Sistemas Informáticos

Curso 2007-2008

Investigación y Desarrollo de Aplicaciones de Mensajería en Symbian

Guillermo Colmenarejo Martín

Carlos Enrique Rivera Cordero

Yaofeng Zhang

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 Resumen

- **En Castellano**

Este proyecto nació con el objetivo de crear una aplicación de mensajería sobre terminales móviles basados en la plataforma Symbian, con vistas a una posterior incorporación de criptografía para garantizar autenticidad y no repudio en las comunicaciones.

Para ello el primer paso fue adquirir formación acerca de los conceptos básicos de la plataforma Symbian, así como sobre la configuración y el manejo de las herramientas de desarrollo necesarias para la realización del proyecto. Posteriormente nos centramos en el estudio de las API´s de comunicaciones que ofrece Symbian y en el diseño y desarrollo de un prototipo de la aplicación.

Además también hemos llevado a cabo diferentes pruebas del prototipo sobre diferentes entornos de emulación, y por supuesto sobre terminales móviles basados en plataforma Symbian.

- **En Inglés**

This project was born with the purpose of creating a messaging application for Symbian platform based on mobile terminals, with a view to a later incorporation of cryptography to guarantee authenticity and not-repudiation in communications.

The first step was to acquire training about Symbian platform basic concepts, as well as the configuration and managing of the development tools necessary for the project accomplishment. Later we focused on the study of the Symbian communications API´s and on the design and development of a prototype application.

Also we have carried out different tests of the prototype on different emulation environments, and of course, on Symbian platform based on mobile terminals

2 Presentación

En la actualidad, en esta sociedad de la información en la que nos encontramos inmersos, la necesidad de comunicación y el uso del teléfono móvil se ha convertido en un elemento tan indispensable como natural en nuestras vidas. Sin embargo, todavía queda mucho trabajo por delante en lo que se refiere a investigación y desarrollo de aplicaciones de comunicación seguras sobre este tipo de terminales. Es decir, cada vez es más necesario no sólo proporcionar servicios de voz y mensajería SMS y MMS, sino también proporcionar mecanismos que garanticen autenticidad y no repudio en este tipo de comunicaciones.

Debido a estos motivos nos planteamos el desarrollo de una aplicación de mensajería SMS y MMS para terminales móviles, cuyos principales objetivos son:

- Crear un medio de comunicación de fácil manejo, transparente y estable para la interacción entre usuario y aplicación, teniendo en cuenta las limitaciones y restricciones del hardware y software que la rodea.
- Incorporar elementos adicionales ya desarrollados, como por ejemplo el uso de mecanismos de criptografía y gestión de datos .

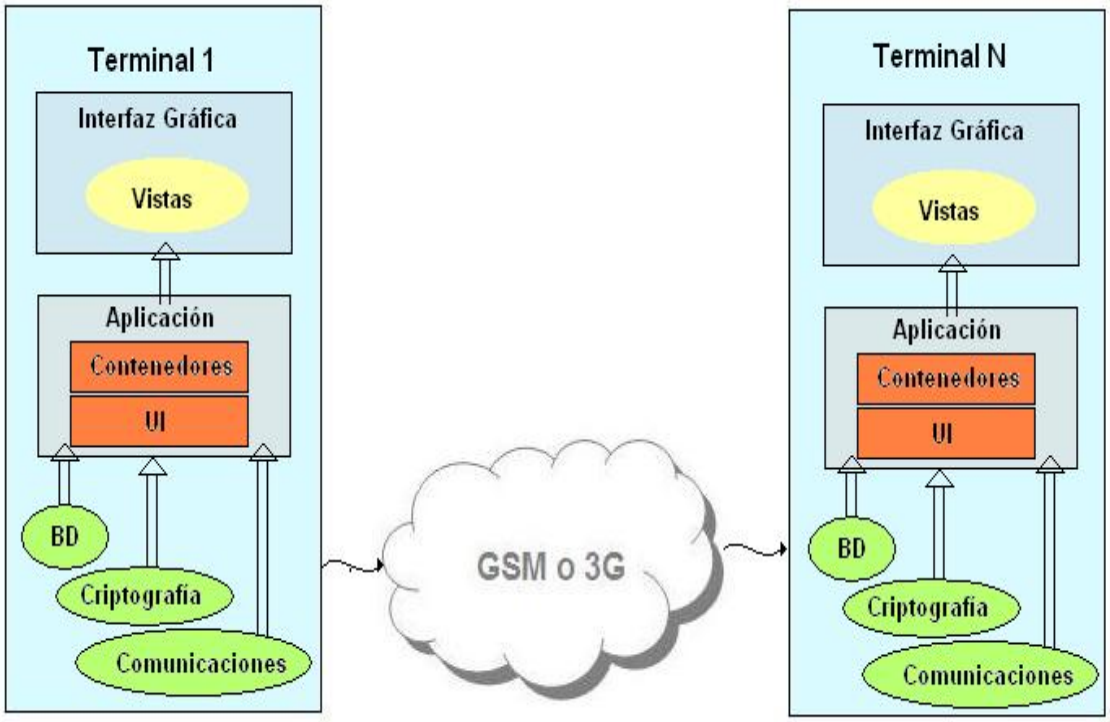
Sin embargo, debido a la gran diversidad y variedad de terminales existentes en el mercado, hemos tenido que centrarnos solamente en un subconjunto de ellos, más concretamente, en aquellos basados en la plataforma Symbian.

Las razones por las que hemos elegido Symbian tienen que ver con el hecho de que es una de las plataformas para dispositivos móviles más extendidas, así como con el hecho de que está respaldada por grandes multinacionales como por ejemplo Nokia. Además también hemos tenido en cuenta otros aspectos como la facilidad de acceso a información y recursos que nos apoyen en el desarrollo sobre Symbian.

Concretamente nos hemos centrado en los dispositivos Symbian encuadrados dentro de la serie S60 3ª Edición, los cuáles están basados en la versión 9.1 de Symbian.

3 Arquitectura

Una primera aproximación a la aplicación se muestra en el siguiente dibujo:



Existen tres módulos fundamentales diferenciados con tareas específicas, el módulo de base de datos, el módulo de Criptografía, y el módulo de comunicaciones. Todos ellos se usan e interactúan a través de la parte del control de la aplicación, que se encarga por otra parte del control de vistas de la interfaz gráfica.

Para la arquitectura de la interfaz gráfica se ha tenido en cuenta que la aplicación tiene mucha interacción con el usuario y constantemente hace uso de la entrada - salida para mostrar o pedir información externa y avanzar con la ejecución.

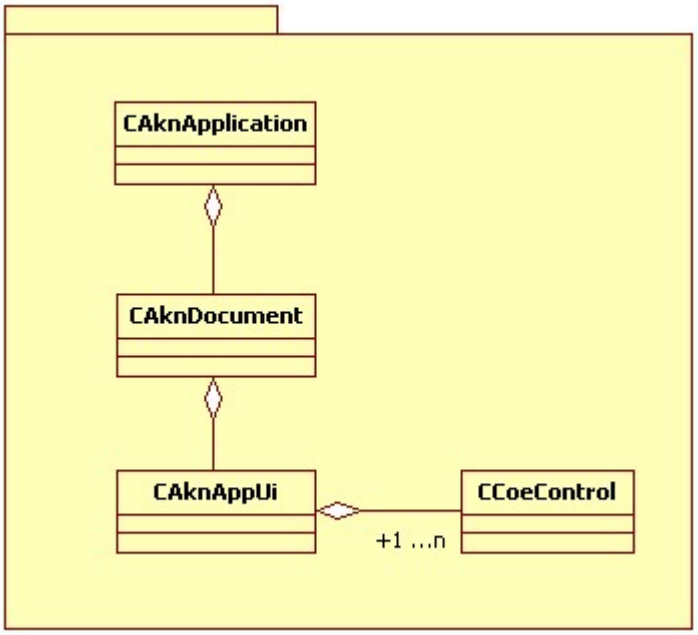
Symbian propone varias alternativas prediseñadas para la creación de aplicaciones.

- Arquitectura tradicional.
- Basadas en Diálogos.
- Arquitectura Avkon.

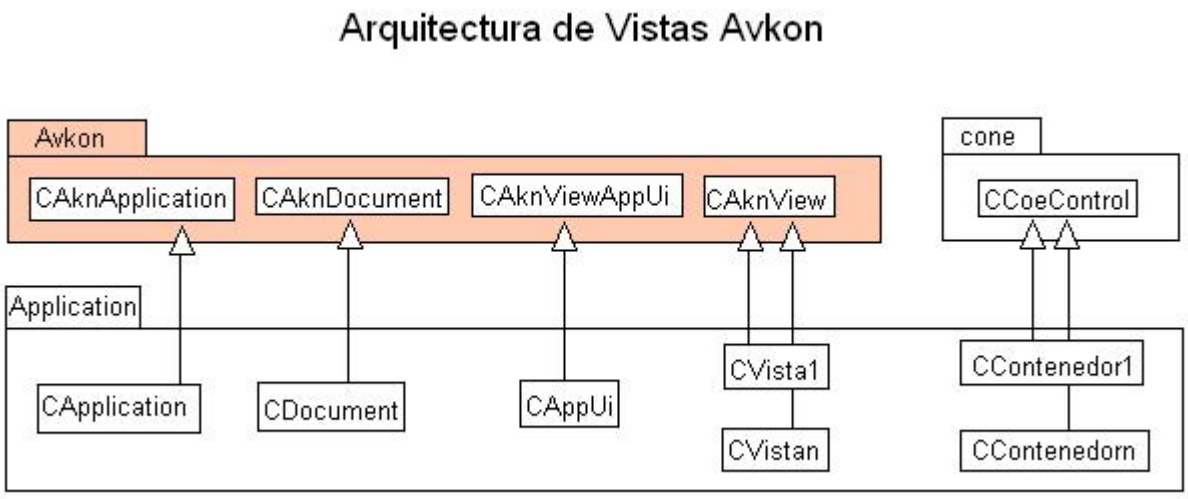
La arquitectura tradicional tiene la desventaja de no proporcionar manejo de vistas, así que es necesario implementarlo aparte. La arquitectura basada en Diálogos proporciona una gran independencia entre aplicaciones que comparten funcionalidad, hace que cada aplicación sea más eficiente y reduce la complejidad entre aplicaciones simples; pero no puede ser usada en todos los casos y es incompatible con el uso de vistas. En arquitectura de Vistas Avkon cada vista maneja sus propios eventos y se proporciona una serie de métodos para el control y cambio entre vistas, de tal manera que el entorno que rodea a cada vista puede ser distinto. Se ha elegido la Arquitectura Avkon por la flexibilidad y modularidad que ofrece junto con Dialogos para mejorar el rendimiento de la aplicación.

Esta arquitectura sigue el modelo tradicional y añade el control de vistas.

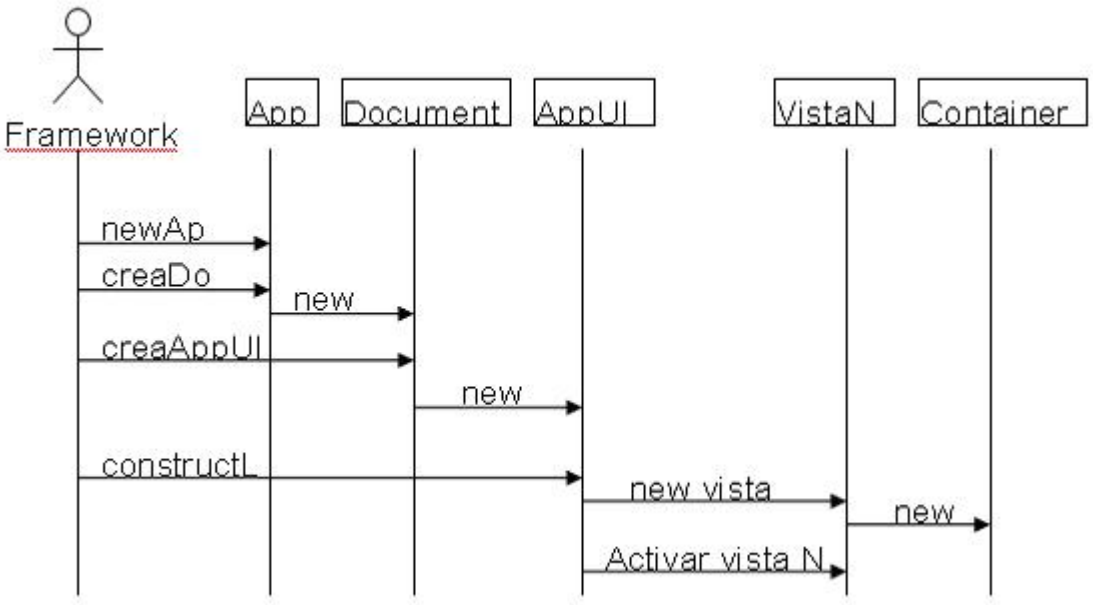
Según el modelo tradicional una aplicación simple se representa con la clase CApplication; para almacenar y manipular los datos internos de la aplicación se usa la clase CDocument; CAppUi es la responsable del menú principal, y los eventos de teclado y por último el contenedor CContenedor se encarga de como dibujar la pantalla y contiene otros objetos de control.



Una aplicación en ejecución según el modelo tradicional de arquitectura, se construye siguiendo un orden específico. La primera clase que se crea es la clase aplicación. Esta clase crea la clase Documento y la clase documento crea y retorna la clase appUi a la clase aplicación. La clase appUI crea cada uno de los contenedores y pone en funcionamiento la gestión de los eventos de teclado.



En la arquitectura Avkon las clases aplicación, documento tienen las mismas tareas y se construyen de la misma forma, sin embargo la clase appUi ya no se encarga de gestionar los eventos de teclado sino que crea las distintas vistas que tienen un identificador único y las apila con ese identificador, además pasa el foco de control a cada vista dependiendo cual esté activa en ese momento. No obstante, debido a que solo existe una en cada aplicación sigue siendo la responsable de la creación y destrucción de los objetos que representen el modelo lógico de la aplicación. Existe entonces una clase nueva, la clase vista que se encarga de gestionar los eventos de tecla y crear la clase contenedor, de tal manera que cada vista tendrá su contenedor, y por tanto, su manera de visualizarse y objetos propios. Esto es, el entorno. La clase contenedor seguirá teniendo las mismas tareas y funciones, salvo por estar asociada a una vista determinada.



En el caso concreto de la aplicación la clase que implementa el módulo que utiliza la base de datos se crea, cierra y destruye en la clase appUi ya que tiene que ser único y permanece abierta durante la ejecución. La clase que representa el módulo de criptografía se utiliza en vistas determinadas y en partes puntuales así que por ahorro de memoria no conviene tener un objeto creado durante toda la ejecución, lo mismo ocurre con la parte de comunicaciones.

3.1 Comunicaciones

3.1.1 Diseño

Las decisiones de diseño para el desarrollo del proyecto se han ido tomando conforme a las dificultades y necesidades que han ido apareciendo.

Inicialmente se establecieron las funcionalidades que tenía que poseer la aplicación, luego se dividieron esas funcionalidades para ir alcanzándolas progresivamente, provocando distintos cambios en el diseño principal.

En las primeras partes del desarrollo y una vez terminado el estudio de la aplicación en java, una de las primeras funcionalidades que se establecieron para su implementación fue el envío simple de mensajes de texto de un terminal móvil a otro. Se eligieron distintas API's para el envío de mensajes teniendo en cuenta las limitaciones y restricciones que fueron apareciendo. La elección de las API's y sus características están explicadas en el apartado de las API's. La posibilidad de envío de mensajes de texto obligó a buscar y decidir que editor de texto se necesitaba para su construcción. En el caso de la aplicación para confeccionar el cuerpo del mensaje de texto se optó por crear una vista que contenía un CEikRichTextEditor, que es un editor de texto enriquecido como objeto que ya proporciona Symbian. Para la inclusión del número de teléfono se optó primero por hardcodearlo y posteriormente usar diálogos para su obtención por parte del usuario. Una modificación de diseño de esta parte fue provocada posteriormente al llevar la aplicación, una vez funcionaba en el emulador, al terminal móvil. El número de teléfono tenía que incluir el código de área y no debía añadirse siempre, ya que en la agenda podía guardarse con ese código o no. Se tomó la decisión de añadir un cuadro de diálogo preguntando el número de área siempre que se quisiera enviar un mensaje a un número de móvil determinado, de esta manera se hace extensible a poder enviar mensajes de texto a móviles de otro área.

Ahora bien, una vez estuviera funcionando el envío, era necesario capturar ese mensaje, para ello se decidió entre dejar los mensajes en la bandeja de entrada por defecto o crear una bandeja privada en cada terminal y mover los mensajes enviados por la aplicación a esta bandeja. Por transparencia y confianza del usuario se decidió que los mensajes permanecieran en la bandeja de entrada, y explorar cada vez que se quisieran ver. Inicialmente eran texto plano, pero en posteriores iteraciones aparecerían encriptados, así que no violaría la finalidad del proyecto.

