Sistemas Informáticos

Curso 2007-2008

Seguridad en SMS/MMS: Estudio y Mejoras de CryptoSMS

Fernando Casas García
José Pablo Ferrero Prieto
Clara Valerio Sánchez

Dirigido por:

Prof. Luis Javier García Villalba
Grupo de Análisis, Seguridad y Sistemas (GASS)
Dpto. Ingeniería del Software e Inteligencia Artificial (DISIA)

Facultad de Informática

Universidad Complutense de Madrid

Índice:

1. Palabras Clave

- CryptoSMS
- Criptografía
- Seguridad
- SMS
- MMS
- Móvil
- J2ME
- WMA
- Symbian
- DBMS

2. Resumen / Abstract

Resumen

Partiendo de la importancia de los SMS en nuestra sociedad como medio de comunicación instantánea, surge la necesidad de dotar de una cierta seguridad a este servicio de mensajería. Existen numerosas aplicaciones que ofrecen diversas soluciones para este problema, entre las que se ha elegido CryptoSMS por ser su código de libre distribución.

CryptoSMS es una aplicación para dispositivos móviles que permite a los usuarios mantener un medio de comunicación privado y fiable mediante mensajes de texto y restringido a un grupo conocido de personas.

El objetivo de este proyecto es estudiar el funcionamiento de dicha aplicación, así como de las tecnologías de las que hace uso, para introducir mejoras que nos permitan ofrecer una solución propia desarrollada bajo la plataforma Symbian.

Dado el carácter académico en el que se ha desarrollado este proyecto, se ha pretendido introducir al lector en el aprendizaje de nuevas tecnologías diseñadas para dispositivos móviles, tales como J2ME o Symbian, que han adquirido una especial relevancia en estos últimos años.

Abstract

Due to the importance of SMS in our society as a means of instantaneous communication there is a need to provide this service with some security.

Nowadays there are a lot of applications that offer several solutions to this problem, among them CryptoSMS has been chosen because of its free code.

CryptoSMS is a mobile application that allows users to keep a private and reliable communication through text messages that can be sent to a restricted group of people.

The aim of this project is to study the running of such applications as well as the technologies that have been used to introduce improvements that allow us to offer a special solution developed under the platform Symbian.

With regards to the academic nature witch this project has been developed in, we have tried to introduce the reader to new mobile technologies such as J2ME or Symbian, which have acquired a special significance in recent years.

3. Presentación

La mensajería SMS ha adquirido una entidad propia en el mundo de la telefonía móvil. Es una forma de comunicación rápida, menos intrusiva que una llamada telefónica puesto que no es necesario que el receptor esté disponible y la información que se transmite puede ser almacenada para su posterior consulta.

A pesar de las ventajas que ofrece, la mensajería SMS no está exenta de problemas de seguridad en ambos extremos de la comunicación, especialmente si se utiliza para intercambiar información confidencial. Los riesgos más destacados son:

- Los mensajes de texto son más fáciles de interceptar que las llamadas telefónicas. Los operadores de telefonía pueden registrar el contenido de los mensajes en busca de palabras clave que les permite, por ejemplo, el envío de publicidad personalizada que el usuario no ha pedido. El problema es que hoy en día existen, por precios muy asequibles en el mercado, dispositivos que hacen estas recepciones y pueden ser adquiridos por terceras personas interesadas en utilizar la información.

- Un mensaje de texto plano, además de interceptado, puede ser editado. Se pueden cambiar tanto el texto como los datos del emisor, no teniendo la seguridad absoluta de quién ha sido el emisor real ni su contenido original.

- Al ser almacenados en los teléfonos los mensajes, cabe la posibilidad de que terceras personas puedan consultar dichos mensajes en el propio teléfono, violando así la intimidad del usuario.

Por estas razones, surge la necesidad de dotar de una cierta seguridad a este servicio de mensajería. Para proveer esta protección se utilizan técnicas de criptografía que se verán más adelante.

Este proyecto de Sistemas Informáticos forma parte de un proyecto de mayor entidad cuyo objetivo es crear una aplicación en Symbian capaz del envío y recepción de mensajes SMS y MMS cifrados. Para ello, ha servido como base una aplicación desarrollada en Java (J2ME) llamada CryptoSMS (http://cryptosms.org/).

Se ha dividido el trabajo en 3 partes:

- envío y recepción de SMS/MMS en Symbian C++
- cifrado/descifrado de claves y mensajes en Symbian C++
- estudio y mejoras de la aplicación original (J2ME); sistema de gestión de la base de datos para el almacenamiento permanente de los contactos de la agenda mediante una base de datos en Symbian.

Este proyecto abarca esta última parte.

Después de hacer un estudio del funcionamiento de CryptoSMS y comprobar sus limitaciones, se propusieron las siguientes mejoras:

- El envío de MMS en vez de SMS: para ello se ha usado el API WMA que ofrece J2ME, que permite la comunicación mediante mensajes entre diferentes dispositivos.
- Integración de la agenda del teléfono en la de CryptoSMS: el API utilizado ha sido PIM (Personal Information Management).

Aparte de estas mejoras, se ha mantenido un contacto con los desarrolladores originales del proyecto con el fin de solucionar y/o mejorar ciertas limitaciones que ofrecía CryptoSMS.

A su vez, este estudio ha servido para conocer la estructura que debe seguir la aplicación implementada en Symbian C++.

Se ofrece también una descripción de las diferentes APIs o librerías que han sido empleadas a lo largo del desarrollo del proyecto en J2ME. A continuación se presenta el estudio de las limitaciones y las mejoras realizadas a la aplicación original.

Por último, también se ha desarrollado un sistema de gestión y almacenamiento de datos que ha sido empleada para la administración de los contactos de la aplicación implementada en Symbian C++.

4. Estudio de Aplicaciones de Cifrado

CryptoSMS es una aplicación J2ME que permite el intercambio de mensajes de texto cifrados entre dispositivos móviles usando los conceptos de criptografía asimétrica, usando en particular curvas elípticas.

La Criptografía de Curva Elíptica (CCE) es una variante de la criptografía asimétrica o de clave pública basada en las matemáticas de las curvas elípticas. Sus autores argumentan que la CCE puede ser más rápida y usar claves más cortas que los métodos antiguos — como RSA — al tiempo que proporcionan un nivel de seguridad equivalente.

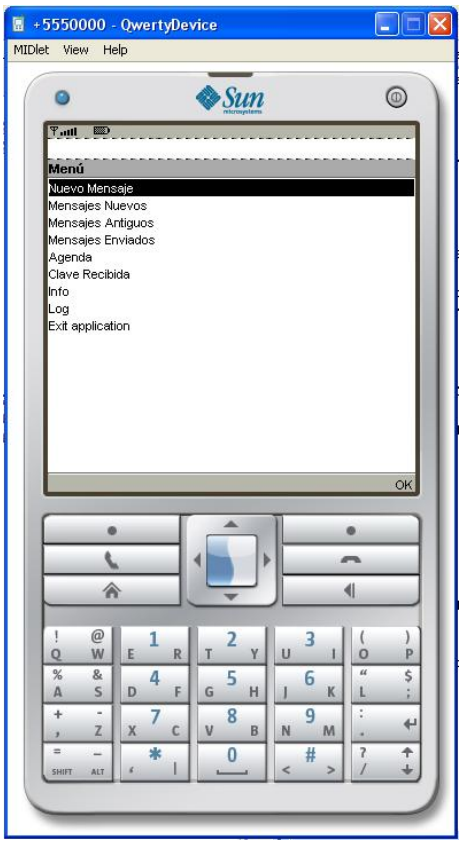
La criptografía asimétrica es el método criptográfico que usa un par de claves para el envío de mensajes. Una de las claves es *pública* y se puede entregar a cualquier persona; la otra clave es *privada* y el propietario debe guardarla de modo que nadie tenga acceso a ella. El remitente usa la clave pública del destinatario para cifrar el mensaje, y una vez cifrado, sólo la clave privada del destinatario podrá descifrar este mensaje.

Los sistemas de cifrado de clave pública o sistemas de cifrado asimétricos se inventaron con el fin de evitar por completo el problema del intercambio de claves de los sistemas de cifrado simétricos. Con las claves públicas no es necesario que el remitente y el destinatario se pongan de acuerdo en la clave a emplear. Todo lo que se requiere es que, antes de iniciar la comunicación secreta, el remitente consiga una copia de la clave pública del destinatario. Es más, esa misma clave pública puede ser usada por cualquiera que desee comunicarse con su propietario. Por tanto, se necesitarán sólo n pares de claves por cada n personas que deseen comunicarse entre sí.

4.1. Funcionamiento

La primera vez que se ejecuta CryptoSMS aparece una pantalla de configuración, en la que habrá que introducir el nombre, el número de teléfono y una contraseña. Esta contraseña se usa para acceder a la aplicación en posteriores ejecuciones, además de para generar el par de claves (pública y privada) del usuario.

Una vez arrancada la aplicación aparecerá el menú principal:

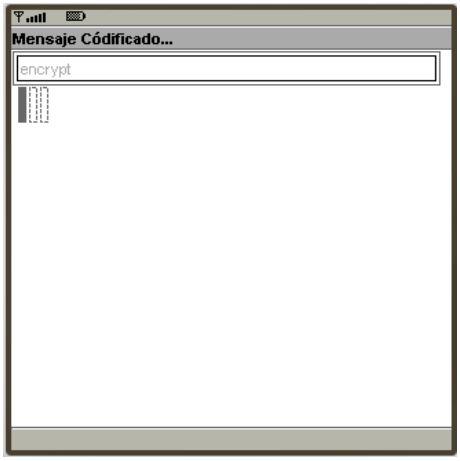


- Envío de un mensaje:

Para poder enviar un mensaje cifrado a otra persona es necesario poseer su clave pública. Para ello se debe haber recibido previamente un mensaje con dicha clave, que la aplicación reconocerá como tal.

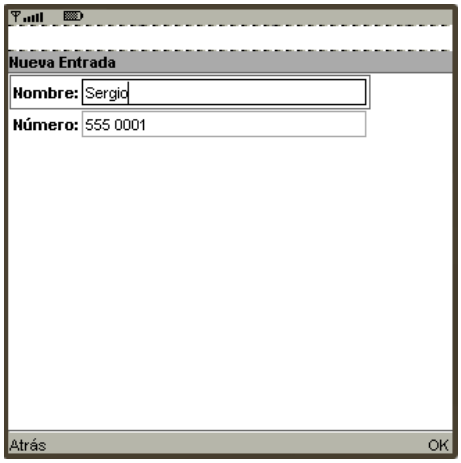
Una vez que se dispone de la clave pública del destinatario, se elige la opción "Nuevo mensaje". Aparecerá un editor de texto con capacidad para 100 caracteres donde se escribirá el contenido del mensaje.

Por último habrá que seleccionar el destinatario de la lista de contactos de cryptoSMS. En esta lista solo aparecerán aquellos contactos de los cuales se dispone de su clave pública. La aplicación procederá entonces al cifrado del mensaje con la clave pública del contacto elegido y se enviará.



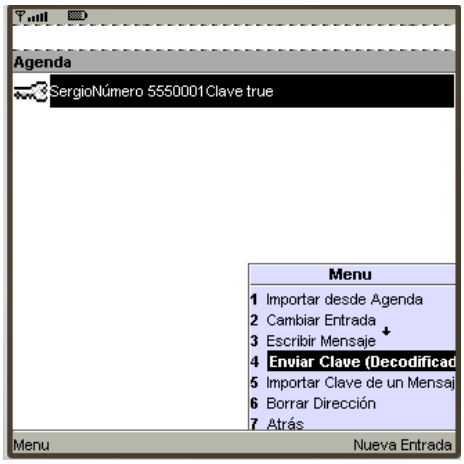
- Agenda:

CryptoSMS posee una agenda propia que almacena de cada contacto su nombre, su número de teléfono y su clave pública. Para añadir un nuevo contacto a esta agenda se elige la opción “Nueva entrada” en el menú “Agenda”. Se pedirá un nombre y un número de teléfono. Ahora solo queda añadir la clave asociada a este contacto.



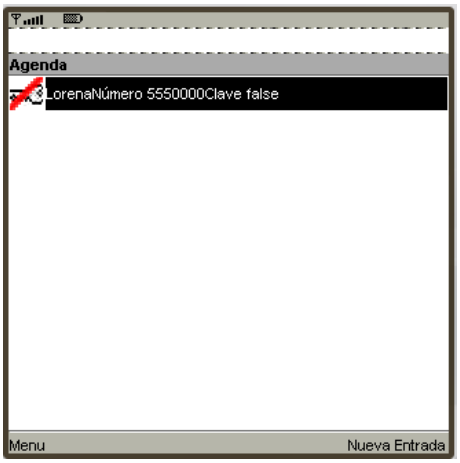
- Enviar/Importar clave:

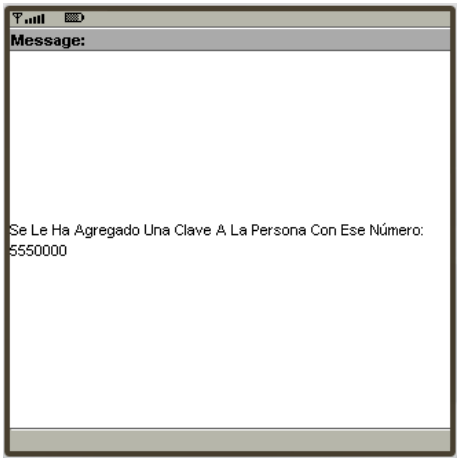
Para importar una clave es necesario haberla recibido previamente. El envío de la clave se realiza seleccionando un contacto de la agenda y eligiendo la opción “Enviar Clave”.



