Capítulo 2

J2ME

2.1 Introducción

El cliente desarrollado en este proyecto está basado en la tecnología J2ME para dispositivos móviles, como pueden ser: teléfonos móviles, PDAs o Palms, o incluso cualquier otro tipo de dispositivo móvil que pudiese ser diseñado en un futuro próximo.

Este cliente se conectará con un servicio Web el cual le facilitará una serie datos de utilidad para el usuario, datos que serán enviados mediante el protocolo de comunicación SOAP.

2.2 Java 2 Platform Micro Edition (J2ME™)

Actualmente Sun Microsystems ha agrupado la tecnología Java en tres tecnologías claramente diferenciadas, cada una de ellas adaptada a un área específica de la industria:

- ***Java 2 Platform, Enterprise Edition (J2EE™)*** pensada para servir las necesidades que puedan tener las empresas que quieran ofrecer servicios a sus clientes, proveedores y empleados.

- ***Java 2 Platform, Standard Edition (J2SE™)*** pensada para satisfacer las necesidades de usuarios y programadores en sus equipos personales y estaciones de trabajo.

- ***Java 2 Micro Edition (J2ME™)*** enfocada tanto para productores de dispositivos portátiles de consumo como para quienes proporcionan servicios de información disponibles para estos dispositivos.

Cada una de estas plataformas define en su interior un conjunto de tecnologías que pueden ser utilizadas con un producto en particular:

- ***Java Virtual Machine*** que encuadra en su interior un amplio rango de equipos de computación.

- Librerías y **APIs** especializadas para cada tipo de dispositivo.

- Herramientas para desarrollo y configuración de equipos.

J2ME abarca un espacio de consumo en rápido crecimiento, que cubre un amplio rango de dispositivos, desde pequeños dispositivos de mano hasta incluso televisores. Todo esto siempre manteniendo las cualidades por las cuales la tecnología Java ha sido mundialmente reconocida: consistencia entre los distintos productos Java, portabilidad de código entre equipos, gratuidad de sus productos y escalabilidad entre productos.

La idea principal de J2ME es proporcionar aplicaciones de desarrollo sencillas, que permitan al programador desarrollar aplicaciones de usuario para el consumidor de dispositivos móviles y portátiles. De esta forma se abre un amplio mercado para todas aquellas empresas que deseen cubrir las necesidades que los usuarios demandan en sus dispositivos móviles, tales como teléfonos móviles o agendas personales.

J2ME está orientada a dos categorías muy concretas de productos, como son:

- Dispositivos de información compartidos y conectados de forma permanente. Esta categoría se conoce con la denominación **CDC (Connected Device Configuration)**. Ejemplos típicos de estos son televisores, teléfonos conectados a Internet y sistemas de navegación y reentretenimiento para el automóvil. Estos dispositivos se caracterizan por tener un amplio rango de interfaces de usuario, conexión permanente a Internet de banda ancha de tipo TCP/IP y unos rangos de capacidad de memoria entre 2 y 16 MB.

- Dispositivos de información personales móviles. Categoría conocida como **CLDC (Connected, Limited Device Configuration)**. De entre todos estos los más representativos son los teléfonos móviles y las agendas personales PDAs. Caracterizados por disponer de interfaces simples, conexión no permanente a Internet, de menor ancho de banda, normalmente no basada en TCP/IP y con rangos de capacidad de memoria muy reducidos, entre 128 y 512 KB.

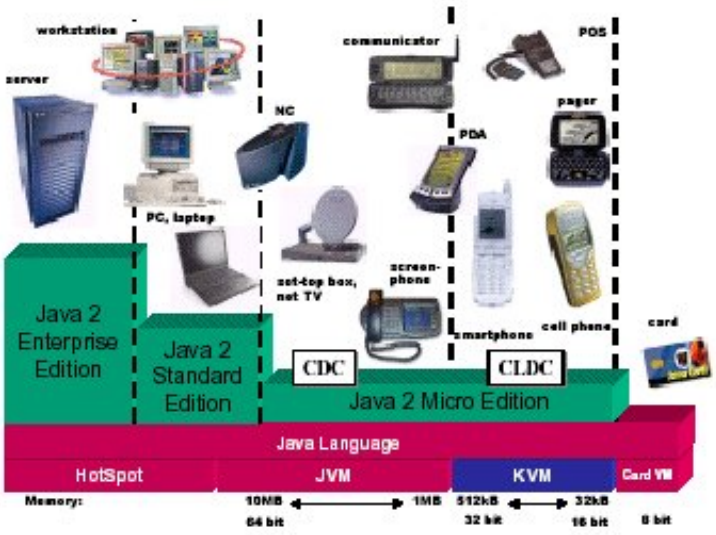

Figura 2.1: Dispositivos Java y sus ediciones

Claro está con el paso de los años, la línea fronteriza que separa ambos grupos es cada vez más fina y cada vez está mas difuminada lo cual, a medida que la tecnología avanza y van apareciendo nuevos tipos de dispositivos y van evolucionando los ya existentes va potenciándose aún más. De esta forma hoy ya se "confunden" los ordenadores y los dispositivos de comunicaciones y cada vez más se va haciendo más uso de conexiones sin cables, lo cual nos lleva a realizar en la práctica una agrupación de equipos basándonos únicamente en sus capacidades de memoria, sus consumos de batería y el tamaño de la pantalla.

2.3 Arquitectura J2ME

Aunque los dispositivos mencionados tales como teléfonos móviles, agendas personales (PDAs) o televisores tienen muchos aspectos en común, también son muy diferentes en cuanto a forma y función. Estos contarán con diferentes configuraciones hardware, diferentes modos de uso (uso de teclados, voz, etc.), diferentes aplicaciones y características software, así como todo un amplio rango de futuras necesidades a cubrir. Para considerar esta diversidad la arquitectura J2ME está diseñada de forma *modular y extensible* de tal forma que se de cabida a esta extensa variedad de dispositivos así como a aquellos que sean desarrollados en un futuro.

La arquitectura J2ME tiene en cuenta todas aquellas consideraciones relativas a los dispositivos sobre los que tendrá que trabajar. De esta forma, son tres los conceptos básicos en los que se fundamenta:

- **KVM (K Virtual Machine):** Ya que el mercado de venta de los dispositivos a los que se refiere J2ME es tan variado y tan heterogéneo, existiendo un gran número de fabricantes con distintos equipos hardware y distintas filosofías de trabajo, J2ME proporciona una máquina virtual Java completamente optimizada que permitirá trabajar con diferentes tipos de procesadores y memorias comúnmente utilizados.

- **Configuración y Perfil*:*** Como los dispositivos sobre los que trabaja **J2ME** son tan reducidos en lo que se refiere a potencia de cálculo y capacidad de memoria J2ME proporciona una máquina virtual Java muy reducida que permite solo aquellas funciones esenciales y necesarias para el funcionamiento del equipo. Además, como los fabricantes diseñan distintas características en sus equipos y desarrollan continuos cambios y mejoras en sus aplicaciones estas configuraciones tan reducidas deben poder ser ampliadas con librerías adicionales. Para conseguir esto se han considerado dos conceptos extremadamente importantes que son las Configuraciones y los Perfiles. El objetivo de realizar esta distinción entre Perfiles y Configuraciones es el de preservar una de las principales características de Java, la portabilidad.

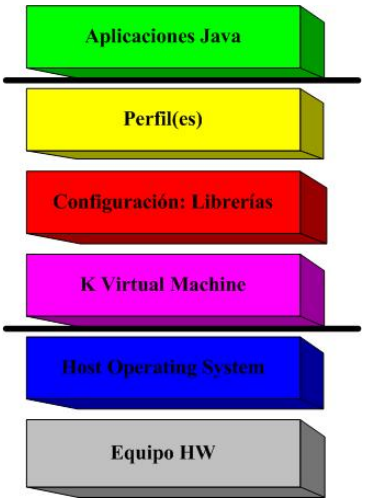

Figura 2.2: Capas en J2ME

2.3.1 K Virtual Machine

La tecnología KVM define una máquina virtual Java específicamente pensada para su funcionamiento en dispositivos de pequeño tamaño y de características muy reducidas y limitadas. El objetivo de esta tecnología fue el de crear la mínima máquina virtual posible, pero que mantuviese los aspectos fundamentales del lenguaje Java todo ello funcionando en un dispositivo con una capacidad de memoria de tan solo unos cuantos centenares de KB.

KVM puede trabajar con microprocesadores de 16/32 bits tales como teléfonos móviles, PDAs, equipos de audio/vídeo portátiles, etc.

La mínima cantidad ideal de memoria necesaria para la KVM es de 128 KB en los cuales se incluyen la propia máquina virtual, un mínimo número de librerías y espacio libre para las aplicaciones. A pesar de esto, una implementación típica real necesita de unos 256 KB de los que aproximadamente la mitad es para las aplicaciones, de 60 a 80 KB es para la máquina virtual y el resto para las librerías.

2.3.2 Configuraciones

Una Configuración J2ME define una plataforma mínima para una categoría horizontal de dispositivos con similares características de memoria y procesamiento. A su vez, proporciona una definición completa de máquina virtual y el conjunto mínimo de clases Java que todo dispositivo o aplicación debería tener. Más concretamente, una configuración especifica:

- Las características del lenguaje Java soportadas por el dispositivo.
- Las características soportadas por la máquina virtual Java.
- Las librerías y APIs Java soportadas.

En un entorno J2ME una aplicación es desarrollada para un Perfil en particular, el cual se basa y extiende una Configuración en particular. De esta forma una Configuración define una plataforma común tanto para la fabricación de dispositivos como para el posterior desarrollo de aplicaciones sobre estos.

Por tanto, todos los dispositivos que se encuadren dentro de una Configuración concreta deben cumplir todas las características de ésta, y todas las aplicaciones que corran sobre estos dispositivos deben cumplir las restricciones del Perfil que se monta sobre esta Configuración.

El objetivo de esto es evitar la fragmentación y limitar en lo posible el número de Configuraciones desarrolladas por los fabricantes. Concretamente, sólo existen dos Configuraciones estándar permitidas, que son:

- **CLDC (Connected Limited Device Configuration)** abarca el conjunto de los dispositivos móviles personales móviles, tales como teléfonos móviles, PDAs, etc. Implementa una serie de librerías y APIs que no se encuentran en J2SE y que son específicas para este tipo de dispositivos.

- **CDC (Connected Device Configuration)** que abarca fundamentalmente el conjunto de dispositivos de información compartida, fijos y de conexión permanente, tales como televisores, terminales de comunicación, etc. Ésta incluye un conjunto de librerías mucho mayor que el del anterior, siendo CLDC un subconjunto de ésta.