Al tener que explorar la bandeja de entrada para mostrar los mensajes que provenían de una aplicación gemela, era necesario almacenar el cuerpo del mensaje y la dirección del remitente para mostrarla por pantalla; esto se hace con un par de arrays de tamaño constante e igual a diez que se establece en el código directamente. El valor del tamaño se tomó teniendo en cuenta que la memoria en Symbian es limitada y para una primera versión no era necesario llevar al límite al terminal, no obstante su modificación es muy sencilla, cambiar el valor a la constante que define el tamaño en el código. Para mostrar la cabecera de cada mensaje se hizo uso de un listbox de tal manera que es posible navegar entre los mensajes y mostrar el texto de uno a elección.

Otra necesidad derivada de la recepción de mensajes de texto fue la distinción de mensajes de la aplicación en desarrollo frente a mensajes de texto normales. Según el estudio de la aplicación en Java, este problema estaba resuelto con el uso de puertos virtuales, pero, en Symbian el uso de estos puertos virtuales no existe. Se decidió entonces marcar los mensajes añadiendo al texto del cuerpo del mensaje una serie de caracteres particulares, en nuestro caso estos caracteres son CRYPTO, esto ha provocado una pérdida de 6 caracteres del mensaje de texto. Si bien es cierto que un mensaje de texto permite hasta 160 caracteres, es una pérdida aceptable.

Una vez definida la distinción entre mensajes y la recepción de los mismos había que mostrar su texto por pantalla, para esto teníamos la alternativa de crear una vista adicional para este fin o utilizar la vista existente de edición de texto. Se optó por usar la vista existente de edición de texto, de tal manera que no era necesario alocar mas memoria para una vista adicional y además era posible reenviar el mensaje. Según la arquitectura escogida para la aplicación se utilizan métodos determinados de la clase appUi para el cambio entre vistas. Ya que todas las vistas tienen una opción para volver a la vista desde la que se invocó, y según los distintos usos de la vista de edición del mensaje, fue necesario crear un objeto que almacenara el estado de nuestra máquina de estados, que se mostrará mas adelante, para el control de vistas. Este objeto toma el nombre de Opcion.

Teniendo el ciclo completo desde el envío de mensajes de texto a la recepción había que decidir como implementar la agenda del teléfono. Se tenía las alternativas de utilizar la agenda propia del usuario o bien implementar una agenda propia de la aplicación, para la primera opción había que tener en cuenta que al añadir el módulo de criptografía sería necesario de almacenar las claves de alguna manera externa a la agenda propia del teléfono y corresponder cada contacto con su clave particular; para la segunda opción era necesario diseñar un mecanismo de almacenamiento de cada contacto con su información. Se decidió usar la segunda opción y de esta manera separar ambas agendas y hacer privada esta información. Se creó una vista nueva para la agenda u se utilizó un listbox para mostrar cada contacto. A la hora de mostrar la información de los contactos inicialmente se usó un listbox simple, pero solo permitía la visualización del nombre, posteriormente se paso a un listbox de doble línea y de aquí a un listbox que incluye los tres campos de cada contacto, el nombre, el teléfono y un icono que indica si tiene clave almacenada o no. Para pasar información entre distintas vistas se reutilizó el objeto opción añadiéndole campos con el número de teléfono y la clave. Y finalmente para almacenar la información se optó por el desarrollo de un DAO.

La inserción de la parte de criptografía hacía necesario un mecanismo para intercambiar claves entre terminales, para el caso de la aplicación se utiliza el envío de un mensaje de texto marcado con la palabra KEY y se ha insertado una opción en el menú principal para escanear la bandeja de entrada en busca de claves e incorporarla a la base de datos.

Finalmente se amplió la funcionalidad permitiendo además mensajes multimedia. Este tipo de mensajes permite enviar mensajes de texto mucho mayores, por lo tanto se tomó la decisión de solo permitir adjuntar texto al mensaje (como cuerpo del mensaje).

Máquina de estados.

La máquina de estados diseñada para el cambio de vistas y envío de mensajes es la siguiente:

El estado inicial comienza en la vista principal.

Vista principal:

Transiciona a vista mensaje si se selecciona crearSMS, opción toma el valor SMS, uso toma valor escribir.

Transiciona a vista mensaje si se selecciona crearMMS, opción toma el valor MMS, uso toma valor escribir.

Transiciona a vista mensaje si se selecciona enviarClave, opción toma el valor Clave, uso toma valor escribir.

Transiciona a vista agenda si se selecciona agenda.

Transiciona a BandejaMMS si se selecciona bandeja MMS, opción toma el valor MMS.

Transiciona a BandejaSMS si se selecciona bandeja SMS, opción toma el valor SMS.

Vista mensaje:

Transiciona a vista principal si se selecciona volver y uso tiene el valor escribir, opción toma el valor "none".

Transiciona a BandejaMMS si se selecciona volver, uso tiene el valor leer y opción tiene el valor MMS.

Transiciona a BandejaSMS si se selecciona volver, uso tiene el valor leer y opción tiene el valor SMS.

Transiciona a vista agenda si se selecciona enviar a contacto, opción toma el valor contacto.

BandejaMMS:

Transiciona a vista principal si se selecciona volver.

Transiciona a vista mensaje si se selecciona abrir, uso toma el valor leer.

BandejaSMS:

Transiciona a vista principal si se selecciona volver.

Transiciona a vista mensaje si se selecciona abrir, uso toma el valor leer.

Vista Agenda:

Transiciona a vista principal si se selecciona volver.

Transiciona a vista mensaje si se selecciona cancelar u ok.

3.1.2 API

Para el desarrollo de los prototipos hemos investigado numerosas API´s de Symbian v9.1, utilizando finalmente sólo aquellas que se adaptaban mejor a nuestros propósitos. En lo que se refiere a esta sección sólo comentaremos las que tienen que ver con el modelo de la aplicación ya que son las que tienen una especial relevancia para el proyecto, omitiendo aquellas más triviales como por ejemplo las utilizadas para construcción de las vistas de la aplicación.

La mayor parte de las API´s estudiadas están orientadas al manejo de las comunicaciones sobre Symbian v9.1, y más concretamente al manejo (envío, manipulación, recepción) de SMS y MMS. A continuación pasamos a explicar más detalladamente cada una de ellas:

- **Manejo de SMS**

  Al ser uno de los principales requisitos de la aplicación el envío de SMS (y MMS), necesitábamos obtener acceso a esos servicios que ofrece Symbian. Lógicamente nuestras primeras investigaciones estuvieron encaminadas al envío de SMS, las cuáles nos llevaron al uso de la API Send UI que pasamos a explicar más detalladamente a continuación:

  - **Send UI**

    1. **Descripción**

       Send UI es una API que proporciona a las aplicaciones cliente las capacidades necesarias para ofrecer al usuario servicios de mensajería interactiva. Dicho de otra manera, Send UI proporciona el uso de cualquier método de mensajería soportado por el dispositivo a través de una serie de elementos gráficos (menús desplegables, editores, ...).

    2. **Funcionamiento**

       Como se ha comentado anteriormente, Send UI ofrece sus servicios a nivel interfaz de usuario de una manera muy definida:

- Primero ofrece la posibilidad de desplegar ante el usuario un menú pop – up ofreciendo los diferentes métodos de mensajería disponibles, esperando a que el usuario seleccione uno de ellos para continuar con el proceso. Una de las cuestiones a resolver en este punto es el hecho de que debe haber una concordancia entre los métodos de mensajería que ofrece la aplicación cliente de Send Ul y los métodos de mensajería disponibles en el dispositivo, ya que no tendría sentido ofrecer la posibilidad de usar un servicio cuyo protocolo no está implementado en el dispositivo. Para evitar este posible problema Send Ul proporciona una serie de mecanismos para validar si un determinado método de mensajería es soportado por el dispositivo y en función de eso ofrecerlo al usuario o no.
- Una vez seleccionado un método por el usuario se abre un editor para que el usuario edite el mensaje, seleccione el destinatario, ... Nótese que Send Ul ofrece la posibilidad de usar diferentes editores dependiendo del método de mensajería seleccionado.
- Por último el editor contiene un menú que ofrece al usuario la posibilidad de enviar el mensaje o cancelar la aplicación

3. Clases Principales

- CSendUl: es la clase principal de esta API. Ofrece el menú pop-up y los servicios de creación y envío de mensajes.
- CMessageData: encapsula los datos y contenido del mensaje
- SendUlConst.h: archivo de cabecera que encapsula las constantes necesarias para la manipulación y selección de los diferentes métodos de mensajería

4. Ejemplo de Funcionamiento

A continuación mostramos un ejemplo de funcionamiento de SendUl. Se trata de un ejemplo muy básico de una arquitectura de vistas Akvon, incorporando una instancia de CSendUl que permite enviar SMS, MMS y e-mails.

Primeramente la creación de la instancia se haría se la siguiente manera:

```
// ----------------------------------------------------------------------
// CMessagingAppUi::ConstructL()
// Symbian two-phased constructor
// ----------------------------------------------------------------------
void CMessagingAppUi::ConstructL(){
    BaseConstructL( CAknAppUi::EAknEnableSkin );
    iAppContainer = CMessagingContainer::NewL( ClientRect() );
    iAppContainer->SetMopParent( this );
    AddToStackL( iAppContainer );
    iSendUi = CSendUi::NewL();
```

A continuación mostramos el switch que maneja la el menú pop-up mencionado anteriormente:

```
// ----------------------------------------------------------------------
// CMessagingAppUi::HandleCommandL(TInt aCommand)
// takes care of command handling
// ----------------------------------------------------------------------
void CMessagingAppUi::HandleCommandL( TInt aCommand ){
  switch ( aCommand ) {
    case EAknSoftkeyExit:
    case EEikCmdExit:{
      Exit();
      break; }
    case EMessagingCmdCreateSMS:{
      // create and send SMS
      CreateAndSendMessageL( KSenduiMtmSmsUid );
      break; }
    case EMessagingCmdCreateMMS:{
      // create and send MMS
      CreateAndSendMessageL( KSenduiMtmMmsUid );
      break;
      }
    case EMessagingCmdCreateEmail:{
      // create and send email
```

```
        CreateAndSendMessageL( KSenduiMtmSmtpUid );
        break;
        }
    default:
      break;
    }
```

Nótese que KSenduiMtmSmsUid, KSenduiMtmMmsUid, ... son las constantes que identifican a los métodos de mensajería (están incluidas en <senduiconst.h>). Ahora mostramos la implementación del método CreateAndSendMessageL y la liberación de los recursos que encapsula :

```
// ----------------------------------------------------------
// CMessagingAppUi::CreateAndSendMessageL(const TUid aServiceUid)
// Starts message editor via CSendUi class.
// Message type is defined in parameter aServiceUid.
// ----------------------------------------------------------
void CMessagingAppUi::CreateAndSendMessageL( const TUid aServiceUid )
        {
        // create empty message
        CMessageData*  message = CMessageData::NewLC();
        // start message editor through SendUI
        iSendUi->CreateAndSendMessageL( aServiceUid, message, KNullUid, EFalse );
        CleanupStack::PopAndDestroy( message );
```

Hay que destacar que cuando se ejecuta el método de iSendUi CreateAndSendMessageL (aService, ..) es el momento en que se abre el editor para la creación del mensaje y su posterior envío.

```
// ----------------------------------------------------------
// CMessagingAppUi::~CMessagingAppUi()
// Destructor
// Frees reserved resources
// ----------------------------------------------------------
//
```

```
CMessagingAppUi::~CMessagingAppUi()
  {
  if ( iAppContainer )
    {
    RemoveFromStack( iAppContainer );
    delete iAppContainer;
            iAppContainer = NULL;
    }
  if ( iSendUi )
        {
        delete iSendUi;
        iSendUi = NULL;
        }
```

Como se ha podido apreciar, el uso de la API Send UI tiene como principal ventaja el hecho de que simplifica bastante el código necesario para el envío de SMS. Esto se debe a que la mayor parte del procesamiento necesario para el envío del mensaje se delega en la clase CSendUI, como muestra por ejemplo el hecho de que el programador no tiene que preocuparse de cómo va a editar el mensaje (Send UI proporciona el editor), cómo va a añadir el destinatario (el editor incluye lo necesario para hacerlo), cómo enviar el mansaje (basta con que el usuario pulse el botón "send" que ofrece el editor).

Sin embargo en el contexto de nuestro proyecto, esa principal ventaja que proporciona SendUI se vuelve totalmente en nuestra contra. Esto se debe a que el hecho de usar Send UI significa perder totalmente el control de los mensajes entre su creación y envío (es totalmente transparente al programador), o lo que es lo mismo, no podemos realizar ningún tipo de procesamiento del cuerpo del mensaje antes de que sea enviado. Por lo tanto no podríamos encriptar el cuerpo del mensaje antes de que sea enviado, lo que supone no cumplir uno de los principales requisitos de nuestro proyecto. Todo esto nos llevó a desestimar el uso de la API SendUI y a tener que buscar otra alternativa para el envío de los SMS.

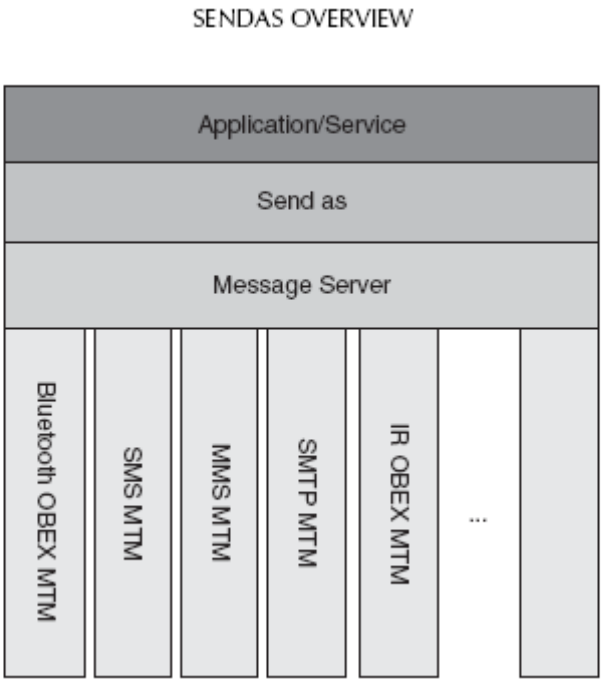
La solución que adoptamos finalmente fue recurrir a la API Send As, combinada con el uso de un editor de texto independiente. Más concretamente utilizamos un tipo de editor encapsulado en la clase CEikRichTextEditor (componente gráfico que ofrece la API de Symbian v9.1), lo cuál nos permitía editar el cuerpo de los SMS de

una manera totalmente independiente de la API utilizada para su envío. A continuación pasamos a comentar más en profundidad la API RSendAs:

- **Send As**

1. Descripción

Conceptualmente Send As se encuentra en un nivel más bajo que Send Ul, de manera que constituye la capa inmediatamente superior al Message Server en la jerarquía de capas de mensajería en Symbian. Dicho de otra manera, Send As es la base de Send Ul, que simplemente se reduce a una serie de librerías gráficas DLL sobre Send As que ayudan a integrar los servicios de mensajería en aplicaciones con interfaz gráfica de usuario sobre Symbian. Todo esto se aprecia mejor en la siguiente figura, donde lo que hay por debajo del Message Server serían los diferentes métodos de mensajería implementados por el dispositivo (SMS, MMS, e-mail, ...) y donde Send Ul se colocaría justo por encima de Send As y por debajo de las aplicaciones de usuario (en la imagen Application Service):



Hay que comentar que el uso de Send As se corresponde más con la clásica arquitectura cliente / servidor, donde el cliente sería la aplicación de usuario (nuestra aplicación en

este caso), las clases de Send As encapsularían la sesión con el servidor y los datos que se intercambian (los mensajes) y el Message Server haría el papel de servidor.

2. Clases Principales

- RSendAs: encapsula una sesión con el Message Server (componente de Symbian encargado de la mensajería) y permite al usuario elegir entre los diferentes métodos de mensajería disponibles en el dispositivo. Los métodos que hemos utilizado en la aplicación son los siguientes:
  - TInt RSendAs::Connect() : establece una conexión con el Message server, lo cuál es necesario para que el Message server atienda nuestras peticiones de envío de mensajes. Devuelve la constante KErrNone si hay éxito en el establecimiento de la conexión.

- RSendAsMessage: encapsula el mensaje propiamente dicho (sea del tipo que sea) y proporciona métodos para enviarlo, almacenarlo, modificarlo, ... Los métodos utilizados en la aplicación son los siguientes:
  - void RSendAsMessage::CreateL(RSendAs &aSendAs, TUid aMessageType): crea un nuevo mensaje en la carpeta "drafts folder", un repositorio en el cual se almacenan los mensajes que se están preparando y que todavía no han sido enviados. Recive como argumentos una referencia a un objeto RSendAs (aSendAs) que representa la conexión con el Message Server y un identificador que representa el tipo de mensaje a crear (aMessageType). Esos identificadores provienen de la clase CSendAsMessageTypes y en nuestro caso usamos *KSenduiMtmSmsUid* que es el que representa el método de mensajería SMS.
  - void RSendAsMessage::AddRecipientL(const TDesC &aAddress, TSendAsRecipientType aRecipientType): añade el número del receptor al mensaje representado mediante un descriptor (aAdress). Además también recibe un argumento que identifica el tipo de receptor

(aRecipientType), los cuáles proceden de la clase TSendAsRecipientType. En nuestro caso usamos la constante *ESendAsRecipientTo* que indica que el receptor debe ser añadido a la única lista de receptores del SMS (para otro tipo de mensajes pueden existir varias listas de receptores)

- void RSendAsMessage::SetBodyTextL(const TDesC &aBody): asigna el cuerpo del mensaje, el cuál se representa mediante un descriptor (aBody).
- void RSendAsMessage::SendMessageAndCloseL(): Envía el mensaje y cierra la conexión.

