READ, WRITE,READ_WRITE) así como si se desean excepciones de tipo TimeOut.

3.7. Creación de un servidor HTTP en J2ME

Los siguientes apartados contienen los pasos seguidos a la hora de implementar un servidor HTTP sobre MIDP. Se partirá desde los requisitos establecidos hasta finalmente mostrar un sencillo ejemplo de funcionamiento.

3.7.1. Introducción

Una vez superada la problemática de utilización de sockets se plantea la construcción de un servidor HTTP. Este servidor deberá soportar las características básicas del protocolo HTTP 1.0 (método GET, HEAD y OPTIONS), permitiendo varias conexiones simultáneas e informando con los códigos de error producidos en las diferentes situaciones que se puedan producir.

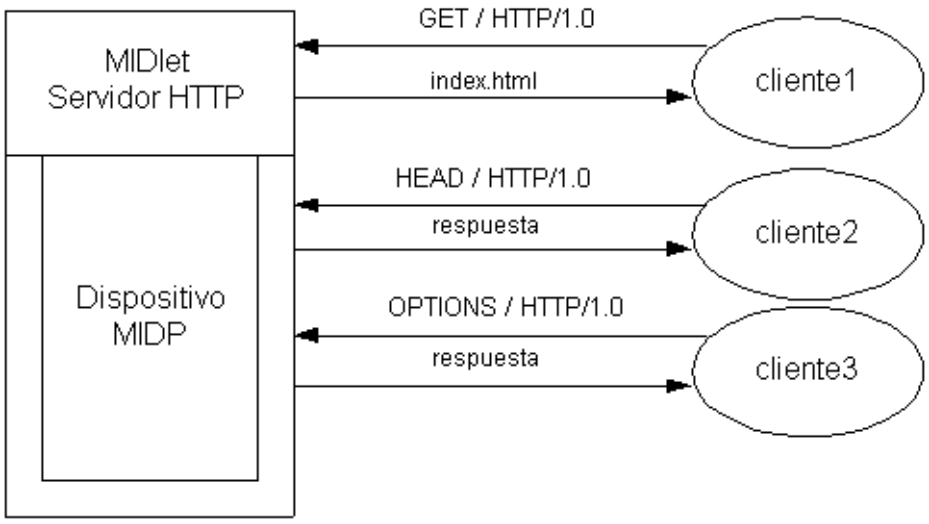

Figura 3.6: Servidor HTTP sobre dispositivo MIDP.

Debido a las limitaciones de espacio y capacidad de procesamiento de los dispositivos MIDP se pretende que el servidor construido requiera una capacidad reducida de memoria tanto para su ejecución como para su almacenamiento.

3.7.2. Requisitos establecidos

Los requisitos planteados inicialmente son los siguientes:

1. **Creación de la clase ServerSocketConnection** de acuerdo a la próxima implementación del perfil MIDP 2.0. A través de esta clase se podrán crear futuras aplicaciones que hagan uso de sockets pasivos.

2. **Construcción de un servidor HTTP** que permita el intercambio de objetos entre dispositivos móviles sin la necesidad de un servidor intermedio.

3. **Soporte de los métodos GET, HEAD y OPTIONS** con posibilidad de futuras ampliaciones.

4. **Sistema de archivos** mediante el cual el servidor pueda recuperar los recursos solicitados. Este sistema de archivos proporcionará las facilidades básicas de almacenamiento y recuperación de un sistema de archivos básico.

5. **Sistema de permisos** de modo que no todos los recursos del sistema estén disponibles a través de HTTP.

6. **Utilización mínima de recursos** tratando de reducir al máximo el tamaño del servidor, así como la memoria utilizada.

7. **Flexibilidad** a la hora de crear futuros servicios sobre MIDP. Con esto se entiende que las clases e interfaces implementados puedan ser susceptibles de reutilizarse en otras aplicaciones.

8. **Utilización de patrones de diseño** minimizando en todo lo posible la complejidad de futuros cambios así como facilitando la flexibilidad planteada anteriormente.

3.7.3. Arquitectura del servidor

El servidor está compuesto por tres módulos principalmente. El primero de ellos es la configuración mediante la cual se definen las opciones básicas de

funcionamiento. A través del sistema de archivos (segundo módulo) se van a poder gestionar los recursos a albergar por el servidor así como resolver las peticiones de los usuarios. Por último es mediante el servicio de tratamiento de peticiones que las diversas solicitudes HTTP podrán ser tratadas.

La arquitectura se puede ver de manera esquemática en la figura 3.7.

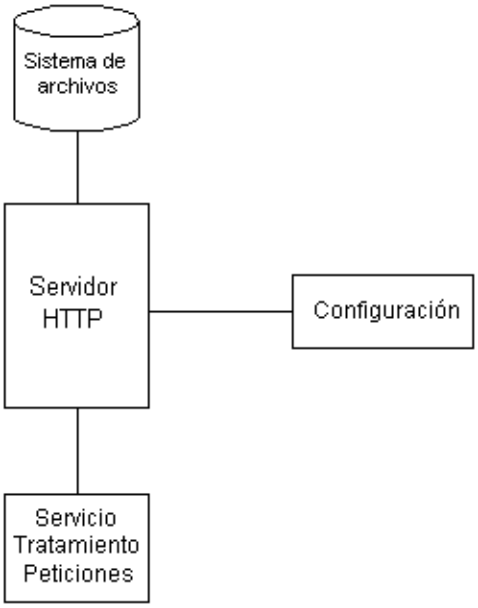

Figura 3.7: Arquitectura del Servidor HTTP sobre J2ME.

El servidor ha sido diseñado de modo que se minimice la complejidad a la hora de añadir futuras ampliaciones y mejoras. La utilización de patrones de diseño ha sido un factor determinante a la hora de lograrlo, ya que simplifican en extremo los cambios a realizar en el código a la hora de modificarlo.

3.7.4. Sistema de archivos

J2ME MIDP define una simple base de datos de registros denominada Record Management System (RMS) con el objetivo de poder almacenar información una vez que el MIDlet finalice.

La unidad de almacenamiento básica dentro de RMS es el *Record* que es almacenado en una base de datos especial denominada *Record Store*. Un *Record* (registro) como su nombre indica es un simple array de bytes, mientras que un *Record Store* es un fichero binario que contiene una colección de registros.

La implementación MIDP proporciona diversas funcionalidades para el manejo de estos registros, aunque no incluye ningún soporte para el manejo de ficheros en su modo tradicional.

Para poder proporcionar los recursos solicitados por los clientes HTTP ha sido necesario crear una pseudo sistema de archivos sobre RMS. Este sistema de archivos se ha implementado siguiendo los patrones **Strategy**[3] y el patrón **Singleton**[4].

Mediante la implementación realizada se podría cambiar el sistema de ficheros de RMS a cualquier tipo soportado por el dispositivo (XML, etc ...) simplemente realizando una nueva clase que implemente su funcionalidad concreta.

El problema inicialmente reside en que no existe un acceso directo a ficheros sobre MIDP, si a esto se añade que entre los clientes que posiblemente utilicen un servidor de este tipo se encuentran los PCs, que manejan un sistema de rutas jerárquico, se ha necesitado **mapear** el formato de rutas manejado por los navegadores a un nuevo formato establecido sobre RMS.

Las rutas de un sistema de archivos cualquiera forma una estructura jerárquica de varios niveles de anidamiento. Al carecer de esta característica en MIDP se ha considerado que cualquier recurso solicitado va a corresponder a un **record**, siendo la estructura jerárquica en la cual se encuentra este recurso un **RecordStore**. Esto se puede ilustrar con la figura 3.8.

Figura 3.8: Ejemplo de mapeado de archivos.

Si el recurso solicitado es */docs/index.html*, el documento *index.html* corresponderá a un record almacenado dentro del *RecordStore* denominado */docs/*. Cualquier petición del tipo */docs/documento* se buscará dentro de */docs/*.

[3]Patrón que proporciona gran flexibilidad a la hora de implementar diferentes estrategias, en este caso políticas de almacenado y recuperación de datos.

[4]Garantiza la existencia de una única instancia de una determinada clase en el sistema, esto puede ser útil por ejemplo, para asegurar que existe un único fichero de configuración.

A partir de ahora se va a seguir la política de denominar a los registros de RMS ficheros y carpeta para referirnos a los RecordSet.

Se ha considerado que peticiones del tipo */documento* equivalen a solicitar un recurso que se encuentra en la carpeta *"/docs/"*, es decir, la "carpeta"por defecto es *"/docs/"*.

Inicialmente existen tres carpetas dentro del servidor, la primera de estas corresponde a la de los documentos HTML a servir, siendo su nombre "/docs/". En esta carpeta se almacenan todas las páginas con formato HTML. En segundo lugar está "/images/" donde se encuentran las imagenes que puedan incluir los documentos de la carpeta "/docs/" o cualquier otra imagen. Por último está "/objects/" que tendrá la finalidad de almacenar objetos serializados.

Cada archivo está compuesto por tres atributos (nombre, longitud y permiso) y un campo de bytes que corresponde al documento en sí. La longitud máxima de un registro es de 65000 bytes y está definida por la implementación.

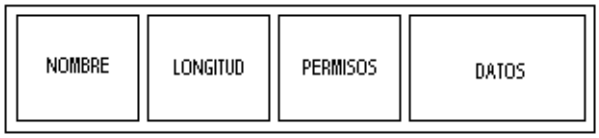

Figura 3.9: Atributos de un archivo.

El atributo **nombre** sirve para identificar cada fichero de manera única, de modo que no puedan existir dos con el mismo nombre dentro de una misma carpeta. Aunque el atributo **longitud** se puede calcular a partir del tamaño de un registro de RMS, se ha optado por incluirlo para facilitar la recuperación de los archivos. Por último el atributo **permisos** sirve para que no todos los ficheros incluidos en el servidor puedan ser solicitados por clientes HTTP.

Esto último se debe a que en RMS no existe un control de acceso establecido con el que proteger determinados registros, con lo que de no existir el atributo **permisos**, cualquier documento existente dentro del servidor podría ser solicitado. El permiso puede ser *público* si el documento es accesible mediante HTTP o *privado* en caso contrario.

Cabe mencionar que al establecer los permisos a nivel de fichero, no existiendo ningún control sobre las carpetas, puede haber documentos tanto públicos como privados dentro de una misma carpeta.

Figura 3.10: Sistema de Archivos sobre RMS.

Respecto a la granularidad de los permisos, puede seguirse cualquier política de acceso simplemente tratando el atributo permiso del fichero. Se podrían crear grupos de usuarios, propietarios del archivo, etc ... Inicialmente sólo se han establecido permiso "público" y "privado", aunque en futuras revisiones cabría ampliar el rango de permisos existentes.

3.7.5. Diagrama de clases

En la figura 3.11 se muestra el diagrama de casos de uso del servidor HTTP implementado. En este diagrama se muestran las diferentes operaciones que puede llevar a cabo un cliente HTTP.

Las clases y los interfaces existentes se muestran en el diagrama de clases de la figura 3.12.

3.7.6. Interfaces

Los interfaces existentes son los siguientes:

- **ServerI**

  Define la interfaz básica de cualquier servidor.

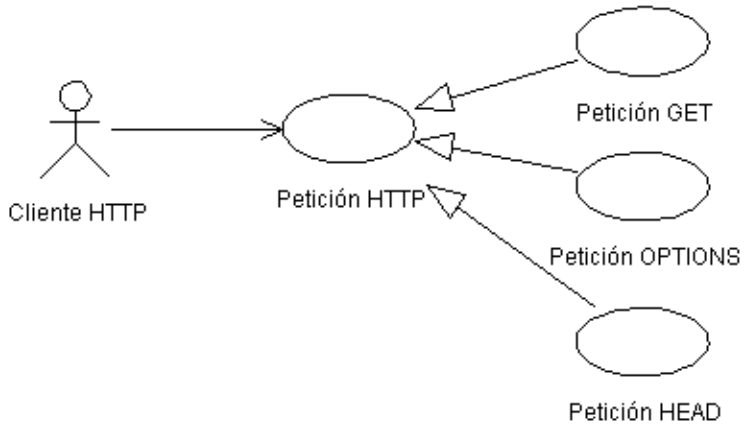

Figura 3.11: Casos de uso servidor HTTP.

- **HTTPServerI**

  Contiene las variables con los tipos de respuesta HTTP que va a poder devolver el servidor.

- **Request**

  Define los métodos comunes a cualquier tipo de petición.

- **Response**

  Define los métodos comunes a cualquier tipo de respuesta.

3.7.7. Clases

A continuación se describen brevemente las clases empleadas en el servidor HTTP J2ME:

- **ServiceHTTP**

  Es la clase encargada de realizar todo el procesamiento de las peticiones de los clientes. Cada vez que se recibe una conexión, se crea un objeto de esta clase que se lanza como un thread independiente, finalizando una vez atendida la petición.

  El número de servicios HTTP que se pueden lanzar va a estar limitado por la variable *maxProcessors* del fichero de configuración así como la variable interna *com.sun.midp.io.http.max_persistent_connections*[5]

[5]La variable interna del sistema es la que controla realmente el número de conexiones

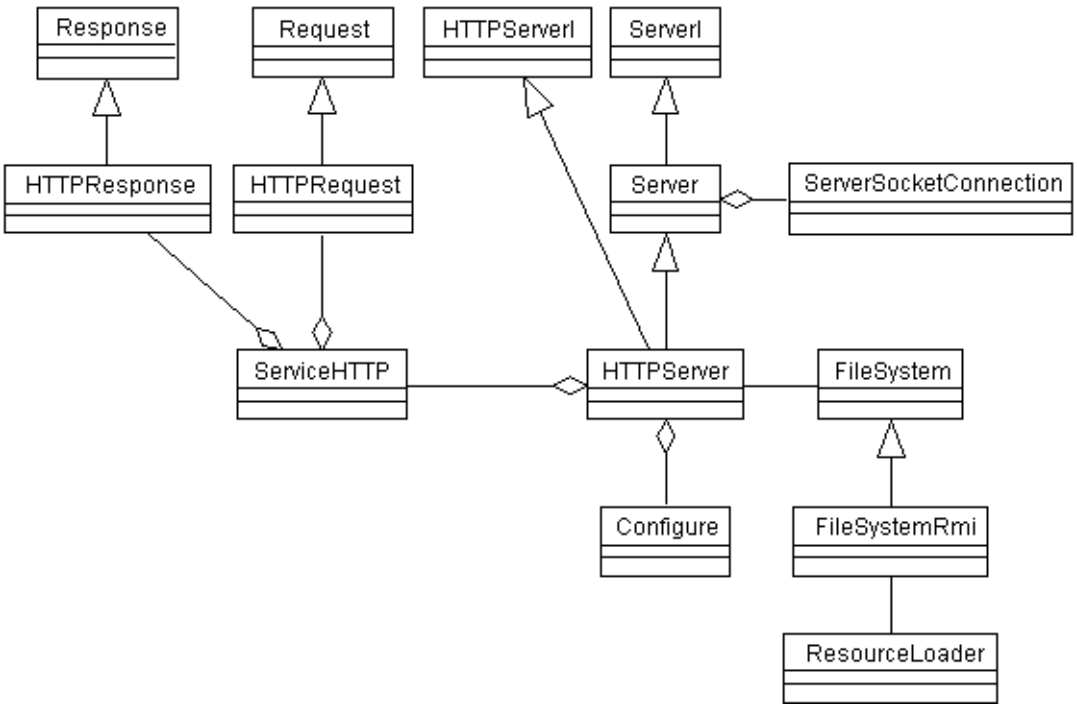

Figura 3.12: Diagrama de clases servidor HTTP.

- **ServerSocketConnection**

  Clase encargada de tratar todo lo relacionado con los sockets tipo servidor o pasivos. Permanece a la escucha y devuelve las conexiones recibidas.

- **Server**

  Clase genérica que representa un servidor cualquiera e implementa el interfaz ServerI. Cualquier servicio deberá heredar de ella e implementar su funcionalidad concreta[6].

- **HTTPServer**

  Representa un servidor HTTP completo que acepta peticiones y lanza hilos de ejecución para atender las solicitudes.

simultáneas que pueden abrirse, aunque a nivel de aplicación la variable del fichero de configuración es la que permite un número determinado de conexiones simultáneas, siempre que no se superen las establecidas internamente.

[6]La clase HTTPServer hereda de Server y añade la funcionalidad específica de un servidor HTTP.

- **HTTPRequest**

  Realiza todas las tareas relacionadas con el tratamiento de las peticiones HTTP (lectura, análisis de las cabeceras, comprobación de los métodos, etc ...).

- **HTTPResponse**

  Lleva a cabo la creación de una respuesta HTTP 1.0, creando las cabeceras de respuesta pertinentes e incluyendo en su caso el recurso solicitado por el usuario.

- **FileSystem**

  Esta clase define la funcionalidad de un sistema de archivos básico. Siguiendo el patrón *Strategy*[7] va a permitir que el sistema de archivos pueda ser de cualquier tipo definido por el programador (RMS, XML, bases de datos, etc ...).

- **FileSystemRms**

  Representa un sistema de archivos concreto. Éste se va a basar en la funcionalidad proporcionada por el paquete *javax.microedition.rms* que incluye una serie de clases e interfaces con el objetivo de permitir el almacenamiento persistente en dispositivos MIDP.

- **Configure**

  Contiene la configuración del servidor HTTP obtenida a través de un fichero de texto incluido junto al servidor. Esta clase proporciona los métodos necesarios para consultar información relativa a la situación actual del servidor así como sobre los tipos MIME soportados.

- **ResourceLoader**

  Clase encargada de obtener recursos externos como pueden ser imágenes, archivos de texto, sonidos, etc ...

  Va a emplearse para realizar la carga inicial de los recursos disponibles en el servidor HTTP.

- **Servidor**

  Clase principal del sistema, que a modo de prueba establece una serie de recursos en el servidor y lanza un servidor HTTP.

[7]Este patrón permite implementar diferentes algoritmos de manera transparente al usuario así como minimizar los cambios a realizar en el caso de querer añadir incluir nuevas "estrategias".

