Consejería de Hacienda y Administración Pública

Manual del integrador del cliente

Sevilla, noviembre de 2010

Hoja de control		
Fecha	**Autor**	**Descripción**
03/11/2010	RPV	Adaptación manual del integrador MPR

1 Introducción

El Cliente de Firma es una herramienta de Firma Electrónica que funciona en forma de Applet de Java integrado en una página Web mediante JavaScript.

El Cliente hace uso de los certificados digitales X.509 y de las claves privadas asociadas a los mismos que estén instalados en el repositorio o almacén de claves y certificados (*keystore*) del navegador web (*Internet Explorer, Mozilla, Firefox*) o el sistema operativo así como de los que estén en dispositivos (tarjetas inteligentes, dispositivos *USB*) configurados en el mismo (el caso de los DNI-e).

El Cliente de Firma, como su nombre indica, es una aplicación que se ejecuta en cliente (en el ordenador del usuario, no en el servidor Web). Esto es así para evitar que la clave privada asociada a un certificado tenga que "salir" del contenedor del usuario (tarjeta, dispositivo USB o navegador) ubicado en su PC. De hecho, nunca llega a salir del navegador, el Cliente le envía los datos a firmar y éste los devuelve firmados.

El Cliente de Firma contiene las interfaces y componentes web necesarios para la realización de los siguientes procesos (además de otros auxiliares como cálculos de hash, lectura de ficheros, etc…):

- Firma de formularios Web.
- Firma de datos y ficheros.
- Multifirma masiva de datos y ficheros.
- Cofirma (CoSignature) → Multifirma al mismo nivel.
- Contrafirma (CounterSignature) → Multifirma en cascada.

Como complemento al cliente de firma, se encuentra un cliente de cifrado que nos permite realizar las funciones de encriptación y desencriptación de datos atendiendo a diferentes algoritmos y configuraciones. Además permite la generación de sobres digitales.

El Cliente viene acompañado de un BootLoader independiente que, además de cargar el cliente, permite instalar en disco local el propio cliente y las librerías que necesita para evitar su reiterada descarga. En cada carga, el instalador comprueba si existe una versión compatible del cliente instalada en el sistema. En caso de no haberla, el cliente se instalará automáticamente en el sistema, previo consentimiento y aceptación de las condiciones de uso por parte del usuario.

El Cliente se distribuye en 3 construcciones distintas (LITE, MEDIA y COMPLETA) de tal forma que un usuario no tiene la necesidad de instalar en su sistema una construcción más pesada que incorpore características que no necesite.

Las funcionalidades de las que dispone cada una de estas construcciones son:

- Construcción LITE: Soporta firmas sin formato, CMS/PKCS#7 y CADES, e incorpora todas las capacidades comunes del cliente (firmas, cifrados, acceso a repositorios…).
- Construcción MEDIA: Soporta firmas XMLdSig, XAdES, ODF y OOXML, más las funcionalidades de la construcción LITE.
- Construcción COMPLETA: Soporta firmas PDF, además de disponer de las funcionalidades de la construcción MEDIA.

2 Objeto y alcance

El presente documento recoge la descripción del cliente @firma y todas sus funcionalidades, así como la información necesaria para permitir a los integradores del cliente incorporarlo como parte de sus aplicaciones Web para la realización de operaciones criptográficas.

Los aspectos detallados del Cliente de Firma v3.1 de @Firma 5 son los siguientes:

- Requisitos del Cliente
  - Sistemas operativos soportados
  - Navegadores soportados
  - Otros requisitos
- Componentes del Cliente
  - Applet bootloader/instalador (Cliente ligero)
  - Applet de firma (Componentes de Administración Electrónica)
- Funcionalidad del Instalador
  - Comprobación si el Cliente está instalado
  - Instalar / Desinstalar el Cliente
  - Ruta de instalación
  - Comprobar si el Cliente está actualizado
  - Actualizar el Cliente

- Funcionalidad básica del Cliente:
  - Firma Web
  - Firma
  - Firma masiva de hashes
  - Multifirma masiva de ficheros
  - Multifirma masiva programática
  - Co-firma
  - Contrafirma
  - Cifrado y descifrado de datos
  - Generación de sobres digitales.
  - Desensobrado de sobres.
- Configuración del cliente:
  - Algoritmos y formatos
  - Selección de certificados
- Otras funcionalidades
- Ejemplos que abarquen los aspectos anteriormente descritos.