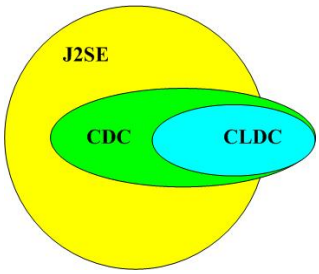

Figura 2.3: Relación J2SE-J2ME

La mayoría de las funcionalidades de CLDC y CDC son heredadas de J2SE, de forma que toda clase perteneciente a CDC y CLDC debe ser exactamente igual a su correspondiente en J2SE o bien un subconjunto de ésta. Pero además ambas configuraciones pueden añadir características que se encuentren en J2SE y que sean específicas del dispositivo en sí.

Las tecnologías CLDC y KVM están íntimamente relacionadas, ya que actualmente CLDC sólo está disponible para KVM y KVM solo soporta CLDC. Esta situación se prevé que cambiará cuando el proceso de desarrollo de J2ME está más avanzado.

2.3.3 Perfiles

Un Perfil de dispositivo es una capa definida sobre una Configuración concreta, de forma que dicho Perfil extienda las posibilidades de dicha Configuración. Un Perfil está pensado para garantizar la interoperabilidad de una familia vertical de dispositivos. Dicho perfil se compone de una serie de librerías de clases más específicas del dispositivo que las de la Configuración.

Para un dispositivo será posible soportar varios tipos de Perfiles. Algunos de estos serán específicos del propio dispositivo y otros serán específicos de alguna aplicación que corra sobre el dispositivo. Las aplicaciones son diseñadas para un Perfil concreto y solo podrán funcionar en él y no en otro perfil, ya que harán uso de las funcionalidades propias de éste. Según esto podemos considerar que un Perfil no es más que un conjunto de librerías de clases que proporciona funcionalidades adicionales a las de las Configuración que reside debajo de éste.

Actualmente el único Perfil que existe y está en funcionamiento es MIDP, el cual está pensado para teléfonos móviles.

2.3.4 Capas altas

Sobre las capas anteriores se implementan las distintas aplicaciones que corren sobre el dispositivo, éstas pueden ser de tres tipos:

- Una **aplicación MIDP** o también llamada MIDlet es una aplicación que solamente hace uso de librerías Java definidas por las especificaciones CLDC y MIDP. En este tipo de aplicaciones está enfocado el desarrollo de la especificación MIDP y por tanto se espera será la más usada.

- Una **aplicación OEM-specific** es aquella que no utiliza clases propias de MIDP, sino que utiliza clases de la especificación OEM las cuales normalmente no son portables de un equipo a otro, ya que son clases propias de cada fabricante.

- Una **aplicación Nativa** es aquella que no está escrita en lenguaje Java y va montada sobre el software nativo del propio equipo.

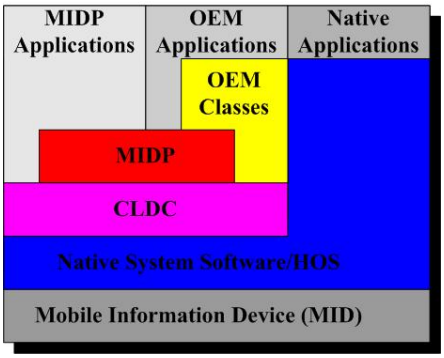

Figura 2.4: Arquitectura de alto nivel

2.4 Seguridad

Tanto empresas como usuarios individuales dependen cada vez más de información crítica almacenada en computadoras y redes, por lo que el uso de sistemas y aplicaciones de seguridad se ha convertido en algo extremadamente importante, y como no esto es aún más importante en el ámbito de dispositivos móviles y redes inalámbricas.

La plataforma de desarrollo de Java es muy útil en estos aspectos debido a su inherente arquitectura de seguridad. En la plataforma J2SE el modelo de seguridad proporciona a los desarrolladores de aplicaciones funcionalidades con las que crear distintas políticas de acceso y articular permisos independientes para cada usuario, manteniendo todo esto transparente al usuario.

Desafortunadamente la cantidad de código con el que se consigue todo esto y que se encuentra en la plataforma J2SE excede en mucho las capacidades de memoria de los dispositivos sobre los que trabaja J2ME, por lo que se hacen necesarias una serie de simplificaciones que reduzcan este código pero que mantengan cierta seguridad. El modelo de seguridad definido en CLDC en conjunto con MIDP se basa en tres niveles:

- **Seguridad de bajo nivel:** También conocida como **seguridad de la máquina virtual** asegura que un fichero de clase, o un fragmento de código malintencionado no afecte a la integridad de la información almacenada en el dispositivo móvil. Esto se consigue mediante el verificador de ficheros de clase el cual asegura que los bytecodes almacenados en el fichero de clase no contengan instrucciones ilegales, que ciertas instrucciones no se ejecuten en un orden no permitido y que no contengan referencias a partes de la memoria no válidas o que se encuentren fuera del rango de direccionamiento real. Debido a esto el estándar CLDC exige que bajo él se encuentre una KVM que realice estas operaciones de seguridad.

- **Seguridad de nivel de aplicación:** Este nivel de seguridad asegura que las aplicaciones que corren sobre el dispositivo solo puedan acceder a aquellas librerías, recursos del sistema y otros dispositivos que tanto el equipo como el entorno de aplicación permitan.

El verificador de ficheros de clase solo puede garantizar que la aplicación dada es un programa Java válido, pero nada más, por lo que hay una serie de aspectos que se escapan del control del verificador de clases, como es el acceso a recursos externos tales como ficheros de sistema, impresoras, dispositivos infrarrojos o la red.

Modelo Sandbox

En el modelo CLDC y MIDP el nivel de seguridad de aplicación se obtiene mediante un sandbox el cual asegura un entorno cerrado en el cual una aplicación solo puede acceder a aquellas librerías que han sido definidas por la Configuración, el Perfil y las clases específicas OEM del dispositivo. El sandbox asegura que la aplicación no se escape de él y acceda a recursos o funcionalidades no permitidas. Más concretamente el uso de un sandbox asegura:

- Los ficheros de clases han sido preperificados y son aplicaciones Java válidas.
- La Configuración, el Perfil y las clases OEM han determinado un conjunto de APIs válidas para la aplicación.
- La descarga y gestión de aplicaciones Java dentro del dispositivo en su lugar correcto de almacenamiento, no permitiéndose el uso de clases y aplicaciones de descarga diseñadas por el usuario que podrían dañar a las del sistema.
- El programador de aplicaciones no puede descargar al dispositivo librerías que contengan nuevas funcionalidades nativas o que accedan a funcionalidades nativas no permitidas.

Protección de clases de sistema

Una de las características del CLDC es la habilidad de soportar la descarga dinámica de aplicaciones a la máquina virtual del dispositivo. Por lo que podría existir una falta de seguridad ya que una aplicación podría no hacer caso de ciertas clases de sistema. CLDC y MIDP se encargan de que esto no ocurra.

Restricciones adicionales en la carga dinámica de clases

Existe una restricción de seguridad muy importante en la carga dinámica de clases que consiste en que por defecto una aplicación Java solo puede cargar clases pertenecientes a su Java Archive (JAR) file. Esta restricción asegura que las distintas aplicaciones que corran sobre un dispositivo no puedan interferir entre sí.

- **Seguridad punto a punto:** Garantiza que una transacción iniciada en un dispositivo móvil de información esté protegida a lo largo de todo el camino recorrido entre el dispositivo móvil y la entidad que provea el servicio. Este tipo de seguridad no está recogida en el CLDC o en el MIDP por lo que será siempre una solución propietaria del fabricante y del proveedor de servicios.

2.5 Objetivos, alcance y requisitos de J2ME

Desde el principio todo el diseño de la tecnología J2ME ha pretendido lograr una serie de objetivos muy concretos con el fin de facilitar un funcionamiento adecuado a los dispositivos sobre los que se va a montar. Además con el fin de lograr una cierta portabilidad ha sido necesario establecer una serie de requerimientos mínimos que deben cumplir los dispositivos sobre los que montar J2ME**.**

2.5.1 Objetivos

Todos los objetivos que la tecnología J2ME ha conseguido son ahora una serie de características que hacen que esta tecnología sea de gran interés, por ello es interesante echarles un vistazo.

- **Reparto dinámico de contenidos y aplicaciones Java**

Uno de los grandes beneficios que aporta la tecnología Java a estos dispositivos de reducidas prestaciones es el reparto dinámico y con total seguridad de servicios interactivos y aplicaciones sobre diferentes redes lo cual se consigue gracias al CLDC y el MIDP. Desde que se desarrollaron por primera vez los teléfonos móviles y hasta el día de hoy en el que el uso de Internet está mundialmente extendido, los fabricantes han intentado desarrollar dispositivos y paliaciones con mayores posibilidades de comunicación y con mayores facilidades en lo que concierne al desarrollo de aplicaciones, motivo que impulsó al desarrollo de esta tecnología.

- **Desarrollo de aplicaciones Third-party**

El enfoque de J2ME de reparto dinámico de aplicaciones que CLDC y MIDP aportan no solo favorece a los diseñadores y fabricantes de equipos y a sus programadores de aplicaciones, sino también a los diseñadores de aplicaciones third-party. Si asumimos que una vez que los dispositivos portátiles sean de uso de común e imprescindible para el usuario serán los desarrolladores third-party los que abarcarán todo el mercado de diseño de aplicaciones, J2ME en CLDC y MIDP ha incorporado una serie de librerías que favorezcan a estos third-party en sus diseños.

- **Independencia de las tecnologías estándar de comunicación**

Existe un amplio rango de tecnologías de comunicación inalámbrica actualmente en uso por todo el mundo, las cuales varían entre sí respecto de niveles de sofisticación,

compatibilidad e interoperabilidad. En la Segunda Generación (2G) destacan entre otras tecnologías tales como GSM, TDMA, CDMA, etc. y en la Tercera Generación (3G) ya se está empezando a implementar de forma práctica y cuya aplicación completa es inminente cuenta con WCDMA, CDMA2000 y TD.SCDMA. Además de estas está la tecnología 2,5G a caballo entre 2G y 3G que cuenta con GPRS, EDGE, CDPD, etc. Viendo el gran número de tecnologías existentes, cada una con sus características propias e incompatibles con las demás J2ME se ha esforzado en definir una serie de soluciones que se adapten a todas estas tecnologías de forma estándar, huyendo de aquellas APIs que solo puedan ser encuadradas en un tipo de tecnología, así como realizando soluciones que no solo se adapten a las características de ancho de banda de comunicación actual, sino también a las futuras características de comunicación de banda ancha.

- **Compatibilidad con otros estándares inalámbricos**

Una características muy interesante de J2ME es que no es solo una nueva tecnología inalámbrica que viene a aparecer para desbancar a las demás ya existentes, sino que además de poder hacer esto también puede ser utilizada para favorecer y mejorar a estas otras tecnologías, por ejemplo utilizando código Java para mejorar los navegadores Web que corren en dichas otras tecnologías. Así teléfonos móviles provistos de las tecnologías WAP o i-Mode pueden mejorar sus posibilidades gracias a J2ME.