Una vez recibida la clave se selecciona el contacto al que se quiere asociar una clave y a continuación se elige la opción “Importar Clave”. CryptoSMS buscará en su agenda un número que coincida con el número de teléfono del que se ha recibido la clave seleccionada. En caso contrario mostrará un mensaje de error.

Cuando un contacto no tiene asociada ninguna clave, aparece un icono con una línea diagonal roja, que desaparecerá al añadir una clave a dicho contacto.





La aplicación se encarga de distinguir cada mensaje dependiendo de si se trata de una clave o de un texto cifrado.

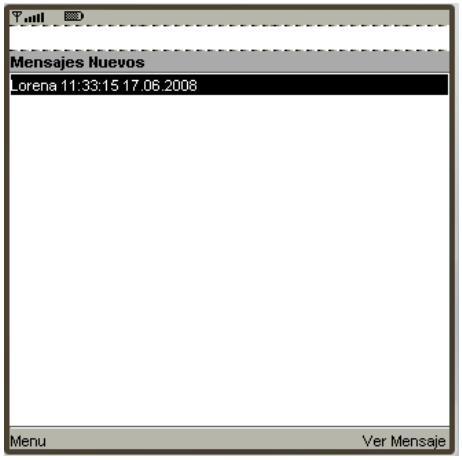
- Recibir un SMS:

Hay dos situaciones diferentes a la hora de recibir un SMS cifrado:

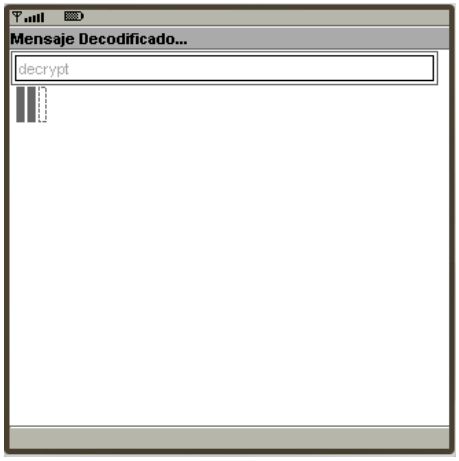
1.- CryptoSMS no está ejecutándose.
2.- CryptoSMS está ejecutándose.

Ambas situaciones son manejadas por CryptoSMS. En el primer caso el teléfono avisa de la recepción de un nuevo SMS que no se encontrará en la bandeja de entrada del teléfono ya que éste tiene como destino un puerto específico asociado a la aplicación. Al ejecutarse CryptoSMS este mensaje podrá encontrase dentro del menú "Nuevos mensajes".

En el segundo caso se notificará en la pantalla la llegada de un nuevo mensaje, al que se podrá acceder del mismo modo que en el caso anterior.



El mensaje entrante se podrá visualizar seleccionando la opción "Ver mensaje". Automáticamente el mensaje se descifrará usando la clave privada. Los mensajes que ya han sido leídos se almacenarán cifrados simétricamente empleando para ello la contraseña de acceso a la aplicación, puesto que descifrar usando este tipo de criptografía es más rápido que hacerlo con el cifrado asimétrico. Una vez leídos, los mensajes podrán encontrarse eligiendo la opción "Mensajes Antiguos".



4.2. Plataforma de desarrollo /Arquitectura

CryptoSMS es un MIDlet para terminales móviles desarrollado en Java usando la plataforma J2ME (Java Platform Micro Edition), que está especialmente diseñada para su uso en este tipo de dispositivos.

Se pueden distinguir dos partes, el uso de las librerías del Bouncy Castle para los aspectos criptográficos (ver apartado 4.2.3 sobre Bouncy Castle) y la gestión del intercambio de mensajes y de la agenda propia. A continuación la atención se centrará principalmente en esta última parte.

4.2.1. Arquitectura

En este apartado se procederá a la explicación de cada uno de los paquetes que forman parte de la aplicación.

- **Book:**

Se encarga de gestionar los diferentes registros de la base de datos (Record Store) que almacenan la información de los contactos, de los mensajes y de usuario del sistema (clave pública y privada, teléfono, contraseña, etc.).

- *Address*: contiene toda la información referente a un contacto, es decir, su clave pública (si la tiene), teléfono y un objeto de clase *Person*.

- *AddressPersonFilter*: implementa la interfaz RecordFilter, que se encargará de realizar una búsqueda eficiente de registros en un Record Store, devolviendo únicamente los registros que coincidan con un determinado patrón de búsqueda que es de tipo *Person.*

- *AddressService*: gestiona los registros que contienen la información personal de los contactos, esto es, crear, modificar, acceder y borrar estos registros de la base de datos.

- *AddressUrlFilter*: implementa la interfaz RecordFilter, que se encargará de realizar una búsqueda eficiente de registros en un Record Store, devolviendo únicamente los registros que coincidan con un determinado patrón de búsqueda que es de tipo *String* (teléfono).

- *CMessage*: todos los mensajes de la aplicación son de este tipo. Analiza los mensajes existentes y crea un objeto que contiene toda la información relacionada con los mensajes: tipo (si es mensaje de clave o si es un mensaje de texto cifrado), versión de la aplicación, destinatario, remitente, fecha, etc.

- *MessageComparator*: la ordenación de registros se realiza a través del interfaz RecordComparator. Se encarga de realizar la comparación entre los campos que deseemos, en este caso la fecha, para ordenar la lista de mensajes.

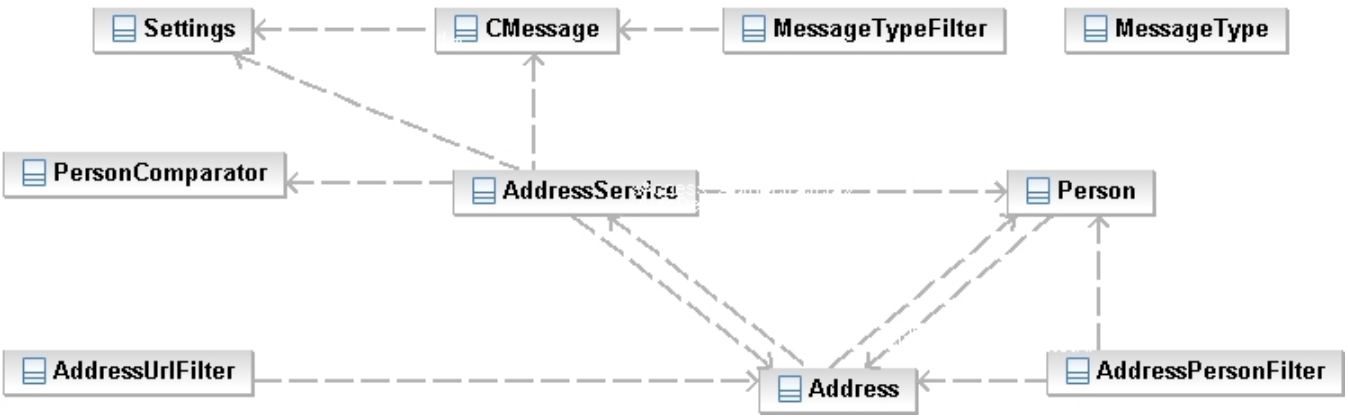
- *MessageType*: define las constantes que identifican el tipo de mensaje que es para poder manejarlos de forma adecuada para poder mostrarlos en los diferentes menús como mensajes recibidos, enviados o claves.

- *MessageTypeFilter*: implementa la interfaz RecordFilter, que se encargará de realizar una búsqueda eficiente de registros en un Record Store, devolviendo únicamente los registros que coincidan con un determinado patrón de búsqueda que es de tipo *byte* (tipo del mensaje).

- *Person*: contiene toda la información referente a una persona, es decir, su nombre y un atributo de tipo *Address*.

- *PersonComparator*: la ordenación de registros se realiza a través del interfaz RecordComparator. Se encarga de realizar la comparación entre los campos que deseemos, en este caso el nombre de las personas, para ordenar la lista de contactos.

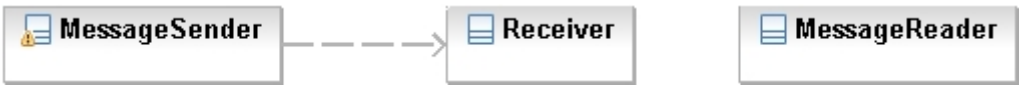
- *Settings*: guarda la información personal (nombre, teléfono y claves) en un registro. Este objeto es guardado en la base de datos Settings.



- **Connection:**

Se encarga de la creación, envío y recepción de mensajes. También inicializa el PushRegistry de la aplicación, que se encarga de escuchar por el puerto 16002 todos los SMS entrantes. Cuando un SMS llega a través de este puerto este paquete se encarga de guardarlo en el RecordStore.

- *MessageReader*: lee los mensajes del RecordStore.
- *MessageSender*: se encarga de crear el mensaje (de tipo TextMessage) cogiendo los atributos necesarios (destino, contenido…) y lo envía utilizando el método send de la clase MessageConnection. Tiene una cola de mensajes pendientes por enviar.
- *Receiver*: inicia el PushRegistry de la aplicación, que se encarga de escuchar por el puerto 16002 todos los SMS entrantes. Cuando un SMS llega a través de este puerto esta clase se encarga de guardarlo en el RecordStore.

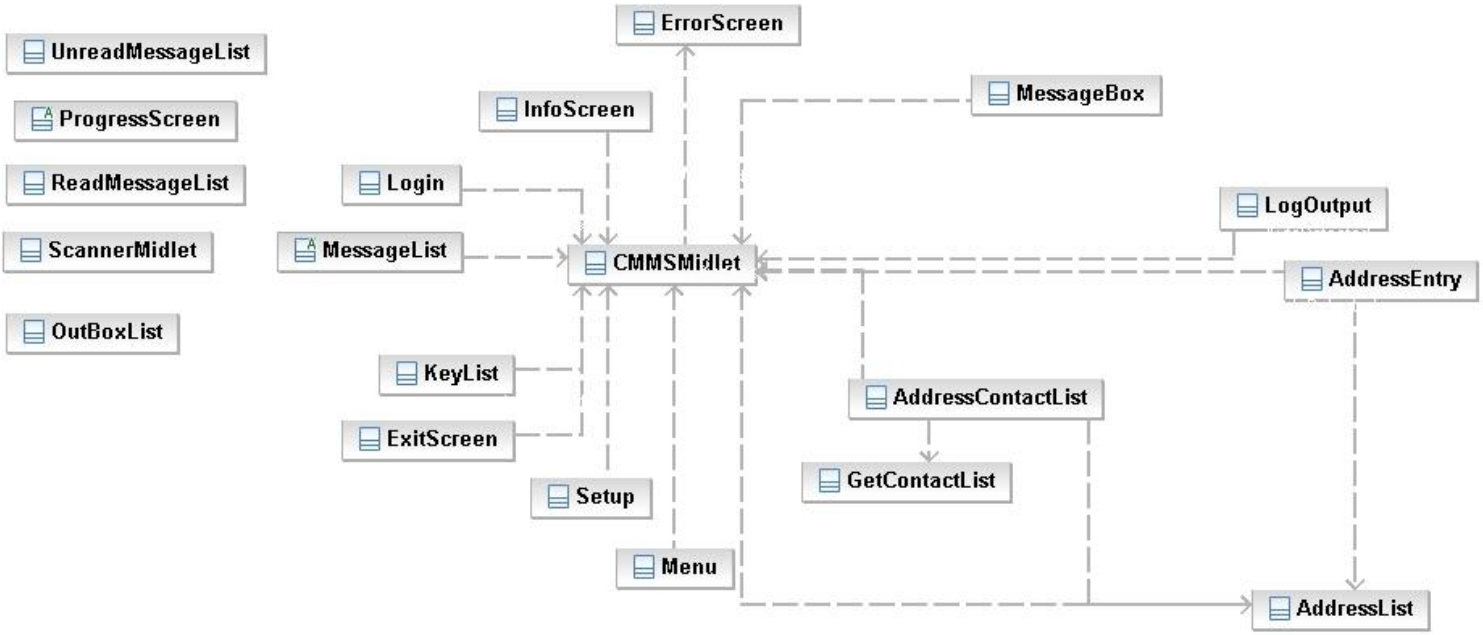


- **Gui:**

Contiene las clases que implementan los menús de la aplicación.