3 Requisitos mínimos

- Sistema Operativo
  - Windows 2000 / XP / Vista / 7 / Server 2003 / Server 2008 y superiores
    - Windows 7, Windows Server 2003 y Windows Server 2008 se soportan únicamente con JRE 1.6u18 y superiores. Ver matriz de compatibilidad para más información.
  - Linux 2.6 (Guadalinex, Ubuntu, etc.) y superiores
  - Sun Solaris/Open Solaris 10 y superiores
  - Mac OS X 10.5 y 10.6 (Leopard y Snow Leopard)
- Navegador web:
  - Firefox 2.0.0.20 o superior (incluido Firefox 3.0 y superiores)
  - Internet Explorer 6 o superior

- Compatible con Microsoft Internet Explorer 9 (Windows Internet Explorer Platform Preview, versión 1.9.7766.6000, Internet Explorer versión 9.0.7766.6000.

- Google Chrome 3.0 o superior
- Apple Safari 4.0 o superior

- JRE:
  - JRE 5 (1.5 update 22) o superior instalada en el navegador (sólo en navegadores compatibles con Java 5).
  - JRE 6 32 Bits instalada en el navegador (1.6 update 18 recomendada)
    - JRE 6 x64 únicamente en Microsoft Windows y almacén de certificados CAPI (Microsoft Internet Explorer, Google Chrome y Apple Safari).
  - Para Mac OS X los últimos paquetes de "Java for Mac OS X 10.x update y" configurado para el uso de Java (JRE 1.6.0_20 recomendado).
- Certificado digital de usuario instalado en el navegador / sistema operativo o disponible a través de un módulo PKCS#11 o CSP instalado en el navegador (caso del DNI-e).

Las siguientes matrices de compatibilidad muestran las distintas combinaciones de sistema operativo y entorno de ejecución de Java y su relación con el correcto funcionamiento del acceso a los distintos almacenes de certificados.

		WINDOWS CAPI					
		IExplorer 6 o sup.	Firefox 2 a 3.5	Firefox 3.6	Chrome 3 o sup.	Apple Safari 4	Opera 10
JRE 1.5_22 32 Bits	Windows 2000, XP, Vista (32 Bits)						
	Windows 7, Server 2003, Server 2008						
	Windows XP x64 (64 bits)						
JRE 1.6_18 32 Bits	Windows x86 (32 bits) [todos]						
	Windows x64 (64 bits) [todos]						
JRE 1.6_18 64 Bits	Windows x64 (64 bits) [todos]						
JRE 1.7 b75 [todos]	Windows [todos]						

		NSS					
		IExplorer 6 o sup.	Firefox 2 a 3.5	Firefox 3.6	Chrome 3 o sup.	Apple Safari 4	Opera 10
JRE 1.5_22 32 Bits	Windows (32 y 64 bits) [todos]						
	Linux 2.6 x86 (32 bits)						
	Solaris 10 x86 (32 bits)						
	Mac OS X 10.x (64 bits)						
JRE 1.6_18 32 Bits	Windows (32 y 64 bits) [todos]						
	Linux 2.6 x86 (32 bits)						
	Solaris 10 x86 (32 bits)						
	Solaris 10 x64 (64 bits)						
	Solaris 10 SPARC (64 bits)						
JRE 1.6_18 64 Bits	Windows 64 Bits (todos)						
	Linux 2.6 x64 (64 bits)						
	Solaris 10 x64 (64 bits)						
	Solaris 10 SPARC (64 bits)						
	Mac OS X 10.x (64 bits)						
JRE 1.7 b75 [todos]	[todos]						

		Apple Mac OS X KeyRing (Llavero de Mac OS X)					
		IExplorer 6 o sup.	Firefox 2 a 3.5	Firefox 3.6	Chrome 3 o sup.	Apple Safari 4	Opera 10
Java for MacOS X 10.6 u2	Mac OS X 10.6						
Java for MacOS X 10.5 u7	Mac OS X 10.5						

Código de colores:

- Verde = Compatible, Rojo = No compatible, Gris = No aplica, Blanco = No probado

CAPI es el almacén central de los sistemas operativos Windows (usado por Internet Explorer, Google Chrome, Apple Safari y Opera en este sistema operativo), mientras que NSS es el almacén propio del navegador Mozilla / Firefox, que lo usa en cualquier sistema operativo y se establece como almacén central de los sistemas Linux / UNIX. Mac OS X KeyChain (llavero de Mac OS X) es el almacén central del sistema operativo Mac OS X, y es usado por todos los navegadores en este sistema operativo excepto Mozilla / Firefox (por defecto). Ya que no es posible acceder al almacén de certificados de Mozilla en Mac OS X debido a la incompatibilidad entre arquitecturas, el cliente @firma utilizará el llavero de Mac OS X cuando se ejecute en Mozilla Firefox desde este sistema operativo.

