Universidad de Murcia
Facultad de Informática
Departamento de Ingeniería de la Información y las Comunicaciones

Proyecto Fin de Carrera

Integración de Tarjetas Criptográficas en Dispositivos Móviles *J2ME*

Autor
Félix Gómez Mármol

Directores del proyecto
D. Daniel Sánchez Martínez
Dr. Antonio F. Gómez Skarmeta

Murcia, septiembre de 2006

A mis padres

A mi hermano Alberto

A mis compañeros y amigos

*A **Dani**, mi director de proyecto*

Porque sin su ayuda, sus consejos, su apoyo y su ánimo,
sin duda este proyecto no habría sido lo mismo.
¡Muchas Gracias Dani!

Murcia, a 8 de septiembre de 2006

Índice general

Índice de Figuras

Índice de Tablas

Abstract

En este proyecto se han analizado distintos escenarios de integración de tarjetas inteligentes o *smart cards* con capacidades criptográficas en dispositivos móviles tales como PDAs o *smart phones*, haciendo uso del lenguaje J2ME (Java 2 Micro Edition).

Esta integración dotará de una mayor seguridad a los dispositivos móviles, ya que en esta nueva situación, toda la información confidencial del usuario se encuentra almacenada en la tarjeta inteligente que él posee y no directamente en el dispositivo, como ocurría hasta ahora.

En el primero de estos escenarios se cuenta con una tarjeta inteligente externa al dispositivo y un lector de tarjetas SDIO. El segundo consiste también en una tarjeta inteligente externa, pero esta vez con un lector de tarjetas Bluetooth. Y por último, en el tercer escenario expuesto en este proyecto, la tarjeta inteligente se encuentra dentro del dispositivo, ya que dicha tarjeta es el propio módulo SIM que contienen los teléfonos móviles (pero con capacidades criptográficas, es decir, una tarjeta WIM).

Por todo esto, en primer lugar se ha realizado un profundo análisis de toda la tecnología, protocolos y estándares necesarios para llevar a cabo un diseño y posterior implementación de cada uno de dichos escenarios.

A continuación se han desarrollado propuestas de diseño para cada uno de los escenarios expuestos y se han implementado total o parcialmente algunos de ellos. En el apartado de conclusiones y vías futuras se resaltarán las principales ventajas e inconvenientes de esos tres escenarios.

Introducción

I. Presentación

Hoy en día, la proliferación de dispositivos móviles tales como teléfonos móviles de última generación, PDA's, *smart phones*, etc. convierten el panorama actual en un nicho de trabajo perfecto para el desarrollo de elementos que vayan dirigidos a tratar de securizar cualquier tipo transmisión, transacción u operación electrónica en este tipo de dispositivos.

Bajo esta perspectiva, en este proyecto se pretende analizar las posibilidades de integración de una tarjeta criptográfica en un entorno móvil J2ME, presentando posibles diseños e implementando alguno(s) de ellos.

Además, el proyecto se enmarca dentro del proyecto FACTO de Telefónica Móviles, donde ya se han diseñado e implementado algunas arquitecturas de seguridad para sistemas Pocket PC/Smartphone (incluyendo el acceso a la tarjeta).

Este ambicioso proyecto tiene un gran interés tecnológico de por sí, pero más aún si tenemos en cuenta aspectos tales como la inminente implantación del nuevo DNI electrónico [20] en todo el territorio nacional, ya que dicho DNI no será sino una tarjeta inteligente que, haciendo uso del estándar PKCS#15, contendrá (entre otros tantos elementos) un par de certificados digitales.

La expansión de las redes de tercera generación y la gran penetración de este tipo de terminales en la sociedad que nos rodea, hace indispensable generar los mecanismos técnicos adecuados que integren la utilización del nuevo DNI en este tipo de dispositivos.

La implantación del DNI electrónico, prevista para los años 2006 y 2007, además de proporcionar nuevas utilidades a los ciudadanos, supondrá un paso importante en el desarrollo de las nuevas tecnologías y la sociedad de la información en nuestro país.

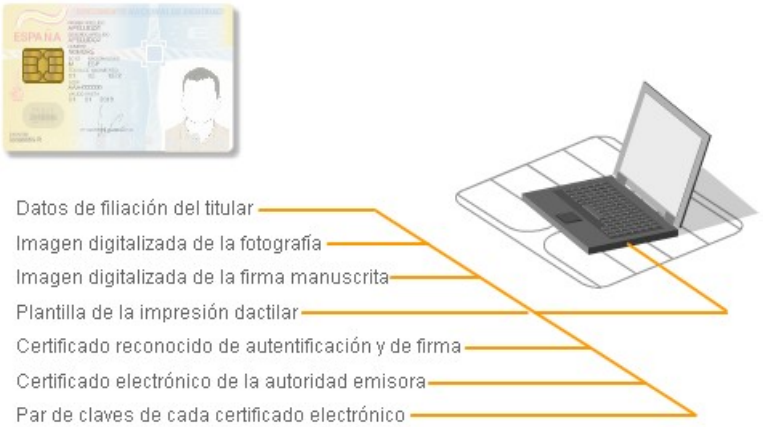

Figura 1: DNI electrónico

En el anverso del nuevo DNI-e, un chip contendrá dos certificados electrónicos (uno de identidad y otro de firma) que servirán para autenticar la personalidad del titular y realizar transacciones de firma electrónica, además de la huella en formato digital, la fotografía digitalizada, la imagen digitalizada de la firma manuscrita y los datos impresos en la tarjeta.

Los usuarios necesitarán un lector de tarjetas inteligentes y una serie de componentes software de criptografía para poder hacer uso del DNI electrónico en los distintos servicios de administración electrónica que sean desarrollados en el futuro.

Además este contexto tecnológico propicia la aparición de nuevos servicios de comercio electrónico basados en el uso de esta tarjeta inteligente que poseerán todos los ciudadanos, impulsando así el negocio de las empresas del sector TIC.

II. Trabajo previo

Como ya hemos comentado antes, este proyecto se enmarca dentro de otro proyecto más grande, denominado FaCTo [2, 3] (Firma electrónica y servicios de certificación en terminales móviles). El proyecto FacTo, que comenzó su andadura allá por el 2003, es un proyecto de investigación en el que colaboran la Universidad de Murcia y Telefónica Móviles de España.

En dicho proyecto se analizaron inicialmente las distintas alternativas para decidir cuál sería el punto de partida óptimo en la tarea de tratar de garantizar algunos requisitos de seguridad tales como no repudio, autenticación e integridad en dispositivos móviles, haciendo uso de los estándares técnicos internacionales sobre seguridad.

El resultado de este análisis fue que ni SATK [1], ni WAP [1], ni J2ME [4], sino *Windows Mobile for Pocket PC* era el marco de trabajo inicial más adecuado. Los principales motivos que condujeron a esta decisión fueron los siguientes:

1. La arquitectura de Windows Mobile es bastante similar a la ofrecida en la versión para PC, por lo que se supuso que la adaptación de una plataforma a otra no debería ser demasiado "traumática".

2. En aquel momento ya existían numerosas librerías y APIs para Windows Mobile, por lo que no se tendría que empezar a trabajar desde cero.

3. Habría sido un error negar la obviedad de que la mayoría de dispositivos existentes en el mercado incorporaban Windows Mobile como sistema operativo y no otro.

4. No se podía recuperar la información criptográfica, tal como los certificados digitales, de las tarjetas inteligentes porque por aquel entonces los lectores de tarjetas para dispositivos móviles casi no existían o estaban en las primeras fases de desarrollo.

Así, se creó una infraestructura y una serie de componentes que permitían el desarrollo de aplicaciones basadas en firma para dispositivos móviles que usaban Windows Mobile.

Dicha infraestructura, que se observa en la figura 2, cuenta con ActiveX y servicios web como componentes principales. ActiveX proporciona las operaciones relacionadas con el proceso de petición y obtención de un certificado, la firma y verificación de documentos (con y sin marca de tiempo) y la validación de un certificado a través de un cliente web.

En la parte del servidor existen aplicaciones (publicación, revocación y renovación de certificados) y servicios web (validación de certificados y marcas de tiempo).

Todos estos componentes fueron aplicados con éxito en la construcción de aplicaciones en entornos reales. Y como trabajo futuro se planteó la posibilidad de incorporar la utilización de tarjetas inteligentes a esta infraestructura, con lo que se dotaría a la misma de un alto grado de seguridad, ya que la información privada del usuario final (su clave privada y su certificado) estarían almacenados en un dispositivo seguro.

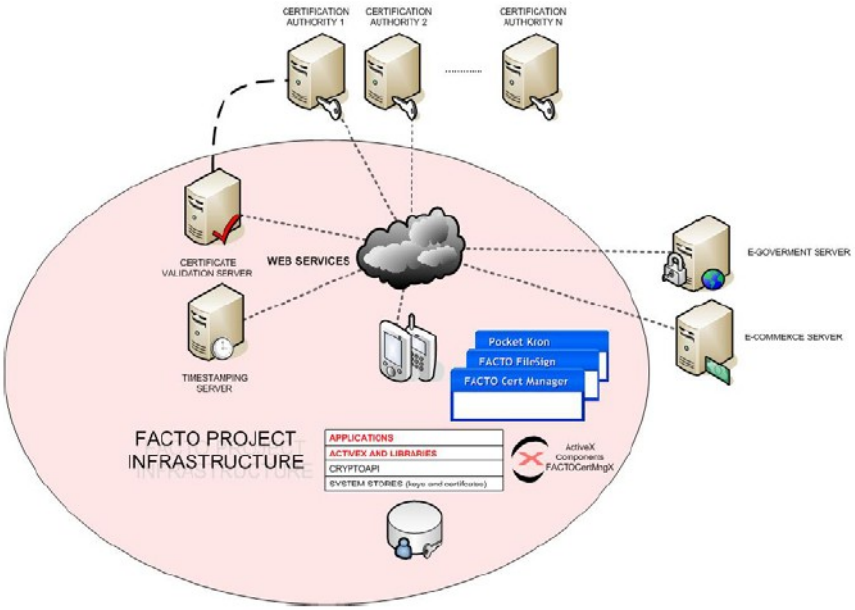

Figura 2: Infraestructura y componentes del proyecto FaCTo

III. Motivación y objetivos

Siguiendo con la línea de trabajo inicial, todo lo realizado hasta ahora con el propósito de integrar las tarjetas inteligentes en los dispositivos móviles se ha llevado a cabo sobre plataformas Windows CE - Pocket PC/Smartphone, utilizando las tecnologías de programación y las APIs de desarrollo que Microsoft ofrece en su sistema operativo.

Así, la Universidad de Murcia desarrolló una librería de acceso a la tarjeta para PC que cumplía con el estándar internacionalmente aceptado PKCS#11.

Figura 3: Ámbito y entorno actuales

Con este proyecto queremos abrir una nueva línea de investigación, orientada a la integración de la tarjeta inteligente en dispositivos móviles, pero desde un entorno J2ME.

J2ME es otra "gran tecnología" de programación para dispositivos móviles y, por tanto, es un tema que resulta interesante abordar. Algunas de las ventajas más destacables de trabajar con J2ME son: la portabilidad de código y la independencia del sistema operativo que el uso de Java implica, la inclusión de una JVM por parte del fabricante en la mayoría de dispositivos móviles, así como una extensa comunidad de desarrolladores en este lenguaje.

Además, el empleo de estándares asegura una interoperabilidad entre distintos fabricantes, tanto de tarjetas inteligentes, como de lectores de tarjetas, como de dispositivos móviles en general. Por lo tanto, todas las soluciones que se desarrollen bajo este proyecto serán, en buena medida, de fácil extensión e implantación en una amplia gama de dispositivos.

Este proyecto tiene, si cabe, especial interés en nuestro país, ya que según el Ministerio del Interior, está previsto que para el 1 de abril de 2008 se esté emitiendo el nuevo DNI electrónico, del que ya hemos hablado, en todas las oficinas de expedición del territorio nacional.

Por lo tanto, todos aquellos desarrollos que estén orientados a la integración de tarjetas inteligentes, como lo será el DNI-e, serán de gran utilidad para toda la sociedad y por consiguiente, bien recibidos por instituciones, empresas y particulares.

De este modo, en este proyecto se plantearán tres escenarios posibles de integración de tarjetas inteligentes en dispositivos móviles, analizando y estudiando posibles diseños para todos y cada uno de ellos. Estos tres escenarios serán:

1. **Tarjeta externa y lector SDIO**

   En este primer escenario se hace uso de un lector de tarjetas SDIO como el que se muestra en el apéndice D de hardware. SDIO es un tipo de conexión que permite comunicar el dispositivo móvil con otros dispositivos externos (en nuestro caso, un lector de tarjetas).

   Todos los desarrollos realizados hasta el momento en este sentido dentro del proyecto FaCTo se basan en el CSP de Microsoft, por lo que son incompatibles con aquellos dispositivos que no incluyan su sistema operativo.

   La mejor solución que hemos encontrado para este problema concreto pasa por emplear el estándar de acceso a tokens criptográficos (tales como una tarjeta inteligente) PKCS#11 [10]. Dicho estándar está ampliamente aceptado y es soportado por la gran mayoría de dispositivos y aplicaciones existentes hoy en día.

2. **Tarjeta externa y lector Bluetooth**

   Un tipo de lector de tarjetas inteligentes para dispositivos móviles que más puede extender su uso es sin duda un lector Bluetooth [7], ya que el número de dispositivos presentes en el mercado que soportan Bluetooth es muy elevado.

   Los desarrollos criptográficos llevados a cabo hasta ahora en FaCTo, que utilizaban tarjetas inteligentes para ello, hacen uso del lector Tactel S300. Este lector va conectado a la ranura SDIO del Pocket PC, lo que hace que esté limitado el tipo de dispositivo Windows Mobile ya que es requisito imprescindible que éste tenga una ranura SDIO. Si se desarrolla la integración del uso del lector de TI Bluetooth con las aplicaciones de firma de FaCTo en Windows CE, utilizando J2ME, se ampliaría el número de terminales con Windows Mobile que pudieran hacer uso de firma electrónica con tarjeta inteligente.

3. **Tarjeta interna WIM**

   Para poder realizar procesos de firma digital a través de una tarjeta SIM se requiere que dicha tarjeta cumpla una serie de requisitos técnicos entre los que cabe destacar que sea JavaCard criptográfica y que tenga la estructura WIM/PKCS#15 [12, 13]. Dicha estructura es una especificación de seguridad que permite la definición, almacenamiento y gestión de credenciales criptográficas (claves privadas, certificados, permisos) de una forma estandarizada, y sin exponerse al exterior.

   Estas características resultan fundamentales para integrar el uso de credenciales criptográficas en aplicaciones seguras. En la actualidad las tarjetas SIM son capaces de soportar esta funcionalidad, conteniendo una aplicación WIM (applet) capaz de realizar toda esta gestión criptográfica.

   En una tarjeta SIM con funcionalidad WIM, los pares de claves se generan dentro de la propia tarjeta, los procesos de firma son realizados por la propia aplicación WIM, y las claves privadas nunca abandonan la tarjeta. Es por ello que una tarjeta SIM/WIM se considera una tarjeta SIM criptográfica.

   Es posible realizar operaciones de firma digital a través del módulo SIM, si éste las soporta, en una aplicación J2ME que se ejecuta en el dispositivo móvil. Esta funcionalidad es factible gracias a la API JSR177 de J2ME, más conocida como SATSA [28].

Dicho todo esto, a continuación se presentan cada uno de estos escenarios, con sus correspondientes apartados de análisis y diseño. El primer escenario contará además, con un apartado de implementación.

Escenario I

Tarjeta Externa y Lector SDIO

Visión General

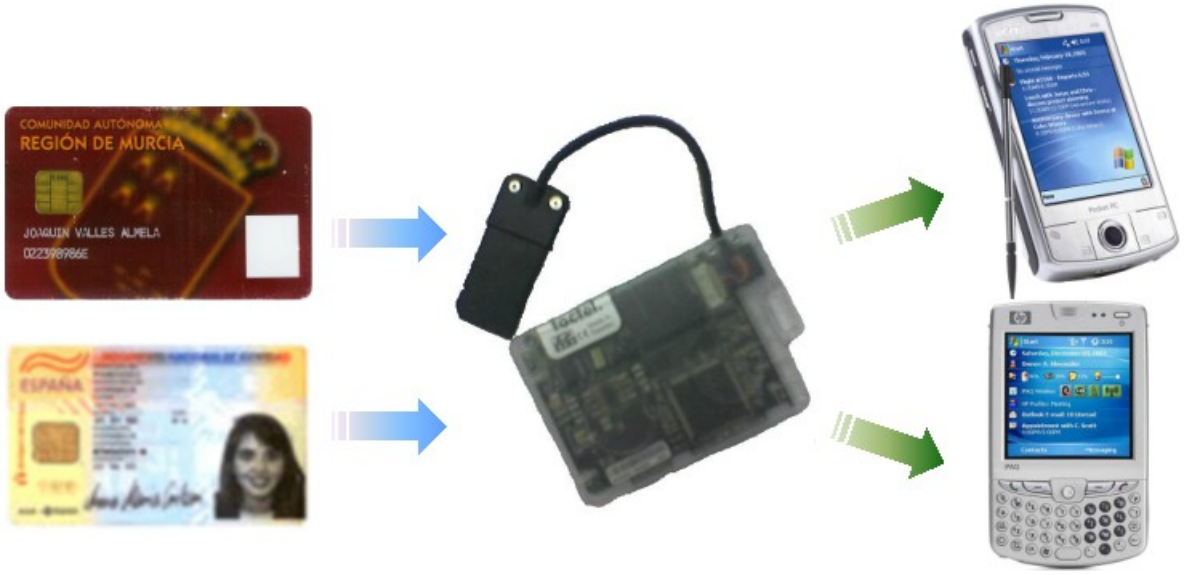

Figura I.1: Primer escenario: tarjeta externa y lector SDIO

En este primer escenario disponemos de una tarjeta inteligente como la de la Universidad de Murcia o (potencialmente) el propio DNI-e, un lector de tarjetas SDIO y una PDA o un *Smartphone* (una PDA con funcionalidad telefónica).

Para llevar a cabo una solución basada en una tarjeta externa y un lector SDIO, en la fase de **análisis** se estudiarán: el lenguaje J2ME, algunas máquinas virtuales para dispositivos móviles, la librería criptográfica Java de IAIK y el estándar PKCS#11.

En primer lugar se realizará una breve presentación de J2ME, su arquitectura y sus características principales (configuraciones y perfiles).

A continuación tendrá lugar la búsqueda de la máquina virtual de Java que mejor se adapte a las necesidades de este proyecto. Para acceder a la tarjeta a través del lector SDIO es necesario desarrollar unas librerías nativas propias del sistema operativo subyacente, que en nuestro caso será Windows Mobile 2003. Por lo tanto, necesitaremos que nuestra máquina virtual tenga soporte, como mínimo, para JNI y que pueda correr sobre Windows Mobile 2003. Así, se analizarán diversas máquinas virtuales con sus ventajas e inconvenientes.

El siguiente punto a tratar consistirá en la evaluación del software criptográfico *IAIK PKCS#11 Provider Micro Edition* [9]. Dicha evaluación incluirá las características básicas del mismo, ejemplos de utilización y ventajas e inconvenientes, entre otros detalles. Esta librería nos proporcionará una API por encima del módulo PKCS#11 que desarrollemos, de tal modo que nos permitirá abstraernos de dicho módulo a la hora de realizar operaciones criptográficas en J2ME. Otra alternativa habría sido construir nosotros mismos esa API.

También se harán algunas pruebas con dicha librería de funciones criptográficas en entorno PC haciendo uso del módulo `pkcs11.dll` desarrollado por la Universidad de Murcia.

Como última sección del análisis referente a este escenario, se mostrará una breve introducción al estándar PKCS#11 [10], explicando en qué consiste dicho estándar, cómo actúa y cuáles son sus funciones y características principales, teniendo siempre presente que el empleo de estándares reconocidos por toda la comunidad científica, como lo es éste, posibilita la aplicación de los desarrollos creados en una amplia gama de dispositivos.

En la fase de **diseño**, se realizará un diseño completo y riguroso del escenario en cuestión, mostrando la arquitectura en capas del mismo, con todos sus componentes y explicando qué se deberá hacer en la fase de implementación para obtener cada uno de éstos.

Como se verá en su momento, dicha arquitectura constará básicamente de: una librería dll que implemente el estándar PKCS#11 y que esté compilada para dispositivos móviles, un máquina virtual de Java que, como mínimo, soporte JNI y corra sobre Windows Mobile 2003, una librería criptográfica construida a partir de la librería de IAIK y alguna aplicación J2ME que, haciendo uso de dicha librería, acceda a la tarjeta criptográfica y sea capaz de realizar algunas operaciones tales como firma, verificación de firma, cifrado y descifrado, entre otras.

Por último, en la fase de **implementación** de este escenario, partiendo de la librería de enlace dinámico desarrollada por la Universidad de Murcia `pkcs11.dll` para entorno PC, se tratará de modificar ese mismo código con el objetivo de poder compilarlo para dispositivos móviles.

El entorno de desarrollo, como veremos más adelante, será Microsoft eMbedded Visual C++ 4.0. Se mostrarán cuáles son las configuraciones necesarias para compilar correctamente el código dado y se verán algunos detalles básicos de su funcionamiento.

En este apartado también se describirán las principales modificaciones del código que en cuanto a incompatibilidades de tipos de datos se refiere.

Como veremos en su momento, este proceso de compilación constará básicamente de dos partes: la compilación de la librería de acceso a la tarjeta `MDFsc8Kdll.dll` y la compilación de la librería que implementa el estándar PKCS#11 para dispositivos móviles propiamente dicha, `mpkcs11.dll`.

Una vez se haya conseguido compilar correctamente la librería `mpkcs11.dll` y antes de pasar al siguiente objetivo, se realizarán algunas pruebas de dicha librería ya en un dispositivo móvil.

El siguiente paso será instalar en el dispositivo móvil aquella máquina virtual que en la fase de diseño se haya decidido, basándose en el análisis previo de varias máquinas virtuales para dispositivos móviles.

A continuación de desarrollará una librería criptográfica a partir de la de IAIK que servirá de soporte a otras aplicaciones J2ME que quieran acceder al contenido de la *smart card* y realizar operaciones criptográficas.

Por último, se desarrollarán algunas aplicaciones J2ME piloto que integren todos los componentes anteriormente obtenidos y que muestren el correcto funcionamiento de los mismos.

Análisis

I.1. Breve introducción a J2ME

Todo comenzó con una versión de Java (ahora conocida como *Java 2 Standard Edition*) y el lema "Escríbelo una vez, ejecútalo donde quieras ^{TM} ". La idea era desarrollar un lenguaje en el cual uno pudiera escribir su código una vez, y ejecutarlo en cualquier plataforma que soportara la máquina virtual de Java.

Desde su lanzamiento en 1995, el panorama ha cambiado significativamente. Java ha extendido su alcance más allá de los ordenadores de sobremesa. Dos años después de la introducción de Java, se publicó una nueva edición, *Java 2 Enterprise Edition*. Y la incorporación más reciente a esta familia es *Java 2 Micro Edition* [4, 5, 24, 25].

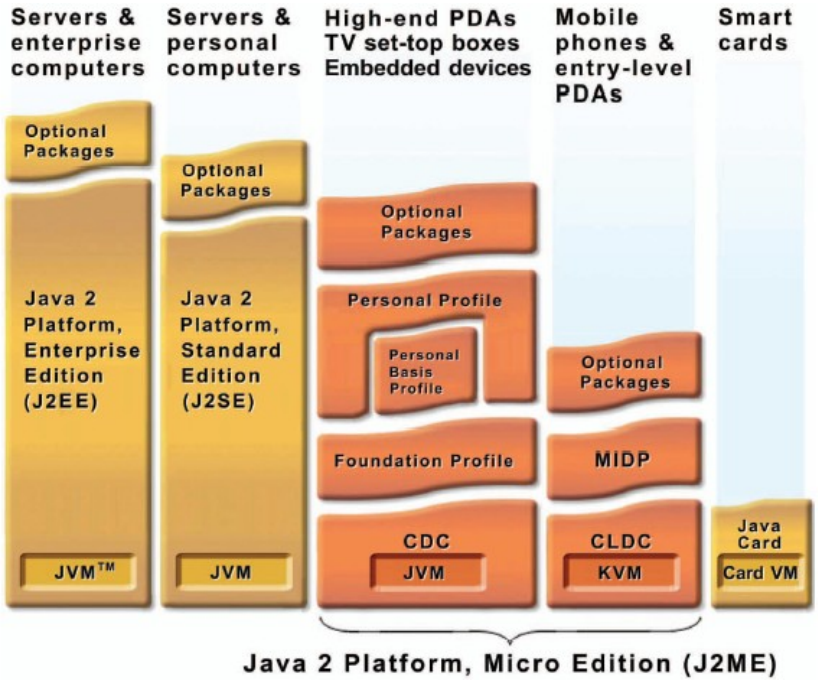

Figura I.2: Plataformas de Java: J2EE, J2SE, J2ME y Java Card

J2ME está totalmente orientado a dispositivos con capacidades limitadas. Muchos de estos dispositivos (tales como un teléfono móvil o una PDA) no permiten descargar e instalar software más allá del que fue configurado en el proceso de fabricación. Con la introducción de J2ME, los "micro" dispositivos son ahora capaces de navegar, descargar e instalar aplicaciones Java y otro tipo de contenido.

Estos dispositivos de bajo consumo han cambiado nuestras vidas. Los teléfonos móviles nos permiten comunicarnos cuando estamos fuera de casa o del trabajo. Las PDAs nos permiten acceder a nuestro correo electrónico, navegar por internet y ejecutar aplicaciones de cualquier índole.

Con la introducción de Java en dichos dispositivos, tenemos acceso a las características inherentes al lenguaje y a la plataforma Java. Esto es, un lenguaje de programación fácil de aprender y manejar, un entorno de ejecución que provee una plataforma portable y segura y acceso a contenido dinámico, por no mencionar la comunidad estimada de más de 2 millones de desarrolladores.

Aunque estaría genial contar con la API completa de J2SE disponible en un dispositivo móvil, no es algo realista. Por ejemplo, un teléfono móvil con su pantalla limitada no puede soportar toda la funcionalidad disponible en la AWT, que es la principal interfaz gráfica de usuario (GUI) que da Java. J2ME se introdujo precisamente dirigido hacia todos esos dispositivos que caen fuera del ámbito de J2SE y J2EE.

Las capacidades de todo este tipo de dispositivos pueden variar mucho de unos a otros. Por ejemplo, un teléfono móvil y una PDA están los dos limitados en tamaño, sin embargo un teléfono móvil "típico" puede tener una pantalla con una resolución total de unos 12288 (96×128) pixeles, mientras que la resolución de una PDA puede rondar de los 20000 pixeles en adelante.

Una única plataforma de Java claramente no encajaría adecuadamente con todos estos dispositivos. Es por ello que J2ME introduce dos nuevos conceptos, las configuraciones y los perfiles.

I.1.1. Configuraciones

Una configuración define una plataforma Java para un amplio rango de dispositivos. Está directamente relacionada con una JVM. De hecho, una configuración específica define las características del lenguaje Java y las librerías de la JVM que serán utilizadas.

La decisión acerca de qué configuración aplicar sobre un dispositivo se basa principalmente en la disponibilidad y capacidades de memoria, pantalla, conexión de red y procesador de dicho dispositivo.

Las características típicas de aquellos dispositivos que se ajustan a cada una de las actuales configuraciones son:

- CDC. *Connected Device Configuration.*
  - Un mínimo de 512 kilobytes de memoria para ejecutar Java
  - Un mínimo de 256 kilobytes de memoria para ejecución de programas
  - Conexión de red, posiblemente persistente y con gran ancho de banda
- CLDC. *Connected Limited Device Configuration.*
  - 128 kilobytes de memoria para ejecutar Java
  - 32 kilobytes de memoria para ejecución de programas
  - Una GUI restringida
  - Típicamente con suministro eléctrico a través de baterías
  - Conexión de red, típicamente inalámbrica, con bajo ancho de banda y acceso intermitente

Sin embargo, aunque esta división parezca bastante clara, en la mayoría de las ocasiones no será así, dado que la tecnología está continuamente cambiando (avanzando). La cuestión es que, conforme la tecnología se vaya desarrollando y se vaya ofreciendo cada vez más capacidad de procesamiento, más memoria y pantallas con mejores capacidades, el solapamiento entre ambas categorías (CLDC y CDC) será también mayor.

I.1.2. Perfiles

Un perfil es, si se quiere ver así, como una extensión de una configuración. Proporciona a un programador las librerías necesarias para desarrollar una aplicación para un tipo de dispositivo en particular. Por ejemplo, MIDP define APIs para componentes de interfaz de usuario, manejo de entrada de datos y eventos, almacenamiento persistente, comunicaciones y temporizadores, todo ello teniendo en cuenta las limitaciones de pantalla y memoria de los dispositivos móviles.

I.1.3. Arquitectura de J2ME

En la figura I.3 se muestra una arquitectura genérica de J2ME. Como se puede comprobar, tiene como base al sistema operativo, seguido por la JVM. Esta JVM será la KVM si el dispositivo se ajusta más a la CLDC, mientras que si el dispositivo se ajusta mejor a la CDC, entonces será la JVM "tradicional" usada en J2SE.

Por encima de la JVM se encuentra una determinada configuración, y por encima de ésta, un perfil concreto.

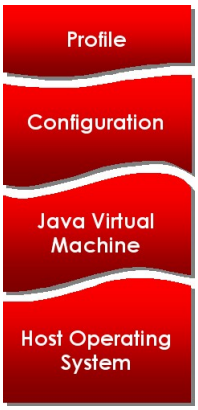

Figura I.3: Arquitectura genérica de J2ME

De hecho en la figura I.4 se muestran las arquitecturas de J2ME para dispositivos que se ajusten a la CLDC y para dispositivos que se ajusten a la CDC.

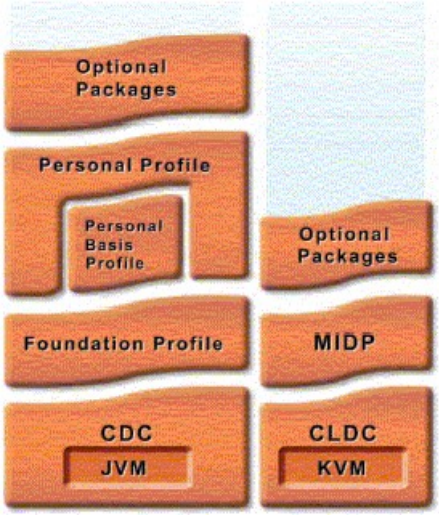

Figura I.4: Arquitecturas CDC y CLDC de J2ME

Entre los paquetes opcionales que se pueden utilizar sobre CLDC se encuentran PIM y FileConnection. Por otra parte FP proporciona versiones J2SE completas de `java.lang`, `java.util`, `java.net`, `java.io`, `java.text` y `java.security`. PBP proporciona un AWT básico, RMI y JavaBeans. Y PP proporciona un AWT completo, applets y `java.math`.

I.1.4. KVM, CLDC y MIDP

Para CLDC Sun ha desarrollado la que se ha considerado como la implementación de referencia de la máquina virtual de Java, conocida como KVM. Esta KVM fue diseñada para adaptarse a las restricciones específicas de los dispositivos móviles de bajas/medias prestaciones.

Las principales características que distinguen a la KVM son:

- La máquina virtual por sí misma sólo requiere entre 40 y 80 kilobytes de memoria.
- La memoria dinámica (o pila) sólo requiere entre 20 y 40 kilobytes.
- Puede ejecutarse sobre procesadores de 16 bits a 25 MHz.
- No soporta matemáticas en punto flotante. Supone que muchos dispositivos no tendrán el hardware necesario para manejar números en punto flotante.
- No soporta JNI. Se pretende reducir la currupción potencial de la información a nivel del sistema, así como los requerimientos de memoria.
- Debe tener un cargador de clases propio. Este cargador de clases es dependiente del dispositivo, esto es, es definido e implementado por el fabricante de cada dispositivo. Esta dependencia incluye la forma en que son cargadas las clases así como el manejo de las condiciones de error.
- No soporta reflexión (metaclases).
- No soporta grupos de hilos.
- No soporta la finalización. En J2SE, mediante el método `finalize()` se liberan los recursos del sistema adquiridos por un objeto después de que el recolector de basura haya liberado la memoria usada por dicho objeto.
- No soporta referencias débiles.