Nótese que en este caso disponemos de métodos para asignar el cuerpo del mensaje una vez que éste se ha creado y antes de que sea enviado pudiendo realizar cualquier procesamiento entre esos dos pasos, lo cuál se adapta perfectamente al objetivo de nuestra aplicación.

Sin embargo debido a los requisitos de nuestra aplicación no podemos conformarnos sólo con el envío de mensajes sino que tenemos que incluir un proceso de recepción para desencriptar los mensajes y poder mostrarlos al usuario. Como se indica en el apartado de Diseño hemos optado por no crear una nueva bandeja de entrada, sino usar la bandeja de entrada estándar del teléfono. Es decir, dejamos que Symbian realice el proceso de recepción de los mensajes de la misma manera que con un mensaje normal, para posteriormente explorar la bandeja de entrada en busca de los mensajes encriptados (los detalles de este proceso se indican en el apartado Diseño). Por lo tanto esto nos obligaba a buscar la manera de manipular las bandejas de entrada de Symbian de una manera segura y sin que afectara al resto datos almacenados en las bandejas que no tenían nada que ver con nuestra aplicación.

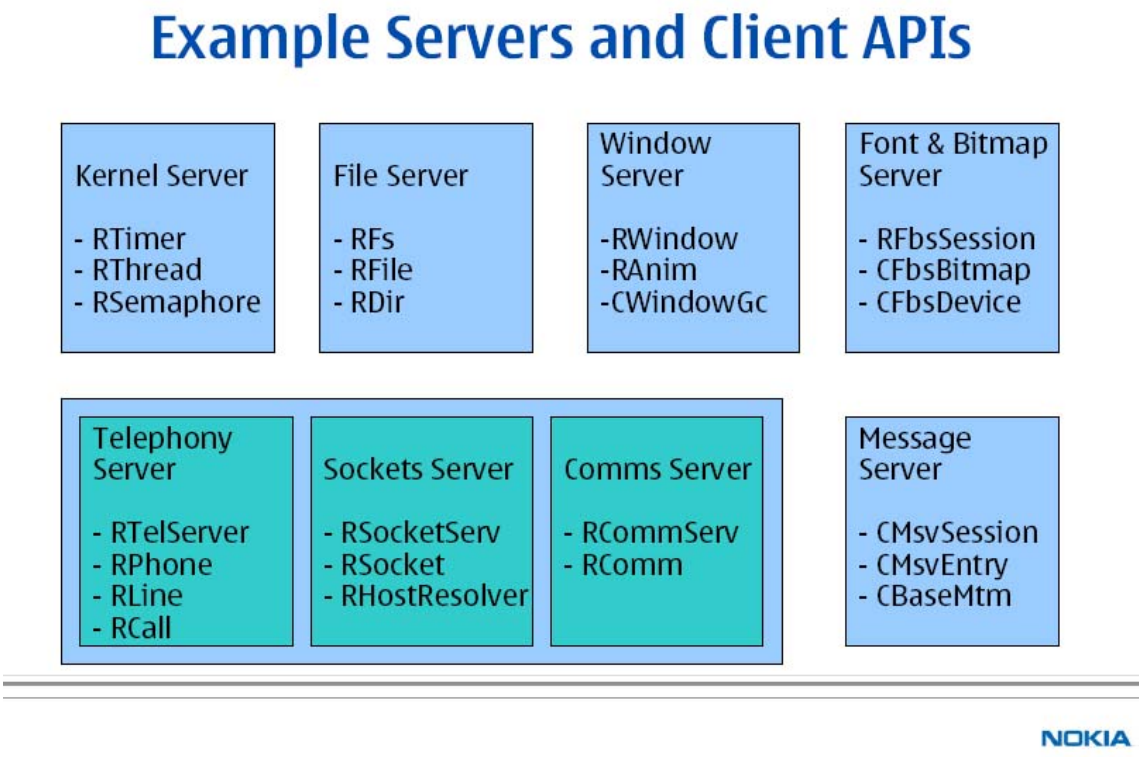
La solución a este problema fue lo que se conoce en Symbian como "Client / Server Framework" y que pasamos a explicar más en detalle a continuación:

- **Client / Server Framework**

  En Symbian este framework se usa para el acceso a todos aquellos recursos del sistema (como por ejemplo abrir archivos, acceder a bandejas de entrada, acceder a servicios de comunicación,...) que puedan ser accesibles de forma concurrente por varios procesos y que por lo tanto

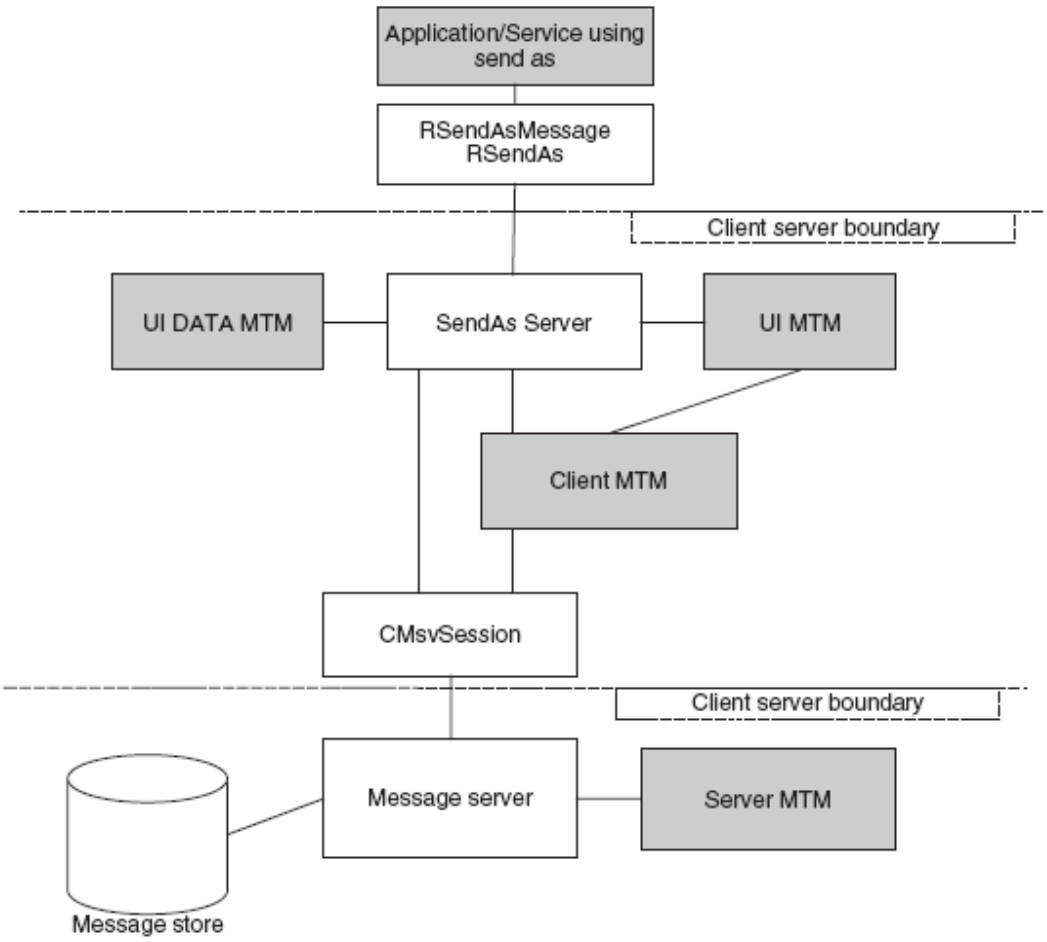
necesitan ser manipulados de una manera cuidadosa. Más concretamente, cada uno de los diferentes servidores de los que se compone Symbian es el encargado de manejar las peticiones de los clientes (aplicaciones de usuario) sobre los recursos que le corresponden.

Cada servidor se ejecuta en su propio proceso o thread y lleva a cabo las peticiones de los clientes de manera que siempre es el servidor el que tiene el control sobre el recurso y el que tiene la potestad de rechazar cualquier petición que pueda causar un problema (por ejemplo que un cliente intente borrar un archivo que este siendo editado por otro cliente).



La imagen muestra los diferentes servidores de Symbian, donde las clases que aparecen dentro de cada servidor conforman la API que permite a los procesos de usuario acceder a los servicios de cada servidor.

En el contexto en el que nos encontramos nos interesa especialmente el Message Server, ya que es el encargado de almacenar los datos de los mensajes (bandejas, ...) y de ofrecer acceso a los diferentes métodos de mensajería. Llegados a este punto resulta interesante para el objetivo de nuestra aplicación mostrar cómo se integra Send As dentro de esta estructura de servidores, ya que ayuda a tener una visión más completa de lo que estamos usando:



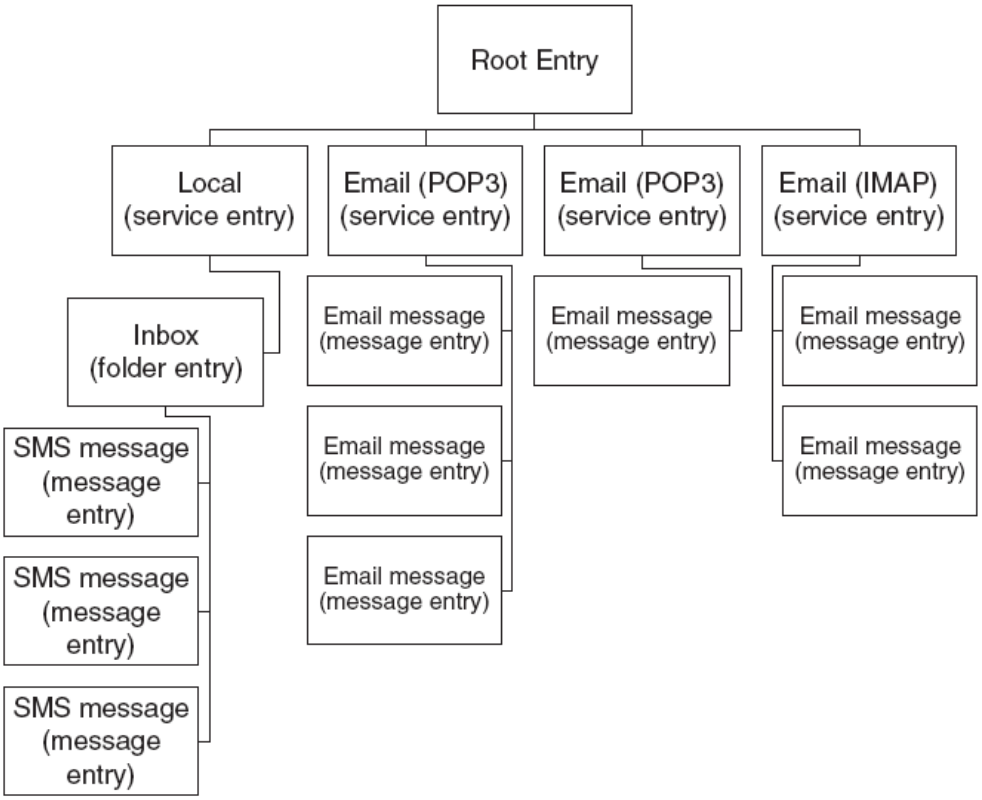
Como se puede apreciar en la figura, Send As hace uso internamente del Client / Server Framework, ya que utiliza un objeto de la clase CMsvSession para establecer una conexión con el Message Server y así poder ofrecer a las aplicaciones de usuario los servicios proporcionados por el servidor. Otro aspecto importante es el significado de las dos fronteras que se aprecian en la imagen, las cuáles representan básicamente dos barreras de permisos. Dicho de otra manera, las aplicaciones que estén por encima de

una determinada barrera necesitarán una serie de permisos adicionales para poder usar las clases que están por debajo de esa barrera (el tema de los permisos se trata en otras secciones más adelante). Nótese que cuanto más abajo nos encontremos más permisos serán necesarios para trabajar directamente con las clases que se encuentren a ese nivel.

Por otra parte, observando la figura se puede apreciar un componente que no se había mencionado hasta ahora, el Message Store, y que es el componente de Symbian encargado del almacenamiento de los mensajes (tanto de los entrantes, los salientes, los que están en construcción, ...). Se trata de una estructura en forma de árbol donde cada nodo del árbol representa una entrada, pudiendo ser cada entrada de unos de los siguientes cuatro tipos:

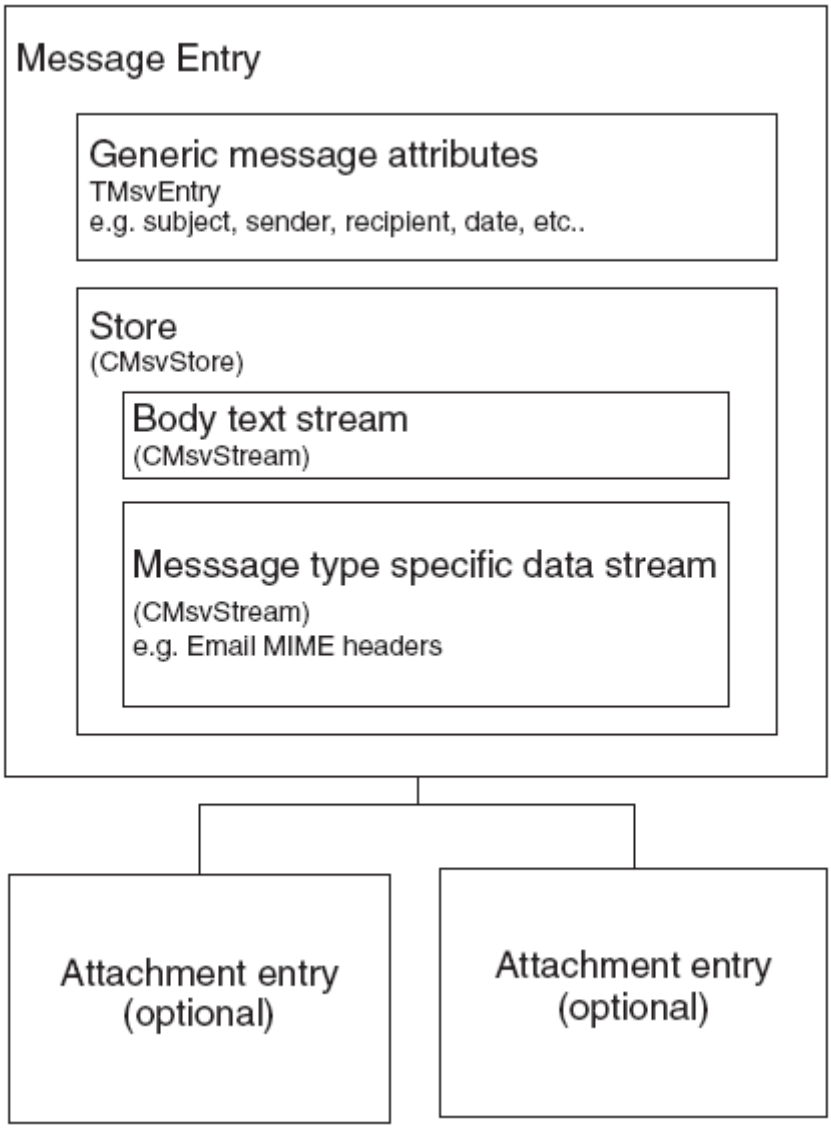
- Folder entry: representa una carpeta dentro del Message Store y sirve para agrupar mensajes de la misma manera que un directorio en el sistema de ficheros de cualquier sistema operativo. La bandeja de entrada y la de salida son ejemplos de este tipo de entradas
- Message entry: almacena el contenido de un determinado mensaje, de manera que existe una carpeta de este tipo por cada mensaje almacenado en el Message Store. Hay que comentar que para determinados tipos de mensajes (como por ejemplo MMS) esta carpetas pueden tener una serie de entradas (o nodos) hijas que representen los datos adjuntos a ese mensaje
- Service entry: representan cuentas externas de mensajería, como por ejemplo un buzón de correo POP3
- Attachment entry: son las entradas usadas para almacenar los datos adjuntos (multimedia, ...) de algunos tipos de mensajes, y que ya han sido mencionadas anteriormente

En la siguiente figura se muestra una posible representación del Message Store en un momento determinado:



Dentro del contexto de nuestro proyecto el tipo de entrada que más nos interesa son las message entry, ya que son las que vamos a manipular. El formato de cada message entry varía dependiendo del tipo de mensaje al que represente y de la etapa en la que se encuentre, es decir, el formato de una entrada que representa un mensaje recibido no es el mismo que el formato de una entrada que representa un mensaje que está siendo preparado para su envío (este tipo de mensajes se almacenan en una carpeta especial del Message Store llamada "Draft folder"). Sin embargo hay una serie de atributos comunes a todas las message entries, como son el asunto (o sumario), la fecha de creación y los detalles del remitente y del destinatario. Estos atributos se manipulan usando las mismas API independientemente del tipo del tipo de mensaje al que pertenezcan, lo que hace más fácil desarrollar código para trabajar con este tipo de entradas.