2.5.2 Requisitos

Tanto el estándar CLDC como el MIDP exigen una serie de requisitos tanto hardware como software a cumplir por los dispositivos sobre los que se montan aplicaciones J2ME**.**

- **CLDC**

  Puesto que CLDC es consciente de la gran cantidad de dispositivos existentes en el mercado y de las grandes diferencias existentes entre estos, solo se imponen restricciones respecto de la capacidad de memoria, de forma que considerando que la KVM, las librerías de Configuración y de Perfil y las aplicaciones corriendo sobre el dispositivo solo deben ocupar entre 160 y 512 KB, se exige que al menos al menos 128 KB deben ser de memoria no volátil para la KVM y las librerías, y al menos 32 KB de memoria volátil deben ser para la KVM en funcionamiento.

  En lo concerniente al software ocurre algo parecido y por tanto debido a la gran diversidad de posibles software nativos que pueden correr en el dispositivo el CLDC solo exige la existencia de un software muy sencillo, tal que cuente con una entidad de control de programas de forma que pueda correr la KVM, no es necesario que de soporte para espacios de direccionamiento separados para cada proceso ni tampoco que de garantías de un buen funcionamiento en tiempo real.

- **MIDP**

Los requisitos exigidos por el MIDP son algo mas estrictos ya que este debe tratar con aspectos de presentación en pantalla etc., de esta forma se exige en memoria 128 KB de memoria no volátil para componentes MIDP, 8 KB de memoria no volátil para almacenamiento de datos de aplicación de forma persistente y 32 KB de memoria volátil para la KVM en ejecución. Los requisitos en el display del equipo son un tamaño de 96 x 54 mm^2, una profundidad de display de 1-bit y una relación de aspecto en el display de 1:1. La interfaz de entrada debe de tener uno o más de uno de los siguientes mecanismos: pantalla táctil y un teclado a una o dos manos. Por último en lo relativo a la red se pide una conexión bidireccional inalámbrica con un ancho de banda limitado y pudiendo ser intermitente.

En lo concerniente al software se exigen solamente varios puntos, la existencia de un kernel mínimo que controle al hardware y proporcione la entidad de control de programas anteriormente mencionada, un mecanismo de lectura/escritura de memoria no volátil para dar soporte a las APIs de almacenamiento de datos persistentemente, un mecanismo de lectura/escritura para dar soporte a las APIs de red, un mecanismo que de soporte de temporización, una capacidad mínima para escribir en la pantalla del dispositivo y un mecanismo para capturar la entrada de datos por parte del usuario.

2.5.3 Alcance

Basado en las decisiones tomadas en el JSR-30 group expert tanto CLDC como MIDP tienen una serie de alcances a los que pueden llegar.

- **CLDC**

La especificación del CLDC indica que los campos que este estándar cubre son los siguientes:

1. Lenguaje Java y características de la KVM
2. Librerías básicas: *java.lang.*, java.io.*, java.util.**.
3. Entrada/salida
4. Red
5. Seguridad

- **MIDP**

De todas las posibles funcionalidades que se pueden encontrar en un dispositivo el MIDP expert group decidió abarcar en MIDP solo aquellas que realmente aseguren la portabilidad:

1. Modelo de Aplicación: estructura que debe seguir una aplicación

2. Interfaz de usuario, tanto pantalla como entrada de datos.
3. Almacenamiento persistente de datos
4. Red
5. Temporización

2.6 CLDC (Connected Limited Device Configuration)

El objetivo del CLDC es el de definir una plataforma Java para dispositivos pequeños y de reducidas prestaciones tales como:

- De 160 a 512 KB de memoria
- Procesador de 16-32 bits
- Bajo consumo de la batería de alimentación
- Conexión a redes inalámbricas, de forma intermitente y de ancho de banda reducido (normalmente 9600 bps o menos)

Teléfonos móviles, PDAs, terminales de venta y muchos otros equipos más son los que entran en este grupo.

Todo lo que aquí se diga se ha extraído del documento CLDC Specification el cual está disponible en la Web de la Sun's Java Community Process (JCP).

2.6.1 CLDC Expert Group

La especificación del CLDC ha sido desarrollada por el JCP y el resultado se encuentra en la norma JSR-30, este grupo de fabricantes y desarrolladores de software está formado por las siguientes empresas:

- **America Online**
- **Bull**
- **Ericsson**
- **Fujitsu**
- **Matsushita**
- **Mitsubishi**
- **Motorola**
- **Nokia**
- **NTT DoCoMo**
- **Oracle**
- **Palm Computing**
- **Research In Motion**
- **Samsung**
- **Sharp**
- **Siemens**
- **Sony**
- **Sun Microsystems**
- **Symbian**

2.6.2 Modelo de Aplicación de CLDC

2.6.2.1 Arquitectura

Tal y como se comentó anteriormente la arquitectura que se sigue en J2ME se divide en tres capas que son el Host Operating System el cual se encarga de proporcionar la capacidad de controlar el hardware subyacente, sobre este está el CLDC el cual tiene en su interior la máquina virtual (KVM) y una serie de librerías con los que damos al móvil la posibilidad de trabajar con lenguaje Java corriendo sobre una máquina virtual Java, y encima de éste se encuentra el Perfil MIDP el cual aporta un conjunto de librerías que completan a las del CLDC.

Para el CLDC el concepto de aplicación Java se refiere a una colección de ficheros de clases Java los cuales contienen un único método public static void main(String[] args)que identifica el punto de lanzamiento de la aplicación. De esta forma la KVM arrancará la ejecución de la aplicación llamando a este método.

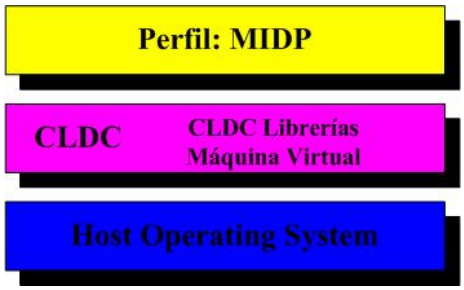

Figura 2.5: Capas software en J2ME

2.6.2.2 Gestor de aplicaciones

Con el fin de poder mantener almacenadas las aplicaciones dentro del dispositivo para posteriormente ser ejecutadas desde memoria existe el denominado gestor de aplicaciones que suele estar programado en lenguaje C o bien en cualquier lenguaje de bajo nivel que controle al hardware y que sea dependiente de él, nunca en lenguaje Java, ya que éste está fuera del alcance de J2ME, siendo tarea del propio dispositivo realizar estas operaciones. Este gestor de aplicaciones es el encargado de operaciones tales como:

- Descarga e instalación de aplicaciones Java
- Inspección de la existencia de aplicaciones Java almacenadas en el dispositivo
- Seleccionar y lanzar aplicaciones Java
- Borrar de memoria aplicaciones Java almacenadas en el equipo

2.6.3 Compatibilidad con el Java™ Language Specification

Debido a las fuertes restricciones de memoria que se dan en los dispositivos con los que trabaja J2ME, existen ciertas diferencias entre la maquina virtual que proporcionada CLDC y la máquina virtual definida en J2SE, diferencias tales como:

- **Inexistencia de punto flotante:** Debida a que la mayoría de dispositivos hardware subyacentes no soportan por sí mismos numeración en punto flotante, tendría que conseguirse ésta mediante el software, pero debido a las fuertes restricciones de capacidad de memoria el CLDC expert group decidió no implementarlo en el estándar.

- **Inexistencia de finalización:** En las librerías del CLDC no aparece el método *java.lang.Object.finalize* por lo que no hay soporte para finalización de instancias de clases.

- **Limitaciones en la captura de errores:** La máquina virtual del CLDC soporta el lanzamiento y captura de excepciones, no obstante existe una gran limitación en el conjunto de clases de errores permitidas respecto de las que se dan en J2SE, esto es debido a dos aspectos: por un lado que los propios dispositivos reaccionan distintamente a los errores (algunos sencillamente resetean mientras que otros intentan recuperarse), y segundo la implementación de la gran riqueza de lanzamientos y capturas de errores que se da en J2SE es extremadamente costoso en términos de memoria, por lo que supondría una gran cantidad de espacio en memoria.

2.6.4 Compatibilidad con Java™ Virtual Machine Specification

Tal y como se ha indicado en el apartado anterior existen ciertas diferencias entre la máquina virtual del estándar J2SE y la máquina virtual que proporciona el CLDC, por lo que se observan en esta última una serie de limitaciones a la hora de desarrollar aplicaciones.

- **Inexistencia de punto flotante**: Lo cual implica la inexistencia de una serie de bytecodes con los cuales se implementan los tipos de datos flotantes. De esta forma, toda clase y método de usuario debe cumplir las siguientes reglas:

    1. No usar los bytecodes prohibidos anteriormente citados.
    2. Ningún campo puede tener los tipos float, o double, o arrays, o arrays de arrays de estos tipos.
    3. Ningún método puede tener como argumento o como tipo devuelto float, o double, o arrays, o arrays de arrays de estos tipos.
    4. Ninguna constante puede ser de tipo float, o double**.**

- **Inexistencia de Java Native Interface (JNI):** La forma en que la máquina virtual del CLDC invoca una funcionalidad nativa es dependiente del dispositivo. El motivo por el que se ha eliminado esta funcionalidad es la falta de capacidad de memoria y que el CLDC asume que el conjunto de funciones nativas es cerrado por motivos de seguridad.

- **Inexistencia clases y programas diseñados por el usuario de carga de aplicaciones:** La propia máquina virtual diseñada por el CLDC tiene ya cargador de clases que por motivos de seguridad no se puede sobrescribir o sustituir por una solución usuaria.

- **Inexistencia de reflexión:** No existen en la máquina virtual del CLDC funcionalidades de reflexión las cuales permitirían a las aplicaciones Java inspeccionar el número y contenido de clases, objetos, métodos, campos, hilos y demás características de ejecución en el interior de la máquina virtual. De esta forma no será posible implementar Remote method invocation (RMI), serialización de objetos, JVMDI (Debbugging Interface), JVMPI (Profiler Interface), etc.

- **Inexistencia de Grupos de hilos e hilos demonio:** No se permite el uso de hilos demonio ni de grupos de hilos, solo podrán ser lanzados y parados uno a uno.

- **Inexistencia de finalización:** En las librerías del **CLDC** no aparece el método *java.lang.Object.finalize* por lo que no hay soporte para finalización de instancias de clases.

- **Limitaciones en la captura de errores:** La máquina virtual del CLDC soporta el lanzamiento y captura de excepciones, no obstante existe una gran limitación en el conjunto de clases de errores permitidas respecto de las que se dan en J2SE.

2.6.5 Verificación de ficheros de clase

Al igual que la máquina virtual de J2SE la máquina que implementa el CLDC debe ser capaz de detectar ficheros de clases no válidas, pero como las necesidades de memoria serían excesivas si implementásemos la solución que aporta J2SE se ha desarrollado una solución alternativa.