- *AddressEntry*: menú que pide nombre y teléfono de un contacto nuevo.
- *AddressList*: menú de la lista de contactos. Permite añadir o editar un contacto, enviar un SMS, importar o borrar una clave y enviar nuestra clave pública.
- *CryptoSMSMidlet*: es el MIDlet con la aplicación.
- *EraserMidlet*: resetea o borra todos los datos de la aplicación.
- *ErrorScreen*: muestra mensaje de error
- *ExitScreen*: pide una confirmación al usuario siempre que se produce una salida de la aplicación
- *InfoScreen*: muestra información de la aplicación tales como el nombre, el número de teléfono y el número del centro de mensajería.
- *KeyList*: menú de claves: permite importar o borrar.
- *Login*: login de la aplicación

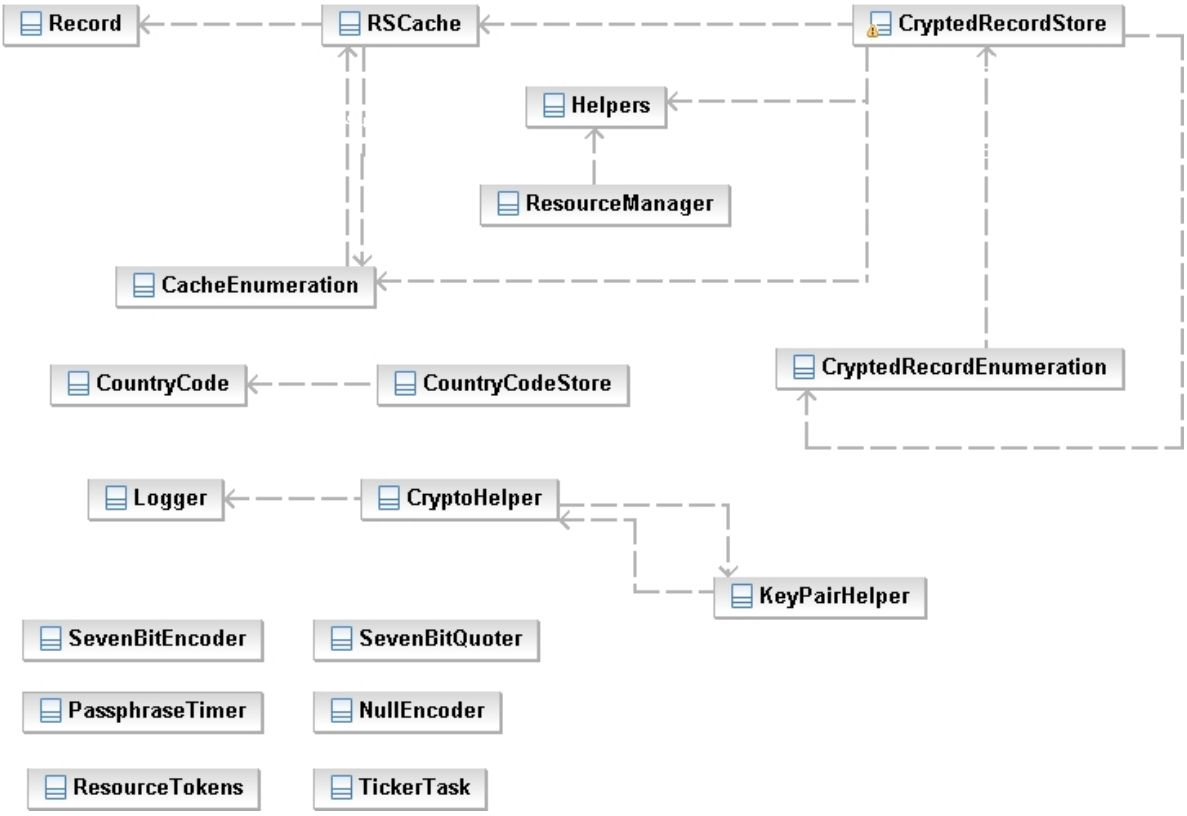
- *LogOutput*: muestra los mensajes almacenados en el log
- *Menu*: menú principal de la aplicación
- *MessageBox*: editor de mensajes.
- *MessageList*: menú que muestra una lista de mensajes.
- *OutBoxList*: hereda de MessageList e implementa los métodos de descifrado y borrado para los mensajes enviados.
- *ProgressScreen*: muestra barras de progreso.
- *ReadMessageList*: hereda de MessageList e implementa los métodos de descifrado y borrado para los mensajes recibidos.
- *Setup*: formulario que pide los datos del usuario.
- *UnreadMessageList*: hereda de MessageList e implementa los métodos de descifrado y borrado para los mensajes no leídos.



- **Helpers:**

  Interfaz entre Bouncy Castle y los paquetes propios de la aplicación.

  - *CacheEnumeration*: contiene la caché con los registros de la agenda y los filtros y comparadores que se usan con los registros.
  - *CountryCode*: gestión del código de país.
  - *CountryCodeStore*: vector que almacena los countrycodes.
  - *CryptedRecordEnumeration*: contiene la caché con los registros de los mensajes cifrados y los filtros y comparadores que se usan con ellos.
  - *CryptedRecordStore*: gestiona toda la información cifrada que se guarda en el teléfono. Se encarga del cifrado antes de guardar la información y del descifrado cuando se quiera acceder a dicha información.
  - *CryptoHelper*: genera el par de claves usando la criptografía de Curva Elíptica que proporciona la librería BouncyCastle.
  - *Helpers*: métodos diversos para conversiones a String.
  - *KeyPairHelper*: almacena el par de claves pública y privada.
  - *Logger*: implementa el log del teléfono que contiene la información relativa de los eventos que realiza la aplicación.
  - *NullEncoder*: asegura que no tengamos mensajes binarios de 0x00 bytes.
  - *PassphraseTimer*: temporizador que cierra la sesión cuando se está un determinado tiempo sin interactuar con la aplicación.
  - *Record*: define el tipo Registro para el almacenamiento de datos.
  - *ResourceManager*: se encarga de gestionar el idioma de la aplicación.
  - *ResourceTokens*: almacena las constantes que contienen el texto de los menús de la aplicación.
  - *RSCache*: caché con los registros que contienen los datos.
  - *SevenBitEncoder*: encoder / decoder de 8 a 7 bits.
  - *TickerTask*: timer para resetear el ticker.



4.2.2. Gestión del intercambio de mensajes

Como se ha comentado anteriormente, la primera vez que se lanza la aplicación se pide una contraseña. Ésta se usará tanto para crear el par de claves asimétricas como para guardar toda la información relevante de la aplicación que es almacenada en el teléfono mediante criptografía simétrica. Esto incluye los mensajes enviados y recibidos, la lista de contactos (nombre, teléfono y clave pública asociado al contacto) y la clave pública y privada del usuario. Estos datos se almacenan en un registro utilizando los mecanismos de almacenamiento proporcionados por RMS (Record Managent System).

Siempre que se inicia la aplicación y una vez que se ha comprobado que la contraseña introducida por el usuario es correcta, se procede a la recuperación del par de claves que se encuentran almacenadas en el teléfono. Esta contraseña se utiliza para descifrar esta información y a partir de este momento se encontrará en el atributo *_settings* de tipo *CryptedRecordStore* en la clase *AddressService* para su rápido acceso.

CryptoSMS utiliza el API WMA que ofrece J2ME para realizar el intercambio de mensajes entre dispositivos móviles.

Los mensajes que envía la aplicación son de tipo BinaryMessage (véase página 34). Hay que distinguir entre dos tipos de mensajes: aquellos que contienen una clave pública y aquellos que son mensajes con texto.

El contenido de los mensajes que se envían y reciben son objetos de la clase CMessage. Los atributos más relevantes de esta clase son *_type* y *data*. El atributo *data* almacena el contenido útil del mensaje mientras que *_type* permite distinguir cuando un mensaje es de de clave o de texto, y dentro de este segundo caso, si es un mensaje enviado (*ENCRYPTED_TEXT_OUTBOX*), recibido sin leer (*SYM_ENCRYPTED_TEXT_INBOX*) o recibido pero ya leído (*ENCRYPTED_TEXT_INBOX*). Esta distinción es necesaria por motivos que se detallarán más adelante. Para poder enviar y recibir correctamente un objeto de tipo CMessage, JAVA proporciona las clases *DataOutputStream* y *DataInputStream*. Estas clases están diseñadas para almacenar tipos complejos de datos, como son en este caso, objetos y hacer mucho más fácil la transmisión de flujos de bytes. Para ello están los métodos *prepareForTransport()* y *createForTransport()* en CMessage. El primero de ellos se encarga de preparar el mensaje para su envío y con el siguiente fragmento:

```
…
DataOutputStream dStrm = new DataOutputStream(bStrm);
dStrm.write(_version);
dStrm.write(_type);
dStrm.writeLong(timestamp);
dStrm.write(data);
```

El segundo de ellos recupera correctamente la información de un mensaje recibido a partir de un DataStream:

```
…
version=Dstrm.readByte();
type=Dstrm.readByte();
timestamp=Dstrm.readLong();
// data
int  len=Dstrm.available();
byte[]   payload=new byte[len];
Dstrm.read(payload);
```

Ahora ya se puede crear un CMessage con esta información y gestionarlo adecuadamente.

- Envío de mensajes:

Se diferencian dos casos distintos dependiendo de si se quiere enviar la clave pública o un mensaje de texto cifrado.

  - Envío de una clave pública:

Para enviar la clave pública a un contacto de la agenda, se crea un objeto de tipo CMessage. Éste objeto será el contenido del SMS que se enviará. Entre sus parámetros cabe destacar dos atributos:

1. *_type*: tipo de mensaje del que se trata. En este caso es un mensaje de envío de clave: “KEY”.
2. *data*: contiene la clave pública.

Este objeto se añade ahora a una cola de tipo CMessage llamada QueuedMessage. La clase enviará de manera asíncrona cada uno de los mensajes que haya en la cola, usando los métodos que ofrece WMA (véase página 33).

- Envío de un mensaje de texto:

El envío de un mensaje de texto se realiza de manera muy parecida al envío de una clave. Una vez que se dispone del texto que el usuario quiere enviar se realizan dos tareas: se utiliza la clave pública del receptor para cifrar el texto y enviárselo al receptor y cifrar utilizando criptografía simétrica usando la contraseña del usuario para posteriormente almacenarlo en la memoria del teléfono. El campo *_type* del mensaje que se envía contiene (*ENCRYPTED_TEXT_INBOX*) mientras que el otro es (*SYM_ENCRYPTED_TEXT_OUTBOX*). *ENCRYPTED_TEXT_INBOX* indica que se trata de un mensaje almacenado con el que es necesario usar la clave privada (del receptor, ya que es el mensaje enviado) para poder acceder a su contenido. *SYM_ENCRYPTED_TEXT_OUTBOX* indica por el contrario que el mensaje está almacenado y es necesario usar la contraseña para poder visualizar su contenido.

- Recepción de mensajes:

Cuando el MIDlet recibe cualquier tipo de mensaje se procede a su almacenamiento en un nuevo registro del RecordStore en la memoria del teléfono y se añade a la caché de la aplicación. De estas dos tareas se encarga la clase CryptedRecordStore. El sistema caché del que dispone la aplicación es muy simple y se usa para almacenar en la memoria del teléfono ese mensaje que acaba de ser recibido. De esta manera puede ser accedido más rápidamente sin necesidad de acceder a la memoria.

A la hora de mostrar una lista de mensajes, dependiendo de la opción del menú que haya seleccionado el usuario (mensajes no leídos, mensajes leídos, mensajes enviados o mensajes de claves), se mostrarán un tipo u otro del conjunto de todos los mensajes almacenados.

- Mensajes de claves:

Los mensajes de claves se muestran cuando el usuario selecciona la opción “Claves recibidas”. Como todos los mensajes se almacenan sin ninguna distinción, se accede uno por uno comprobando el atributo *_type* de cada uno de ellos:

```
RecordEnumeration e = _messages.enumerateCache(new
MessageTypeFilter(MessageType.KEY), null);
```

De aquellos que su tipo es “KEY” se muestra el número de teléfono al que pertenece la clave o, en el caso de que esté ya asociado a un contacto de la agenda, el nombre de dicho contacto:

```
if (address==null)
     this.append(number,null);
else
     this.append(address.getPerson().getName(),null);
```

Por último, para importar una clave, simplemente hay que seleccionarla y la aplicación tratará de encontrar el contacto al que pertenece con el método *findAdress4Url()*, comparando con cada elemento de la agenda:

```
Address a = findAdress4Url(url);
```

Si se encuentra, simplemente se le asocia el contenido del mensaje (que corresponde a su clave pública) al registro del contacto.

- Mensajes de texto:

Los mensajes de texto se almacenan del mismo modo que los de claves. Hay tres tipos de mensajes de texto: leídos (*SYM_ENCRYPTED_TEXT_INBOX*), no leídos (*ENCRYPTED_TEXT_INBOX*) y enviados (*SYM_ENCRYPTED_TEXT_OUTBOX*). *SYM_ENCRYPTED_TEXT_INBOX* son los mensajes recibidos en los que es necesario el uso de la contraseña para poder visualizar su contenido. Los otros dos tipos tienen un significado idéntico al explicado anteriormente.

Cuando se accede a un mensaje por primera vez, su atributo *_type* es *ENCRYPTED_TEXT_INBOX* e indica que hay que usar la clave privada del usuario para poder descifrarlo:

```
text1 = new String(CryptoHelper.decrypt(m.getData(),
gui.getSettings().getPair().getKeyPair()));
```

Una vez que se ha accedido correctamente al texto del mensaje, aparte de mostrarlo por pantalla al usuario, se procede a borrar dicho mensaje de RecordStore para almacenarlo de nuevo, pero esta vez usando el cifrado simétrico. La principal razón para realizar esta operación es que el descifrado mediante criptografía simétrica es mucho más rápido que con asimétrica, siendo este detalle de vital importancia en dispositivos móviles con una capacidad de cálculo limitada. El tipo de este mensaje pasa a ser *SYM_ENCRYPTED_TEXT_INBOX.*

Los otros dos tipos de mensajes, están cifrados con criptografía simétrica, y dependiendo de la opción del menú a la que acceda el usuario se mostrarán unos u otros.

4.2.3. Criptografía / Bouncy Castle

Bouncy Castle es un proyecto de software libre que pretende desarrollar una serie de librerías criptográficas libres y, entre otros, ofrece un proveedor para el JCE de java.

En la página web del proyecto: http://www.bouncycastle.org/ es posible descargar la versión actual del proveedor Bouncy Castle para distintas versiones de la máquina virtual de Java.

Ofrece un API (application programming interface) que permite:

- generación de claves (claves secretas y pares de claves pública y privada)
- cifrado simétrico (DES, 3DES, IDEA, etc.)
- cifrado con curva elíptica (ECIES)
- cifrado asimétrico (RSA, DSA, Diffie-Hellman, ElGamal...)
- funciones de resumen (MD5 y SHA1) y algoritmos MAC (Message Authentication Code)
- acuerdo de claves

Dentro de dicha librería, se usan las siguientes clases:

- *org.bouncycastle.crypto.AsymmetricCipherKeyPair:* una clase de mantenimiento para los pares de parámetros públicos/privados.

- *org.bouncycastle.crypto.BasicAgreement:* interfaz con los métodos init y calculateAgreement del método Diffie-Hellman. Éste método permite el intercambio secreto de claves entre dos partes que no han tenido contacto previo, utilizando un canal inseguro, y de manera anónima (no autenticada).