NOTAS:

- Dado que Java 7 (1.7) está aún en fase Beta de desarrollo, no se garantiza la compatibilidad del cliente con esta versión.
- El Cliente @firma no funciona sobre Windows IA64 [Intel Architecture 64 / Intel Itanium] (Windows Server 2003 IA64, Windows Server 2008 IA64).
- No se han hecho pruebas de Internet Explorer 5.x en ninguna de sus plataformas soportadas (Windows, Solaris, Mac OS X, etc.) por considerarse obsoleto.
- No se han hecho pruebas en Linux / UNIX con entornos de Java de 64 bits.
- En los entornos Linux / UNIX se presupone siempre que el NSS instalado es de la misma arquitectura (32 ó 64 bits) que el Mozilla / Firefox.
- Java 1.4 no está soportado en ninguna plataforma ni entorno.
- Java 5 (1.5) no está oficialmente soportado en Windows 7 (cualquier variante). Es recomendable en cualquier caso actualizar a Java 6 (1.6.0_18 o superior).
- Firefox 3.6 (y versiones superiores) necesitan al menos Java 1.6_10 (recomendado 1.6_18 o superior). Más detalles en: http://www.java.com/en/download/faq/firefox_newplugin.xml

4 Componentes del cliente

4.1 Cliente

El cliente se compone de:

- **Clases** de la aplicación, agrupadas en ficheros *.jar* y *.jar.pack.gz* almacenados en un directorio de usuario o en un directorio en el servidor.
- **Librerías** (bibliotecas) de sistema (*.dll* en *Windows*, *.so* y *.bin* en *Linux, .dylib* en Mac OS X) almacenadas durante el proceso de instalación en directorios del sistema. En el servidor y en la distribución para su instalación en local se almacenan en ficheros comprimidos ZIP.
- **Bibliotecas JavaScript**: Contienen funciones para la automatización de los procesos de firma. Almacenadas en el servidor Web. Son opcionales y se puede operar sin ellas, pero facilitan los procesos más comunes.
    - **El conjunto principal de bibliotecas JavaScript no están diseñadas para ser modificadas directamente por el integrador excepto en caso de necesidades muy específicas.** No obstante, existe una biblioteca JavaScript llamada *constantes.js* que sí contiene parámetros modificables que permiten una mayor personalización del comportamiento del cliente.

4.2 Instalador

El instalador, también llamado B*ootLoader* es el encargado de copiar las bibliotecas nativas y las clases Java en almacenamiento local, de modo que el usuario no necesite descargarlos cada vez que hace uso del cliente.

El instalador tiene sus propias clases Java y sus bibliotecas JavaScript.

NOTA IMPORTANTE:

Si el integrador quisiese prescindir de la copia local de ficheros Java al disco del usuario, para así adecuarse mejor a las buenas prácticas de Java, deberá tener en cuenta los siguientes aspectos:

- El cliente, para el uso de firmas XML (XMLDsig, XAdES y ODF) en Java 5, necesita la instalación de Apache Xalan y Apache Xerces como API *ENDORSED* de Java (consulte la documentación técnica de Java para más información sobre los API *ENDORSED*).

- El cliente busca siempre las bibliotecas nativas en las localizaciones específicas dentro del directorio de usuario de @firma, en almacenamiento local.
- Para el correcto soporte de los almacenes de claves de Windows e Internet Explorer en Java 5, se incorporan como API EXTENDIDOS el proveedor de seguridad SunMSCAPI de Sun Microsystems.

5 Instalación del cliente

Para evitar repetidas cargas del cliente y mejorar su integración con el sistema, el Cliente @firma comprueba automáticamente si está instalado en el sistema y pide permiso al usuario para instalarse en caso de no estarlo. Hasta la versión 3.1 del Cliente @firma, era necesario tenerlo instalado para poder ejecutarlo cuando se desplegase mediante las bibliotecas JavaScript que se distribuían con el mismo.