En cuanto a las clases de CLDC, el GCF es un conjunto de clases e interfaces diseñado para facilitar el acceso a sistemas de almacenamiento y conexión, sin necesidad de especificar ningún requisito software o hardware.

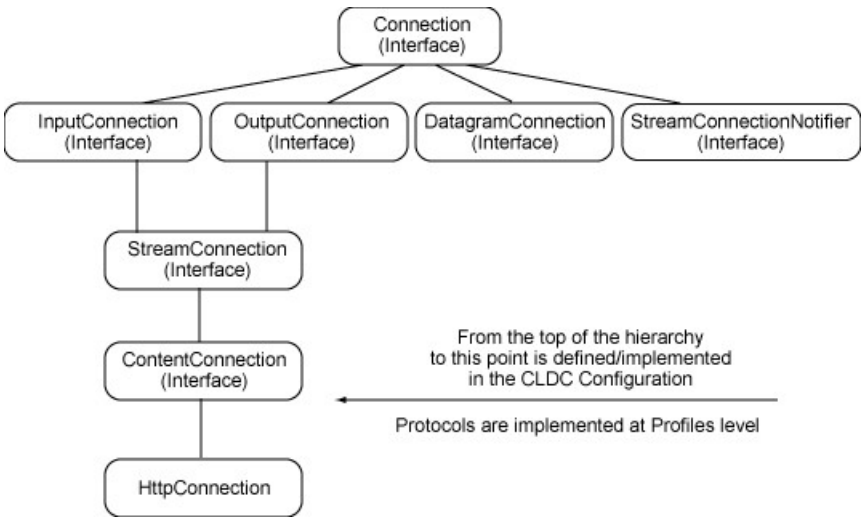

Figura I.5: Jerarquía de interfaces del GCF

Lo primero que se debe tener claro en cuanto al GCF es que éste no incluye implementaciones de ningún protocolo, simplemente sirve de soporte para ello. Las clases que verdaderamente llevan a cabo la implementación de dichos protocolos se encuentran ya en cada perfil. Por ejemplo, cualquier fabricante de dispositivos que dé soporte a MIDP debe implementar el protocolo HTTP.

La clase `Connector` contenida en el GCF permite abrir cualquier tipo de conexión, como por ejemplo:

```
Connector.open("http://www.um.es");
```

Si dicha llamada tiene éxito, se devuelve un objeto que implementa alguna de las seis interfaces disponibles definidas en el GCF:

```
javax.microedition.io.InputConnection
javax.microedition.io.OutputConnection
javax.microedition.io.StreamConnection
javax.microedition.io.ContentConnection
javax.microedition.io.StreamConnectionNotifier
javax.microedition.io.DatagramConnection
```

Por otra parte, el número de propiedades del sistema que se pueden consultar con CLDC también está limitado. En concreto, se trata de éstas cuatro:

1. Plataforma o dispositivo actual.

   ```
   System.getProperty("microedition.platform");
   ```

2. Codificación por defecto de caracteres.

   ```
   System.getProperty("microedition.encoding");
   ```

3. Nombre y versión de las configuraciones soportadas.

   ```
   System.getProperty("microedition.configuration");
   ```

4. Nombre de los perfiles soportados.

   ```
   System.getProperty("microedition.profiles");
   ```

La figura I.6 muestra la jerarquía de clases e interfaces del perfil MIDP. En el apéndice A se ampliará la información acerca de este perfil.

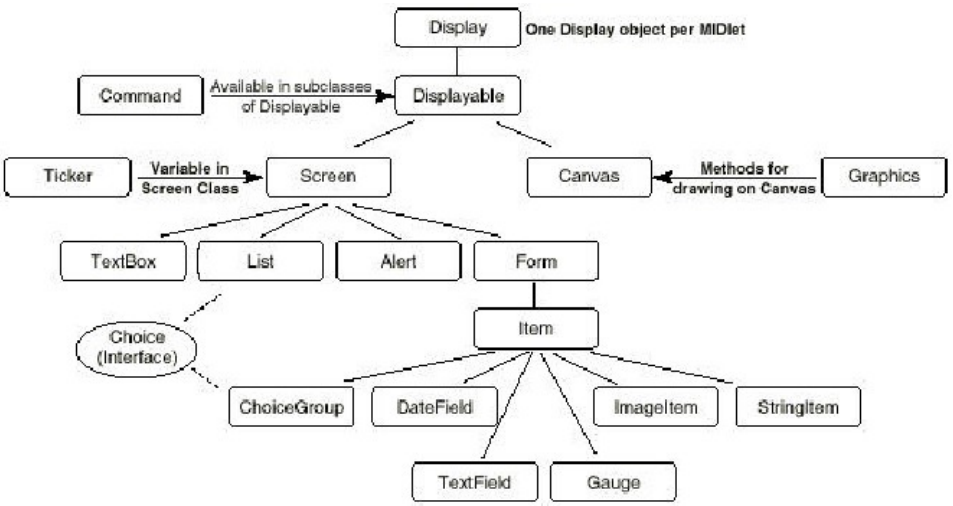

Figura I.6: Jerarquía de clases e interfaces de MIDP

I.1.5. JVM, CDC, FP, PBP y PP

En la arquitectura de CDC ya no se tiene restricciones sobre la JVM, ya que CDC debe incluir soporte total de la especificación del lenguaje Java, así como de la especificación de la JVM. Las únicas restricciones en cuanto a la JVM las pondrá el propio dispositivo con sus capacidades de almacenamiento y procesamiento.

CLDC es un subconjunto de CDC. Dicho de otro modo, CDC incluye todas las APIs definidas en CLDC, incluyendo también todos los paquetes `java.*` y `javax.microedition.*`. Fue diseñado paran dispositivos con más capacidad de memoria (2 MB de memoria, o más, para la plataforma Java) y mejor conectividad (de 9600 bps en adelante).

Toda implementación de CDC debe ofrecer soporte para la E/S de ficheros y datagramas. Los paquetes de CDC fueron tomados del J2SE 1.3, quitando las APIs desaconsejadas (*deprecated*). Y así, los paquetes resultantes fueron los siguientes:

- `java.io`
- `java.lang`
- `java.lang.ref`
- `java.lang.reflect`
- `java.math`
- `java.net`
- `java.security`
- `java.security.cert`
- `java.text`
- `java.util`
- `java.util.jar`
- `java.util.zip`
- `javax.microedition.io`

Foundation Profile

Foundation Profile, desarrollado por el JCP como JSR46 [26], es un perfil CDC. Este perfil pretende ser usado en dispositivos que requieren una implementación completa de la JVM así como todas las APIs de J2SE. Las implementaciones típicas usarán un subconjunto de todas esas APIs dependiendo de los perfiles que adicionalmente se soporten.

Cualquier implementación de FP, además de incluir todos los protocolos contenidos en CDC, debe soportar también sockets y el protocolo HTTP.

Personal Basis Profile

El PBP, desarrollado por el JCP como JSR129, está orientado a aplicaciones donde se requiere un soporte completo de funcionalidad, pero sin GUI. Proporciona la capacidad de presentación de interfaces gráficas de usuario básicas, pero no está diseñado para soportar aplicaciones que requieran una GUI completa. El PBP sólo soporta componentes ligeros (*light-weight*) de la AWT.

Las principales diferencias entre el PBP y PersonalJava son:

- PersonalJava está orientada al JDK 1.1.8, mientras que PBP está orientado al JDK 1.3.
- En PersonalJava, la inclusión de RMI es opcional, mientras que en PBP, RMI es soportado en un paquete concreto.
- En PBP no se incluyen una serie de APIs desaconsejadas.

Personal Profile

El PP, desarrollado por el JCP como JSR62, es la vía de migración a J2ME de las aplicaciones desarrolladas con PersonalJava. PP es el perfil CDC orientado hacia las PDAs. Las aplicaciones escritas con el PP son compatibles (en dirección ascendente) con el J2SE JDK 1.3.

Está diseñado para desarrollar aplicaciones que requieran una AWT JDK 1.1 completa. Así, define tres modelos de aplicación:

- Applets. Applets JDK 1.1 estándar.
- Xlets. Un Xlet es una interfaz de gestión del ciclo de vida. Una aplicación gestora controla un Xlet mediante métodos definidos en esta interfaz. Dicha aplicación provoca que el Xlet cambie de estado a alguno de los siguientes: Destruido, Pausado y Activo.
- Aplicaciones. Una aplicación Java estándar, definida como una clase con un método `public static void main(String[])`.

El PP añade al FP los siguientes paquetes:

- `java.applet`
- `java.awt`
- `java.awt.color`
- `java.awt.datatransfer`
- `java.awt.event`
- `java.awt.image`
- `java.beans`
- `java.math`
- `java.rmi`
- `java.rmi.registry`
- `javax.microedition.xlet`
- `javax.microedition.xlet.ixc`

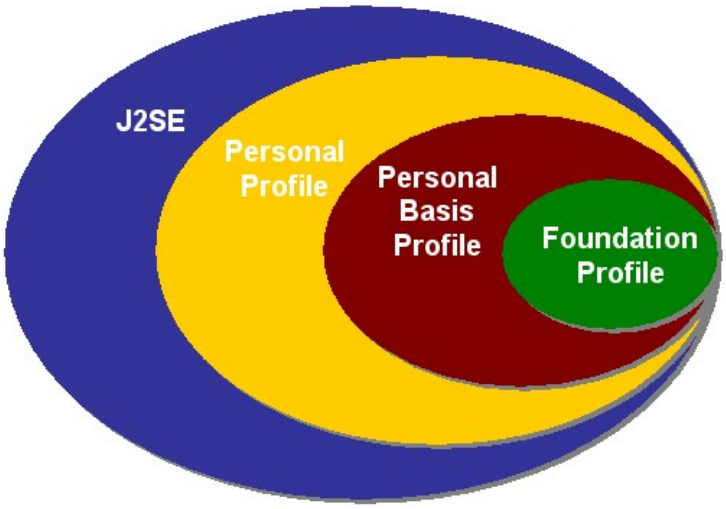

Figura I.7: J2SE, Personal Profile, Personal Basis Profile y Foundation Profile

Diferencias entre PP y JDK 1.3

Los paquetes y clases de PP son típicamente un subconjunto de los paquetes y clases de JDK 1.3. Existen unas pocas APIs en PP que tienen restricciones en su utilización, y son:

- `java.awt.AlphaComposite`. Aproximación de la regla `SRC_OVER`.
- `java.awt.Component`. La implementación debería ignorar las llamadas para hacer establecer el cursor visible.
- `java.awt.Dialog`. La implementación debería limitar el tamaño, prohibir el redimensionamiento, limitar la localización de la pantalla, y no soportar un título visible.
- `java.awt.Frame`. Se tienen las mismas restricciones que con los `Dialogs`.
- `java.awt.Graphics2D`. Solamente se debería poder usar instancias de `AlphaComposite` mediante el método `setComposite()`.
- `java.awt.TextField`. La implementación debería prohibir que se escribiera el carácter de *echo*.

Para cada uno de estos casos se pondrá a `true` una propiedad del sistema si se tiene la correspondiente restricción en la implementación actual. Estas propiedades del sistema son las que a continuación se listan:

- `java.awt.AlphaComposite.SRC_OVER.isRestricted`
- `java.awt.Component.setCursor.isRestricted`
- `java.awt.Dialog.setSize.isRestricted`
- `java.awt.Dialog.setResizable.isRestricted`
- `java.awt.Dialog.setLocation.isRestricted`
- `java.awt.Dialog.setTitle.isRestricted`
- `java.awt.Frame.setSize.isRestricted`
- `java.awt.Frame.setResizable.isRestricted`
- `java.awt.Frame.setLocation.isRestricted`
- `java.awt.Frame.setState.isRestricted`
- `java.awt.Frame.setTitle.isRestricted`
- `java.awt.TextField.setEchoChar.isRestricted`

Diferencias entre PP y PersonalJava

PersonalJava era el Java original de los dispositivos de altas prestaciones y aplicaciones empotradas, pero está siendo reemplazado por el PP, que es una nueva definición, basada en J2ME. Los diseñadores de PP comenzaron a definirlo a partir del JDK 1.3 y eliminaron todas las APIs desaconsejadas así como todas las APIs que consideraron innecesarias para los nuevos dispositivos móviles. PersonalJava, sin embargo, estaba basado en el JDK 1.1.

I.2. Máquinas Virtuales de Java para dispositivos móviles

En este apartado analizaremos diferentes máquinas virtuales de Java [6, 30, 31, 32] con el fin de encontrar aquella que mejor se ajuste a nuestras necesidades específicas. Las características que nosotros requeriremos que posea la JVM son las siguientes:

- **Que soporte J2ME**, preferiblemente la arquitectura CDC/PP.
- **Que soporte JNI**. Éste es un requisito indispensable para poder acceder a las funciones de la librería `mpkcs11.dll` que desarrollemos.
- Que soporte la versión más actual de JDK, preferiblemente.
- Que soporte el mayor número de procesadores, preferiblemente.
- Que soporte el mayor número de sistemas operativos, preferiblemente, **incluyendo siempre Windows CE**. Ésta es una característica importante, ya que facilitará la portabilidad de nuestro diseño a una gran cantidad de dispositivos.
- Que sea de libre distribución, preferiblemente.

I.2.1. Waba

Waba [6] fue una de las primeras máquinas virtuales para dispositivos móviles (no en vano, fue creada a principios del 2000). Inicialmente se desarrolló para PalmOS y Windows CE, pero programadores independientes la han portado a otros dispositivos, como por ejemplos PCs con MS-DOS, Nintendo GameBoy y PDAs Zaurus con Linux.

Waba define un lenguaje, una máquina virtual, un formato de ficheros de clase y un conjunto de clases. No es un derivado de Java y no tiene ningún tipo de conexión con Sun Microsystems, que es el auténtico propietario de Java. La sintaxis del lenguaje de programación Waba es un pequeño subconjunto de la sintaxis del lenguaje Java. Lo mismo ocurre con el fichero de clase Waba y el formato bytecode.

Waba viene con un conjunto de clases puente que permiten a los programas Waba ejecutarse en cualquier entorno Java. Los programas Waba puede ejecutarse como applets Java, e incluso como aplicaciones.

I.2.2. Superwaba

SuperWaba [6, 33] es una máquina virtual y entorno de ejecución para dispositivos móviles, distribuida bajo LGPL y con soporte para PalmOS, Windows CE y Pocket PC, entre otros.

SuperWaba se creó a partir de Waba, pero su desarrollo ha sido totalmente distinto, siendo en este caso muy activo y donde se ha ampliado toda la funcionalidad que ofrecía el original Waba. Aún así, este proyecto sigue siendo un 99 % compatible con la funcionalidad que ofrecía su predecesor.

SuperWaba tiene una comunidad relativamente grande y activa, además de un producto competitivo. Al igual que Waba, presenta el gran problema de ser una plataforma, aún escrita en Java, totalmente propia. Las clases a usar son propias de Waba/SuperWaba, y no tienen nada que ver con J2ME.

	Waba	**Superwaba**
Número de clases	36	69
Tamaño (clases + VM)	72'3 Kb	293 Kb
Comparación de velocidad	390860 ms	162900 ms
Memoria para programas	32 kb	Sin límite
Soporte de exceptiones	No	Sí
Soporte de color	No	Escala de grises y color
Soporte de `double/long`	No	Sí
Interfaz de usuario	Muy simple	Bastante completa

Tabla I.1: Tabla comparativa entre Waba y SuperWaba

I.2.3. Jeode

La Jeode EVM [6, 34, 35] es completamente compatible con la especificación de PersonalJava, y soporta todas las librerías de clases de PersonalJava 1.2, incluidas las clases opcionales.

A diferencia de otros proveedores de la JVM que han implementado capas propietarias de software para gráficos, el paquete de AWT de Jeode ha sido implementado para manejar el sistema nativo de ventanas de cada plataforma. Este acercamiento arquitectónico preserva el ver y sentir *look-and-feel* familiar de ese entorno; habilita el uso de fuentes específicas de cada plataforma, así como el soporte de lenguajes, y asegura integración con teclados virtuales.

Esmertec ha desarrollado además una completa implementación del protocolo de la Interfaz Nativa de Java (JNI) que permite a los desarrolladores soportar funcionalidades específicas de cada plataforma a través de clases Java.

Para soportar aún más las necesidades de la comunidad de las PDAs, Esmertec ha hecho mejoras significantes en el tiempo de arranque del motor de arranque de Jeode EVM por medio de la implementación de librerías de clases pre-cargadas.

I.2.4. PERC

PERC [6, 36] es una máquina virtual que soporta la ejecución de aplicaciones de Java en sistemas embebidos. Es un modelo sencillo, elegante, y eficiente de Java, pero no sacrifica la integridad, ni la funcionalidad, ni los beneficios de las aproximaciones en tiempo real.

Las principales características de PERC son:

- Las opciones de compilación ofrecen una velocidad de ejecución de unas 20 veces por encima de otras implementaciones de intérpretes.
- La recolección de basuras se mantiene dentro de los 100 μs.
- Presenta accesos a memoria muy optimizados, con un algoritmo de mapeo de memoria muy rápido.
- También presenta un algoritmo de cambio de tareas muy rápidos, ya que permite programación con hilos.
- Tiene un emulador de las librerías de clases del JDK 1.4, es decir, permite trabajar con JNI, RMI, JDBC, Collecciones, etc.
- Para interfaces gráficas, soporta J2SE AWT y Eclipse SWT toolkits.
- Además proporciona una API de mapeo de memoria para optimizar los accesos.

Como ya hemos comentado, PERC nació para sistemas embebidos, pero sus características hacen que sea una máquina interesante para dispositivos móviles, que suelen usar sistemas operativos embebidos.

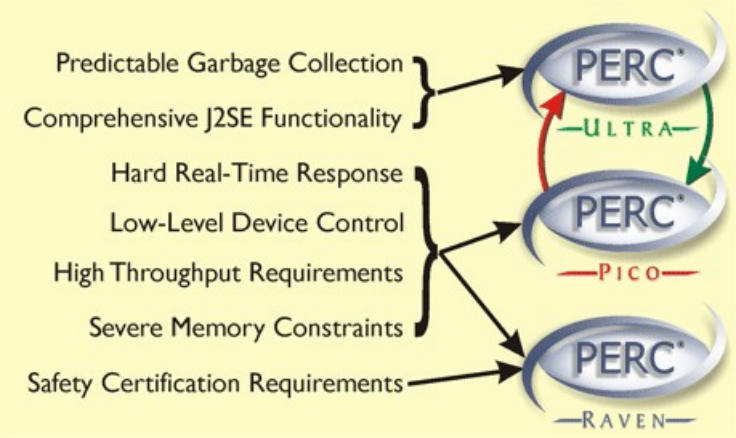

Figura I.8: PERC EVM

I.2.5. Jbed

Jbed [6, 31] es una de las soluciones de Esmertec para Java. Ésta implementa J2ME soportando una gran cantidad de servicios y proporciona capacidades Java multitarea para CLCD/MIDP, extendiendo significativamente las capacidades de servicios que ofrece un dispositivo móvil.

Es decir, Jbed es capaz de ejecutar varias aplicaciones simultáneamente o, dicho de otra manera, permite la ejecución concurrente de aplicaciones.

Además, Jbed ofrece un diseño de software optimizado para pequeños dispositivos móviles, con operaciones de movilidad extremadamente fáciles, consiguiendo así, una plataforma modular unificada que facilita un tiempo rápido de desarrollo.

I.2.6. CrEme

La JVM CrEme [30, 31, 37] está disponible para la mayoría de las plataformas Windows CE, y existen versiones de evaluación gratuitas para algunas plataformas actuales.

CrEme 4 VM implementa la especificación 1.0 del J2ME CDC/Personal Profile, reemplazando el estándar 1.3 de Personal Java incluido en CrEme 3. No en vano, se soporta Swing. Sin embargo, la librería de gráficos por defecto y aconsejada es la AWT, la cual está altamente optimizada y soporta diversos idiomas. También se ofrece, sin embargo, la *Tiny* AWT como alternativa.

Con la instalación del puglin de CrEme, los applets de Java funcionan perfectamente sobre Internet Explorer. CrEme está altamente integrado en el entorno de Windows CE y es fácilmente configurable para requisitos de una aplicación particular.

Por ejemplo, las aplicaciones pueden tomar el control total del dispositivo, se pueden ejecutar a pantalla completa, pueden cambiar la visibilidad del teclado del dispositivo, utilizar puertos de comunicaciones, reaccionar cuando se pulsan los botones de acceso directo, controlar el estado de la batería, etc.

Algunas aplicaciones pueden necesitar acceder a características del dispositivo que aún no se encuentren soportadas al nivel de Java. Tal funcionalidad siempre puede ser implementada en C o C++, ya que CrEme soporta por completo el estándar JNI.

I.2.7. ChaiVM

ChaiVM [6] es un proyecto de HP que pretende realizar una máquina virtual que implemente J2ME para dispositivos móviles. Sabemos que en un principio era un proyecto con grandes perspectivas de futuro, pero no sabemos si porque lo han abandonado, o porque quieren ocultarlo, no ofrecen información sobre el mismo.

Desde entornos fuera de HP, se habla de esta plataforma como una de las mejores implementaciones de J2ME, reflejando fielmente la idea de CLCD y de MIDP.

I.2.8. J9

J9 [30, 31, 38] es la JVM de IBM. Implementa las dos especificaciones de Java para dispositivos móviles: J2ME CLDC (para pequeños dispositivos como teléfonos móviles, pero también válido para dispositivos Palm) y J2ME CDC (para PocketPc, Linux, etc.). Se encuentra integrada dentro del *WebSphere Everyplace Micro Environment* y del *WebSphere Everyplace Custom Environment*.

La máquina virtual J9 implementa una capa configurable, compacta, rápida y de arquitectura predecible que provee una interfaz común a los programas independientemente del hardware del dispositivo o del sistema operativo subyacentes.

J9 se ejecuta sobre un sistema operativo (tal como PalmOS, PocketPC, QNX, embedded Linux, etc.) y gestiona las interfaces específicas de dicho sistema operativo con el dispositivo hardware. Por este motivo, la máquina virtual está cuidadosamente diseñada pensando en la portabilidad. Está implementada a través de una capa independiente del sistema operativo.

En un entorno con recursos limitados la flexibilidad de configuraciones es muy importante. Por tanto, J9 tiene un amplio rango de características configurables, como pueden ser la carga de clases dinámicas, el tamaño de la pila o la utilización de memoria, entre otras.

Una licencia individual para desarrolladores cuesta 6 $, mientras que por 599 $ se puede adquirir una licencia sin limitaciones.

I.2.9. Mysaifu

Mysaifu [31, 39] es una máquina virtual de java desarrollada bajo licencia GNU y, por lo tanto, es gratuita. Pretende implementar la especificación J2SE y parece tener una AWT funcionando actualmente.

Es un proyecto en el que se está trabajando bastante (al contrario que sucede con CrEme y J9, por ejemplo). El soporte para JNI se encuentra todavía en las primeras fases de desarrollo.

Aún no existe demasiada información acerca de esta máquina virtual, pero al tratarse de un proyecto de código abierto, es de esperar que en un tiempo se convierta en un producto competitivo.

I.2.10. KVM

Sobre la KVM [4, 5] ya se trató en el apartado I.1.4 de la página 24. Para más información, léase dicho apartado.

I.2.11. Resumen

JVM	J2ME	JNI	JDK	S.O.	Procesadores	¿Gratuita/ precio?
Waba	No	**No**	1.2	**Windows CE** PalmOS MS-DOS Linux Solaris Gameboy	MIPS SH3	Gratis
Superwaba	No	**No**		PalmOS **Windows** 98/ME/ 2000/XP/**CE** Linux Symbian	PocketPC ARM Pal	Gratis
Jeode	Sí	**Sí**	1.3	**Windows CE** Linux Windows NT4 VxWorks ITRON Nucleus BSDi UNIX pSOS	ARM MIPS x86 SH3 SH4 PowerPC	50 $
PERC	Sí	**Sí**	1.4	**Windows CE** Linux LynxOS MontaVista Linux OSE OSE Softkernel PikeOS QNX VxWorks VxSim	x86 PowerPC XScale ARM MIPS Coldfire	
Jbed	Sí	**??**		**Windows CE** Linux Qtopia Unix VxWorks	ARM MIPS SH3 SH4 PowerPC x86 Intel XScale TI OMAP Qualcomm	500 unidades → 7500 $

Tabla I.2: Resumen de las distintas JVMs analizadas (I)

JVM	J2ME	JNI	JDK	S.O.	Procesadores	¿Gratuita/ precio?
CrEme	Sí	**Sí**	1.3	**Windows CE**	ARM XScale x86 MIPS SH3 PowerPC	Prueba gratis 40 unidades → 1000 $
ChaiVM	Sí	**??**	1.1.8	**Windows CE**	ARM	
J9 IBM	Sí	**Sí**	1.4	**Windows CE** Palm Linux QNX OSE iTron PocketPC	ARM StrongARM x86 MIPS PowerPc SuperH	6 $
Mysaifu	Sí	**??**		Pocket PC 2003 **Pocket PC 2003 SE**	StrongARM XScale	Gratis
KVM	Sí	**No**				

Tabla I.3: Resumen de las distintas JVMs analizadas (II)

I.3. Evaluación del IAIK PKCS#11 Provider Micro Edition

I.3.1. Descripción

IAIK PKCS#11 Provider Micro Edition [9, 29] es una librería que permite integrar fácilmente aplicaciones escritas en Java y *smart cards*. Soporta cualquier tipo de *smart card* para la que esté disponible un módulo PKCS#11.

Está orientada a aplicaciones para entornos con recursos limitados tales como una PDA o un *smart phone*, y consta principalmente de dos componentes:

- Las clases de Java contenidas en el fichero JAR `iaik_p11_me.jar` (de unos 50 kBytes).
- La parte nativa, que en el caso de Windows se encuentra en el fichero `pkcs11wrapper.dll` (de unos 70 kBytes).

Siempre se necesitan ambos componentes para que pueda funcionar, ya que las clases Java necesitan la parte nativa para acceder al módulo PKCS#11 de la *smart card*.

I.3.2. Requisitos

Los requisitos mínimos que se deben cumplir para poder ejecutar esta librería son los siguientes:

- Java 1.1 o compatible con soporte para JNI 1.1
- Windows 95/98/ME/2000/XP/CE, Linux, Solaris. Si se desea añadir soporte para nuevas plataformas, basta con recompilar la parte nativa del IAIK PKCS#11 Wrapper para dichas plataformas.
- Un módulo PKCS#11 compatible con PKCS#11 2.01, 2.10 ó 2.11 compilado para la plataforma en la que se desee trabajar (en nuestro caso será Windows CE).

No se precisa ninguna otra librería para poder funcionar correctamente.

I.3.3. Características

A continuación se listan las principales características que se incluyen en esta librería:

- **Lista, lee, añade y suprime todas las claves y certificados** en un token
- Operaciones criptográficas directamente sobre el token:
  - **Resumen digital** (*hash*)
    - Algoritmos soportados: MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPEMD 128 y RIPEMD 160
  - **Creación y verificación de firma digital**
    - Algoritmos soportados: MD2 con RSA, MD5 con RSA, SHA-1 con RSA, SHA-256 con RSA, SHA-384 con RSA, SHA-512 con RSA, RIPEMD 128 con RSA, RIPEMD 160 con RSA y raw RSA (con cálculo externo del resumen digital)
    - Algoritmos PSS soportados: SHA-1 con RSA y MFG1, SHA-256 con RSA y MFG1, SHA-384 con RSA y MFG1, y SHA-512 con RSA y MFG1
  - **Cálculo de MAC**
    - Algoritmos soportados: MD2 HMAC, MD5 HMAC, SHA-1 HMAC, SHA-256 HMAC, SHA-384 HMAC, SHA-512 HMAC, RIPEMD 128 HMAC y RIPEMD 160 HMAC

- **Cifrado y descifrado**
  - Algoritmos asimétricos soportados: RSA con PKCS#1 versión 1.5 padding, RSA con PKCS#1 versión 2.0 OAEP padding (SHA-1 con MFG-1)
  - Algoritmos simétricos soportados
    - Cifrado de bloque, cada uno con ECB sin padding y CBC con PKCS padding: AES, DES, Triple DES, IDEA y RC2 (el tamaño efectivo de la clave es el tamaño real de la clave)
    - Cifrado de bloque: RC4
- **Envoltura y desenvoltura de claves simétricas**
- **Generación de pares de claves**
  - Algoritmos soportados: RSA
- **Generación de claves**.
  - Algoritmos soportados: AES, DES Triple DES (168 bits), IDEA, RC2, RC4, clave genérica

- Generación de información aleatoria en el token
- Importación tanto de claves como de certificados en el token
- Inicialización del token
- Inicialización y cambio del PIN de usuario
- Login con un PIN dado explícitamente o a través de otros medios

Naturalmente es preciso que el módulo PKCS#11 subyacente soporte cada una de estas características para que puedan ser accedidas a través de esta librería.

I.3.4. Instalación

La instalación del PKCS#11 Provider Micro Edition consta básicamente de dos pasos:

1. Inclusión del fichero `iaik_p11_me.jar`.
2. Inclusión de la parte nativa `pkcs11wrapper.dll` (para el caso de Windows).

Existen a su vez dos formas distintas de instalación, según se pretendan desarrollar aplicaciones o *applets*. Nosotros explicaremos sólo la primera de ellas, enfocada al desarrollo de aplicaciones.

Así, lo primero que se debe hacer es incluir el fichero `iaik_p11_me.jar` en el `CLASSPATH`, tal y como se haría con cualquier otra librería. Sin embargo, hay que llevar especial cuidado en determinados entornos, ya que puede ocurrir que se introduzcan restricciones en aquellas librerías que incluyen código nativo a través de JNI.

Para la inclusión de la parte nativa de la librería (fichero `pkcs11wrapper.dll` para el caso de Windows) existen diferentes alternativas dependiendo de la máquina virtual que se tenga instalada. La mayoría de máquinas virtuales soportan que dicha parte nativa se incluya en alguno de los directorios por defecto en los cuales buscan librerías de código nativo.

Normalmente estos directorios incluyen los directorios de librerías del sistema operativo como `C:\WINNT\System32`, en el caso de Windows. Estos directorios se listan en la variable de entorno `PATH`, en un entorno Windows.

Otra alternativa pasaría por crear un nuevo directorio que contenga la parte nativa y añadir dicho directorio a la lista de directorios en los que la máquina virtual busca librerías de código nativo.

I.3.5. Modo de empleo

Los tipos básicos de objetos en el paquete `iaik.pkcs.pkcs11.me` son módulos y tokens. Un token normalmente representa una *smart card* o un HSM. Un token nunca existe como un objeto aislado; sólo existe como una entidad lógica dentro de un módulo.

Así, una aplicación sólo puede tener acceso al token a través de su correspondiente módulo. El módulo es el software en el token que gestiona la comunicación con el token físico.

El módulo por sí mismo no procesa ninguna operación criptográfica ni gestiona los objetos de las claves. Esto sólo lo hacen los tokens. Cualquier aplicación debe instanciar un módulo y obtener el token antes de realizar cualquier operación criptográfica. Y por último, debe cerrar el módulo antes de terminar.

En la mayoría de los casos se seguirá siempre el mismo patrón de empleo:

1. Se instancia el módulo
2. Se obtiene el token
3. Se hace login del usuario
4. Se obtiene el *key store* del token
5. Se selecciona una clave
6. Se procesa la operación criptográfica
7. Se hace logout del usuario
8. Se cierra el módulo

I.3.6. Interoperabilidad

Esta librería es independiente de cualquier otra librería criptográfica o de seguridad y además no precisa incluir ninguna librería más para poder usarla. En la práctica puede ser necesario traducir certificados o simplemente integrar el uso de un token en la implementación de un protocolo tal como TLS o S/MIME. Dichas implementaciones están basadas habitualmente en el marco de trabajo JCA o JCE, o bien en una solución propietaria como IAIK-JCE Micro Edition. Pues bien, la librería IAIK Provider Micro Edition es compatible con ambas soluciones.