- *org.bouncycastle.crypto.BlockCipher:* cifra bloques de datos.

- *org.bouncycastle.crypto.BufferedBlockCipher:* clase para manejar los bloques encriptados mediante un buffer.

- *org.bouncycastle.crypto.CipherParameters:* parámetros de la encriptación.

- *org.bouncycastle.crypto.DataLengthException:* esta excepción se lanza si un buffer que tiene que almacenar datos es pequeño, o le hemos dado datos insuficientes para almacenar. Esta excepción se lanza antes que la excepción *ArrayOutOfBounds*.

- *org.bouncycastle.crypto.DerivationFunction:* interfaz general para las clases de derivación. Se usa para la generación de bytes.

- *org.bouncycastle.crypto.DerivationParameters:* interfaz de la clase básica de los parámetros usados para la derivación.

- *org.bouncycastle.crypto.Digest:* interfaz genérica para objetos que generan bytes aleatorios.

- *org.bouncycastle.crypto.ExtendedDigest:* Clase de apoyo a digest.

- *org.bouncycastle.crypto.InvalidCipherTextException:* excepción que se lanza cuando encontramos algo anormal en un mensaje.

- *org.bouncycastle.crypto.Mac:* interfaz para la implementación de los códigos de autenticación de los mensajes (MACs).

- *org.bouncycastle.crypto.agreement.ECDHBasicAgreement:* método que obtiene un valor secreto a partir de una clave privada y otra pública.

- *org.bouncycastle.crypto.digests.SHA256Digest:* implementación del algoritmo SHA-256 mediante FIPS 180-2.

- *org.bouncycastle.crypto.digests.GeneralDigest:* implementación básica del resumen de la familia MD4, tal y como se explica en el libro "Handbook of Applied Cryptography", páginas 344 – 347.

- *org.bouncycastle.crypto.digests.SHA1Digest:* implementación del algoritmo SHA-1 como se explica en "Handbook of Applied Cryptography", páginas 346 - 349. A parte del IV extra, la otra diferencia que tiene con MD5 es que la palabra que se procesa no tiene final.

- *org.bouncycastle.crypto.engines.AESFastEngine:* implementación del Motor AES (Rijndael), a partir de FIPS-197.

- *org.bouncycastle.crypto.engines.IESEngine:* clase de apoyo para construir encriptadores para el intercambio básico de mensajes en la aplicación. En resumen, procesa (encripta, desencripta) bloques de bits de los mensajes.

- *org.bouncycastle.crypto.generators.KDF2BytesGenerator:* construye un generador de KDF2 bytes.

- *org.bouncycastle.crypto.macs.HMac:* algoritmo para la autenticación del mensaje.

- *org.bouncycastle.crypto.params.ECDomainParameters:* clase para la creación de parámetros de las claves de curva elíptica.

- *org.bouncycastle.crypto.params.ECPublicKeyParameters:* parámetros de la clave pública en curva elíptica.

- *org.bouncycastle.crypto.params.ECPrivateKeyParameters:* parámetros de la clave privada en curva elíptica.

- *org.bouncycastle.crypto.params.KDFParameters:* parámetros de la Key Derivation Functions (KDF). Estas funciones sacan una o más keys a partir de un valor secreto.

- *org.bouncycastle.crypto.params.KeyParameter:* parámetros de la key.

- *org.bouncycastle.crypto.params.IESParameters:* parámetro para usar cifras en modo streaming. Es la clave de agreement (Diffie-Hellman) usada como base de la encriptación.

- *org.bouncycastle.crypto.params.IESWithCipherParameters:* clase base para las claves de curva elíptica.

- *org.bouncycastle.crypto.prng.RandomGenerator:* interfaz genérica para objetos que generan bytes aleatorios.

- *org.bouncycastle.crypto.prng.DigestRandomGenerator:* generación aleatoria basada en el "digest" con contador. Llamar a "addSeeMaterial" incrementará siempre la entropía del hash. El acceso interno al "digest" está sincronizado de manera que uno de estos puede ser compartido.

- *org.bouncycastle.crypto.prng.DigestRandomGenerator:* generación aleatoria basada en el "digest" con contador.

- *org.bouncycastle.crypto.prng.RandomGenerator:* interfaz genérica para objetos que generan bytes aleatorios.

- *org.bouncycastle.math.BigInteger:* clase que implementa los enteros grandes.

- *org.bouncycastle.math.ec.ECPoint:* clase base para puntos y curvas elípticas.

- *org.bouncycastle.math.ec.ECConstants:* clase de apoyo para los parámetros de las claves de curva elíptica.

- *org.bouncycastle.math.ec.ECCurve:* clase base para una curva elíptica.

- *org.bouncycastle.math.ec.ECFieldElement:* realiza operaciones matemáticas.

- *org.bouncycastle.security.SecureRandom:* implementación del SecureRandom especificada para una API con "poco peso", JDK 1.0 y J2ME. La generación aleatoria está basada en SHA1 con contador. Llama a setSEED, lo que incrementará la entropía de la tabla hash.

4.3. Plataforma

4.3.1. J2ME

La edición Java 2 Micro Edition fue presentada con el propósito de habilitar aplicaciones Java para pequeños dispositivos. En esta presentación, lo que realmente se enseñó fue una primera versión de una nueva Java Virtual Machine (JVM), denominada (Kilo Virtual Machine, debido a que requiere sólo unos pocos Kilobytes de memoria para funcionar), que podía ejecutarse en dispositivos Palm.

Se puede entender J2ME como una versión simplificada de J2SE. SUN separó estas dos versiones ya que J2ME estaba pensada para dispositivos con limitaciones de proceso y capacidad gráfica.

Un entorno de ejecución determinado de J2ME se compone de una selección de:

a) Máquina virtual.
b) Configuración.
c) Perfil.
d) Paquetes Opcionales.

Máquinas Virtuales J2ME

Las implementaciones tradicionales de JVM son, en general, muy pesadas en cuanto a memoria ocupada y requerimientos computacionales. J2ME define varias JVMs de referencia adecuadas al ámbito de los dispositivos electrónicos que, en algunos casos, suprimen algunas características con el fin de obtener una implementación menos exigente.

Se pueden usar dos posibles configuraciones: CLDC (Connected Limited Device Configuration) y CDC (Connected Device Configuration). Cada una de las posibles configuraciones requiere su propia máquina virtual. La VM (Virtual Machine) de la configuración CLDC se denomina KVM y la de la configuración CDC se denomina CVM.

- **KVM**

KVM proviene de Kilobyte (haciendo referencia a la baja ocupación de memoria, entre 40Kb y 80Kb). Se trata de una implementación de Máquina Virtual reducida y especialmente orientada a dispositivos con bajas capacidades computacionales y de memoria. La KVM está escrita en lenguaje C, y fue diseñada para permitir una alta portabilidad y una gran modularidad intentando que fuese lo más completa y rápida posible sin sacrificar las características para las que fue diseñada.

Sin embargo, esta baja ocupación de memoria hace que posea algunas limitaciones con respecto a la clásica *Java Virtual Machine* (JVM):

- No hay soporte para tipos en coma flotante. Esta limitación está presente porque los dispositivos carecen del hardware necesario para estas operaciones.

- No existe soporte para JNI (*Java Native Interface*) debido a los recursos limitados de memoria.

- No existen cargadores de clases (*class loaders*) definidos por el usuario. Sólo existen los predefinidos.

- No se permiten los grupos de hilos o hilos *daemon*. Cuándo queramos utilizar grupos de hilos utilizaremos los objetos *Colección* para almacenar cada hilo en el ámbito de la aplicación.

- No existe la finalización de instancias de clases. No existe el método *Object.finalize()*.

- No hay referencias débiles.

- Limitada capacidad para el manejo de excepciones debido a que esto depende en gran parte de las APIs de cada dispositivo por lo que son éstos los que controlan la mayoría de las excepciones.

- Reflexión o, lo que es lo mismo, los objetos pueden obtener información de otros objetos en tiempo de ejecución como, por ejemplo, los archivos de clases cargados o sus campos y métodos.

Aparte de la no inclusión de estas características, la verificación de clases merece un comentario aparte. El verificador de clases estándar de Java es demasiado grande para la KVM. De hecho es más grande que la propia KVM y el consumo de memoria es excesivo, más de 100Kb para las aplicaciones típicas. Por esta razón los dispositivos que usen la configuración CLDC y KVM introducen un algoritmo de verificación de clases en dos pasos.

- **CVM**

La CVM (Compact Virtual Machine) ha sido tomada como Máquina Virtual Java de referencia para la configuración CDC y soporta las mismas características que la Máquina Virtual de J2SE. Está orientada a dispositivos electrónicos con procesadores de 32 bits de gama alta y en torno a 2Mb o más de memoria RAM. Las características que presenta esta Máquina Virtual son:

- Sistema de memoria avanzado.
- Tiempo de espera bajo para el recolector de basura.
- Separación completa de la VM del sistema de memoria.
- Recolector de basura modularizado.
- Portabilidad.
- Rápida sincronización.
- Ejecución de las clases Java fuera de la memoria de sólo lectura (ROM).
- Soporte nativo de hilos.
- Baja ocupación en memoria de las clases.
- Proporciona soporte e interfaces para servicios en Sistemas Operativos de Tiempo Real.
- Conversión de hilos Java a hilos nativos.
- Soporte para todas las características de Java2 v1.3 y librerías de seguridad, referencias débiles, Interfaz Nativa de Java (JNI), invocación remota de métodos (RMI), Interfaz de depuración de la Máquina Virtual (JVMDI).

Configuraciones

Una configuración es un conjunto de APIs Java que permiten desarrollar aplicaciones para un grupo de dispositivos. Éstas APIs describen las características básicas, comunes a todos los dispositivos.

- **Configuración de dispositivos con conexión, CDC**

CDC usa una Máquina Virtual Java similar en sus características a una de J2SE, pero con limitaciones en el apartado gráfico y de memoria del dispositivo. Ésta Máquina Virtual es la que hemos visto como CVM.

La CDC está basada en J2SE v1.3 e incluye varios paquetes Java de la edición estándar. Las peculiaridades de la CDC están contenidas principalmente en el paquete javax.microedition.io, que incluye soporte para comunicaciones http y basadas en datagramas.

- **Configuración de dispositivos limitados con conexión, CLDC**

Como se ha comentado con anterioridad, CLDC está orientado a dispositivos con ciertas restricciones, dadas en algunos casos por el uso de la KVM, necesaria al trabajar con la CLDC debido a su pequeño tamaño.

La CLDC aporta las siguientes funcionalidades a los dispositivos:

- Un subconjunto del lenguaje Java y todas las restricciones de su Máquina Virtual.
- Un subconjunto de las bibliotecas Java del núcleo.
- Soporte para E/S básica.
- Soporte para acceso a redes.
- Seguridad.

Perfiles

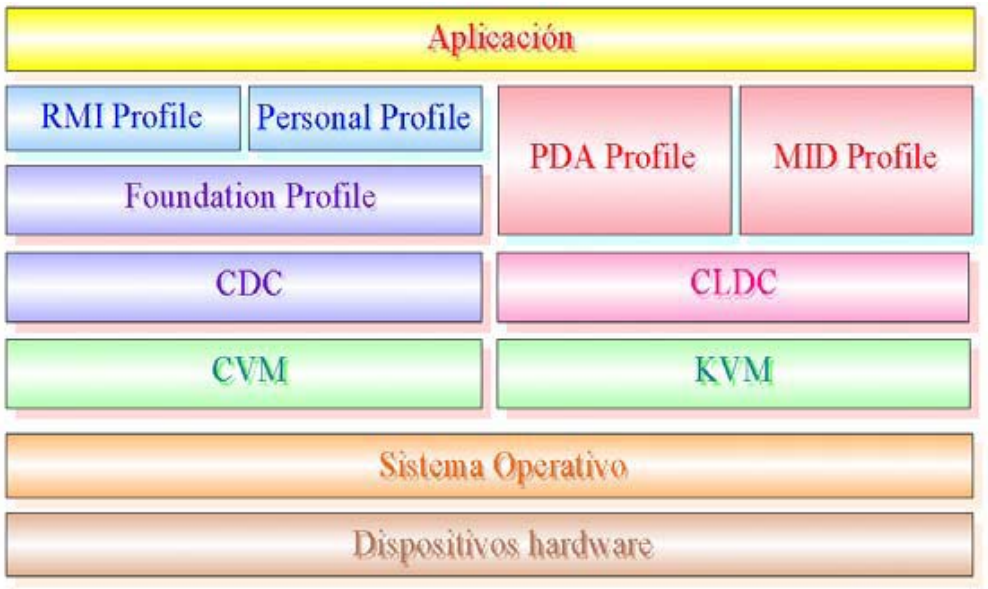
Un perfil es un conjunto de APIs orientado a un ámbito de aplicación determinado. Los perfiles identifican un grupo de dispositivos por la funcionalidad que proporcionan (electrodomésticos, teléfonos móviles, etc.) y el tipo de aplicaciones que se ejecutarán en ellos. Las librerías de la interfaz gráfica son un componente muy importante en la definición de un perfil. Aquí se pueden encontrar grandes diferencias entre interfaces, desde el menú textual de los teléfonos móviles hasta los táctiles de los PDAs.

El perfil establece unas APIs que definen las características de un dispositivo, mientras que la configuración hace lo propio con una familia de ellos. Esto hace que a la hora de construir una aplicación se cuente tanto con las APIs del perfil como de la configuración. Hay que tener en cuenta que un perfil siempre se construye sobre una configuración determinada. De este modo, se puede pensar en un perfil como un conjunto de APIs que dotan a una configuración de funcionalidad específica.

Existen unos perfiles construidos sobre la configuración CDC y otros que sobre la CLDC. Para la configuración CDC existen los siguientes perfiles:

- *Foundation Profile*: define una serie de APIs sobre la CDC orientadas a dispositivos que carecen de interfaz gráfica.

