CAPÍTULO 3: J2ME

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 a una entidad, que le proporcionará un determinado recurso, mediante **HTTP**. Para conseguir ese recurso, la citada entidad, tendrá que interaccionar con otras entidades mandándose datos que serán enviados mediante el protocolo de comunicación **SOAP**.

3.1.- 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 que se pueden comprobar en la figura de abajo. Estos productos 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 de entretenimiento 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.

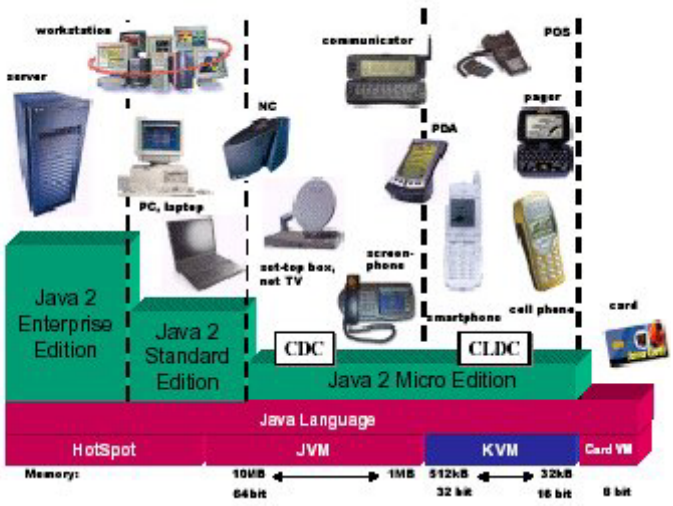

Figura 3. 1: Plataforma J2ME y dispositivos del mercado

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.

3.2.- 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**.

Estos componentes se relacionan de la manera mostrada en la siguiente figura:

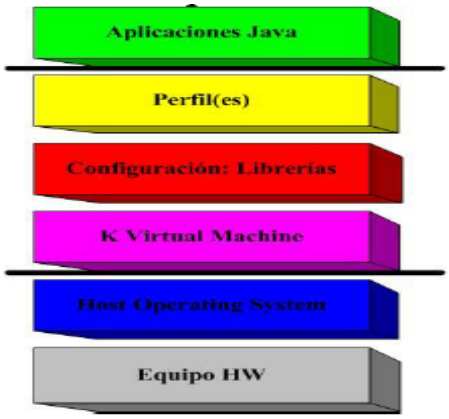

Figura 3. 2: Capas en J2ME

3.2.1.- 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** y todo ello funcionando en un dispositivo con una capacidad de memoria de tan solo unos cuantos centenares de KB. (*Precisamente esto es lo que le da nombre).*

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.

3.2.2.- 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.

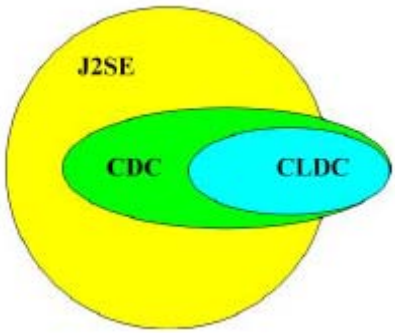

Figura 3. 3: Relación J2SE-J2ME

Ésta incluye un conjunto de librerías mucho mayor que el del anterior, siendo **CLDC** un subconjunto de ésta.

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.*

3.2.3.- 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.*

3.2.4.- 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 que 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.

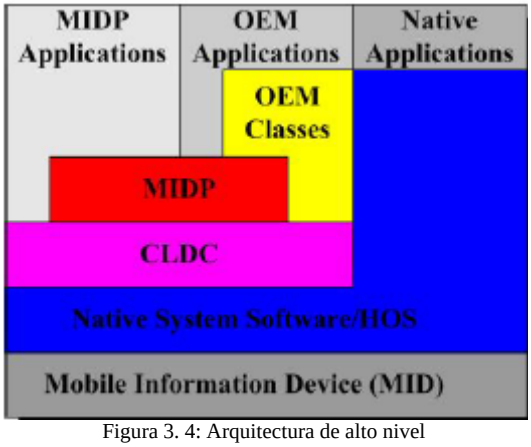

Figura 3. 4: Arquitectura de alto nivel

3.3.- 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 sólo 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.