I.3.7. Ventajas e inconvenientes

Ventajas

IAIK es una empresa seria y confiable con amplia experiencia en el desarrollo de productos orientados a la criptografía, y que ofrece un buen soporte técnico.

Esta librería soporta una gran variedad de sistemas operativos, ofrece una gran cantidad de operaciones criptográficas, es fácil de aprender y manejar, tiene pocos requerimientos de memoria, no tiene dependencia con otras librerías para poder ser usada, es fácil de instalar y altamente interoperable, entre otras ventajas.

Inconvenientes

Es una solución propietaria y, por tanto, de pago. Salvo que desee utilizar para fines educativos y/o de investigación, el precio por adquirir una licencia para poder comercializar un producto final que haga uso de esta librería es de entre 100 y 2000 euros.

Además, es la única API J2ME que hemos encontrado en el mercado que sirva como *wrapper* de un módulo PKCS#11, por lo que no hay muchas más alternativas (salvo la de construir nosotros mismos nuestra API).

I.4. Breve introducción al estándar PKCS#11

I.4.1. Introducción

El estándar PKCS#11 [10, 22] especifica una API, denominada "Cryptoki" para dispositivos que contienen información criptográfica y que realizan funciones criptográficas. Cryptoki sigue una aproximación sencilla basada en objetos, abordando los objetivos de independencia tecnológica (cualquier tipo de dispositivo) y compartición de recursos (múltiples aplicaciones accediendo a múltiples dispositivos), presentando a la aplicación una visión lógica común del dispositivo denominado "token criptográfico".

Desde el principio se pretendió que Cryptoki fuera una interfaz entre aplicaciones y todo tipo de dispositivos criptográficos portables, tales como aquellos basados en *smart cards*, tarjetas PCMCIA o *smart diskettes*.

Cryptoki aísla a la aplicación de los detalles del dispositivo criptográfico. La aplicación no tiene que cambiar de interfaz para cada tipo distinto de dispositivo o ejecutarse en un entorno diferente. Así, la aplicación es portable.

Esta versión del estándar soporta varios mecanismos (algoritmos) criptográficos. Es más, posteriormente se podrán añadir nuevos mecanismos sin necesidad de cambiar la interfaz general. Los fabricantes de tokens también pueden definir sus propios mecanismos (aunque, en favor de la interoperabilidad, es preferible el registro de nuevos mecanismos a través del proceso PKCS).

La versión 2.11 de Cryptoki está orientada hacia dispositivos criptográficos asociados con un único usuario, por lo que algunas de las características incluidas en la interfaz de propósito general han sido omitidas. Por ejemplo, en esta versión no hay intención de distinguir múltiples usuarios. El foco de atención se centra en claves pertenecientes a un único usuario y quizás unos pocos certificados relacionados con ellas. El énfasis se pone en la criptografía.

I.4.2. Modelo general

El modelo comienza con una o más aplicaciones que necesitan realizar determinadas operaciones criptográficas, y termina con uno o más dispositivos criptográficos, en los cuales se ejecutan algunas o todas las operaciones. Un usuario puede o no estar asociado con una aplicación.

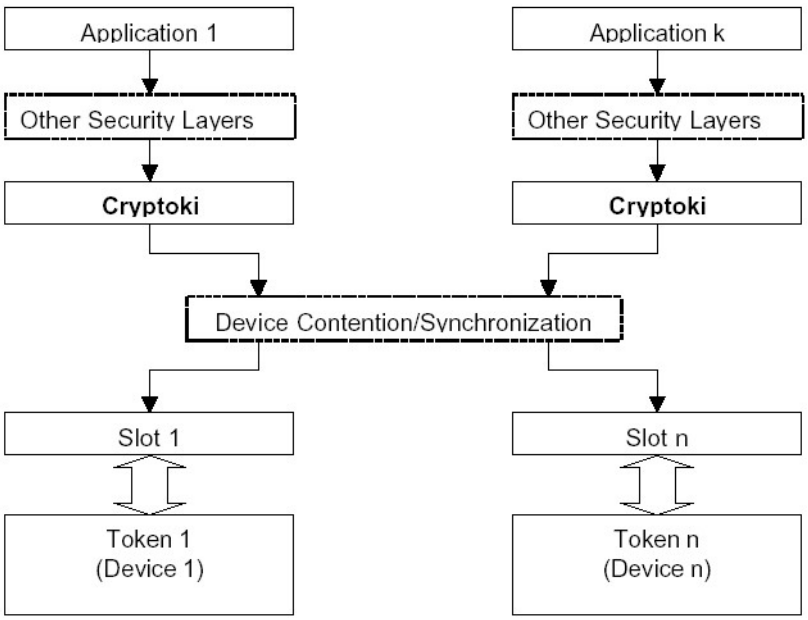

Figura I.9: Modelo general de Cryptoki

Cryptoki proporciona una interfaz para uno o más dispositivos criptográficos que se encuentren activos en el sistema por medio de un determinado número de ranuras o *slots*. Cada ranura, que se corresponde con un lector físico u otra interfaz criptográfica, puede contener un token. Un token se encuentra típicamente presente en la ranura cuando un dispositivo criptográfico está presente en el lector.

Un dispositivo criptográfico puede realizar algunas operaciones criptográficas, siguiendo unos determinados comandos. Estos comandos se pasan típicamente a través de los controladores estándar del dispositivo. Cryptoki hace que cada dispositivo se vea lógicamente como cualquier otro, independientemente de la tecnología de implementación. Así no necesita interactuar directamente con los controladores del dispositivo (o incluso saber cuáles de ellos están involucrados); Cryptoki oculta estos detalles.

Los tipos de dispositivos soportados, así como las capacidades de éstos, dependerán de cada librería Cryptoki en particular. El estándar solamente especifica la interfaz con la librería, no sus características. En particular, no todas las librerías soportarán todos los mecanismos (algoritmos) definidos en esta interfaz, puesto que se supone que no todos los tokens soportan todos los algoritmos.

I.4.3. Visión lógica de un token

La visión lógica de un token por parte de Cryptoki es la de un dispositivo que almacena objetos y puede realizar funciones criptográficas. Cryptoki define tres tipos de objetos: datos, certificados y claves. Un objeto de tipo dato es definido por una aplicación, un objeto de tipo certificado almacena un certificado, mientras que un objeto de tipo clave almacena una clave criptográfica. Esta clave puede ser pública, privada o secreta.

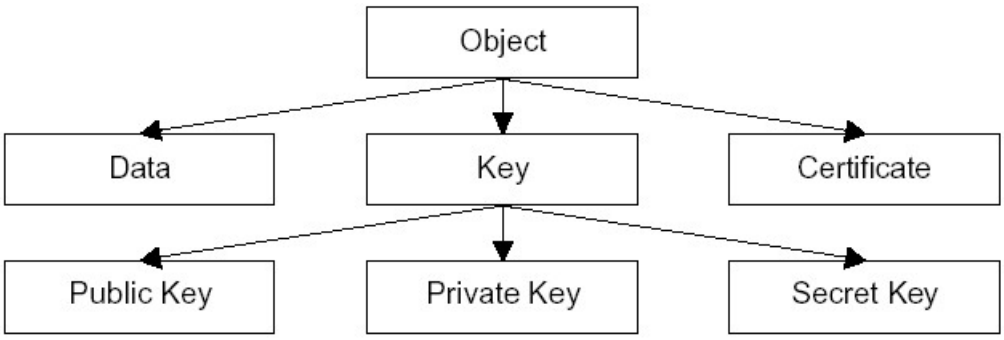

Figura I.10: Jerarquía de objetos de Cryptoki

Los objetos también se pueden clasificar de acuerdo a su tiempo de vida y visibilidad. Los "objetos de token" son visibles para todas las aplicaciones conectadas al token que tengan suficientes permisos, y permanecen en el token incluso después de que se cierren las sesiones (conexiones entre la aplicación y el token) y el token sea retirado de su ranura. Los "objetos de sesión" son más temporales: cada vez que se cierra una sesión por cualquier motivo, todos los objetos de sesión creados por dicha sesión son automáticamente destruidos. Es más, los objetos de sesión sólo son visibles a la aplicación que los creó.

Otra clasificación más puede atender a los requisitos de acceso. Las aplicaciones no necesitan registrarse (hacer *log in*) en el token para acceder a los "objetos públicos", sin embargo para acceder a los "objetos privados" sí es necesario que un usuario se autentique frente al token mediante un PIN o algún mecanismo similar.

Un token puede crear y destruir objetos, manipularlos y buscarlos. También puede realizar operaciones criptográficas con dichos objetos y puede contener un generador interno de números aleatorios.

I.4.4. Usuarios

Esta versión de Cryptoki reconoce dos tipos de usuarios: el operario de seguridad (SO) y el usuario normal. Sólo a éste último se le permite el acceso a los objetos privados del token y dicho acceso es permitido solamente después de que tal usuario se haya autenticado. Algunos tokens pueden incluso requerir que el usuario se autentique antes de realizar cualquier operación criptográfica, independientemente de que manipule o no objetos privados.

El papel del SO es inicializar el token y establecer el PIN del usuario normal, y puede que también tenga que manipular algunos objetos públicos. El usuario normal no puede registrarse (hacer *log in*) hasta que el SO no haya establecido el PIN para dicho usuario en el token.

I.4.5. Aplicaciones y su uso de PKCS#11

Para Cryptoki, una aplicación consiste en un espacio de memoria concreto y todos los hilos que se ejecutan en él. Una aplicación se convierte en una "aplicación Cryptoki" mediante la llamada a la función `C_Initialize` desde alguno de sus hilos. Una vez se haya hecho esta llamada, entonces se pueden realizar llamadas a otras funciones de Cryptoki.

Cuando la aplicación ha terminado de utilizar Cryptoki, entonces llamada la función `C_Finalize` y en ese momento deja de ser una aplicación Cryptoki. En general, en la mayoría de plataformas, todo esto sucede en aplicaciones consistentes en un único proceso.

I.4.6. Sesiones

Cryptoki precisa que una aplicación abra una o más sesiones con un token para conseguir el acceso a los objetos y funciones de dicho token. Una sesión proporciona una conexión lógica entre la aplicación y el token, y puede ser de lectura/escritura (R/W) o de sólo-lectura (R/O). Lectura/escritura y sólo-lectura se refieren al acceso a los objetos de token, no a los objetos de sesión.

Después de abrir una sesión, una aplicación tiene acceso a los objetos públicos del token. Todos los hilos de una aplicación dada tienen exactamente las mismas sesiones y los mismos objetos de sesión. Para obtener el acceso a los objetos privados del token es necesario que el usuario normal se autentique frente a éste.

Cuando se cierra una sesión, todos los objetos de sesión creados por dicha sesión son destruidos. Esto incluye también a los objetos de sesión que se encuentren en uso por parte de otras sesiones.

Cryptoki soporta múltiples sesiones sobre múltiples tokens. Una aplicación puede tener una o más sesiones sobre uno o más tokens. Y un token puede tener múltiples sesiones con una o más aplicaciones. También puede suceder que un token en particular permita a una aplicación solamente un número limitado de sesiones sobre él (o un número limitado de sesiones de lectura/escritura).

Una sesión abierta se puede encontrar en varios estados, los cuales determinan la accesibilidad a los objetos, así como a las funciones que se pueden realizar sobre éstos.

Estados de las sesiones de sólo-lectura

Una sesión de sólo-lectura puede encontrarse en dos estados diferentes. Cuando la sesión es inicialmente abierta, ésta se encuentra o bien en el estado "*R/O Public Session*" (si la aplicación no ha abierto previamente sesiones que estén registradas), o bien en el estado "*R/O User Functions*" (si la aplicación ya tiene una sesión abierta registrada).

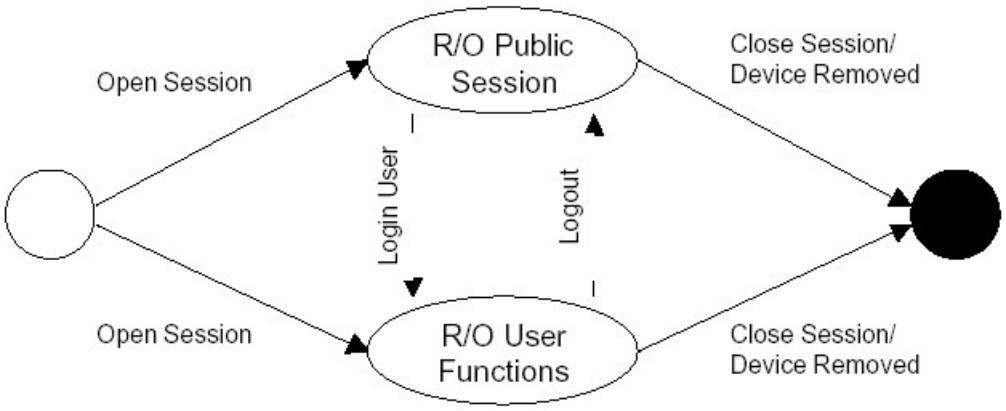

Figura I.11: Estados de las sesiones de sólo-lectura

Estado	Descripción
R/O Public Session	La aplicación ha abierto una sesión de sólo-lectura. La aplicación tiene acceso de sólo-lectura a los objetos públicos de token y acceso de lectura/escritura a los objetos públicos de sesión
R/O User Functions	El usuario normal se ha autenticado frente al token. La aplicación tiene acceso de sólo-lectura a todos los objetos de token y acceso de lectura/escritura a todos los objetos de sesión

Tabla I.4: Estados de las sesiones de sólo-lectura

Estados de las sesiones de lectura/escritura

Una sesión de lectura/escritura puede estar en tres estados diferentes. Cuando la sesión es inicialmente abierta, ésta se encuentra en el estado "*R/W Public Session*" (si la aplicación no ha abierto previamente sesiones que estén registradas), o en el estado "*R/W User Functions*" (si la aplicación ya tiene una sesión abierta en la que el usuario normal está registrado), o bien en el estado "*R/W SO Functions*" (si la aplicación ya tiene una sesión abierta en la que el SO está registrado).

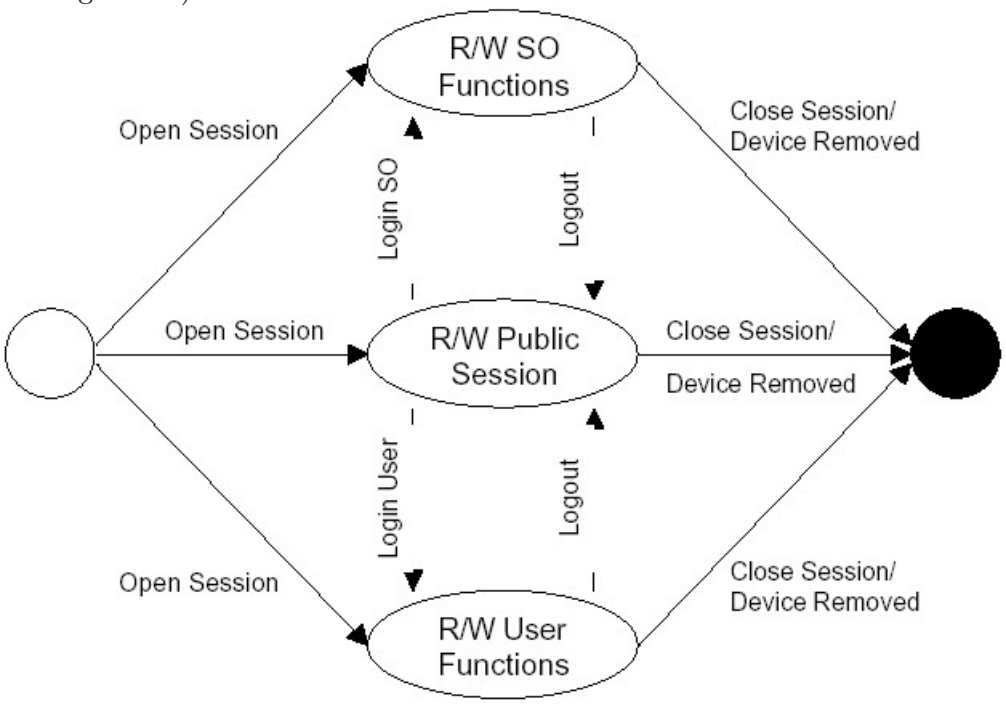

Figura I.12: Estados de las sesiones de lectura/escritura

Estado	Descripción
R/W Public Session	La aplicación ha abierto una sesión de lectura/escritura y tiene acceso de lectura/escritura a todos los objetos públicos
R/W SO Functions	El SO se ha autenticado frente al token. La aplicación tiene acceso de lectura/escritura solamente a los objetos públicos del token
R/W User Functions	El usuario normal se ha autenticado frente al token. La aplicación tiene acceso de lectura/escritura a todos los objetos

Tabla I.5: Estados de las sesiones de lectura/escritura

Accesibilidad a los objetos del token para cada sesión

	Tipo de sesión				
Tipo de objeto	*R/O Public*	*R/W Public*	*R/O User*	*R/W User*	*R/W SO*
Objetos públicos de sesión	R/W	R/W	R/W	R/W	R/W
Objetos privados de sesión			R/W	R/W	
Objetos públicos de token	R/O	R/W	R/O	R/W	R/W
Objetos privados de token			R/O	R/W	

Tabla I.6: Accesibilidad a los objetos del token para cada sesión

Eventos de las sesiones

Los eventos de las sesiones provocan el cambio de estado en éstas.

Evento	Ocurre cuando...
Log in del SO	El SO es autenticado frente al token
Log in del usuario	El usuario normal es autenticado frente al token
Log out	La aplicación desregistra al usuario actual (normal o SO)
Cierre de sesión	La aplicación cierra la sesión, o bien cierra todas las sesiones
Dispositivo retirado	El dispositivo subyacente al token es retirado de su ranura

Tabla I.7: Eventos de las sesiones

Cuando se retira el dispositivo de su ranura, todas las sesiones de todas las aplicaciones son desregistradas. Más aún, todas las sesiones de todas las aplicaciones son cerradas (puesto que una aplicación no puede tener abierta una sesión con un token que no se encuentre presente en su ranura).

Siendo realistas, Cryptoki no puede estar constantemente controlando si el token está presente o no en su ranura, y por lo tanto la ausencia del token puede que no sea descubierta hasta que se realice la próxima ejecución de una función de Cryptoki. Si el token es reinsertado en su ranura antes de que esto ocurra, puede que Cryptoki nunca sepa que dicho token estuvo ausente.

En la versión 2.11 de Cryptoki todas las sesiones que una aplicación tiene con un token determinado deben tener el mismo estado de login o logout. Cuando una sesión de una aplicación se registra en el token, todas las sesiones de esa aplicación con ese token pasan a estar registradas. Lo mismo ocurre al desregistrarse.

I.4.7. Resumen de las principales funciones de Cryptoki

Categoría	Función	Descripción
Funciones de propósito general	`C_Initialize`	Inicializa el Cryptoki
	`C_Finalize`	Libera los recursos asociados al Cryptoki
	`C_GetInfo`	Obtiene información general acerca del Cryptoki
	`C_GetFunctionList`	Obtiene los puntos de entrada de las funciones de la librería Cryptoki
Funciones de manejo del slot y del token	`C_GetSlotList`	Obtiene una lista de las ranuras en el sistema
	`C_GetSlotInfo`	Obtiene información acerca de una ranura en particular
	`C_GetTokenInfo`	Obtiene información acerca de un token en particular
	`C_WaitForSlotEvent`	Espera a que ocurra un evento de sesión
	`C_GetMechanismList`	Obtiene una lista de los mecanismos soportados por el token
	`C_GetMechanismInfo`	Obtiene información acerca de un mecanismo en particular
	`C_InitToken`	Inicializa un token
	`C_InitPIN`	Inicializa el PIN del usuario normal
	`C_SetPIN`	Modifica el PIN del usuario actual
Funciones de gestión de sesiones	`C_OpenSession`	Abre una conexión entre una aplicación y un token en particular
	`C_CloseSession`	Cierra una sesión
	`C_CloseAllSessions`	Cierra todas las sesiones con el token
	`C_GetSessionInfo`	Obtiene información acerca de la sesión
	`C_GetOperationState`	Obtiene el estado de las operaciones criptográficas de una sesión
	`C_SetOperationState`	Establece el estado de las operaciones criptográficas de una sesión
	`C_Login`	Se registra en un token
	`C_Logout`	Se desregistra de un token
Funciones de gestión de objetos	`C_CreateObject`	Crea un objeto
	`C_CopyObject`	Crea una copia de un objeto
	`C_DestroyObject`	Destruye un objeto
	`C_GetObjectSize`	Obtiene el tamaño de un objeto en bytes
	`C_GetAttributeValue`	Obtiene el valor de un atributo de un objeto
	`C_SetAttributeValue`	Modifica el valor de un atributo de un objeto
	`C_FindObjectsInit`	Inicializa una operación de búsqueda de un objeto
	`C_FindObjects`	Continua una operación de búsqueda de un objeto
	`C_FindObjectsFinal`	Finaliza una operación de búsqueda de un objeto
Funciones de cifrado	`C_EncryptInit`	Inicializa una operación de cifrado
	`C_Encrypt`	Cifra datos
	`C_EncryptUpdate`	Continua una operación de cifrado multi-parte
	`C_EncryptFinal`	Finaliza una operación de cifrado multi-parte
Funciones de descifrado	`C_DecryptInit`	Inicializa una operación de decifrado
	`C_Decrypt`	Descifra datos cifrados
	`C_DecryptUpdate`	Continua una operación de descifrado multi-parte
	`C_DecryptFinal`	Finaliza una operación de descifrado multi-parte

Tabla I.8: Resumen de las principales funciones de Cryptoki (I)

Categoría	**Función**	**Descripción**
Funciones de resumen digital	`C_DigestInit`	Inicializa una operación de resumen digital
	`C_Digest`	Realiza el resumen digital de datos
	`C_DigestUpdate`	Continua una operación de resumen digital multi-parte
	`C_DigestKey`	Realiza el resumen digital de una clave
	`C_DigestFinal`	Finaliza una operación de resumen digital multi-parte
Funciones de firma y MAC	`C_SignInit`	Inicializa una operación de firma
	`C_Sign`	Firma unos datos
	`C_SignUpdate`	Continua una operación de firma multi-parte
	`C_SignFinal`	Finaliza una operación de firma multi-parte
	`C_SignRecoverInit`	Inicializa una operación de firma, donde los datos pueden ser recuperados de la propia firma
	`C_SignRecover`	Firma datos, donde éstos pueden ser recuperados de la propia firma
Funciones de verificación de firma y MAC	`C_VerifyInit`	Inicializa una operación de verificación
	`C_Verify`	Verifica una firma de unos datos
	`C_VerifyUpdate`	Continua una operación de verificación multi-parte
	`C_VerifyFinal`	Finaliza una operación de verificación multi-parte
	`C_VerifyRecoverInit`	Inicializa una operación de verificación, donde los datos pueden ser recuperados de la propia firma
	`C_VerifyRecover`	Verifica datos, donde éstos pueden ser recuperados de la propia firma
Funciones criptográficas de doble propósito	`C_DigestEncryptUpdate`	Continua operaciones simultaneas multi-parte de resumen digital y cifrado
	`C_DecryptDigestUpdate`	Continua operaciones simultaneas multi-parte de resumen digital y descifrado
	`C_SignEncryptUpdate`	Continua operaciones simultaneas multi-parte de firma y cifrado
	`C_DecryptVerifyUpdate`	Continua operaciones simultaneas multi-parte de verificación y descifrado
Funciones de gestión de claves	`C_GenerateKey`	Genera una clave secreta
	`C_GenerateKeyPair`	Genera un par de claves pública y privada
	`C_WrapKey`	Envuelve (cifra) una clave
	`C_UnwrapKey`	Desenvuelve (descifra) una clave
	`C_DeriveKey`	Deriva una clave a partir de una clave base
Funciones de generación de nº aleatorios	`C_SeedRandom`	Introduce material adicional para la semilla en el generador de números aleatorios
	`C_GenerateRandom`	Genera datos aleatorios
Funciones de gestión de funciones paralelas	`C_GetFunctionStatus`	Función que siempre devuelve `CKR_FUNCTION_NOT_PARALLEL`
	`C_CancelFunction`	Función que siempre devuelve `CKR_FUNCTION_NOT_PARALLEL`

Tabla I.9: Resumen de las principales funciones de Cryptoki (II)

Diseño

I.5. Introducción

Tras haber analizado todas las tecnologías y estándares que participan en este escenario, se ha llegado a la conclusión de que en el diseño del mismo se planteará desarrollar una librería PKCS#11 específica del sistema operativo subyacente (Windows Mobile 2003 en nuestro caso).

El acceso a dicha librería desde J2ME se realizará a través de JNI, por lo que será necesario que la máquina virtual que se instale en el dispositivo incorpore esta funcionalidad. Y por último, para facilitar todos los desarrollos posteriores basados en la librería criptográfica J2ME desarrollada, ésta deberá hacer uso de la API de IAIK, que actuará como interfaz entre el módulo PKCS#11 y el mundo J2ME.

Así, se decidió realizar el diseño expuesto en la figura I.13.

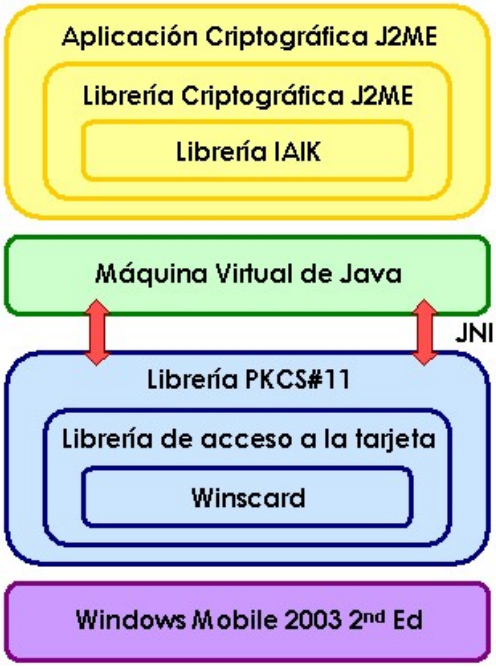

Figura I.13: Diseño del primer escenario

En dicha figura se observa la arquitectura por capas que supondrá la solución aquí planteada para el primer escenario de integración de tarjetas criptográficas en dispositivos móviles.

I.5.1. Hardware

En cuanto al hardware empleado en el diseño de este escenario, se utilizarán, la PDA Acer N50, el lector USB de tarjetas GemPC Twin, la tarjeta criptográfica y el lector SDIO de tarjetas Tactel S300 que se describen y se detallan en el apéndice D.

I.6. Librería PKCS#11

El punto de partida de este diseño no es otro sino la creación de un módulo para dispositivos móviles que implemente el estándar de acceso a tokens criptográficos PKCS#11 que se vio en la sección I.4 (página 38).

Uno de los principales motivos por los que se tomó éste como primer escenario fue la ventaja con la que se partía debido a que ya existía un módulo PKCS#11 desarrollado por la Universidad de Murcia para PC. Por lo tanto, no se tendría que crear un módulo para dispositivos móviles desde cero, sino más bien se trataría de una adaptación del módulo ya existente para PC.

Dicha adaptación debería ser tal que todas aquellas modificaciones que se hicieran sobre el código original (como por ejemplo cambios en los tipos de datos) no alteraran el funcionamiento global del módulo, sino que fueran únicamente dirigidos a lograr la compatibilidad del mismo con la nueva plataforma de ejecución.

I.6.1. Librería de acceso a la tarjeta

En ese nuevo módulo PKCS#11 una parte debe estar orientada a la realización de todas las operaciones criptográficas pertinentes usando la información recuperada de la tarjeta, y otra parte deberá encargarse de la recuperación de dicha información de la tarjeta a través del lector SDIO.

Esta segunda parte será otro módulo aparte que, una vez desarrollado correctamente, se integrará con el módulo PKCS#11 para dispositivos móviles.

Este módulo de acceso a la tarjeta, a su vez, estará basado en la librería Winscard de acceso a tarjetas inteligentes para Windows y será más bien una interfaz de dicha librería.

I.7. Máquina Virtual de Java: CrEme 4.11

El segundo punto a tratar en este diseño consiste en la elección de una máquina virtual de Java apropiada. Antes de eso se debe decir que se eligieron la configuración CDC y el perfil Personal Profile como arquitectura J2ME para desarrollar cualquier aplicación en dicha plataforma.

Esta elección se debió a que los lectores SDIO son prácticamente exclusivos de las PDAs (es raro que un móvil lleve ranura SDIO) y las PDAs tienen, en general, suficientes capacidades como para soportar aceptablemente dicha arquitectura J2ME.

De esta manera, se decidió usar la máquina virtual CrEme en su versión 4.11. Aunque dicha máquina fue tratada en la sección I.2.6 (página 31), vamos a recordar las principales características de la misma por las cuales se decidió utilizarla:

- Es totalmente compatible con J2ME
- Implementa la configuración CDC y el perfil PP
- Soporta el sistema operativo Windows Mobile 2003
- Soporta varios tipos de procesador, incluido ARM
- Implementa la funcionalidad JNI
- Es compatible con JDK 1.3
- Implementa la AWT y es compatible con swing 1.1
- Es gratuita para desarrollos de investigación

I.8. Librería criptográfica J2ME

Una vez se tenga desarrollado el módulo PKCS#11 para dispositivos móviles y se haya elegido la máquina virtual de Java que se haya considerado más apropiada, el siguiente paso consiste en el desarrollo de una librería J2ME que provea de capacidades criptográficas a otras aplicaciones que deseen usarla.

Esta librería se construirá a partir de la librería de IAIK (descrita en la sección I.3 de la página 35), como una interfaz de la misma, simplificándola. Los motivos por los cuales se decidió usar la librería de IAIK fueron:

- Es la única API de Java que hemos encontrado que se adecúa perfectamente a nuestras necesidades, esto es, ofrece una comunicación con el módulo PKCS#11 transparente al programador J2ME (éste no tiene que saber que por debajo hay un módulo PKCS#11).
- Ofrece una gran cantidad de operaciones criptográficas.
- Tiene pocos requerimientos sobre el dispositivo en el que se vayan a ejecutar las aplicaciones que hagan uso de ella.
- Es fácil de aprender, instalar y manejar.
- Tiene un buen soporte técnico.
- Es gratuita para desarrollos de investigación y cualquier otro propósito no comercial.

Así, la librería criptográfica construida a partir de la de IAIK deberá tener, como mínimo, las siguientes funcionalidades:

- Mostrar el contenido de la tarjeta inteligente (certificados digitales, DNI, número de serie, claves públicas, etc.)
- Realizar operaciones de firma y verificación de firma
- Realizar operaciones de cifrado y descifrado
- Realizar operaciones de resumen digital o *hash*
- Realizar operaciones de generación de pares de claves

Todas las funciones que realicen las operaciones criptográficas descritas (a saber, firma, verificación de firma, cifrado, descifrado, resumen digital y generación de pares de claves) deben tener como parámetro de entrada el algoritmo criptográfico (DES, RSA, MD5, etc.) con el que se llevarán a cabo.

I.9. Aplicaciones criptográficas J2ME

Por último se desarrollarán aplicaciones J2ME que, haciendo uso de la librería criptográfica previamente construida, accedan a la tarjeta inteligente, recuperen la información en ella contenida y realicen operaciones criptográficas orientadas a securizar algún aspecto tal como transacciones electrónicas.

Algunas aplicaciones J2ME piloto de este estilo podrían ser, por ejemplo, un portafirmas o un gestor de credenciales.

I.10. Ventajas e inconvenientes

I.10.1. Ventajas

Este primer escenario aumenta la seguridad de los desarrollos realizados en FaCTo, ya que ahora la información criptográfica se encuentra almacenada en la tarjeta y no directamente en el dispositivo.

Además, para acceder a la información privada de la tarjeta, se requiere que el usuario inserte su PIN, por lo que nos encontramos con un sistema de seguridad fuerte, ya que el usuario posee la tarjeta y conoce su PIN.