3.7.8. Diagramas de secuencia

A continuación se muestran los diagramas de secuencia de diversas situaciones planteadas en el servidor HTTP.

En la figura 3.13 se observa la secuencia de una petición GET.

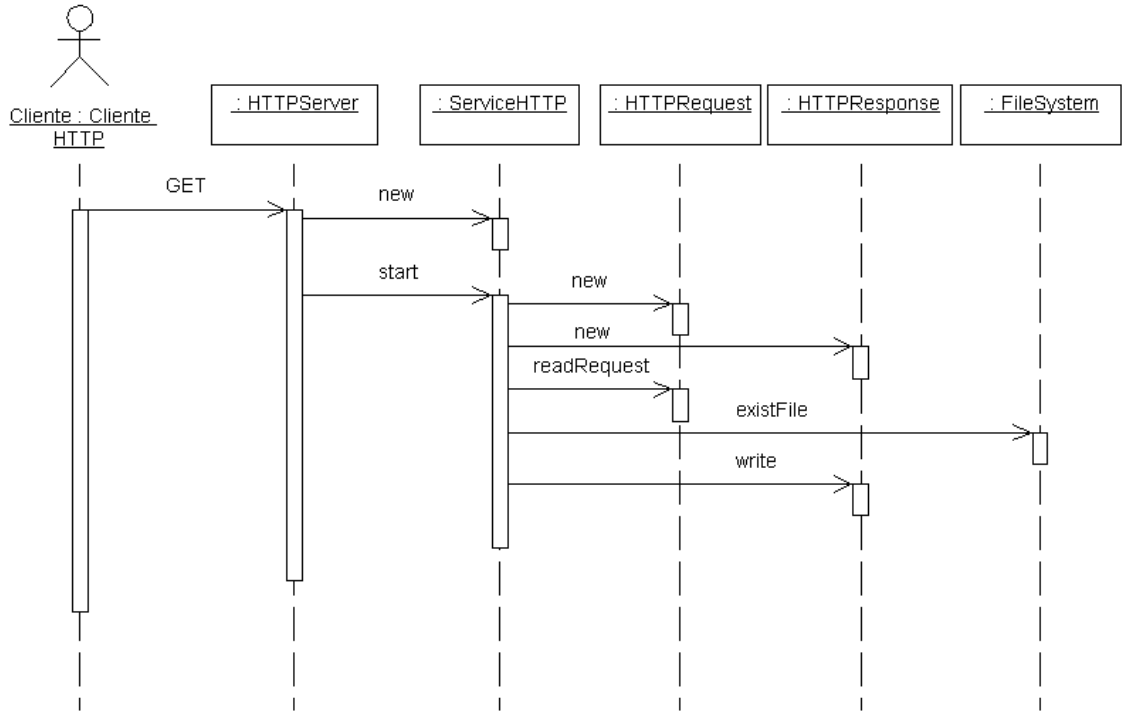

Figura 3.13: Diagrama de secuencia petición GET.

La secuencia de pasos seguidos para lanzar un servidor HTTP son los mostrados en la figura 3.14.

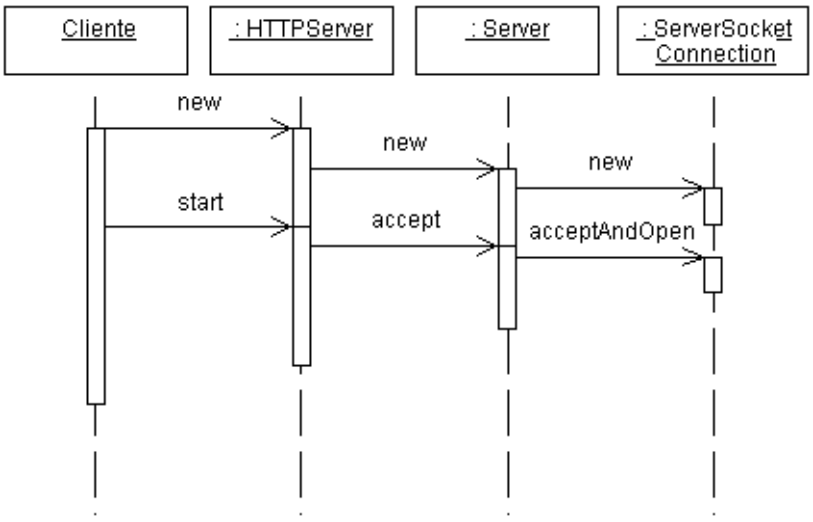

Figura 3.14: Diagrama de secuencia lanzamiento servidor HTTP.

3.7.9. Threads

Aunque J2ME no soporta las grupos de threads, los threads indivualmente sí son admitidos por esta plataforma.

J2ME incluye diferentes posibilidades de crear hilos de ejecución[8] así como métodos de establecimiento de prioridades y control de los mismos.

La clase encargada de realizar el tratamiento de las peticiones se ha implementado heredando de la clase *java.lang.Thread* e incluyendo el método *run* de modo que cada vez que se acepte una conexión, el servidor creará un nuevo hilo que, de manera independiente, tratará dicha conexión, finalizando su ejecución una vez atendida la solicitud.

3.8. Funcionalidad implementada

En los siguientes apartados se va a comentar cuáles son los métodos soportados por el servidor así como las cabeceras de respuesta con las que va a poder responder.

3.8.1. Métodos soportados

Inicialmente se encuentran implementados los métodos **GET**, **HEAD** y **OPTIONS**.

[8]Ver clase Thread e interfaz Runnable.

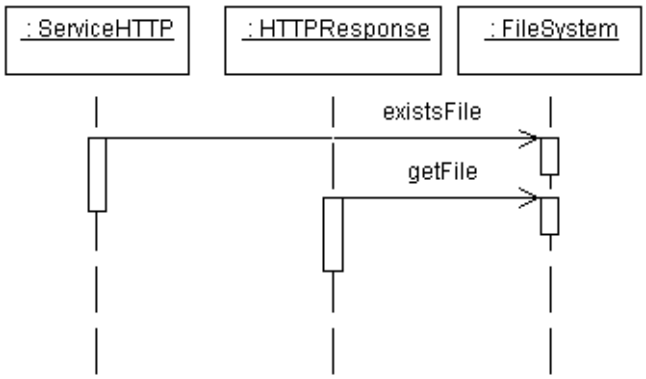

Figura 3.15: Diagrama de secuencia acceso a archivos en servidor HTTP.

Mediante el método GET se va a poder solicitar cualquier recurso que se encuentre en el sistema de archivos del servidor. Las peticiones realizadas por el servidor van a realizarse con "tipo de acceso" *público* no pudiendo acceder a aquéllos recursos con acceso *privado* [9].

Método	**Descripción**
GET	Solicitud de un documento
HEAD	Información de un documento
OPTIONS	Métodos disponibles

Cuadro 3.1: Métodos HTTP soportados.

El método OPTIONS ofrece información sobre los métodos disponibles en el servidor y mediante HEAD se puede obtener información relativa a un recurso específico.

3.8.2. Cabeceras de petición

El servidor captura todas las cabeceras recibidas en las peticiones de los clientes. Estas cabeceras son almacenadas para que en futuras versiones de la aplicación se realice un tratamiento adecuado de las mismas.

Por el momento ninguna cabecera es tratada. Es la línea de petición lo único que va a tener sentido para el servidor, que analizará el método, el recurso y la versión HTTP del cliente que la realice.

[9] En posteriores versiones se implementará un mayor nivel de granularidad, creando una mayor variedad de tipos de acceso.

3.8.3. Cabeceras de respuesta

Las posibles cabeceras de respuesta del servidor HTTP van a ser las mostradas en la tabla 3.2.

Cabecera	**Valor**
Server	Servidor J2ME Experimental
Connection	close
Language	en
Cache-Control	no-cache
Content-Type	tipo mime obtenido del fichero de configuración
Content-Length	tamaño del documento solicitado

Cuadro 3.2: Cabeceras de respuesta del servidor HTTP.

3.8.4. Errores soportados

Los errores que devuelve el servidor son del tipo 2XX, 4XX y 5XX, ya que no existen respuestas parciales ni redirecciones.

Los códigos devueltos son los mostrados en la tabla 3.3.

En futuras versiones del servidor cabe la posibilidad de contemplar un mayor rango de errores. Inicialmente sólamente se contemplan aquéllos relativos a comandos incorrectos, errores del servidor así como accesos ilegales a recursos del sistema (inexistencia de un documento o acceso no permitido a un recurso).

3.9. Fichero de configuración

El servidor HTTP incluye un fichero de configuración a través del cual se puede controlar alguna de las opciones de funcionamiento del mismo, así como incluir los tipos MIME reconocidos por el servidor.

Lo primero a especificar en el fichero de configuración es el *DefaultPort* o puerto por defecto en el que se lanzará el servidor. A continuación el campo *Server* indica el nombre del servidor, que será el valor indicado en la cabecera de respuesta *Server*. Mediante *maxProcessors* se permite seleccionar el número máximo de conexiones que va a poder atender simultáneamente[10].

[10]Este dato es relativo, ya que existe una variable interna del MIDP que indica el número máximo de conexiones permitidas.

Resultado	Código	Descripción
HTTP_OK	200	La petición es correcta, se muestra el resultado
HTTP_BAD_REQUEST	400	La petición es incorrecta
HTTP_FORBIDDEN	403	No se tiene acceso al recurso solicitado
HTTP_NOT_FOUND	404	El recurso solicitado no existe
HTTP_BAD_METHOD	405	El método empleado no existe
HTTP_INTERNAL_ERROR	500	Se ha producido algún error interno en el servidor
HTTP_NOT_IMPLEMENTED	501	El método especificado no se encuentra implementado
HTTP_UNAVAILABLE	503	No es posible atender la solicitud debido a sobrecarga del sistema. Si se repite la solicitud más adelante es posible que pueda ser atendida
HTTP_VERSION	505	El servidor no entiende la versión HTTP del cliente

Cuadro 3.3: Códigos de espuesta del servidor HTTP

Por último a través de *connectionTimeOut* se controla el tiempo que puede pasar inoperativa una conexión abierta.

Una vez establecidas las variables de configuración del servidor se indican los tipos MIMEs del servidor, indicando en primer lugar el tipo y a continuación la extensión de los archivos de ese tipo.

Un ejemplo de fichero de configuración podría ser el siguiente:

```
DefaultPort: 80
Server: Servidor Experimental J2ME
maxProcessors: 6
connectionTimeOut: 60000

text/plain .txt
application/pdf .pdf
application/msword .doc
application/pdf: .pdf
application/postscript: .ps
application/vnd.ms-excel: .xls
application/vnd.ms-powerpoint: .ptt
```

```
aplication/x-gzip: .gzip
application/x-java-archive: .jar
application/x-java-serialized-object: .rms
application/x-java-vm: .class
audio/basic .snd
audio/x-aiff .aif
audio/x-wav .wav
audio/midi .mid
text/html .html
text/plain .txt
image/gif .gif
image/jpeg .jpeg
image/png .png
image/tiff .tiff
image/x-xbitmap .bmp
video/mpeg .mpg
video/quicktime .qtm
```

El servidor buscará en esta lista el tipo MIME del documento solicitado, estableciendo como tipo por defecto[11] el .html.

3.10. Aspectos relevantes a considerar

A la hora de diseñar el servidor ha sido necesario barajar distintas posibilidades en cuanto al funcionamiento del mismo. Existen una serie de aspectos a tener en cuenta y se mencionan a continuación:

- **Configuración persistente frente a no persistente**

  Aunque tanto la versión HTTP 1.0 como la 1.1 soporta las conexiones persistentes[12] se ha optado por el soporte único de conexiones no persistentes con el fin de simplificar el tratamiento de las peticiones HTTP.

- **Número máximo de conexiones**

  El número máximo de conexiones simultáneas permitido va a depender en primer lugar de la variable interna del MIDP

[11] Esto se dará en la situación de que el tipo solicitado sea desconocido por el servidor.

[12] En la versión HTTP 1.1 el funcionamiento por defecto es con conexiones persistentes, en HTTP 1.0 por defecto es no persistente.

com.sun.midp.io.http.max_persistent_connections, no obstante la variable del fichero de configuración del servidor es la que va a controlar el número máximo de conexiones simultáneas, siempre que este número sea inferior al especificado por la variable interna.

- **Obligatoriedad versión protocolo**

  En el RFC del protocolo HTTP se contempla la posibilidad de no especificar la versión del protocolo utilizada por el cliente. En este servidor se ha establecido como algo obligatorio el que ésta sea especificada.

- **No soporte de peticiones condicionales**

  No se ha incluido soporte de peticiones condicionales por no considerarlo de utilidad. Este servidor está orientado a atender peticiones con las cuales los clientes puedan solicitar documentos y no a realizar ningún tipo de acción sobre el propio servidor.

3.11. Ejemplo de funcionamiento

En este apartado se muestra un sencillo ejemplo del tratamiento de una petición HTTP empleando para ello el emulador incluido en el Wireless Toolkit.

3.11.1. Descripción

El formato de distribución de la aplicación va a ser un archivo jar que junto a su descriptor va a ser todo lo que necesita el emulador para ejecutarlo.

Dentro del archivo jar se van a incluir no sólo las clases necesarias, si no aquéllos recursos (archivos HTML e imágenes en este caso) que va a incluir el servidor HTTP en su sistema de archivos.

3.11.2. Inclusión de recursos en el servidor

En primer lugar y dentro de la aplicación se han incluido una serie de imágenes y archivos HTML que constituirán los recursos que el servidor va a poder proporcionar[13]. Estos recursos se incluyen en el MIDlet Suite a través de las clases *ResourceLoader* y *FileSystem*.

[13]El J2ME Wireless Toolkit situa todos los recursos que se desee estén disponibles en el MIDlet, en la carpeta res/icons del directorio de la aplicación del servidor.

Para la inclusión de estos archivos se han utilizado los métodos proporcionados por el sistema de archivos, *FileSystem* así como por el cargador de recursos *ResourceLoader*[14].

3.11.3. Funcionamiento

Al lanzar la aplicación aparece una pantalla como la mostrada en la figura 3.16.

Figura 3.16: Pantalla de lanzamiento del servidor HTTP.

El servidor ha sido lanzado y permanece a la escucha en el puerto 80

[14]Para más información sobre la inclusión de recursos en el sistema recurrir al apéndice D.

(puerto por defecto), cualquier cliente que se conecte al dispositivo y en este puerto será atendido por el servidor.

A continuación se plantean tres escenarios diferentes siendo el primero el de un usuario que se conecta mediante telnet al puerto 80 y realiza una solicitud incorrecta. El resultado obtenido se muestra en la figura 3.17.

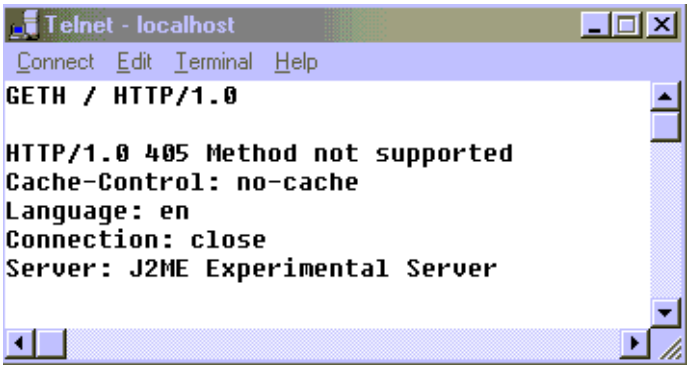

Figura 3.17: Solicitud incorrecta mediante telnet.

Como se puede observar, el cliente ha especificado un método no existente, GETH. El servidor informa del error producido con un código del tipo 4XX.

Si el método empleado es correcto y el documento solicitado, *index.html*, existe en el servidor, se obtendría el resultado mostrado en la figura 3.18.

Por último, al solicitar una imagen el resultado es el mostrado en la figura 3.19.

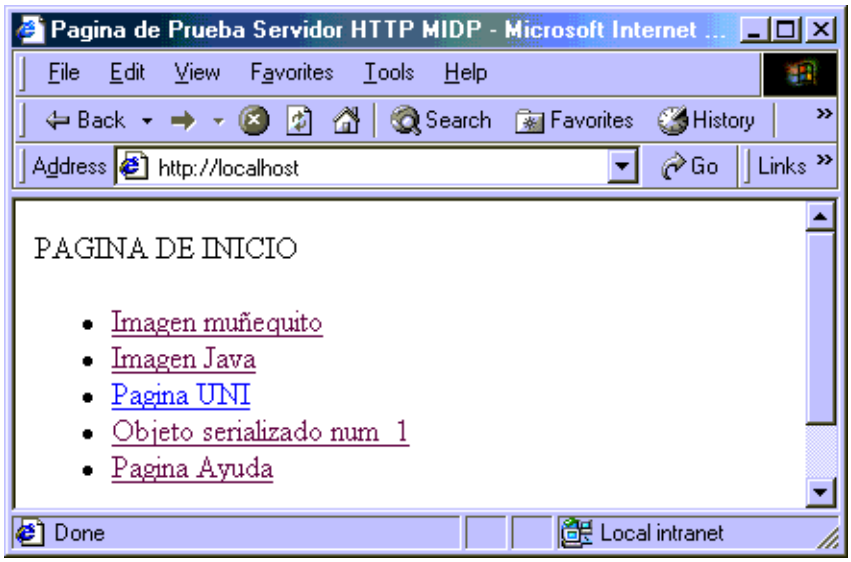

Figura 3.18: Solicitud index.html mediante navegador.

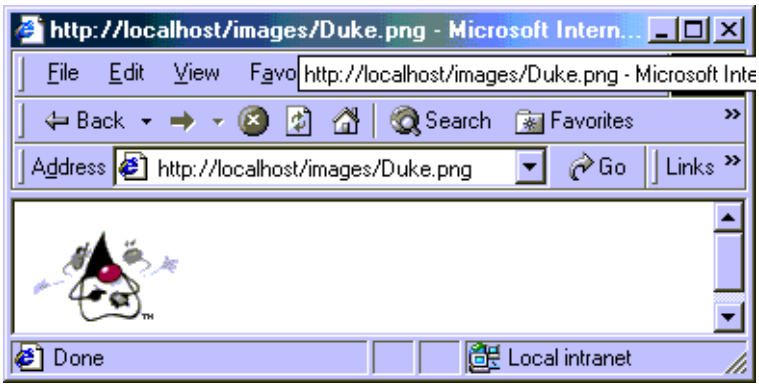

Figura 3.19: Solicitud imagen mediante navegador.