- *Personal Profile*: es un subconjunto de la plataforma J2SE v1.3, y proporciona un entorno con un completo soporte gráfico AWT.
- *RMI Profile*: requiere una implementación del *FoundationProfile*. El perfil RMI soporta un subconjunto de las APIs J2SE v1.3 RMI.



Para la configuración CLDC existen:

- *PDA Profile*: pretende abarcar PDAs de gama baja
- *Mobile Information Device Profile* (MIDP): al igual que CLDC fue la primera configuración definida para J2ME, MIDP fue el primer perfil definido para esta plataforma.

Las aplicaciones que implementan utilizando MIDP reciben el nombre de *MIDlets*. Se dice así que un MIDlet es una aplicación Java realizada con el perfil MIDP sobre la configuración CLDC.

Los MIDLets

Los MIDlets son aplicaciones creadas usando la especificación MIDP. Están diseñados para ser ejecutados en dispositivos con poca capacidad gráfica, de cómputo y de memoria. En estos dispositivos no se dispone de línea de comandos donde poder ejecutar las aplicaciones que se quieran, si no que reside en él un software que es el encargado de ejecutar los MIDlets y gestionar los recursos que éstos ocupan (gestor de aplicaciones).

- **El Gestor de Aplicaciones**

El gestor de aplicaciones o AMS (*Application Management System*) es el software encargado de gestionar los MIDlets. Este software reside en el dispositivo y es el que permite ejecutar, pausar o destruir las aplicaciones J2ME. El AMS realiza dos grandes

funciones: gestiona el ciclo de vida de los MIDlets y es el encargado de controlar los estados por los que pasa el MIDlet mientras está en la memoria del dispositivo, es decir, en ejecución.

- **Ciclo de vida de un MIDlet**

El ciclo de vida de un MIDlet pasa por 5 fases: descubrimiento, instalación, ejecución, actualización y borrado. El AMS es el encargado de gestionar cada una de estas fases de la siguiente manera:

1. Descubrimiento: consiste en seleccionar a través del gestor de aplicaciones la aplicación a descargar.

2. Instalación: en esta fase el gestor de aplicaciones controla todo el proceso informando al usuario tanto de la evolución de la instalación como de si existiese algún problema durante ésta.

3. Ejecución: mediante el gestor de aplicaciones se podrá iniciar la ejecución de los MIDlets. En esta fase, el AMS tiene la función de gestionar los estados del MIDlet en función de los eventos que se produzcan durante esta ejecución.

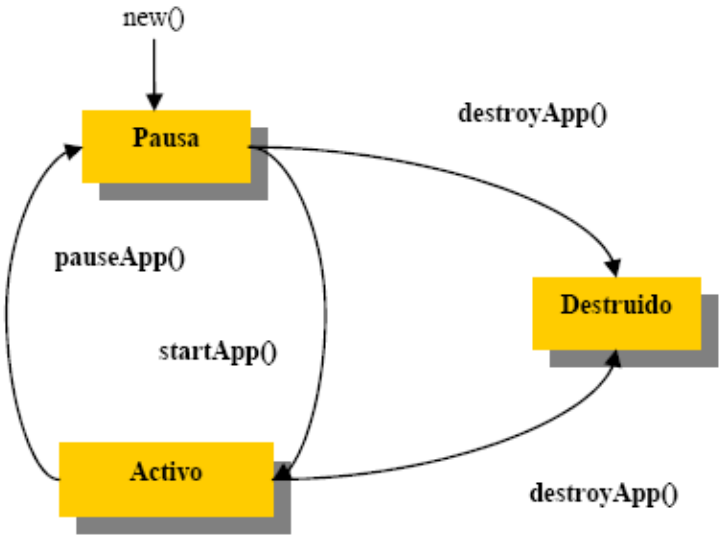
4. Actualización: el AMS tiene que ser capaz de detectar después de una descarga si el MIDlet descargado es una actualización de un MIDlet ya presente en el dispositivo. Si es así, nos tiene que informar de ello.

5. Borrado: en esta fase el AMS es el encargado de borrar el MIDlet seleccionado del dispositivo.

- **Estados de un MIDlet en fase de ejecución**

El AMS es el encargado de controlar los estados del MIDlet durante su ejecución. Durante ésta el MIDlet es cargado en la memoria del dispositivo y es aquí donde puede transitar entre 3 estados diferentes: *Activo, En pausa* y *Destruido*.

Cuando un MIDlet comienza su ejecución, ésta pasa al estado "*Activo*" pero el gestor de aplicaciones debe ser capaz de cambiar el estado de la aplicación en función de los eventos externos al ámbito de ejecución de la aplicación que se vayan produciendo. En este caso, el gestor de aplicaciones interrumpiría la ejecución del MIDlet sin que se viese afectada la ejecución de éste y lo pasaría al estado de "*Pausa*" para atender al evento externo. Una vez que se termine de trabajar con el MIDlet y se salga de él, éste pasará al estado de "*Destruido*" dónde será eliminado de la memoria volátil del dispositivo que es usada para la ejecución de aplicaciones.



Como se observa en el diagrama, un MIDlet puede cambiar de estado mediante una llamada a los métodos *MIDlet.startApp()*, *MIDlet.pauseApp()* o *MIDlet.destroyApp()*. El gestor de aplicaciones cambia el estado de los MIDlets haciendo una llamada a cualquiera de los métodos anteriores. Un MIDlet también puede cambiar de estado por sí mismo.

Durante una ejecución típica, en primer lugar se realiza la llamada al constructor del MIDlet pasando éste al estado de "*Pausa*". El AMS por su parte crea una nueva instancia del MIDlet. Cuándo el dispositivo está preparado para ejecutar el MIDlet, el AMS invoca al método *MIDlet.startApp()* para entrar en el estado de "*Activo*". El MIDlet entonces, ocupa todos los recursos que necesita para su ejecución. Durante este estado, el MIDlet puede pasar al estado de "*Pausa*" por una acción del usuario, o bien, por el AMS que reduciría en todo lo posible el uso de los recursos del dispositivo por parte del MIDlet.

Tanto en el estado "*Activo*" como en el de "*Pausa*", el MIDlet puede pasar al estado "*Destruido*" realizando una llamada al método *MIDlet.destroyApp()*. Esto puede ocurrir porque el MIDlet haya finalizado su ejecución o porque una aplicación prioritaria necesite ser ejecutada en memoria en lugar del MIDlet. Una vez destruido el MIDlet, éste libera todos los recursos ocupados.

- **Clase MIDlet**

La aplicación debe extender a esta clase para que el AMS pueda gestionar sus estados y tener acceso a sus propiedades.

- **Estructura de los MIDlets**

Hay que decir que los MIDlets, al igual que los *applets* carecen de la función main(). Aunque existiese, el gestor de aplicaciones la ignoraría por completo.

Un MIDlet tampoco puede realizar una llamada a *System.exit()*. Una llamada a este método lanzaría la excepción *SecurityException*.

Los MIDlets tienen la siguiente estructura:

```
import javax.microedition.midlet.*

public class MiMidlet extends MIDlet{

     public MiMidlet() {
/* Éste es el constructor de clase. Aquí debemos inicializar
nuestras variables. */
     }

     public startApp(){
/* Aquí incluiremos el código que queremos que el MIDlet ejecute
cuándo se active. */
     }

     public pauseApp(){
/* Aquí incluiremos el código que queremos que el MIDlet ejecute
cuándo entre en el estado de pausa (Opcional). */
     }

     public destroyApp(){
/* Aquí incluiremos el código que queremos que el MIDlet ejecute
cuándo sea destruido. */
     }
}
```

Record Managent System

MIDP proporciona un mecanismo a los MIDlets que les permite almacenar datos de forma persistente para su futura recuperación. Este mecanismo está implementado sobre una base de datos basada en registros que llamaremos Record Management System o RMS (Sistema de gestión de registros).

- **Conceptos Básicos**

El sistema de gestión de registros o RMS nos permite almacenar información entre cada ejecución de nuestro MIDlet. Esta información será guardada en el dispositivo en una zona de memoria dedicada para este propósito. La cantidad de memoria y la zona asignada para ello dependerán de cada dispositivo. El RMS está implementado en una base de datos basada en registros.

Los MIDlets son los encargados de crear los Record Stores que quedan almacenados en el dispositivo y pueden ser accedidos por cualquier MIDlet que pertenezca a la misma suite.

Un Record Store, tal como su nombre indica, es un almacén de registros. Estos registros son la unidad básica de información que utiliza la clase RecordStore para almacenar datos.

Cada uno de estos registros está formado por dos unidades:

- Un número identificador de registro (*Record ID*) que es un valor entero que realiza la función de clave primaria en la base de datos.
- Un array de bytes que es utilizado para almacenar la información deseada.

Además de un nombre, cada Record Store también posee un número de versión que se actualiza conforme vayamos modificando el registro y una marca temporal.

- **Operaciones con Record Stores**

Una vez creado o abierta la comunicación con el Record Store, podemos leer, escribir, modificar o borrar registros a nuestro gusto. Para ello, usaremos los métodos de la clase RecordStore que se ven en la siguiente tabla:

Método	Descripción
int addRecord(byte[] datos, int offset, int numBytes)	Añade un registro al Record Store
void deleteRecord(int id)	Borra el registro "id" del Record Store
int getNextRecordId()	Devuelve el siguiente "id" del registro que se vaya a insertar
byte[] getRecord(int id)	Devuelve el registro con el identificador "id"
int getRecord(int id, byte[] buffer, int offset)	Devuelve el registro con el identificador "id" en "buffer" a partir de "offset"
int getRecordSize(int id)	Devuelve el tamaño del registro "id"
void setRecord(int id, byte[] datoNuevo, int offset, int tamaño)	Sustituye el registro "id" con el valor de "datoNuevo"

La clase RecordStore proporciona algunas interfaces que facilitan el trabajo a la hora de manipular registros, como *RecordEnumeration*, *RecordFilter* o *RecordComparator*. Básicamente, estas interfaces nos ayudan a la hora de navegar por los registros de un Record Store, realizar búsquedas en ellos y ordenaciones.

Existe además otra interfaz que permite capturar eventos que se produzcan a la hora de realizar alguna acción en un Record Store. La interfaz *RecordListener* es la encargada de recoger estas acciones.

4.3.2. WMA

Introducción a WMA

WMA, son las siglas de Wireless Message API, una extensión de la especificaciones del CLDC y MIDP para el envío, recepción y gestión de SMS desde MIDlets.

A pesar de ser una extensión opcional, la gran mayoría de los terminales la llevan instalada y lista para ser usada desde nuestras aplicaciones J2ME.

WMA es una **especificación y no una implementación**. Su implementación dependerá del terminal y del protocolo de comunicación que use (GSM, CDMA, etc.).

El API está compuesto exclusivamente de interfaces ubicadas bajo el paquete `javax.wireless.messaging`. Estas interfaces son:

javax.wireless.messaging.Message: Define la funcionalidad genérica de todos los tipos de mensajes. Permite:
- Especificar el destinatario del mensaje. `public void setAddress(String addr)`
- Obtener el emisor del mensaje. `public String getAddress()`
- Obtener la fecha de envío del mensaje. `java.util.Date getTimestamp()`

javax.wireless.messaging.TextMessage: Representa a un mensaje de texto. Hereda la funcionalidad de `javax.wireless.messaging.Message` añadiendo los métodos `public void setPayloadText(String data)` y `public String getPayloadText()` para especificar u obtener los datos del mensaje.

javax.wireless.messaging.BinaryMessage: Representa a un mensaje binario. Hereda la funcionalidad de `javax.wireless.messaging.Message` añadiendo los métodos `public void setPayloadData(byte[] data)` y `public byte[] getPayloadData()` para especificar u obtener los datos del mensaje.

javax.wireless.messaging.MessageListener: Oyente de mensajes entrantes. Esta interfaz es útil en `javax.wireless.messaging.MessageConnection` que funcionan en modo servidor.

javax.wireless.messaging.MessageConnection: Interfaz a través de la cual se realiza el envío y la recepción de mensajes.

Tipos de mensajes

WMA 2.0 define cuatro tipos o representaciones para los mensajes. Además, define un `MessagePart` que es parte de los `MultipartMessage` y que es usado para los mensajes multimedia.

- **La Interfaz Message**

La interfaz `javax.wireless.messaging.Message` es la base de todos los mensajes que se usan con WMA, un Message es lo que se envía y se recibe. En algunos aspectos, un Message es similar a un datagrama, tiene código, una dirección de destino y un payload.

La interfaz especifica métodos para acceder y modificar las direcciones fuente y destino de los mensajes.

WMA 2.0 define tres subinterfaces de Message:

`BinaryMessage:` es para los mensajes binarios cortos como los SMS.
`TextMessage:` es para los mensajes cortos de texto, los SMS.

`MultipartMessage:` es para los mensajes multimedia, los MMS.

- **La Interfaz BinaryMessage**

La subinterfaz representa un mensaje con un payload binario, típico de los SMS binarios. Un payload se codifica usando datos de 8-bit, permitiendo 140 bytes por segmento o 133 si se usa un número de puerto.

Esta interfaz declara métodos para acceder y modificar el payload binario como un array de bytes:

```
byte[] getPayloadData();
void setPayloadData(byte[] bytes);
```

Los métodos de acceder y modificar la dirección del mensaje y para acceder a su timestamp se heredan todos de Message.

- **La Interfaz TextMessage**

La subinterfaz `TextMessage` representa un mensaje con un payload de texto, típico de los SMS de texto. Esta interfaz proporciona métodos para acceder y modificar el texto como una instancia de String:

```
String getPayloadText();
void setPayloadText(String data);
```

La implementación por debajo es la responsable de codificar y decodificar adecuadamente el String al formato apropiado para que el mensaje de texto se pueda enviar y recibir.