Como se ha comentado anteriormente, ya existía un módulo PKCS#11 desarrollado para PC, por lo que el trabajo consistiría más bien en adaptar este módulo para dispositivos móviles y no en tener que crear uno nuevo desde cero, lo que constituye una verdadera ventaja de este escenario.

Por último, todas las aplicaciones criptográficas J2ME que se desarrollen para este escenario serán igualmente válidas para el resto de escenarios y asimismo portables a otros sistemas operativos.

I.10.2. Inconvenientes

Aunque en la introducción se dijo que la utilización de Java nos permitiría abstraernos del sistema operativo subyacente en el dispositivo, gracias a la JVM, en este escenario en concreto esto no es del todo cierto.

Si bien es verdad que tanto la librería criptográfica J2ME desarrollada como las aplicaciones J2ME piloto pueden ejecutarse en un dispositivo móvil con el único requisito de tener instalada una JVM apropiada, si no se cuenta con una librería dll que implemente el estándar PKCS#11 para la plataforma con la que se esté trabajando, dichas aplicaciones no podrán realizar ninguna operación que tenga que ver con el token.

Por lo tanto, podemos decir que este escenario sí es independiente del sistema operativo subyacente, siempre que se cuente con dicha librería dll.

Otro inconveniente que cabría resaltar es que no todos los dispositivos móviles, ni siquiera todas las PDAs, cuentan con una ranura SDIO. Por lo tanto, una solución basada en un lector de tarjetas de este estilo está restringida a aquellos dispositivos que sí cuenten con una interfaz de comunicación SDIO.

Es más, el propio hecho de tener que utilizar un lector de tarjetas externo, aparte del propio dispositivo móvil, ya supone tener que adquirirlo, con el consecuente gasto económico (actualmente, entre 200 y 300 euros), y llevarlo encima lo cual, en algunas ocasiones puede resultar ser un engorro.

Y no podemos obviar que la adaptación del módulo PKCS#11 para PC a un entorno Pocket PC resulta ser una tarea minuciosa y muy laboriosa.

Implementación

I.11. Introducción

Llegados a este punto, describiremos someramente todo el trabajo realizado con el objetivo de desarrollar una implementación estable del diseño que se acaba de plantear en la sección anterior.

En primer lugar se hablará de cómo se consiguió adaptar el módulo PKCS#11 existente para PC a un entorno de Pocket PC. Se describirá brevemente el entorno de desarrollo utilizado, así como los principales componentes del nuevo módulo PKCS#11.

A continuación se mostrarán algunos resultados obtenidos tras comprobar el correcto funcionamiento del nuevo módulo en un dispositivo móvil.

El siguiente punto consistirá en describir el proceso de instalación de la máquina virtual de Java seleccionada, para después mostrar cómo fue el desarrollo de la librería criptográfica J2ME basada en la librería de IAIK.

Por último, se explicará cuáles fueron las aplicaciones criptográficas J2ME implementadas y sus principales características.

I.12. Adaptación del módulo PKCS#11 para PC

La cuestión que aquí se plantea consiste en la adaptación del módulo PKCS#11 (librería `pkcs11.dll`) desarrollado dentro del proyecto FaCTo para PC a un entorno para Pocket PC.

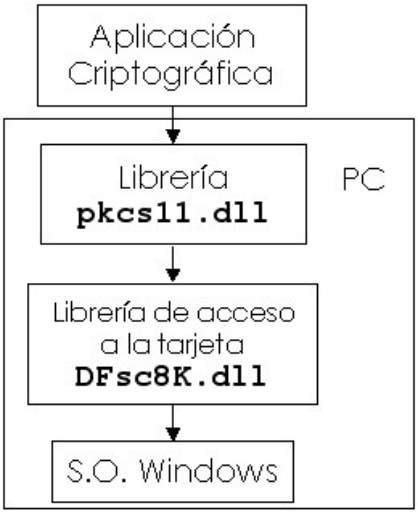

Figura I.14: Arquitectura del módulo `pkcs11.dll`

Como se observa en la figura I.14, este módulo PKCS#11 no es otra cosa que una librería dll compuesta principalmente por dos elementos: la librería de acceso a la tarjeta propiamente dicha y la parte del código que implementa el estándar PKCS#11 en sí mismo.

I.12.1. Entorno de desarrollo: Microsoft eMbedded Visual C++ 4.0

Empezaremos describiendo el entorno de desarrollo en el cual se llevaron a cabo todas las transformaciones necesarias para lograr la adaptación que pretendíamos. Dicho entorno fue *Microsoft eMbedded Visual C++ 4.0*.

Tras instalarlo satisfactoriamente, el primer paso consiste en crear un nuevo proyecto. Para ello pinchamos en *File->New* o bien pulsamos Ctrl+N. Entonces nos aparecerá una ventana como la mostrada en la figura I.15.

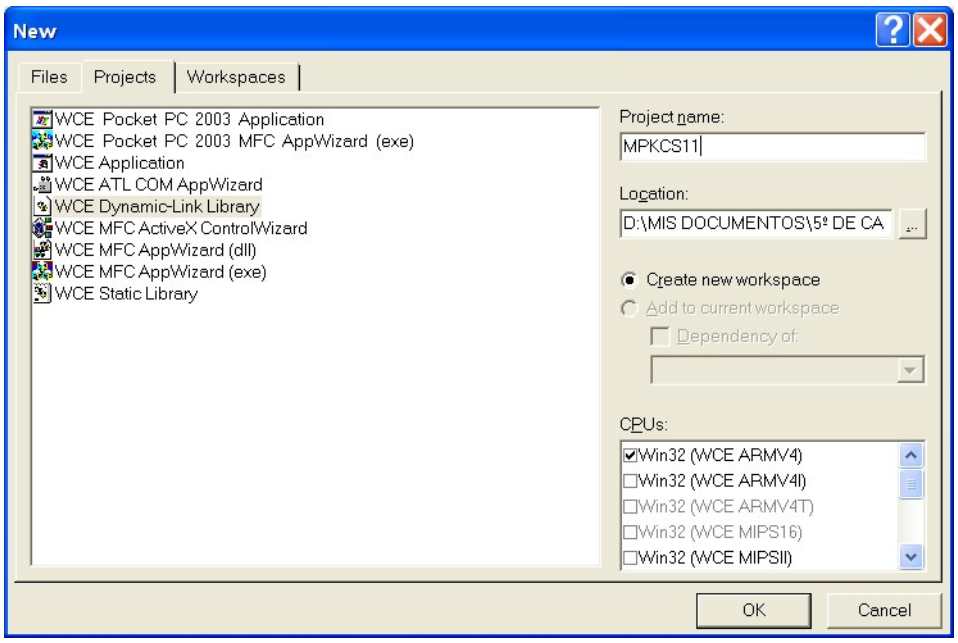

Figura I.15: Creación de un nuevo proyecto (I)

En dicha ventana, pinchamos sobre la pestaña *Projects*, seleccionamos *WCE-Dynamic Link Library*, damos un nombre a nuestro proyecto (en nuestro caso, MPKCS11), en la lista de CPUs seleccionamos *Win32 (WCE ARMV4)* y pulsamos *OK*.

Entonces nos parecerá una ventana como la de la figura I.16, en la cual tendremos que seleccionar la opción *A DLL that exports some symbols* y pulsar el botón *Finish*. Luego nos aparece otra ventana con información sobre el proyecto que vamos a crear; pulsamos *OK*.

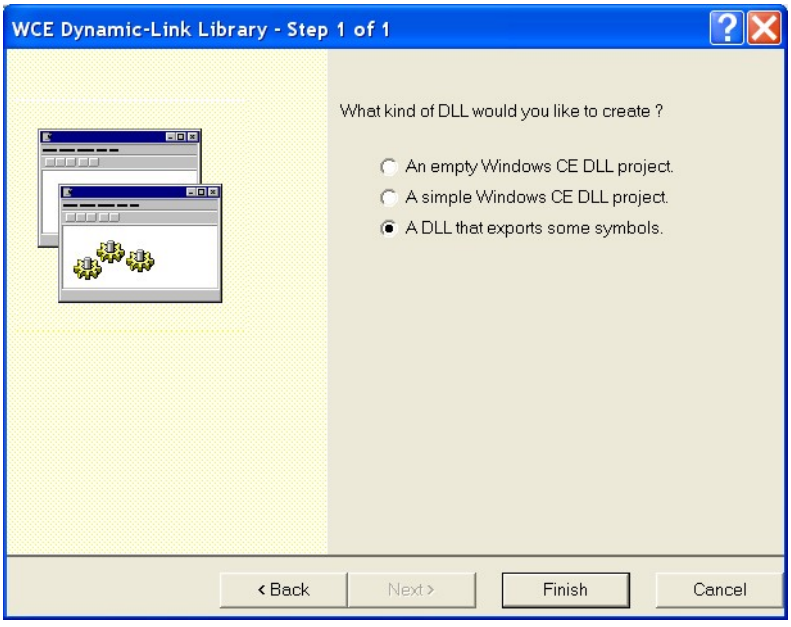

Figura I.16: Creación de un nuevo proyecto (II)

Una vez hayamos creado el proyecto, se copian todos los ficheros `.cpp` y `.h` del proyecto `pkcs11.dll` en la nueva carpeta que se habrá creado. Y entonces se siguen los pasos que se describen en la imagen de la figura I.17.

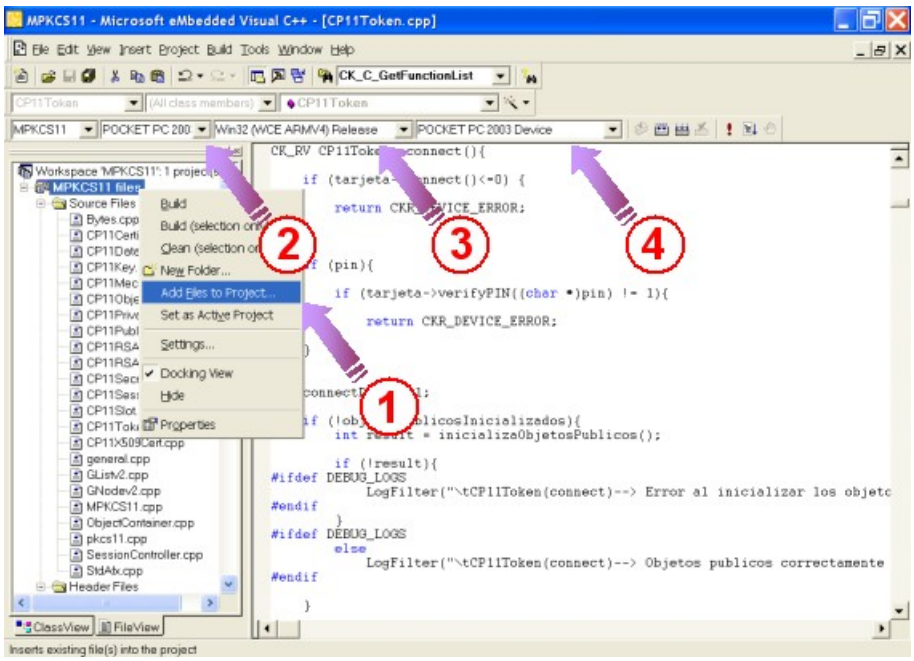

Figura I.17: Primeras configuraciones del entorno de desarrollo

1. En el panel situado a la izquierda, se pincha con el botón derecho del ratón sobre *MPKCS11 Files->Add Files to Project...*. Se nos abre un explorador de archivos donde seleccionamos todos los archivos `.cpp` y `.h` que hemos copiado en el directorio del nuevo proyecto y pulsamos *OK*.

2. Seleccionamos *POCKET PC 2003*.

3. Seleccionamos *Win32 (WCE ARMV4) Release*.

4. Seleccionamos *POCKET PC 2003 Device*.

El siguiente paso consiste en añadir todas las librerías de las que depende nuestro proyecto e indicarle rutas adicionales donde buscar ficheros `.h`. Para ello debemos acceder a las propiedades del proyecto, o bien pinchando en el menú *Project->Settings...*, o bien pulsando Alt+F7.

Nos aparecerá entonces una ventana como la que se observa en la figura I.18. Pinchamos en la pestaña *Link*, en *Category* seleccionamos *Input* y en *Object/library modules* añadimos todas las librerías que sean necesarias.

En nuestro caso estas librerías son: `MDFsc8Kdll.lib`, `commctrl.lib`, `libeay32.lib`, `ssleay32.lib`, `wcecompat.lib` y `winscard.lib`.

Por último, en *Additional library path* añadimos las rutas donde están las librerías que hemos añadido anteriormente, y que en nuestro caso son: `X:\OpenSSL-0.9.7-PPC\out32_ARM` y `X:\OpenSSL-0.9.7-PPC\wcecompat\lib`.

> **Nota.-** El orden en el que se insertan las librerías en *Object/library modules* es importante, debido a las posibles dependencias entre ellas. Una inserción de las mismas en un orden incorrecto puede provocar fallos a la hora de compilar el módulo PKCS#11.

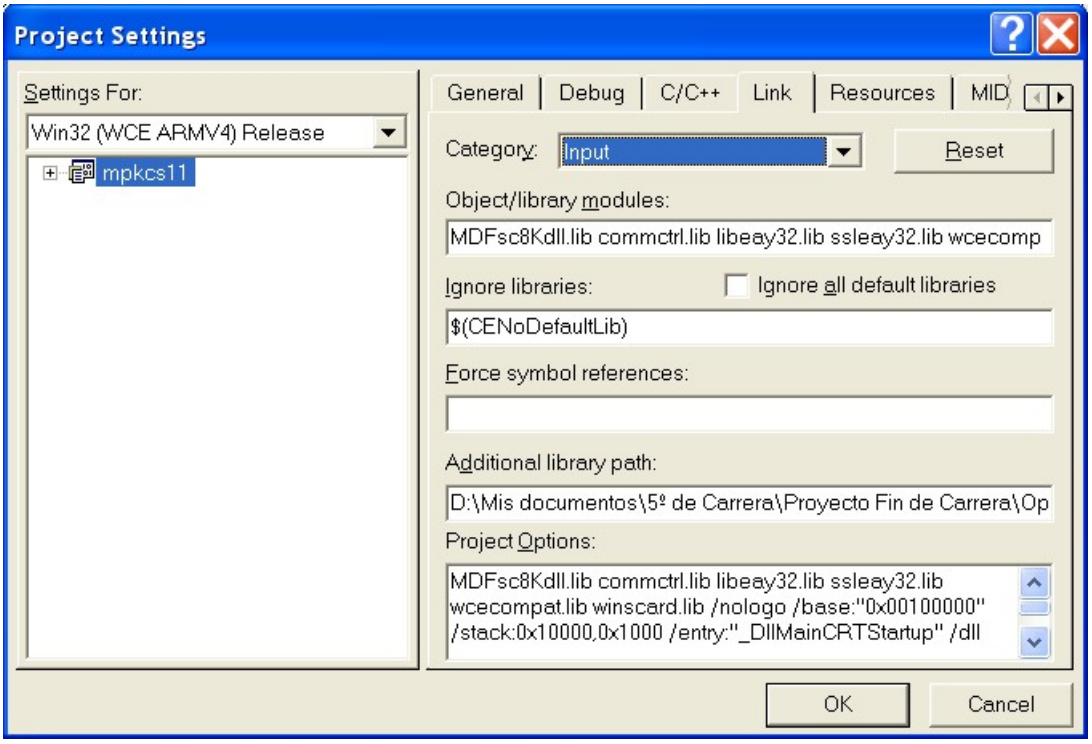

Figura I.18: Propiedades del proyecto (I)

En la misma ventana de *Project Settings*, pinchamos en la pestaña *C/C++* y en *Category* seleccionamos *Preprocessor*. En el campo *Additional include directories* escribimos los directorios donde se deberán buscar los ficheros `.h` adicionales empleados en el proyecto.

En nuestro caso dichos directorios son los siguientes: `X:\OpenSSL-0.9.7-PPC\inc32` y `X:\OpenSSL-0.9.7-PPC\wcecompat\include`.

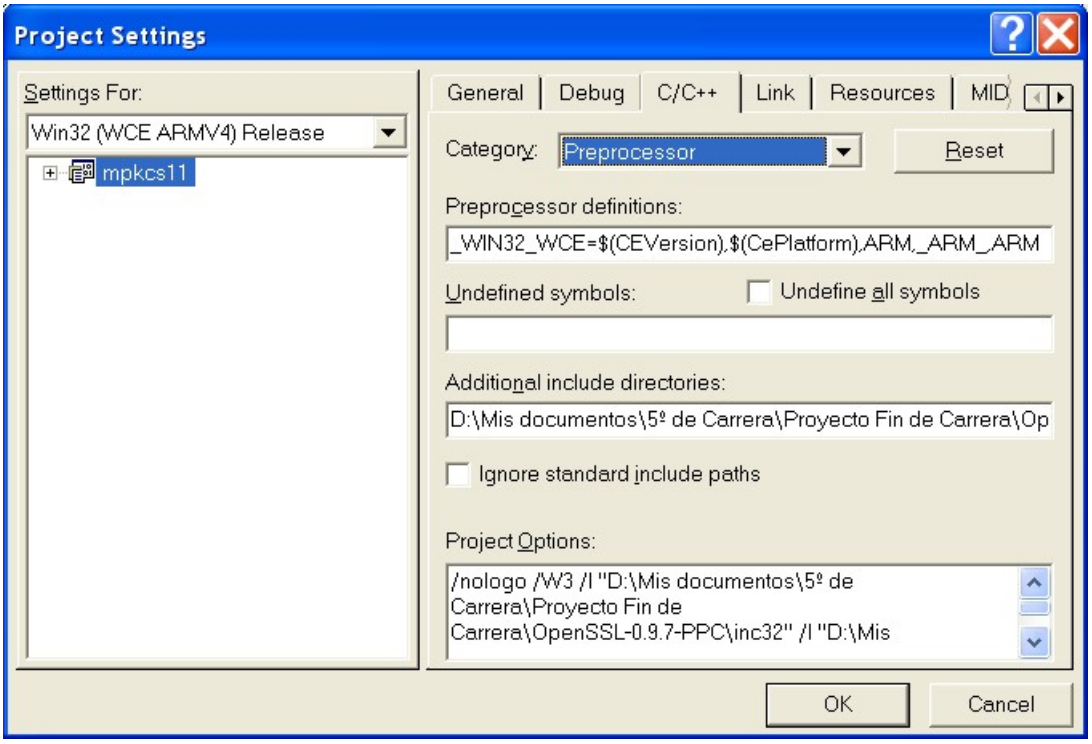

Figura I.19: Propiedades del proyecto (II)

Llegados a este punto, ya se está en condiciones para comenzar a modificar el código con el propósito de adaptarlo a la nueva plataforma requerida.

I.12.2. Incompatibilidades de tipos de datos

Las principales incompatibilidades de tipos de datos encontradas en la adaptación del módulo PKCS#11 para PC fueron las referentes al tipo de datos `char*`, el cual no está soportado por la mayoría de las funciones que existen para la plataforma Windows CE.

En dicha plataforma, `char*` debe ser sustituido por el tipo de datos `TCHAR`, definido en el fichero de cabecera `tchar.h`.

Texto genérico o nombre de tipo de datos	**_UNICODE y _MBCS no definidos**	**_MBCS definido**	**_UNICODE definido**
_TCHAR	char	char	wchar_t
_TINT	int	int	wint_t
_TSCHAR	signed char	signed char	wchar_t
_TUCHAR	unsigned char	unsigned char	wchar_t
_TXCHAR	char	unsigned char	wchar_t
_T o _TEXT	Sin efecto (eliminado por el preprocesador)	Sin efecto (eliminado por el preprocesador)	L (convierte el carácter o cadena siguiente en su equivalente de Unicode)

Tabla I.10: Asignaciones de tipos de datos de texto genérico

En la tabla I.10 se muestra la equivalencia de `TCHAR` [53] y otros tipos de datos, dependiendo de si están definidas o no las macros `_UNICODE` y `_MBCS`. En nuestro proyecto `_UNICODE` sí está definido.

I.12.3. Winscard

Para desarrollar la librería de acceso a la tarjeta se hace uso de la librería Winscard [40], la cual nos ofrece todas las funciones enumeradas en la tabla I.11.

Funciones de la librería Winscard	
SCardEstablishContext	SCardStatus
SCardReleaseContext	SCardGetStatusChange
SCardSetTimeout	SCardControl
SCardConnect	SCardTransmit
SCardReconnect	SCardListReaderGroups
SCardDisconnect	SCardListReaders
SCardBeginTransaction	SCardCancel
SCardEndTransaction	SCardGetAttrib
SCardCancelTransaction	SCardSetAttrib

Tabla I.11: Funciones de la librería Winscard

Dicha librería nos permitirá manejar con cierta facilidad la comunicación con el lector de tarjetas SDIO, mediante funciones como las de establecer un contexto, liberarlo, conectar con la tarjeta, desconectarnos de ella, mostrar todos los lectores de tarjetas enchufados al dispositivo, o enviar APDUs, entre otras.

Sin embargo, no nos conviene usar estas funciones de "tan bajo nivel" directamente en nuestro módulo PKCS#11, y es por ello que construimos otro módulo específico de acceso a la tarjeta, que ofrecerá una API de un nivel un poco más alto, gracias a la cual se facilitará la programación de muchas operaciones en el módulo PKCS#11 relacionadas con la comunicación con la tarjeta criptográfica.

I.12.4. MDFsc8Kdll

Como se observa en la figura I.14, en el módulo PKCS#11 para PC ya existe una librería de acceso a la tarjeta denominada `DFsc8K.dll`. Por lo tanto, nuestro siguiente (sub)objetivo consistió en la adaptación de este módulo de acceso a la tarjeta para PC al nuevo entorno de Pocket PC.

Para ello, en primer lugar se siguieron exactamente los mismos pasos que se han descrito en la sección I.12.1 mediante los cuales se crea un proyecto de desarrollo de una dll que exporta algunos métodos.

El nombre del nuevo proyecto es `MDFsc8Kdll`, y las librerías añadidas esta vez fueron: `commctrl.lib`, `coredll.lib`, `wcecompat.lib`, `ssleay32.lib`, `libeay32.lib` y `ttscard.lib`.

Las rutas para dichas librerías son: `X:\MDFsc8Kdll`, `X:\OpenSSL-0.9.7-PPC\out32_ARM` y `X:\OpenSSL-0.9.7-PPC\wcecompat\lib`. Y las rutas donde buscar ficheros `.h` adicionales son las mismas que en el proyecto `MPKCS11`.

Haciendo esto y modificando los tipos de datos pertinentes, no sin inumerables quebraderos de cabeza, se consiguió compilar satisfactoriamente esta librería de acceso a la tarjeta y se comprobó su correcto funcionamiento en el dispositivo móvil.

Método	**Descripción**
`MDFsc8K()`	Constructor de la clase
`~MDFsc8K()`	Destructor de la clase
`int connect()`	Establece la conexión con la tarjeta
`int readPrivateKey(Bytes **privKey)`	Lee la clave privada
`int generatePrivateKey(int type, int bits)`	Borra clave privada, certificado y certificado de CA y genera una clave privada que inserta en el campo correspondiente
`int readCert(Bytes **cert)`	Lee el certificado de usuario
`int readCACert(Bytes **cert)`	Lee el certificado de la CA
`int readDNI(Bytes **dni)`	Lee el DNI del usuario
`int verifyPIN(char *pin)`	Verifica el PIN del usuario
`int setCertificate(Bytes *certificate)`	Establece el certificado de usuario en formato DER

Tabla I.12: Métodos públicos de la clase `MDFsc8K`

La tabla I.12 muestra los métodos públicos de la clase `MDFsc8K` que serán exportados en la dll correspondiente, junto con una breve descripción de los mismos. De entre todos ellos, solamente `generatePrivateKey` hace uso de las funciones de la librería de OpenSSL.

I.12.5. OpenSSL-0.9.7-PPC

El proyecto OpenSSL [21] es un esfuerzo colaborativo para desarrollar un juego de herramientas robusto, con todas las capacidades y de código libre, que implemente los protocolos SSL y TLS, así como una librería criptográfica de propósito general.

En nuestro módulo PKCS#11 hemos usado la versión 0.9.7 compilada para PocketPC principalmente en la generación de claves y certificados, generación de los parámetros de RSA, DH y DSA y, en general, en la generación de información aleatoria.

I.12.6. MPKCS11dll

Una vez se hayan realizado todos los pasos indicados en la sección I.12.1 y se haya desarrollado correctamente la librería `MDFsc8Kdll.dll`, lo siguiente que hay que hacer es añadir el fichero `MDFsc8K.h` creado en el proyecto de la librería de acceso a la tarjeta en el proyecto del módulo PKCS#11 para PokcetPC.

Tras añadir dicho fichero de cabecera en el proyecto, copiar los ficheros `MDFsc8Kdll.lib` y `MDFsc8Kdll.dll` en el directorio `X:\mpkcs11` y realizar los oportunos cambios en los tipos de datos, compilamos, no sin innumerables quebraderos de cabeza igualmente, la librería `mpkcs11.dll`.

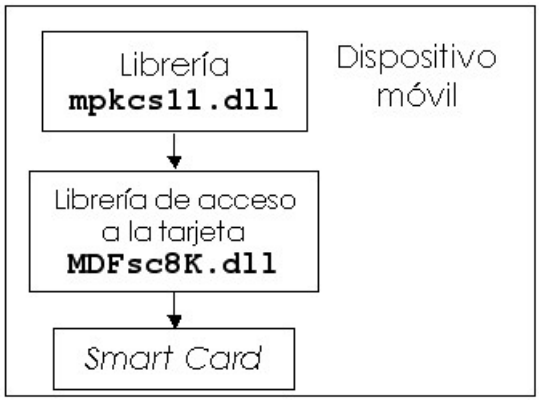

Figura I.20: Arquitectura del módulo `mpkcs11.dll`

En el momento en que se tuvo compilada la librería y se hicieron las primeras pruebas sobre el dispositivo móvil, inmediatamente se observó que la eficiencia de dicha librería dejaba bastante que desear.

Y es que no podía ser de otra manera, ya que esa librería era en realidad la librería que en su momento fue desarrollada para PC, un entorno, evidentemente, con muchos más recursos en cuanto a CPU, memoria, etc., que cualquier dispositivo móvil de hoy en día.

Por este motivo, se decidió cambiar la estructura en dos capas que tenía la librería para PC y que consistía en que, para cada operación criptográfica, se comprobaba si el token era capaz de realizarla; si era capaz, la realizaba el token y, en caso contrario, la realizaba el propio módulo PKCS#11.

Con el objetivo de mejorar la eficiencia en la versión de la librería para PocketPC, se eliminaron esas comprobaciones, de manera que en esta nueva librería, todas las operaciones criptográficas las lleva siempre a cabo el propio módulo PKCS#11 y no el token en sí, aunque éste fuera capaz de realizarlas.

Clases principales de la librería `mpkcs11.dll`	
`CP11Certificate`	`CP11RSAPrivKey`
`CP11Data`	`CP11RSAPubKey`
`CP11Key`	`CP11SecretKey`
`CP11Mechanism`	`CP11Session`
`CP11Objet`	`CP11Slot`
`CP11PrivateKey`	`CP11Token`
`CP11PublicKey`	`CP11X509Cert`

Tabla I.13: Clases principales de la librería `mpkcs11.dll`

I.13. Pruebas en el dispositivo móvil

Con la librería `mpkcs11.dll` ya compilada y sin errores, se realizaron varias pruebas sobre el dispositivo móvil para comprobar su correcto funcionamiento.

La prueba más completa de todas incluyó, la inicialización del token, la conexión con el mismo, la creación de sesiones, la creación de objetos, la búsqueda de objetos, la obtención de información sobre los mecanismos, el cifrado y descifrado de datos, la destrucción de objetos y sesiones, la desconexión con el token y la finalización del mismo.

El fichero de log resultante de realizar todas estas operaciones tenía el siguiente contenido:

```
--------MPKCS11Test LogFile---------

C_Initilize OK
C_GetSlotList OK
El número de elementos es: 1
C_GetSlotInfo OK
Manufacturer ID: Universidad de Murcia
C_OpenSession OK
Session (10124518) abierta Correctamente
C_Login OK
C_OpenSession OK
Session (10118086) abierta Correctamente
C_CreateObject OK
C_OpenSession OK
Session (10117302) abierta Correctamente
C_EncryptInit OK
C_EncryptUpdate OK
C_Encrypt OK
C_EncryptFinal OK
C_CreateObject OK
C_DecryptInit OK
C_DecryptUpdate OK
C_Decrypt OK
C_DecryptFinal OK
C_CloseSession OK
C_GetMechanismInfo OK
C_FindObjectsInit OK
C_FindObjects OK
C_FindObjectsFinal OK
C_OpenSession OK
Session (10111734) abierta Correctamente
C_DestroyObject OK
C_CloseSession OK
C_CloseAllSessions OK
C_Finalize OK
```

I.14. Instalación de la máquina virtual de Java

La instalación de la máquina virtual CrEme es bien sencilla si se siguen los siguientes pasos [16]:

1. Instalar `CrE-ME401_ARM_CE42_PPC.CAB` en el PocketPC.

   Aparecerá en la carpeta `\Windows\CrEme` y en la carpeta `Programas\Creme`.

2. Instalar `CrEmeDevSup400.exe` en el PC. El directorio donde se queda instalado es `C:\Archivos de programa\NSIcom\CrE-ME V4.00`. Es posible que aparezcan mensajes de error al acabar la instalación, con ignorarlos se arregla. Se debe a que la instalación busca la carpeta `Program Files`, pero en Windows castellano es `Archivos de Programa`.

3. Copiar todos los archivos y carpetas que hay en la carpeta `lib` del directorio creado en el paso 2 y **que no existan ya** en la carpeta `\Windows\CrEme\lib` del PocketPC. ¡NO sobreescribir ningún archivo existente en esa carpeta! Esto es necesario para que la distribución del PPC sea completa, con las librerías `swing` y `rmi`, por ejemplo.

4. Las librerías que normalmente van en la carpeta `ext` de las distribuciones JVM también han de ser copiadas en `\Windows\CrEme\lib` del PocketPC. Por ejemplo, en nuestro caso hay que traer la librería `iaik_p11_me.jar`. Aunque el manual de CrEme diga que las pongamos ahí, no hacer caso, copiarlas en la carpeta `lib` de Creme en el PocketPC.

5. Para ejecutar la JVM de CrEme, ir a `Programas\Creme\` y ejecutar JRun. Aparece una línea de comandos en la que se han de meter los parámetros; para más información sobre esto ver el manual de CreMe que está en la carpeta `doc` del directorio creado en el PC en el paso 2. Aquí pongo un ejemplo de cómo se puede ejecutar una aplicación:

   ```
   -Ob -wd \j2mempkcs11\ -ml 30000 -jar j2mempkcs11.jar
   ```

   `-Ob` es para que aparezca una consola que muestra los mensajes del compilador Java de CrEme. Otra opción sería `-Of`, que escribe estos mensajes en un archivo (`jscpout.txt`) en el directorio raíz del PocketPC.

   `-wd \j2mempkcs11\` indica que el directorio de trabajo es el de mi proyecto.

   `-ml 30000` le indica al PocketPC cuánta memoria va utilizar la aplicación, en este caso 30MB.

6. Finalmente, pinchar sobre el botón `Run` de la línea de comandos de Jrun, y en breves instantes comienza la aplicación.

I.15. Desarrollo de la librería criptográfica J2ME

La implementación de la librería criptográfica J2ME diseñada en la sección I.8 (página 47) se ha llevado a cabo de la siguiente manera: se ha creado una clase abstracta llamada `J2MEMPKCS11` que contiene todos los métodos descritos en las tablas I.14 y I.15.

Todos y cada uno de dichos métodos han sido declarados como estáticos, de tal forma que el empleo de esta librería en otra aplicación J2ME es bien sencillo. Un ejemplo de código que usara esta librería para cifrar, por ejemplo, podría ser el siguiente:

```
...
J2MEMPKCS11.connect(pin);
J2MEMPKCS11.encrypt(datosParaCifrar);
J2MEMPKCS11.disconnect();
...
```

Como se puede observar, para cada operación criptográfica se han implementado varios métodos, con la intención de que el programador que los utilice pueda decidir si quiere usar el algoritmo y/o la clave por defecto, o bien, introducir él mismo los que considere oportunos.