La estructura de una entrada de tipo message entry podría ser la siguiente:



Como se puede apreciar en la figura, además de los atributos comunes una entrada puede tener una serie de atributos específicos del tipo de mensaje al que representan. Algunos de estos atributos pueden ser el cuerpo del mensaje y el número de la centralita de mensajes en los SMS, las cabeceras MIME en los e-mails, los adjuntos en un MMS, ...

Además en la figura también se muestran algunas de las clases que representan esos atributos, como por ejemplo TMsvEntry que es la clase que encapsula los atributos genéricos o comunes a todos los tipos de entradas.

Volviendo al tema de la recepción de los SMS, una vez visto el Client / Server Framework y el Message Store ya podemos pasar a

comentar las API utilizadas para la exploración de la bandeja de entrada en busca de los SMS cifrados.

- **SMS Client MTM API**

  1. Descripción

  Como se puede ver en una de las figuras de más arriba esta API forma parte del Client / Server Framework de Symbian, más concretamente es la que proporciona a las aplicaciones de usuario la posibilidad de mostrarse como clientes del Message Server dentro del marco del método de mensajería SMS. Dicho de otra manera, nos permite acceder a todos los servicios relacionados con el método de mensajería SMS que proporciona el Message Server.

  2. Funcionamiento

  Como todas las API que pertenecen a este framework, su funcionamiento se basa en la creación y uso de una conexión o sesión entre el thread que representa al servidor adecuado y el thread que representa al cliente adecuado. En este caso el servidor será el Message Server y el cliente será un cliente específico del método de mensajería SMS.

  3. Clases Principales

  - CMsvSession: encapsula una sesión de comunicación entre un cliente del Message Server y el propio Message Server. En nuestro caso abrimos la sesión de manera síncrona con el Message Server. Los principales métodos usados son:
    - CMsvSession*CMsvSession::OpenSyncL (MMsvSessionObserver): abre la sesión con el Message Server de manera síncrona y devuelve la propia sesión. Recibe como argumento una instancia de la clase MMsvSessionObserver, lo cuál se debe al hecho de que cualquier clase de la aplicación de usuario que quiera hacer uso de esta API debe estar declarada como una subclase de MMsvSessionObserver. Esta clase es

una clase abstracta que representa un oyente del Message Server y que tiene como principal ventaja el hecho de que puede recibir notificaciones del Message Server cuando se produce algún evento relacionado con él (por ejemplo la llegada un mensaje, un cambio en el Message Store, ..), lo cuál es muy útil para muchas aplicaciones

- CClientMtmRegistry: la aplicación usa esta clase para acceder a los servicios que proporciona la sesión con el Message Server desde el lado del cliente. Al crearse un objeto de esta clase se le indica la sesión con el Message Server a la que está asociado.
- CSmsClientMtm: Esta clase representa el lado cliente de la sesión con el Message Server. Los objetos de esta clases los crea una instancia de la clase CClientMtmRegistry indicándole el tipo de cliente que queremos obtener. En nuestro caso al tratarse de un cliente del método de mensajería SMS tendríamos que pasarle el identificador *KUidMsgTypeSMS*
- CMsvEntry: Esta clase encapsula una entrada del Message Store y proporciona métodos para manipularla.
- CMsvEntrySelection: Esta clase encapsula un array de objetos de la clase CMsvEntry. Se suele utilizar para almacenar los nodos hijos de una determinada entrada del Message Store

La estrategia seguir para la recepción de los mensajes será la siguiente:

- Utilizar el cliente SMS (clase CSmsClientMTM) para colocarnos apuntando a la carpeta “Inbox” del Message Store (la que contiene los mensajes recibidos)
- Obtener todos los nodos hijos del Inbox que sean mensajes de tipo SMS
- Iterar sobre esos hijos buscando aquellos mensajes cuyo cuerpo del mensaje comience con la palabra clave “Crypto”, y en ese caso introducirlos en la Bandeja de Entrada de nuestra aplicación para mostrarlos en caso de que el usuario lo solicite

- **Manejo de MMS**

Otro de los requisitos de nuestra aplicación es el envío de MMS, por lo cuál también hemos tenido que investigar la forma de acceder a ese servicio proporcionado por Symbian.

En un primer momento, y motivados por los conocimientos que habíamos adquirido cuando investigamos el uso de SMS, pensamos que la solución más simple sería usar Send As también para el envío de MMS. En base a esta idea comenzamos a desarrollar el código para el envío de MMS, resultando prácticamente idéntico al desarrollado para el envío de SMS pero con una pequeña particularidad: al crear el objeto RSendAsMessage que encapsula el mensaje usando el método CreateL(...) teníamos que pasar como identificador del tipo de mensaje la constante *KSenduiMtmMmsUid*, para indicar que queremos crear un mensaje de tipo MMS.

Sin embargo cuando comenzamos a realizar las primeras pruebas con el emulador nos llevamos la sorpresa de que esta implementación no funcionaba correctamente, ya que al intentar enviar el MMS el emulador nos lanzaba un error del tipo "Feature not supported". En un primer momento pensamos que la causa del error podría ser el propio emulador, ya que revisando la documentación del mismo comprobamos que no era capaz de emular todos los posibles servicios que ofrece un terminal móvil. Para comprobar si estábamos en lo cierto decidimos crear un ejecutable del código para el móvil e intentar ejecutar la aplicación. Sin embargo el resultado obtenido fue el mismo que con el emulador, así que en ese momento ya teníamos la certeza de que no era un problema del emulador y si más un problema que tenía que ver con la API utilizada.

A partir de ese momento centramos nuestras investigaciones en la búsqueda de una alternativa a Send As para el envío de MMS, obteniendo como resultado algo que ya nos era bastante familiar, el uso del Client / Server Framework de Symbian explicado anteriormente.

Además una vez que encontramos la manera correcta de implementar el envío de MMS usando este framework descubrimos otra característica del protocolo MMS que no habíamos tenido en cuenta hasta ahora, la naturaleza asíncrona del mismo. Básicamente este hecho se resume en que pueden existir potenciales retardos entre la solicitud de un servicio MMS al servidor y su correspondiente procesamiento (y por lo tanto

respuesta) por parte del servidor. Por ejemplo, puede producirse un cierto retardo entre que un cliente le solicita al servidor que envíe el contenido asociado a un MMS y la resolución completa del envío de ese contenido por parte del servidor. Para gestionar este comportamiento asíncrono del protocolo MMS no bastaba sólo con el Client / Server framework, sino que había que incorporar otro de los mecanismos más característicos de Symbian, el "Active Object Framework". Para una mejor comprensión, es necesario comentar más en detalle este framework antes de pasar a comentar las API´s usadas para el envío y recepción de MMS.

- **Active Object Framework**

  Symbian esta diseño de manera que las aplicaciones de usuario se caracterizan por estar constituidas por un único thread y por emplear la mayor parte del tiempo esperando a la ocurrencia o finalización de eventos asíncronos, como por ejemplo pulsaciones de teclas , etc, ... Además al tratarse de un sistema operativo para dispositivos móviles es desaconsejable el uso de aplicaciones multithread, ya que los mecanismos de planificación , protección de recursos compartidos, comunicación entre threads supondrían un sobrecoste en el tiempo ejecución y en el consumo de energía.

  Además al tratarse de aplicaciones con un único thread no resulta práctico que ese thread tenga que estar esperando a que se produzca / finalice un evento asíncrono, ya que durante ese tiempo la aplicación no podría hacer nada más (como manejar pulsaciones de tecla, repintar la pantalla, ...) e incluso desde el punto de vista del usuario podría parecer que la aplicación se ha quedado "colgada".

  Debido a que Symbian dispone de numerosas API´s asíncronas, y para manejar posibles situaciones como las que se acaban de mencionar, Symbian ofrece la posibilidad de utilizar el Active Object Framework.

  Básicamente su propósito es manejar la naturaleza asíncrona de Symbian a través de dos componentes principales:

  - Active Objects: los objetos de este tipo son subclases de la clase CActive y se usan para delegar en ellos la responsabilidad de manejar un evento asíncrono. En Symbian las API´s que tienen operaciones de comportamiento asíncrono se caracterizan por disponer de un parámetro del tipo TRequestStatus, el

cuál se asocia a ese tipo de operaciones que ofrece esa API. Concretamente, en el caso de que se invoque una de esas operaciones, ésta no se habrá completado cuando devuelva un valor, sino que lo hará cuando su correspondiente TRequestSatus tenga un valor distinto a la constante *KRequestPending*.

- Active Scheduler: es el encargado de planificar los eventos, es decir, en caso de que se produzcan varios eventos al mismo tiempo es el encargado de decidir cuál de ellos se atiende primero. Para ello cada Active Object que se crea lleva asociada una prioridad, que es usada por el Active Scheduler a la hora de planificar los eventos. Además es el encargado de notificar al Active Object adecuado que el evento que maneja ha finalizado. El sistema operativo crea uno por cada aplicación.

En particular, todas las clases que hereden de CActive deben implementar una serie de métodos de los cuáles los más importantes son los siguientes:

- RunL(): se ejecuta cuando se completa el evento que maneja ese Active Object. Esta función debe ser lo más pequeña posible para devolver lo antes posible el control al Active Scheduler
- DoCancel(): se usa para cancelar la petición asíncrona hecha por el usuario en caso de que sea necesario
- RunError(): se usa para llevar a cabo una posible liberación de recursos en caso de que la función RunL termine de forma inesperada (debido a algún tipo de error)
- SetActive(): es necesario invocarla cuando se llama a una operación asíncrona, ya que al invocar esta función el Active Object le está indicando al Active Scheduler que le avise cuando finalice la operación asíncrona que ha solicitado. Si no se invoca esta función el Active Scheduler no notificará nada al Active Object, y por tanto éste no tendrá constancia de cuando ha finalizado el evento ni podrá realizar el procesamiento asociado a esa finalización (el contenido de RunL())

El funcionamiento básico de este framework se puede entender de la siguiente manera: cada vez que una aplicación crea un objeto de la clase CActive lo añade al Active Scheduler de esa aplicación para que éste conozca su existencia. Mientras tanto el Active Scheduler ejecuta un bucle donde en cada iteración comprueba si hay algún evento finalizado. En caso de que lo haya elige el que tenga mayor prioridad y se lo notifica al Active Object correspondiente ejecutando su método RunL(). Nótese que para que esto ocurra el Active Object tuvo que ejecutar SetActive() cuando solicitó la operación asíncrona.

En este momento ya estamos en disposición de comenzar a comentar las API´s usadas para el manejo de MMS. En este caso hemos usado la misma API tanta para el envío como la recepción, la API MMS Client MTM API:

- **MMS Client MTM API**

  1. Descripción

  Esta API es prácticamente idéntica a SMS Client MTM API, con la única diferencia de que en este caso nos permite acceder a todos los servicios relacionados con la mensajería MMS proporcionados por el Message Server.

  2. Funcionamiento

  Es el mismo que en SMS Client MTM API, con la particularidad de que en este caso el cliente será un cliente específico del método de mensajería MMS.

  3. Clases Principales

  Usaremos las clases CMvsSession, CClientMtmRegistry y CMsvEntry de la misma forma que en SMS Client MTM API. Las únicas variaciones serán que en este caso en vez de usar la clase CSmsClientMtm usaremos en su lugar CMmsClientMtm y que incorporamos una nueva clase llamada CMvsStore :

  - CMmsClientMtm: Esta clase representa el lado cliente de la sesión con el Message Server. Los objetos de esta clases los crea una instancia de la clase CClientMtmRegistry indicándole el tipo de cliente que

queremos obtener. En este caso al tratarse de un cliente del método de mensajería MMS tendríamos que pasarle el identificador *KUidMsgTypeMultimedia*

- CMvsStore: encapsula los atributos específicos de una entrada del Message Store. Dispone de una serie de métodos para manipularlos.

A diferencia de lo que ocurre con SMS, el envío de MMS entraña un proceso más laborioso, debido principalmente a una serie de diferencias entre ambos protocolos:

- El protocolo MMS es de naturaleza asíncrona, mientras que SMS es de naturaleza síncrona. Por lo tanto para el envío de MMS es necesario el uso de Active Objects y del Active Scheduler
- En el protocolo MMS no existe el concepto de "cuerpo " del mensaje de la misma forma que en SMS. En este caso el cuerpo del mensaje se trata como un archivo de texto adjunto al mensaje MMS, mientras que en SMS el cuerpo del mensaje forma parte del propio mensaje. Visto desde el punto de vista del Message Store el cuerpo del mensaje de un MMS sería un nodo hijo del nodo que representa al mensaje, mientras que en SMS el cuerpo del mensaje estaría dentro del nodo que representa al mensaje

Todo esto hace que el proceso de creación de un MMS sea completamente diferente al de un SMS, y por lo tanto necesita una explicación más detallada que se ofrece a continuación:

1) La clase que tenga como atributos la sesión con el servidor (CMvsSession) y el cliente MMS del Message Server (CMmsClientMtm) debe ser subclase de CActive (Active Object) y por lo tanto implementar los métodos mencionados en la explicación del Active Object Framework. Sin embargo en este caso no hace falta que los métodos RunL() y DoCancel() tengan contenido específico, ya que el tipo de peticiones que haremos serán solicitudes de envío de MMS y por lo tanto una vez finalizado el envío no hace falta ningún tipo de procesamiento adicional.

2) Una vez abierta las sesión de manera asíncrona y creado el cliente MMS, el siguiente paso será crear el mensaje MMS. Para ello el primer paso es situarse en la carpeta "Drafts folder" (ya se ha hablado de ella). Esto es posible debido a

que el cliente MMS dispone de métodos para situar un apuntador en una determinada entrada del Message Store, e incluso obtener su información en forma de un objeto de la clase CMsvEntry. Además existe un archivo de cabecera llamado <msvids.h> donde están definidas una serie de constantes que identifican las carpetas especiales existentes en el Message Store. En este caso usaremos la constante *KMsvDraftEntryId,* que hace referencia a la carpeta "Drafts folder". Para situarnos en esa carpeta usamos el siguiente método del cliente MMS:

- void CMmsClientMtm::SwitchCurrentEntryL(TMsvId aId): sitúa el apuntador del cliente a la entrada del Message Store indicada por el identificador aId. El tipo TMsvId al que pertenece el identificador está declarado en el archivo de cabecera <msvids.h>.

Una vez situados en esa carpeta ya podemos crear el mensaje. Para ello usamos el siguiente método del cliente MMS:

- void CMmsClientMtm::CreateMessageL(TMsvId aServiceId): este método crea un nuevo mensaje MMS vacío, y lo sitúa en el Message Srore como un nodo hijo de la entrada a la que apunta el cliente MMS. Por lo tanto en nuestro caso estamos creando un mensaje dentro de la carpeta "Drafts Fólder". El identificador de servicio aServiceId lo obtenemos a través del método DefaultServiceL() del cliente MMS, que devuelve el servicio por defecto que en este caso es MMS

3) Llegados a este punto el mensaje ya está creado en la carpeta Drafts Fólder, sin embargo, al estar el mensaje en construcción no es deseable que sea visible para un usuario que revisa esa carpeta o para otra aplicación que tenga acceso a ella. Para evitarlo podemos usamos los siguientes métodos del cliente MMS:

   - CMsvEntry CMmsClientMtm::Entry(): obtiene la entrada del Message Store a la que apunta el cliente MMS
   - TMsvEntry CMsvEntry::Entry(): obtiene los atributos genéricos de una determinada entrada del Message Store.
   - void TMsvEntry::SetInPreparation(TBool aId): sirve para marcar una entrada del Message Store como "en

construcción". Recibe como argumento un booleano que permite activar / desactivar (ETrue / EFalse) ese flag de la entrada

- void TMsvEntry::SetVisible(TBool aId): sirve para marcar como no visible/visible (ETrue/EFalse) una determinada entrada del Message Store. Si una entrada está marcada como no visible, no podrá ser examinada por ningún otro usuario o aplicación.

Hay que comentar que una vez que el cliente MMS crea el mensaje deja de apuntar a la entrada correspondiente a la carpeta "Drafts Folder" y pasa a apuntar al nuevo mensaje creado. Debido a esto una vez creado el mensaje podemos obtenerlo con el puntero del cliente MMS, obtener sus atributos genéricos y marcar el mensaje como en construcción y no visible. Por último hay que guardar los cambios aplicados a esa entrada del Message Store utilizando el método ChangeL() de la clase CMsvEntry y el método SaveMessageL() del cliente MMS.

4) Ahora ya podemos empezar a rellenar el mensaje, así que lo primero que haremos será asignarle un destinatario. Para ello usaremos el siguiente método del cliente MMS:

   - void CMmsClientMtm::AddAddreesseL(const TDesC &aAddress, TMsvRecipientType aRecipientType): añade un destinatario mediante un número de teléfono almacenado en un descriptor de texto (aAddress) y un tipo de destinatario, que en este caso será la constante EMsvRecipientTo para indicar que es un destinatario normal (se debe añadir a la lista de destinatarios del mensaje).