Capítulo 4

Serialización en J2ME

En este capítulo se hace en primer lugar una breve introducción de los conceptos relativos a la serialización y la reflexión con el objetivo de asentar las bases para el posterior desarrollo del mecanismo de serialización en J2SE.

Una vez introducidos todos los conceptos necesarios se expondrá de manera detallada los pasos dados así como los resultados obtenidos al implementar un mecanismo básico de serialización/deserialización en J2ME.

4.1. Introducción

La *serialización* es un mecanismo mediante el cual se puede convertir un objeto en un flujo de bytes que represente su estado, y consecuentemente ser transportado a través de la red o almacenado de manera persistente en un sistema de ficheros. Esta conversión no tendría sentido si no fuese a haber una posterior recuperación, mecanismo denominado *deserialización*.

La serialización viene siendo ampliamente utilizada en protocolos como RPC/XDR que de manera casi transparente al usuario serializan y deserializan objetos constantemente.

Tecnologías como Java (J2SE) también hacen uso de esta tecnología soportando la escritura y lectura de objetos en flujos de bytes, y definiendo una serie de características que van a proteger aquélla información no susceptible de ser serializada.

4.2. Objetivos

Se pretende llevar a cabo la creación de un soporte básico de serialización en J2ME. Este lenguaje no incluye ninguna de las características básicas de

serialización y reflexión[1] de J2SE. Debido a esto, habrá que buscar alguna manera de conseguir una serialización semiautomática, de forma que podamos almacenar el estado de los objetos de J2ME.

Para conseguir este objetivo será necesario definir una serie de interfaces y clases que de manera análoga a la arquitectura J2SE den soporte a la serialización.

En todo momento existirán las limitaciones propias de la arquitectura J2ME y teniendo en cuenta que se pretende que la solución obtenida sea genérica a cualquier dispositivo que soporte unas características mínimas, estará limitada en ciertos aspectos.

Finalmente y una vez conseguido un mecanismo básico de serialización se llevará a cabo un ejemplo que haga uso de lo anteriormente descrito, probándolo en diferentes entornos y dispositivos y cerciorándose del correcto funcionamiento del sistema.

4.3. La necesidad de la serialización

La necesidad de la serialización se ha venido dando implícitamente mediante la utilización de protocolos como RMI, no obstante, existen múltiples campos, como es el de la programación de agentes móviles[2], en el que resulta de vital importancia.

Otro problema que ha venido a resolver la serialización es el de poder mantener el estado de un objeto de manera persistente. En un principio, cuando la serialización no existía el programador tenía que recurrir a soluciones intermedias que pasaban desde escribir de manera explícita los campos de los objetos para posteriormente volver a cargarlos, hasta la necesidad de utilizar pequeñas bases de datos que se leían para recuperar de manera masiva el estado de los objetos.

Respecto a este último punto hay que distinguir cuándo la utilización de la serialización se hace necesaria y cuándo se requiere una base de datos.

El tipo de flujo de datos que se va a emplear en la serialización está así mismo íntimamente relacionado con la utilidad que se quiere dar a la serialización. De este modo se pueden distinguir las siguientes utilidades:

- **Fichero:** si el flujo que se utiliza para llevar a cabo la serialización es un fichero, los datos del objeto se copiarán automáticamente en éste.

[1] Mecanismo de introspección por el cual se puede interrogar a un objeto acerca de cuáles son sus métodos y atributos.

[2] Programas que son capaces de viajar de forma autónoma a través de la red., serializándose y deserializándose a medida que pasan de una máquina a otra.

- **Array de bytes:** al emplear la serialización para escribir los objetos en un array de bytes en memoria, éste puede ser utilizado para crear duplicados del original.
- **Sockets:** cuando se serializa a través de un socket, los datos son automáticamente enviados a través de éste para ser recuperados en el otro extremo, en donde se decidirá qué hacer con éstos.

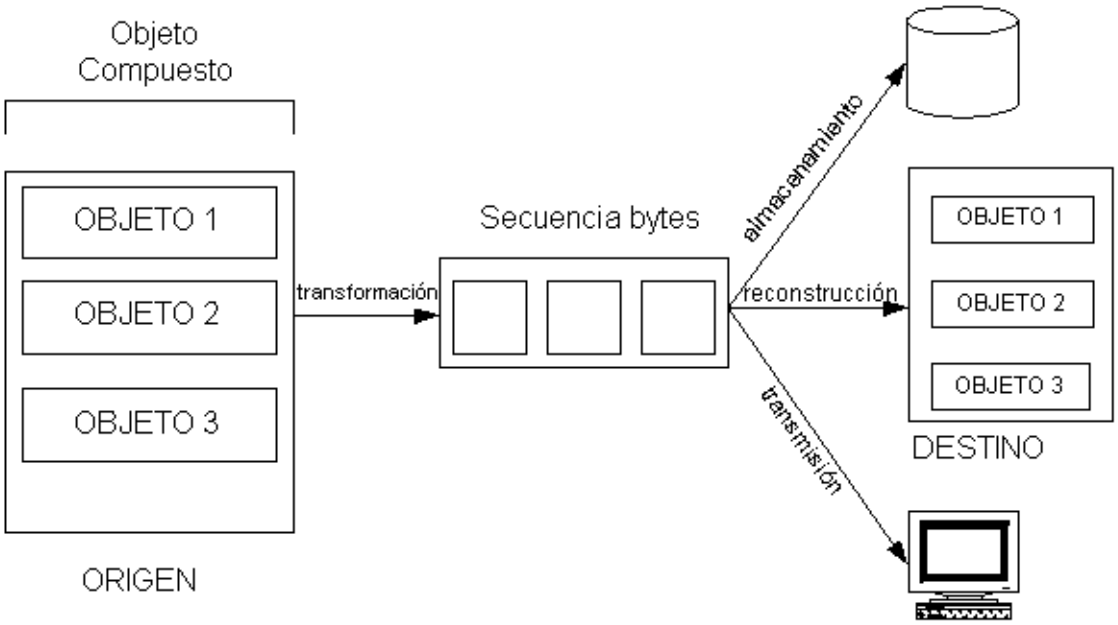

Figura 4.1: Proceso de serialización y deserialización.

Existen otras aplicaciones prácticas de la serialización como son determinadas aplicaciones distribuidas o programas complejos de cálculo. En este tipo de programas, existe un servidor muy potente que va a llevar a cabo una serie de cálculos complejos. Para ello los clientes le envían una serie de objetos serializados con la información necesaria para operar, seguidamente el servidor lleva a cabo los cálculos correspondientes y devolve el resultado a los clientes en forma de objetos serializados.

4.4. El problema de la seguridad

Uno de los problemas planteados por el uso de la serialización es la seguridad. El hecho de almacenar nuestros objetos como un flujo de bytes hace

que cualquiera pueda acceder a éstos y consiguientemente a su contenido.

Así mismo, campos establecidos en las clases como privados, van a poder ser accedidos sin ningún problema.

Posteriormente se verá qué mecanismos se emplean para evitar estos problemas. Éstos van desde la utilización de protocolos seguros como el HTTPS hasta el empleo de mecanismos de cifrado en la propia serialización o la utilización de modificadores en los campos que no se desea que aparezcan en la serialización.

4.5. Serialización en J2SE

A continuación se describe el proceso de serialización y reflexión existente en J2SE con el objetivo de asentar aquéllos conceptos básicos que van a servir en posteriores capítulos para comprender el desarrollo llevado a cabo en este campo.

4.5.1. Descripción

A partir de la versión 1.1 de Java se ha añadido la característica conocida como serialización de objetos que permite tomar cualquier objeto que implemente el interfaz **Serializable** y convertirlo en una secuencia de bytes.

Mediante la serialización se podría, por ejemplo, construir un objeto en una máquina Windows, serializarlo y enviarlo a través de la red para que una máquina Unix pueda reconstruirlo. No hay que preocuparse de la representación interna de los datos o del orden de los bytes, ya que el proceso de deserialización es el que conoce cómo se organizan los bytes y qué significado tiene cada uno.

La serialización se añadió al lenguaje Java para soportar dos características principalmente. La primera de ellas, conocida como RMI (*Remote Method Invocation*), permite a objetos que viven en otras máquinas comportarse como si se encontrasen en la tuya. Las aplicaciones RMI están compuestas generalmente de un servidor y de un cliente. El servidor será el encargado de crear objetos remotos y de servir referencias a los clientes de modo que sus métodos sean accesibles. Por otro lado el servidor puede también transferir estos objetos, serializándolos en primer lugar y tranportándolos a través de la red hacia el cliente que lo haya solicitado.

Las aplicaciones pueden utilizar dos métodos diferentes para obtener referencias a objetos remotos. El primer método consiste en registrar sus objetos remotos a través de la utilidad *rmiregistry* que será la encargada de publicar los nombres de las clases disponibles. Una segunda posibilidad es el hecho de

que la aplicación puede devolver referencias de objetos remotos como parte de su funcionalidad.

Un último aspecto a tener en cuenta es que todos los mensajes que se envían a los objetos remotos han sido serializados en primer lugar, ya que tanto los comandos como los argumentos y valores de retorno son objetos.

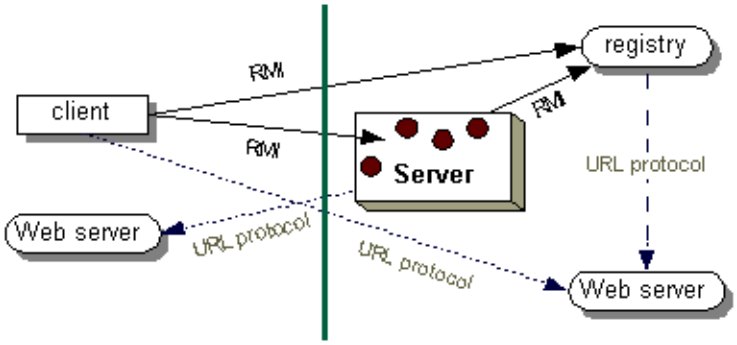

Figura 4.2: Funcionamiento de RMI.

La segunda característica para la cual es necesaria la serialización son los JavaBeans (también introducidos en la versión 1.1 de Java). Los JavaBeans son componentes de software reutilizable que pueden ser manejados con una herramienta visual de construcción. Una de sus principales características es la introspección, que consiste en que una herramienta de construcción puede examinarle para ver cómo está construido. La información de estado de un Bean se configura generalmente en tiempo de diseño. Al utilizar un Bean, su información de estado deber ser almacenada y posteriormente recuperada cuando se lanza el programa; la serialización de objetos se encarga de llevar a cabo estas tareas.

Desde la versión 1.1 muchas de las librerías estándar de Java se han cambiado para que sean serializables, incluyendo todos los wrappers de los tipos primitivos. La interfaz serializable actúa a modo de bandera, indicando que una determinada clase puede ser serializable.

4.5.2. Reflexión

La reflexión es una característica del lenguaje de programación Java que permite a un programa inspeccionar clases y objetos en tiempo de ejecución. Esta característica no está disponible en otros lenguajes de programación como Pascal, C o C++.

Mediante la reflexión un programa escrito en Java puede acceder a información sobre los campos, métodos y constructores de las clases cargadas. El API de la reflexión representa las clases, objetos e interfaces que se encuentran en la Máquina Virtual Java (JVM).

El paquete en el que se encuentra toda la funcionalidad de la reflexión es el **java.lang.reflect**.

Con el API de la reflexión se puede hacer lo siguiente:

- Determinar la clase de un objeto.
- Obtener información sobre los modificadores de las clases, campos, métodos, constructores y superclases.
- Encontrar constantes y declaraciones de métodos que pertenecen a un interfaz.
- Crear una instancia de una clase cuyo tipo no se conoce hasta el tiempo de ejecución.
- Obtener o establecer el valor de un campo de un objeto, aunque no se conozca su tipo hasta la ejecución.
- Realizar una llamada al método de una clase, desconocido en compilación.

La reflexión trabaja íntimamente relacionada con un tipo especial de objetos, los *Class*. Estos objetos contienen información sobre las clases (meta clases) y son empleados para crear todos los objetos regulares de nuestras clases.

Existe un objeto Class por cada clase de nuestro programa que se almacena en el fichero .class con su mismo nombre, previamente a la ejecución la JVM busca los objetos clase necesarios y los carga en memoria.

Uno de los principales métodos de la clase Class es *forName(String)* que devuelve un manejador a la clase pasada como parámetro. Este manejador nos permitirá poder acceder mediante la reflexión a las propiedades de esa clase.

4.5.3. Mecanismo de serialización

La clave para almacenar y posteriormente recuperar objetos en formato serializable radica en representar el estado de los objetos con la información suficiente para poder recuperarlos. Cualquier objeto susceptible de ser

serializado debe implementar obligatoriamente el interfaz *Serializable* o *Externalizable*.

Existen situaciones en la que un objeto está a su vez compuesto por otros objetos. El mecanismo de serialización debe garantizar que éstos objetos son también almacenados manteniendo la relación con el que los alberga.

Contenedores de objetos serializados

Todos los objetos que actúan como contenedores deben implementar un interfaz que permite a los datos primitivos y a los objetos ser almacenados y recuperados de éste. Estos interfaces son *ObjectOutput* y *ObjectInput*.

Es tarea de estos interfaces:

- Proporcionar un flujo en dónde escribir y de dónde leer.
- Manejar peticiones para escribir objetos en el flujo
- Manejar peticiones para leer objetos en el flujo

Cada objeto que se vaya a almacenar en el flujo debe permitir explícitamente ser almacenado y recuperado, implementando para ello los protocolos necesarios (*Serializable* o *Externalizable*).

Estos protocolos permiten al contenedor pedir al objeto que escriba o lea su estado.

Escritura en un Object Stream

Los pasos a seguir para escribir un objeto en un Stream son los siguientes:

1. Crear un Stream de salida (puede ser un socket, fichero, etc ...).
2. Crear un *ObjectOutputStream* que es la clase encargada de serializar los objetos en Java.
3. Realizar la serialización de los objetos mediante el método *writeObject(Objeto)* empleando para ello el *ObjectOutputStream* asociado a un flujo de salida.

A continuación se incluye un ejemplo de cómo se puede serializar dos objetos (una cadena y una fecha) en un fichero temporal llamado *tmp*.

```
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s = new ObjectOutputStream(f);

s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

El método *writeObject* serializa el objeto especificado y recorre sus referencias a otros objetos recursivamente para crear una representación serializada completa del grafo de objetos.

Los datos primitivos como *String.Float* o *Int* pueden ser escritos en el stream con los métodos del interfaz *DataOutput writeInt*,*writeFloat*, *writeUTF* , etc ...

Lectura de un Object Stream

Para deserializar un objeto es necesario crear un ObjectInputStream que lea del flujo de entrada en donde se encuentra el objeto.

Generalmente los objetos se leen con el método *readObject* y los datos primitivos con los métodos de la clase *DataInput*.

En la lectura de un Object Stream, el método *readObject* deserializa el siguiente objeto que encuentre en el flujo recorriendo todas las referencias recursivamente a otros objetos para crear el grafo completo de objetos serializados.

El código de lectura de un objeto almacenado en un Object Stream podría ser el siguiente:

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

4.5.4. Validación

Debido a que la serialización nos permite almacenar un objeto fuera de su entorno de ejecución, puede haber situaciones en las que sea necesario validar el objeto previamente a la deserialización, cerciorándose de que el estado de este objeto es el esperado.

Una de las características adicionales que proporciona la serialización en Java consiste en la validación de objetos.

La manera de asegurar que un objeto está en el estado adecuado es registrar un interfaz *ObjectInputValidation* con la clase *ObjectInputStream*, que

dispone del método *registerValidation*. Este método recibe dos parámetros, el primero,*validator*, identifica el objeto que recibe la llamada del validación, el segundo parámetro es empleado para ordenar los distintos validadores (en caso de que haya varios).

4.5.5. Versionamiento de clases

A la hora de deserializar puede que nos encontremos con que los datos serializados no corresponden con los de nuestra clase. Esto se puede deber a que la clase haya ido evolucionando y el archivo que tenemos serializado se encuentra obsoleto. Si se da esta situación y tratamos de deserializar una clase de este tipo obtendríamos una excepción del tipo *java.io.InvalidClassException*.

El mecanismo de serialización de Java controla las modificaciones de las clases calculando una especie de firma de 64 bits para cada clase. Esta firma recibe el nombre de *serialver* y se basa en los diferentes atributos de la clase. Cuando añadimos o eliminamos algún atributo, el identificador de versión que se generará será diferente, con lo que dos versiones de una misma clase tendrán un identificador diferente.

Respecto a esto se puede pensar que el hecho de añadir un atributo, por ejemplo, no tiene que hacer que la clase sea incompatible con la versión anterior que no tenía este atributo. Esto se puede solucionar en Java estableciendo en la nueva versión de la clase el serialVersionUID a mano, de manera que coincida con el de la antigua versión.

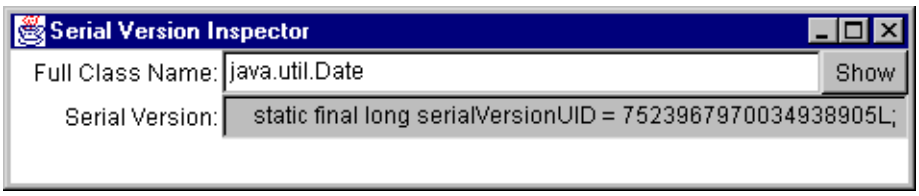

Figura 4.3: Herramienta Serial Version Inspector.

Bastaría añadir una línea del tipo:

```
static final long serialVersionUID = 4070409649239230568L
```

El problema que surge aquí es cómo conocer el número de versión de una clase, para ello a partir de la versión 1.2 del JDK existe la herramienta **Serial Version Inspector** que permite obtener el número de versión de serialización a partir de una clase. El resultado de esta deserialización sería el objeto completo, salvo que el nuevo atributo aparecería con valor nulo.

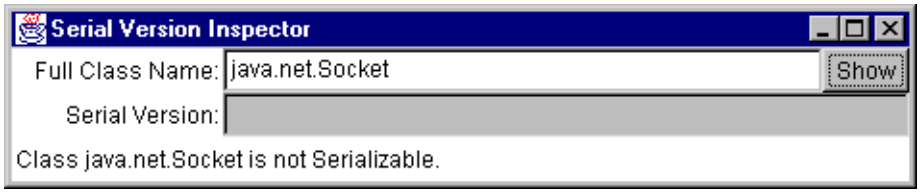

Figura 4.4: Aplicación de la herramienta Serial Version Inspector sobre la clase Socket.

Es a tener en cuenta que no todas las clases son serializables, tal es el caso de los sockets, ya que no tiene sentido alguno su serialización. En esta situación el resultado de aplicar la herramienta *serialserver* se puede apreciar en la figura 4.4.

4.5.6. Seguridad

Uno de los problemas planteados en la serialización es el hecho de que puede que haya campos que no deseamos sean serializados. Para ello existe el modificador *transient* que aplicado a un campo hace que éste no aparezca en la serialización.

Esto dice a la serialización por defecto que ignore este campo. Otro caso en que puede que nos interesa aplicar esta palabra clave es cuando determinados campos no nos interesa que aparezcan, bien porque suponen información redundante o tenemos limitaciones de memoria.

4.6. Serialización en J2ME

En los siguientes apartados se procede a describir el sistema básico de serialización llevado a cabo. Se comenzará planteando unos objetivos a cumplir teniendo en cuenta las limitaciones existentes. Tras comentar los elementos necesarios para la solución elegida se pasará al apartado de implementación que describirá la solución más a fondo.

4.6.1. Descripción

El lenguaje de programación Java 2 Micro Edition no contempla características tales como la serialización o la reflexión. Su orientación a dispositivos móviles con limitaciones de memoria y capacidad de procesamiento, así como las restricciones de seguridad impuestas hacen que no se disponga de tales mecanismos.

Por otro lado, y teniendo en cuenta que los dispositivos móviles son un entorno propicio para la programación de agentes, la cual emplea la serialización y que el CLDC de J2ME tiene soporte para los sockets, puede ser interesante llevar a cabo algún tipo de implementación de la serialización de forma que pueda hacer uso de todas estas características.

4.6.2. Objetivos

Se pretende conseguir un mecanismo de serialización semiautomático en el que cada clase serializable sepa qué tiene que hacer para convertirse en un flujo de bytes. Para ello se deberán emplear una serie de interfaces y clases que sirvan de soporte para este proceso, facilitándolo y automatizándolo en la medida de lo posible.

El programador deberá serializar manualmente sus nuevas clases creadas a partir del soporte mencionado anteriormente. Este proceso tratará de simplificarse al máximo de modo que solamente se tengan que implementar los métodos de escritura y lectura según los interfaces definidos.

Por último se realizará una prueba básica de serialización que incluya diversas clases base así como otras creadas por el usuario. Se almacenarán de manera persistente para posteriormente ser recuperadas y deserializadas.

4.6.3. Problemas existentes

El principal problema a afrontar a la hora de serializar es la imposibilidad de conocer en tiempo de ejecución los métodos y atributos de las clases. J2SE

dispone del mecanismo de reflexión, que como ya se ha dicho, permite conocer las características de las clases en tiempo de ejecución.

Esta carencia va a suponer que no pueda llevarse a cabo la serialización automática, siendo el programador el que tenga que escribir cada uno de los atributos en un flujo de bytes. La serialización llevada a cabo facilitará y se encargará de que el programador se despreocupe de cómo se lleva a cabo ésta escritura, del formato interno de los datos así como del proceso de **recuperación** de las clases serializadas.

Un segundo inconveniente encontrado es el hecho de que en J2SE la gran mayoría de las clases base de que dispone son serializables y disponen de los mecanismos necesarios para que se puedan serializar/deserializar. Por el contrario en J2ME no existe este concepto, no hay ninguna clase serializable, con lo que habrá que buscar alguna manera con la que en caso de que el programador incluya alguna de estas clases en su nueva clase serializable, todas puedan ser serializadas.

4.6.4. Elementos necesarios

Si se descarta la reflexión como elemento principal a la hora de serializar, debido a que el perfil MIDP no proporciona los mecanismos necesarios para examinar las clases en tiempo de ejecución, lo siguiente más importante es disponer de algún modo de escribir y leer tipos básicos en un flujo de bytes.

Para ello J2ME proporciona las clases **DataInputStream** y **DataOutputStream** que permiten a una aplicación leer y escribir tipos primitivos Java en y de un flujo de bytes.

La clase **DataOutputStream** tiene métodos como *writeBoolean*, *writeByte*, *writeInt*, *writeLong*, *writeShort* y *writeUTF* que escriben datos primitivos en un flujo de bytes de manera independiente y portable. Por otro lado la clase **DataInputStream** permite leer estos datos de un flujo de bytes mediante llamadas a *readBoolean*, *readByte*, *readInt*, *readLong*, *readShort* y *readUTF*.

Basándose en estas dos clases y creando los interfaces y clases necesarias se podrá lograr un mecanismo básico de serialización/deserialización que si bien necesitará de la ayuda del programador, facilitará en la medida de lo posible su funcionamiento.

4.6.5. Solución planteada

La solución planteada pasa en primer lugar por la creación de un interfaz *Serializable* con el fin de *marcar* aquéllas clases que sí saben cómo seria-

lizarse/deserializarse[3]. Estas clases deberán implementar sus dos métodos (*readObject* y *writeObject*) de modo que cuando se requiera, sean capaces de llevar a cabo cualquiera de estas dos acciones.

El siguiente paso es la creación de dos nuevas clases **ObjInputStream** y **ObjOutputStream**. Estas clases son las que van a controlar el proceso completo realizando las llamadas necesarias a los métodos *readObject* y *writeObject* que todas las clases serializables deben poseer.

Finalmente se desea incluir la implementación serializable de alguna clase estándar (como por ejemplo *Vector*) así como un ejemplo de serialización que la utilice.

4.7. Implementación de la serialización en J2ME

En este apartado se describe el formato de los datos internos del proceso de serialización así como el algoritmo empleado para este fin.

4.7.1. Formato de los datos

El formato de los datos es un aspecto muy a tener en cuenta. Existen diversas situaciones en las que el tamaño de las clases serializadas es crítico. La serialización fue pensada inicialmente para transmitir objetos por la red, siendo el ancho de banda un recurso limitado que hay que optimizar en todo momento.

Al llevar a cabo el proceso de serialización los diferentes objetos son convertidos en un flujo lineal de bytes, que si bien conservan un formato interno, mediante el cual posteriormente se podrán reconocer sus partes, externamente no conservan ninguna estructura.

A grandes rasgos el formato serializado de una clase sería el siguiente:

1. Nombre de la clase
2. Datos primer atributo
3. ...
4. Datos último atributo

[3]A partir de ahora se va a emplear serializar tanto para la serialización como la deserialización.

Debido a que cualquiera de los atributos puede ser a su vez un objeto podría darse la siguiente situación:

1. Nombre de la clase
   a) Nombre clase primer atributo
   b) Datos primer atributo
   c) ...
   d) Datos último atributo
2. ...
3. Datos último atributo

Si nos centramos en la clase VectorS[4] el formato sería:

1. Nombre de la clase: *VectorS*
2. Número de elementos
3. Tipo primer elemento (entero, cadena, boolean, serializable, ...)
4. Dato primer elemento
5. ...
6. Tipo último elemento
7. Dato último elemento

El tipo establecido para los diferentes tipos que un vector pueda contener son los siguientes:

```
NULL         = 0;
ENTERO       = 1;
LONG         = 2;
BOOLEAN      = 3;
CADENA       = 4;
BYTE         = 5;
CHAR         = 6;
SHORT        = 7;
SERIALIZABLE = 9;
```

El tipo NULL representa un elemento del vector vacío, si el tipo es SERIALIZABLE quiere decir que en esa posición del vector hay un objeto de tipo serializable.

[4]Esta clase corresponde a un Vector serializable al que se le han añadido los métodos writeObject y readObject.

4.7.2. Algoritmo de serialización y deserialización

A continuación se describe el algoritmo de serialización y deserialización diseñado.

Para los objetos serializables el método *writeObject* permite a una clase controlar la serialización de sus propios campos. Cada subclase de un objeto serializable debe definir su propio método *writeObject*.

El método de la clase *writeObject*, si está implementado, es el responsable de salvar el estado de la clase. El formato y estructura es responsabilidad completa de la clase.

Serialización

Es la clase *ObjOutputStream* la encargada del proceso de serialización. Una vez creada una instancia de ésta, a la cual se le ha asociado un flujo de bytes de escritura está todo listo para comenzar el proceso.

Cuando la clase *ObjOutputStream* recibe una llamada al método *writeObject* obtiene mediante un parámetro[5] el objeto a serializar.

El primer paso consiste en inspeccionar la clase del objeto a serializar, para ello se realiza una llamada al método *object.getClass().getName()* con el que se obtiene el nombre de la clase. Seguidamente se escribe el nombre de la clase en el flujo de bytes de escritura y se llama al método *writeObject* del objeto recibido para que lleve a cabo la serialización de sus atributos.

Cabe destacar que es un proceso recursivo, por lo que no habrá problemas a la hora de serializar objetos anidados, ya que son las propias llamadas a sucesivos métodos *writeObject* las que se encargarán de aplanar esta recursividad.

Deserialización

Este proceso está íntimamente relacionado con el de la serialización, ya que dependiendo del orden que éste haya seguido habrá que leer una cosa u otra.

Si se parte de un flujo de bytes que alberga uno o varios objetos serializados, el primer paso consiste en leer el nombre de la clase que se encuentra serializada. A continuación se crea un objeto de este tipo mediante *object = (Serializable)Class.forName(className).newInstance()* tras lo cual bastará con llamar al método *readObject* para que todos los valores de sus atributos sean inicializados.

[5] Ver apéndice D.

La llamada al método *readObject* simplemente hace que se vayan leyendo los diferentes valores de los atributos que componen el objeto en el orden escritos, ya que es el propio objeto el que anteriormente se ha serializado, con lo que conocerá cómo deserializarse.

4.7.3. Clases serializables

A continuación se incluyen a modo de ejemplo una serie de clases serializables implementadas a modo de ejemplo.

- **VectorS**

  Esta clase es capaz de serializarse y deserializarse tal y como lo haría un vector de J2SE mediante sus respectivos métodos *writeObject* y *readObject*.

  Representa un vector que puede almacenar cualquier tipo de objetos básicos (cadenas, enteros, booleanos, etc ...) así como cualquier tipo de objeto complejo siempre que éste sea serializable.

- **Prueba**

  Representa una clase que contiene a su vez otras clases también serializables (VectorS).

- **Config**

  Representa una configuración de un sistema así como las operaciones para escribirse y leerse en y de un flujo de bytes.

4.7.4. Diagrama de clases

Inicialmente se muestra en la figura 4.5 el diagrama de casos de uso de la serialización. En éste se muestran las dos operaciones que se pueden llevar a cabo, *serialización* y *deserialización*.

En la figura 4.6 se muestra el diagrama de clases de la serialización. Con el fin de simplificar se han incluido únicamente aquéllas clases relacionadas directamente con la serialización así como la clase Vector y Serial que las utilizan para llevar a cabo un sencillo ejemplo de serialización/deserialización.

Cualquier aplicación que desee serializar una clase deberá crear un *ObjOutputStream*. Si lo que se desea es deserializar habrá que emplear un *ObjInputStream*.

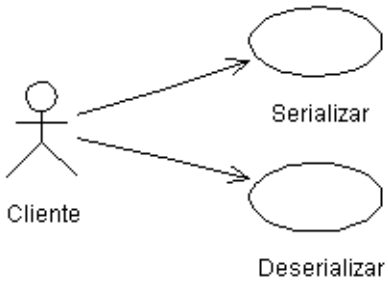

Figura 4.5: Diagrama de casos de uso serialización.

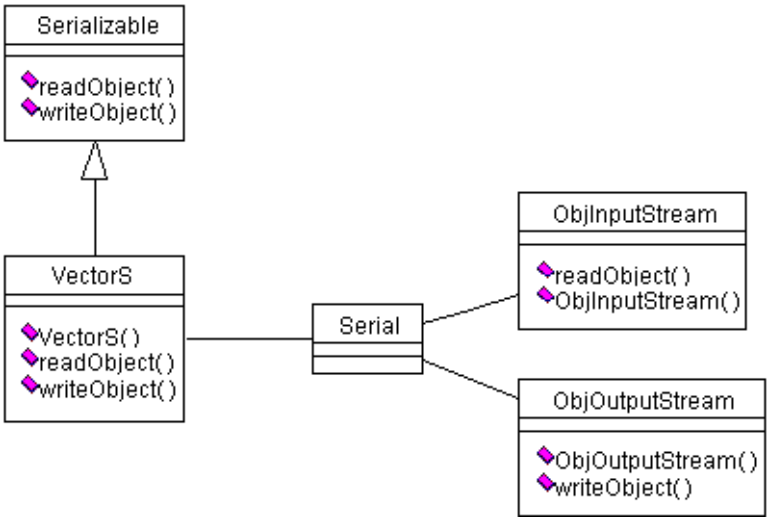

Figura 4.6: Diagrama de clses serialización.

4.7.5. Interfaces

El único interfaz existente es el siguiente:

- **Serializable**

  Sirve para **marcar** que una clase es serializable, ya que al implementarlo tendrá definidos los métodos necesarios para poderse escribir en flujo de bytes con un formato dato y posteriormente recuperarse de ese mismo flujo.

4.7.6. Clases

Aunque se dispone de varias clases que pueden servir de soporte para la serialización (VectorS, Prueba, Config, ...) aquí sólo se incluyen aquéllas que forman parte y son necesarias para llevar a cabo el proceso de la serialización.

- **ObjInputStream**

  Representa un objeto serializado del cual se van a poder leer los diferentes objetos que éste alberga.

  Esta clase está asociada a un flujo de bytes que es realmente la información y hereda de la clase *DataInputStream*, con lo que todos sus métodos estarán disponibles en esta clase.

- **ObjOutputStream**

  Esta clase se va a encargar de serializar los objetos recibidos en su método *writeObject* en el flujo de bytes obtenido en su constructor.

  Esta clase se apoya en los métodos de *DataOutputStream*, clase de la cual hereda.

  A medida que se realicen llamadas a su método *writeObject* los diferentes objetos se irán convirtiendo en un flujo lineal de bytes con una estructura predefinida mediante la cual podrán ser posteriormente reconstruidos a través de la clase *ObjInputStream*.

4.7.7. Diagramas de secuencia y componentes

Diagramas de secuencia

En la figura 4.7 y 4.8 se muestran los diagramas de secuencia respectivamente para el proceso de serialización y deserialización.

Diagrama de Componentes

Finalmente se muestra la figura 4.9 con el diagrama de componentes de la serialización y el servidor HTTP integrados.

4.7.8. Transporte de objetos a través de sockets

Como se verá a continuación una de las utilidades de la serialización es el transporte de objetos a través de sockets.

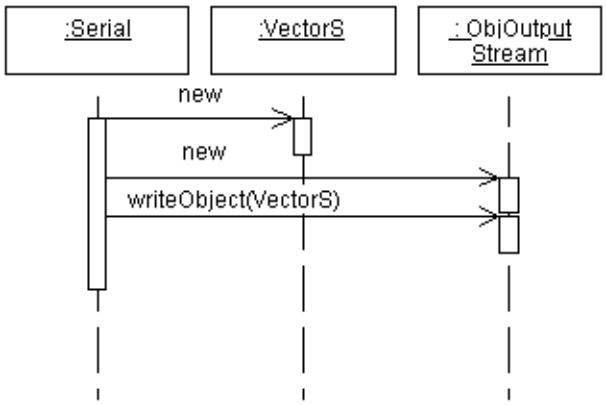

Figura 4.7: Diagrama de secuencia serialización de un Vector.

Si consideramos el caso del protocolo de fechas, ampliamente utilizado en sistemas UNIX, basta con conectarnos al puerto 13 de un servidor para obtener la fecha y la hora. El problema aquí es la necesidad de utilizar caracteres ASCII estándar. Por el contrario si se emplea la serialización para enviar un objeto **Date**, este problema dejará de existir.

Otra utilidad de los sockets en cuanto a la serialización se refiere es la posibilidad de construir un servidor que proporcione objetos serializados que los diferentes clientes reconstruirán previamente a su utilización.

Un último ejemplo de la utilidad de la serialización y los sockets es la creación de una plataforma de agentes, programas capaces de serializarse y viajar a otra máquina a través de la red empleando para ellos los sockets.

Para llevar a cabo cualquiera de las posibilidades mencionadas bastaría asociar como flujo de bytes de las clases *ObjInputStream* y *ObjOutputStream* los correspondientes sockets de entrada y salida.

4.7.9. Serialización y RMS

Centrándose en J2ME la serialización no sólo puede resultar de gran utilidad en el transporte mediante sockets, si no que caben innumerables posibilidades mediante el almacenamiento persistente que J2ME proporciona con el paquete *javax.microedition.rms*.

Podría utilizarse RMS para almacenar los diferentes objetos de una aplicación sin necesidad de una transformación en un formato externo de representación, ya que la aplicación sería capaz de escribir y recuperar sus propios objetos independientemente de el sistema de almacenaje empleado.

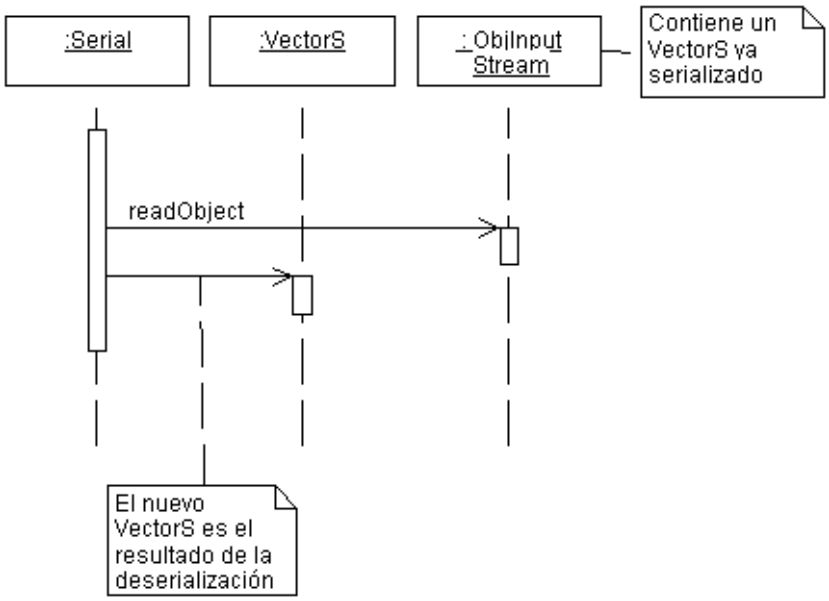

Figura 4.8: Diagrama de secuencia deserialización de un Vector.

4.8. Ejemplo de funcionamiento

En este apartado se describen los pasos a seguir para llevar a cabo un caso básico de serialización.

4.8.1. Descripción

Se pretende implementar una clase compuesta por diversos objetos, de modo que ésta sea serializable.

Una vez inicializados los valores de esta clase, se procederá a su serialización, obteniendo un flujo de bytes que represente el estado de este objeto.

Seguidamente se creará un nuevo objeto del mismo tipo que será inicializado a partir del resultado obtenido con la deserialización del flujo de bytes anteriormente mencionado.

4.8.2. Cómo hacer una clase serializable

El primer paso para hacer una clase serializable consiste en hacer que implemente el interfaz *Serializable*, con lo cual se obliga al programador a proporcionar los mecanismos de serialización(*writeObject*) y deserialización(*readObject)*.

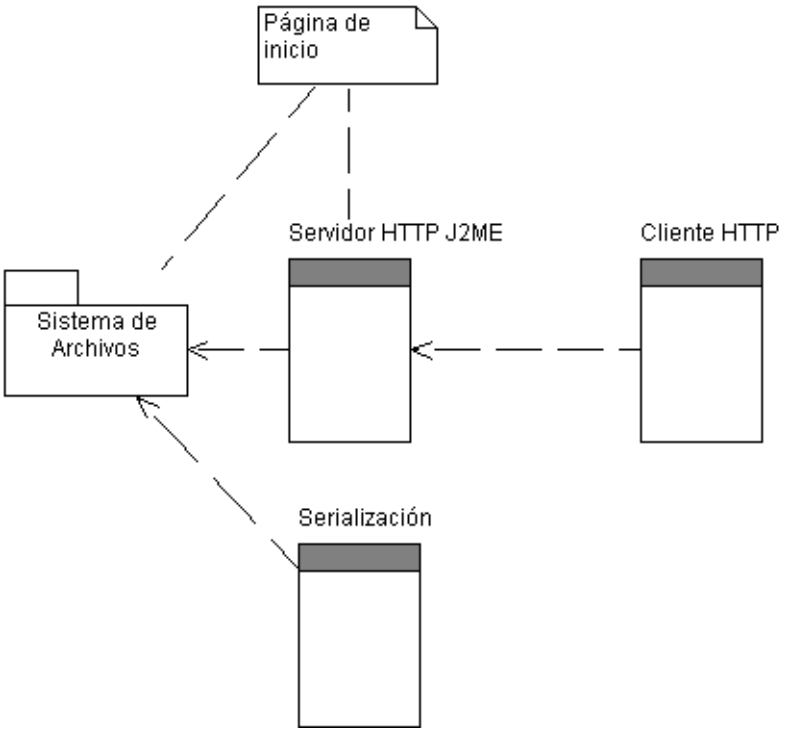

Figura 4.9: Diagrama de Componentes.