Como en los `BinaryMessage,` los métodos de acceder y modificar la dirección del mensaje y su timestamp se heredan de Message.

- **La Interfaz MultipartMessage**

La subinterfaz representa un mensaje que consiste en varias partes, típico de un mensaje multimedia MMS. La interfaz define un contenedor de uno o más `MessagePart`, y proporciona métodos para manejar las direcciones del emisor y el receptor, la cabecera de los mensajes, la identificación ID del primer mensaje y las partes del mensaje:

```
boolean addAddress(String type, String address);
void addMessagePart(MessagePart messagePart) throws
SizeExceededException;
String getAddress();
String[] getAddresses(String type);
String getHeader(String headerField);
MessagePart getMessagePart(String contentID);
MessagePart[] getMessageParts();
String getStartContentId();
String getSubject();
boolean removeAddress(String type, String address);
void removeAddresses();
void removeAddresses(String type);
boolean removeMessagePart(MessagePart messagePart);
boolean removeMessagePartId(String contentID);
boolean removeMessagePartLocation(String contentLocation);
void setAddress(String address);
```

```
void setHeader(String headerField, String headerValue);
void setStartContentId(String contentID);
void setSubject(String subject);
```

`MultipartMessage` sobrescribe los métodos de acceder y modificar las direcciones del mensaje que hereda de `Message`.

Los `MultipartMessage` siguen el formato standard de los emails.

La interfaz `MultipartMessage` representa el mensaje multimedia y sus cabeceras, mientras que una instancia de la clase `MessagePart` representa cada parte individual.

- **La clase MessagePart**

Como su nombre indica, representa una parte de un mensaje. Además de varias constructoras, esta clase proporciona métodos para recuperar el contenido y la información sobre el contenido:

```
MessagePart(byte[] contents, int offset, int length, String
mimeType, String contentId, String contentLocation, String
encoding) throws SizeExceededException;
MessagePart(byte[] contents, String mimeType, String contentId,
String contentLocation, String encoding) throws
SizeExceededException;
MessagePart(java.io.InputStream is, String mimeType, String
contentId, String contentLocation, String encoding) throws
IOException, SizeExceededException;
public byte[] getContent();
public InputStream getContentAsStream();
public String getContentID();
public String getContentLocation();
public String getEncoding();
public int getLength();
public String getMIMEType();
```

Se puede construir un `MessagePart` desde un array de bytes o un `java.io.InputStream`.

Conexiones

Las conexiones en WMA se basan en la interfaz `MessageConnection`, que es una subinterfaz de `javax.microedition.io.Connection.`

La interfaz `MessageConnection` define método para crear `TextMessages`, `BinaryMessages`, y `MultipartMessages`, un método para calcular el número de segmentos de protocolo que se necesitan para mandar un mensaje, métodos para recibir y enviar mensajes y un método para fijar el oyente de esta conexión:

```
Message newMessage(String type);
Message newMessage(String type, String address);
int numberOfSegments(Message message);
Message receive() throws IOException, InterruptedIOException;
void send(Message message) throws IOException,
InterruptedIOException;
void setMessageListener(MessageListener messageListener) throws
IOException;
```

Además, la interfaz define constantes de tipo `String`, que es lo que se pasa al método `newMessage()` para identificar el tipo de `Message` que se va a crear:

```
String TEXT_MESSAGE = "text";
String BINARY_MESSAGE = "binary";
String MULTIPART_MESSAGE = "multipart";
```

Cuando se crea un `MessageConnection` siempre hay que usar una de estas constantes.

Para crear la conexión se usa el método `javax.microedition.io.Connector.open()` y para cerrarla se usa el método `javax.microedition.io.Connection.close()`. Se pueden tener varios `MessageConnection` abiertos simultáneamente.

Los MessageConnection se pueden crear de dos formas, como una conexión cliente o como una conexión servidor. Una conexión cliente solo puede enviar mensajes mientras que una conexión servidor puede enviar y recibir.

Hay que especificar la conexión como cliente o servidor como un URL:
`scheme://address-part` `[params]`
donde:

- *`scheme`* es el protocolo a usar
- *`address-part`* es la dirección especifica del protocolo para las conexiones cliente o servidor.
- *`params`* es un parámetro opcional de la forma `x=y` .

El *`scheme`* para crear un `MessageConnection` no es específico de un solo protocolo. Algunos de los protocolos que WMA define son:

- `sms` para mensajes cortos. SMS es bidireccional, se pueden crear, enviar y recibir mensajes y soporta tanto la conexión cliente como la conexión servidor.
- `mms` para mensajes multimedia. Como los SMS, los MMS son bidireccionales y soportan tanto la conexión cliente como la conexión servidor.

Crear y cerrar un MessageConnection

Para crear un `MessageConnection` se llama al método `Connector.open()`, pasándole una URL válida para WMA.

Ejemplo:

```
public static final String SERVER_URL = "sms://:" + PORT;

connection = (MessageConnection) Connector.open(SERVER_URL);
log.write("register " + SERVER_URL);
connection.setMessageListener(this);
```

Para cerrar una conexión, se llama al método `close`(). Este método primero desregistra cualquier oyente de `message` y luego cierra la conexión.

Crear y enviar mensajes

Para crear un mensaje se llama al método `newMessage()` y para enviar un mensaje se usa el método `send()`. Hay que destacar que `newMessage()` devolverá un mensaje incluso si la conexión ha sido cerrada.

Se pueden crear 3 tipos de mensajes:

TextMessages:

```
final public void sendTextMessage(MessageConnection mc,
String msg,String url) {
    TextMessage tmsg = null;
    tmsg = (TextMessage)
         mc.newMessage(MessageConnection.TEXT_MESSAGE);
    tmsg.setPayloadText(msg);
    sendTextMessage(mc, tmsg, url);
}
```

BinaryMessages:

```
final public void sendBinaryMessage(MessageConnection mc, byte[] msg,
                String url) {
    BinaryMessage bmsg;
    bmsg = (BinaryMessage)
         mc.newMessage(MessageConnection.BINARY_MESSAGE);
    bmsg.setPayloadData(msg);
    sendMessage(mc, bmsg, url);
}
```

MultipartMessages:

```
final public void sendMultipartMessage(mc, MultipartMessage msg,
                subject, url) {
    sendMessage(mc, msg, url);
}
```

Estos métodos asumen que el `TextMessage` , el `BinaryMessage` o el `MultipartMessage` han sido creados con anterioridad.

Para los MultipartMessages se puede definir un método que tenga como parámetro de entrada un array de `MessagePart` y crea y envía el `MultipartMessage`. Antes de enviar, el método tiene que añadir al `MultipartMessage` cada `MessagePart` individualmente.

Para construir un `MessagePart` de un array de bytes o un `java.io.InputStream`, hay tres constructores disponibles.

```
MessagePart(byte[] contents, int offset, int length, String
mimeType, String contentId, String contentLocation, String
encoding) throws SizeExceededException;
MessagePart(byte[] contents, String mimeType, String contentId,
String contentLocation, String encoding) throws
SizeExceededException;
MessagePart(java.io.InputStream contents, String mimeType,
String contentId, String contentLocation, String encoding)
throws IOException, SizeExceededException;
```

Los dos primeros permiten crear el `MessagePart` de un array de bytes, mientras que el último permite crearlo partiendo de un `InputStream`. Si el tamaño del contenido del mensaje ronda los 10KB hay que usar el último constructor.

Mensajes entrantes

Hay que tener en cuenta que solo el modo servidor de los `MessageConnection` puede recibir mensajes. Hay dos maneras de recibir mensajes:

1) Síncrona: tiene un hilo que espera a los mensajes entrantes.

2) Asíncrona: tiene una notificación en la aplicación que avisa cuando llegan los mensajes nuevos a la aplicación.

Procesado de mensajes

Una vez que el mensaje ha sido leído llamando al método `messageConnection.receive()`, entonces está listo para ser procesado. El tipo que devuelve `receive()` es un tipo genérico `Message` y antes de procesar el mensaje se debe descubrir si su tipo es un `BinaryMessage`, un `MultipartMessage`, o un `TextMessage`.

Push Registry

El Push Registry de MIDP 2.0 permite a los MIDlets que se lancen automáticamente sin el inicio por parte del usuario. El Push Registry permite que una conexión de red entrante o una alarma despierten el MIDlet.

Los MIDlets pueden ser activados por conexiones entrantes de WMA si la implementación que hay por debajo lo soporta. Antes de que un MIDlet pueda ser activado, se debe registrar con el Push Registry usando un registro estático o dinámico. El registro estático se hace poniendo una entrada en el JAD o en el fichero MANIFEST. El registro dinámico se hace en tiempo de ejecución, usando el objeto del API del Push Registry `registerConnection()`.

Se puede descubrir si el MIDlet ha sido activado usando el método `listConnections()`.

4.3.3. PIM

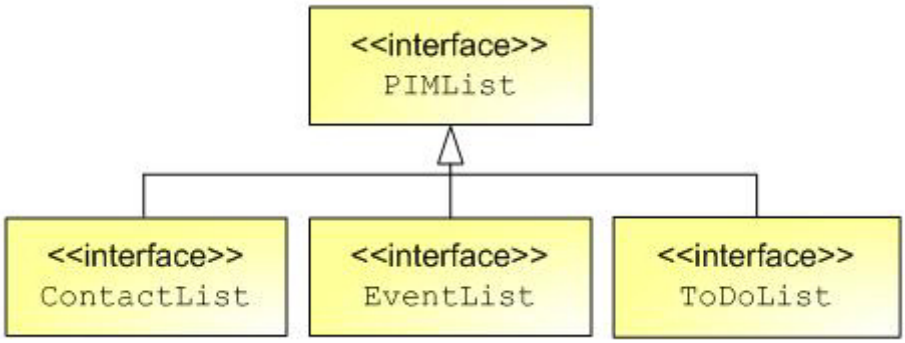
PIM (Personal Information Management) se encarga de gestionar de manera electrónica todos los tipos de información personal que los usuarios quieren manejar tales como calendario de citas, la lista de contactos o la lista de tareas pendientes. Esta

información se suele almacenar de manera permanente en el dispositivo y suele ser accedida por el usuario mediante una o más aplicaciones dedicadas específicamente a ello.

- **El paquete opcional PIM:**

El principal objetivo del PIM API es permitir a los todos los dispositivos J2ME acceder de manera semejante a la información personal. Objetivos básicos de este paquete opcional son:

- Permitir el acceso a información personal desde cualquier formato, ya sea desde el propio dispositivo, dispositivos extraíbles o a través de la red.
- Soportar la importación y exportación de la agenda, el calendario y la lista de tareas pendientes.
- Dar seguridad en el uso y acceso de este API.
- El PIM API define tres tipos de información PIM, conocidos como listas PIM (o PIM ítem):
- Listas de contactos, que contiene los nombres, direcciones, números de teléfonos.
- Listas de eventos, las cuales guardan recordatorios y otro tipo de fechas
- Listas de tareas pendientes.



4.4. Mejoras

4.4.1. Propuestas

Después del estudio de la aplicación, se hicieron varias propuestas para su mejora:

Cambio de SMS a MMS

La aplicación usa SMS para mandar tanto los mensajes de claves como los de texto. Puesto que el límite de los SMS está en 160 caracteres y los mensajes mandados

por la aplicación usan una cabecera de 60 caracteres, esto limita a solo 100 los caracteres disponibles, lo que provoca el envío de múltiples mensajes SMS (2 para la clave y 4 para un SMS de menos de 100 caracteres).

El uso de MMS soluciona este problema ya que envía un único mensaje tanto para la clave como para cualquier texto sea cual sea su tamaño, siempre que no supere los 250 Kb.

Con esta modificación se reducen costes para el usuario y se evitan posibles errores en el envío de varios mensajes.

Importar contactos de la agenda del teléfono a la agenda de la aplicación

CryptoSMS tiene una agenda para los contactos totalmente independiente de la agenda del propio teléfono. Esto obliga al usuario a introducir de nuevo en la agenda de la aplicación todos los contactos en los que se tenga interés. Para evitar esto, se planteó la opción de poder consultar la propia agenda del móvil e importar los datos necesarios (nombre y teléfono) a la agenda de CryptoSMS. Este cambio ofrece la ventaja de que el usuario no tiene que volver a copiar sus contactos, con seleccionarlos es suficiente ya que así quedan añadidos.

Se estuvo barajando también la posibilidad de eliminar por completo la agenda de CryptoSMS y usar únicamente la del teléfono pero finalmente esta opción fue desechada porque su implementación dependía del Sistema Operativo del móvil en concreto y la aplicación en Java perdía toda posibilidad de poderse utilizar en cualquier móvil.

Envío de imágenes cifradas

Al usar MMS también existe la posibilidad de mandar imágenes cifradas. Esto es interesante ya que se evita la posible intercepción de imágenes confidenciales por terceros.

Intercambio de claves por Bluetooth

En la aplicación el envío de claves se hace mediante un mensaje MMS igual que los de texto con la excepción de que dichos mensajes llevan una cabecera especial que hace que se puedan diferenciar de los demás. Parece interesante que el intercambio de claves pueda ser también vía Bluetooth. De esta manera aseguramos que las claves públicas son enviadas por el usuario interesado y no hay posibilidad de manipulación externa de ellas.

Firma digital

La posibilidad de añadir firma digital no supone un gran problema de implementación. De esta manera se conseguiría que el receptor pudiera verificar que el

autor del mensaje es quien dice quien es. Para ello, el emisor solo necesitaría cifrar el mensaje primero con su clave privada y después con la clave pública del receptor. Aunque esto pareciera una opción muy interesante, hay que destacar que estamos manejando la aplicación en dispositivos con capacidades de cómputo muy reducidas. Tener que aplicar dos funciones de cifrado aumentaría el coste en tiempo tanto en su codificación como en su decodificación.