El espacio en memoria que se necesitaría si usásemos la solución aportada por J2SE sería de un mínimo de 50 KB solo para el verificador de clases, al menos de 20 a 100 KB de memoria RAM para ejecución y además la potencia de cálculo de la CPU debería ser bastante alta.

En cambio la solución aportada por CLDC solo toma 12 KB para la verificación y al menos 100 Bytes de memoria RAM para ejecución.

El nuevo verificador pensado para CLDC solamente realiza un escaneo lineal de bytecodes y se fundamenta en dos fases:

- Una primera fase en la que los ficheros de clases se hacen pasar por una herramienta de **preverificación** con el fin de completar dichos ficheros con información adicional que permita realizar la verificación a mayor velocidad y con menor carga de trabajo. Este proceso se realiza en la propia estación de trabajo en la que se esté diseñando la aplicación mediante emuladores de dispositivo.

- La segunda fase se da en ejecución, momento en el que el **verificador** en tiempo de ejecución de la máquina virtual utiliza la información adicional que el preverificador añadió para verificar por completo los ficheros de clase.

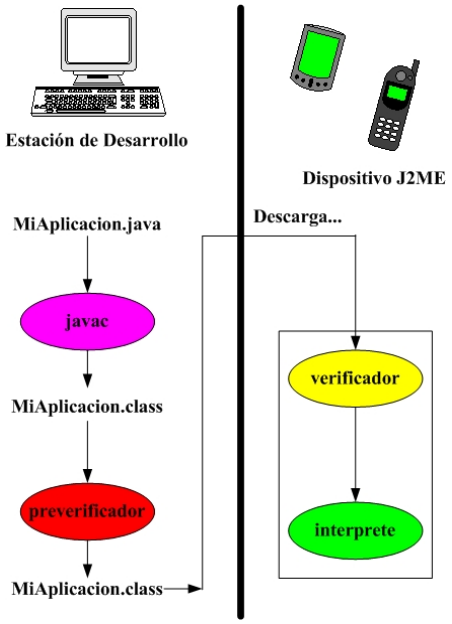

Figura 2.6: Verificación de clases

La interpretación de los bytecodes del fichero de clases solo podrá comenzar cuando el fichero de clases haya pasado correctamente la verificación.

Puesto que el preverificador añade a los ficheros de clases una serie de atributos adicionales podría pensarse que dichos ficheros tras sufrir el proceso de preverificación no son compatibles con el verificador del estándar J2SE, pero no es así ya que éste ignora automáticamente dichos atributos añadidos, por lo que existe compatibilidad con el estándar J2SE.

2.6.6 Librerías de CLDC

Tanto J2SE como J2EE cuentan con unas librerías de funciones muy ricas con funcionalidades que sacan todo el provecho de las estaciones servidoras o de trabajo

sobre las que se montan. Desafortunadamente en J2ME no ocurre lo mismo debido al escaso espacio de almacenamiento con el que cuentan los dispositivos.

Las librerías que se han desarrollado para J2ME cuentan con un número mínimo de funciones, pero debido al gran número de dispositivos existentes en el mercado todos ellos distintos entre sí es totalmente imposible en tan reducido espacio de almacenamiento contar con librerías que cubran todas las necesidades de todos estos dispositivos, aún así se han conseguido una serie de funcionalidades bastante útiles para todos los dispositivos.

Con el fin de asegurar compatibilidad entre las librerías de J2ME y de sus homólogos J2SE y J2EE la mayoría de las librerías del CLDC son un subconjunto de las que se dan en J2SE y J2EE. Las librerías definidas en el CLDC se pueden dividir en dos categorías:

- **Clases derivadas del J2SE**

  Estas clases son un subconjunto de las existentes en las librerías del J2SE, todas tienen el mismo nombre que se les daba en J2SE y pertenecen a paquetes del mismo nombre que en el J2SE. Los paquetes en los que se encuentran son *java.lang, java.util, java.io:*

  1. **Clases de sistema**

```
java.lang.Object
java.lang.Class
java.lang.Runtime
java.lang.System
java.lang.Thread
java.lang.Runnable
java.lang.String
java.lang.StringBuffer
java.lang.Throwable
```

  2. **Clases de tipos de datos**

```
java.lang.Boolean
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Character
```

  3. **Clases de Colecciones de objetos**

```
java.util.Vector
java.util.Stack
java.util.Hashtable
java.util.Enumeration
```

4. Clases de Entrada/Salida

```
java.io.InputStream
java.io.OutputStream
java.io.ByteArrayInputStream
java.io.ByteArrayOutputStream
java.io.DataInput
java.io.DataOutput
java.io.DataInputStream
java.io.DataOutputStream
java.io.Reader
java.io.Writer
java.ioInputStreamReader
java.ioOutputStreamReader
java.ioPrintStream
```

5. Clases de Temporización y Calendario

```
java.util.Calendar
java.util.Date
java.util.TimeZone
```

6. Clases de Adicionales

```
java.util.Random
java.util.Math
```

7. Clases de Errores

```
java.lang.Error
java.lang.VirtualMachineError
java.lang.OutOfMemoryError
```

8. Clases de Excepciones

```
java.lang.Exception
java.lang.ClassNotFoundException
java.lang.IlegalAccesException
java.lang.InstantiationException
java.lang.InterruptedException
java.lang.RuntimeException
java.lang.ArithmeticException
java.lang.ArrayStoreException
java.lang.ClassCastException
java.lang.IlegalArgumentException
java.lang.IlegalThreadStateException
java.lang.NumberFOrmatException
java.lang.IlegalMonitorStateException
java.lang.IndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException
java.lang.StringIndexOutOfBoundsException
java.lang.NegativeArraySizeException
java.lang.NullPointerException
java.lang.SecurityException
java.util.EmptyStackException
java.util.NoSuchElementException
java.io.EOFException
java.io.IOException
java.io.InterruptedIOException
java.io.UnsupportedEncodingException
java.io.UTFDataFormatException
```

- **Clases específicas del CLDC**

Estas clases son específicas del CLDC y tienen como objetivo el permitir una serie de funcionalidades que el dispositivo puede realizar y que las clases anteriores no realizan. Estas clases se encuentran en el paquete *javax.microedition.*

El conjunto de librerías que existen en J2SE y J2EE y que están enfocadas a dar funcionalidades de entrada y salida de datos desde y hacia redes es extremadamente rico, tan solo el paquete java.io de J2SE cuenta con 60 clases e interfaces y más de 15 excepciones, y el paquete *java.net* cuenta con unas 20 clases y unas 10 excepciones. Claro está para almacenar todas estas clases se necesita aproximadamente 200 KB, cantidad de memoria imposible de almacenar en un dispositivo móvil. Debido a esto no es posible reaprovechar dichas librerías y por tanto hay que diseñar todo un conjunto de librerías propias del J2ME que aporten estas funcionalidades. Teniendo en cuenta que no todas las funcionalidades de red son aplicables a los dispositivos móviles y que los fabricantes no hacen uso del amplio rango de posibles comunicaciones en red existentes sino que, en la mayoría de los casos estos dispositivos no cuentan con soporte TCP/IP o incluso se restringen a ciertos protocolos como por ejemplo IrDA o Bluetooth, el problema se simplifica enormemente.

De esta forma, para los dispositivos con los que trabaja J2ME toda la funcionalidad de red se encuentra en el Generic Connection framework. Éste, proporciona en la medida de lo posible un subconjunto de las funcionalidades que aportan J2SE y J2EE y que son útiles para el dispositivo pero además aporta mejor extensibilidad, flexibilidad y coherencia a la hora de permitir nuevos equipos y protocolos de comunicación.

1. **Objeto *javax.microedition.io.Connector***

En este objeto reside una interfaz en la cual J2ME encapsula las conexiones a red. Dependiendo del dispositivo, y de la red a la que se conecta será necesario un protocolo u otro, en este objeto el Generic Connection framework puede encapsular cualquiera de las posibles conexiones que se puedan dar. Indistintamente del tipo que sea, el Generic Connection framework se encarga de asegurar compatibilidad.

El Generic Connection framework es implementado según una jerarquía de interfaces de conexión encapsuladas en el objeto Connection, que suman en total seis tipos: una entrada Serie, una salida Serie, una conexión orientada a datagrama, una conexión orientada a circuito, un mecanismo de notificación de conexiones, una conexión a un servidor Web:

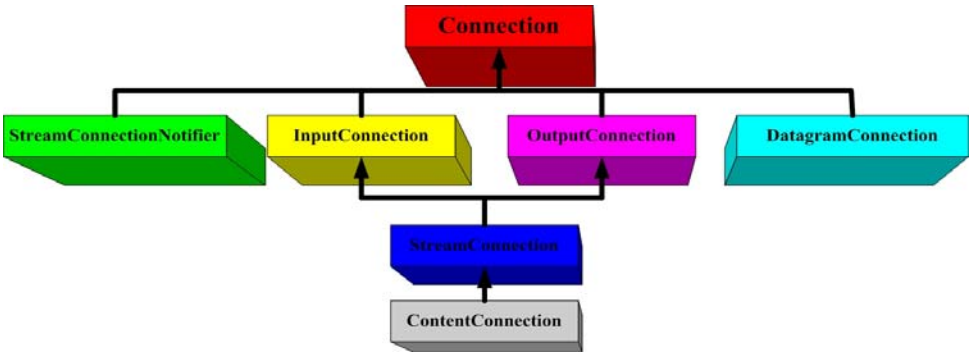

Figura 2.7: Jerarquía de la interfaz de conexión

2. Métodos

Sobre el objeto *Connector* existen una serie de métodos con los cuales es posible hacer uso de las distintas funcionalidades de comunicación que J2ME permite, estos métodos son:

public void open("<protocol> : <address> ; <parameters>")

Con este método es posible abrir y establecer una comunicación, cuenta con la ventaja de permitir varios protocolos de comunicación, no solo los protocolos que ya se están utilizando, sino que está preparado para adaptarse a los nuevos posibles protocolos que puedan llegar en el futuro, así algunos ejemplos son:

```
HTTP
        Connector.open("http://www.sun.com");
Sockets
        Connector.open("socket://129.144.111.222:2800");
Comunicación por puertos
        Connector.open("comm:0;baudrate=9600");
Datagramas
        Connector.open("datagram://129.144.111.222:2800");
Ficheros
        Connector.open("file:/foo.dat");
```

Interfaz Connection

```
public void close() throws IOException
```

Interfaz InputConnection

```
public InputStream openInputStream() throws IOException
public DataInputStream openDataInputStream() throws
      IOException
```

Interfaz OutputConnection

```
public OutputStream openOutputStream() throws IOException
public DataOutputStream openDataOutputStream() throws
      IOException
```

Interfaz Connection

```
public String getType()
public String getEncoding()
public long getLength()
```

Interfaz StreamConnectionNotifier

```
public StreamConnection acceptAndOpen() throws
      IOException
```

Interfaz DatagramConnection

```
public int getMaximumLength() throws IOException
public int getNominalLength() throws IOException
public void send(Datagram datagram) throws IOException
public void receive(Datagram datagram) throws IOException
public Datagram newDatagram(int size) throws IOException
public Datagram newDatagram(int size, String addr) throws
      IOException
public Datagram newDatagram(byte[] buf, int size) throws
      IOException
public Datagram newDatagram(byte[] buf, int size, String
      addr) throws IOException
```

2.7 MIDP (Mobile Information Device Profile)

El estándar MIDP es una arquitectura y un conjunto de librerías que aportan un entorno de desarrollo de aplicaciones abiertas Third-Party para dispositivos móviles de información (MIDs). Ejemplos típicos de dispositivos MIDP son teléfonos móviles, PDAs con conexión inalámbrica, etc.