```
public class Prueba implements Serializable
```

Supongamos que la clase se ha llamado *Prueba*. El siguiente paso consiste en implementar el método para convertir la clase en un flujo de bytes **writeObject** tal y como se muestra a continuación.

```
public void writeObject(ObjOutputStream out) throws IOException{
  out.writeLong(puerto);
  out.writeLong(bb);
  out.writeInt(cc);
  out.writeUTF(ruta);
  out.writeObject(vect);
  }
```

En el código anterior se observa que el programador debe encargarse de escribir los atributos de la clase en el flujo de bytes recibido por parámetro. Para el caso de objetos que ya disponen de serialización (como la clase VectorS), basta con realizar una llamada a su método de escritura.

Una vez implementado el método de serialización hay que proceder al de la deserialización, **readObject**, que deberá ser en todo momento el proceso inverso al de la escritura.

```
public void readObject(ObjInputStream in) throws IOException{
  puerto = in.readLong();
  bb = in.readLong();
  cc = in.readInt();
  ruta = in.readUTF();
  vect2 = (VectorS)in.readObject();
}
```

Con este método la clase tomaría los valores obtenidos del resultado de deserializar el flujo de bytes incluido en el objeto *ObjInputStream* recibido por parámetro.

4.8.3. Funcionamiento

Tras haber creado una clase serializable, llega el momento de utilizarla. En este caso se va a emplear la serialización con dos fines diferentes.

El primero de ellos va a ser el de inicializar un objeto con el resultado de otro ya serializado. La segunda funcionalidad será la de almacenar el objeto serializado en un soporte permanente. Para ello se empleará el sistema de archivos descrito en apartados anteriores[6].

Los pasos a seguir son los siguientes:

1. Crear el flujo de datos en el que se va a serializar

   ```
   ByteArrayOutputStream datos = new ByteArrayOutputStream();
   ObjOutputStream out = new ObjOutputStream(datos);
   ```

2. Inicializar el objeto a serializar

   ```
   Prueba origen = new Prueba();
   origen.setParametros(11,22,33);
   ```

3. Serializar el objeto

[6]Ver apartado 3.7.4.

```
try{
  out.writeObject(serializo);
  out.flush();
  out.close();
}
catch(IOException e){
  System.out.println(e.getMessage());
}
```

4. Almacenar el objeto en el sistema de archivos

```
FileSystem fs = FileSystem.getInstance();
fs.addFolder("/objects/");
fs.addFile("/objects/","object1.rms","r",datos.toByteArray());
```

5. Inicializar un segundo objeto deserializando el primero. Para ello hay que crear en primer lugar el *ObjInputStream* necesario, así como el objeto a inicializar.

```
ByteArrayInputStream leo =
new ByteArrayInputStream(datos.toByteArray());
ObjInputStream sal = new ObjInputStream(leo);