Métodos de conexión y desconexión	
`void connect(char[] pin)`	Se conecta al primer token que encuentra y hace *login*
`void disconnect()`	Hace *logout* y se desconecta del token
Métodos para cifrar	
`byte[] encrypt (byte[] data, long algorithm, Key encryptKey)`	Cifra los datos *data* usando el algoritmo *algorithm* y la clave *encryptKey*
`byte[] encrypt (byte[] data, long algorithm)`	Cifra los datos *data* usando el algoritmo *algorithm* y una clave por defecto
`byte[] encrypt (byte[] data, Key encryptKey)`	Cifra los datos *data* usando un algoritmo por defecto y la clave *encryptKey*
`byte[] encrypt (byte[] data)`	Cifra los datos *data* usando un algoritmo por defecto y una clave por defecto
Métodos para descifrar	
`byte[] decrypt (byte[] cipheredData, long algorithm, Key decryptKey)`	Descifra los datos *cipheredData* usando el algoritmo *algorithm* y la clave *decryptKey*
`byte[] decrypt (byte[] cipheredData, long algorithm)`	Descifra los datos *cipheredData* usando el algoritmo *algorithm* y una clave por defecto
`byte[] decrypt (byte[] cipheredData, Key decryptKey)`	Descifra los datos *cipheredData* usando un algoritmo por defecto y la clave *decryptKey*
`byte[] decrypt (byte[] cipheredData)`	Descifra los datos *cipheredData* usando un algoritmo por defecto y una clave por defecto
Métodos para hacer resumen digital	
`byte[] hash (byte[] data, long algorithm)`	Hace un resumen digital de los datos *data* usando el algoritmo *algorithm*
`byte[] hash (byte[] data`	Hace un resumen digital de los datos *data* usando un algoritmo por defecto

Tabla I.14: Métodos de la librería criptográfica J2ME (I)

Métodos para firmar	
`byte[] sign (byte[] dataToBeSigned, long algorithm, Key signKey)`	Firma los datos *dataToBeSigned* usando el algoritmo *algorithm* y la clave *signKey*
`byte[] sign (byte[] dataToBeSigned, long algorithm`	Firma los datos *dataToBeSigned* usando el algoritmo *algorithm* y una clave por defecto
`byte[] sign (byte[] dataToBeSigned, Key signKey)`	Firma los datos *dataToBeSigned* usando un algoritmo por defecto y la clave *signKey*
`byte[] sign (byte[] dataToBeSigned`	Firma los datos *dataToBeSigned* usando un algoritmo por defecto y una clave por defecto
Métodos para verificar firmas	
`boolean verifySignature (byte[] data, byte[] signedData, long algorithm, Key verifySignatureKey)`	Verifica si *signedData* es la firma de los datos *data* usando el algoritmo *algorithm* y la clave *verifySignatureKey*
`boolean verifySignature (byte[] data, byte[] signedData, long algorithm`	Verifica si *signedData* es la firma de los datos *data* usando el algoritmo *algorithm* y una clave por defecto
`boolean verifySignature (byte[] data, byte[] signedData, Key verifySignatureKey)`	Verifica si *signedData* es la firma de los datos *data* usando un algoritmo por defecto y la clave *verifySignatureKey*
`boolean verifySignature (byte[] data, byte[] signedData`	Verifica si *signedData* es la firma de los datos *data* usando un algoritmo por defecto y una clave por defecto
Métodos para generar pares de claves	
`Key[] generateKeyPair(String name, long algorithm, int bits)`	Genera un par de claves de *bits* bits con nombre *name* usando el algoritmo *algorithm*
`Key[] generateKeyPair(String name, int bits)`	Genera un par de claves de *bits* bits con nombre *name* usando un algoritmo por defecto
Métodos para recuperar claves	
`Key getSignKey()`	Devuelve una clave para firmar
`Key getVerifySignatureKey()`	Devuelve una clave para verificar firmas
`Key getEncryptKey()`	Devuelve una clave para cifrar
`Key getDecryptKey()`	Devuelve una clave para descifrar
Métodos para obtener algoritmos	
`long getSignAndVerifySignAlgorithm()`	Devuelve un algoritmo con el que firmar y verificar firmas
`long getEncryptAndDecryptAlgorithm()`	Devuelve un algoritmo con el que cifrar y descifrar
`long getHashAlgorithm()`	Devuelve un algoritmo con el que hacer resumen digital
Métodos para obtener información del token	
`String supportedAlgorithms()`	Devuelve una cadena con todos los algoritmos soportados por el token
`String tokenContent()`	Devuelve una cadena con todo el contenido (claves, certificados, ...) del token

Tabla I.15: Métodos de la librería criptográfica J2ME (II)

Escenario II

Tarjeta Externa y Lector Bluetooth

Visión General

Figura II.1: Segundo escenario: tarjeta externa y lector Bluetooth

En este segundo escenario disponemos de una tarjeta inteligente como la de la Universidad de Murcia o (potencialmente) el propio DNI-e, un lector de tarjetas Bluetooth y una PDA, o bien un teléfono móvil, o bien un *Smartphone* (una PDA con funcionalidad telefónica).

El **análisis** de J2ME, así como de las máquinas virtuales de java más adecuadas para dispositivos móviles que sirvieron para el primer escenario, sirven también para el segundo. Además, en dicha fase de análisis se incluirá también un breve estudio del protocolo de comunicación Bluetooth [7], así como su correspondiente API de Java [11, 27].

En la fase de **diseño**, también se mostrará la arquitectura en capas del escenario en cuestión y se expondrán los componentes que deberán ser desarrollados en la siguiente fase de implementación.

Dichos componentes serán, básicamente: una máquina virtual de Java adecuada para dispositivos móviles, una librería de comunicación en Java que haga uso del protocolo Bluetooth y una aplicación J2ME similar a las desarrolladas para el primer escenario que, haciendo uso de dicha librería y de una librería criptográfica J2ME, acceda a la *smart card* y sea capaz de realizar algunas operaciones criptográficas como las que se plantearon en el escenario anterior.

Análisis

II.1. Breve introducción a J2ME

Todo el análisis realizado en la sección I.1 de la página 21 es igualmente válido para este escenario. Para más información, consúltese dicho apartado.

II.2. Bluetooth

II.2.1. Arquitectura Bluetooth

La unidad básica de un sistema Bluetooth [7] es una *piconet*, que consta de un nodo maestro (*master*) y hasta siete nodos esclavos (*slave*) activos a una distancia de 10 metros. Varias *piconets* pueden conectarse a través de nodos puente, como se muestra en la figura II.2, formando lo que se denomina una *scatternet*.

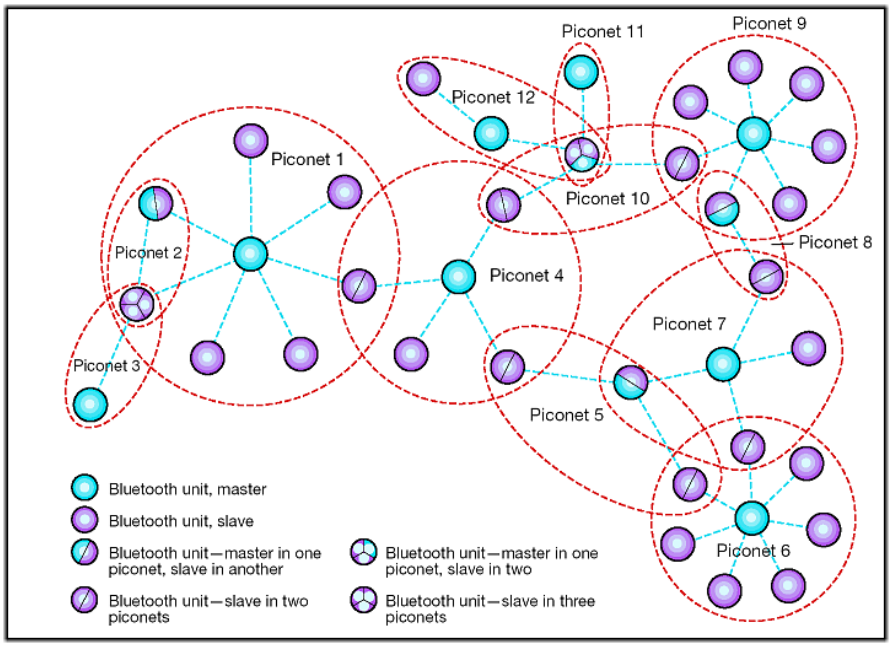

Figura II.2: *Scatternet* formada por 12 *piconets*

Además de los siete nodos esclavos activos de una *piconet*, puede haber hasta 255 nodos estacionarios en la red. Éstos son dispositivos que el maestro ha cambiado a un estado de bajo consumo para reducir el desgaste innecesario de sus baterías. Lo único que un dispositivo en estado estacionario puede hacer es responder a una señal de activación por parte del maestro.

En esencia, una *piconet* es un sistema TDM centralizado, en el cual el maestro controla el reloj y determina qué dispositivo se comunica en un momento determinado. Todas las comunicaciones se realizan entre el maestro y el esclavo; no existe comunicación directa de esclavo a esclavo.

II.2.2. Perfiles de Bluetooth

Para poder utilizar Bluetooth, un dispositivo debe ser compatible con ciertos perfiles de Bluetooth [50]. El SIG de Bluetooth define y adopta los perfiles mostrados en la tabla II.1.

Perfil	**Descripción**
Advanced Audio Distribution Profile	Transferencia de un flujo de audio estéreo
Audio/Video Remote Control Profile	Interfaz estándar para control remoto de TVs, Hi-Fi,...
Basic Imaging Profile	Envío de imágenes
Basic Printing Profile	Envío de texto, e-mails, ... a impresoras
Common ISDN Access Profile	Acceso a servicios, datos y señales ofrecidas por un ISDN
Cordless Telephony Profile	Uso de teléfonos inalámbricos con Bluetooth
Device ID Profile	Identificación de un dispositivo
Dial-up Networking Profile	Acceso a internet y otros servicios de marcación sobre Bluetooth
Fax Profile	Interfaz entre un teléfono móvil y un PC con software para FAX
File Transfer Profile	Acceso al sistema de ficheros de otro dispositivo
General Audio/Video Distribution Profile	Es la base para A2DP y VDP
Generic Access Profile	Es la base para todos los demás perfiles
Generic Object Exchange Profile	Es la base para otros perfiles de transferencia de datos
Hard Copy Cable Replacement Profile	Alternativa inalámbrica a la conexión por cable entre un dispositivo y una impresora
Hands-Free Profile	Comunicación entre los kits manos libres de los coches y los teléfonos móviles
Human Interface Device Profile	Soporte para dispositivos como ratones, joysticks, teclados, etc.
Headset Profile	Soporte para auriculares Bluetooth usados con teléfonos móviles
Intercom Profile	Permite llamadas de voz entre dos auriculares Bluetooth
Objet Push Profile	Envío de "objetos" como imágenes, citas, etc.
Personal Area Networking Profile	Permite el uso del protocolo de encapsulación de red de Bluetooth
Phone Book Access Profile	Intercambio de objetos de la agenda entre dispositivos
Serial Port Profile	Emulación de una conexión a través de un cable serie
Service Discovery Application Profile	Permite averiguar los perfiles ofrecidos por un dispositivo
SIM Access Profile	Conexión entre dispositivos como teléfonos de coche y módulos SIM en teléfonos con Bluetooth
Synchronization Profile	Sincronización de elementos del PIM
Video Distribution Profile	Transporte de un flujo de vídeo
Wireless Application Protocol Bearer	Soporte de WAP sobre PPP sobre Bluetooth

Tabla II.1: Perfiles de Bluetooth

II.2.3. La pila de protocolos de Bluetooth

En la figura II.3 se observa la pila de protocolos de Bluetooth.

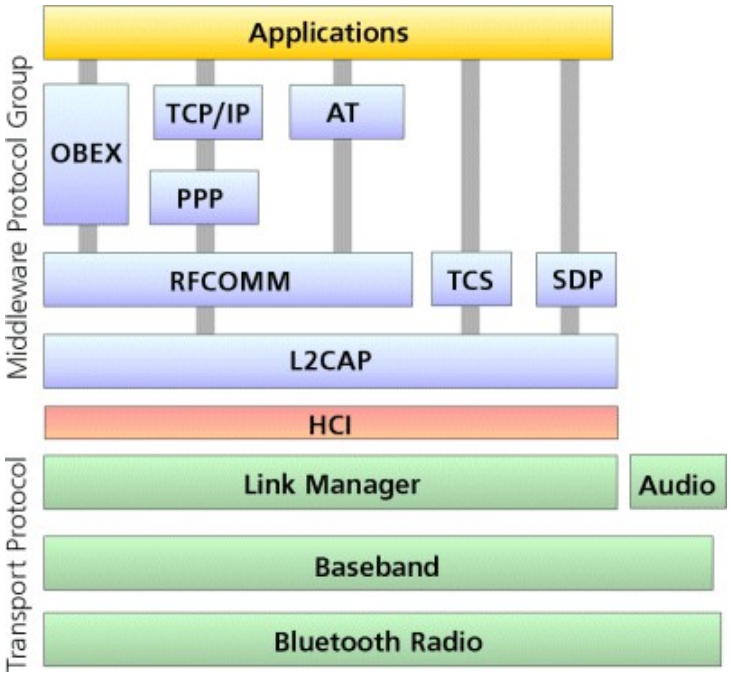

Figura II.3: Pila de protocolos de Bluetooth

A continuación describiremos las distintas capas y protocolos que forman la pila de protocolos de Bluetooth [7, 51].

La capa de radio

La capa de radio define los requisitos para un transmisor-receptor de radio Bluetooth, el cual opera en la banda de los 2'4 GHz. Define los niveles de sensibilidad de dicho transmisor-receptor, establece los requisitos para utilizar las frecuencias del espectro expandido y clasifica a los dispositivos Bluetooth en tres clases, según se indica en la tabla II.2.

Clase	**Potencia máxima permitida**		**Potencia mínima permitida**		**Distancia**
	mW	**dBm**	**mW**	**dBm**	
Clase 1	100 mW	20 dBm	1 mW	0 dBm	100 metros
Clase 2	2'5 mW	4 dBm	0'25 mW	-6 dBm	10 metros
Clase 3	1 mW	0 dBm	N/A	N/A	1 metros

Tabla II.2: Clases de dispositivos Bluetooth

La capa de banda base

La capa de banda base representa a la capa física de Bluetooth. Se usa como controladora de enlace, la cual trabaja junto con el *link manager* para llevar a cabo operaciones tales como la creación de conexiones de enlace con otros dispositivos.

Controla el direccionamiento de dispositivos, el control del medio (cómo los dispositivos se buscan unos a otros), operaciones de ahorro de energía, así como control de flujo y sincronización entre dispositivos.

Link Manager

El administrador de enlaces o *Link Manager*, se encarga de establecer canales lógicos entre dispositivos, incluyendo administración de energía, autenticación y calidad de servicio.

Host Controller Interface

El HCI [52] permite el acceso mediante línea de comandos a la capa de banda base y al LMP para controlar y recibir información acerca del estado. Se compone de tres partes:

1. El *firmware* HCI, o programa oficial del fabricante, actualmente forma parte del hardware Bluetooth.
2. El controlador HCI, o *driver*, que se encuentra en el software del dispositivo Bluetooth.
3. El *Host Controller Transport Layer*, que conecta el *firmware* con el *driver*.

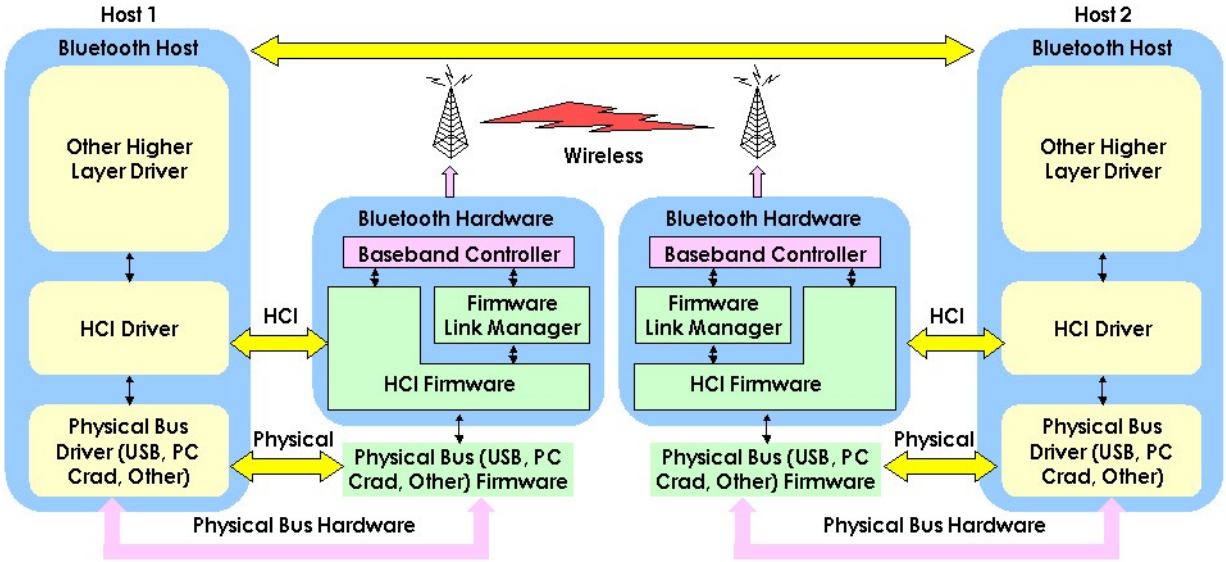

Figura II.4: *Host Controller Interface*

Protocolo de adaptación y control de enlaces lógicos L2CAP

El protocolo L2CAP aísla a las capas superiores de los detalles de transmisión. Es análogo a la subcapa LLC del estándar 802, pero difiere de ésta en el aspecto técnico. Proporciona servicios orientados y no orientados a la conexión a las capas superiores.

L2CAP permite a los protocolos de alto nivel y a las aplicaciones enviar y recibir paquetes de datos de hasta 64 Kb. Buena parte de su tiempo la dedica a la segmentación y reemsablado.

RFCOMM

RFCOMM es el protocolo que emula el puerto serie estándar RS232 de los ordenadores para la conexión de teclados, ratones y módems, entre otros dispositivos. Su propósito es que dichos dispositivos no tenga por qué saber nada acerca de Bluetooth.

TCS y SDP

El protocolo de telefonía TCS es un protocolo de tiempo real y está destinado a los tres perfiles orientados a voz (HFP, HSP e ICP). También se encarga del establecimiento y terminación de llamadas. Por su parte, el protocolo de descubrimiento de servicios (SDP) se emplea para la detección automática de dispositivos dentro de la red, así como de servicios ofrecidos por dichos dispositivos.

II.2.4. La capa de radio de Bluetooth

La capa de radio transfiere los bits del maestro al esclavo o viceversa. Es un sistema de baja potencia con un rango de entre 1 y 100 metros que opera en la banda ISM de los 2'4 GHz. La banda se divide en 79 canales de 1 MHz cada uno. La modulación es por división en frecuencia, con 1 bit por Hz, lo cual da una tasa de datos aproximada de 1 Mbps, pero gran parte de este espectro la consume la sobrecarga.

Para asignar los canales de manera equitativa, el espectro de saltos de frecuencia se utiliza a 1600 saltos por segundo y un tiempo de permanencia de 625 μseg. Todos los nodos de una *piconet* saltan de manera simultánea, y el maestro establece la secuencia de salto.

Debido a que tanto el 802.11 como Bluetooth operan en la banda ISM de 2'4 GHz en los mismos 79 canales, interfieren entre sí. Puesto que Bluetooth salta mucho más rápido que 802.11, es más probable que un dispositivo Bluetooth dañe las transmisiones del 802.11 que en el caso contrario.

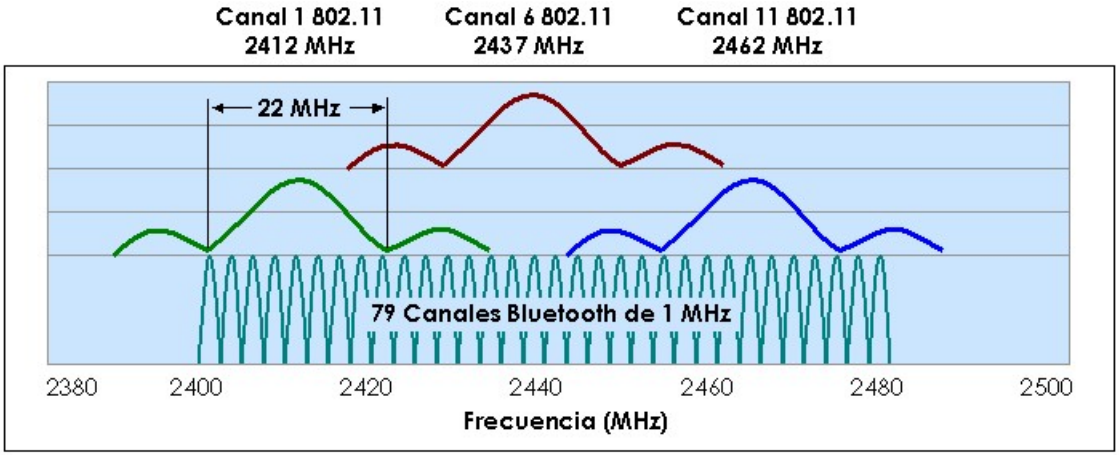

Figura II.5: Coexistencia de Bluetooth y 802.11 en la banda de los 2'4 GHz

II.2.5. La capa de banda base de Bluetooth

La capa de banda base de Bluetooth es lo más parecido a una subcapa MAC. Esta capa convierte el flujo de bits puros en tramas. En la forma más sencilla, el maestro de cada *piconet* define una serie de ranuras de tiempo de 625 μseg y las transmisiones del maestro empiezan en las ranuras pares, y las de los esclavos en las impares. Ésta es la tradicional multiplexación por división de tiempo, en la cual el maestro acapara la mitad de las ranuras y los esclavos comparten la otra mitad. Las tramas pueden tener 1, 3 ó 5 ranuras de longitud.

Cada trama se transmite por un canal lógico llamado enlace entre el maestro y un esclavo. Hay dos tipos de enlaces:

- ACL, que se utiliza para datos conmutados en paquetes disponibles a intervalos irregulares. Estos datos provienen de la capa L2CAP en el nodo emisor y se entregan en la capa L2CAP en el nodo receptor. No hay garantías de entrega del tráfico ACL. Las tramas se pueden perder y tienen que retransmitirse. Un esclavo sólo puede tener un enlace ACL con su maestro.

- SCO, para datos en tiempo real, como ocurre en las conexiones telefónicas. A este tipo de canal se le asigna una ranura fija en cada dirección. Las tramas que se envían a través de SCO nunca se retransmiten. En vez de ello se puede usar la corrección de errores hacia adelante para conferir una alta fiabilidad al sistema. Un esclavo puede establecer hasta tres enlaces SCO con su maestro. Cada enlace de este tipo puede transmitir un canal de audio PCM de 64000 bps.

II.2.6. La capa L2CAP de Bluetooth

La capa L2CAP tiene tres funciones principales:

1. Acepta paquetes de hasta 64 KB provenientes de las capas superiores y los divide en tramas para transmitirlos. Las tramas se reensamblan nuevamente en paquetes en el otro extremo.

2. Maneja la multiplexación y desmultiplexación de múltiples fuentes de paquetes. Cuando se reensambla un paquete, la capa L2CAP determina qué protocolo de las capas superiores lo manejará, por ejemplo, RFCOMM o el de telefonía.

3. Se encarga de la calidad de servicio, tanto al establecer los enlaces como durante la operación normal. Asimismo, durante el establecimiento de los enlaces se negocia el tamaño máximo de carga útil permitido, para evitar que un dispositivo que envíe paquetes grandes sature a uno que recibe paquetes pequeños.

II.2.7. Estructura de la trama de Bluetooth

En la figura II.6 se muestra el formato de una trama de Bluetooth.

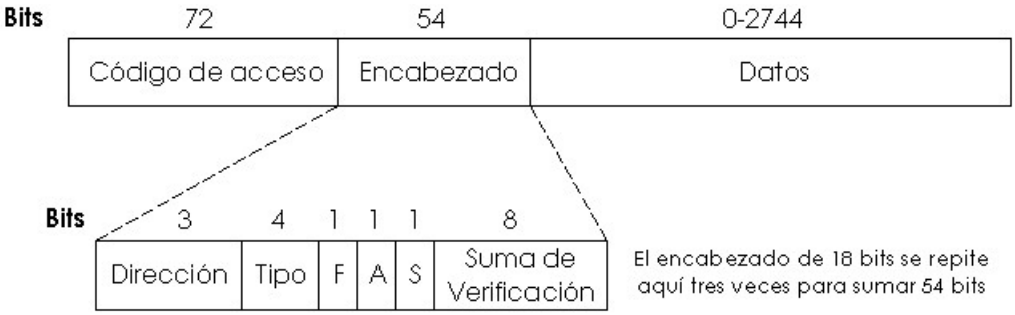

Figura II.6: Estructura de la trama de Bluetooth

Como se puede observar, las tramas comienzan con un código de acceso, el cual identifica al maestro de cada *piconet* y cuyo propósito es que los esclavos que se encuentren en el rango de alcance de dos maestros sepan qué tráfico está destinado a ellos.

A continuación se encuentra un encabezado de 54 bits que contiene campos comunes de la subcapa MAC. Y por último está el campo de datos, de hasta 2744 bits (para una transmisión de cinco ranuras). Para una sola ranura de tiempo, el formato es el mismo excepto que el campo de datos es de 240 bits.

Profundizando un poco en el encabezado, el campo *Dirección* identifica a cuál de los ocho dispositivos activos está destinada la trama. El campo *Tipo* indica el tipo de trama (ACL, SCO, de sondeo o nula), el tipo de corrección de errores que se utiliza en el campo de datos y cuántas ranuras de longitud tiene la trama.

Un esclavo establece el bit F (de flujo) cuando su búfer está lleno y no puede recibir más datos. Ésta es una forma primitiva de control de flujo. El bit A (de confirmación de recepción o *Acknowledgement*) se utiliza para incorporar un ACK en la trama. Por último, el bit S (de secuencia) sirve para numerar las tramas con el propósito de de detectar retransmisiones. El protocolo es de parada y espera, por lo que 1 bit es suficiente.

A continuación se encuentra el encabezado *Suma de verificación* de 8 bits. Todo el encabezado de 18 bits se repite tres veces para formar el encabezado de 54 bits. En el receptor se comprueban las tres copias de cada bit y si coinciden, se acepta. De lo contrario se impone la opinión de la mayoría.

II.3. Java y Bluetooth

La JSR82 [11, 27] define una API de alto nivel para la programación de dispositivos Bluetooth. Depende de la configuración CLDC de J2ME (paquete `javax.microedition.io`), y se divide en dos paquetes principales:

- Paquete `javax.bluetooth`: provee la funcionalidad para la realización de búsquedas de dispositivos, búsquedas de servicios y comunicación mediante flujos de datos (streams) o arrays de bytes.
- Paquete `javax.obex`: permite la comunicación mediante el protocolo OBEX; se trata de un protocolo de alto nivel muy similar a HTTP.

El objetivo de la especificación JSR-82 era definir una API estándar abierto, no propietario que pudiera ser usado en todos los dispositivos que implementen J2ME. Por consiguiente fue diseñado usando los APIs J2ME y el entorno de trabajo CLDC/MIDP.

El API intenta ofrecer las siguientes capacidades:

- Registro de servicios.
- Descubrimiento de dispositivos y servicios.
- Establecer conexiones RFCOMM, L2CAP y OBEX entre dispositivos.
- Usar dichas conexiones para mandar y recibir datos (las comunicaciones de voz no están soportadas).
- Manejar y controlar las conexiones de comunicación.
- Ofrecer seguridad a dichas actividades.

Y está orientado a dispositivos que cumplan las siguientes características:

- Al menos 512K de memoria libre (ROM y RAM)
- Conectividad a la red inalámbrica Bluetooth
- Que tengan una implementación del J2ME CLDC/MIDP

La estructura básica de una aplicación Bluetooth está dividida en cuatro partes: la inicialización de la pila, el descubrimiento de servicios, el manejo del dispositivo y la comunicación.

II.3.1. Inicialización de la pila

La pila Bluetooth es la responsable de controlar el dispositivo Bluetooth, por lo que es necesario inicializarla antes de hacer cualquier otra cosa. El proceso de inicialización consiste en un número de pasos cuyo propósito es dejar el dispositivo listo para la comunicación inalámbrica.

Desafortunadamente, la especificación deja la implementación de esta inicialización a los vendedores, y cada vendedor maneja la inicialización de una manera diferente. En un dispositivo puede haber una aplicación con un interfaz GUI, y en otra puede ser una serie de configuraciones que no pueden ser cambiados por el usuario.

II.3.2. Descubrimiento de dispositivos y servicios

Dado que los dispositivos inalámbricos son móviles, necesitan un mecanismo que permita encontrar, conectar, y obtener información sobre las características de dichos dispositivos.

Cualquier aplicación puede obtener una lista de dispositivos a los que es capaz de encontrar, usando, o bien `startInquiry()` (no bloqueante) o `retrieveDevices()` (bloqueante).

`startInquiry()` requiere que la aplicación tenga especificado un *listener*, el cual es notificado cuando un nuevo dispositivo es encontrado después de haber lanzado un proceso de búsqueda.

Por otra parte, si la aplicación no quiere esperar a descubrir dispositivos (o a ser descubierta por otro dispositivo) para comenzar, puede utilizar `retrieveDevices()`, que devuelve una lista de dispositivos encontrados en una búsqueda previa o bien unos que ya conozca por defecto.

Las interfaces que se usan en el descubrimiento de dispositivos son:

- `interface javax.bluetooth.DiscoveryListener`
- `interface javax.bluetooth.DiscoveryAgent`

La clase `DiscoveryAgent` provee de métodos para buscar servicios en un dispositivo servidor Bluetooth e iniciar transacciones entre el dispositivo y el servicio. Este API no da soporte para buscar servicios que estén ubicados en el propio dispositivo.

Para descubrir los servicios disponibles en un dispositivo servidor, el cliente primero debe recuperar un objeto que encapsule funcionalidad SDAP. Este objeto es del tipo `DiscoveryAgent`.

Las clases e interfaces que se usan en el descubrimiento de servicios son:

- `class javax.bluetooth.UUID`
- `class javax.bluetooth.DataElement`
- `class javax.bluetooth.DiscoveryAgent`
- `interface javax.bluetooth.ServiceRecord`
- `interface javax.bluetooth.DiscoveryListener`

En cuanto al registro de servicios, las responsabilidades de una aplicación servidora de Bluetooth son:

1. Crear un *Service Record* que describa el servicio ofrecido por la aplicación.
2. Añadir el *Service Record* al SDDB del servidor para avisar a los clientes potenciales de este servicio.
3. Registrar las medidas de seguridad Bluetooth asociadas a un servicio.
4. Aceptar conexiones de clientes que requieran el servicio ofrecido por la aplicación.
5. Actualizar el *Service Record* en el SDDB del servidor si las características del servicio cambian.
6. Quitar o deshabilitar el *Service Record* en el SDDB del servidor cuando el servicio no está disponible.

A las tareas 1,2,5 y 6 se las denominan registro del servicio (Service Registration), que comprenden unas tareas relacionadas con advertir al cliente de los servicios disponibles.

II.3.3. Manejo del dispositivo

Los dispositivos inalámbricos, son más vulnerables a ataques del tipo *spoofing* y *eavesdropping* que los demás dispositivos. Es por ello, que la tecnología Bluetooth incluye una serie de medidas para evitar estas vulnerabilidades, como es por ejemplo el salto de frecuencia; más aún, Bluetooth provee además de otros mecanismos opcionales como son la encriptación y autenticación.