El conjunto mínimo de requisitos que debe cumplir cualquier dispositivo sobre el que se quiera implementar MIDP es el siguiente:

- **Memoria**

  - 128 KB de memoria no volátil para los componentes MIDP
  - 8 KB de memoria no volátil para datos de aplicaciones almacenados de forma persistente
  - 32 KB de memoria volátil para la ejecución de la máquina virtual

- **Pantalla**

  - Tamaño de pantalla: 96 x 54 mm^2
  - Profundidad de display: 1 bit
  - Relación de aspecto: 1:1

- **Interfaz de entrada:** Al menos uno de los siguientes mecanismos:

  - Teclado a una o dos manos
  - Pantalla táctil

- **Red**

  - Conexión bidireccional, inalámbrica, posiblemente inalámbrica con limitación de ancho de banda

2.7.1 MIDP Expert Group

La MIDP Specification fue producida por el MIDPEG (Mobile Information Device Profile Expert Group) como parte del Java Community Process (JCP) en la norma de estandarización JSR-37. Este grupo está formado por un conjunto de empresas:

- **America Online**
- **DDI**
- **Ericsson**
- **Espiral Group**
- **Fujitsu**
- **Hitachi**
- **J-Phone**
- **Matsushita**
- **Mitsubishi**
- **Motorola**
- **NEC**
- **Nokia**
- **NTT DoCoMo**
- **Palm Computing**
- **Research In Motion**
- **Samsung**
- **Sharp**
- **Siemens**
- **Sony**
- **Sun Microsystems**
- **Symbian**
- **Telecordia Technologies**

El MIDP ha sido diseñado para extender las funcionalidades del CLDC, de esta forma la MIDP Specification define un conjunto de APIs que añaden un conjunto mínimo de funciones que son comunes a los distintos tipos de dispositivos que el MIDP contempla:

- **Soporte de interfaz de usuario**: LCDUI (Limited Connected Device Uder Interface)

- **Soporte de red**, basado en el protocolo http y en el Generis Connection framework introducido por el CLDC

- **Soporte de almacenamiento persistente de datos**: RMS (Record Management System)

- Una serie de clases adicionales de interesante utilidad como temporizadores y excepciones.

2.7.2 Modelo de Aplicación de MIDP

Debido a las fuertes restricciones de memoria con las que se enfrenta y a los requisitos exigidos por el estándar, el MIDP no soporta el modelo de Applet introducido en el J2SE sino que utiliza un nuevo modelo de aplicación gráfica que permita compartir e intercambiar datos entre aplicaciones, así como un funcionamiento concurrente sobre la KVM.

En el MIDP la unidad mínima de ejecución es el MIDlet el cual no es más que una clase que extiende a la clase *javax.microedition.MIDlet.*

Un buen ejemplo de un MIDlet es el clásico “Hola Mundo”, ya que sirve para implementar el MIDlet más sencillo posible:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class HolaMundo extends MIDlet implements CommandListener{

    private TextBox tb;
    private Command Salir;

    public HolaMundo(){
        tb = new TextBox("Hola MIDlet", "Hola Mundo!", 15, 0);
        Salir = new Command ("Salir", Command.EXIT, 1);
        tb.addCommand(Salir);
        tb.setCommandListener(this);
    }

    protected void startApp(){
        Display.getDisplay(this).setCurrent(tb);
    }