En la versión v3.1 y superiores del Cliente se ha integrado un modo de despliegue vía JNLP que permite evitar su instalación. Sin embargo, este modo de despliegue sólo podrá darse cuando se ejecute sobre Java 6 update 10 o superior (32 bits). En las versiones anteriores de Java y en las JVM con arquitectura de 64bits, el proceso de instalación sigue siendo necesario.

En caso de que el cliente se ejecute en un entorno que requiera instalación y detecte que no está instalado, o que la instalación es incorrecta, inicia de forma automática el proceso de instalación. Este proceso requerirá que el usuario acepte las condiciones de uso del cliente, tras lo cual no será necesaria su intervención. La duración del proceso variará según la velocidad de la conexión del cliente.

En caso de no concretarse una construcción en instalación del cliente, siempre se instalará la construcción LITE, que es la instalación mínima funcional. Si alguna aplicación concreta va a utilizar capacidades de las construcciones MEDIA o COMPLETA, se deberá indicar la construcción concreta en la carga o instalación del cliente para que se compruebe que se dispone de esa construcción o una superior y, en caso negativo, se instalase.

Durante el proceso de instalación, el cliente v3 comprueba si existen instaladas versiones anteriores (2.4 o previas). Para mayor detalle sobre esta funcionalidad consulte el apartado **¡Error! No se encuentra el origen de la referencia.**.

El Cliente v3 puede operar normalmente con versiones anteriores instaladas, y, de igual manera, las versiones anteriores no tienen ningún problema si se encuentra instalada la versión 3 del Cliente.

5.1 Despliegue del cliente

En un entorno Web, en donde el cliente es descargado para su ejecución, deberá disponerse de todos los ficheros de instalación para **las tres construcciones disponibles** y las distintas configuraciones de sistema. Esto es así porque no es conocido a priori el entorno desde el que el usuario ejecutará el cliente y hasta este momento no se comprueban los requisitos necesarios para el mismo. En todo caso, las funciones JavaScript de despliegue del cliente optimizan el proceso de instalación y carga del cliente haciendo que este sólo descargue los ficheros que le son necesarios para su sistema concreto.

A partir de la versión 3.1 del Cliente @firma se integra en los JavaScript de soporte el mecanismo de despliegue mediante JNLP (Java Networking Launching Protocol). Este mecanismo permite hacer uso del Cliente @firma sin necesidad de que el usuario lo tenga instalado. El despliegue JNLP evita que los usuarios deban instalar y actualizar del cliente de forma explícita y permite utilizar el cliente en entornos con permisos restringidos.

El modo de despliegue vía JNLP se activará automáticamente cuando se detecte que el usuario dispone de Java 6 update 10 o superior (32 bits). En caso de disponer de Java 5, versión anterior de Java 6 o un Java 6 de 64bits, el Cliente se desplegará del modo tradicional a través del Bootloader.

El posible obligar a que se ejecute una versión instalada del Cliente @firma, sea cual sea la versión de Java sobre la que se ejecute, y que se instale en caso de ser necesario. Para más detalle consulte el apartado "Carga del Cliente" de este manual.

El *BootLoader* debe situarse en el mismo directorio que la página HTML que lo carga. Si se desea utilizar otro directorio o la página HTML se genera al vuelo, puede indicarse el directorio mediante la constante JavaScript "base" del fichero *constantes.js*. De igual forma, las distintas construcciones del núcleo del cliente y el resto de ficheros instalables deben copiarse en el mismo directorio de la página HTML o en el indicado mediante la constante JavaScript "baseDownloadURL" del mismo fichero. Las rutas de establecidas mediante las constantes "base" y "baseDownloadURL" podrán ser absolutas o relativas. Siempre usará la barra separadora "/" (nunca "\") y no podrán empezar o terminar por este carácter. Rutas de ejemplo:

- **Absolutas:** "file:///C:/ficheros","http://www.mpr.es/ficheros", "https://ficheros"…
- **Relativas:** "afirma/ficheros".

5.2 Despliegue JNLP

En entornos Windows Vista y Windows 7 en los que los usuarios tienen permisos muy restringidos, no es posible instalar el Cliente @firma, ya que el proceso de instalación necesita descargar y copiar clases Java y bibliotecas nativas, operaciones prohibidas en muchos casos.