try{
  sal = new ObjInputStream(leo);
  Prueba leido = (Prueba) sal.readObject();
}
catch(IOException e){
  System.out.println(e.getMessage());
}
```

En este apartado se he mostrado un sencillo ejemplo, aunque las posibilidades de la serialización son mucho más amplias y dependen de la utilidad que se le quiera dar.

Capítulo 5

Pruebas

En este capítulo se incluyen las principales pruebas llevadas a cabo a lo largo del desarrollo de este proyecto.

Las pruebas se encuentran divididas en cuatro grupos diferentes: serialización, servidor HTTP, sistema de archivos y pruebas de integración.

5.1. Pruebas servidor HTTP

A continuación se incluye una lista de las pruebas más destacables llevadas a cabo en la implementación del servidor HTTP sobre MIDP.

1. *Solicitud de documentos variados*

   Con esta prueba se han solicitado diversos tipos de documentos (imágenes, archivos de texto, documentos HTML) para comprobar el correcto funcionamiento del servidor.

2. *Comprobación del cierre de las conexiones*

   Llevando a cabo conexiones canceladas en medio de las transacciones se ha comprobado que el servidor HTTP controla su "cierre" de manera que pueda aceptar nuevas solicitudes.

3. *Superación del número máximo de conexiones*

   Al intentar abrir un número de conexiones superior al permitido el servidor muestra un error informando de la imposibilidad de atender más peticiones en ese momento.

   El número máximo de conexiones simultáneas está determinado inicialmente por el fichero de configuración del servidor, aunque es en última instancia la configuración del MIDP la que define cuántas conexiones pueden existir en un momento dado.

4. *Apertura de conexiones con el navegador Opera*

   Cabe destacar las diferencias existentes entre los diferentes navegadores. Al probar el servidor con el cliente Opera se detectaron varios fallos en cuanto al formato de las cabeceras se refiere.

   Si se solicita un determinado tipo de documento y el servidor responde con un Content-Type diferente al correspondiente el navegador *Internet Explorer* deduce el tipo de documento, sin embargo *Opera* muestra un error por pantalla al no corresponder los tipos MIME.

5. *Introducción de comandos erróneos*

   Mediante esta prueba se ha podido comprobar que el servidor responde con los códigos de error correspondientes de cada situación. Mediante esta prueba todos los errores están correctamente asociados a las situaciones en las que se pueden producir.

6. *Recuperación de archivos protegidos*

   Se ha intentado solicitar documentos protegidos en el sistema[1] a los cuales los clientes HTTP no deberían tener acceso. El resultado ha sido el error correspondiente a la falta de permisos[2].

7. *Atendimiento de peticiones simultáneas*

   Se han llevado a cabo múltiples pruebas referentes a la apertura simultánea de conexiones, monitorizando el número de threads lanzados en el sistema así como su disminución una vez se iban atendiendo las peticiones.

8. *Conexiones mediante telnet y navegador*

   Muchas de las pruebas llevadas a cabo se han ido ejecutando paralelamente tanto desde un cliente telnet como un navegador web.

9. *Pruebas con J9*

   La J9, es una máquina virtual desarrollada por IBM. Esta prueba ha consistido en instalar dicha máquina en un dispositivo PalmOS así como el servidor HTTP implementado. Se ha procedido a intentar conectarse mediante clientes Telnet y HTTP con resultados no del todo satisfactorios.

[1] Ver sistema de archivos sobre RMS.

[2] En este punto hay que diferenciar los errores de autentificación por reinos, no implementados en este servidor y aquéllos relativos a los accesos a documentos no permitidos.

Una de las pruebas más interesantes realizadas al servidor consistió en dejarlo lanzado varios días y monitorizar las conexiones de los diferentes clientes. Un porcentaje bastante elevado de las conexiones pertenecían a usuarios que intentaban llevar a cabo infinitud de acciones *ilegales* sobre el servidor.

La gran mayoría de las peticiones ilegales eran resueltas exitosamente, pero algunas de ellas terminaban haciendo fallar el servidor. El posterior análisis de los ficheros de log generados ha ayudado en extremo a detectar fallos en el servidor.

5.2. Pruebas serialización

Las pruebas más destacables llevadas a cabo sobre la serialización son:

1. *Serialización de una clase sencilla*

   En esta prueba se ha creado una clase con dos atributos primitivos. Tras serializarla se ha probado a recuperarla y comprobar que conservaba su estado.

2. *Serialización de clases compuestas*

   Mediante la serialización de una clase que a su vez contenía otras clases serializables se ha probado el correcto funcionamiento de la recursividad en el proceso de serialización.

3. *Serialización sobre RMS*

   Tras serializar una clase, se ha almacenado sobre RMS para posteriormente recuperarla desde otro MIDlet diferente que se ha encargado de construir una nueva clase a partir del resultado de su deserialización.

4. *Utilización de la clase VectorS*

   En esta prueba se ha creado un vector de tipo serializable, VectorS, al que se le ha inicializado con diferentes tipos, tanto primitivos como clases ya serializables.

   Tras llevar a cabo el proceso de serialización se ha mostrado una traza por pantalla con el resultado del correcto funcionamiento del proceso.

5.3. Pruebas del sistema de archivos

Por considerar el nombre de las pruebas suficientemente descriptivo, sólo se incluye la descripción de aquéllas que contienen algún detalle relevante.

1. *Duplicación de directorios*

2. *Duplicación de archivos en un mismo directorio*

3. *Duplicación de archivos en diferentes directorios*

4. *Recuperación de archivos con permisos incorrectos*

   Se intenta recuperar un archivo existente al cual no se tiene acceso. Se trata de comprobar que los códigos de error establecidos para las diversas situaciones que puedan darse son los esperados.

5. *Recuperación de archivos inexistentes*

6. *Borrado e inserción de un mismo archivo*

7. *Búsqueda de un directorio*

8. *Búsqueda de un archivo*

9. *Borrado de directorios*

5.4. Pruebas de integración

En este apartado se incluyen aquéllas pruebas que manejan las diferentes partes desarrolladas en este proyecto, es decir, el servidor HTTP, la serialización y el sistema de archivos.

1. *Serialización sobre RMS y solicitud HTTP*

   Para llevar a cabo esta prueba es necesario lanzar dos emuladores diferentes. En el primero de ellos se procede a serializar una clase la cual va a ser almacenada en el sistema de archivos creado sobre RMS.

   Seguidamente se lanza un servidor HTTP al cual se conectará un cliente que solicitará el objeto recién serializado. El cliente obtendrá un archivo con extensión rms[3] que contendrá el flujo de bytes que representan el objeto serializado.

2. *Transmisión de objetos serializados mediante sockets*

   En esta prueba se ha creado una versión modificada del servidor HTTP a la que se la han incluido un nuevo comando: SEND. Este método

[3]La extensión establecida en esta prueba para los archivos serializados es RMS, aunque esto queda totalmente abierto.

incluye un parámetro que especifica un recurso. Mediante el comando SEND el servidor envía el recurso especificado al cliente que lleva a cabo la petición.

Los recursos solicitados en este ejemplo van a ser objetos serializados previamente y el resultado de llevar a cabo una petición SEND es la recepción de un objeto serializado que alberga el servidor.

Capítulo 6

Historia del proyecto

En este capítulo se indica la duración total del proyecto así como el tiempo dedicado a cada una de sus partes. Así mismo, se mencionarán las decisiones tomadas y los replanteamientos llevados cabo durante este tiempo.

6.1. Estimación temporal

El desarrollo de este proyecto ha llevado un total de 10 meses divididos entre todas las tareas llevadas a cabo. Cabe destacar que en el momento que una de las tareas planteadas iba avanzando se trabajaba simultáneamente con las siguientes.

A grandes rasgos las tareas y el tiempo dedicado a cada una de ellas ha sido el siguiente:

- **Documentación inicial:** 2 meses

  La primera fase del proyecto consistió en documentarse en dos campos concretos. El primero de ellos es el soporte de sockets en MIDP. Aquí hubo que investigar sobre las diferentes posibilidades existentes a la hora de utilizarlos en modo servidor así como las limitaciones de la versión MIDP 1.0.

  Paralelamente hubo que documentarse en los aspectos relativos a la serialización en J2SE, qué mecanismos la rigen, los recursos que utiliza así como su posible aplicación en J2ME.

  A medida que se llevaba a cabo la fase de documentación se iban diseñando los primeros diagramas de las posibles implementaciones de la serialización y el servidor HTTP en J2ME.

- **Programación:** 5 meses

Una vez asentados todos los conceptos se procede a la fase de programación. Aquí se comenzó por el servidor HTTP con el que aparecieron múltiples problemas en cuanto a la utilización de los sockets se refiere. Una vez superados estos aspectos y creada una primera versión del servidor se procedió a implementar el mecanismo de serialización, que tras un mes y medio aproximadamente estaba finalizado en la mayoría de sus aspectos.

- **Pruebas y depuración:** 1 mes

  La fase de pruebas y depuración se ha llevado a cabo simultáneamente junto a la del desarrollo de la memoria. En esta se han ido llevando a cabo variedad de pruebas con el objetivo de detectar situaciones no contempladas así como para mejorar y optimizar el funcionamiento del sistema.

  En esta fase cabe destacar la reducción del tamaño de las aplicaciones en aproximadamente un 30 por ciento.

- **Desarrollo de la memoria:** 2 meses

  Por último la fase del desarrollo de la memoria ha llevado aproximadamente dos meses. En esta fase se han realizado así mismo diversas pruebas con la máquina virtual de IBM, J9, con el fin de probar el servidor HTTP en el sistema operativo Palm OS.

La tabla 6.1 muestra de manera resumida las diferentes actividades así como su duración.

Fase	**Duración**
Documentación inicial	2 meses
Servidor HTTP	3.5 meses
Serialización	1.5 meses
Pruebas y Depuración	2 meses
Desarrollo memoria	2 meses

Cuadro 6.1: Planificación temporal del proyecto.

6.2. Problemas encontrados

Inicialmente se planteó la posibilidad de implementar la reflexión en J2ME. Tras el análisis del perfil MIDP se comprobó que únicamente con

la funcionalidad proporcionada por éste no había manera de realizarla, con lo que el proceso "automático" de serialización tendría que pasar a ser "semiautomático", ya que la reflexión no iba a poder utilizarse. Se estableció que en futuros proyectos se profundizase en las posibilidades de llevar a cabo reflexión en J2ME.

Otro aspecto que ha dado problemas es el de los threads. Inicialmente se desconocía que el número máximo de conexiones que podían existir simultáneamente estaba definido por un fichero de configuración interno del MIDP. Esto llevó a REplantear los aspectos relativos al número máximo de peticiones simultáneas, ya que ahora iba a estar limitado internamente por el MIDP.

Capítulo 7

Conclusiones y trabajos futuros

7.1. Conclusiones

El objetivo principal en el momento de plantearse el presente proyecto fue el desarrollo de un sistema básico de serialización así como de un servidor HTTP en J2ME, mecanismo que permitiría el desarrollo de una plataforma de agentes sobre dispositivos móviles.

A medida que se iba avanzando en el proyecto fueron surgiendo diversas dificultades, que de un modo u otro fueron solventadas satisfactoriamente.

Finalmente se consiguieron los resultados propuestos e incluso se mejoraron ciertos aspectos con los que no se había contado inicialmente. El desarrollo de un sistema de archivos sobre RMS permitirá a las aplicaciones gestionar la información almacenada de manera óptima, y el paradigma de programación de patrones seguido facilitará en gran medida la reutilización de las clases implementadas.

En todo momento se ha pretendido que el desarrollo sea válido en la próxima versión del MIDP 2.0, con lo que se garantiza la continuidad y validez de la implementación.

Un aspecto destacable del proyecto, es la posibilidad de desarrollo de nuevas aplicaciones que va a ofrecer un servidor HTTP sobre los dispositivos limitados, despareciendo el papel único de clientes que hasta ahora habían desempeñado.

Como aspecto negativo cabe mencionar la imposibilidad de conseguir el código fuente del MIDP para PalmOS, con lo que se pretendía crear un nuevo perfil MIDP que sí soportase los sockets. Este perfil permitiría ejecutar el servidor HTTP en dispositivos con este sistema operativo. Debido a este impedimento se recurrió a la máquina virtual de IBM J9, con la cual se han llevado a cabo algunas de las pruebas realizadas.

Los conocimientos adquiridos a lo largo de estos meses de trabajo han servido para comprender el funcionamiento de una plataforma de programación orientada a dispositivos móviles, el tipo de limitaciones existentes así como las crecientes posibilidades que este tipo de plataformas ofrecen en cuanto al desarrollo de aplicaciones sobre dispositivos móviles se refiere.

7.2. Líneas futuras Servidor HTTP

Algunas de las posibles mejoras y líneas futuras de trabajo del servidor HTTP en J2ME podrían ser:

- **Soporte HTTPS**

  Estableciendo para ello la variable de entorno INCLUDE_HTTPS a true y realizando los cambios necesarios en el código fuente para que se haga uso de este soporte. El empleo de este soporte incrementa el tamaño del perfil en aproximadamente 87 K si se lleva a cabo en la plataforma Windows y 102 K si es Solaris

- **Control de acceso**

  El control de acceso se podría estudiar desde dos puntos de vista diferente. El primero de ellos sería realizando un control de qué dominios pueden acceder al servidor así como estableciendo políticas de autenticación basadas en reinos.

  En segundo lugar se podría incrementar el nivel de granularidad en cuanto a los permisos existentes en el actual sistema de archivos. Podría emplearse una política basada en grupos de usuarios y propietarios o cualquier otra en la que se pueda controlar qué usuarios van a poder acceder a los recursos del sistema y cuáles no podrán.

- **Tratamiento de las cabeceras de peticiones HTTP**

  Actualmente no se realiza ningún tipo de control con las cabeceras recibidas en las peticiones HTTP. En futuras versiones se podrían tratar algunas de estas para, por ejemplo, tener en cuenta el tipo de archivos que soporta el cliente o controlar las peticiones en las que se solicitan documentos modificados a partir de una fecha[1], etc ...

[1]Para controlar peticiones condicionales en cuanto a las fechas se refiere sería necesario realizar modificaciones en el sistema de archivos de modo que existiese una especie de TimeStamp con el momento de modificación de los archivos.

- **Otros sistemas de archivos**

  Dado que J2ME tiene las APIs necesarias para el tratamiento de documentos XML una posibilidad interesante sería almacenar todos los documentos del sistema de archivos en formato XML, para ello habría que tener en cuenta el formato de los documentos así como el uso que se fuese a hacer de ellos, ya que puede haber situaciones en las que un sistema de archivos basado en XML no resulte interesante.

  Otra posibilidad sería la existencia de un sistema de archivos que accediese a la información de manera remota, es decir, que se encargase de obtener los recursos almacenados en algún otro soporte (por ejemplo un PC). Esta posibilidad puede resultar útil si la cantidad de información a almacenar es excesiva para un dispositivo MIDP y los tiempos de acceso a los recursos no supusiesen un coste elevado ni una pérdida de eficiencia.

- **Ampliación de la funcionalidad**

  Podría ampliarse la funcionalidad del servidor HTTP implementando otros métodos HTTP como el POST o el DELETE, también podría resultar de utilidad las solicitudes condicionales, aunque todo depende de la utilidad a la que se vaya a destinar el servidor.

7.3. Líneas futuras de trabajo en la serialización

Inicialmente se ha conseguido un soporte básico para la serialización, si bien se podría continuar avanzando en las siguientes líneas:

- **Parser**

  Que genere automáticamente los métodos *writeObject* y *readObject* de las clases serializables.

  Esto se podría hacer con algún analizador léxico y sintáctico como LEX y YACC, generando para ello una gramática sencilla que a partir de los ficheros fuentes los atributos de las clases y sus tipos para finalmente generar el código fuente de estos métodos.

- **Automatización del proceso**

  Con esta solución se pretendería generar de manera manual los métodos de serialización/deserialización. Una posible solución sería, por ejemplo,

pasar los nombres y tipos de los atributos de la clase mediante un vector a una clase que supiese serializar cualquier cosa.

- **Reflexión**

  Una línea de trabajo bastante interesante consiste en intentar implementar la **reflexión** en su totalidad o al menos parcialmente mediante la utilización de métodos nativos o bien buscando algún mecanismo que permitiese la introspección de las clases en tiempo de ejecución.

- **Interfaz de validación**

  Creación de un interfaz de validación de modo que se solventase transparentemente los problemas con las diferentes versiones de las clases serializadas.

- **Atributo serialversion**

  Empleo del atributo *serialversion* de manera análoga a J2SE.

- **Librería estándar de clases serializables**

  Inicialmente se ha incluido la serialización de la clase Vector. Sería interesante aumentar el número de clases ya serializables de manera que el programador pueda incluirlas en sus nuevas clases sin tener que preocuparse de su serialización.

  Entre las clases más interesantes a serializar inicialmente están las relacionadas con formatos de fechas, tablas Hash así como aquéllas estructuras de almacenamiento dinámico de memoria.

- **Atributo protocolVersion**

  En futuras versiones del mecanismo de serialización desarrollado sería interesante incluir un atributo indicativo de la versión del protocolo empleado. De modo que se pudiese seleccionar la versión a emplear en cada momento y se guardase compatibilidad respecto a anteriores versiones.

Una vez establecidas todas las mejoras oportunas tanto en la serialización como en el servidor HTTP el siguiente paso a dar consistiría en llevar a cabo la carga dinámica de clases.

La existencia de un servidor HTTP sobre MIDP, únicamente proporciona el medio a través del cual los diferentes dispositivos podrán comunicarse a la hora de solicitar las clases que necesiten, así como transferirlas. Restaría pues, el dotar del soporte necesario para que la carga de las clases se hiciese efectiva en tiempo de ejecución.

Una vez funcione la carga dinámica de clases podría ponerse en marcha una primera plataforma de agentes en la que, por ejemplo, un simple MIDlet recorriese diferentes dispositivos, para finalmente retornar a su origen con algún tipo de información recabada en su recorrido.

Apéndice A

Atributos de un MIDlet

En este anexo se incluyen los posibles atributos que puede tener un MIDlet. Estos se dividen en tres tipos:

- **Obligatorios**
- **Opcionales**
- **MIDlet-n**, éstos se emplean para especificar el nombre de cada uno de los MIDlets dentro del MIDlet Suite, así como su icono en formato PNG y la clase principal

En la primera columna se indica el tipo de atributo que es, después se comenta brevemente la finalidad para a continuación establecer en qué archivos aparece (manifiesto y/o jad).

Nombre	**Descripción**	**Fichero**
Obligatorios		
MIDlet-Name	Nombre del MIDlet-Suite	jad y manifiesto
MIDlet-Version	Número de versión del MIDlet Suite	jad y manifiesto
MIDlet-Version	Fabricante del MIDlet Suite	jad y manifiesto
MIDlet-Jar-URL	URL de donde se puede descargar el JAR	jad
MIDLet-Jar-Size	Tamaño en bytes del archivo JAR	jad
MicroEdition-Profile	Perfil J2ME requerido	manifiesto
MicroEdition-Configuration	Configuración J2ME requerida	manifiesto
Opcionales		
MIDlet-Icon	Nombre de la imagen PNG que representa el MIDlet	jar y/o manifiesto
MIDlet-Description	Descripción del MIDlet Suite	jar y/o manifiesto
MIDlet-Info-URL	URL en donde se puede encontrar más información sobre el MIDlet Suite	jad y/o manifiesto
MIDlet-Data-Size	Cantidad mínima en bytes de almacenamiento persistente requerido por el MIDlet	jad y/o manifiesto
[Atributos específicos]	Atributos propios del usuario, nunca comienzan por MIDlet-	jad
Atributos MIDlet-<n>	Nombre del MIDlet, imagen PNG y nombre de la clase del MIDlet	manifiesto

Cuadro A.1: Atributos de los MIDlet

Apéndice B

Variables de compilación y configuración del entorno MIDP

En este apéndice se incluyen las tablas con las variables que afectan a la compilación del MIDP así como los ficheros que definen el funcionamiento del MIDP.

B.1. Variables de compilación del MIDP

Esta sección describe las opciones de configuración que se pueden seleccionar a la hora de construir una configuración concreta del MIDP.

La tabla B.1 muestra la descripción de las variables, así como sus valores por defecto.

B.2. Opciones de configuración del MIDP

La tabla B.2 describe las características configurables desde los ficheros de configuración del MIDP.

Si se desea establecer cualquiera de estas variables de manera temporal, hay que utilizar la opción "-D" desde la línea de comandos. Si el cambio va a ser permanente habrá que modificar los ficheros *internal.config* y *system.config* ubicados en el directorio *build/share/lib* del paquete que contiene los fuentes del MIDP.

Nombre	Tipo	Defecto	Descripción
ALT_BOOTDIR	String	C:\JDK1.3	Localización del entorno de desarrollo (JDK) empleado en la compilación del MIDP.
KVM_DIR	String	../kvm	Localización del directorio raíz de una KVM utilizada para construir el MIDP. (establecido en el archivo Platform.gmk.)
ROMIZING	boolean	true	Si es true, los ficheros class del archivo classes.zip son ROMized convirtiéndolos al formato de un fichero objeto para linkarlos con la imágen de la KVM.
DEBUG	boolean	false	Si DEBUG es true, el fichero objeto es compilado con símbolos de depuración.
ENABLE_DEBUGGER	boolean	false	Si es true, todas las clases Java se compilan con información de depuración.
INCLUDE_ALL_CLASSES	boolean	true	Si es true, la implementación incluirá soporte para todas las clases de la referencia de imlementación CLDC. Si es false, entonces la implementación incluirá sólamente los interfaces y clases definidas en la especificación CLDC.
INCLUDEDEBUGCODE	boolean	false	Si es true, se incluye código extra que permite trazar aspectos de la máquina virtual. Esto conlleva el efecto secundario de disminuir la velocidad de la máquina virtual así como aumentar ligeramente su tamaño.
ENABLEPROFILING	boolean	false	Si es true, y INCLUDEDEBUGCODE está activada, a la salida del MIDP se imprime un mensaje con información estadística sobre la máquina virtual tal como memoria utilizada, aspectos relativos al recolector de basura, etc ...
INCLUDE_I18N	boolean	true	Si está a false, la implementación no incluirá soporte de internacionalización[1].
INCLUDE_HTTPS	boolean	false	Si está a false, la implementación no incluirá soporte HTTPS[2].

Cuadro B.1: Variables de configuración del entorno de compilación del MIDP

Opción	Tipo	Defecto	Descripción
microedition.configuration	String	CLDC-1.0	Número de versión de la configuración J2ME.
microedition.profiles	String	MIDP-1.0	Número de versión del perfil J2ME.
microedition.locale	String	en_US	Ubicación actual del soporte I18N.
microedition.platform	String	j2me	Plataforma actual del host.
system.jam_space	Integer	1000000	Cantidad de espacio reservada para el JAM y el almacenamiento RMS.
system.display.double_buffered	Boolean	true	Si está a true, se va a utilizar doble buffering al escribir en el display gráfico.
system.display.screen_depth	Integer	8	Bandera establecida a 1, 2, 4, u 8 para 2, 4, 16, o 256 colores.
system.i18n.lang	String	en Default	Lenguaje para el procesamiento I18N.
system.i18n.encoding	String	ISO8859_1	Codificacion por defecto para el procesamiento I18N.
system.display.slow_time_interval	Integer	-1	Reduce el repintar gráfico a la velocidad del dispositivo (en microsegundos).

Cuadro B.2: Opciones de configuración del MIDP

Opción	Tipo	Defecto	Descripción
system.display.debug_screen	String	NULL	Marco de depuración en X Windows.
system.display.visual_type	String	TrueColor	Utiliza todas las características visuales X.
system.throttle_per_milli	Integer	50 bytecodes por milisegundo	Disminuye la velocidad de la máquina virtual adaptándola a la del dispositivo.
com.sun.midp.io.enable_extra_protocols	Boolean	false	Permite establecer protocolos adicionales vía Connector.open.
com.sun.midp.midlet.scheduler	String	com.sun.midp.midlet.Scheduler	Clase MIDlet scheduler por defecto.
com.sun.midp.midletsuite.installer	String	com.sun.midp.midletsuite.Installer	Clase instaladora de MIDlet por defecto.
com.sun.midp.lcdui.eventHandler	String	com.sun.midp.lcdui.DefaultEventHandler	Clase manejadora de eventos por defecto.
com.sun.midp.lcdui.skin	String	classes/icons/skin.gif	Imagen de skin alternativo para utilizar en el teléfono del la plataforma J2SE.
com.sun.midp.lcdui.inputHandler	String	com.sun.midp.lcdui.DefaultInputMethodHandler	Clase manejadora de entrada por defecto.
com.sun.midp.io.http.proxy	String	NULL	Valor del servidor proxy HTTP que puede utilizarse para redirigir las peticiones HTTP a través de un proxy cortafuegos, ej. webcache:8080
com.sun.midp.io.http.max_persistent_connections	Integer	4	Máximo número de conexiones persistenes que pueden mantenerse abiertas simultáneamente. Ej, para Palm 1.
com.sun.midp.io.http.force_non_persistent	Boolean	false	Si está establecido a true no se utilizan conexiones persistentes, siempre conexiones close.

Cuadro B.3: Opciones de configuración del MIDP II

Apéndice C

Instalación y configuración del Wireless Toolkit

En este apéndice se describe el proceso de instalación y configuración del Wireless Toolkit de Sun, herramienta utilizada a lo largo del desarrollo de este proyecto para compilar, ejecutar y empaquetar los MIDlets generados.

Existen diversas versiones para Windows y Linux. Se va a comentar brevemente el proceso de instalación para la versión Windows, siendo la de Linux muy similar.

El Wireless Toolkit es un entorno que facilita el desarrollo y testeo de aplicaciones J2ME permitiendo:

- Desarrollar aplicaciones de principio a fin, desde los fuentes Java hasta el MIDletSuite.

- Seleccionar entre dos tipos de herramientas de desarrollo, la basada en un entorno gráfico así como el entorno de la línea de comandos. Es posible compilar y ejecutar las aplicaciones desde la barra gráfica de herramientas así como desde la línea de comandos, desde la cual se pueden especificar un mayor número de opciones.

- Ejecutar las aplicaciones a través de un emulador, así como seleccionar el tipo de dispositivo en el que se desea se lleve a cabo la emulación de la aplicación.

C.1. Componentes del J2ME Wireless Toolkit

En el kit de desarrollo del Wireless Toolkit se incluyen los siguientes componentes:

- *Herramientas* básicas de desarrollo
  - **Preverificador:** se encarga de la preverificación [1] de los bytecode.
  - **Emulador:** ejecuta las aplicaciones en el dispositivo seleccionado (en el J2ME Wireless Toolkit se incluyen cuatro tipos de dispositivos distintos).
- *Utilidades* como son la herramienta de generación de archivos con extensión .prc utilizados por el sistema operativo Palm OS.
- *Componentes* del entorno de desarrollo
  - **KToolBar:** entorno de desarrollo gráfico bastante sencillo
  - **Módulo J2ME Wireless:** módulo incluido para integrar y poder desarrollar aplicaciones J2ME bajo el entorno de desarrollo *Forte for Java*.

C.2. Requerimientos del sistema

Los requerimientos necesarios para instalar el J2ME Wireless Toolkit son:

- **Sistema Operativo** Microsoft Windows 98 Second Edition.
- **Memoria** mínima de 64MB, aunque recomendable 128.
- **Disco Duro** 15 MB.
- **Entorno Java** de desarrollo Java 2, Standard Edition(J2SE), versión 1.3.

Como componente adicional se puede instalar el kit Forte for Java Community Edition[2] 1.0 Update release 2.

[1] El proceso de preverificación se ha explicado anteriormente.

[2] Este entorno de desarrollo tiene unos requerimientos mínimos de un Pentium II a 300Mhz, 128 MB de memoria Ram y 30 MB de espacio libre en disco.

C.3. Instalación del J2ME Wireless Toolkit

Previamente a la instalación del J2ME Wireless Toolkit es necesario desintalar cualquier versión anterior existente.

Seguidamente se deben instalar los siguientes elementos y en el orden especificado a continuación:

1. Java 2 Standard Edition (J2SE), siendo necesaria la versión 1.3 o superior.
2. Forte for Java (si es que se desea que funcione conjuntamente con el J2ME Wireless Toolkit).
3. J2ME Wireless Toolkit.
   - Ejecutar el archivo **j2me_wireless_toolkit-1_0_1-fcs.exe** haciendo doble click en su icono[3].
   - Seleccionar el tipo de instalación deseada, siendo *Stand Alone* si no va a funcionar con Forte for Java e *Integrated* en caso contrario.
   - Continuar el proceso de instalación pinchando en *Next*. Una vez realizado esto el proceso está completo.

[3]Al instalar el J2ME Wireless Toolkit o Forte for Java hay que asegurarse de que no haya espacios en el nombre del directorio de instalación, ya que el ejecutable encargado de la preverificación no los entiende.

Apéndice D

Manual del programador

En este capítulo se incluye la información necesaria para que el programador pueda hacer uso tanto del mecanismo de serialización desarrollado como del servidor HTTP implementado.

D.1. Instalación del entorno de desarrollo

Para consultar la instalación del entorno de desarrollo referirse al apartado C.3.

D.2. Configuración del entorno

Es necesaria la activación de los sockets para el caso del servidor HTTP, para ello existe el fichero de configuración internal.config en el que habrá que establecer a true el parámetro *con.sun.midp.io.enable_extra_protocols*.

La serialización no requiere ninguna configuración en especial.

Para la puesta en funcionamiento en un dispositivo MIDP del servidor HTTP es necesaria la instalación de una máquina virtual que sí soporte los sockets, ya que de lo contrario se producirá una excepción a la hora de lanzar el socket que se encuentre a la escucha de peticiones.

D.3. Fichero de configuración

A continuación se muestran los ficheros de configuración del MIDP.

D.3.1. Internal.config