    protected void pauseApp(){}
    protected void destroyApp(boolean u){}

    public void commandAction(Command c, Displayable s){

        if(c == Salir){
            destroyApp(false);
            notifyDestroyed();
        }
    }
}
```

La salida de este sencillo código ejemplo se muestra en la figura. El estado en el cual se encuentra el MIDlet en esta figura es tal que el método *startApp* acaba de terminar su ejecución. Si en este momento pulsásemos el botón superior izquierdo del teclado del móvil se invocaría al método *commandAction* y éste, al detectar el comando ejecutado, llamaría en este caso al método *destroyApp* y al método *notifyDestroyed*.

En éste ejemplo se encuentran un conjunto de elementos típicos de todos los MIDlets:

- En primer lugar la clase HolaMundo extiende a la clase *javax.microedition.midlet.MIDlet*.

- En segundo lugar, la clase *HolaMundo* cuenta con un constructor que en el modelo de aplicación del MIDP se ejecuta una sola vez, exactamente una sola vez al instanciar el MIDlet. Por lo tanto, todas aquellas operaciones que queramos que se realicen una sola vez al lanzar el MIDlet por primera vez deben estar en este constructor.

- La clase *javax.microedition.midlet.MIDlet* define tres métodos abstractos los cuales deben ser sobrescritos en todos los MIDlets, estos métodos son: *startApp, pauseApp y destroyApp*.

Figura 2.8: Resultado de la emulación del midlet

El método *startApp* es ejecutado al arrancar y rearrancar el MIDlet, su función es la de adquirir al arrancar por primera vez o readquirir al rearrancar tras una pausa una serie recursos necesarios para la ejecución. Este método puede ser llamado más de una

vez, la primera vez al arrancar el MIDlet por primera vez y de nuevo cada vez que se quiera "resumir" al MIDlet.

El método *pauseApp* es llamado por el sistema con el fin de parar momentáneamente al MIDlet. Este método suele usarse con el fine de parar al MIDlet, liberando así durante la pausa una serie de recursos que puedan ser usados durante ese tiempo por otros MIDlets.

Por último, el método *destroyApp* es llamado por el sistema cuando el MIDlet está apunto de ser destruido, también puede ser llamado indirectamente por el propio MIDlet mediante el método *notifyDestoyed*. Este método juega un papel muy importante, ya que es el encargado de liberar definitivamente todos aquellos recursos que el MIDlet ha tomado y ya no necesitará después de ser destruido.

Debido a la existencia de estos tres métodos un MIDlet puede pasar por una serie de estados diferentes a lo largo de su ejecución dándose en cada estado una serie de procesos sobre dicho MIDlet, y pasando de un estado a otro mediante cada uno de estos métodos. Gráficamente es sencillo entender estas transiciones:

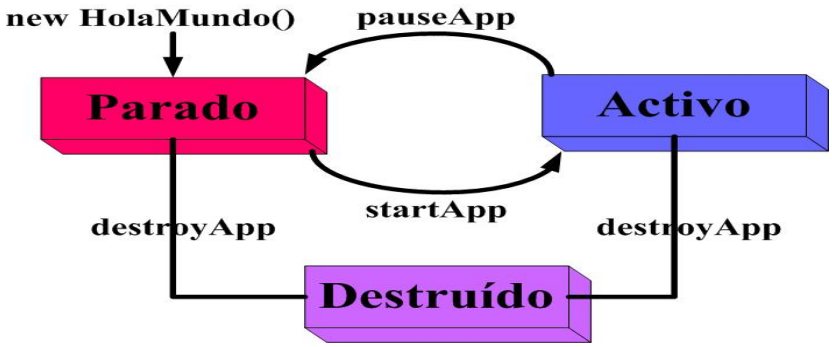

Figura 2.9: Estados y transiciones de un MIDlet

Como podemos ver en la figura, un MIDlet puede encontrarse en tres posibles estados:

- **Parado**: Un MIDlet se encuentra en este estado cuando está apunto de ser arrancado pero aún no ha sido ejecutado el método *startApp* o también como resultado de *pauseApp* o *notifyPaused*. En este estado el MIDlet debería ya haber acaparado tantos recursos como le sean necesarios, además en esta estado puede también recibir notificaciones asíncronas.

- **Activo:** Un MIDlet entra en este estado mediante el método *startApp* o bien desde el estado Parado mediante la ejecución del método *resumeRequest*. En este estado el MIDlet puede acaparar y emplear todos aquellos recursos que necesite para realizar una ejecución óptima.

- **Destruido:** El MIDlet se encuentra en este estado cuando vuelve de los métodos *destroyApp* o *notifyDestroyed*. Tras llegar a este estado, el MIDlet no puede pasar a ningún otro estado.

Para pasar de un estado a otro existen una serie de métodos que hacen posibles las distintas transiciones del MIDlet, ya hemos comentado tres de estos métodos, los métodos *startApp*, *pauseApp* y *destroyApp* que son llamados por el sistema de forma automática, pero además como ya hemos comentado antes existen también una serie de métodos que realizan estas transiciones y que en su caso son utilizados por el programador de la aplicación con la intención de forzar dichas transiciones cuando sea conveniente, estos métodos son:

- ***resumeRequest:*** Este método puede ser llamado por un MIDlet parado para indicar su intención de volver a estar activo. Un ejemplo típico es el caso del cumplimiento de un temporizador el cual necesite resumir la aplicación para continuar con la ejecución del MIDlet**.**

- ***notifyPaused:*** Éste permite al MIDlet indicarle al sistema que voluntariamente se ha pasado a estado parado. Un ejemplo de uso sería el del inicio de cuenta de un temporizador de forma que sea más conveniente liberar una serie de recursos, para que otras aplicaciones los usen, hasta que dicho temporizador cumpla y se pase de nuevo a ejecución.

- ***notifyDestroyed:*** Con este método la aplicación puede indicarle al sistema que ya ha liberado todos los recursos y ha almacenado todos los datos convenientemente y por tanto se va a pasar a estado parado.

2.7.3 MIDlet Suites

Uno de los principales objetivos del Modelo de Aplicación del MIDP es dar soporte para compartir datos y recursos entre varios MIDlets que, incluso podrían estar funcionando simultáneamente. Para conseguir esto todos aquellos MIDlets que quieran compartir datos y recursos deben encapsularse en un mismo fichero JAR. Este fichero es lo que se llama un MIDlet Suite. Todos los MIDlets que se encuentren dentro de este MIDlet Suite comparten un espacio común de nombres para almacenamiento persistente de datos y un mismo conjunto de clases y campos estáticos. Con el fin de mantener la seguridad, el MIDlet Suite es tratado como un "todo", de forma que ningún elemento de éste puede ser instalado, actualizado o eliminado individualmente.

El contenido del fichero JAR que contiene al MIDlet Suite es el siguiente:

- Los **ficheros de clases** que implementan los distintos MIDlets.

- Los distintos **recursos** utilizados por estos MIDlets, tales como iconos o ficheros de imagen, por ejemplo.

- Un **manifiesto** que describa el contenido del JAR.

El manifiesto proporciona un mecanismo de información acerca del contenido del fichero JAR por medio de una serie de atributos de los cuales unos están reservados

para el MIDP Expert Group (son los que sus nombres empiezan por *MIDlet-*) y otros son atributos propios de los desarrolladores de MIDlet Suites.

Dentro del manifiesto deben ir obligatoriamente los siguientes atributos:

- **MIDlet-Name**
- **MIDlet-Version**
- **MIDlet-Vendor**
- **MIDlet-<nombre>,** uno por cada MIDlet
- **MicroEdition-Profile**
- **MicroEdition-Configuration**

Adicionalmente al fichero JAR está el application descriptor, de carácter opcional, que se encarga de verificar que su MIDlet Suite asociado se ajusta convenientemente al dispositivo sobre el que será descargado desde la estación de trabajo donde se está desarrollando la aplicación. Este fichero tiene la extensión *.jad* y sigue una sintaxis muy concreta. Debe contener los siguientes atributos:

- **MIDlet-Name**
- **MIDlet-Versión**
- **MIDlet-Vendor**
- **MIDlet-Jar-URL**
- **MIDlet-Jar-Size**

2.7.4 Librerías de MIDP

Teniendo en cuenta el gran número de restricciones con las que cuentan los dispositivos sobre los que se monta J2ME es conveniente tener en mente una serie de requisitos a la hora de diseñar el conjunto de librerías con las que contará:

- Los equipos y aplicaciones tienen que ser de fácil uso para los usuarios, que no necesariamente serán expertos en el uso de computadores.

- Estos equipos y aplicaciones deben ser de fácil uso en situaciones comprometidas en las que el usuario no pueda poner total atención, situaciones como estar conduciendo, cocinando, etc.

- La forma e interfaz de usuario varía considerablemente de un equipo a otro.

- Las aplicaciones Java para móviles deben tener interfaces de usuario compatibles con las de las aplicaciones nativas de forma que los usuarios las encuentren fáciles de usar.

Dados estos requisitos el MIDP expert group decidió que el API AWT (Abstract Windowing Toolkit) proporcionado por J2SE no fuese utilizado para los dispositivos con los que trata J2ME. Los motivos fueron varios:

- AWT fue diseñado expresamente para ordenadores de sobremesa, por lo que existen una serie de características no compatibles con los dispositivos con los que tratamos.

- AWT necesita de un complejo mecanismo de recolección de basura para eliminar todos aquellos elementos que no se estén utilizando. Esta recolección de basura no se ha implementado en J2ME.

- Por último, AWT desarrolla su diseño pensando en que el usuario hará uso de las posibilidades gráficas que se le presentan mediante un puntero como el del ratón, pero en los dispositivos del J2ME muy pocos cuentan con este ratón sino que trabajan directamente con un teclado reducido de flechas de dirección.

2.7.4.1 Librerías de la Interfaz Gráfica de Usuario

La principal abstracción en la interfaz de usuario es el objeto Displayable (visible) perteneciente al paquete *javax.microedition.lcdui* el cual encapsula los distintos gráficos que se presentan por pantalla. La visualización de un MIDlet es controlada por un objeto de tipo *Displayable*, para ser visible por pantalla un MIDlet debe crear un objeto que derive de la clase *Displayable*. Esta clase será entonces la responsable de dibujar la pantalla correspondiente. Para que un objeto *Displayable* sea visible, se debe invocar al método *void setCurrent(Displayable next).*

Siempre se debe recoger el objeto de tipo *Display* que gestiona lo que muestra la pantalla del dispositivo *display = Display.getDisplay(this).* La clase *Display* incluye una serie de métodos útiles para gestionar lo que se va a presentar en la pantalla del dispositivo y la interacción con el usuario.

Gráficamente podemos ver la jerarquía de clases, para comprenderlo mejor:

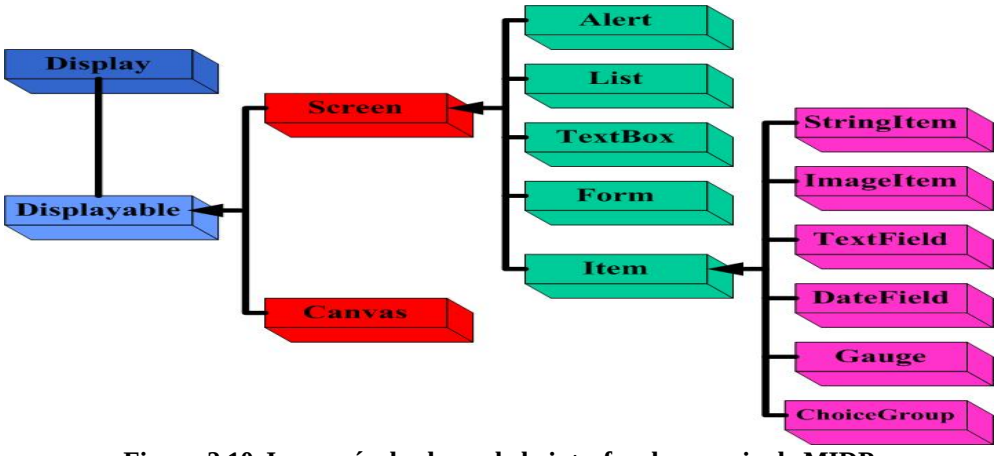

Figura 2.10:Jerarquía de clases de la interfaz de usuario de MIDP

Existen dos tipos de objetos ***Displayable***:

- ***Canvas:*** Objeto de **bajo nivel** que permite a la aplicación tener gráficos y una serie de entradas.

- ***Screen:*** Objeto de **alto nivel** que encapsula una interfaz de usuario completa.

En cualquier aplicación estos dos tipos de objetos pueden combinarse sin ningún tipo de problemas.

1. Interfaz de usuario de Bajo Nivel

El API de la interfaz de usuario de bajo nivel está especialmente diseñado para aplicaciones que necesitan un control preciso de los elementos gráficos en la pantalla. Ejemplos típicos de esto serían aplicaciones de dibujo de gráficos por parte del usuario, juegos, etc.

Con este API podemos controlar varios aspectos, tales como:

- Control de lo que se está dibujando en la pantalla.
- Captura de eventos primitivos tales como presión de teclas cornetas del teclado.
- Acceso a teclas concretas del teclado y de vías de entrada de datos.

A la hora de realizar cualquier gráfico sobre la pantalla mediante el API de bajo nivel necesitamos de un sistema de coordenadas, el cual sitúa el punto (0,0) origen de coordenadas en la esquina superior-izquierda de la pantalla. Así la coordenada x crece hacia la derecha de la pantalla y la coordenada y crece hacia abajo. Los valores de las coordenadas siempre son enteros positivos.

La clase *Graphics* contiene la mayoría de funcionalidades para dibujar a bajo nivel, ésta proporciona la capacidad de dibujar gráficas primitivas (líneas, rectángulos...), texto e imágenes tanto en la pantalla como en un buffer de memoria. El método *paint()*, será el método para dibujar.

La clase *Graphics* tiene una serie de atributos que determinan cómo se llevan a cabo las diferentes operaciones. El más importante de estos atributos es el atributo *color*, que determina el color usado del dibujo que se esté realizando. Este atributo se puede modificar con el método *setColor*, que recibe tres parámetros enteros que especifican el color en función de los tres colores primarios, o bien el método *setGrayScale()* que toma como parámetro un único valor entero que establece el grado de gris en un rango que va desde 0 hasta 255.

Los objetos de tipo *Graphics* contienen también un atributo denominado *font* que determina el tamaño y la apariencia del texto, éste atributo se modifica a través del método *setFont()*.

Las primitivas gráficas que podemos dibujar son líneas, rectángulos y arcos. La clase *Graphics* proporciona métodos para realizar estas operaciones y además, nos proporciona métodos para rellenar las áreas.

El método que permite dibujar una línea es *void drawLine (int x1, int y1, int x2, int y2), l*os parámetros *x1* e *y1*, indican el punto de comienzo de la línea y *x2* e *y2* indican el punto final. Para cambiar el estilo de la línea tenemos el método *setStrokeStyle(),* que acepta un valor de los siguientes:

- ***Graphics.SOLID***: Línea sólida
- ***Graphics.DOTTED*:** Línea de puntos

El método para dibujar rectángulos es *void drawRect (int x, int y, int width, int height),* los parámetros *x* e *y* especifican la localización de la esquina superior-izquierda del rectángulo y los parámetros *width* y *height* especifican el ancho y el alto respectivamente del rectángulo. Para dibujar rectángulos con las esquinas redondeadas tenemos *void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight),* método que contiene los parámetros *arcWidth* y *arcHeigth* que configuran el arco de las esquinas del rectángulo. La clase *Graphics* también tiene métodos para dibujar rectángulos rellenos de un color determinado, el color de relleno será el color configurado en el atributo *color,* los métodos para hacer esto son *fillRect()* y *fillRoundRect().*

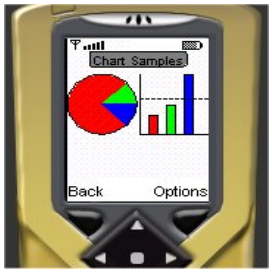

Figura 2.11: Ejemplo de pantalla gráfica

Tenemos además los arcos que no son más que una sección de un óvalo. El método que dibuja un óvalo es *void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle),* donde los primeros cuatro parámetros definen el óvalo del que el arco forma parte y los últimos dos parámetros definen el arco como una sección del óvalo. Para dibujar un óvalo, tendremos que utilizar la clase *drawArc*, especificando un ángulo de 360º. El método para dibujar un arco relleno de color es *fillArch().*

El texto que se escribe usando la clase *Graphics* está configurado por el atributo *font* el cual se cambia con el método *void setFont (Font font).* El objeto *Font* indica el tipo de fuente, el estilo y el tamaño del texto. Para crear un objeto de tipo *Font* usamos el método *static Font getFont(int face, int sytle, int size).* Una vez que tenemos la fuente usamos el método *setFont()* para que se use la misma en operaciones sucesivas.

El método para dibujar el texto es *void drawstring (String str, int x, int y, int anchor),* donde el primer parámetro es el texto a escribir y los dos siguientes (*x* e *y*) especifican la localización del texto. A parte de este método tenemos algunos otros métodos más para dibujar texto como son los métodos *drawChar()* y *drawChars()* usados para dibujar caracteres de texto individuales, también podemos usar el método *drawSubstring()* que permite escribir una parte de un cadena.

Por último están las imágenes que son objetos gráficos rectangulares compuestos de píxel de color o grises. Antes de dibujar una imagen, es necesario cargarla. Las

imágenes son cargadas y creadas usando un método de la clase *Image* denominado *Public static Image createImage (String name) throws IOException* donde el parámetro indica el nombre del fichero que contiene la imagen, este método retorna un objeto de tipo *Image* que puede ser usado dentro del MIDP. La clase *Image* representa una imagen gráfica como puede ser un fichero PNG, GIF o JPEG. También incluye un método para recuperar un objeto *Graphics* que permite dibujar directamente sobre la imagen *boolean drawImage (Image img, int x, int y, int anchor).*