5) El siguiente paso es añadir el cuerpo del mensaje. Ya hemos comentado que en MMS el cuerpo del mensaje es un archivo de texto adjunto al mensaje, así que lo primero que haremos será crear ese adjunto mediante el siguiente método del cliente MMS:

   - void CMmsClientMtm::CreateTextAttachmentL(CMsvStore& aStore, TMsvAttachmentId & aAttachmentId, const TDesC& aText, const TDesC& aFile, TBool aConvertParagraphSeparator = ETrue): este método

crea un archivo de texto adjunto al mensaje a partir de un descriptor de texto que almacena el contenido del cuerpo del mensaje y que se le pasa como argumento (aText). Además es necesario incluir como parámetros un puntero a los atributos específicos del mensaje (aStore), el cuál se consigue obteniendo la entrada a la que apunta el cliente MMS y posteriormente usando el método EditStoreL() de esa entrada. El resto de parámetros son un nombre para el archivo de texto adjunto (aFile) y un identificador de adjunto (aAttachmentId), ambos a elegir por el programador.

6) El último paso es enviar el mensaje y notificarle al Active Scheduler que nuestro Active Object ha solicitado una operación asíncrona (el envío del mensaje). De esta manera cuando finalice el envío del mensaje el Active Scheduler se lo comunicará a nuestro Active Object ejecutando su método RunL() (tal como explicó al comentar el Active Object Framework). Para enviar el mensaje utilizamos el siguiente método del cliente MMS SendL, que recibe como parámetro el atributo TRequestStatus asociado a nuestro Active Object (se comentó en la explicación del Active Object Framework) . Por último para notificar al Active Scheduler que hemos solicitado una operación asíncrona usamos el método SetActive() del Active Object. Hay que comentar que justo antes de enviar el mensaje deberíamos quitarle la marca de "en construcción" y hacerlo visible, guardando por supuesto los cambios en esa entrada del Message Store.

Una vez explicado el proceso de envío de MMS ya sólo queda comentar cómo sería la recepción. Debido a que seguimos la misma estrategia que con los SMS, usaremos también la API MMS Client MTM. La estrategia a seguir sería:

1) Utilizar el cliente MMS para colocarnos apuntando a la carpeta "Inbox" del Message Store (la que contiene los mensajes recibidos)
2) Obtener todos los nodos hijos del Inbox que sean mensajes de tipo MMS
3) Iterar sobre esos hijos buscando aquellos mensajes cuyo cuerpo del mensaje comience con la palabra clave

"Crypto", y en ese caso introducirlos en la Bandeja de Entrada de nuestra aplicación para mostrarlos en caso de que el usuario lo solicite

3.1.3 Implementación

La implementación de los métodos más importantes, que son envío y recepción de SMS y MMS se realizó en las clases CMensajeContainer y CMMSSvervice. Hay que comentar que la clase CMMSServive es subclase de CActive como se indicó en el apartado API, de manera que la clase CMensajeContainer delega en ella para el envío de los MMS.

Los métodos donde se realizan los envíos son los siguientes :

```
void EnviarSMSL(TDes& aTel, TDes& aKey);
void EnviarMMSL();
void MostrarMensajeL();
SendMMSL(TDesC& aMessage, TDesC& aNumber)
```

El Envío de SMS

Para realizar un envío de SMS se necesitan un número de teléfono del destinatario y una clave para realizar la encriptación del contenido del mensaje, que son respectivamente aTel y aKey. El método coge el contenido del SMS en el editor con:

```
iEditor->GetText(SMSText);
```

Después se crea un objeto de criptografía:

```
CriptoCryptoAES* iCryptoaes = CriptoCryptoAES::NewL();
```

Usando ese objeto y la clave se encripta el SMS:

```
iCryptoaes->setAesKey(aeskey);
iCryptoaes->EncryptAesL(txt0,SMSTextEncrypted);
```

El resultado de esta encriptación es una cadena cuyo contenido incluye numerosos caracteres especiales, lo cuál es problemático para el envío con la clase RSendAs, ya que esta última solo envía caracteres en formato de 7-bits-SMS. Por lo tanto se necesita realizar una conversión previa, más concretamente codificar el mensaje en Base64, para posteriormente enviarlo correctamente:

```
CnvUtfConverter iConverter;
iConverter.ConvertFromUnicodeToUtf8( utftmp, SMSTextEncrypted );
HBufC8* crypted = HBufC8::NewLC(SMSTextEncrypted.Size());
```

```
TPtr8 tmp2 = crypted->Des();
tmp2.SetLength(SMSTextEncrypted.Size());
TImCodecB64 MiBase64;
MiBase64.Initialise();
MiBase64.Encode(utftmp,tmp2);
```

Una vez preparado el mensaje, el objeto serv de tipo RSendAs establece la conexión con el Message Server a través del método serv.Connect(), y una vez establecida la conexión, se define el objeto msg de tipo RSendAsMessage. Este ultimo crea el mensaje, define el tipo de mensaje (KSenduiMtmSmsUid, "SMS"), asigna el número del destinatario con el método AddRecipientL(), pone el cuerpo de mensaje, y finalmente envía el SMS.

```
RSendAs serv;
 if ( serv.Connect() == KErrNone )
 {
        CleanupClosePushL(serv);
        RSendAsMessage msg;
        CleanupClosePushL(msg);
        msg.CreateL(serv, KSenduiMtmSmsUid);


 msg.AddRecipientL(aTel,RSendAsMessage::ESendAsRecipientTo);
        msg.SetBodyTextL(TextoEnviar);
        msg.SendMessageAndCloseL();
        CleanupStack::Pop(&msg);
        }
Else {  CleanupClosePushL(serv);    }
```

El Envío de MMS

El método EnviarMMSL() de la clase CMensajeContainer realiza una llamada al método SendMMSL(TDesC& aMessage, TDesC& aNumber) de la clase MMSService, pasándole como parámetros el mensaje a enviar y el número del destinatario.

La clase CMMSService, que hereda de de la clase CActive, posee 3 atributos :

```
CMsvSession*    iSession;
CClientMtmRegistry*  iMtmReg;
```

```
CMmsClientMtm* iMmsMtm;
```

Estos atributos serán utilizados para el envío de MMS dentro del método SendMMSL():

```
iMmsMtm->SwitchCurrentEntryL(KMsvDraftEntryId);
iMmsMtm->CreateMessageL(iMmsMtm->DefaultServiceL());
iMmsMtm->AddAddresseeL(EMsvRecipientTo,aNumber);
```

Como se puede ver el cliente MMS del Message Server sitúa su apuntador a la entrada "Drafts folder" del Message Store, crea un nuevo mensaje que será de tipo MMS, y asigna el número de teléfono del destinatario de ese MMS.

```
TMsvEntry ent = iMmsMtm->Entry().Entry();
ent.SetInPreparation(EFalse);
ent.SetVisible(ETrue);
iMmsMtm->Entry().ChangeL(ent);
iMmsMtm->SetMessagePriority(EMmsPriorityHigh);
iMmsMtm->SaveMessageL();
```

En la variable ent de tipo TMsvEntry guarda los atributos genéricos de la entrada del Message Store a la que apunta el cliente MMS, modifica alguna propiedad como por ejemplo poner "en construcción" a falso y poner la propiedad visible a cierto. Por último se cambia la prioridad del mensaje y se guardan los cambios realizados usando el método iMmsMtm->SaveMessageL().

```
CMsvStore* store = iMmsMtm->Entry().EditStoreL();
CleanupStack::PushL(store);
TMsvAttachmentId theAttachId = KMsvNullIndexEntryId;
_LIT(KMMSTextFilename, "hello.txt");
iMmsMtm->CreateTextAttachmentL(*store, theAttachId ,aMessage
,KMMSTextFilename);
store->CommitL();
iMmsMtm->SaveMessageL();
```

Como cuerpo del mensaje MMS será un archivo de texto adjunto, se crea dicho archivo adjunto usando el método iMmsMtm->CreateTextAttachmentL(...), y posteriormente se guardan los cambios realizados en el mensaje con.

El paso final de envío de MMS es:

```
iOp=iMmsMtm->SendL(iStatus);
SetActive();
```

Como se puede ver se envía el MMS, y al ser una operación asíncrona, hay que notificárselo al Active Scheduler con el método SetActive().

Visualización de Mensajes

Las clases CBandejaContainer y CBandejaMMSContainer se encargan extraer los mensajes SMS y MMS respectivamente del Message Store. En cuando la implementación esas dos clases son muy parecidas porque debido a la estructura de Message Store es independiente del tipo de mensajes a buscar. Aquí solo se explica el caso de SMS, el caso de MMS es parecido solo sustituyendo los constantes de SMS por las de MMS.

El método más importante de CBandejaContainer es :

```
void CBandejaContainer::GetFolderSMSMessageInformationL(CDesCArrayFlat*
arrayAddr, CDesCArrayFlat* arrayMsgBody)
```

Su misión principal es sacar en el Message Store todos los SMS que contengan la marca especial:

```
iSmsMtm->SwitchCurrentEntryL( KMsvGlobalInBoxIndexEntryId );
CMsvEntry& entry = iSmsMtm->Entry();
CMsvEntrySelection* entries = entry.ChildrenWithMtmL ( KUidMsgTypeSMS );
```

La variable iSmsMtm de tipo CSmsClientMtm pone su apuntador en la entrada del Message Store que representa a la carpeta Inbox. Después selecciona únicamente los nodos hijos de tipo SMS. Es importante llamar al método LoadMessageL() para cargar los datos del mensaje.

```
for (TInt i = 0; i < entries->Count(); i++ )
        {
        iSmsMtm->SwitchCurrentEntryL( (*entries)[i] );
        iSmsMtm->LoadMessageL();
        listString.Copy( iSmsMtm->Entry().Entry().iDescription );
        if ( listString.Left(6) == KHeader )
                {
                listString.Copy( iSmsMtm->SmsHeader().FromAddress() );
                listString.Insert( 0, KStringHeader );
                AddListBoxItemL( iListBox, listString );
                iSmsMtm->Body().Extract(listString,0);
                arrayMsgBody->InsertL ( 0, listString);
                arrayAddr->AppendL ( iSmsMtm->SmsHeader().FromAddress() );
                }
        }
```

Se realiza un recorrido de las entradas que son SMS, se comprobando si el mensaje contiene la marca especial guardada en el literal KHeader. En caso afirmativo, se guarda el cuerpo del mensaje en un array de texto, el arrayMsgBody, y el número de teléfono del remitente en el arrayAddr, un array de números de teléfono.

Después de extraer los mensajes del Inbox ya se puede realizar la visualización del contenido de los mismos. El método de MostrarMensajeL() de la clase CMensajeContainer se encarga de realizar el proceso inverso del que se hizo en el método enviar, es decir, decodificar, desencriptar, y finalmente mostrar en el editor.

```
TBuf8<256> utftmp;
CnvUtfConverter iConverter;
iConverter.ConvertFromUnicodeToUtf8( utftmp, iMensaje );
    HBufC8* crypted = HBufC8::NewLC(iMensaje.Size());
    TPtr8 tmp2 = crypted->Des();
    tmp2.SetLength(iMensaje.Size());
    TImCodecB64 MiBase64;
    MiBase64.Initialise();
    MiBase64.Decode(utftmp,tmp2);
    iConverter.ConvertToUnicodeFromUtf8(Decodificado,tmp2);
iCryptoaes->DecryptAesL(Decodificado,SMSTextDecrypted);
iEditor->SetTextL(&SMSTextDecrypted);
```

4 Entorno de Desarrollo

Para empezar a desarrollar aplicaciones con el sistema operativo symbian es necesario también definir que lenguaje se va a usar, el caso del proyecto es C++. Si se quiere utilizar C++ hace falta instalar los siguientes componentes:

- Active Perl 5.6.1.635: Active Perl es requerido por Carbide, su versión actual es la 5.10.0.1002, no obstante, hasta esta versión y desde la 5.6.1 es incompatible con Carbide.
- Carbide C++ v1.3: hay distintas ediciones dentro de cada versión, dependiendo de la que se escoja se requerirá de un registro gratuito o previo pago de un importe. La edición Express limita las herramientas que pueden utilizarse pero permite la construcción de aplicaciones. Sin embargo nosotros hemos utilizado la versión Professional
- JRE 1.4.2: Es requerido por la herramienta NCF que se explicará mas adelante.
- Symbian C++ S60_SDK_3rd_Edition: paquete de desarrollo de software proporcionado por Nokia y que permite utilizar las librerías estándar de Symbian, además de suministrar un emulador para probar las aplicaciones antes de pasarlas a un terminal real.
- Nokia Connectivitty Framework: para simular el envío de mensajes entre distintos emuladores.

4.1 Herramientas de desarrollo

Estructura de un programa en Symbian

Todo proyecto que se quiera crear con Carbide C++ para un SDK determinado utiliza una serie de archivos y una estructura de directorios básica para la creación del instalable. Si bien es posible alterar el orden que propone Symbian añadiendo o uniendo distintos archivos.

Hay 6 directorios básicos en todo proyecto Symbian: data, inc, src, gfx, group y sis.

En la carpeta data se suelen incluir los archivos con extensión .rls y .rss. Los archivos .rls contienen el texto que se suele usar para etiquetas, opciones, etc. Es el texto que se va a visualizar en la pantalla. Las cadenas de texto se deben parametrizar con una etiqueta, de esta manera es fácil definir distintos .rls para los distintos idiomas que se quiera tener en la aplicación. Los archivos .rss son los archivos de recursos. Un archivo de recurso especifica estructuras de interfaz de usuario como CBA (Command button Area), Status Pane (panel de estado), menús, listbox, etc.

En la carpeta inc se encuentran los archivos .h y los archivos .hrh. Los archivos .h son los archivos de cabecera de código y los archivos .hrh contienen identificadores, como por ejemplo identificadores de vistas, comandos, etc.

La carpeta src contiene los archivos de código .cpp.

La carpeta gfx almacena los archivos gráficos .svg, .bmp, etc.

La carpeta group es una de las carpetas principales, ya que contiene los archivos .mmp, .mk y .inf. El archivo .inf es el archivo principal que dice cómo construir el proyecto global y para qué plataforma. Incluye los archivos .mmp y .mk. Los archivos .mk dicen que gráficos se incluyen, sus características y en qué directorio se almacenan para su posterior carga en memoria. El archivo .mmp es el archivo de especificación del proyecto, incluye el identificador de la aplicación, del desarrollador, el tamaño por defecto para la pila de ejecución, los archivos de recursos que se utilizarán, las librerías que se incluyen, permisos o capacidades que requiere la aplicación, lenguajes de la aplicación y el código a compilar.

Y por último la carpeta .pkg que contiene los archivos .sis, .sisx y .pkg. El archivo .pkg especifica como se construye el archivo de instalación .sis, en que ruta dentro del terminal móvil se almacena cada archivo adicional, las plataformas que soporta la aplicación, y características del instalador. El archivo .sis es el archivo de instalación de la aplicación en un terminal. A partir de la tercera edición de la serie 60 este .sis debe ir

firmado para poder instalarse en el móvil. El archivo .sisx es un archivo de instalación firmado, no es necesario que tenga extensión .sisx pero es una estratagema que utiliza Carbide para diferenciar entre .sis y .sisx.

4.1.1 Plataforma

Symbian

Symbian es un sistema operativo que fue producto de la alianza de varias empresas de telefonía móvil, entre las que se encuentran Nokia, Sony Ericsson, PSION, Samsung, Siemens, Arima, Benq, Fujitsu, Lenovo, LG, Motorola, Mitsubishi Electric, Panasonic, Sharp, etc. Sus origenes provienen de su antepasado EPOC32, utilizado en PDA's y Handhelds de PSION.

Symbian cuenta con cinco interfaces de usuario o plataformas para su sistema operativo, las denominadas Serie 60, Serie 80, Serie 90, UIQ y FOMA. La mayoría de los móviles utilizan la Serie 60, todos los de Sony Ericsson trabajan bajo UIQ, así como también Motorola.

La plataforma para la que se ha desarrollado la aplicación es la serie 60. La plataforma S60 (formalmente, Interfaz de usuario de serie 60) consiste en un conjunto de bibliotecas y aplicaciones informáticas estándar, tales como telefonía, herramientas de gestión de información personal, y reproductores multimedia Helix. Está pensada para potenciar terminales móviles modernos de amplias características, con pantallas a color muy grandes, que son conocidos comúnmente como terminales smartphone. Se han ido proporcionando nuevas librerías hasta alcanzar la tercera edición y se han ido proporcionando funcionalidades en forma de paquetes FP (Feature pack), actualmente la versión mas moderna es la serie 60 3ª edición Feature Pack 2, aunque no han lanzado ningún terminal que soporte esta edición por el momento.

4.1.2 Herramientas

On-Device Debug

Los emuladores de los SDK's emulan en entornos de ejecución similares a dispositivos reales, pero no pueden reproducir exactamente el entorno de ejecución en un dispositivo real por diversas razones. Por lo tanto existen errores que solo aparecen cuando se prueba la aplicación en un terminal real, entones para estos casos surge la necesidad de un depurador que trabaje directamente en un dispositivo real.