Las clases que representan los objetos Bluetooth esenciales son:

- `class javax.bluetooth.LocalDevice`
- `class javax.bluetooth.RemoteDevice`

Las clases que dan soporte a la clase `LocalDevice` son:

- `class javax.bluetooth.BluetoothStateException extends java.io.IOException`
- `class javax.bluetooth.DeviceClass`

II.3.4. Comunicación

Para usar un servicio en un dispositivo Bluetooth remoto, el dispositivo local debe comunicarse usando el mismo protocolo que el servicio remoto. Los APIs permiten usar RFCOMM, L2CAP u OBEX como protocolo de nivel superior

El Generic Connection Framework (GFC) del CLDC provee la conexión base para la implementación de protocolos de comunicación. CLDC provee de los siguientes métodos para abrir una conexión:

- `Connection Connector.open(String name);`
- `Connection Connector.open(String name, int mode);`
- `Connection Connector.open(String name, int mode, boolean timeouts);`

La implementación debe soportar abrir una conexión con una conexión URL servidora o con una conexión URL cliente, con el modo por defecto `READ_WRITE`.

II.4. Librerías de Bouncy Castle

Las librerías de Bouncy Castle [54] son unas librerías Java, de código libre, que ofrecen funcionalidades criptográficas. Nosotros utilizaremos las librerías de Bouncy Castle J2ME en este escenario y serán el equivalente a las librerías de OpenSSL que utilizábamos en el primer escenario y que ofrecían operaciones criptográficas para poder construir el módulo PKCS#11.

En este caso no crearemos tal módulo en J2ME (aunque podría hacerse, sería demasiado ambicioso para este proyecto), sino que utilizaremos las librerías criptográficas desarrolladas en FaCTo que se basan en estas librerías de Bouncy Castle.

Diseño

II.5. Introducción

Tras analizar todas las tecnologías y protocolos que participan en este escenario, se ha llegado a la conclusión de que en el diseño del mismo se planteará desarrollar una librería J2ME de comunicación Bluetooth que haga uso de la API JSR82.

Por lo tanto, ya no serán necesarias las librerías nativas del sistema operativo, ni tampoco una máquina virtual que soporte JNI, por lo que nos bastará con la máquina virtual que incorpore el dispositivo, siempre que ésta implemente la API JSR82 y consecuentemente, la arquitectura J2ME CLDC/MIDP.

Sobre dicha librería de comunicación Bluetooth se desarrollarán aplicaciones criptográficas J2ME que estarán basadas en las librerías criptográficas desarrolladas en FaCTo, construidas a partir de las librerías de Bouncy Castle, ya que éstas librerías J2ME de FaCTo, además de estar ya desarrolladas y poder reutilizarlas sin problemas, proporcionan toda la funcionalidad criptográfica que para este escenario se ha pensado.

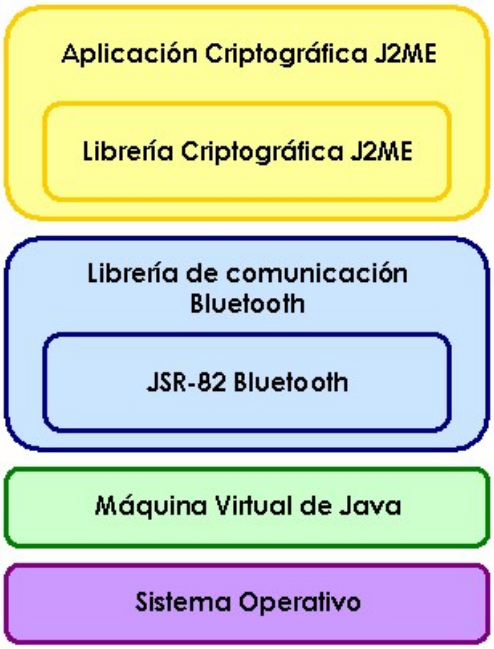

Figura II.7: Diseño del segundo escenario

En la figura II.7 se observa la arquitectura por capas que supondrá la solución planteada para el segundo escenario de integración de tarjetas criptográficas en dispositivos móviles.

II.5.1. Hardware

En cuanto al hardware empleado en el diseño de este escenario, se utilizarán, la PDA Acer N50, el lector USB de tarjetas GemPC Twin, la tarjeta criptográfica y el lector Bluetooth de tarjetas CASPAD C100 que se describen y se detallan en el apéndice D.

II.6. Máquina Virtual de Java

El primer elemento de nuestra arquitectura por encima del sistema operativo es la máquina virtual de Java. Para que este escenario se pueda llevar a cabo en un dispositivo móvil real, éste debe implementar, como mínimo, la JSR82 y, por consiguiente, el perfil MIDP 2.0.

Por lo tanto, la máquina virtual de Java que se empleará será la que el fabricante haya incluido en su dispositivo.

II.7. Librería de comunicación Bluetooth J2ME

En la sección II.3 (página 70) se realizó un breve análisis de la API de Java para Bluetooth JSR82. A pesar de tratarse de una API de alto nivel, se ha pensado que sería conveniente desarrollar otra librería J2ME de comunicación Bluetooth que, haciendo uso de la API JSR82, ofrezca unas capacidades y unas facilidades al programador que use dicha librería tales como:

- Conectar y desconectar con la tarjeta criptográfica.
- Recuperar las claves y certificados almacenados en la tarjeta de forma intuitiva, fácil y segura.
- Almacenar nuevas claves y certificados en la tarjeta de forma intuitiva, fácil y segura.
- Ordenar a la tarjeta que realice cualquier operación criptográfica que ésta soporte.
- Recuperar información acerca de la tarjeta como el número de serie, o los algoritmos criptográficos soportados.

Todas estas operaciones deben tener un nivel de abstracción tal que eviten a aquellos desarrolladores que las empleen tener que conocer los detalles de bajo nivel tales como las APDUs que se deban enviar a la tarjeta, o la gestión que el sistema operativo internamente haga de la comunicación Bluetooth.

II.8. Librería criptográfica J2ME

Para este diseño se ha decidido reutilizar la librería criptográfica J2ME que actualmente ya se encuentra desarrollada dentro del proyecto FaCTo y que se llama `cryptomidp`.

La librería `cryptomidp` está basada en las librerías criptográficas Bouncy Castle. Desde la API de criptografía ligera se obtienen los algoritmos básicos de firma y cifrado. A partir de ahí, se aumenta la funcionalidad, creando una serie de clases que encapsulan dichos algoritmos y crean un nivel de abstracción mayor para poder trabajar con certificados, claves y algoritmos.

La librería de criptografía ligera no dispone de funcionalidad para tratar con archivos PKCS#7 ni PKCS#12, por lo que se optó por adaptar el código existente en las versiones de J2SE a J2ME.

Además, se realizaron pequeñas mejoras en el código original de la API de criptografía ligera, como el aumento de la velocidad del algoritmo SHA-1, usado en la clase que maneja archivos PKCS#12, y la limpieza de algunas clases.

El trabajo en este proyecto consistiría en hacer que dicha librería criptográfica J2ME usara la librería de comunicación Bluetooth previamente desarrollada para cualquier operación relacionada con la comunicación con la tarjeta inteligente.

Por ejemplo, ahora mismo dicha librería recupera los certificados digitales directamente almacenados en el dispositivo móvil. La tarea consistiría en este caso en conseguir que dichos certificados los tomara de la tarjeta criptográfica.

II.9. Aplicaciones criptográficas J2ME

Todas las aplicaciones criptográficas J2ME que se hubieran desarrollado para el primer escenario serían igualmente válidas para éste, después de modificarlas convenientemente para que se ajusten a la arquitectura CLDC/MIDP.

Además, su adaptación para que usaran la librería criptográfica desarrollada específicamente para este escenario no debería ser demasiado complicada si en el desarrollo de dichas aplicaciones se siguieron buenos patrones de diseño como el modelo Vista-Controlador.

II.10. Ventajas e inconvenientes

A continuación se describen las principales ventajas e inconvenientes que plantea este diseño concreto de integración de tarjetas criptográficas en dispositivos móviles.

II.10.1. Ventajas

Con este escenario sí que se alcanza realmente una independencia total del sistema operativo subyacente en el dispositivo móvil. Si las librerías criptográficas y las aplicaciones J2ME se desarrollan convenientemente, ambas serían plenamente compatibles en cualquier plataforma que tuviera instalada una máquina virtual de Java apropiada; es decir, una máquina virtual que implementara la API JSR82 (así como J2ME CLDC/MIDP) y que corriera sobre la plataforma en cuestión.

Además, se sigue manteniendo la seguridad conseguida en el primer escenario porque, aunque las claves y los certificados puedan "salir" de la tarjeta criptográfica y "viajar" a través de una comunicación inalámbrica (siempre que la tarjeta no sea criptográfica), dicha comunicación puede securizarse como se comentó en la sección II.3.3 con mecanismos de autenticación y cifrado.

Sería de necios obviar que hoy en día existen muchos (muchísimos) más dispositivos móviles que soportan Bluetooth (PDAs, teléfonos móviles, *smartphones*), que los que incorporan una ranura de comunicación SDIO, por lo que esta solución sería fácilmente desplegable en un mayor número de dispositivos.

II.10.2. Inconvenientes

Al tratarse de una comunicación inalámbrica, ésta es siempre más propicia a las interferencias que una comunicación por cable (véase sección II.2.4 de la página 68).

Aunque el dispositivo móvil incluya comunicación vía Bluetooth, es necesario que el fabricante del mismo implemente el JSR82 (así como J2ME CLDC/MIDP); de lo contrario, todo este diseño no tendrá validez alguna sobre tal dispositivo.

En este escenario, a diferencia del anterior, no se sigue ningún estándar de acceso al token, por lo que potencialmente podrían producirse incompatibilidades al cambiar de token criptográfico.

Otro inconveniente de usar un lector de tarjetas Bluetooth se refiere a la necesidad del mismo de recargar sus baterías periódicamente. Mientras que el suministro de corriente de un lector de tarjetas SDIO viene del propio dispositivo al que se encuentre conectado, con un lector de tarjetas Bluetooth su autonomía depende de la batería que lleve.

Pero es que el propio hecho de tener que utilizar un lector de tarjetas externo al dispositivo móvil ya supone un inconveniente en sí, puesto que, no sólo hay que adquirir dicho lector (con su gasto económico asociado de unos 1000 euros en la actualidad), sino que en ocasiones, llevarlo encima puede resultar ser un engorro.

Escenario III

Tarjeta Interna WIM

Visión General

Figura III.1: Tercer escenario: tarjeta interna

Por último, en este tercer escenario disponemos de una tarjeta interna SIM con funcionalidad WIM y un *Smartphone* (una PDA con funcionalidad telefónica) o un teléfono móvil.

Para llevar a cabo una solución basada en un escenario en el que la tarjeta criptográfica se encuentra en el interior del dispositivo móvil, en primer lugar, en la fase de **análisis** se verá una breve introducción al lenguaje J2ME, un estudio de las diferentes máquinas virtuales de Java para dispositivos móviles, la API de Java SATSA y el estándar PKCS#15 [12].

Sin embargo, todo el análisis referente a J2ME así como de las máquinas virtuales de Java para dispositivos móviles que se realizó en el primer escenario es igualmente aplicable a este nuevo escenario. Por lo tanto, no se repetirá dicho análisis, sino que se remitirá al lector a los apartados en los que ya se hizo dicho estudio.

En la fase de **diseño**, se mostrará la arquitectura en capas del escenario en cuestión y se expondrán los componentes que deberán ser desarrollados en la siguiente fase de implementación.

Dichos componentes serán, básicamente: una máquina virtual de java adecuada para dispositivos móviles, una librería de comunicación en Java que haga uso de la API de Java SATSA, una librería criptográfica J2ME y una aplicación J2ME similar a las desarrolladas para el primer escenario que, haciendo uso tanto de la librería de comunicación, como de la librería de comunicación con la tarjeta, acceda a la misma y sea capaz de realizar algunas operaciones criptográficas como las que se plantearon en el primer escenario.

Análisis

III.1. Breve introducción a J2ME

Todo el análisis realizado en la sección I.1 de la página 21 es igualmente válido para este escenario. Para más información, consúltese dicho apartado.

III.2. PKCS#15/WIM

Un módulo de identidad WAP (WIM [13], *WAP Identity Module*) no es otra cosa que un módulo que realiza operaciones de seguridad en la capa de aplicación y, principalmente, almacena y procesa la información necesaria para la identificación y autenticación de un usuario en un entorno inalámbrico.

Una posible implementación de un módulo WIM puede ser una *smart card* o el propio módulo SIM que incorporan los teléfonos móviles y *smartphones*, que será lo que nosotros tendremos en este escenario.

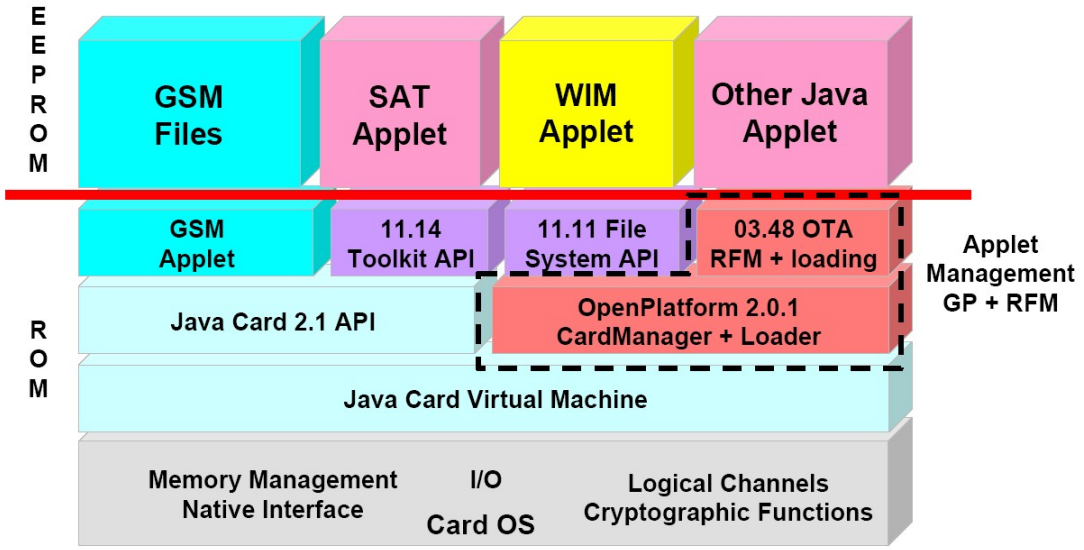

Figura III.2: Arquitectura SIM JavaCard

La información almacenada en el módulo de la que hablamos son simple y llanamente certificados digitales y claves públicas y privadas asociadas a la identidad de un usuario concreto. Dicha información se almacena siguiendo un formato específico detallado en el estándar PKCS#15, que a continuación describimos.

III.2.1. Introducción

PKCS#15 es un estándar que define la sintaxis de la información almacenada en tokens criptográficos. Dicha sintaxis o formato de almacenamiento tiene las siguientes características:

- Su estructura dinámica permite la implementación en una amplia variedad de medios, incluyendo los valores almacenados en tarjetas.
- Permite que múltiples aplicaciones se puedan almacenar simultáneamente en la tarjeta.
- Soporta el almacenamiento de cualquier tipo de objeto (claves, certificados y datos).
- Soporta múltiples PINs, siempre que el token también los soporte.

La información PKCS#15 del token debería ser leída cuando un token se presentara conteniendo dicha información, la cual es usada por un intérprete PKCS#15 que es parte del entorno software.

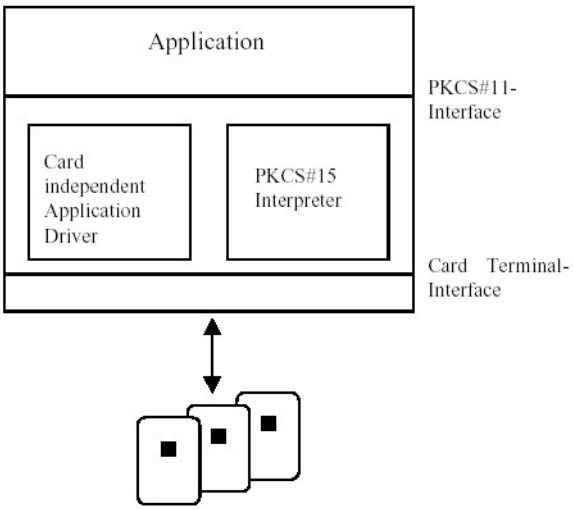

Figura III.3: Ejemplo de integración de un intérprete PKCS#15

III.2.2. Visión General

En PKCS#15 existen cuatro clases generales de objetos: claves, certificados, objetos de autenticación y objetos de datos. Cada una de estas clases tiene, a su vez, subclases tales como claves privadas, públicas y secretas cuyas instanciaciones son los objetos que realmente se almacenan en las tarjetas.

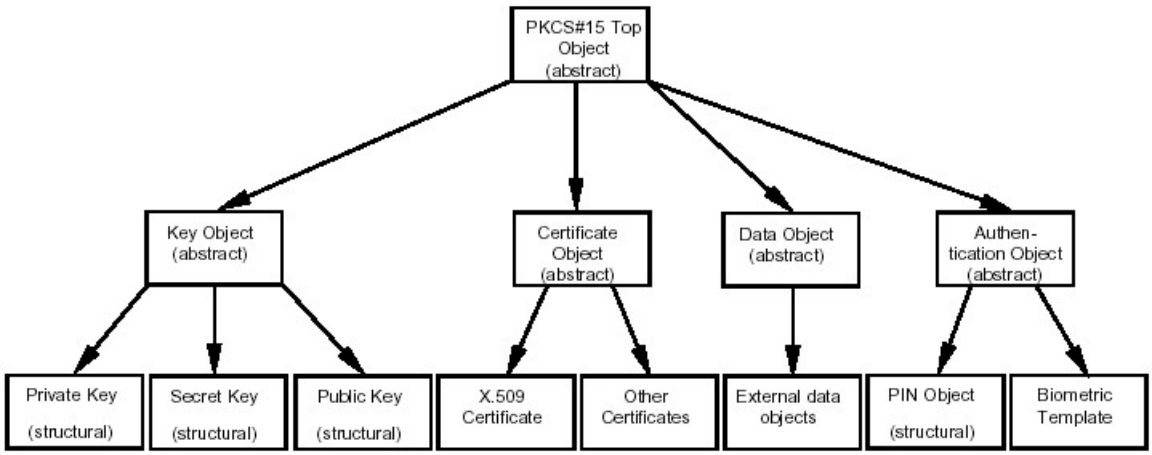

Figura III.4: Jerarquía de Objetos de PKCS#15

III.2.3. Formato de archivo de las tarjetas IC

En general, un formato de ficheros para tarjetas IC especifica cómo representar elementos concretos abstractos y de alto nivel, tales como claves y certificados, en términos de elementos de bajo nivel, tales como ficheros de la tarjeta IC y estructuras de directorios.

El formato también puede sugerir cómo y bajo qué circunstancias se puede acceder a estos elementos de alto nivel desde una fuente externa, e incluso cómo implementar dichas reglas de acceso.

Así una tarjeta típica que soporte esta especificación tendrá una estructura de ficheros como la que se muestra en la figura III.5.

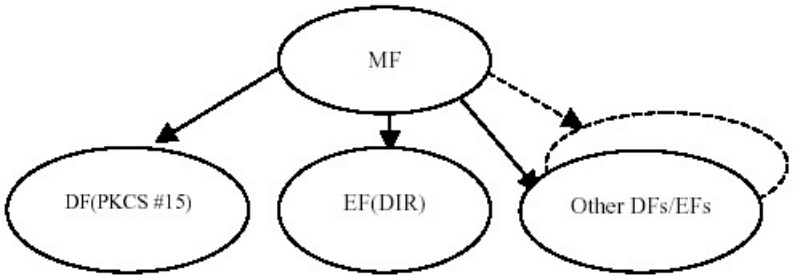

Figura III.5: Contenido típico de una tarjeta PKCS#15

Por su parte, los contenidos del directorio de PKCS#15 `DF(PKCS#15)` son de alguna manera dependientes del tipo de tarjeta IC y del uso que a ésta se le vaya a dar. Sin embargo, en la figura III.6 se muestra la estructura que se espera sea la más común.

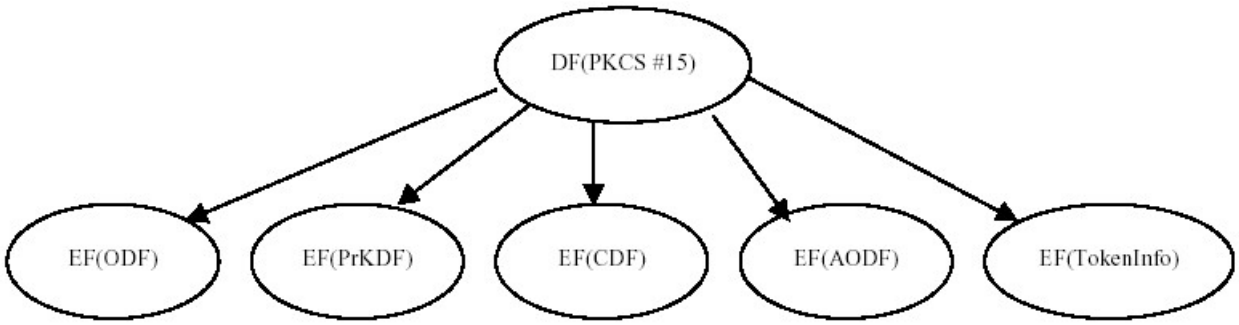

Figura III.6: Contenido típico del `DF(PKCS#15)`

La explicación detallada del `EF(DIR)` que cuelga del `MF`, así como de los EFs `EF(ODF)`, `EF(PrKDF)`, `EF(CDF)`, `EF(AODF)` y `EF(TokenInfo)`, que cuelgan del `DF(PKCS#15)`, no conciernen a este proyecto.

Para obtener más información sobre los mismos, consúltese la especificación del estándar PKCS#15 v1.1 [12] desarrollada por RSA Laboratories. En ella se puede encontrar también, entre otras muchas cosas, la sintaxis de ASN.1 utilizada para almacenar la información en las tarjetas.

III.3. SATSA

SATSA [14, 28], actualmente en versión 1.0, es una API de Java que establece la posibilidad de abrir un canal de comunicación entre un MIDlet y un "Security Element" (tarjeta SIM). De esta forma la tarjeta SIM realiza operaciones de seguridad (firma y autenticación de usuarios) en aplicaciones J2ME.

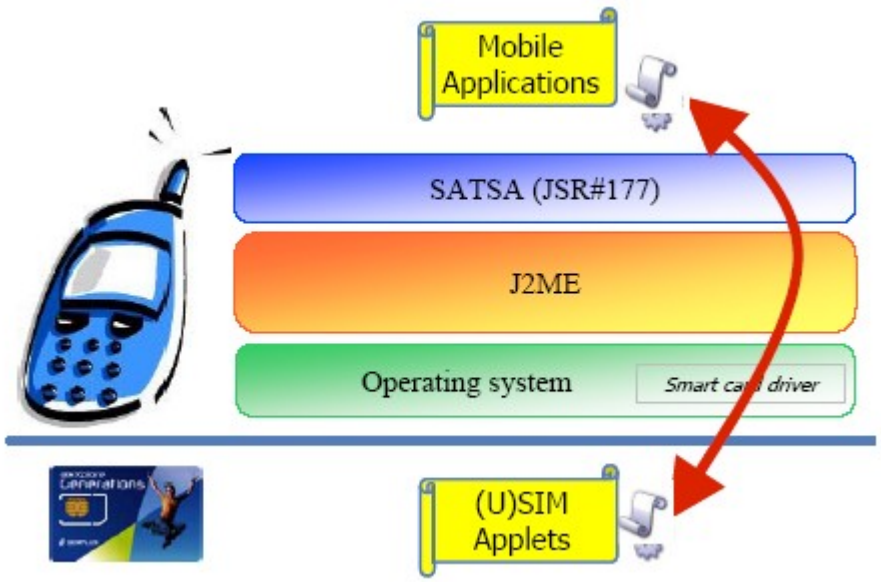

Figura III.7: Arquitectura de SATSA

SATSA define 3 formas diferentes de comunicación, en función del tipo de aplicación SIM con la que se desee interactuar, basadas en el uso de distintos paquetes:

- APDU. Para enviar comandos APDU a applets JavaCard o USAT, a través del protocolo ISO 7816-4.
- JCRMI. Para instanciar objetos JavaCard RMI.
- PKI. Para utilizar el applet WIM de la tarjeta SIM en operaciones de solicitud de certificado, autenticación de usuarios y firma digital (pero no en operaciones de verificación de firma). Éste será el paquete que nosotros utilicemos en este escenario.

SATSA especifica también la existencia de un cuarto paquete CRYPTO con utilidades criptográficas para facilitar las implementaciones de seguridad de los MIDlets instalados en el dispositivo móvil.

III.3.1. Paquete opcional APDU

El paquete opcional APDU se encuentra en el paquete `javax.microedition.apdu` y permite a las aplicaciones J2ME comunicarse con las aplicaciones de la tarjeta inteligente. Para ello se debe crear una `APDUConnection` que será la responsable de intercambiar las APDUs codificadas en el formato ISO7816-4.

Cada `APDUConnection` cuenta con un canal lógico reservado exclusivamente para ella, cuya gestión es manejada por la implementación de la API, la cual solicita a la tarjeta inteligente que proporcione un canal lógico libre.

Es posible crear más de una `APDUConnection` con el fin de comunicarse simultáneamente con las aplicaciones de la tarjeta inteligente.

Sin embargo, una `APDUConnection` no soporta ni las sesiones ni los comandos proactivos, por lo que es la aplicación J2ME la que debe responsabilizarse de que no se cree una sesión se este tipo.

III.3.2. Paquete opcional JCRMI

El paquete JCRMI permite la comunicación con una tarjeta inteligente haciendo uso del protocolo del mismo nombre. Dicho protocolo permite que las aplicaciones que no están almacenadas en la tarjeta utilicen objetos que sí lo están.

Dichas aplicaciones usan una interfaz remota para trabajar con los objetos presentes en la tarjeta. Así, cuando una aplicación llama a un método en la interfaz remota, la llamada a dicho método es enviada a la tarjeta a través del protocolo JCRMI. El método entonces se ejecuta en la tarjeta y los resultados se devuelven a la aplicación.

Cada aplicación que quiera hacer uso de este paquete debe abrir una `JavaCardRMIConnection` para acceder al contenido remoto de la tarjeta. Y cada `JavaCardRMIConnection` lleva asociado un canal lógico exclusivo para ella, cuya gestión es idéntica a la del paquete APDU.

También aquí se puede crear más de una `JavaCardRMIConnection` con el fin de comunicarse simultáneamente con los applets de Java Card.

III.3.3. Paquete opcional PKI

Este paquete que sólo incluye las clases `javax.microedition.pki.UserCredentialManager` y `javax.microedition.securityservice.CMSMessageSignatureService`, permite a las aplicaciones J2ME solicitar a la tarjeta inteligente operaciones de firma digital, y también incluye algunos métodos de gestión básica de certificados.

Una aplicación J2ME utiliza el `CMSMessageSignatureService` para firmar mensajes con una clave privada almacenada en un módulo WIM/PKCS#15. Dichos mensajes pueden ser firmados con el objetivo de conseguir autenticación, o bien, no repudio.

La autorización para utilizar una clave concreta del módulo WIM vendrá dada por la política de seguridad de dicho módulo; por ejemplo, puede requerirse que se inserte el PIN.

Por otra parte, una aplicación utiliza el `UserCredentialManager` para llevar a cabo las siguientes tareas:

- Formular una petición de registro de certificado, que puede ser enviada a una autoridad de registro de certificados.
- Añadir un certificado, o la URI del mismo a un almacén de certificados.
- Eliminar un certificado, o la URI del mismo de un almacén de certificados.

Este paquete es el que sin duda mejor se adapta a nuestras necesidades y es por ello que será el que usemos en nuestro escenario.

III.3.4. Paquete opcional CRYPTO

El paquete opcional CRYPTO provee a las aplicaciones J2ME de una serie de herramientas criptográficas que les permiten realizar operaciones como resumen digital, firma digital o cifrado. Este paquete es un subconjunto de la JCE (*Java Cryptography Extension).*

Así, todas las clases e interfaces de este paquete se encuentran en los mismos paquetes que en la JCE de la plataforma J2SE: `java.security`, `java.security.spec`, `javax.crypto` y `javax.crypto.spec`.

Este paquete, sin embargo, no implica a la tarjeta WIM en las operaciones criptográficas más que para extraer de ella las claves y certificados digitales necesarios para llevar a cabo dichas operaciones. Por este mismo motivo, se desaconseja usar este paquete si se van a manejar datos sensibles y/o valiosos almacenados en la tarjeta WIM (como puede ser la clave privada del usuario).

Diseño

III.4. Introducción

Después de haber analizado todas las tecnologías y estándares necesarios para este escenario, se ha llegado a la conclusión de que en el diseño que supondrá la solución para este escenario deberá existir una aplicación criptográfica J2ME que, haciendo uso de la API JSR177, o SATSA, acceda al módulo WIM y realice operaciones criptográficas tales como firma digital.

El motivo por el que se decidió usar SATSA-PKI fue que dicha API J2ME es la única de las que conocemos que nos permite acceder a una tarjeta WIM y solicitarle que realice operaciones criptográficas. Existían otras alternativas J2ME, como por ejemplo usar SATSA-APDU, e incluso no J2ME, como USAT-i, pero dichas alternativas no se ajustaban tan bien como la que finalmente escogimos.

Figura III.8: Diseño del tercer escenario

En la figura III.8 se muestra la arquitectura por capas que constituye el diseño que se ha decidido desarrollar para este escenario en el que la tarjeta criptográfica se encuentra dentro del dispositivo.

III.5. Máquina Virtual Java

El primer elemento de nuestra arquitectura por encima del sistema operativo es la máquina virtual de Java. Para que este escenario se pueda llevar a cabo en un dispositivo móvil real, éste debe implementar, como mínimo, la JSR177 y, por consiguiente, el perfil MIDP 2.0.

Por lo tanto, la máquina virtual de Java que se empleará será la que el fabricante haya incluido en su dispositivo.

III.6. API de acceso SATSA

Como hemos visto anteriormente, de los cuatro paquetes opcionales que ofrece SATSA, el que mejor se ajusta nuestras necesidades es el de SATSA-PKI. Éste es el único de los cuatro que accede a un módulo con funcionalidad WIM/PKCS#15 y realiza peticiones al módulo SIM para que éste lleve a cabo operaciones criptográficas.

Además, con este paquete SATSA-PKI es posible añadir y eliminar certificados de un repositorio de certificados, así como solicitar un nuevo certificado a una autoridad de registro.

Esta API supone, por lo tanto, el marco de trabajo idóneo para nuestro tercer escenario de integración de tarjetas criptográficas en dispositivos móviles.

III.7. Librería criptográfica J2ME

Al igual que se planteó en el segundo escenario, las librerías criptográficas desarrolladas como parte del proyecto FaCTo pueden ser aplicadas aquí, tras realizar sobre ellas las pertinentes adaptaciones que permitan que dichas librerías usen la API SATSA en vez de tomar los certificados y claves directamente del repositorio del dispositivo móvil.

III.8. Aplicaciones criptográficas J2ME

También aquí es válido todo lo expuesto en la sección homónima del escenario II. Es decir, todas las aplicaciones criptográficas J2ME que se hubieran desarrollado en el primer escenario podrían servir para éste después de adaptarlas a la arquitectura J2ME CLDC/MIDP.

III.9. Ventajas e inconvenientes

III.9.1. Ventajas

Este es el primer escenario en el que es requisito indispensable que la tarjeta criptográfica tenga funcionalidad WIM/PKCS#15 y por lo tanto es el primero en el que se asegura que toda la información confidencial sensible tal como la clave privada del usuario nunca sale de la tarjeta, lo cual supone un aumento cualitativo de la seguridad de este sistema frente a sus predecesores.