Finalmente se llevaron a cabo las dos primeras propuestas cuya implementación está explicada a continuación.

4.4.2. Implementación

Cambio de SMS a MMS

- ***Receiver.java:***

Receiver.java (CRYPTOSMS ORIGINAL)

```
public static final String PORT = "16002";
public static final String SERVER_URL = "sms://:" + PORT;
```

saveMessage

```
if (msg instanceof TextMessage) {
...
log.write("incoming: "+((TextMessage)msg).getPayloadText());
CMessage cmsg = CMessage.createFromTransport((TextMessage)msg);
```

Receiver.java (CRYPTOSMS MODIFICADO)

```
public static final String PORT = "org.cryptosms.gui.CMMSMidlet";
public static final String SERVER_URL = "mms://:" + PORT;
```

saveMessage (cambiar los TextMessage por los MultipartMessage)

```
if (msg instanceof MultipartMessage) {
...
String msgCont= new
String((((MultipartMessage)msg).getMessageParts())[0].getContent());
log.write("incoming: "+msgCont);
CMessage cmsg = CMessage.createFromTransport((MultipartMessage)msg);
```

La clase Receiver declara dinámicamente el PushRegistry de la aplicación. Se define como cliente y servidor del protocolo MMS. Para ello se crea una conexión usando el protocolo MMS por el puerto del nombre del MIDlet "org.cryptosms.gui.CMMSMidlet". En este punto surgió un problema porque el anterior nombre del MIDlet sobrepasaba los 32 caracteres a los que está limitado, por lo que hubo que renombrar la aplicación.

El método *saveMessage()* es llamado cuando el método `notifyIncomingMessage` detecta la llegada de un nuevo mensaje. Se ha cambiado el código para que la aplicación cree un objeto de tipo CMessage que guarde la información contenida en el MMS, que cambia su tipo a MultipartMessage.

Para el rellenar el log de la aplicación se coge la información del primer MessagePart del MMS que es la que se ha establecido para almacenar el texto del mensaje.

- ***MessageSender.java***

MessageSender.java (CRYPTOSMS ORIGINAL)

```
private static final String PROTOCOL = "sms://";

run
if (connection != null) {
     TextMessage binary = (TextMessage)
      connection.newMessage(MessageConnection.TEXT_MESSAGE);
     binary.setAddress(address);
     String message=msg.getPayload();
     log.write("send: "+message);

     binary.setPayloadText(message);

     log.write("message: " + address+" Length: "+message.length());
     midlet.setTicketText("message sent to "+address);
     connection.send(binary);
}
```

MessageSender.java (CRYPTOSMS MODIFICADO)

```
private static final String PROTOCOL = "mms://";

run
if (connection != null) {
     MultipartMessage binary = (MultipartMessage)

     connection.newMessage(MessageConnection.MULTIPART_MESSAGE);
     binary.setSubject("MMS Text");
     binary.setAddress(address);
    //cogemos el contenido del mensaje
     String message=msg.getPayload();
     log.write("send: "+message);

     MessagePart textPart;
     String mimeType = "text/plain";
     String encoding = "UTF-8";//"ISO-8859-1";
     byte[] contents = message.getBytes(encoding);
     textPart = new MessagePart(contents, 0,
     contents.length,mimeType,"id",null, encoding);

     MessagePart[] msgParts;
     msgParts = new MessagePart[1];
     msgParts[0] = textPart;
```

```
        binary.addMessagePart(msgParts[0]);

        log.write("message: " + address+" Length: "+message.length());
        midlet.setTicketText("message sent to "+address);

        connection.send(binary);
}
```

La clase MessageSender es la encargada de crear el MMS con todos los datos necesarios para su envío.

Se crea un objeto de tipo MessagePart que va a contener los datos del mensaje. Para ello se define como texto plano (text/plain) con una codificación UTF-8, ya que no es necesario utilizar otras opciones multimedia.

La codificación elegida ha sido UTF-8 porque es la que se utiliza en toda la aplicación y no se puede combinar varias codificaciones diferentes.

A continuación se añade el objeto creado al MultipartMessage y se añade la dirección y el asunto del mensaje.

- ***CMessage.java***

CMessage.java (CRYPTOSMS ORIGINAL)

```
import javax.wireless.messaging.TextMessage;
```

createFromTransport

```
public static CMessage createFromTransport(TextMessage msg) throws
Exception {
     if (msg != null) {
…
```

CMessage.java (CRYPTOSMS MODIFICADO)

```
import javax.wireless.messaging.MessagePart;
import javax.wireless.messaging.MultipartMessage;
import javax.wireless.messaging.TextMessage;
```

createFromTransport

```
public static CMessage createFromTransport(MultipartMessage msg)
throws
Exception {

if (msg != null) {

          MessagePart[] messageParts = msg.getMessageParts();
          MessagePart messagePart = messageParts[0];
          byte[] content = messagePart.getContent();
          String text = new String(content,"UTF-8");
…
```

En la clase CMessage se ha añadido código al método *createFromTransport()*, que es el encargado de convertir el MultipartMessage en un CMessage, el tipo de los mensajes que gestiona la aplicación.

Se coge el contenido del primer y único MessagePart y se convierte a String. En este punto hubo problemas en el descifrado de los mensajes porque no estaba especificada la codificación UTF-8 y por defecto usaba ISO-8859-1.

Integración de la agenda del teléfono

En CryptoSMS se ha modificado la lista de contactos para poder añadir a la agenda propia del CryptoSMS contactos desde la agenda del teléfono de una manera cómoda y rápida. De estos contactos solo es interesante el nombre y el número de teléfono.

- **Apertura de la lista de contactos:**

```
PIM pim = PIM.getInstance();
ContactList clist;
try {
   clist = (ContactList) pim.openPIMList(PIM.CONTACT_LIST,
PIM.READ_ONLY);
}
```

De esta manera se accede a la lista de contactos del teléfono móvil. Como no se necesita modificar ningún tipo de información y por motivos de seguridad, se accede en modo de sólo lectura.

- **Recibimos toda la información de los contactos:**

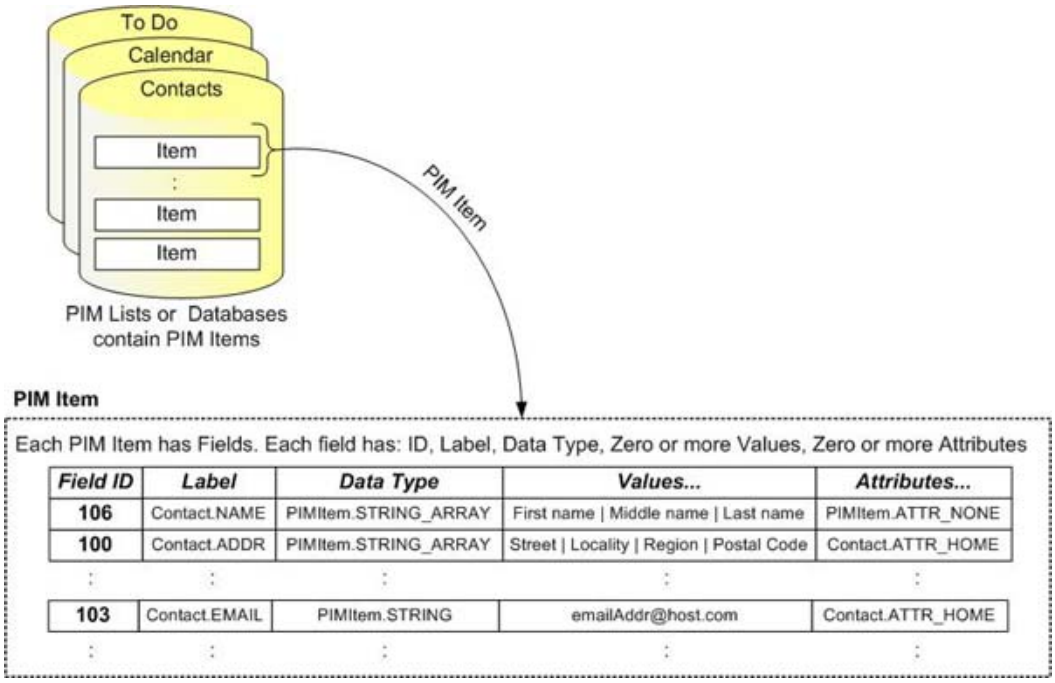
```
try {
        Enumeration cont = clist.items();
} catch (PIMException e) {
        e.printStackTrace();
}
```

Clist devuelve un tipo enumerado con todos los contactos de la agenda.

- **Acceso a los contactos seleccionando su nombre y número de teléfono para almacenarlos en vectores:**

```
Vector nombres=new Vector();
Vector telfs=new Vector();
While (cont.hasMoreElements())
     {
          Contact c = (Contact) cont.nextElement();
          this.append( c.getString(105 , 0), null);
          nombres.addElement(c.getString(105, 0));
          telfs.addElement(c.getString(115,0));
}
```

Las listas PIM son agrupaciones de campos. Los campos son específicos para el tipo de lista PIM. Por ejemplo, la lista PIM de la agenda de contactos (ContactList) contiene campos tales como nombre, apellidos, teléfonos fijos y móviles, direcciones de correo, etc.:



Los campos de una lista PIM son identificados por el campo ID (Field ID). Estos IDs son constantes que están definidos por la lista ContactList, lista Event o lista ToDo.

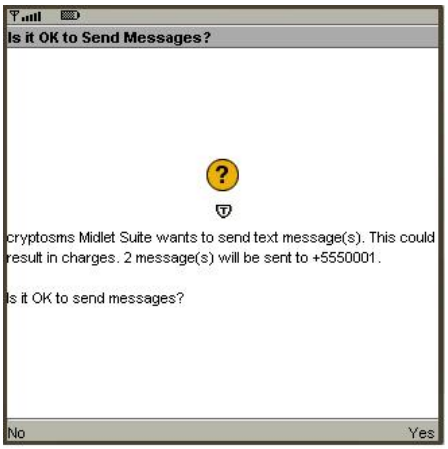
Para poder acceder al nombre y al teléfono hay que usar los Field IDs 105 y 115 respectivamente.

4.4.3. Resultados

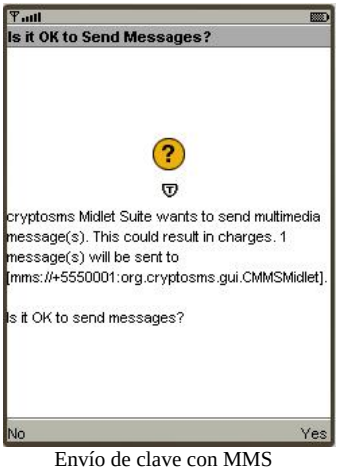
Para evaluar los resultados de las mejoras introducidas se realizaron diversas pruebas, empleando tanto emuladores como terminales móviles (Nokia E61), de las que se obtuvieron las siguientes conclusiones:

Cambio de SMS a MMS

El envío de una clave usando la versión original da lugar a dos mensajes de texto mientras que, utilizando la versión mejorada, se envía un único mensaje multimedia.



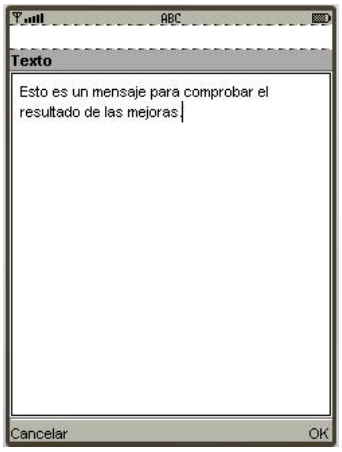
Envío de clave con SMS



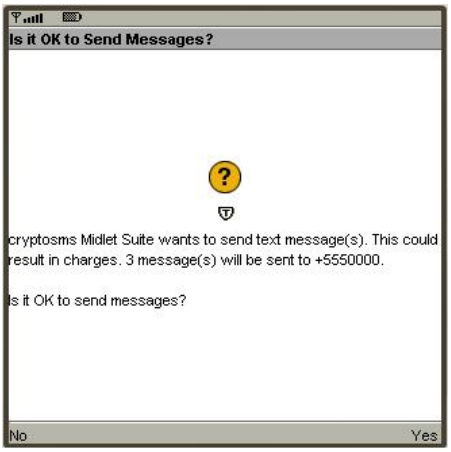
Envío de clave con MMS

Para enviar un mensaje de texto se han realizado pruebas utilizando diferentes tamaños de mensaje.

Por ejemplo, si se quiere enviar el siguiente texto:



Utilizando CryptoSMS original se envían tres SMS, sin embargo, gracias a la mejora de los mensajes multimedia, se envía un único MMS.