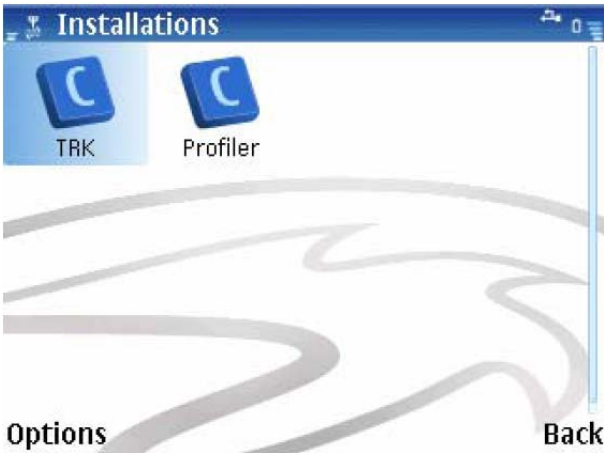
En las versiones de Carbide C++ Developer, Professional, y OEM se dispone de la opción On-Device Debugging.

Preparación para On-Device Debugging

Antes de poder usar el depurador con Carbide C++, es necesario instalar la aplicación TRK en el dispositivo destino y este último debe estar conectado al PC de desarrollo a través de una conexión serie.

Instalación de la Aplicación TRK

La aplicación es un archivo .sisx que está dentro de la carpeta: <directorio_de_carbide>\plugins\com.nokia.carbide.trk.support_1.3.0.024\trk\s60. Se instala la aplicación con el entorno PC Suite de Nokia.

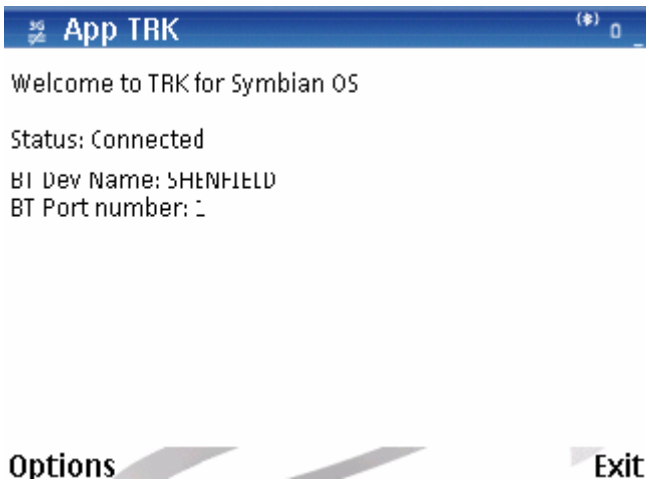


Configuración de la Conexión

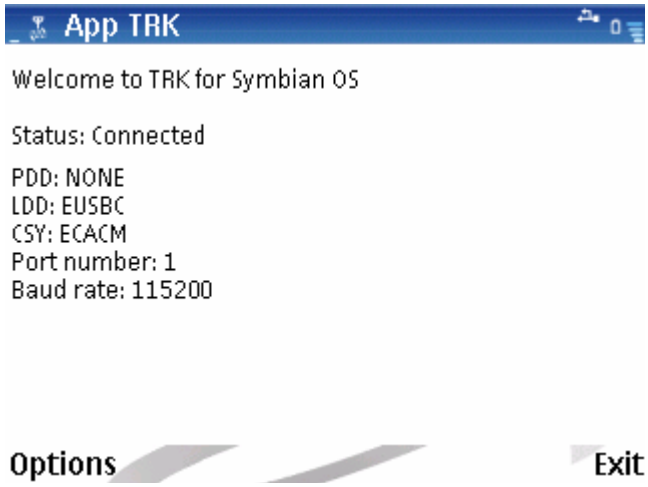
Existen dos vías para conectar el dispositivo al PC, por cable USB o por Bluetooth, en cualquiera de los dos casos hay que tener especialmente en cuenta el número de puerto COM que está siendo utilizado para después poder configurarlo correctamente en Carbide C++. Dicho puerto se puede comprobar en el Device Manager (administrador de dispositivos) de Windows.

En el dispositivo se ejecuta la aplicación TRK, pudiendo elegir la conexión por cable o por bluetooth, dependiendo de la selección aparecen diferentes situaciones.

Bluetooth:

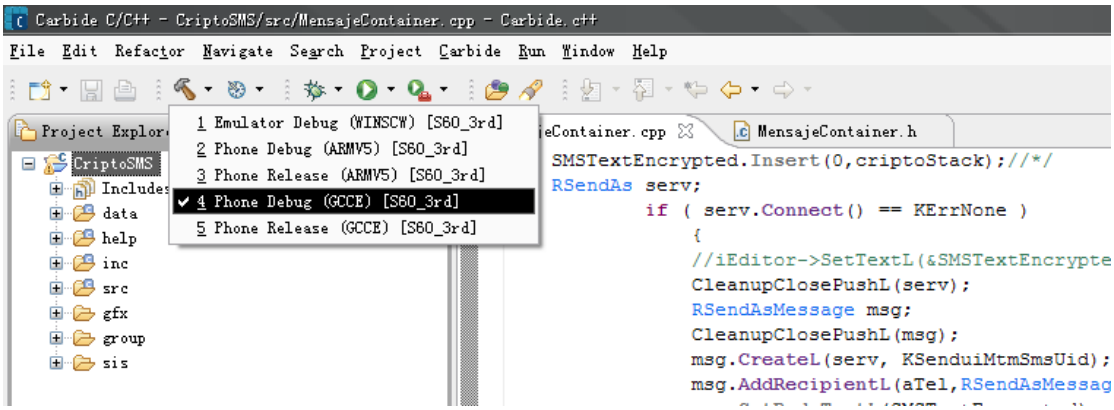


Cable:



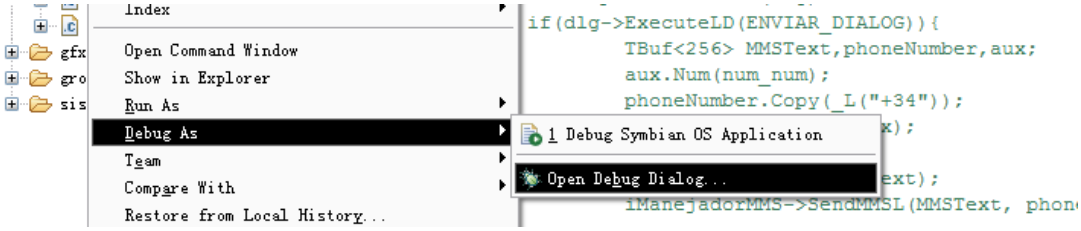
Compilación del Proyecto

Antes de realizar este paso hay que tener en cuenta que es necesario firmar el paquete de instalación, seleccionando Phone Debug para generar el .sis y .sisx:

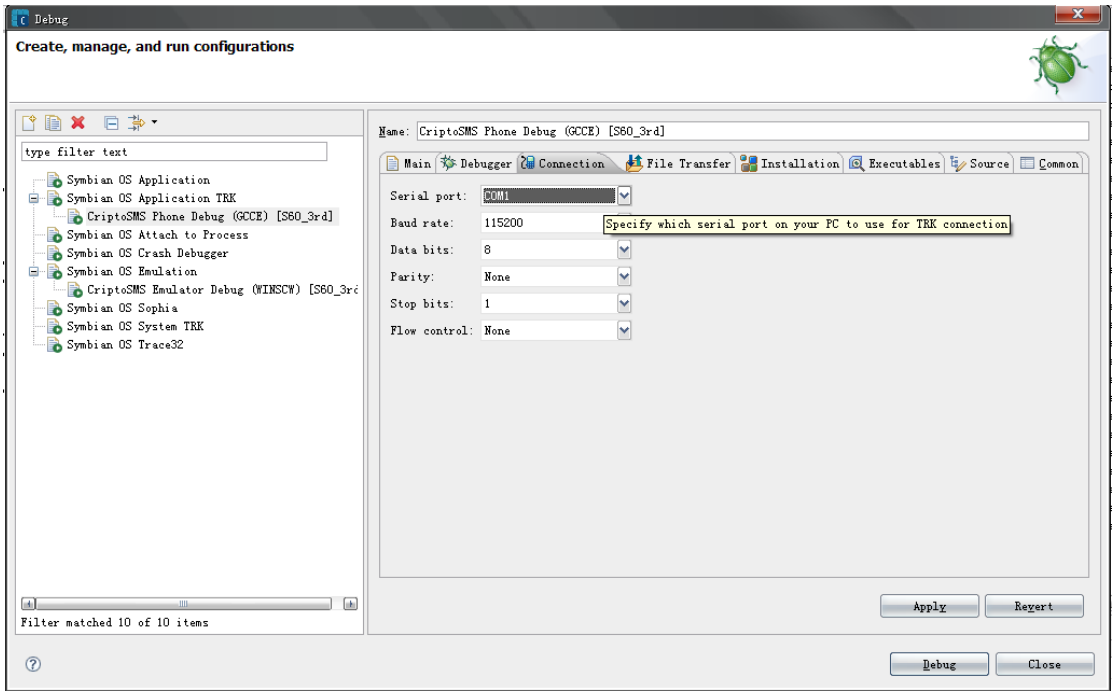


Configuración de Carbide C++

Antes de lanzar el debug, es conveniente verificar la configuración de debug. Para eso se abre el dialogo de debug:

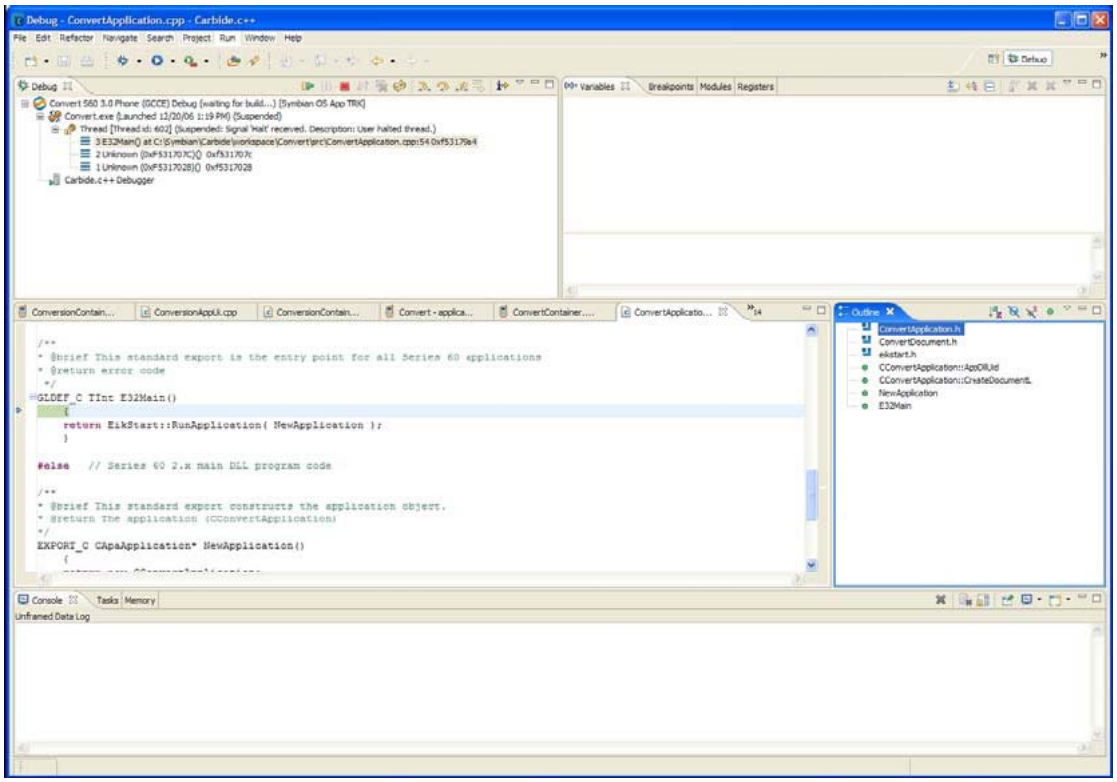


En la parte izquierda de esta ventana seleccionar "Symbian OS Application TRK", aquí se debe prestar especial atención a la pestaña de configuración de conexión, el puerto serie debe coincidir con el puerto de conexión del dispositivo.



Ejecutar On-Device Debug

Después de haber ajustado las configuraciones, al ejecutar el debug en Carbide C++ se ejecutará automáticamente TRK en el dispositivo conectado al PC. Entonces se puede depurar la aplicación como si se estuviera ejecutando en el emulador, como se puede ver en la siguiente figura:


Debug - ConvertApplication.cpp - Carbide.c++
File Edit Refactor Navigate Search Project Run Window Help
Debug
Variables Breakpoints Modules Registers
Carbide.c++ Debugger
ConversionContain... ConversionAppUi.cpp ConversionContain... Convert - applica... ConvertContainer.... ConvertApplicatio...
Outline
ConvertApplication.h
ConvertDocument.h
eikstart.h
CConvertApplication::AppDllUid
CConvertApplication::CreateDocumentL
NewApplication
E32Main
/**
* @brief This standard export is the entry point for all Series 60 applications
* @return error code
*/
GLDEF_C TInt E32Main()
{
return EikStart::RunApplication(NewApplication);
}
#else // Series 60 2.x main DLL program code
/**
* @brief This standard export constructs the application object.
* @return The application (CConvertApplication)
*/
EXPORT_C CApaApplication* NewApplication()
{
Console Tasks Memory
Unframed Data Log

4.1.3 Sistema de Gestión de Base de Datos

La aplicación tiene la necesidad de almacenar de forma permanente en el dispositivo los datos de los contactos que constituyen la agenda de la aplicación: 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 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, de un lenguaje de manipulación de datos y de 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 bases de datos 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 éstas están compuestas por campos.

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

Creación de una Base de Datos:

Existen dos API's 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 necesitamos 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 vamos a concederle permisos de lectura y escritura sobre él. Por último creamos la única tabla que va a tener nuestra 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. Necesitamos 4: el nombre, el número de teléfono, la clave pública y un flag que nos indique si tiene clave o no. Todas son de tipo texto. Una vez declarados, añadimos 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 primero abrirla indicando el nombre del fichero que la alberga y declarando qué permisos queremos sobre ella. Intentar abrir una base de datos que ya esté abierta provoca un fallo en la aplicación que provoca su terminación.

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 antes, existen dos maneras de insertar datos: usando sentencias SQL o usando métodos nativos. Queríamos experimentar las llamadas nativas que ofrece Symbian, por lo que decidimos usar éste último. Indicamos a la tabla que queremos insertar datos usando table. Insert(). Para rellenar los campos de cada columna usamos SetColL(), pasándole como parámetros la columna y el dato que queremos introducir. Por último actualizamos los cambios usando PutL().

4.2 Nokia Connectivity Framework

Nokia Connectivity Framework 1.2 (NCF) es una herramienta para visualizar, construir, mantener, y modificar un entorno de emulación basado en los emuladores del SDK. NCF facilita la conectividad entre dos emuladores de Nokia Developer Platform SDK, entre un emulador y un dispositivo o entre un emulador y un servidor de aplicación. De esta manera ayuda a la creación y prueba de aplicaciones de comunicación sobre dispositivos Nokia. Soporta tecnología Bluetooth, Short Message Service (SMS), y Multimedia Messaging Service (MMS).

El NCF monitoriza las comunicaciones entre los componentes en un entorno de prueba, proporcionando un extenso registro de actividades.

El NCF resulta muy práctico para probar el envío y recepción de los mensajes antes de llevarlo a dispositivos reales. También se utiliza para realizar pruebas de juegos de multijugador , aplicaciones de Chat, seguimiento de trazas en una interacción cliente/servidor...

Instalación de NCF 1.2

En instalador se puede descargar del siguiente link:

http://sw.nokia.com/id/fb0180d3-5b0b-43e4-8792-147488267c2d/ncf_v1_2.zip (http://www.forum.nokia.com)

Nota : es necesario registrarse en Forum Nokia para descargar el software.