En este escenario que, por cierto, sigue siendo independiente del sistema operativo subyacente, no es necesaria la adquisición de ningún lector de tarjetas externo al dispositivo móvil. Es más, ni siquiera sería necesario cambiar de dispositivo móvil (si éste ya soporta el JSR177, esto es, SATSA), sino que sería suficiente con cambiar el actual módulo SIM que llevara nuestro dispositivo por otro módulo SIM que tuviera funcionalidad WIM (en caso de que no la tuviera ya).

III.9.2. Inconvenientes

Al tratarse de una propuesta totalmente novedosa, el trabajo previo realizado en este campo es prácticamente nulo, por lo que casi no existen ejemplos o documentación que específicamente aborde el problema aquí planteado.

Es evidente que al encontrarse la tarjeta criptográfica en el interior del dispositivo móvil, este escenario de integración no es aplicable al nuevo DNI-e.

Además, en principio, este escenario no abarcaría tantos dispositivos móviles como lo hacía, por ejemplo, el segundo escenario, ya que en este escenario se requiere que el dispositivo móvil tenga funcionalidad telefónica (es decir, que lleve un módulo SIM) y las PDAs no cumplen esta restricción, ya que si llevaran un módulo SIM no serían PDAs, sino *Smartphones*.

Conclusiones y trabajo futuro

1. Conclusiones

En este proyecto se han descrito y diseñado tres escenarios posibles de integración de tarjetas criptográficas en dispositivos móviles usando en todos ellos J2ME. Para cada uno de ellos se ha realizado un profundo análisis de todas las tecnologías, estándares y protocolos presentes en los mismos, así como un detallado diseño de la solución que nosotros hemos planteado.

Sin embargo, sólo el primero de los escenarios ha contado además con una fase de implementación pormenorizada de su correspondiente diseño.

En cada escenario se han expuesto sus ventajas e inconvenientes. No obstante, la tabla 1 muestra, a modo de resumen comparativo, dichas ventajas e inconvenientes para cada uno de los tres escenarios estudiados.

	Escenario I	**Escenario II**	**Escenario III**
Independiente del Sistema Operativo	No	Sí	Sí
Interfaz de comunicación	Ranura SDIO	Bluetooth	Comunicación interna
Lector de tarjetas con baterías	No	Sí	No
Grado de interferencias en la comunicación con la tarjeta	Despreciable	Medio	Despreciable
API de acceso a la tarjeta	IAIK	JSR82	SATSA
Estándares criptográficos empleados	PKCS#11	Ninguno	PKCS#15/WIM
Requisitos "especiales" de la JVM	JNI	JSR82	JSR177/SATSA
Arquitectura J2ME	CDC/PP	CLDC/MIDP	CLDC/MIDP
Requisitos de la SIM	Ninguno	Ninguno	WIM/PKCS#15
Aplicable al DNI-e	Sí	Sí	No
Dispositivos móviles a los que se aplica	PDAs y *Smartphones*	PDAs, *Smartphones* y teléfonos móviles	*Smartphones* y teléfonos móviles
Seguridad del sistema	Alta	Alta	Muy Alta
Dificultad de desarrollo	Alta	Media	Media-Alta
Trabajo previo	Suficiente	Escaso	Muy escaso
Coste económico	Medio-Alto	Alto	Medio
Nivel de penetración en el mercado	Medio-Bajo	Medio-Alto	Alto

Tabla 1: Resumen de las principales características de cada escenario

Como vimos, el empleo de JNI para acceder a la librería dll del primer escenario hacía que éste fuera **dependiente del sistema operativo** subyacente. Dicha dependencia fue eliminada en los escenarios II y III.

Una desventaja del segundo escenario frente a los otros dos es que el lector de tarjetas Bluetooth se alimenta de corriente a través de una **batería** que, en el caso del CASPAD C100 no tiene mucho más de 4 horas de autonomía, por lo que si se tuviera la mala suerte de tener la batería descargada en un momento importante, sería imposible que el lector funcionara.

Otro inconveniente de este segundo escenario está implícito en la propia comunicación del dispositivo móvil con la tarjeta. Puesto que dicha comunicación tiene lugar vía inalámbrica, es más propensa a las **interferencias** del medio que la comunicación que entre el dispositivo móvil y la tarjeta tiene lugar tanto en el primer como en el tercer escenario.

El empleo de **estándares** ampliamente reconocidos y aceptados por la comunidad científica asegura la interoperabilidad de los productos desarrollados independientemente del fabricante del dispositivo.

En este sentido, en el primer escenario la solución planteada se ajusta perfectamente al estándar PKCS#11; el tercer escenario hace lo propio con el estándar PKCS#15; sin embargo, el segundo escenario no tiene que ajustarse a ningún estándar criptográfico como requisito de diseño. El empleo o no de dichos estándares quedará en manos de los desarrolladores de aplicaciones criptográficas J2ME que quieran usar ese segundo escenario.

El primer escenario es el único de los tres que realmente necesita una característica muy específica de la máquina virtual que se incluya en el mismo. Dicha característica es el soporte de **JNI**. Este requisito obliga necesariamente a utilizar una JVM con más capacidades que la que el propio fabricante incorpora en el dispositivo lo cual, por otra parte, permite utilizar la arquitectura CDC/PP de J2ME.

Como vimos en la presentación de este proyecto, todos los esfuerzos que vayan orientados a desarrollar herramientas que faciliten la integración del nuevo **DNI-e**, son de gran utilidad.

En este sentido podemos decir que, evidentemente, en el tercer escenario es el único en el que, por sus características especiales (la tarjeta criptográfica es el propio módulo SIM integrado en el dispositivo móvil), no se puede utilizar el DNI-e como tarjeta criptográfica.

Como también vimos en su momento, el primer escenario sólo es aplicable en aquellos dispositivos móviles que cuenten con una ranura **SDIO**, lo cual excluye a un gran número de teléfonos móviles. El segundo escenario, sin embargo, se puede aplicar en cualquier dispositivo móvil que tenga conectividad Bluetooth y que implemente la **JSR82**. Y el tercer escenario, por su parte, es aplicable en aquellos dispositivos móviles que tengan **funcionalidad telefónica** (lo que descarta a las PDAs) e implementen la **JSR177**.

El sistema criptográficamente **más seguro** de todos es el planteado en el tercer escenario, ya que en éste la tarjeta criptográfica debe tener funcionalidad WIM/PKCS#15, y por tanto, toda la información criptográfica sensible y valiosa (tal como la clave privada del usuario) jamás sale fuera del módulo WIM. Es más, es el propio módulo quien realiza las operaciones criptográficas y no la aplicación criptográfica J2ME instalada en el dispositivo móvil.

También es innegable que el escenario que supone un mayor **esfuerzo de desarrollo** es el primero de los tres. En él, además de tener que adaptar el módulo PKCS#11 para PC a un entorno Pocket PC (lo cual ya es de por sí una tarea ardua), es necesario instalar satisfactoriamente una máquina virtual de Java en el dispositivo móvil que soporte JNI, así como desarrollar aplicaciones J2ME CDC/PP que se ejecuten sin errores en dicho dispositivo.

Un inconveniente a tener en cuenta de los escenarios II y III es el escaso **trabajo previo** que en cada uno de ellos existe. Este hecho se acentúa quizás más en el tercer escenario, donde no existen ni siquiera ejemplos de algo parecido a lo que aquí se pretende lograr.

Por último, pero no por ello menos importante, destacar el **coste económico** de cada escenario. En este sentido, hoy en día el precio de un lector de tarjetas Bluetooth ronda los 1000 euros, mientras que el de un lector de tarjetas SDIO está en torno a los 200/300 euros.

A lo largo de todo el proyecto se ha observado cómo cada escenario subsanaba alguna deficiencia del anterior y lo mejoraba en algún otro aspecto.

Así, en el escenario I se introducía la primera integración de tarjetas criptográficas en dispositivos móviles. Además, se hacía uso del estándar PKCS#11 y se empleaba también por vez primera el lenguaje J2ME en el diseño de una solución de este tipo.

Con el segundo escenario se adquiere realmente la independencia del sistema operativo subyacente en el dispositivo móvil. Dicha independencia, que "debería" ser una consecuencia de utilizar Java, junto con el innegable hecho de que la mayoría de dispositivos móviles soportan Bluetooth, hacen de éste un escenario ampliamente proyectable sobre multitud de dispositivos.

Por último, en el tercer escenario, además de preservarse la independencia del sistema operativo subyacente, se aportan otras ventajas. Al tratarse de una tarjeta interna, no es necesaria la adquisición de ningún tipo de lector de tarjetas, con el consecuente gasto económico.

Además, como dicha tarjeta debe tener necesariamente capacidades criptográficas, este escenario es el único de los tres en el que realmente toda la información criptográfica permanece siempre almacenada en la tarjeta, lo cual supone un aumento cualitativo de la seguridad del sistema.

2. Vías futuras

El panorama de los dispositivos móviles es un ámbito en pleno auge y crecimiento que precisa de desarrollos que los acerquen cada vez más a los usuarios de a pie, y que faciliten su manejo por parte de estos ciudadanos.

En este proyecto se ha abarcado y nos hemos centrado en un área del amplio abanico de posibilidades que estos terminales nos ofrecen a los investigadores, siendo éste un marco de trabajo que ahora comienza a ser seriamente considerado por sus numerosos beneficios y ventajas. Dichos beneficios y ventajas repercuten directa e indirectamente en toda la sociedad en la medida que ésta se orienta cada vez más hacia lo que se ha venido a denominar la "sociedad de la información".

Pero como decimos, aún queda mucho trabajo por hacer en este campo tecnológico. De este modo, algunas de las vías futuras de investigación que este proyecto ha podido abrir son:

- Mejorar y ampliar la implementación del primer escenario, creando, por ejemplo, nuevas aplicaciones criptográficas J2ME que hagan uso de la librería criptográfica desarrollada, o ampliando las funcionalidades de ésta última.
- Estudiar nuevas alternativas de integración de tarjetas criptográficas en dispositivos móviles que se puedan plantear en el futuro. Algunas de ellas podrían ser:
  - Utilizar USAT-i para interactuar con aquellas tarjetas SIM que posean funcionalidad WIM/PKCS#15 mediante el envío de comandos proactivos a la misma. De esta manera, no sería necesario instalar ningún software en el dispositivo, sino que bastaría con que el módulo SIM contuviera un intérprete USAT-i.

    Con esta solución se conseguiría, además de independencia del sistema operativo, independencia del dispositivo móvil que se estuviera utilizando.
  - Emplear una Mega SIM como tarjeta criptográfica. Dicha Mega SIM no es más que un módulo SIM con capacidad de hasta 1 GB.
- Intensificar un poco más el análisis referido a la parte servidora, ya que en este proyecto nos hemos centrado más en la parte del cliente, analizando algunos servicios de firma móvil o la especificación MSSP del ETSI.

Apéndice A

CLDC y MIDP

A.1. API CLDC

Clases	Interfaces	Excepciones	Errores
Boolean	Runnable	ArithmeticException	Error
Byte		ArrayIndexOutOfBoundsException	OutOfMemoryError
Character		ArrayStoreException	VirtualMachineError
Class		ClassCastException	
Integer		ClassNotFoundException	
Long		Exception	
Math		IllegalAccessException	
Object		IllegalArgumentException	
Runtime		IllegalMonitorStateException	
Short		IllegalThreadStateException	
String		IndexOutOfBoundsException	
StringBuffer		InstantiationException	
System		InterruptedException	
Thread		NegativeArraySizeException	
Throwable		NullPointerException	
		NumberFormatException	
		RuntimeException	
		SecurityException	
		StringIndexOutOfBoundsException	

Tabla A.1: Paquete `java.lang`

Clases	Interfaces	Excepciones	Errores
Calendar	Enumeration	EmptyStackException	
Date		NoSuchElementException	
Hashtable			
Random			
Stack			
TimeZone			
Vector			

Tabla A.2: Paquete `java.util`

Clases	Interfaces	Excepciones	Errores
ByteArrayInputStream	DataInput	EOFException	
ByteArrayOutputStream	DataOutput	InterruptedIOException	
DataInputStream		IOException	
DataOutputStream		UnsupportedEncodingException	
InputStream		UTFDataFormartException	
InputStreamReader			
OutputStream			
OutputStreamWriter			
PrintStream			
Reader			
Writer			

Tabla A.3: Paquete `java.io`

Clases	Interfaces	Excepciones	Errores
Connector	ContentConnection	ConnectionNotFoundException	
	Datagram		
	DatagramConnection		
	InputConnection		
	OutputConnection		
	StreamConnection		
	StreamConnectionNotifier		

Tabla A.4: Paquete `javax.microedition.io`

A.2. API MIDP

Clases	Interfaces	Excepciones	Errores
		IllegalStateException	

Tabla A.5: Extensión de MIDP al paquete `java.lang`

Clases	Interfaces	Excepciones	Errores
Timer			
TimerTask			

Tabla A.6: Extensión de MIDP al paquete `java.util`

Clases	Interfaces	Excepciones	Errores
RecordStore	RecordComparator	InvalidRecordIDException	
	RecordEnumeration	RecordStoreException	
	RecordFilter	RecordStoreFullException	
	RecordListener	RecordStoreNotFoundException	
		RecordStoreNotOpenException	

Tabla A.7: Extensión de MIDP al paquete `javax.microedition.rms`

Clases	Interfaces	Excepciones	Errores
MIDlet		MIDletStateChangeException	

Tabla A.8: Extensión de MIDP al paquete `javax.microedition.midlet`

Clases	Interfaces	Excepciones	Errores
	HttpConnection		

Tabla A.9: Extensión de MIDP al paquete `javax.microedition.io`

Clases	Interfaces	Excepciones	Errores
Alert	Choice		
AlertType	CommandListener		
Canvas	ItemStateListener		
ChoiceGroup			
Command			
DateField			
Display			
Displayable			
Font			
Form			
Gauge			
Graphics			
Image			
ImageItem			
Item			
List			
Screen			
StringItem			
TextBox			
TextField			
Ticker			

Tabla A.10: Extensión de MIDP al paquete `javax.microedition.lcdui`

Apéndice B

Algoritmos Criptográficos

B.1. Conceptos básicos

Un algoritmo de cifrado o encriptación [7, 8] es un algoritmo que pertenece a una familia de transformaciones invertibles, que obtiene representaciones sin sentido del texto original y que está controlado por una clave.

Formalmente, un algoritmo de cifrado se puede expresar como

$$E_k : M \rightarrow C$$

donde $k \in K$, K es el espacio de claves, M es el espacio de mensajes y C el de criptogramas.

Así se tiene que $E_k(m) = c$ y se cumple que $E_{k_1}(m) \neq E_{k_2}(m)$ sii $k_1 \neq k_2$.

Por otra parte, el descifrado consiste en obtener un texto legible a partir de una representación sin sentido (criptograma), haciendo uso de una clave.

El algoritmo de descifrado correspondiente a E_k se puede expresar formalmente como $D_k = E_k^{-1}$, o también como

$$D_k : C \rightarrow M$$

Por lo tanto ahora se cumple que $D_k(c) = m$ y $D_k(c) = D_k(E_k(m)) = m$.

En cuanto a la clave k, se trata de datos aleatorios que representan números con valor matemático. La clave para cifrado y descifrado puede ser la misma o no.

B.2. Criptografía simétrica

Este tipo de criptografía se basa en que la clave k usada en el proceso de cifrado y descifrado es la misma. A esta clave única se la llama también secreto compartido.

La gran ventaja de este tipo de sistemas es la velocidad en los procesos de encriptación y desencriptación de datos. Por contra, tienen la desventaja del número y distribución de los secretos compartidos.

B.2.1. CBC

Este algoritmo [41] pertenece a los algoritmos de cifrado por bloques. En criptografía, una unidad de cifrado por bloques es una unidad de cifrado de clave simétrica que opera en grupos de bits de longitud fija, llamados bloques, aplicándoles una transformación invariante.

Cuando realiza cifrado, una unidad de cifrado por bloques toma un bloque de texto en claro como entrada y produce un bloque de igual tamaño de texto cifrado. La transformación exacta es controlada utilizando una segunda entrada (la clave secreta). El descifrado es similar: se introducen bloques de texto cifrado y se producen bloques de texto en claro.

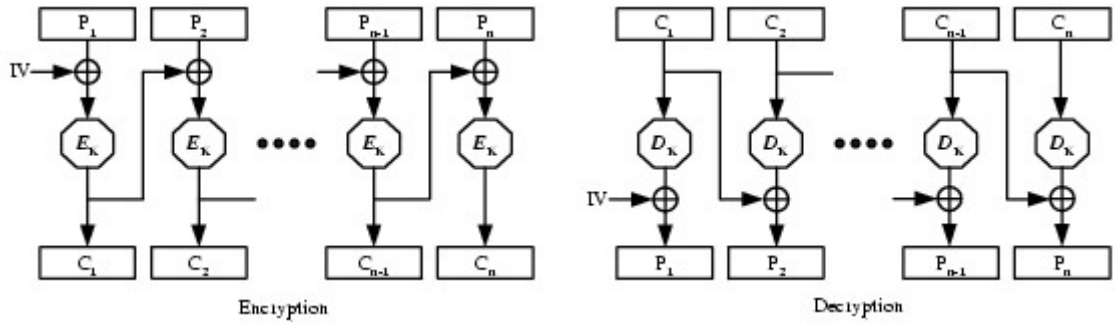

Figura B.1: Cifrado y descifrado en CBC

En CBC, a cada bloque de texto en claro se le aplica la operación XOR con el bloque cifrado anterior antes de ser cifrado. De esta forma, cada bloque de texto cifrado depende de todo el texto en claro procesado hasta este punto. También, para hacer cada mensaje único se utiliza un vector de inicialización.

B.2.2. DES

DES [42] es el algoritmo prototipo del cifrado por bloques. En el caso de DES el tamaño del bloque es de 64 bits. DES utiliza también una clave criptográfica para modificar la transformación, de modo que el descifrado sólo puede ser realizado por aquellos que conozcan la clave concreta utilizada en el cifrado. La clave mide 64 bits, aunque en realidad, sólo 56 de ellos son empleados por el algoritmo. Los ocho bits restantes se utilizan únicamente para comprobar la paridad, y después son descartados. Por tanto, la longitud de clave efectiva en DES es de 56 bits, y así es como se suele especificar.

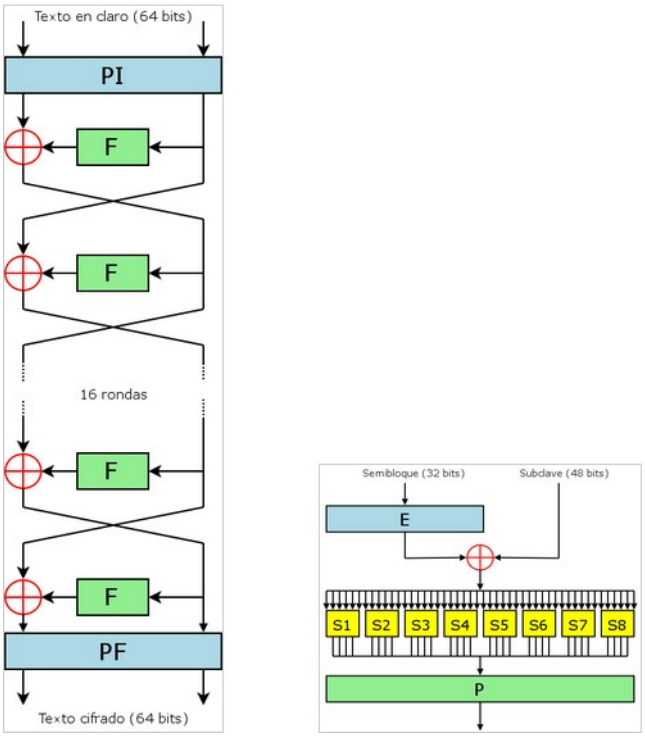

Figura B.2: Estructura básica de DES y función de Feistel

En DES hay 16 fases idénticas, denominadas rondas. También hay una permutación inicial y final denominadas PI y PF, que son funciones inversas entre sí (PI "deshace" la acción de PF, y viceversa). Antes de las rondas, el bloque es dividido en dos mitades de 32 bits y procesadas alternativamente. Este entrecruzamiento se conoce como esquema Feistel.

La estructura de Feistel asegura que el cifrado y el descifrado sean procesos muy similares (la única diferencia es que las subclaves se aplican en orden inverso cuando desciframos). El resto del algoritmo es idéntico.

Hoy en día, DES se considera inseguro para muchas aplicaciones. Esto se debe principalmente a que el tamaño de clave de 56 bits es corto; las claves de DES se han roto en menos de 24 horas. Existen también resultados analíticos que demuestran debilidades teóricas en su cifrado, aunque son inviables en la práctica. Se cree que el algoritmo es seguro en la práctica en su variante de Triple DES, aunque existan ataques teóricos.

Desde hace algunos años, el algoritmo ha sido sustituido por el nuevo AES.

Triple DES

Consiste básicamente en aplica DES sobre la salida de DES, utilizando dos claves de la siguiente manera:

- Cifrado: $C = E_{k_1}(D_{k_2}(E_{k_1}(P)))$
- Descifrado: $P = D_{k_1}(E_{k_2}(D_{k_1}(C)))$

B.2.3. AES

AES [43] tiene un tamaño de bloque fijo de 128 bits y tamaños de llave de 128, 192 ó 256 bits. Trabaja sobre un array de 4×4 bytes, llamado *state*. Para el cifrado, cada ronda de la aplicación del algoritmo AES (excepto la última) consiste en cuatro pasos:

1. `SubBytes`: en este paso se realiza una sustitución no lineal donde cada byte es reemplazado con otro de acuerdo a una tabla.
2. `ShiftRows`: en este paso se realiza un transposición donde cada fila del *state* es rotado de manera cíclica un número determinado de veces.
3. `MixColumns`: operación de mezclado que opera en las columnas del *state*, combinando los cuatro bytes en cada columna usando una transformación lineal.
4. `AddRoundKey`: cada byte del *state* es combinado con la clave "round"; cada clave "round" se deriva de la clave de cifrado usando una *key schedule*.

La ronda final omite la fase `MixColumns`.

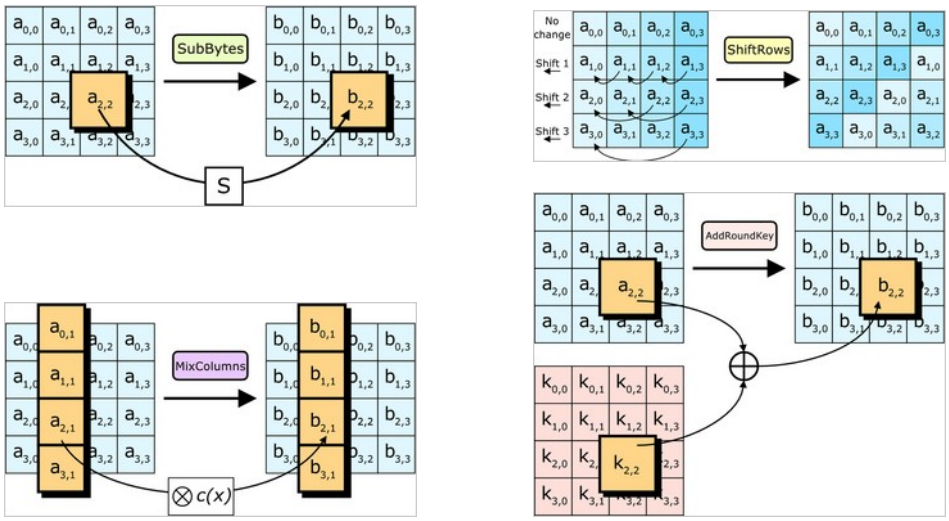

Figura B.3: Los 4 pasos del algoritmo AES

B.2.4. IDEA

IDEA [44] opera con bloques de 64 bits usando una clave de 128 bits y consiste en ocho transformaciones idénticas (ver figura B.4) y una transformación de salida (media ronda). El proceso para encriptar y desencriptar es similar.

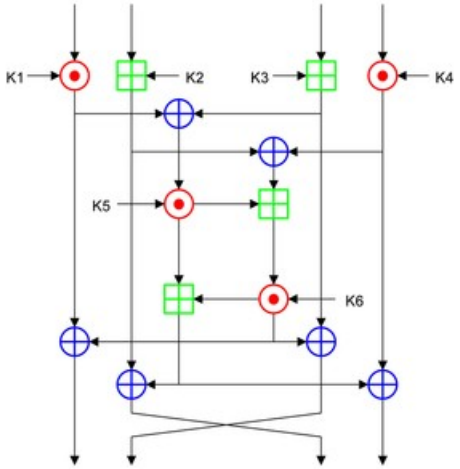

Figura B.4: Algoritmo IDEA

IDEA utiliza tres operaciones en su proceso con las cuales logra la confusión, se realizan con grupos de 16 bits y son las siguientes:

- Suma y aplica módulo 2^{16} (cuadrado con un más +)
- Multiplica y aplica módulo $2^{16}+1$ (círculo con punto $\cdot$)
- Operación O exclusiva (XOR) (círculo con un más +)

($2^{16} = 65536$; $2^{16}+1 = 65537$, que es primo)

B.2.5. RC4

Dentro de la criptografía RC4 [45] es el sistema de cifrado de flujo más utilizado y se usa en algunos de los protocolos más populares como TLS/SSL y WEP.

RC4 genera un flujo pseudoaleatorio de bits (un *keystream*) que, para encriptación, se combina con el texto plano usando la función XOR como en cualquier Cifrado Vernam. La fase de descifrar el mensaje se realiza del mismo modo.

Para generar el *keystream*, el algoritmo de cifrado tiene un estado interno secreto que consiste en,

1. Una permutación de todas las posibles con 256 bytes
2. Dos índices-apuntadores de 8 bits

La permutación se inicializa con una clave de longitud variable, habitualmente entre 40 y 256 bits usando un algoritmo de programación de claves (o KSA). Una vez completado , el flujo de bits cifrados se genera usando un algoritmo de generación pseudoaleatoria (o PRGA).

RC4 fue excluido en seguida de los estándares de alta seguridad por los criptógrafos y algunos modos de usar el algoritmo de criptografía RC4 lo han llevado a ser un sistema de criptografía muy inseguro, incluyendo su uso WEP. No está recomendado su uso en los nuevos sistemas, sin embargo, algunos sistemas basados en RC4 son lo suficientemente seguros para un uso común.

B.3. Criptografía asimétrica

La criptografía asimétrica [7, 8] se basa en el uso de dos claves distintas para cifrar y descifrar. Una de ellas es la clave pública k_p y la otra es la clave privada (o secreta) k_s, y todo lo que se cifra/descifra con la primera sólo puede ser descifrado/cifrado con la segunda, y viceversa.

Ambas claves pertenecen al emisor del mensaje, con lo que desaparece el problema de la distribución de claves. Por contra, tiene la desventaja de ser más lenta que la criptografía simétrica en cuanto a tiempo de computación se refiere.

Figura B.5: Criptografía asimétrica

Las propiedades que debe cumplir todo algoritmo criptográfico asimétrico son las siguientes:

- Dada la clave k_p y el mensaje P, debe poder calcularse $C = E_{k_p}(P)$ en $O(n)$.
- Dada la clave k_s y el criptograma C, debe poder calcularse $P = D_{k_p}(C)$ en $O(n)$.
- Obtener k_s a partir de k_p debe ser intratable.
- Obtener P a partir de k_p y C debe ser intratable.

B.3.1. Intercambio Diffie-Hellman

El modelo Diffie-Hellman [46] permite el intercambio secreto de claves entre dos partes que no han tenido contacto previo. Se emplea generalmente como medio para acordar claves simétricas que serán empleadas para el cifrado de una sesión.

Su robustez se basa en la dificultad del cálculo de logaritmos discretos en un espacio finito. Sin embargo, es sensible a problemas del tipo *man-in-the-middle*, ya que no autentica a los participantes en el intercambio. Tampoco puede ser usado para cifrar o descifrar.

El funcionamiento básico de Diffie-Hellman es el siguiente:

- Tanto el emisor A como el receptor B acuerdan un número primo p, así como un generador $g \in Z_p^*$ (donde Z_p^* es el conjunto de los enteros menores que p que son primos relativos de p).
- A escoge un número aleatorio $x \in Z_{p-1}$ y envía $X = g^x \ mod \ p$.
- B escoge un número aleatorio $y \in Z_{p-1}$ y envía $Y = g^y \ mod \ p$.
- A calcula $k = Y^x$
- B calcula $k' = X^y$

De esta forma, $k = k' = g^{xy}$, siendo k el secreto compartido. Por otra parte, calcular k a partir de p, g, X e Y es computacionalmente intratable.

B.3.2. RSA

El sistema criptográfico con clave pública RSA [47] se basa en que todo usuario de dicho sistema hace pública una clave de cifrado y oculta una clave de descifrado. Cuando se envía un mensaje, el emisor busca la clave pública de cifrado del receptor y una vez que dicho mensaje llega al receptor, éste se ocupa de descifrarlo usando su clave oculta.

Los mensajes enviados usando RSA se representan mediante números y el funcionamiento se basa en el producto de dos números primos grandes (mayores que 10^{100}) elegidos aleatoriamente para conformar la clave de descifrado. La seguridad de este algoritmo radica en la intratabilidad de la factorización de grandes números en sus factores primos. Sin embargo, puede que la computación cuántica resuelva este problema de factorización.

Cada participante genera su par de claves de la siguiente manera:

- Se eligen dos números primos grandes distintos p y q, de igual longitud
- Se calcula $n = pq$
- Se elige aleatoriamente e tal que $1 < e < (p-1)(q-1)$ y tales que e y $(p-1)(q-1)$ sean primos entre sí
- Se calcula d tal que $de \equiv 1 \ mod \ (p-1)(q-1)$

De esta forma, la clave pública está compuesta por n (el módulo) y e (el exponente público), mientras que la clave privada la conforman n (el módulo) y d (el exponente privado).

Este algoritmo, que fue la primera implementación del modelo Diffie-Hellman, es válido tanto para cifrado como para firma digital. Resulta muy fácil de implementar, aunque las implementaciones actuales son verdaderamente lentas.

B.4. Envoltura digital

La envoltura digital consiste básicamente en la negociación de claves simétricas mediante criptografía asimétrica. Así, por un lado se usa un método seguro para compartir claves y por otro lado se usa un método rápido para cifrar los datos.

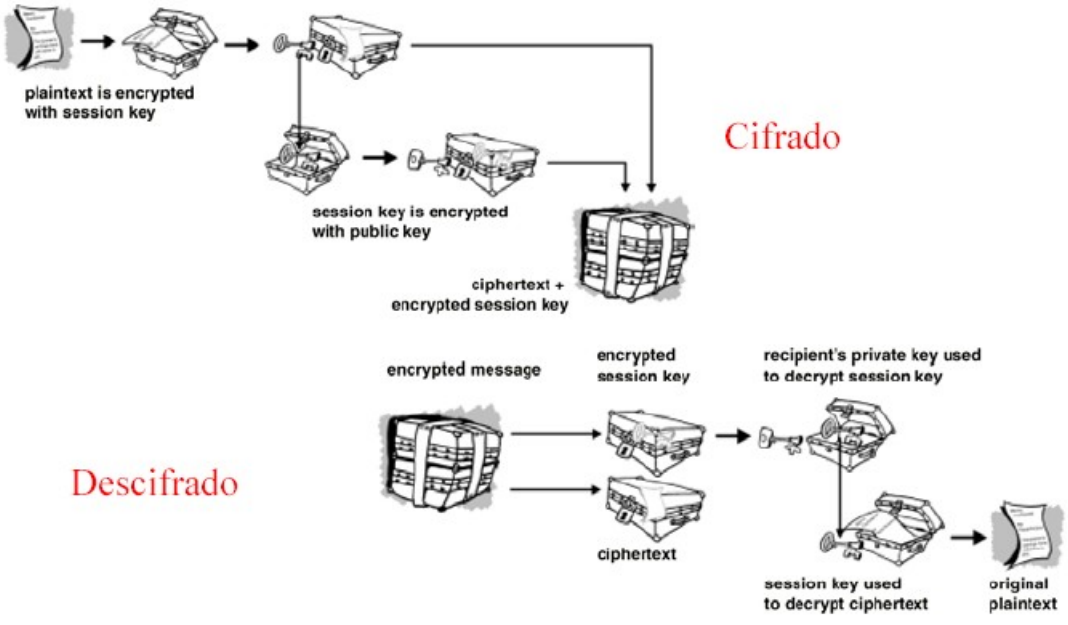

Figura B.6: Esquema de Envoltura Digital

B.5. Resumen digital

Una función hash transforma mensajes de longitud arbitraria en mensajes de longitud fija, denominados resúmenes digitales. Toda función hash que se considere segura debe cumplir los siguientes criterios:

- **Unidireccionalidad**: Dado un resumen $h(m)$, debe resultar computacionalmente imposible encontrar el mensaje m.
- **Compresión**: A partir de un mensaje m de cualquier longitud, el resumen $h(m)$ debe tener, como ya hemos dicho, una longitud fija.
- **Facilidad de cálculo**: Debe ser fácil calcular el resumen $h(m)$ a partir del mensaje m.
- **Difusión**: El resumen $h(m)$ debe ser el resultado de una función compleja de todos los bits del mensaje m.