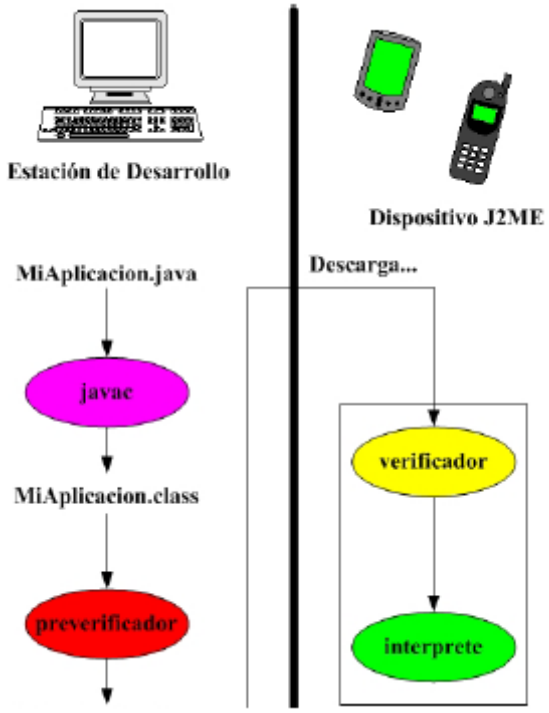

Figura 3. 5: Verificación de clases

Modelo Sandbox

En el modelo **CLDC** y **MIDP** el nivel de seguridad de aplicación se obtiene mediante un **sandbox**. 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, su uso asegura:

• Los ficheros de clases han sido preverificados 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:** Es el tercer nivel en el que se basa el modelo de seguridad explicado. 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.

3.4.- 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 permite 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.*

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.

```
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();
                }
        }
}
```

Figura 3. 6: Ejemplo de un **MIDlet**

- 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***.

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 fin 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:

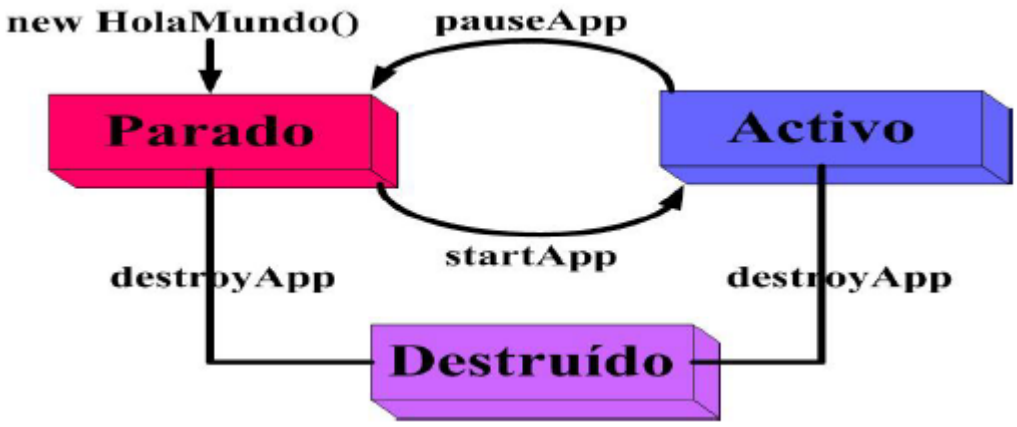

Figura 3. 7: Estados y transiciones de un MIDlet

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.*

3.5.- 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.

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**.

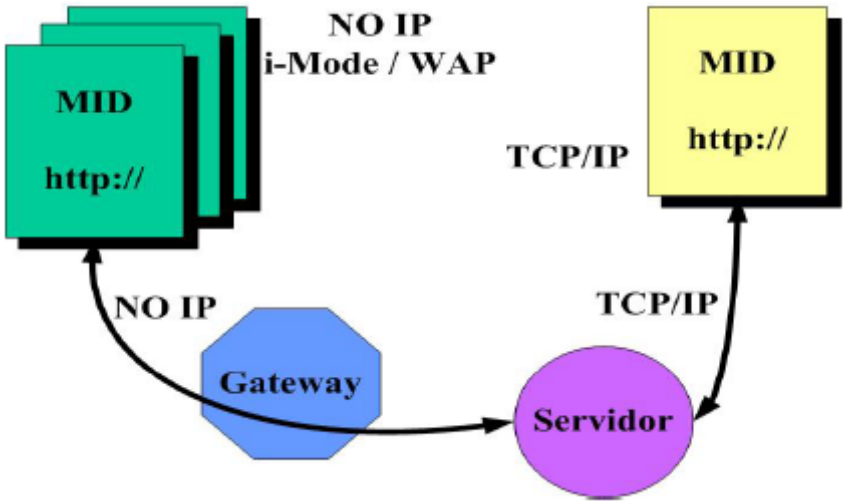

Figura 3. 8: Conexiones de red HTTP

Toda la funcionalidad **HTTP** se encuentra en la interfaz **javax.microedition.io.HttpConnection** la cual añade una serie de facilidades que permiten 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);
```

Figura 3. 9: Apertura de **conexión HTTP**

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 unas 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, sólo 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** con la diferencia 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.
```

Figura 3.10: Escritura y lectura mediante una conexión HTTP

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.