```
# Visible from com.sun.midp.Configuration.getProperty(key)
#
# System tunable parameters
# system prefix indicates native only access
#
system.jam_space: 1000000
system.display.double_buffered:true
system.display.screen_depth: 8
system.i18n.lang: en
system.i18n.encoding: ISO8859_1
#
# system.display.slow_time_interval:
# system.display.debug_screen:
# system.display.visual_type: TrueColor
# system.display.kh_skins: TrueColor
# system.throttle_per_milli:
#
com.sun.midp.io.enable_extra_protocols: true
#
com.sun.midp.midlet.scheduler: com.sun.midp.midlet.Scheduler
 com.sun.midp.lcdui.eventHandler:
 com.sun.midp.lcdui.DefaultEventHandler
com.sun.midp.lcdui.inputHandler:
 com.sun.midp.lcdui.DefaultInputMethodHandler
#
#com.sun.midp.io.http.proxy: webcache:8080
#
# com.sun.midp.midletsuite.installer:
com.sun.midp.midletsuite.Installer
# com.sun.midp.lcdui.skin: classes/icons/skin.gif
com.sun.midp.io.http.max_persistent_connections: 4
com.sun.midp.io.http.force_non_persistent: false
```

D.3.2. System.config

```
# Properties visible from System.getProperty()
microedition.configuration: CLDC-1.0
microedition.profiles: MIDP-1.0
microedition.locale:  en_US
microedition.platform: j2me
microedition.hostname: localhost
```

```
# microedition.encoding: ISO8859_1
#
# I18N aliases for common character encodings
SHIFT_JIS_InternalEncodingName: SJIS
X_SJIS_InternalEncodingName: SJIS
#
# CLDC implementation parameters
# java.lang.Character.caseConverter:
  com.sun.cldc.i18n.uclc.DefaultCaseConverter
# com.sun.cldc.io.j2me.comm.buffersize: 256
# com.sun.cldc.i18n.Helper.i18npath: com.sun.cldc.i18n.j2me
# microedition.implpath: com.sun.cldc
# javax.microedition.io.Connector.protocolpath: com.sun.cldc.io
# com.sun.cldc.io.j2me.socket.buffersize: 256
# com.sun.cldc.util.j2me.TimeZoneImpl.timezone: UTC
```

D.4. Compilación

La compilación del servidor HTTP y del sistema de serialización pueden realizarse tanto desde la línea de comandos como mediante el interfaz gráfico del Wireless Toolkit.

D.4.1. Compilación Wireless Toolkit

Si la compilación se realiza mediante el interfaz gráfico, deberá existir una estructura de directorios (generada automáticamente) al crear un nuevo proyecto con los siguientes directorios:

- **Servidor—Serial**
  - bin
  - classes
  - lib
  - res
    - icons
  - src
  - tmpclasses
  - tmplib

En el directorio src se deberán situar todos los archivos fuentes y es en la carpeta */res/icons* donde se colocan todos los recursos externos utilizados por el MIDlet[1].

Una vez dispuesta toda la estructura de directorios necesario basta con seleccionar la opción *Build* del Wireless Toolkit para proceder a la construcción de la aplicación.

D.4.2. Compilación mediante línea de comandos

Compilación

Para compilar los fuentes del servidor o de la serialización basta con escribir el siguiente comando.

```
%COMPI%javac.exe -g:none -bootclasspath %RUTAJ2ME%\lib\midpapi.zip -d

%RUTAFUENTES%\tmpclasses -classpath %RUTAFUENTES% %RUTAFUENTES%\*.java
```

Se han utilizado variables de entorno para simplificar el comando. COMPI es la ruta donde se encuentra instalado el jkd1.3, RUTAJ2ME indica el directorio base de el J2ME Wireless Toolkit y por último en RUTAFUENTES como su nombre indica contiene la ruta de los fuentes.

Las opciones empleados son *-g:none* para no incluir información de depuración, *-bootclasspath* para indicar la ruta de las clases j2me a utilizar, con *-d directorio* indicamos dónde queremos que nos deje el compilador las clases recién compiladas y por último *-classpath* que será la ruta dónde se encuentran los fuentes a compilar.

Preverificación

Como ya se ha mencionado anteriormente, J2ME realiza parte de la preverificación "off-line", con este comando podemos llevarlo a cabo. Las opciones y rutas son las mismas que en la compilación salvo el directorio *tmpclasses* que será donde se almacenarán las clases preverificadas.

```
%RUTAJ2ME%\bin\preverify.exe -classpath

%RUTAJ2ME%\lib\midpapi.zip;%RUTAFUENTES%\tmpclasses -d

%RUTAFUENTES%\classes %RUTAFUENTES%\tmpclasses
```

[1] Es en este directorio donde el MIDlet buscará los recursos externos que necesite, como pueden ser los iconos de los programas o cualquier archivo utilizado internamente.

Generación del .jar

Una vez compiladas y preverificadas las clases se pasa al empaquetamiento mediante la utilidad jar, para ello basta indicarle las clases a empaquetar así como los recursos adicionales a incluir.

En este caso los recursos serán el manifiesto que será distribuido con el MIDlet Suite así como el directorio *res* con las páginas Web, el fichero de configuración y las imágenes correspondientes[2].

Las opciones *cmf* indican que se desea empaquetar (c) incluyendo un manifiesto (m) para obtener un fichero (f) que en este caso será Servidor.jar o Serial.jar [3].

```
%COMPI%jar cmf %RUTAFUENTES%\META-INF\MANIFEST.MF

%RUTAFUENTES%\Aplicacion.jar -C %RUTAFUENTES%\classes
```

D.5. Descriptor y Manifiesto

A continuación se muestra el descriptor y manifiesto del servidor HTTP y el mecanismo de serialización.

D.5.1. Servidor HTTP

Descriptor

```
MIDlet-1: Servidor, /icons/Duke.png, Servidor
MIDlet-Jar-Size: 21599
MIDlet-Jar-URL: Servidor
MIDlet-Name: Servidor
MIDlet-Vendor: Guillermo Diez-Andino
MIDlet-Version: 1.0
```

Manifiesto

```
MIDlet-1: Servidor, /icons/Duke.png, Servidor
MIDlet-Name: Servidor
MIDlet-Vendor: Guillermo Diez-Andino
MIDlet-Version: 1.0
```

[2]Estos recursos son necesarios únicamente en el caso del servidor HTTP, para la serialización no es necesario ningún recurso.

[3]En la línea de generación de jar se ha establecido el nombre genérico del paquete como Aplicacion.jar.

```
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

D.5.2. Serialización

Descriptor

```
MIDlet-1: Serial, /icons/Duke.png, Serial
MIDlet-Jar-Size: 14999
MIDlet-Jar-URL: Serial
MIDlet-Name: Serial
MIDlet-Vendor: Guillermo Diez-Andino
MIDlet-Version: 1.0
```

Manifiesto

```
MIDlet-1: Servidor, /icons/Duke.png, Serial
MIDlet-Name: Serial
MIDlet-Vendor: Guillermo Diez-Andino
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

D.6. Ejecución

Al igual que en el caso de la compilación existen dos maneras de ejecutar las aplicaciones, en las dos siguientes secciones se explica brevemente la ejecución mediante el Wireless Toolkit y a través de la línea de comandos.

D.6.1. Ejecución Wireless Toolkit

Para ejecutar un MIDlet con el Wireless Toolkit es necesario haber procedido previamente a la construcción de la aplicación (opción *Build*[4]). Una vez generados los archivos .class bastará con ejecutar la opción *Run* de este entorno.

Cabe mencionar que mediante este tipo de compilación la generación de los archivos de descriptor y manifiesto es automática. El interfaz gráfico del WTK permite controlar qué nuevos atributos se desea añadir, el valor de aquéllos obligatorios, así como la inclusión de los iconos que aparecerán al lanzar las aplicaciones.

[4]Ver apartado D.4.2.

D.6.2. Ejecución Línea de comandos

Como se ha mencionado en capítulos anteriores a la hora de ejecutar un MIDlet es necesario incluir un archivo denominado descriptor. Para ello basta con establecer la opción *-descriptor* en la ejecución mediante línea de comandos.

En el caso del servidor recibirá el nombre de Servidor.jad, siendo Serial.jad para el MIDlet correspondiente a la serialización.

```
set CLASSPATH=%RUTAJ2ME%\lib\kvem.jar; %RUTAJ2ME%\lib\kenv.zip;

%RUTAJ2ME%\lib\lime.jar;%RUTAFUENTES%

%COMPI%java.exe -Dkvem.home=%RUTAJ2ME% -classpath %CLASSPATH%
com.sun.kvem.midp.Main DefaultGrayPhone -descriptor Servidor.jad
```

Con la opción *DefaultGrayPhone* se indica el tipo de emulador a utilizar (existen varios con diferentes tamaños de pantalla, forma, etc ..). Finalmente, con la opción *-Dkvem.home=<nombre>* se establece la ruta para buscar las clases de la aplicación, así como sus recursos.

D.7. Manejo de clases

En este apartado se describe el manejo de las clases implementadas así como sus métodos.

D.7.1. Clases del servidor HTTP

- *ServerSocketConnection*

  Representa la creación de un socket pasivo así como los métodos necesarios para manejarlo.

  Para el correcto funcionamiento de esta clase es necesario activar las funcionalidades relacionadas con los sockets mencionadas en capítulos anteriores.

  - *ServerSocketConnection(String port) throws IOException*
    Realiza una llamada del tipo *serversocket://:port* y devuelve un socket a la escucha en el puerto especificado.
  - *StreamConnection acceptAndOpen() throws IOException*
    Acepta una conexión de entrada y la devuelve[5].
  - *String getLocalAddress()*
    Obtiene el nombre de la máquina en la que se ejecuta el socket. Este nombre corresponderá al obtenido mediante la llamada del sistema *System.getProperty(microedition.hostname)*, que en caso de no existir devolverá *null*.
  - *String getLocalPort()*
    Devuelve el puerto en el que escucha el socket pasivo.
  - *void close()*
    Cierra el socket pasivo.

- Server

  Representa la abstracción de un servidor cualquiera con una funcionalidad mínima implementada.

  - *Server(String port) throws IOException*
    Crea un servidor básico en el puerto especificado por parámetro.

[5]El máximo número de conexiones simultáneas es establecido en el fichero de configuración Internal.config.

- *String getPort()*

  Obtiene el puerto del servidor.

- *String getServer()*

  Obtiene el nombre de la máquina en la que se ejecuta el servidor.

- *StreamConnection accept() throws IOException*

  Acepta una conexión.

- *void stop() throws IOException*

  Cierra el socket del servidor que se encuentra a la escucha.

- HTTPServer

  Representa un servidor HTTP completo que acepta peticiones y lanza hilos de ejecución para atender las solicitudes.

  - *public HTTPServer(String port) throws IOException*

    Crea un servidor HTTP en el puerto especificado por parámetro. Esta clase implementa el interfaz HTTPServerI que a su vez lo hace de Server.

    La creación del servidor implica la creación de todas las clases necesarias para su funcionamiento (Server, ServerSocketConnection, etc ...), aunque no es hasta la llamada del método *start* cuando todas las clases se inicializan y comienzan a ejecutarse.

  - *public HTTPServer() throws IOException*

    Crea un servidor HTTP en el puerto establecido por defecto.

  - *public void start() throws IOException*

    Arranca la ejecución del servidor HTTP. Este método establece en primer lugar la configuración del servidor[6] para a continuación entrar en un bucle que acepta conexiones entrantes.

    Por cada conexión entrante (y mientras no se supere el número máximo de procesos simultáneos establecidos) irá arrancando hilos de ejecución que atenderán las peticiones HTTP debidamente.

  - *public void rest()*

    Decrementa el número de procesos que se pueden arrancar en un momento dado. De este modo se controla el número de procesos simultáneos existentes.

[6] Ver clase Configure.

- *public int currentConnections()*
  Informa del número de conexiones atendiéndose en el momento de la llamada a la función.
- *public Configure getConfiguration()*
  Devuelve la configuración del sistema.

- ServiceHTTP

  Clase encargada de realizar todo el procesamiento de las peticiones de los clientes. Cada vez que se recibe una conexión, se crea un objeto de esta clase que se lanza como un thread independiente, finalizando una vez atendida la petición.

  El número de servicios HTTP que se pueden lanzar va a estar limitado por la variable *maxProcessors* del fichero de configuración así como la variable interna *com.sun.midp.io.http.max_persistent_connections*.

  - *public ServiceHTTP(StreamConnection sc,HTTPServer server)*
    Crea un objeto encargado de atender una determinada petición HTTP. Este objeto va a implementar la clase Thread para poder lanzarse como un proceso en segundo plano.
    Una vez finalizado el tratamiento de la petición HTTP el proceso se elimina automáticamente.
  - *public void run()*
    Crea las clases necesarias para el tratamiento de las peticiones (HTTPRequest y HTTPResponse) y realiza la llamada al método que se encarga realmente de atender las peticiones.
    Este método es al que se llama automáticamente al lanzar cualquier thread mediante *Thread.start()*.
  - *private void treatment()*
    Atiende las peticiones llevando a cabo las acciones correspondientes para cada tipo de petición.
  - *private String checkType(String file)*
    Comprueba el tipo MIME de un recurso solicitado empleando los MIMEs incluidos en el fichero de configuración.

- HTTPRequest

  Realiza todas las tareas relacionadas con el tratamiento de las peticiones HTTP (lectura, análisis de las cabeceras, comprobación de los métodos, etc ...).

- *public HTTPRequest(InputStream ins)*
  Crea un objeto encargado de leer una petición.
- *protected void Analize_Headers()*
  Analiza las cabeceras de la petición HTTP.
- *public String getMethod()*
  Devuelve el método especificado en la petición HTTP.
- *public String getVersion()*
  Obtiene la versión del protocolo HTTP empleada.
- *public String getResource()*
  Devuelve el recurso solicitado en la petición HTTP.
- *public Hashtable getHeaders()*
  Devuelve las cabeceras HTTP especificadas en la petición HTTP.
- *boolean Analize_Request()*
  Analiza el método, recurso y versión empleado por el cliente HTTP.
- *boolean Read_Request()*
  Lee la petición del flujo de datos entrante.

▪ HTTPResponse

Lleva a cabo la creación de una respuesta HTTP 1.0, creando las cabeceras de respuesta pertinentes e incluyendo en su caso el recurso solicitado por el usuario.

- *public HTTPResponse(OutputStream os)*
  Crea un objeto encargado de responder a una petición HTTP.
- *public Hashtable getHeaders()*
  Devuelve las cabeceras generadas por el servidor al atender una petición HTTP de un cliente.
- *public void setHeader(String key,String value)*
  Establece una nueva cabecera de respuesta.
- *public void setCode(String code,String message)*
  Establece el código de respuesta resultado de atender una petición.
- *public void setResource(String folder,String file)*
  Establece el directorio y el archivo del recurso solicitado[7].

[7]Ver información relativa al Sistema de Archivos.

- *public String getHeader(String key)*
  Devuelve el valor de la cabecera especificada por parámetro.
- *protected void initHeaders()*
  Establece las cabeceras comunes a cualquier respuesta. Estas cabeceras son *Server*, *Connection*, *Language* y *Cache-Control*.
- *public void write()*
  Escribe la respueta HTTP en la conexión del cliente.

- Configure

  Contiene la configuración del servidor HTTP obtenida a través de un fichero de texto incluido junto al servidor. Esta clase proporciona los métodos necesarios para consultar información relativa a la situación actual del servidor así como sobre los tipos MIME soportados.

  - *private Configure()*
    Crea un objeto configuración. El constructor es privado para que sólo se pueda existir un solo objeto configuración en el sistema[8].
    Es a través del método estático *getInstance* la única manera de obtener una instancia de esta clase.
  - *public static Configure getInstance(String path)*
    Devuelve un objeto Configure que contiene la información de configuración del fichero indicado en el parámetro path.
  - *public static Configure getInstance()*
    Devuelve un objeto Configure estableciendo una ruta por defecto para el fichero de configuración.
  - *public static boolean loadConfig(String path)*
    Realiza la carga del fichero de configuración. Es necesario llamar a este método antes de cualquier llamada a los métodos mostrados a continuación.
  - *public String getServer()*
    Devuelve el nombre del servidor HTTP.
  - *public String getDefaultPort()*
    Devuelve el puerto por defecto en el que se lanzará el servidor.
  - *public int getMaxProcessors()*
    Devuelve el número máximo de procesos simultáneos que el servidor atenderá.

[8] Ver patrón Singleton.

- *public long getConnTimeOut()*
  Devuelve el tiempo máximo de inactividad.
- *public Hashtable getMimes()*
  Devuelve los tipos MIME establecidos en el fichero de configuración.

D.7.2. Clases de la serialización

- ObjInputStream

  Representa un objeto serializado del cual se van a poder leer los diferentes objetos que éste alberga.

  Esta clase está asociada a un flujo de bytes que es realmente la información y hereda de la clase *DataInputStream*, con lo que todos sus métodos estarán disponibles en esta clase.

  - *public ObjInputStream(InputStream in) throws IOException*
    Constructor de la clase. Crea una nueva instancia que contendrá un objeto serializado ubicado en el flujo de entrada recibido por parámetro.
  - *public synchronized Object readObject() throws IOException*
    Reconstruye y devuelve deserializado el objeto almacenado en el flujo de entrada. Este método no conoce realmente la estructura interna de cada tipo de objeto serializado, ya que de esto se encargan las propias clases.

    La funcionalidad de este método es crear en primer lugar una instancia del tipo del objeto almacenado internamente para a continuación llamar a los métodos de deserialización de cada clase.

    Finalmente devuelve un objeto reconstruido a partir de un flujo de bytes.

- ObjOutputStream

  Esta clase se va a encargar de serializar los objetos recibidos en su método *writeObject* en el flujo de bytes obtenido en el constructor.

  Esta clase se apoya en los métodos de *DataOutputStream*, clase de la cual hereda.

  A medida que se realicen llamadas a su método *writeObject* los diferentes objetos[9] se irán convirtiendo en un flujo lineal de bytes con

[9]Los objetos recibidos en este método deben implementar el interfaz *Serializable*.

una estructura predefinida mediante la cual podrán ser posteriormente reconstruidos a través de la clase *ObjInputStream*.

- *public void ObjOutputStream(OutputStream out) throws IOException*

  Constructor de la clase. Recibe un flujo de bytes de salida en el cual podrá escribir el resultado de serializar los objetos recibidos el método *writeObject*.

- *public void writeObject(Serializable object) throws IOException*

  Serializa el objeto recibido por parámetro mediante un algoritmo de serialización predefinido.

  Este método desconoce cómo ha de serializarse cada objetos, ya que esto es propio de cada clase que implementa el interfaz *Serializable*.

- VectorS

  Debido a que J2ME no soporta la serialización y consecuentemente no incluye ninguna clase serializable se han incluido a modo de ejemplo la implementación de dos clases que sí la soportan.

  Las clases VectorS y Configure mediante sus respectivos métodos *writeObject* y *readObject* son capaces de serializarse y deserializarse tal y como lo harían las clases serializables de J2SE.

  La clase VectorS representa un vector que puede almacenar cualquier tipo de objetos básicos (String, Int, Boolean, etc ...) así como cualquier tipo de objeto complejo siempre que éste sea serializable.

  - *public VectorS()*

    Crea un vector serializable.

  - *public void writeObject(ObjOutputStream out) throws IOException*

    Escribe todos los elementos del vector en un flujo de bytes.

  - *public void readObject(ObjInputStream in) throws IOException*

    Lee un vector serializado de un flujo de bytes para inicializarse a sí mismo.

- Config

  Esta clase es el segundo ejemplo de serialización incluido. Representa una configuración de un sistema así como las operaciones de escribirse y leerse en y de un flujo de bytes.

- *public Config()*
  Constructor de la clase.
- *public void setValor(int v,String c)*
  Establece los dos valores de la configuración.
- *public void imprimir()*
  Muestra por pantalla el valor de la configuración.
- *public void writeObject(ObjOutputStream out) throws IOException*
  Serializa la configuración.
- *public void readObject(ObjInputStream in) throws IOException*
  Inicializa la configuración a partir del resultado obtenido mediante la deserialización del flujo de entrada obtenido mediante parámetro.

D.7.3. Clases de ejemplo

Con el fin de probar tanto la serialización como el servidor HTTP implementado en este proyecto, se han incluido dos MIDlets que llevan a cabo sendas tareas.

El primer MIDlet, es el de un servidor HTTP que proporcionará acceso HTTP a los recursos que se le han incluido.

El segundo ejemplo constituye la serialización/deserialización de un objeto compuesto por vectores, cadenas, etc ... con el fin de comprobar el correcto funcionamiento de todo el proceso.

- Servidor

  Representa un servidor HTTP a la escucha en el puerto especificado. Inicialmente se incluyen una serie de recursos a través de la clase *FileSystem* para seguidamente lanzar el proceso de tratamiento de peticiones HTTP del servidor.

  - *public void startApp() throws MIDletStateChangeException*
    Añade una serie de recursos al sistema y lanza la ejecución del servidor HTTP.
  - *public void pauseApp()*
    Informa al MIDlet que entre en estado de pausa.
  - *public void destroyApp(boolean unconditional)*
    Informa al MIDlet de que finalice su ejecución y entre en el estado destruido.

- Serial

  Este MIDlet es un sencillo ejemplo de cómo una clase se puede inicializar a partir del resultado de la deserialización de otra.

  - *public void startApp() throws MIDletStateChangeException*
    Crea dos objetos serializables, inicializa el primer de ellos y lo serializa. A continuación iguala el segundo de los objetos con el resultado de deserializar el primero de ellos mostrando el resultado por pantalla.
  - *public void pauseApp()*
    Informa al MIDlet que entre en estado de pausa.
  - *public void destroyApp(boolean unconditional)*
    Informa al MIDlet de que finalice su ejecución y entre en el estado destruido.

D.8. Inclusión de recursos en el sistema

Con la finalidad de poder incluir recursos al sistema se han implementado dos clases que conjuntamente realizan las funcionalidades de un sistema de archivos básico.

En primer lugar está la clase *ResourceLoader* que permite especificar el nombre de un recurso del sistema al que se quiere acceder (archivos de imágenes, sonidos, etc ....) para obtener dicho recurso como un array de bytes.

La llamada necesaria para cargar un recursos del sistema es la siguiente:

- *byte[] ResourceLoader.loadResource(String path)*

  El método *loadResource* se ha establecido como *static* y devuelve en forma de array de bytes el recurso indicado en path o null en caso de no encontrarse.

Todos los recursos que se desea poder incluir dentro del MIDlet han de introducirse en el directorio *res/icons* de la carpeta de desarrollo[10].

Debido a que en MIDP no existe una estructura de archivos y carpetas como en la mayoría de los sistemas de archivos, ha sido necesario crear una clase denominada *FileSystem* para proporcionar una estructura basada en archivos y carpetas sobre RMS. Este sistema de archivos define e implementa los métodos de búsqueda y creación de carpetas y archivos[11], búsquedas, recuperaciones, borrados así como tratamiento de permisos sobre los recursos.

Las funciones de manejo básicas del sistema de archivos son las siguientes:

- *FileSystem FileSystem.getInstance(String type)*

  Proporciona una referencia al **único** sistema de archivos existente en la aplicación[12]. Mediante el parámetro *type* se puede seleccionar el tipo de sistema de archivos deseado, actualmente sólo existe el implementado sobre RMS, aunque en futuras ampliaciones no se descarta incluir otros como el basado en XML.

  Mediante la referencia obtenida al sistema de archivos se podrán realizar todas las operaciones disponibles descritas a continuación.

[10] Esto es requisito únicamente en el J2ME Wireless Toolkit, en entornos como el IBM WebSphere basta con indicar la localización del recurso a incluir.

[11] Es en la inserción de nuevos archivos al sistema donde entra la clase ResourceLoader, que cargará un determinado recurso del sistema para insertarlo en el sistema de archivos interno de la aplicación.

[12] La clase FileSystem se ha implementado siguiendo el patrón de diseño **Singleton** que únicamente permite una intancia de la clase en el sistema.

- *FileSystem FileSystem.getInstance()*

  Al igual que en la situación anterior se obtiene una referencia al sistema de archivos establecido, pero esta vez al no especificar el tipo se obtendrá el establecido por defecto (en la situación actual al existir uno sólo se obtendrá igualmente el de tipo RMS).

- *boolean existsFolder(String folder)*

  Indica si la carpeta especificada por parámetro existe.

- *int existsFile(String folder,String file)*

  Informa sobre la existencia de un determinado archivo en una carpeta devolviendo un código positivo en caso de que exista y cero de lo contrario.

- *int existsFile(String folder,String file,String access)*

  Realiza la misma función que el método anterior con la salvedad de que esta vez se le indica el tipo de acceso de la entidad que busca el fichero. El resultado es el mismo, salvo que en caso de no tener acceso al archivo devolverá un código negativo.

- *boolean addFolder(String folder)*

  Añade una carpeta en caso de que ésta no exista (resultado true), de lo contrario el código devuelto será false.

- *int addFile(String folder,String file,String permission,byte[] content)*

  Incluye el archivo especificado por file y cuyos datos se encuentran en la variable content dentro de la carpeta folder. Mediante permission se especifica el tipo de permiso que tiene el archivo[13].

- *boolean delFolder(String folder)*

  Elimina una carpeta, obteniendo true en caso de que ésta exista y false de lo contrario.

- *boolean delFile(String folder,String file)*

  Elimina un determinado fichero perteneciente a la carpeta folder, los códigos retornados son los mismos que para la eliminación de una carpeta.

[13]La gestión de los permisos así como su granularidad es una cosa que queda totalmente abierta a futuras modificaciones, habiéndose establecido en esta primera versión permisos "r" para acceso público y "w" para acceso restringido.

- *byte[] getFile(String folder,String file,String access)*

  Obtiene el fichero indicado siendo access el permiso del solicitante.

A continuación se incluye un ejemplo de inicialización de un sistema de archivos así como de la inclusión de algunos recursos en el sistema.