B.5.1. MD5

MD5 [48] es uno de los algoritmos de resumen digital más ampliamente usado. Permite calcular un código hash no reversible de 128 bits, siguiendo los siguientes pasos (suponiendo un mensaje de entrada m de b bits de longitud):

1. **Inserción de bits**. El mensaje m se extiende hasta que se forme el menor número múltiplo de 512 bits. Se añade un único 1, y el resto son ceros.
2. **Longitud del mensaje**. Una representación de 64 bits de b se añade al resultado del paso anterior.
3. **Inicialización del búfer MD**. Un búfer de cuatro palabras (4×32 bits) A, B, C y D se usa para calcular el resumen del mensaje.
4. **Procesado del mensaje**. Se procesa el mensaje m en bloques de 16 palabras. Este paso usa una tabla de 64 elementos construida con la función seno.
5. **Salida**. El resumen del mensaje m es la salida producida por A, B, C y D. Esto es, se comienza por el byte de menor peso de A y se termina con el byte de mayor peso de D.

B.5.2. SHA-1

El algoritmo SHA-1 [49] genera un resumen digital de 160 bits a partir de un mensaje m de longitud aleatoria, pero siempre menor que 2^{64} bits. Además, está basado en principios similares a los empleados en el diseño del algoritmo MD5.

Existen cuatro variantes más cuyas diferencias se basan en un diseño algo modificado y rangos de salida incrementados. Estas variantes son: SHA-224, SHA-256, SHA-384 y SHA-512 (todos ellos referidos como SHA-2).

B.6. Firma digital

La figura B.7 expresa claramente el concepto básico de la firma digital. Ésta consiste en adjuntar al texto claro a enviar un resumen digital cifrado con la clave privada k_s del emisor.

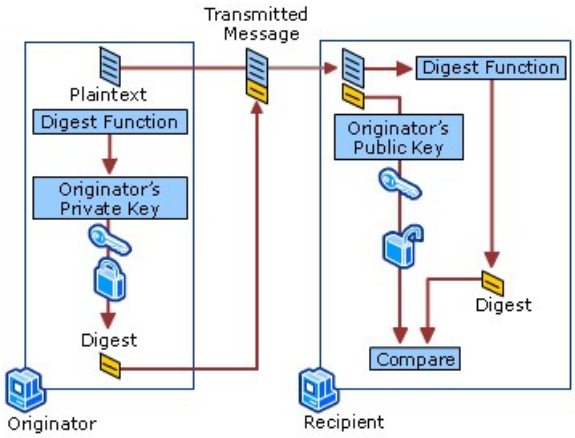

Figura B.7: Esquema de Firma Digital

Del lado del receptor, éste toma por un lado el resumen digital cifrado y lo descifra con la clave pública k_p del emisor, y por otro lado realiza un resumen digital (con el mismo algoritmo que el emisor) del texto en claro. Entonces compara ambos resúmenes y, si coinciden, el receptor tiene la seguridad de que el emisor es quién dice ser, y de que el mensaje no ha sido modificado en la transmisión.

La firma digital, por tanto, asegura la identidad del emisor, así como la integridad del mensaje. Sin embargo, implica la confianza en el poseedor de la clave pública k_p y/o en la entidad que certifica la validez de dicha clave.

Apéndice C

Ejemplos de código de la librería de IAIK

A continuación se muestran una serie de ejemplos de utilización de la librería *IAIK PKCS#11 Provider Micro Edition* [9]:

Ejemplo 1 *Instanciación de un módulo.*

```
String moduleName = "mpkcs11.dll";
Module module = Module.getInstance(moduleName);
```

Ejemplo 2 *Obtención del token.*

```
Token[] tokens = module.getTokens();
```

Ejemplo 3 *Login del usuario.*

```
if (token.loginRequired())
  if (!token.userLoggedIn()) {
    if (token.hashProtectedAuthenticationPath()) {
      token.loginUser(null);
    } else {
      char[] pin = ...; //Obtención del PIN por algún método
      token.loginUser(pin);
    }
  }
```

Ejemplo 4 *Logout del usuario.*

```
if (token.userLoggedIn()) {
  token.logout();
}
```

Ejemplo 5 *Obtención de la etiqueta y el número de serie del token.*

```
String label = token.getLabel();
String serialNumber = token.getSerialNumber();
```

Ejemplo 6 *Generación de información aleatoria.*

```
byte[] randomData = new byte[16];
if (token.supportsSecureRandom()) {
  SecureRandom randomGenerator = token.getSecureRandom();
  randomGenerator.nextBytes(randomData);
}
```

Ejemplo 7 *Recuperación de claves y certificados del token.*

```
KeyStore keyStore = token.getKeyStore();
Vector keyAliases = new Vector();
Vector certificateAliases = new Vector();
for (Enumeration aliases = keyStore.aliases(); aliases.hasMoreElements(); ) {
  String alias = (String) aliases.nextElement();
  if (keyStore.isKeyEntry(alias)) {
    keyAliases.addElement(alias);
  }
  if (keyStore.isCertificateEntry(alias)) {
    certificateAliases.addElement(alias);
  }
}
```

Ejemplo 8 *Resumen digital de datos mediante el algoritmo SHA-1, vía hardware.*

```
byte[] hashValue;
if (token.supportsAlgorithm(MessageDigest.ALGORITHM_SHA_1,Token.HARDWARE)) {
  MessageDigest hash = token.getMessageDigest(MessageDigest.ALGORITHM_SHA_1);
  hash.update(data);
  hashValue = hash.digest();
}
```

Ejemplo 9 *Generación de pares de claves mediante el algoritmo RSA, vía hardware, para ser usadas en proceso de firma.*

```
Key publicKey;
Key privateKey;
if (token.supportsAlgorithm(KeyPairGenerator.ALGORITHM_RSA,Token.HARDWARE)) {
  KeyPairGenerator keyPairGen =
                   token.getKeyPairGenerator(KeyPairGenerator.ALGORITHM_RSA);
  keyPairGen.initialize(1024,KeyPairGenerator.USAGE_SIGNATURE, "KeyPair 1");
  Key[] keyPair = keyPairGen.generateKeyPair();
  publicKey = keyPair[0];
  privateKey = keyPair[1];
}
```

Ejemplo 10 *Generación de una clave mediante el algoritmo RC2, vía hardware para ser usada en proceso de firma.*

```
Key key;
if (token.supportsAlgorithm(KeyGenerator.ALGORITHM_RC2,Token.HARDWARE)) {
  KeyGenerator keyGenerator = token.getKeyGenerator(KeyGenerator.ALGORITHM_RC2);
  keyGenerator.initialize(128, KeyGenerator.USAGE_SIGNATURE, "Key 1",true);
  Key key = keyGenerator.generateKey();
}
```

Ejemplo 11 *Inicialización de un token.*

```
char[] soPIN;
if (token.hashProtectedAuthenticationPath()) {
  soPIN = null;
} else {
  soPIN = promptSoPIN();
}
token.init(soPIN, tokenLabel);
```

Ejemplo 12 *Inicialización del PIN de usuario.*

```
if (!token.userPinInitialized()) {
  char[] soPIN;
  char[] userPIN;
  if (token.hashProtectedAuthenticationPath()) {
    soPIN = null;
    userPIN = null;
  } else {
    soPIN = promptSoPIN();
    userPIN = promptUserPIN();
  }
  token.initPIN(soPIN, userPIN);
}
```

Ejemplo 13 *Cambio del PIN de usuario.*

```
char[] userPIN;
char[] newUserPIN;
if (token.hashProtectedAuthenticationPath()) {
  userPIN = null;
  newUserPIN = null;
} else {
  userPIN = promptUserPIN();
  newUserPIN = promptNewUserPIN();
}
token.setUserPIN(userPIN, newUserPIN);
```

Ejemplo 14 *Generación de información aleatoria con uso de semilla.*

```
byte[] randomData = new byte[16];
if (token.supportsSecureRandom()) {
  SecureRandom randomGenerator = token.getSecureRandom();
  try {
    randomGenerator.setSeed(seed);
  } catch (PKCS11RuntimeException ex) {
    // the application may ignore this, maybe write a log entry
  }
  randomGenerator.nextBytes(randomData);
}
```

Ejemplo 15 .

```
byte[] digestInfo =
   MessageDigest.makeDigestInfo(hashValue,MessageDigest.ALGORITHM_SHA_1);
byte[] dataToBeSigned = digestInfo;
```

Ejemplo 16 *Eliminación de una clave o certificado del token.*

```
keyStore.delete(alias);
```

Ejemplo 17 *Recuperación del certificado asociado a una clave del token.*

```
if (keyStore.isKeyEntry(alias))
{
  Key key = keyStore.getKey(alias);
  byte[] assicoatedCertificate = keyStore.getCertificate(alias);
}
```

Ejemplo 18 *Generación de una clave privada mediante el algoritmo de RSA, almacenamiento de dicha clave en el token y generación y almacenamiento del correspondiente certificado digital.*

```
long[] keyUsage = new long[] {Key.USAGE_SIGNATURE_CREATION};
KeyTemplate keyTemplate =
       new KeyTemplate(Key.TYPE_PRIVATE_KEY,Key.ALGORITHM_RSA, keyUsage);
keyTemplate.setComponent(Key.COMPONENT_MODULUS, modulus);
keyTemplate.setComponent(Key.COMPONENT_PRIVATE_EXPONENT,privateExponent);
keyTemplate.setComponent(Key.COMPONENT_PUBLIC_EXPONENT,publicExponent);
keyTemplate.setComponent(Key.COMPONENT_PRIME_1,primeP);
keyTemplate.setComponent(Key.COMPONENT_PRIME_2, primeQ);
keyTemplate.setComponent(Key.COMPONENT_EXPONENT_1,primeExponentP);
keyTemplate.setComponent(Key.COMPONENT_EXPONENT_2,primeExponentQ);
keyTemplate.setComponent(Key.COMPONENT_COEFFICIENT,crtCoefficient);
KeyStore keyStore = token.getKeyStore();
String alias = "imported key";
keyStore.setKey(alias, keyTemplate);
X509Certificate certificate = new X509Certificate(encodedCertificate);
keyStore.setCertificate(alias,
             encodedCertificate,certificate.getSubjectDN().getEncoded());
```

Ejemplo 19 *Firma de datos mediante el algoritmo raw RSA, vía hardware.*

```
if ((key.getType() == Key.TYPE_PRIVATE_KEY)
&& key.canBeUsedFor(Key.USAGE_SIGNATURE_CREATION)
&& (key.getAlgorithm() == Key.ALGORITHM_RSA))
{
  byte[] signatureValue;
  if (token.supportsAlgorithm(Signature.ALGORITHM_RawRSA,Token.HARDWARE))
  {
    Signature signatureEngine = token.getSignature(Signature.ALGORITHM_RawRSA);
    signatureEngine.initSign(key);
    signatureEngine.update(dataToBeSigned);
    signatureValue = signatureEngine.sign();
  }
}
```

Ejemplo 20 *Verificación de una firma electrónica mediante el algoritmo raw RSA, vía hardware.*

```
if ((key.getType() == Key.TYPE_PUBLIC_KEY)
&& key.canBeUsedFor(Key.USAGE_SIGNATURE_VERIFICATION)
&& (key.getAlgorithm() == Key.ALGORITHM_RSA))
{
  boolean signatureVerified;
  if (token.supportsAlgorithm(Signature.ALGORITHM_RawRSA,Token.HARDWARE))
  {
    Signature signatureEngine = token.getSignature(Signature.ALGORITHM_RawRSA);
    signatureEngine.initVerify(key);
    signatureEngine.update(dataToBeSigned);
    signatureVerified = signatureEngine.verify(signatureValue);
  }
}
```

Ejemplo 21 *Cifrado de datos con una clave pública mediante el algoritmo RSA PKCS1 padding, vía hardware.*

```
if ((key.getType() == Key.TYPE_PUBLIC_KEY)
&& key.canBeUsedFor(Key.USAGE_ENCRYPTION)
&& (key.getAlgorithm() == Key.ALGORITHM_RSA))
{
  byte[] ciphertext;
  if (token.supportsAlgorithm(Cipher.ALGORITHM_RSA_PKCS1PADDING,Token.HARDWARE))
  {
    Cipher cipher = token.getCiphere(Cipher.ALGORITHM_RSA_PKCS1PADDING);
    cipher.init(Cipher.ENCRYPT_MODE, key, null);
    cipher.update(plaintext);
    ciphertext = cipher.doFinal();
  }
}
```

Ejemplo 22 *Descifrado de datos haciendo uso de una clave privada mediante el algoritmo RSA PKCS1 padding, vía hardware.*

```
if ((key.getType() == Key.TYPE_PRIVATE_KEY)
&& key.canBeUsedFor(Key.USAGE_DECRYPTION)
&& (key.getAlgorithm() == Key.ALGORITHM_RSA))
{
  byte[] plaintext;
  if (token.supportsAlgorithm(Cipher.ALGORITHM_RSA_PKCS1PADDING,Token.HARDWARE))
  {
    Cipher cipher = token.getCiphere(Cipher.ALGORITHM_RSA_PKCS1PADDING);
    cipher.init(Cipher.DECRYPT_MODE, key, null);
    cipher.update(ciphertext);
    plaintext = cipher.doFinal();
  }
}
```

Ejemplo 23 *Envoltura de una clave privada haciendo uso de una clave pública mediante el algoritmo RSA PKCS1 padding, vía hardware.*

```
if ((key.getType() == Key.TYPE_PUBLIC_KEY)
&& key.canBeUsedFor(Key.USAGE_WRAP)
&& (key.getAlgorithm() == Key.ALGORITHM_RSA)) {
  byte[] wrappedKey;
  if (token.supportsAlgorithm(Cipher.ALGORITHM_RSA_PKCS1PADDING,Token.HARDWARE)) {
    Cipher cipher = token.getCiphere(Cipher.ALGORITHM_RSA_PKCS1PADDING);
    cipher.init(Cipher.WRAP_MODE, key, null);
    wrappedKey = cipher.wrap(secretKey);
  }
}
```

Ejemplo 24 *Desenvoltura de una clave usada para cifrar con el algoritmo AES haciendo uso de una clave privada mediante el algoritmo RSA PKCS1 padding, vía hardware.*

```
if ((key.getType() == Key.TYPE_PRIVATE_KEY)
&& key.canBeUsedFor(Key.USAGE_UNWRAP)
&& (key.getAlgorithm() == Key.ALGORITHM_RSA)) {
  Key unwrappedKey;
  if (token.supportsAlgorithm(Cipher.ALGORITHM_RSA_PKCS1PADDING,Token.HARDWARE)) {
    Cipher cipher = token.getCiphere(Cipher.ALGORITHM_RSA_PKCS1PADDING);
    cipher.init(Cipher.UNWRAP_MODE, key, null);
    unwrappedKey = cipher.unwrap(wrappedKey, Key.ALGORITHM_AES,
          KeyGenerator.USAGE_CIPHER, "unwrapped-AES", false);
  }
}
```

Ejemplo 25 *Cierre del módulo.*

```
module.close();
```

Apéndice D

Hardware

En este proyecto, se dispondrá de los siguientes elementos "hardware":

- PDA Acer N50 [23]
  - Procesador Intel®PXA272 a 312 MHz con Tecnología Intel XScale
  - 64 MB de memoria del sistema
  - 64 MB de flash ROM para aplicaciones y documentos
  - IEEE 802.11b wireless LAN
  - Infrarrojos y Bluetooth®1.2
  - CF Type II y SD/MMC (con SDIO) slots de expansión
  - Pantalla LCD de 3,5"
  - Micrófono y altavoces integrados
  - Batería Li-ion recargable 1060mAh
  - Dimensiones 120 x 70 x 17,4 mm
  - Peso 150g
  - Sistema operativo Microsoft®Windows Mobile 2003 Second Edition

Figura D.1: PDA Acer N50

- Una *smart card* con capacidad criptográfica.
  - 8 Kb de almacenamiento mínimo
  - Certificado almacenado

Figura D.2: Smart Card

- Un USB *smart card* reader *GemPC Twin* de GEMPLUS [17].
  - Lector/grabador para *smart cards* ISO7816 (protocolos T=0 y T=1)
  - Soporte para tarjetas de 3V y 5V
  - Hasta 100.000 ciclos de inserción
  - Interfaz USB (CCID)
  - Comunicación con la *smart card* programable desde 9600 baudios hasta 115200
  - Comunicación de alta velocidad con el PC a través del puerto USB
  - Alimentado de corriente a través del puerto USB
  - Controladores PC/SC (Win98, WinME, NT4, Win2000, XP, Win2003)

Figura D.3: USB smart card reader *GemPC Twin* de GEMPLUS

- Lector de tarjetas SDIO Tactel S300 [18]
  - Lector/grabador para *smart cards* ISO7816 y EMV
  - Conector SDIO v1.1, 2.0
  - Soporte de los protocolos T=0 y T=1
  - Sistema operativo Pocket PC Windows Mobile 5.0

Figura D.4: Lector de tarjetas SDIO Tactel S300

- Lector de tarjetas para PDA Bluetooth CASPAD C100 [19]
  - Lector/grabador para *smart cards* ISO7816 y EMV
  - Conexión vía Bluetooth o USB
  - Soporte de los protocolos T=0 y T=1
  - Soporte del estándar AS2805
  - Comunicación cifrada con algoritmos como Triple DES o RSA
  - Sistema operativo Pocket PC Windows Mobile 5.0

Figura D.5: Lector de tarjetas Bluetooth CASPAD C100

Apéndice E

Acrónimos

A2DP, Advanced Audio Distribution Profile

ACL, Asynchronous Connection-Less

AES, Advanced Encryption Standard

APDU, Application Protocol Data Unit

API, Application Programming Interface

ARM, Acorn RISC Machine

AWT, Abstract Windows Toolkit

BSD, Berkeley Software Distribution

CA, Certification Authority

CBC, Cipher Block Chaining

CCID, Chip/Smart Card Interface Devices

CDC, Connected Device Configuration

CLDC, Connected Limited Device Configuration

Cryptoki, Cryptographic token interface

CSP, Cryptographic Service Provider

DES, Data Encryption Standard

DF, Dedicated File

DH, Diffie Hellman

DLL, Dynamic Link Library

DSA, Digital Signature Algorithm

ECB, Electronic codebook mode

EF, Elementary File

EMV, Europay Mastercard Visa

ETSI, European Telecommunications Standards Institute

EVM, Embedded Virtual Machine

FACTO, Firma electrónicA y servicios de Certificación en Terminales mOviles

FAQ, Frequently Asked Questions

FP, Foundation Profile

GFC, Generic Connection Framework

GUI, Graphic User Interface

HCI, Host Controller Interface

HFP, Hands-Free Profile

HMAC, keyed-Hash Message Authentication Code

HP, Hewlett Packard

HSM, Hardware Security Module, o bien Host Security Module

HSP, HeadSet Profile

IAIK, Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie, Institute for Applied Information Processing and Communications

IBM, International Business Machines corporation

IC, Integrated Circuit

ICP, IterCom Profile

IDEA, International Data Encryption Algorithm

IEEE, Institute of Electrical and Electronics Engineers

ISDN, Integrated Services Digital Network

ISM, Industrial Scientific and Medical

ISO, International Standards Organization

ITRON, Industrial TRON

J2EE, Java 2 Enterprise Edition

J2ME, Java 2 Micro Edition

J2SE, Java 2 Standard Edition

JAR, Java ARchive

JAD, Java Application Descriptor

JCA, Java Cryptography Architecture

JCE, Java Cryptography Extension

JCP, Java Community Process

JCRMI, Java Card Remote Method Invocation

JDK, Java Development Kit

JNI, Java Native Interface

JSCP, Java Software Co-Processor

JSR, Java Specification Request

JVM, Java Virtual Machine

KSA, Key Scheduling Algorithm

KVM, K-Virtual Machine

L2CAP, Logical Link Control and Adaption Protocol

LCD, Liquid Crystal Display

LGPL, Lesser General Public License

LLC, Logical Link Control

LMP, Link Manager Protocol

MAC, Message Authentication Code o Media Access Control

MD5, Message-Digest Algorithm 5

MF, Master File

MIDP, Mobile Information Device Profile

MIPS, Millions of Instructions Per Second

MMC, Multi Media Card

MPKCS, Mobile PKCS

MSSP, Mobile Signature Service Provider

N/A Not Available o Not Applicable

OAEP, Optimal Asymmetric Encryption Padding

OBEX, OBject EXchange

OEM, Original Equipment Manufacturer

OSE, Operating System Embedded

PBP, Personal Basis Profile

PCM, Pulse Code Modulation

PCMCIA, Personal Computer Memory Card International Association

PIM, Personal Information Manager

PKCS, Public Key Cryptography Standard

PKI, Public Key Infrastructure

PP, Personal Profile

PPP, Point-to-Point Protocol

PRGA, Pseudo-Random Generation Algorithm

PSS, Probabilistic Signature Scheme

RC4, Rivest Cipher 4

RFCOMM, Radio Frequency COMMunication

RIPEMD, Race Integrity Primitives Evaluation Message Digest

RISC, Reduced Instruction Set Computer

RMI, Remote Method Invocation

R/O, Read-Only

ROM, Read Only Memory

RS232, Recommended Standard 232

RSA, Rivest Shamir Adleman

R/W, Read/Write

SATK, SIM, Application Toolkit

SATSA, Security And Trust Services API

SCO, Synchronized Connection Oriented

SD, Secure Digital

SDAP, Service Discovery Application Profile

SDDB, Service Discovery Data Base

SDIO, Secure Digital Input Output

SDP, Service Discovery Profile

SH3, Super H-3

SHA, Secure Hash Algorithm

SIG, Special Interest Group

SIM, Subscriber Identity Module

S/MIME, Secure Multipurpose Internet Mail Extensions

SO, Security Officer

SSL, Secure Socket Layer

TCS, Telephony Control Specification

TDM, Time Division Multiplexing

TI, Tarjeta Inteligente

TI OMAP, Texas Instruments OMAP

TIC, Tecnologías de la Información y las Comunicaciones

TLS, Transport Layer Security

USAT, Universal SIM Application Toolkit

VDP, Video Distribution Profile

WAP, Wireless Application Protocol

WEP, Wired Equivalent Privacy o Wireless Encryption Protocol

WIM, WAP Identity Module

Bibliografía

[1] D. Sánchez. "*Diseño e implementación de un entorno móvil de comercio seguro*", Proyecto Fin de Carrera, Facultad de Informática, Universidad de Murcia, septiembre 2001.

[2] T. Jiménez, D. Sánchez. "*Tarjeta inteligente y servicios móviles seguros en entornos administrativos*", ÁTICA, Vicerrectorado de Investigación y Nuevas Tecnologías, Universidad de Murcia, marzo 2006.

[3] M.D. Barnés, D.S. Gómez, A.F. Gómez, M. Martínez, A. Ruiz, D. Sánchez. "*An infraestructure to develop applications based on electronic signature for mobile devices*", Departament of Information and Communications Engineering, University of Murcia, Spain.

[4] J.W. Muchow. "*Core J2ME. Technology and MIDP*", Sun Microsystems Press, Prentice Hall, 2002, ISBN 0-13-066911-3.

[5] D. Wilding-McBride, "*Java Development on PDAs. Building Applications for PocketPC and Palm Devices*", Addison-Wesley, Pearson Education Inc., 2003, ISBN 0-201-71954-1.

[6] F. Bernal, A. Esteban, "*Análisis Comparativo de las Diferentes Máquinas Virtuales Java para Dispositivos Móviles Disponibles en la Actualidad*", Nuevos Servicios y Aplicaciones en Redes, 5º curso de Ingeniería Informática, Universidad de Murcia, marzo 2006.

[7] A. S. Tanenbaum, "*Redes de Computadoras*", 4ª edición, Prentice Hall, Pearson Education Inc., 2003, ISBN 970-26-0162-2.

[8] W. Stallings, "*Comunicaciones y Redes de Computadores*", 7ª edición, Prentice Hall, Pearson Education Inc., 2004, ISBN 84-205-4110-9.

[9] "*IAIK PKCS#11 Provider Micro Edition. User Manual*", Version 1.0, IAIK Stiftung SIC, 27 january 2005.

[10] "*PKCS#11 v2.11: Cryptographic Token Interface Standard*", RSA Laboratories, RSA Security Inc. Public-Key Cryptography Standards (PKCS), November 2001.

[11] P. Borches, "*Java 2 Micro Edition: Soporte Bluetooth*", Universidad Carlos III de Madrid, Marzo 2004.

[12] "*PKCS#15 v1.1: Cryptographic Token Information Syntax Standard*", RSA Laboratories, RSA Security Inc. Public-Key Cryptography Standards (PKCS), June 2000.

[13] L. Hilt, "*Wireless Identity Module. Part: Security*", Wireless Application Protocol Forum, July 2001.

[14] D. Sánchez, M. Martínez "*[eFirma SIM] Análisis de escenarios de firma electrónica a través de una tarjeta SIM criptográfica*", CYUM, Universidad de Murcia, Julio 2006.

[15] "*Transferencia de datos en tarjetas inteligentes*", Tema 5, Sistemas Integrados, Departamento de Ingeniería y Tecnología de los Computadores, Universidad de Murcia, curso 2005-2006.

[16] Muñoz A., "*Manual para instalar CrEme en un PocketPC con Windows CE*", Departamento de Ingeniería y Tecnología de los Computadores, Universidad de Murcia, 2006.

[17] "*GEMPLUS Smart Card Readers*", EPSYS - Smartcards & Systems, 2006.

`http://www.epsys.no/sreaders.htm`

[18] Lars, "*S300 Mobile SC Readers*", Axcess-mobile.com, 2006.

`http://www.axcess-mobile.com/nw/products_smart_card_readers.htm`

[19] "*CASPad C100*", Mobile Payment Solutions, 2006.

`http://www.cardaccess.com.au/pages/caspad.php`

[20] "*.:Portal Oficial sobre el DNI Electrónico:.*", Ministerio del Interior, 2006.

`http://www.dnielectronico.es`

[21] "*OpenSSL: The Open Source toolkit for SSL/TLS*", The OpenSSL Project, 2006.

`http://www.openssl.org`

[22] "*RSA Security - PKCS#11: Cryptographic Token Interface Standard*", RSA Laboratories, RSA Security Inc. Public-Key Cryptography Standards (PKCS), 2004.

`http://www.rsasecurity.com/rsalabs/node.asp?id=2133`

[23] "*Acer España*", 2006.

`http://www.acer.es`

[24] "*Java Platform, Micro Edition (Java ME)*", Sun Microsystems, 2006.

`http://java.sun.com/javame/index.jsp`

[25] "*Java ME Documentation*", Sun Microsystems, 2006.

`http://java.sun.com/javame/reference/docs/index.html`

[26] The Java Community Process(SM) Program,"*JSR 46: Foundation Profile*", Sun Microsystems, 2005.

`http://www.jcp.org/en/jsr/detail?id=46`

[27] The Java Community Process(SM) Program, "*JSR 82: Java APIs for Bluetooth*", Sun Microsystems, 2006.

`http://www.jcp.org/en/jsr/detail?id=82`

[28] The Java Community Process(SM) Program, "*JSR 177: Security and Trust Services API for J2ME*", Sun Microsystems, 2004.

`http://www.jcp.org/en/jsr/detail?id=177`

[29] "*PKCS#11-ME / Mobile Security / Products / Home - Stiftung SIC*", Stiftung Secure Information and Communication Technologies SIC, 2006.

`http://jce.iaik.tugraz.at/sic/products/mobile_security/pkcs_11_me`

[30] Vik David, "*Fair and Biased: Java on PocketPC (Unofficial FAQ)*", 6 de diciembre de 2004.

`http://blog.vikdavid.com/2004/12/java_on_pocketp.html`

[31] S. Berka, "*JAVA for PocketPC PDA's*", 2006.

`http://www.berka.name/stan/jvm-ppc/java_for_pda.html`

[32] "*Java Virtual Machine - All about JVMs*", 2006.

`http://java-virtual-machine.net/other.html`

[33] "*Superwaba. Develop portable mobile applications*", Agosto 2001.

`http://www.superwaba.com.br/etc/SuperWabaFolderEnV3.pdf`

[34] "*(jeode) Insignia Jeode Runtime Environment for WinCE*", 2006.

`http://www.cs.unc.edu/~lindsey/7ds/notes/jeode`

[35] Java Developer's Journal, "*Product Review: Jeode*", Agosto 2001.

`http://jdj.sys-con.com/read/36664.htm`

[36] "*Aonix - PERC Products*", Aonix, 2006.

`http://www.aonix.com/perc.html`

[37] NSIcom, "*CrEme V3.23 User's Guide*", Agosto 2003.

`http://www.berka.name/stan/jvm-ppc/CrEmeUsersGuide.pdf`

[38] WebSphere Web/Durham/IBM, "*IBM Software - WebSphere Everyplace Micro Environment - Features and benefits*", IBM Corporation, 2006.

`http://www-306.ibm.com/software/wireless/weme/features.html`

[39] "*Mysaifu JVM - A free Java Virtual Machine for Windows CE (PocketPC 2003)*", 2005.

`http://www2s.biglobe.ne.jp/~dat/java/project/jvm/index_en.html`

[40] "*pcsc-lite: winscard.h File Reference*", 2006.

`http://pcslite.alioth.debian.org/api/winscard_8h.html`

[41] "*Block cipher - Wikipedia, the free encyclopedia*", Wikimedia Foundation Inc., 2006.

`http://en.wikipedia.org/wiki/Block_cipher`

[42] "*Data Encryption Standard - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.

`http://es.wikipedia.org/wiki/DES`

[43] "*AES - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.

`http://es.wikipedia.org/wiki/AES`

[44] "*IDEA - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.

`http://es.wikipedia.org/wiki/IDEA`

[45] "*RC4 - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.

`http://es.wikipedia.org/wiki/RC4`

[46] "*Diffie-Hellman - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.
`http://es.wikipedia.org/wiki/Diffie-Hellman`

[47] "*RSA - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.
`http://es.wikipedia.org/wiki/RSA`

[48] "*MD5 - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.
`http://es.wikipedia.org/wiki/MD5`

[49] "*SHA - Wikipedia, la enciclopedia libre*", Wikimedia Foundation Inc., 2006.
`http://es.wikipedia.org/wiki/SHA`

[50] "*Bluetooth - Wikipedia, the free encyclopedia*", Wikimedia Foundation Inc., 2006.
`http://en.wikipedia.org/wiki/Bluetooth`

[51] "*Bluetooth Technology Guide at BlueTomorrow.com - Bluetooth Protocol Architecture*", BlueTomorrow.com, 2006.
`http://www.bluetomorrow.com/content/section/90/141`

[52] Jan Beutel, "*Bluetooth @ TIK*", Computer Engineering and Networks Laboratory (TIK), Departement of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology (ETH), Zurich, 2001.
`http://www.tik.ee.ethz.ch/~beutel/bluetooth.html`

[53] MSDN, "*Asignaciones de texto genérico en TCHAR.H*", Microsoft Corporation, 2006.
`http://www.msnd2.microsoft.com/es-es/library/c426s321.aspx`

[54] "*The Legion of the Bouncy Castle*", Bouncy Castle, 2006.
`http://www.bouncycastle.org`