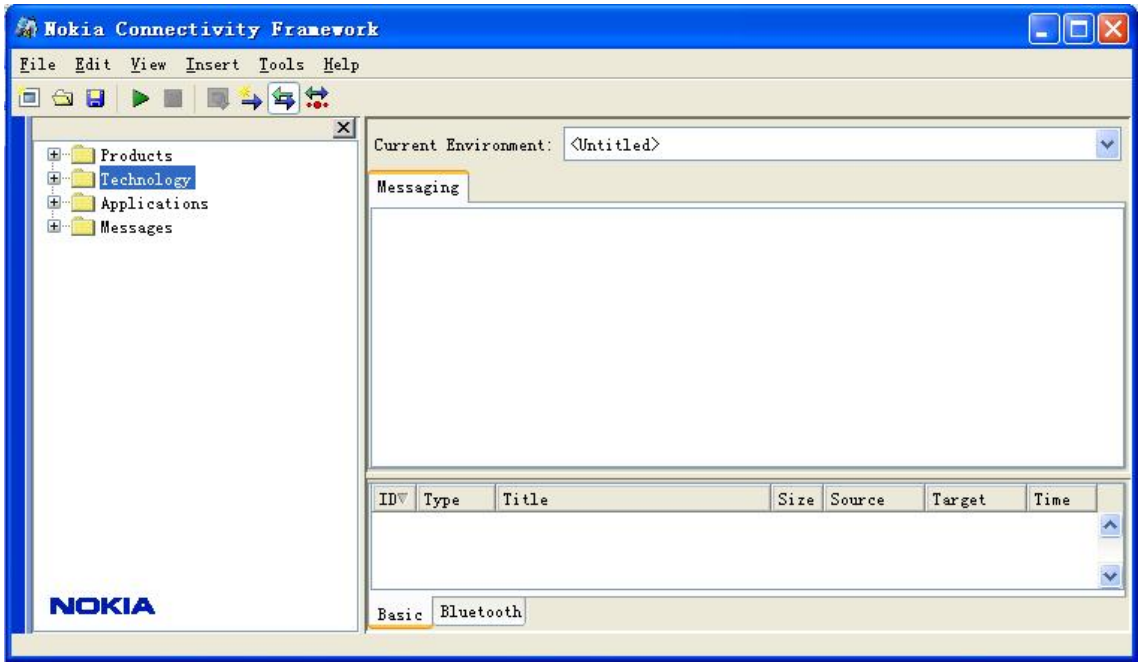
Se pedirá un número de serial para la instalación, ese número será proporcionado por Forum Nokia.

Funcionamiento de NCF 1.2

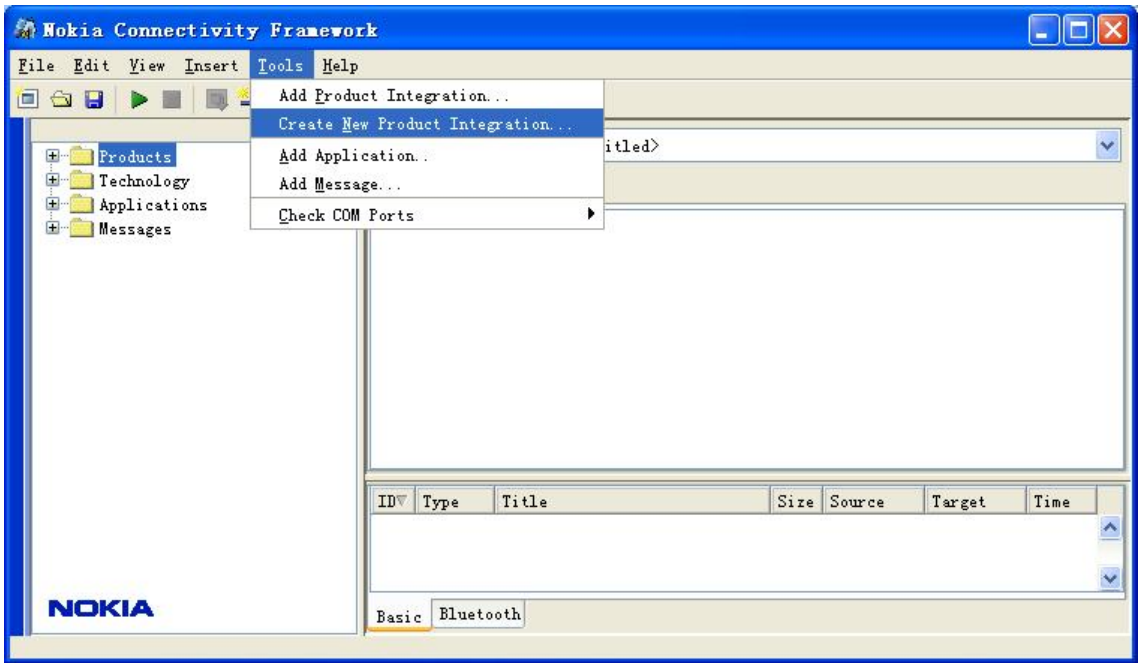
Antes de explicar el funcionamiento de NCF es necesario comentar que es posible tener instalados en un mismo equipo varios SDK's, tanto versiones diferentes como varias instancias de la misma versión. Sin embargo, sólo una de ellas será la instalación a usar por defecto.

Una vez instalado, NCF detecta la versión a usar por defecto y crea lo que él llama un "Product Integration" para esa versión. Un Product Integration viene a ser una especie de terminal virtual que estará disponible para ser usado en el entorno de emulación del NCF. Dicho esto pasamos a comentar más detalladamente el funcionamiento de NCF:

Al ejecutar NCF 1.2 aparece la siguiente ventana.

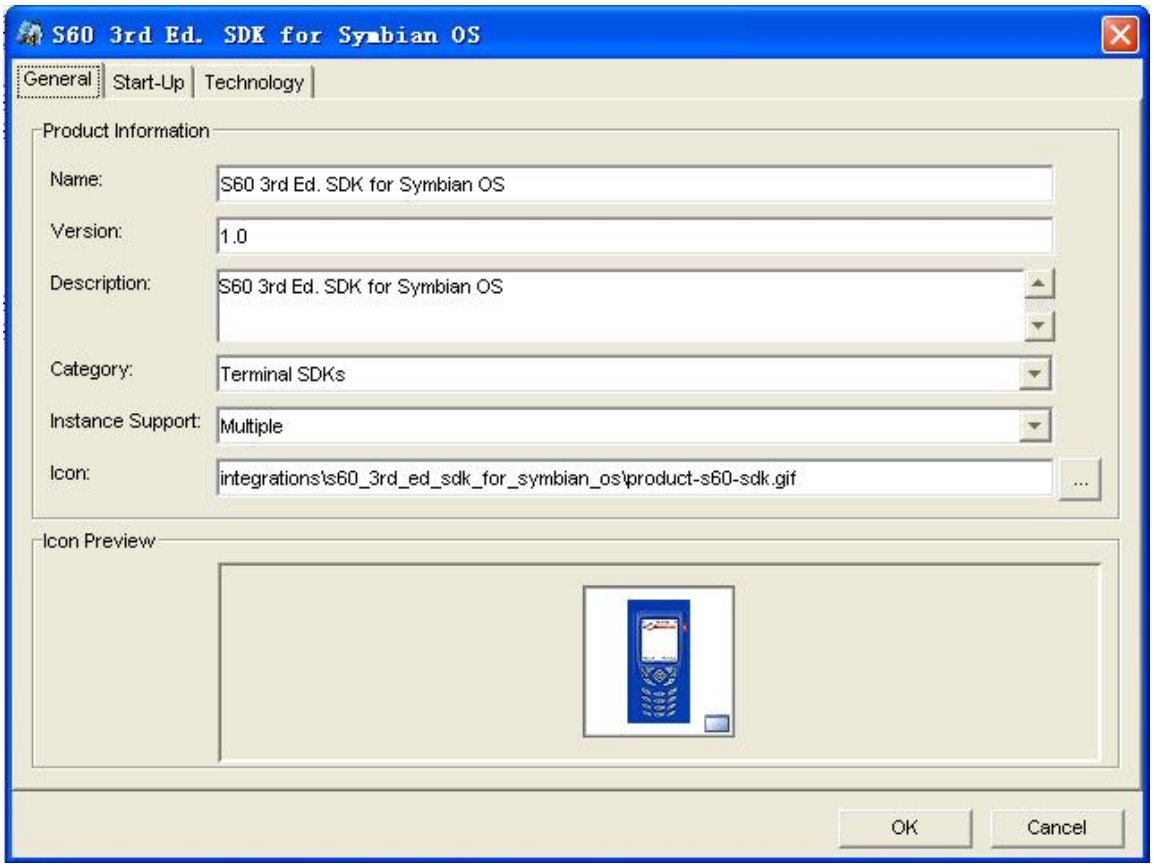


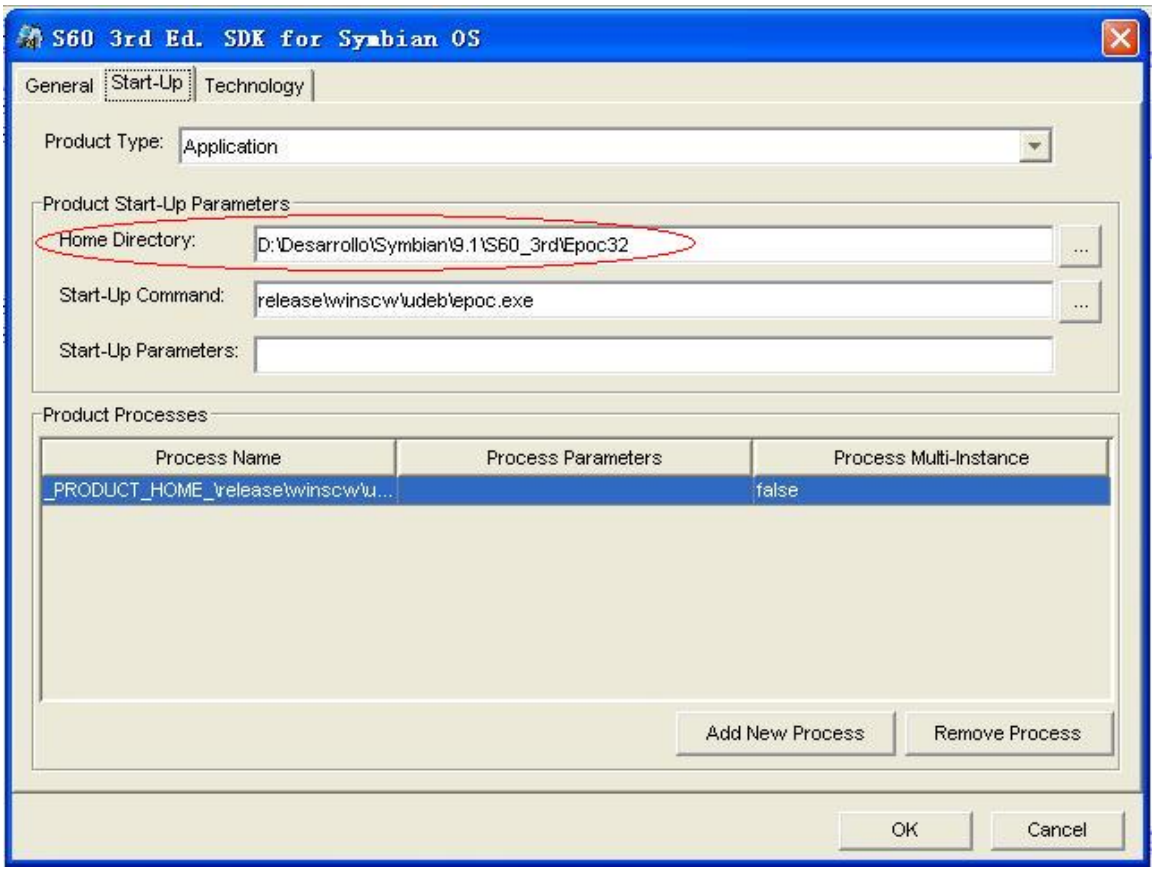
Para ejecutar dos instancias del emulador a la vez, es necesario añadir un nuevo "Product Integration" (ya que hasta ahora sólo disponemos del que creó NCF por defecto):

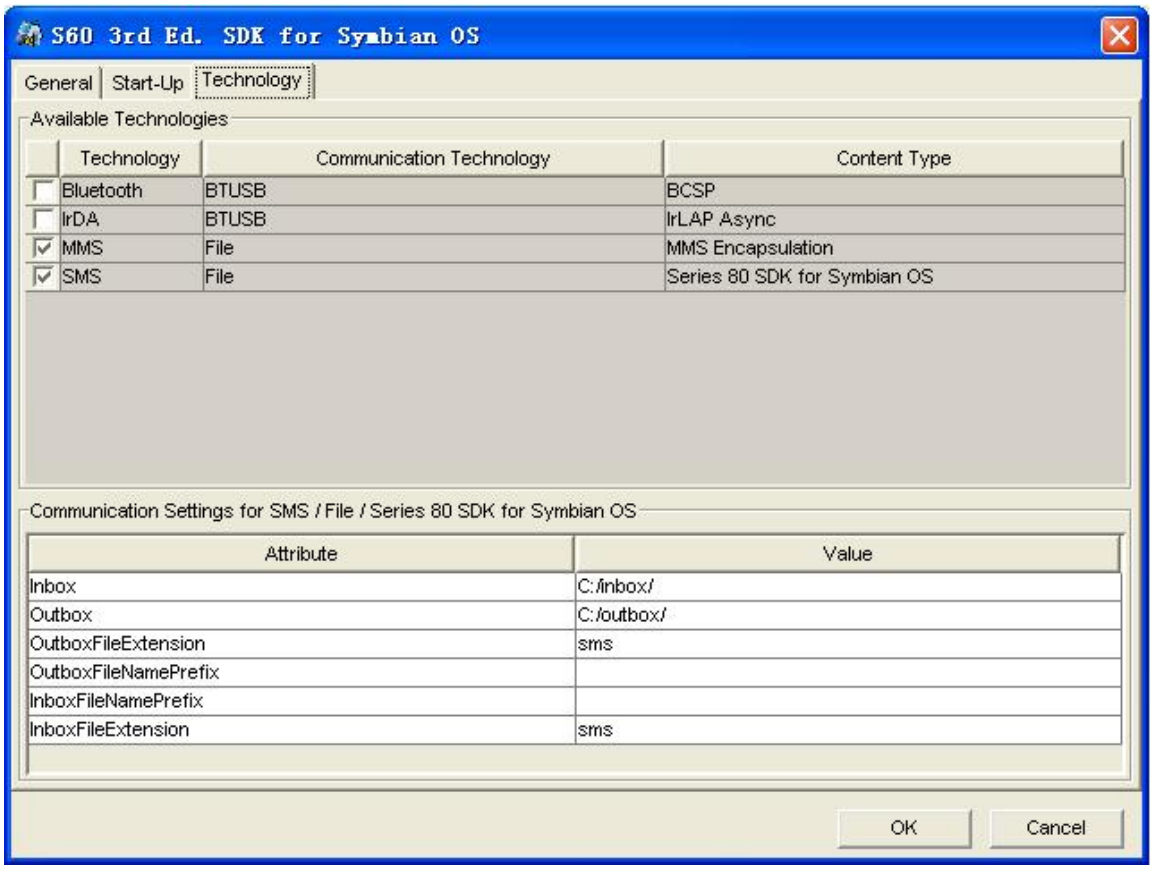


Ajustamos las propiedades del nuevo "Product Integration" al ya existente, copiando casi todos los campos, excepto el valor de Home Directory en la pestaña de Stara-Up (marcado con un circulo rejo en la

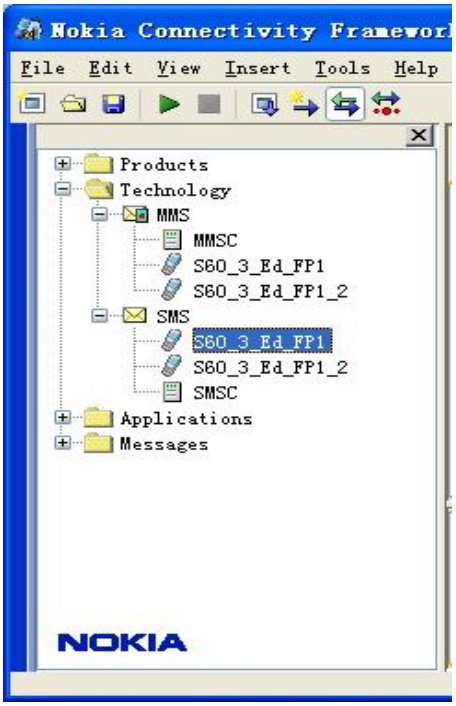
figura), hay que cambiar el valor de ese campo poniendo el directorio del SDK. En las siguientes figuras podemos ver una posible configuración.