2. Interfaz de usuario de Alto Nivel

El API del interfaz de usuario de alto nivel está pensado para aplicaciones profesionales dirigidas a usuarios trabajando en dispositivos móviles de información. Por tanto la portabilidad de código de un equipo a otro es muy importante, lo cual se consigue empleando un nivel de abstracción alto el cual da lugar a una pérdida de control sobre el dispositivo, así las aplicaciones no definen la apariencia de las pantallas, sino que lo hace el sistema; la navegación el scroll de las pantallas y demás operaciones de interacción con el usuario lo realiza el sistema; y las aplicaciones no pueden acceder a mecanismos de entrada de datos concretos, como por ejemplo una tecla individual de todo el teclado.

La clase *Screen*, que es una clase abstracta, proporciona la funcionalidad básica para una pantalla, que principalmente consiste en un título que aparecerá en la parte superior de ésta. Se puede modificar este título con los métodos *String getTitle*() y *Void setTitle (String s).* Además del atributo de título, las pantallas también pueden tener una línea de texto en movimiento que aparecerá sobre la pantalla. Este texto se configura a través de la clase *Ticket,* elemento que se puede manejar con los métodos *Ticker getTicker()* y *void setTicker (Ticker ticker).*

Las funcionalidades de este API se dan a través de cuatro clases las cuales heredan de *Screen*, que son:

- ***List:*** Esta clase proporciona una pantalla que contiene una lista de elementos que el usuario puede seleccionar. Existen tres tipos básicos de listas:
  - ***EXCLUSIVE:*** Una lista que permite seleccionar un solo elemento a la vez.
  - ***IMPLICIT:*** Un lista *EXCLUSIVE* en la que el elemento seleccionado es lanzado como respuesta a un comando.
  - ***MULTIPLE:*** Una lista que permite seleccionar uno o más elementos a la vez.

  Los constructores de la clase *List* son *List(String title, int listType)* y *List(String title, int listType, String[] stringElements, Image[] imageElements).*

  Una vez creada la lista se puede interactuar con ella. Para recuperar el elemento seleccionado en una lista *exclusive* o *implicit* se usa el método

getSelectedIndex() que devuelve el índice del elemento seleccionado dentro de la lista y para listas de tipo *múltiple*, se usa el método *getSelectedFlags()* que devuelve una matriz cuyos valores son de tipo *bolean* e indican si el elemento correspondiente está seleccionado o no.

Figura 2.12: Ejemplo de lista

- ***TextBox:*** La clase *TextBox* implementa un componente de edición de texto que ocupa toda la pantalla. El constructor de la clase es *TextBox(String title, String text, int maxSize, int constraints)* donde el parámetro título es un texto que aparecerá en la parte superior de la pantalla, mientras que el parámetro texto es usado para inicializar el texto que contendrá el *TextBox*, si es necesario. El parámetro *maxSize* especifica el número máximo de caracteres de texto que pueden ser introducidos.

Figura 2.13: Ejemplo de TextBox

Por último, el parámetro *constraints* describe las limitaciones a aplicar sobre el texto. Estas limitaciones son especificadas según unas constantes:

- ***ANY:*** No hay limitaciones en el texto.
- ***EMAILADDR:*** Sólo se puede introducir una dirección de correo electrónico.
- ***NUMERIC:*** Sólo se puede introducir un valor entero.
- ***PASSWORD:*** El texto es protegido para que no sea visible.
- ***PHONENUMBER:*** Sólo se puede introducir un número de teléfono.
- ***URL:*** Sólo se puede introducir una URL

- ***Alert:*** La clase *Alert* implementa una pantalla que muestra un texto informativo al usuario. Esta información se muestra durante un determinado espacio de tiempo o hasta que el usuario seleccione un comando, según se configure. El constructor de la clase *Alert* es: *Alert(String title, String alertText, Image alertImage, AlertType alertType)* donde el título es un texto que aparece en la parte superior de la pantalla, mientras que el texto de la alerta actúa como el cuerpo del mensaje de alerta, con el tercer parámetro se puede indicar una imagen para que se muestre en la alerta y el último parámetro indica el tipo de alerta. Los tipos de alertas pueden ser:

  - ***ALARM***
  - ***CONFIRMATION***
  - ***ERROR***
  - ***INFO***
  - ***WARNING***

Figura 2.14: Ejemplo de Alert

Mediante el método *AlertType.ERROR.playSound(display)* se puede hacer sonar un sonido de error. Por defecto, las alertas se mostrarán durante unos segundos y desaparecerán automáticamente. Este periodo de visualización de la alerta se puede configurar con el método *void setTimeout(int time)*

- ***Form:*** Sirve como contenedor para construir interfaces de usuario con otros componentes o *Items*. Para añadir elementos de tipo *Item* al *Form* y componer una interfaz de usuario personalizada se utiliza el método *int append (Item item).* Cada *Item* añadido a un *Form* posee un índice dentro del Form y este índice es el número que retorna el método *append*. Para hacer referencia a un *Item* se usa este índice.

  Para eliminar un *Item* de un *Form* se usa el método *void delete(int index).* Se puede insertar un *Item* en un punto concreto del *Form* con el método *void insert(int index, Item item).* Para modificar un *Item* debemos usar el método *void set(int index, Item item).* Para recuperar un determinado *Item* disponemos del método Item get(int index), finalmente, para saber el número total de *Items* en un *Form* tenemos el método *int size().*

La clase *Form* está pensada para aquellos casos en los que una pantalla con una única funcionalidad no es suficiente, sino que es necesario que contenga un pequeño número de elementos de interfaz de usuario, o *Items.* Si el número de *Items* fuese tal que no entrasen en una sola pantalla, el propio sistema se encarga de implementar un scroll

que permita subir o bajar a lo largo de la pantalla para visualizar a los distintos *Items.* En cualquier orden y número, un *Form* puede contener los siguientes *Items:*

- ***StringItem:*** Esta clase representa un elemento que contiene una cadena de texto, es usada para mostrar texto en un *Form*. Está compuesta de dos partes, una etiqueta y un texto. El constructor de esta clase es *StringItem(String label, String text).*

- ***ImageItem:*** La clase *ImageItem* es similar a *StringItem* pero está diseñada para mostrar imágenes en lugar de texto. El constructor de esta clase es *ImageItem(String label, Image img, int layout, String altText).* El primer parámetro es un texto a mostrar en pantalla, la imagen es el segundo parámetro y el parámetro *layout* determina cómo la imagen es posicionada en el *Form* respecto a otros elementos, el último parámetro, especifica un texto para mostrar en lugar de la imagen.

- ***TextField:*** Esta clase proporciona un editor de texto diseñado para ser usado dentro de los *Forms* (ésta es la principal diferencia con respecto a la clase *TextBox*) por lo demás se trabaja con ella igual. El constructor de la clase es *TextField(String label, String text, int maxSize, int constraints)* donde el primer parámetro establece la etiqueta que se muestra junto al componente, el segundo el texto utilizado para inicializar el elemento, el parámetro *maxSize* indica el máximo número de caracteres que pueden ser introducidos y el último parámetro, de forma similar a lo indicado en la clase *TextBox*, indica las restricciones del texto a introducir.

Figura 2.15:Ejemplo de TextField

- ***DateField:*** Presenta al usuario una interfaz intuitiva para introducir fechas y horas. El constructor de esta clase es *DateField (String label, int mode)*. En el primer parámetro tenemos la etiqueta a mostrar con el elemento, el parámetro *mode* indica el modo de funcionar del componente. Estos modos de funcionamiento son:

  - ***DATE:*** Se introduce solo una fecha (día, mes y año).
  - ***TIME:*** Se introduce solo una hora (horas y minutos).
  - ***DATE_TIME:*** Se introduce el día y la hora.

- ***Gauge:*** La clase *Gauge* implementa una barra gráfica que puede ser usada para visualizar un valor dentro de un rango. Un objeto de tipo *Gauge* tiene un valor máximo que define el rango del objeto (de 0 al

máximo) y un valor actual que determina el estado actual de la barra. La clase *Gauge* puede funcionar interactivamente y no interactivamente. Si funciona de forma no interactiva, simplemente se muestra la barra, mientras que si trabaja de forma interactiva, se usará la barra como método para introducir un valor por parte del usuario, manipulando la barra.

Figura 2.16: Ejemplo de Gauge

Para crear un objeto de tipo *Gauge* tenemos el constructor *Gauge(String label, boolean interactive, int maxValue, int initialValue),* el primer parámetro es la etiqueta asociada al componente, el segundo parámetro indica si es interactivo o no interactivo. Se puede acceder al valor actual y cambiar dicho valor con *int getValue()* y *void setValue(int value),* también es posible manipular el valor máximo con los métodos *int getMaxValue()* y *void setMaxValue(int value).*

- ***ChoiceGroup:*** Esta clase presenta una lista de elementos los cuales el usuario puede seleccionar. Esta clase es similar a la clase *List*, pero la clase *ChoiceGroup* está pensada para ser usada dentro de un *Form*. A parte de esto, no hay muchas más diferencias entre ambas. Los constructores son *ChoiceGroup(String label, int choiceType)* y *ChoiceGroup(String label, int choiceType, String[] stringElements, Image[] imageElements).*

3. Comandos de Pantalla

Puesto que el MIDP implementa una interfaz de usuario de alto nivel, con una abstracción muy alta no se facilita una técnica concreta de interacción con el usuario, sino que se utiliza un mecanismo de gestión de comandos abstracto de forma que se pueda ajustar a los distintos dispositivos físicos sobre los que trabaja J2ME.

En una aplicación MIDP se definen una serie de comandos (objeto *Command*) y la forma de gestionarlos por parte del usuario mediante botones, menús o cualquier otro mecanismo que pueda existir en un dispositivo.

El objeto *Command* ofrece un constructor con tres parámetros concretos:

Command(String label, int commandType, int priority)

- ***label:*** Texto que se muestra al usuario en la pantalla para identificar el comando.

- ***commandType:*** Tipo del comando o significado del comando, funcionalidad que realiza el comando. Existen varias funcionalidades:

  - **OK**: Verifica que se puede comenzar una acción
  - **CANCEL**: Cancela la acción
  - **STOP:** Detiene la acción
  - **EXIT:** Sale del MIDlet
  - **BACK:** Envía al usuario a la pantalla anterior
  - **HELP:** Solicita la ayuda
  - **ITEM:** Un comando especifico de la aplicación que es relativo a un determinado *item* de la pantalla actual
  - **SCREEN:** Un comando especifico de la aplicación que es relativo a la pantalla actual

- ***priority:*** Permite al sistema emplazar al comando en función de su prioridad, de forma que si hubiese demasiados comandos que no entrase en una sola pantalla se presentarían en esta por los de mayor prioridad, pasando los otros a un submenú adicional.

Existe un comando especial que es el *List.SELECT_COMMAND* el cual está pensado para albergar la selección realizada de entre las posibles existentes en un objeto *List.*

Por último, para poder capturar la selección de un comando en cada objeto *Displayable* existe un *"listener"* que se encarga de estar a la escucha de comandos, de forma que cuando un comando sea seleccionado, este *listener* lo detecta y pasa a ejecutar el código relacionado con dicho comando. Para registrar este *listener* contamos con el método *Displayable.setCommandListener* y para gestinonar la ejecución de código relativo a un comando se debe implementar la interfaz *CommandListener* y su método *commandAction.*

2.7.4.2 Librerías de Red

Los equipos con los que trabaja MIDP operan en una amplia variedad de redes inalámbricas de comunicación cada una de ellas con un protocolo de comunicación distinto e incompatible con todos los demás.

Un objetivo del MIDP Specification es de adaptar su modelo de conexión a la red no solo a todos los protocolos existentes hoy en día, sino también a todos aquellos que puedan llegar en un futuro.

Además de lo ya visto anteriormente al respecto de las comunicaciones de red, el API de MIDP añade la interfaz *HttpConnection* que proporciona un componente nuevo en el GCF para conexiones HTTP. Estas conexiones permiten a los MIDlets conectarse con páginas Web. La especificación MIDP indica que las conexiones HTTP son el único tipo de conexiones obligatorio en las implementaciones de MIDP.

Después de un largo estudio acerca de la situación actual de los protocolos de red, el MIDP expert group decidió tomar el protocolo HTTP como protocolo base para las comunicaciones de red. HTTP es un protocolo muy rico y ampliamente utilizado que puede implementarse sobre redes inalámbricas de forma sencilla, sin olvidar que hace posible favorecerse de la extensa infraestructura de servidores ya existente que corren con este protocolo.

Dar soporte al protocolo HTTP no implica necesariamente que el equipo implemente el protocolo TCP/IP , el quipo podría utilizar otros protocolos como WAP o i-Mode conectándose en la red con un Gateway que sirva de puente de unión con los servidores de Internet.

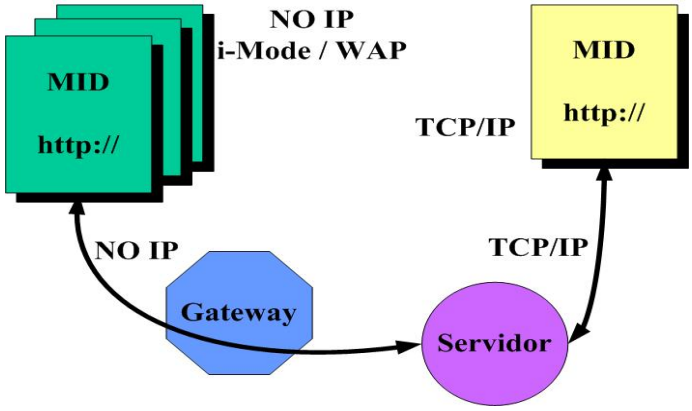

Figura 2.17: Conexiones de red HTTP

Toda la funcionalidad HTTP se encuentra en la interfaz *javax.microedition.io.HttpConnection* la cual añade una serie de facilidades que permitan el uso del protocolo HTTP.

El protocolo HTTP es un protocolo de petición-respuesta en el cual los parámetros de la petición deben establecerse antes de que la petición sea lanzada. En una conexión hay tres estados:

- **Setup:** Antes de que se haya establecido la conexión con el servidor nos encontramos en el estado setup. En este estado se prepara toda la información necesaria para conectar con el servidor, como pueden ser los parámetros de la petición y las cabeceras, lo cual se hace con los métodos *setRequestMethod* o *setRequestProperty*.

- **Connected:** En este estado nos encontramos cuando se haya establecido la conexión con el servidor, dependiendo del método utilizado se enviará en esta etapa la información establecida en el estado setup.

- **Closed:** En este estado estaremos cuando la conexión se haya cerrado y no pueda ser usada más.

Para abrir una conexión *HttpConnection* es necesario indicar la URL completa con la dirección destino a la que nos queremos conectar (bastará con crear un *String* que la contenga y que le pasaremos al método *open* como parámetro) incluyendo el

protocolo, el *host*, el puerto y otros parámetros. Sería *HttpConnection c = (HttpConnection) Connector.open(String URL);*

El protocolo HTTP proporciona un conjunto amplio de cabeceras en la fase de petición para que el MIDlet pueda negociar la forma, el formato, el lenguaje, la sesión y otros atributos más de la petición a realizar al servidor. Para esto tenemos los métodos *setRequestMethod, setRequestProperty,* etc. De entre todas estas cabeceras hay un muy interesantes que son *User-Agent* que permite al cliente identificarse en el servidor, y *Accept-Language* que permite al cliente indicarle al servidor el lenguaje en el que quiere que le devuelva la respuesta.

Una vez establecidos los parámetros y cabeceras de la conexión HTTP podemos usarla en tres modos distintos de conexión, los modos *GET, POST* o *HEAD*. Esto se indica con el método *c.setRequestMethod(HttpConnection.POST).* Los métodos *GET* y *POST* son similares, solo que en aquellos casos en los que sea necesario pasarle al servidor una serie de datos y parámetros adicionales es mejor usar el segundo método. En cambio el método *HEAD* es idéntico al *GET* solo que en el primero no se devuelve un cuerpo de mensaje en la respuesta del servidor.

Para lanzar al servidor una serie de datos en la petición podemos, por ejemplo, hacer:

```
OutputStream os = c.openOutputStream();
os.write("Cadena de prueba".getBytes());
os.flush();
```

Y para recibir los datos que el servidor devuelva:

```
InputStream is = c.openInputStream();
// Lectura carácter a carácter.
```

Por último, el servidor HTTP responde al cliente lanzándole la respuesta a la petición del cliente junto con una serie de cabeceras de respuesta donde se describe el contenido, la codificación, la longitud, etc..

2.7.4.3 Librerías de almacenamiento persistente

La MIDP Specification provee de un mecanismo simple de base de datos para que los MIDlets almacenen datos de forma persistente llamado RMS (Record Management System).

RMS trabaja con *records* y *record stores* de forma que un *record store* es una colección de *records* cuyos datos almacenados se mantienen de lo largo de múltiples invocaciones al MIDlet. Cada *record* es un *array* de bytes, cuya longitud no tiene que ser la misma que la del resto de *records* que haya en ese mismo *record store* y cuyos datos no tienen que ser del mismo tipo dentro de dicho *array* de forma que , por ejemplo en la figura, el Record ID 1 puede contener un *String* seguido de un *int*, mientras que el Record ID 5 puede contener un *array* de de números de tipo *short.*

Cada *record* dentro de un *record store* tiene un recordId único que lo identifica. *Records* adyacentes no tienen por qué tener *recordIds* consecutivos, en incluso no tienen que estar almacenados en memoria de forma consecutiva.

El sistema software del equipo se encarga de mantener la integridad de los *RMS record stores* en el uso normal de la aplicación y también en caso de caídas del sistema por falta de batería o fallos.

Tal y como se mencionó anteriormente, por motivos de seguridad si trabajamos con MIDlet suites solo aquellos MIDlets que pertenezcan al mismo MIDlet suite podrán acceder a los mismos *RMS record stores* no pudiendo acceder a los de otros MIDlet suites.

Toda la funcionalidad de almacenamiento de datos de forma persistente se encuentra en el paquete *javax.microedition.rms*, el cual clases e interfaces que proporcionan un marco de trabajo para los registros, los almacenes y otras características. Así tenemos capacidad para añadir y borrar registros de un almacén, para compartir almacenes por parte de todos los MIDlets de un MIDlet *suite*.

Para representar el *record store* tenemos la clase *RecordStore* que nos permite abrir, cerrar y borrar almacenes de registros, también podemos añadir, recuperar y borrar registros, así como enumerar los registros de un almacén. Los métodos son:

- ***openRecordStore:*** Abre el almacén de registros.
- ***closeRecordStore:*** Cierra el almacén de registros.
- ***deleteRecordStore:*** Borra el almacén de registros.
- ***getName:*** Recupera el nombre del almacén de registros.
- ***getNumRecords:*** Recuperar el número de registros del almacén.
- ***addRecord:*** Añade un registro al almacén de registros.
- ***getRecord:*** Recupera un registro del almacén de registros.
- ***deleteRecord:*** Borra un registro del almacén de registros.
- ***enumerateRecord:*** Obtiene un *enumeration* del almacén de registros.

A parte de la clase *RecordStore* tenemos las siguientes interfaces:

- ***RecordEnumeration:*** Describe una enumeración del almacén de registros.
- ***RecordComparator:*** Describe la comparación de registros.
- ***RecordFilters:*** Describe como usar filtros sobre los registros.
- ***RecordListener:*** Describe un *listener* que recibe notificaciones cuando un registro es añadido, modificado o borrado del almacén de registros.

El primer paso para trabajar con RMS es abrir el almacén de registros usando el método estático *openRecordStore* de la clase *RecordStore*. Con este método también podemos crear un almacén nuevo y abrirlo. *RecordStore rs = RecordStore.openRecordStore(“data”, true);* el primer parámetro indica el nombre del almacén y el segundo indicará si el almacén debe ser creado o si ya existe.

Para añadir un nuevo registro a un almacén, debemos tener antes la información en el formato correcto, es decir, como una matriz de bytes. El siguiente código muestra como añadir un registro:

```
int id = 0;
try{
           id = recordStore.addRecord(bytes, 0,
bytes.length);
}catch (RecordStoreException e){
          e.printStackTrace();
}
```

El primer parámetro es la matriz de bytes, el segundo es el desplazamiento dentro de la matriz y el tercero es el número de bytes a añadir. El método *addRecord* devuelve el identificador del registro añadido, que lo identifica unívocamente en el almacén.

Para recuperar un registro de un almacén utilizamos el método *getRecord* que recupera el registro del almacén a través de su identificador. La información devuelta por este método es una matriz de bytes. Un ejemplo de uso sería:

```
byte[] recordData = null;
try{
            recordData = recordStore.getRecord(id);
}catch (RecordStoreException ex){
          ex.printStackTrace();
}
```

De forma similar a cómo se recuperan registros, se pueden borrar. El método para borrar un registro es *deleteRecord*. Un ejemplo de uso sería:

```
try{
recordStore.deleteRecord(id);
}catch (RecordStoreException ex){
ex.printStackTrace();
}
```

Es importante cerrar el almacén una vez que hemos acabado de trabajar con él. La clase *RecordStore* proporciona el método *closeRecordStore* con este fin. Este método tiene la siguiente forma: *recordStore.closeRecordStore().*