```
FileSystem fs = FileSystem.getInstance("rms");

fs.addFolder("/docs/");
fs.addFolder("/images/");

fs.addFile("/docs/","index.html","r",ResourceLoader.
  loadResource("/icons/index.html"));

fs.addFile("/docs/","ayuda.html","r",ResourceLoader.
  loadResource("/icons/ayuda.html"));

fs.addFile("/images/","Duke.png","r",ResourceLoader.
  loadResource("/icons/Duke.png"));

fs.addFile("/images/","Java.png","r",ResourceLoader.
  loadResource("/icons/Java.png"));
```

Como se puede observar en el ejemplo inicialmente se realiza la llamada al método *getInstance* para obtener la referencia al sistema de archivos, seguidamente se crean dos carpetas (*/docs/* e */images/*) para finalmente incluir los archivos *index.html* y *ayuda.html* dentro de la carpeta */docs/* y *Duke.png* y *Java.png* en */images/*.

Debido a que los archivos que se quieren incluir no se encuentran dentro del MIDlet (son externos) se recurre a la clase ResourceLoader, que dada una ruta devolverá el contenido en bytes de ésta.

Si se quisiese eliminar cualquiera de las carpetas existentes bastaría con escribir:

```
fs.delFolder("/docs/");
```

o

```
fs.delFolder("/images/");
```

Apéndice E

Distribución

A continuación se detalla la estructura de directorios incluida en el CD-ROM del proyecto.

E.1. Estructura de Directorios

La estructura de directorios es la siguiente:

- **Fuentes** En esta carpeta se encuentran los fuentes correspondientes al servidor HTTP sobre MIDP, al mecanismo de serialización así como los códigos fuente del CLDC Y MIDP para Sun OS.

  En la carpeta Apps se incluye la estructura completa de fuentes, recursos, clases, etc ... creada por el Wireless Toolkit.

  - *Servidor*
  - *Serial*
  - *CLDC-MIDP*

- **Apps** Contiene la estructura completa generada automáticamente por el Wireless Toolkit. Copiando cada una de los subdirectorios incluidos (Servidor y Serial) en el directorio apps del Wireless Toolkit se podría tanto compilar como ejecutar las aplicaciones correspondientes.

  - *Servidor*
  - *Serial*

- **Recursos**

  Incluye las páginas web de prueba incluidas con sus correspondientes imágenes así como varios objetos serializados.

- **Ejecutables**

  En esta carpeta se incluyen los .JAR generados (carpeta WTK) así como su conversión al formato PRC del sistema operativo de Palm (Palm OS).

  - *Palm*
  - *WTK*

- **Documentación**

  La documentación empleada en el desarrollo de este proyecto se incluye en esta carpeta clasificada por su categoría así como por su materia.

  - *Artículos*
  - *Capítulos libros*
  - *HTTP*
  - *RMI*
  - *Especificaciones*
  - *Otros*

- **Memoria**

  Incluye el archivo fuente de Latex así como su conversión a pdf. Todas las imágenes empleadas en la memoria están incluidas.

- **Software**

  Software empleado en el desarrollo de este proyecto así como alguna aplicación considerada de utilidad.

  - *J9*

    Entorno de desarrollo de IBM para multitud de plataformas. Cabe destacar que incluye una máquina virtual para Palm OS (J9) con soporte de sockets así como la posibilidad de compilar los programas J2ME para múltiples plataformas.
  - *Wireless Toolkit*

    Kit de desarrollo empleado en el proyecto.
  - *Java 1.3*

    Versión de Java necesaria para el correcto funcionamiento del WTK.

- *Forte for Java*

  Herramienta opcional a la hora de desarrollar aplicaciones J2ME. Puede integrarse con WTK ofreciendo un entorno completo de edición, compilación y ejecución de MIDlets.

- *Rational Rose Real Time*

  Herramienta de Rational que permite la generación automática de múltiples diagramas para aplicaciones embebidas.

- **Readme.txt**

  Fichero con la descripción y el contenido de este CD-ROM.

Apéndice F

Glosario de términos

AMS Application Management Software

API Application Programming Interface

CDC Connected Device Configuration

cHTML Compact HyperText Markup Language

CLDC Connected Limited Device Configuration

CVM C Virtual Machine

DA Application Discovery

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

I-MODE Internet Mode

KB Kilo Byte

J2EE Java 2 Enterprise Edition

J2ME Java 2 Micro Edition

J2SE Java 2 Standard Edition

JAD Java Application Descriptor

JAM Java Application Management

JAR Java Archive

JNI Java Native Interface

KVM K Virtual Machine

MB Mega Byte

Mhz Mega Hertzio

MIDlet Mobile Information Device

MIDlet Suite Mobile Information Device Suite

MIDP Mobile Information Device Profile

MIME Multipurpose Internet Mail Extensions

MGC Marco Genérico Conexión

OTA On The Air

PalmOS Palm Operating System

PC Personal Computer

PDA Personal Digital Assistant

PNG Portable Network Graphic

RAM Random Access Memory

RMI Remote Method Invocation

RMS Record Management System

ROM Read Only Memory

RPC Remote Procedure Call

SMS Short Message Service

TCK Compatibility Test Kit

URL Uniform Resource Locator

WAP Wireless Access Protocol

WTK Wireless ToolKit

WWW World Wide Web

XDR eXternal Data Representation

XML eXtensible Markup Language

Bibliografía

[1] T. Baustista. *Una Descripción de LaTex2*, Feb. 1996. Manual de Latex bastante sencillo.

[2] E. Davids. Java object serialization in parsable ascii, 1998. AUUG-Vic/CAUUG: Summer Chapter Technical Conference 98.

[3] B. Day. Developing wireless applications using java technology, 2001. Transparencias resumen de J2ME.

[4] R. Dragomirescu. Dynamic classloading in the kvm, Apr. 2000.

[5] B. Eckel. *Thinking in Java*. IC, 1998.

[6] E. Giguere. Making connections with the cldc, Feb. 2001. http://www.ericgiguere.com.

[7] E. Giguere. Animating images in midp, June 2002.

[8] E. Giguere. Simple store and forward messaging for midp applications, Mar. 2002. http://www.ericgiguere.com.

[9] W. Grosso. Serialization, Oct. 2001. http://www.onjava.com.

[10] S. Halloway. Serialization in the real world, Feb. 2000. http://developer.java.sun.com/developer/TechTips/2002/tt0229.html.

[11] P. R. Jan-Peter Stromann, Stephan Hartwig. Wireless microservers. *Pervasive Computing*, pages 58–69, Apr.-June 2002.

[12] A. Kaminsky. Jinime: Jini connection technology for mobile devices, Aug. 2000. Rochester Institute of Technology.

[13] J. Knudsen. Mobile media api overview, June 2002.

[14] B. Kurniawan. Learning polymorphism and object serialization, Aug. 2001. http://www.oreillynet.com.

[15] T. Letsch. Redesign and Implementation of a Mobile Agent System compliant with the MAFFinder part of MASIF standard. Master's thesis, Technische Universitat Munchen. Institut fur Informatik, 2000.

[16] G. H. Mahmoud. Secure java midp programming using https, June 2002.

[17] Q. Mahmoud. Midp network programming using http and the connection framework. page 11, Nov. 2000.

[18] Q. H. Mahmoud. Tranporting objects over sockets, Dec. 2002. http://developer.java.sun.com.

[19] G. McCluskey. Using java reflection, Jan. 1998. http://developer.java.sun.com.

[20] S. Microsystems. System architecture, 1997. Descripción detallada de todo el proceso de serialización/deserialización en J2SE.

[21] S. Microsystems. Applications for mobile information devices, 2000. White Paper.

[22] S. Microsystems. Connected, limited device configuration.specification version 1.0a, May 2000.

[23] S. Microsystems. Java 2, micro edition, wireless toolkit user's guide, Nov. 2000.

[24] S. Microsystems. Java2 platform micro edition technology for creating mobile devices, May 2000.

[25] S. Microsystems. Building the mobile information device profile, 2001. http://www.sun.com.

[26] S. Microsystems. Mobile information device profile build configuration, 2001.

[27] S. Microsystems. Over the air user initiated provisioning recommended practice for mobile information device profile, May 2001. Version 1.0.

[28] S. Microsystems. Running the mobile information device profile, 2001.

[29] S. Microsystems. Java technology and the new world of wireless portals and mcommerce, Feb. 2002.

[30] S. Microsystems. The k virtual machine datasheet, 2002.

[31] S. MicroSystems. Validatin deserialized objects, Mar. 2002. http://java.sun.com/j2se/1.4/docs/guide/serialization.

[32] E. Miedes. Serialización de objetos en java.

[33] M. Morrison. *Wireless Java with J2ME*. SAMS, 2001. Libro de aprendizaje.

[34] J. W. Muchow. Midlet packaging with j2me, Apr. 2001. http://www.onjava.com/lpg/a/793.

[35] E. Ort. Opening a socket connection and midp, Jan. 2000. http://wireless.java.sun.com/midp/questions/rawsocket.

[36] E. paquete java.io. Leo suarez, July 2001. http://www.javahispano.com.

[37] G. P. Picco. Mobile agents: an introduction, Feb. 2001.

[38] K. Topley. The mobile information device profile and midlets, 2000.

[39] M. C. C. Vázquez. Agentes móviles en computación ubicua, 2001.