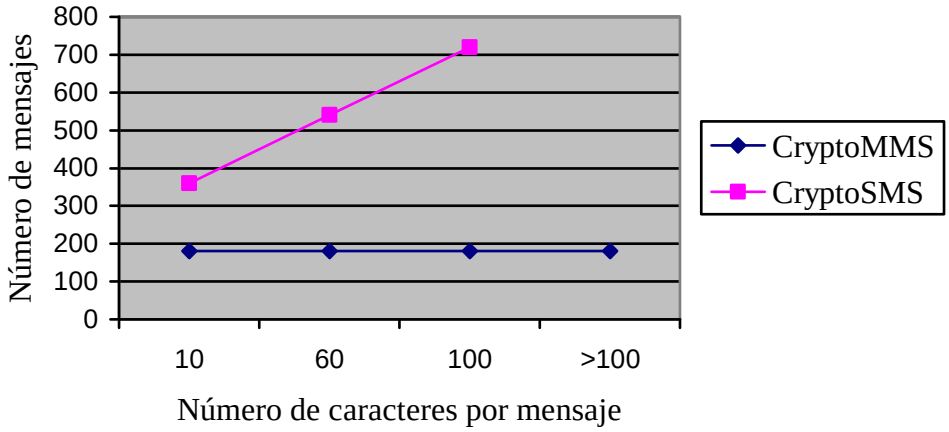
Envío de mensaje con SMS

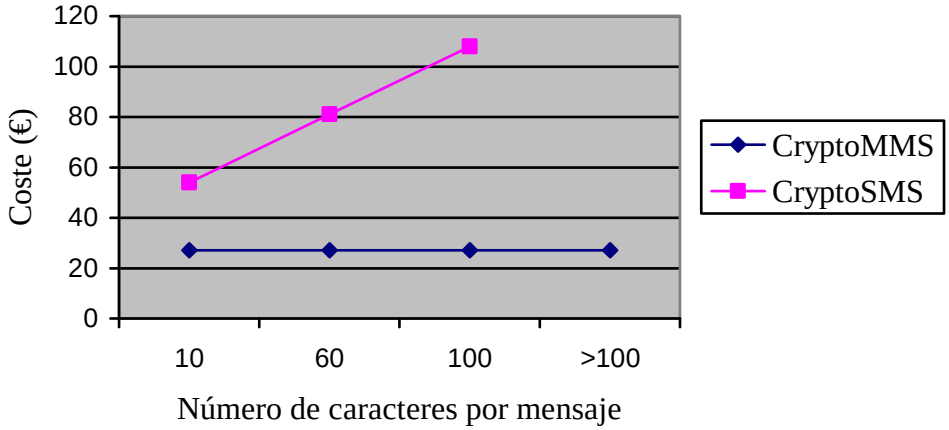
Envío de mensaje con MMS

La siguiente tabla muestra la comparativa entre la longitud del texto a enviar y el número de mensajes que se utiliza para ello.

Número caracteres	SMS	MMS
1	2	1
60	3	1
100	4	1
>100	No se permite	1

Suponiendo que se mandan 6 mensajes diarios de alrededor de 90 caracteres, a un coste 0.15 € por SMS o MMS de texto enviado, pasados 30 días obtenemos los siguientes resultados:





Como se puede observar en las gráficas, la mejora introducida reduce el número de mensajes enviados, lo que supone un ahorro en el coste para el usuario. Otra de las ventajas es que se elimina la restricción del envío de mensajes de menos de 100 caracteres que limitaba CryptoSMS, permitiendo enviar mensajes de mayor longitud sin aumentar el coste. Se reduce el riesgo de que ocurra un error durante el envío,

perdiéndose alguna de las partes del mensaje cifrado, provocando la pérdida total del mensaje al no poder reconstruirse correctamente en el receptor.

Por último, cabe destacar que esta mejora ofrece la infraestructura necesaria para implementar, en un futuro, código para el intercambio de contenido multimedia cifrado, tales como imágenes, sonidos, video, etc.

Importar contactos de la agenda del teléfono a la agenda de la aplicación

Esta mejora aumenta la manejabilidad de la aplicación. Permite el acceso a la agenda de contactos del teléfono desde CryptoSMS para agregarlos a la agenda interna de éste.

Si se quiere enviar un mensaje a un contacto que no está en la agenda interna de CryptoSMS, se tiene que proceder de la siguiente manera:

1. Cerrar la aplicación.
2. Acceder a la agenda del teléfono.
3. Buscar el contacto.
4. Apuntar/memorizar el número.
5. Iniciar de nuevo CryptoSMS.
6. Agregar a la agenda el nuevo contacto.
7. Enviar.

Con esta mejora no es necesario realizar todos y cada uno de estos pasos, ya que se importa directamente el contacto desde la agenda del teléfono a la de CryptoSMS, con el consiguiente ahorro de tiempo.

5. Gestión de la Información en dispositivos Móviles para Symbian (DBMS)

Introducción:

La aplicación tiene la necesidad de almacenar de forma permanente en el dispositivo los datos de los contactos: nombre, teléfono y la clave pública asociada al contacto. La manera más cómoda y eficiente para realizar esta tarea es mediante una base de datos. Symbian ofrece el API DBMS (contenido en la librería *edbms.dll*) para la gestión de las bases de datos.

Para el funcionamiento de la base de datos es necesario crearla especificando las tablas que la componen y los atributos de cada una de ellas.

Symbian DBMS:

Los **sistemas de gestión de base de datos** (*Database management system*, abreviado **DBMS**) son un tipo de software muy específico, dedicado a servir de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan. Se compone de un lenguaje de definición de datos, un lenguaje de manipulación de datos y un lenguaje de consulta.

Symbian OS DBMS ofrece mecanismos para la creación y el mantenimiento de bases de datos, implementando un fiable y seguro acceso a estas usando sentencias SQL o mediante métodos nativos.

Se trata pues de una potente y ligera base de datos que implementa las funcionalidades de añadir, buscar, acceder, actualizar y borrar además de la especificación para el manejo SQL, DDL y DML

DBMS: Estructura y elementos:

Symbian OS DBMS es una base de datos relacional. Está representada por una jerarquía de elementos. Las tablas son contenedores de filas, que a su vez están compuestas por campos.

Para su almacenamiento permanente usa la funcionalidad que ofrece Symbian OS en el manejo de ficheros: File Server, Permanent File Stores y Streams.

Creación de una Base de Datos:

Existen dos APIs para la creación de la base de datos:

- **RDbStoreDatabase** ofrece una interfaz para crear y abrir una base de datos que no va a ser compartida, es decir, solo una aplicación va a acceder a ella. Las operaciones son realizadas directamente a un fichero.

- **RDbNamedDatabase** ofrece por el contrario una interfaz para crear y abrir una base de datos identificado por su nombre y formato. Esta clase permite tanto el acceso exclusivo desde el cliente como el acceso compartido cliente-servidor.

Para implementar la agenda de la aplicación es más conveniente usar la primera API, ya que solo se necesita un acceso exclusivo a la base de datos, donde estarán guardados los datos de los contactos: nombre, teléfono y clave pública de dicho contacto.

```
TInt CBookDb::CreateDb(const TFileName& aNewBookFile)
    {
    Close();

    // Crear una base de datos vacía.
TRAPD(error,
iFileStore = CPermanentFileStore::ReplaceL(iFsSession,
aNewBookFile, EFileRead|EFileWrite);

iFileStore->SetTypeL(iFileStore->Layout());

TStreamId id = iBookDb.CreateL(iFileStore);

     iFileStore->SetRootL(id);

     iFileStore->CommitL();

     CreateBooksTableL();

     );

    if(error!=KErrNone)
    {
     return error;
    }
    iOpen = ETrue;
    return KErrNone;
    }
```

Este método recibe el la ruta y el nombre del fichero donde se va a guardar la base de datos. El fichero que se va a crear va a ser permanente y se le van a conceder permisos de lectura y escritura sobre él. Por último se crea la única tabla que va a tener la base de datos.

```
void CBookDb::CreateBooksTableL()
    {

    TDbCol nombreCol(KNombreCol, EDbColText);

    TDbCol telCol(KTelCol, EDbColText);

    TDbCol claveCol(KClaveCol, EDbColText);

    TDbCol tieneClaveCol(KTieneClaveCol, EDbColText);

    CDbColSet* agendaColSet = CDbColSet::NewLC();
    agendaColSet->AddL(nombreCol);
    agendaColSet->AddL(telCol);
    agendaColSet->AddL(claveCol);
    agendaColSet->AddL(tieneClaveCol);

    // Creamos la tabla
    User::LeaveIfError(iBookDb.CreateTable(KAgendaTabla,
        *agendaColSet));
    CleanupStack::PopAndDestroy(agendaColSet);
      }
```

Cada tipo TDbCol es una columna. Se necesitan 4: el nombre, el número de teléfono, la clave pública y un flag que indique si tiene clave o no. Todas son de tipo texto. Una vez declaradas, se añade el conjunto de columnas a la base de datos usando el método *CreateTable()*.

Apertura de una base de datos:

```
TInt CBookDb::OpenDb(const TFileName& aExistingBookFile)
    {
    Close();

    if(!BaflUtils::FileExists(iFsSession, aExistingBookFile))
        {
            return KErrNotFound;
        }

TRAPD(error,

iFileStore = CPermanentFileStore::OpenL(iFsSession,
aExistingBookFile, EFileRead|EFileWrite);

     iFileStore->SetTypeL(iFileStore->Layout());

     iBookDb.OpenL(iFileStore,iFileStore->Root())
);
    if(error!=KErrNone)
    {
     return error;
    }

    iOpen = ETrue;
    return KErrNone;
      }
```

Para poder acceder a una base de datos ya creada es necesario abrirla en primer lugar indicando el nombre del fichero que la alberga y declarando qué permisos se desean sobre ella. Intentar abrir una base de datos que ya esté abierta provoca un fallo en la aplicación que causa su finalización

Inserción de datos:

```
TInt CBookDb::crearEntrada(const TDesC& aNombre,const TDesC& aTel,
const TDesC& clave, const TDesC& tieneClave)
    {

    RDbTable table;
    TInt err = table.Open(iBookDb, KAgendaTabla,
    table.EUpdatable);

    if(err!=KErrNone)
    {
     return err;
    }

     CDbColSet* booksColSet = table.ColSetL();

    CleanupStack::PushL(booksColSet);

    table.Reset();

    RDbColWriteStream writeStream;

    TRAPD(error,
     table.InsertL();
     table.SetColL(booksColSet->ColNo(KNombreCol), aNombre);
     table.SetColL(booksColSet->ColNo(KTelCol), aTel);
     table.SetColL(booksColSet->ColNo(KClaveCol),clave);
     table.SetColL(booksColSet->ColNo(KTieneClaveCol),
     tieneClave);
     );

if(error!=KErrNone)
    {
     return error;
    }
writeStream.Close();

    TRAP(err, table.PutL());    // Actualiza cambios
if(err!=KErrNone)
    {
     return err;
    }

    CleanupStack::PopAndDestroy(booksColSet);
    table.Close();

      return KErrNone;
```

Como se ha mencionado anteriormente, existen dos maneras de insertar datos: usando sentencias SQL o usando métodos nativos. Se quería experimentar con ambas opciones, tanto con las llamadas nativas que ofrece Symbian como con sentencias SQL. Para la creación de tablas en la base de datos y la inserción de datos se usó los métodos nativos, mientras que para la modificación y eliminación de datos se usaron sentencias SQL.

En la inserción de datos ha de indicarse la tabla a la que se quiere insertar los datos usando *table.Insert()*. Los campos de cada columna se rellenan usando *SetColL()*, pasando como parámetros la columna y el dato a introducir. Por último, se actualizan los cambios usando *PutL()*.

6. Bibliografía

- J2ME:

"Programming the Java 2 micro edition for Symbian OS. A developer guide to MIDP2."
Martin de Jode
John Wiley & Sons, Ltd
2001

"Programación de dispositivos móviles con J2ME"
Lozano Ortega, Miguel Ángel
Universidad de Alicante, Servicio de Publicaciones
2005

"Java a tope: J2ME (JAVA 2 MICRO EDITION)". Edición Electrónica.
Sergio Gálvez Rojas
Lucas Ortega Díaz
E.T.S. de Ingeniería Informática
Universidad de Málaga

A Brief Introduction to Secure SMS Messaging in MIDP.
Forum Nokia, 2002
http://www.forum.nokia.com

- WMA:

"Enterprise J2ME: Developing Mobile Java Applications"
Michael JuntaoYuan, Jim Colson
Prentice Hall PTR
2003

Wireless Messaging API
http://java.sun.com/products/wma/

J2ME Wireless Toolkit 2.0
http://java.sun.com/products/j2mewtoolkit/

MIDP 2.0 Specification
http://java.sun.com/products/midp

- PIM:

PIM Optional Package Specification
http://jcp.org/aboutJava/communityprocess/final/jsr075/index.html

Managing Personal Information - An Introduction to the PIM API for Java ME
http://developers.sun.com/mobility/apis/pim/pim1/

- DBMS Symbian:

S60 Platform: Using DBMS APIs
http://www.forum.nokia.com/info/sw.nokia.com/id/5e8d013f-d81d-4089-a34c-76858c3a0b58/S60_Platform_DBMS_Example_v2_0_en.zip.html

Symbian C++ API Reference
http://www.symbian.com/developer/techlib/v70docs/sdl_v7.0/doc_source/reference/cpp/DBMSColumnsColumnSetsAndKeys/index.html

- CryptoSMS:

Página del proyecto CryptoSMS:
http://cryptosms.org/

BouncyCastle:
http://www.bouncycastle.org/

7. Glosario

- **SMS**: Short Message Service (Servicio de mensajes cortos)
- **MMS**: Multimedia Messaging System (Sistema de mensajería multimedia)
- **J2ME**: Java 2 Micro Edition
- **CryptoSMS**: Aplicación en Java para el envío de mensajes cifrados
- **API**: Application Programming Interface (Interfaz de Programación de Aplicaciones): conjunto de métodos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
- **WMA**: Wireless Messaging API (API para la mensajería sin cables)
- **PIM**: Personal Information Management (Gestión de la información personal)
- **JVM**: Java Virtual Machine
- **VM** : Virtual Machine (Máquina virtual)
- **CLDC**: Connected Limited Device Configuration
- **CDC**: Connected Device Configuration
- **KVM**: VM de la configuración CLDC
- **CVM**: VM de la configuración CDC
- **MIDP**: Mobile Information Device Profile
- **MIDlet**: Aplicación para teléfonos móviles

- **Symbian**: Sistema Operativo para terminales móviles
- **DBMS**: DataBase Management System (sistema de gestión de bases de datos)

8. Autorización

Los abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales, la memoria, el código, la documentación y/o el prototipo desarrollado.

Firmado,

Clara Valerio Sánchez	NIF: 03906083-Q
José Pablo Ferrero Prieto	NIF: 53413764-J
Fernando Casas García	NIF: 50886402-Y