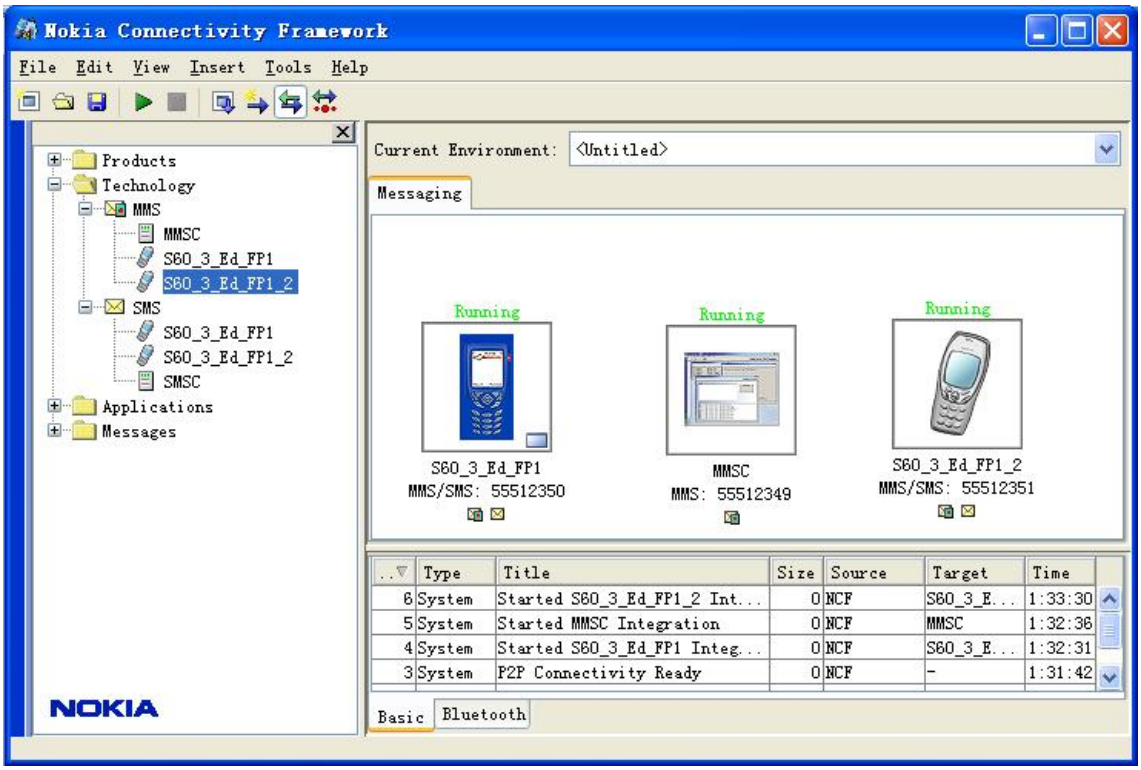

S60 3rd Ed. SDK for Symbian OS
General
Start-Up
Technology
Product Type:
Application
Product Start-Up Parameters
Home Directory:
D:\Desarrollo\Symbian\9.1\S60_3rd\Epoc32
Start-Up Command:
release\winscw\udeb\epoc.exe
Start-Up Parameters:
Product Processes
Process Name
Process Parameters
Process Multi-Instance
_PRODUCT_HOME_\release\winscw\u...
false
Add New Process
Remove Process
OK
Cancel


S60 3rd Ed. SDK for Symbian OS
General
Start-Up
Technology
Available Technologies
Technology
Communication Technology
Content Type
Bluetooth
BTUSB
BCSP
IrDA
BTUSB
IrLAP Async
MMS
File
MMS Encapsulation
SMS
File
Series 80 SDK for Symbian OS
Communication Settings for SMS / File / Series 80 SDK for Symbian OS
Attribute
Value
Inbox
C:/inbox/
Outbox
C:/outbox/
OutboxFileExtension
sms
OutboxFileNamePrefix
InboxFileNamePrefix
InboxFileExtension
sms
OK
Cancel

En este caso hemos creado los siguientes terminales para realizar la emulación: S60_3_Ed_FP1 y S60_3_Ed_FP1_2.



Finalmente para probar el envío y recepción de SMS o MMS, solo tenemos que arrastrar los terminales a la ventana de la derecha y su correspondiente servidor de mensajes SMS o MMS.

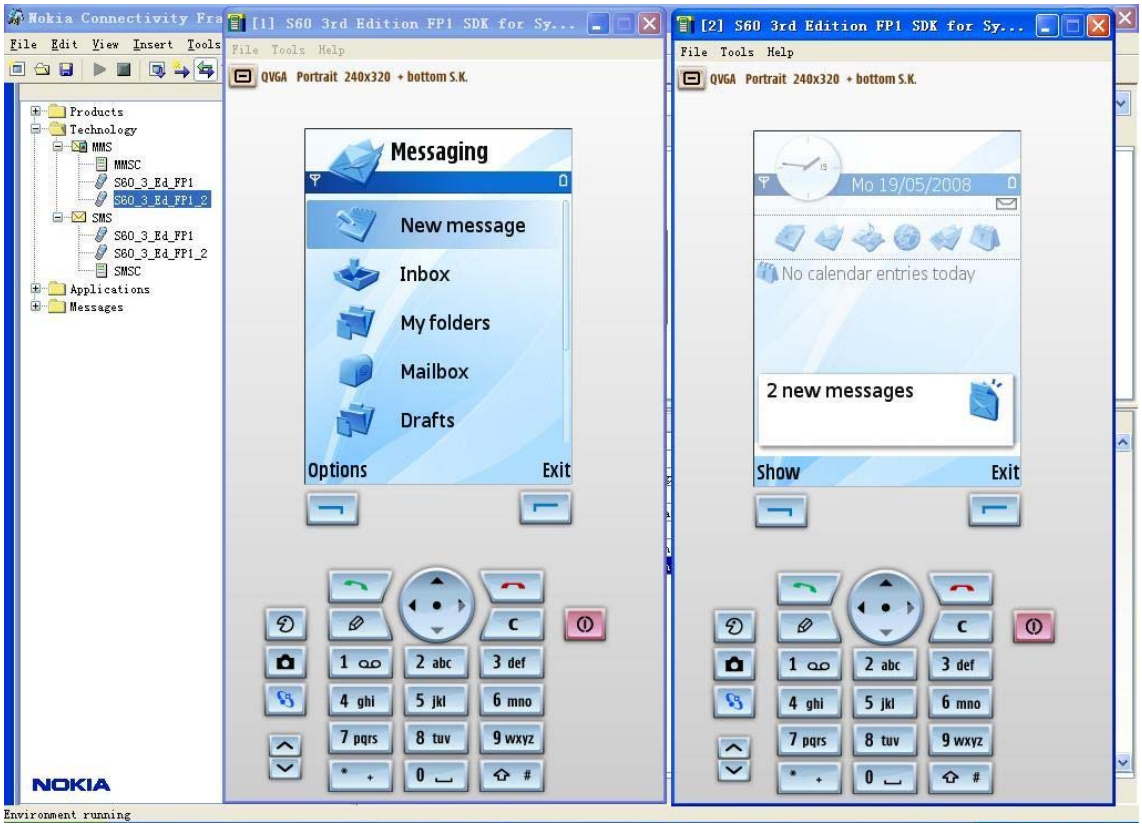


Una vez puesto en marcha el servidor y los terminales, ya se puede emular el envío y reopción de los SMS y los MMS. Es necesario configurar el centro de mensajes para el correcto funcionamiento del envío y recepción. Dicha configuración del emulador se encuentra en Messaging -> Options -> Settings -> Text Messaging -> Message centre.

En la parte inferior derecha hay log que registra todos los eventos del entorno.

ID	Type	Title	Size	Source	Target	Time
8	MMS	MMS Message	464	55512350	55512351	1:43:23
7	MMS	MMS Message	464	55512350	55512351	1:39:37
6	System	Started S60_3_Ed_FP1_2 Integration	0	NCF	S60_3_Ed_FP1_2	1:33:30
5	System	Started MMSC Integration	0	NCF	MMSC	1:32:36
4	System	Started S60_3_Ed_FP1 Integration	0	NCF	S60_3_Ed_FP1	1:32:31
3	System	P2P Connectivity Ready	0	NCF	-	1:31:42
2	System	NCF Integration Library Connectivity Ready	0	NCF	-	1:31:40
1	System	NCF Integration Library Connectivity Ready	0	NCF	-	1:31:40

Ejemplo de envío y recepción de MMS:



4.3 Firmando un SIS

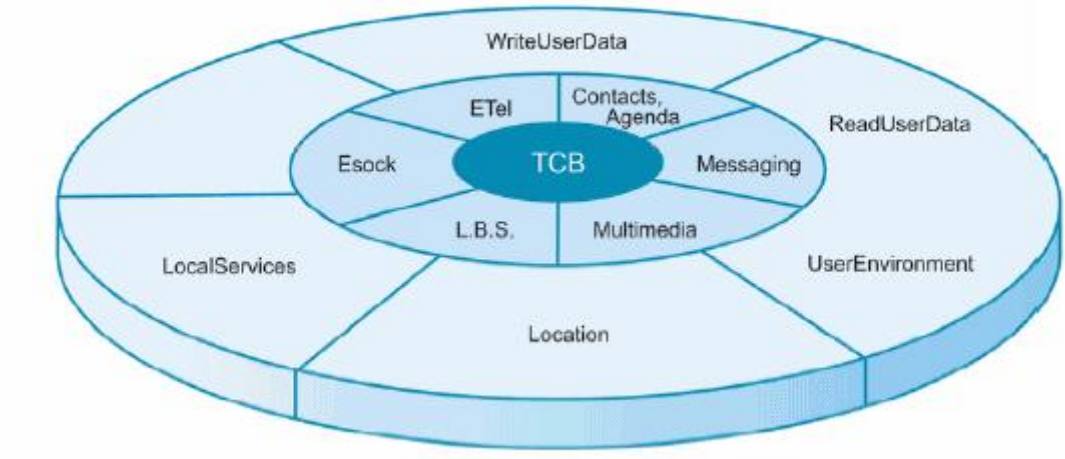
Lo primero que hay que saber es que en symbian se ha organizado el acceso a las API's según el paradigma de las 7 capas: TCB, TCE, Capabilities, SID&VID, DataCaging, SB&SP, Software Installer.

Cada capa tiene una serie de tareas determinadas, TCB se encarga de mantener la integridad del software del sistema, aquí se encuentra el kernel, el servidor de ficheros y el instalador de software; TCE protege las API's sensitivas que manejan recursos del dispositivo. Capabilities controla que puede hacer una aplicación incluyendo las librerías que utilice; SID&VID se utiliza para identificar la aplicación y el vendedor de la aplicación durante la carga en memoria; DataCaging restringe el acceso a los directorios del dispositivo basándose en las capacidades y su identidad, se apoya en TCB; SB&SP es la capa que se encarga de la seguridad de las llamadas entre distintos procesos; y por último Software Installer que verifica la firma de los archivos a instalar así como que el identificador de la aplicación sea único y crea valores Hash en caso de que las aplicaciones sean instaladas en memorias extraíbles del dispositivo para asegurar su integridad.

Lo segundo que se requiere de la firma para poder instalar un .sis en un terminal móvil que use la tercera edición del SDK de symbian o posterior.

Y lo tercero que hay 2 maneras de firmar digitalmente un paquete de instalación o .sis; la auto-firma y la firma a través de una autoridad firmante.

En cuanto al acceso a las API's así como código que requiera ciertas librerías, aproximadamente el 60% no requiere de permisos el resto se han agrupado en "capacidades" dependiendo de donde acceden y su funcionalidad.



Al intentar instalar la aplicación que queramos en un dispositivo móvil, la capa mas baja, Software Installer, se encarga de comprobar:

1. Si el paquete de instalación está firmado o no (no siempre es necesario).

2. Que capacidades vienen registradas en el .sis (siempre podríamos tener menos de las necesarias, entonces se instalaría pero fallaría al intentar ejecutarlo).Las capacidades se especifican en el .mmp insertando una línea CAPABILITY cap1 cap2 ... NONE ... ALL ... -cap1 -cap2, de tal manera que podemos enumerar una detrás de otra ,conceder todas, y quitar las que no interesen o no conceder ninguna.

CAPACIDADES DEL SISTEMA:

- TCB: Máximo privilegio, el código TCB puede hacer cualquier cosa en el móvil.
- ALLFILES: Acceso de lectura de todo el sistema de ficheros y permiso de escritura a los directorios privados de otros procesos.
- COMMDD: Acceso directo a todos los dispositivos de comunicación.
- DISKADMIN: Acceso al sistema de ficheros del administrador.
- DRM: Acceso al contenido DRM.
- MULTIMEDIADD: Acceso a funciones multimedia críticas.
- NETWORKCONTROL: Modificar o acceder a protocolos de control de red.
- POWERMGMT: Apagar, encender, hibernar y matar cualquier proceso.
- PROTSERV: permite a un proceso servidor registrarse con un nombre protegido.
- READDEVICEDATA: Permiso de lectura a un operador de red y ajustes del dispositivo.
- SURROUNDIGNSDD: Acceso al controlador de dispositivos que proveen información del entorno del teléfono.
- SWEVENT: Habilidad de emular pulsaciones de teclas y capturar esos eventos desde cualquier programa.

- TRUSTEDUI: Habilidad de crear sesiones UI confiables y mostrar diálogos en un entorno de interfaz seguro.
- WRITEDEVICEDATA: Acceso de escritura para ajustar el control del comportamiento del dispositivo.

CAPACIDADES DE USUARIO:

- LOCALSERVICES: Acceso a servicios como bluetooth o infrarrojos; servicios que normalmente no ocasionan gasto para el usuario.
- LOCATION: Acceso a datos de posición dados por el teléfono.
- NETWORKSERVICES: Acceso a servicios remotos como wifi; servicios que pueden ocasionar gastos para el usuario.
- READUSERDATA: Capacidad de lectura a datos de usuario confidenciales.
- USERENVIROMENT: Acceso a datos en directo del usuario y su entorno.
- WRITEUSERDATA: Acceso de escritura a datos confidenciales de usuario.

En cuanto a las firmas, la auto-firma la realiza automáticamente Carbide cuando construimos el .sis generando un certificado y un par de claves además del .sis firmado (.sisx, es el que hay que instalar con el PCSuite). Si queremos hacerlo nosotros "a mano" podemos hacerlo a través de una ventana de comandos.

Hay 3 comandos encargados de la creación y auto-firma del .sis, makesis, makekeys y signáis; makesis se encarga de leer el .pkg y construir el paquete de instalación, makekeys crea un certificado y un par de claves pública y privada, signsis firma el .sis con un certificado y un par de claves . Hay otro comando mas, createsis que lo único que hace es llamar a los anteriores, por si no queremos ejecutar cada uno independientemente.

La auto-firma sólo es válida siempre que se utilicen capacidades que puedan ser concedidas por usuario, de esta manera en la instalación aparecerán consultas de autorización a los servicios que se requieran.

Las capacidades que pueden usarse con la auto-firma son: ninguna, LocalServices, ReadUserData, WriteUserData, NetworkServices y UserEnviroment. El resto requieren de una firma por autoridad firmante de

otra manera aparecerá un mensaje en la instalación y el Software Installer no instalará la aplicación.

Para la firma por autoridad firmante hay que generar un certificado de programador y solicitar la firma.

Hay otro método de firmar el .sis con symbian signed website:"https://www.symbiansigned.com/app/page" Te permite firmar un .sis para un IMEI determinado de tal manera que te envían el firmado a una dirección de correo electrónico. Lo único que solo sirve para las capacidades: LocalServices, Location, NetworkServices, PowerMgmt, ProtServ, ReadDeviceData, ReadUserData, SurroundingsDD, SwEvent, TrustedUI, UserEnvironment, WriteDeviceData y WriteUserData.

5 Bibliografía

Libros

Symbian S60 3rd Edition SDK Documentation

Symbian S60 3rd Edition SDK Examples

Symbian_OS_Basics_Workbook_v3_0. Symbian_OS_Getting_Started_v1_0_.

Series_60_Platform_3rd_Edition_Overview

Symbian OS Basic Course Pack 04300

Symbian OS Basic Course Pack 05300

Symbian OS Basic Course Pack 06300

Symbian OS Communications Programming Second Edition (Iain Campbell - Wiley)

Symbian OS Explained – Effective C++ Programming for Smartphones (Jo Stichbury - Wiley)

The Symbian Architecture Sourcebook – Design and Evolution of a Mobile Phone OS (Ben Morris – Wiley)

Rapid Mobile Enterprise Development for Symbian OS – An Introduction to OPL Application Design and Programming (Ewan Spence – Wiley)

S60 Programming – A Tutorial Guide (Paul Coulton and Reuben Edwards with Helen Clemson – Wiley)

Symbian OS C++ coding standards (Symbian DevNet)

Páginas Web

Forum Nokia (http:\\www.forum.nokia.com)

Wikipedia (http:\\es.wikipedia.org)

http://www.todosymbian.com/

http://www.newlc.com/

http://www.elrincondesymbian.com/

http://www.symbianforever.com

http://symbianexample.com/

http://www.symbian.com/Developer/techlib/v9.1docs/doc_source/index.html

6 Glosario

- 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.
- CryptoSMS: Aplicación en Java para el envío de mensajes cifrados.
- DAO: Database Access Object (Objeto de acceso a datos).
- DBMS: DataBase Management System (sistema de gestión de bases de datos).
- Debug: Proceso metódico de encontrar y reducir el número de errores o defectos existentes en un programa, con el objetivo de que el comportamiento de dicho programa sea el esperado.
- JRE: Java Runtime Enviroment (entorno en tiempo de ejecución java).
- J2ME: Java 2 Micro Edition.
- MMS: Multimedia Messaging System (Sistema de mensajería multimedia).
- OEM: Original Equipment Manufacturer (Fabricante de equipos originales)
- SDK: software development kit (kit de desarrollo de software).
- SMS: Short Message Service (Servicio de mensajes cortos).
- Symbian: Sistema Operativo para terminales móviles.

7 Palabras Clave

- Carbide
- Criptografía
- C++
- Mensajería
- MMS
- Móvil
- Nokia
- SMS
- Symbian
- S60

8 Autorización de los Alumnos

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado

Firmado:

Guillermo Colmenarejo Martín

Carlos Enrique Rivera Cordero

Yaofeng Zhang