Para tratar estas situaciones, es especialmente útil el nuevo medio de carga del cliente vía JNLP.

El Cliente se cargará vía JNLP cuando se ejecute sobre Java 6 update 10 o superior (arquitectura de 32 bits).

Este método de carga introduce las siguientes ventajas:

- No necesita el proceso previo de instalación del cliente
  - No necesita descargar ni copiar ningún fichero de instalación, ni clases Java ni bibliotecas nativas.
- Usa la tecnología JNLP (*Java Network Launch Protocol*) para ejecutar el Applet integrado en la página Web
  - Se mantiene un caché de aplicaciones JNLP que hace que. si la aplicación ejecutó correctamente (vía JNLP) en ocasiones anteriores, no es necesario volver a descargar ningún fichero.
- No interfiere con posibles instalaciones previas del Cliente @firma. El Applet cargado vía JNLP es completamente independiente.
- No necesita ningún permiso de usuario, excepto el de acceso a los certificados de firma.
- No necesita un control de versionado propio, siempre se ejecuta la versión que el integrador publique. Evita las actualizaciones innecesarias, si el integrador publica una nueva versión, el caché de aplicaciones JNLP se encarga automáticamente y de forma completamente transparente de la actualización.

El Cliente se desplegará automáticamente vía JNLP cuando se ejecute sobre una JRE 6u10 o superior.

En caso de que surjan problemas o el integrador considere preferible que el Cliente se descargue e instale siempre, es posible deshabilitar la carga mediante este sistema. Para más información diríjase al apartado 6.1 del manual.

El despliegue vía JNLP requiere que se configuren de los ficheros "*COMPLETA_afirma.jnlp*", "*MEDIA_afirma.jnlp*" y "*LITE_afirma.jnlp*" con la ruta al directorio en donde se encuentran los ficheros con el núcleo del Cliente @firma. La estructura de estos ficheros es la que sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="RUTA_DIRECTORIO_DESPLIEGUE">
 <information>
   <title>Cliente @firma</title>
   <vendor>Junta de Andalucía</vendor>
   <homepage href="http://www.juntadeandalucia.es" />
   <description>Cliente de firma @firma</description>
 </information>
 <offline-allowed/>
 <security>
   <all-permissions/>
 </security>
 <resources>
   <property name="jnlp.packEnabled" value="true"/>
   <j2se version="1.6+" href="http://java.sun.com/products/autodl/j2se" />
   <jar href="COMPLETA_j6_afirma5_coreV3.jar" main="true"/>
 </resources>
 <applet-desc
   name="Cliente @firma"
   main-class="es.gob.afirma.cliente.SignApplet"
   width="1"
   height="1">
 <update check="background"/>
</jnlp>
```

El integrador deberá modificar la cadena "**RUTA_DIRECTORIO_DESPLIEGUE**", de cada uno de estos ficheros, por la ruta absoluta al directorio en donde se encuentren los ficheros "*COMPLETA_j6_afirma5_coreV3.jar*", "*COMPLETA_j6_afirma5_coreV3.jar.pack.gz*", "*MEDIA_j6_afirma5_coreV3.jar*", "*MEDIA_j6_afirma5_coreV3.jar.pack.gz*" y "*LITE_j6_afirma5_coreV3.jar*", "*LITE_j6_afirma5_coreV3.jar.pack.gz*"etc.

Por ejemplo:

http://www.juntadeandalucia.es/afirma/

5.3 Ficheros para el despliegue del cliente

El listado completo de archivos que cubren todas las construcciones y configuraciones de entorno soportadas por el cliente y ayudan a su optimización en la carga son:

Fichero	Descripción
afirmaBootLoader.jar	Lanzadera del cliente.
LITE_ j5_afirma5_coreV3.jar[.pack.gz]	Construcción LITE del núcleo del cliente de firma compatible con Java 5.
MEDIA_ j5_afirma5_coreV3.jar[.pack.gz]	Construcción MEDIA del núcleo del cliente de firma compatible con Java 5.
COMPLETA_ j5_afirma5_coreV3.jar[.pack.gz]	Construcción COMPLETA del núcleo del cliente de firma compatible con Java 5.
LITE_j6_afirma5_coreV3.jar[.pack.gz]	Construcción LITE del núcleo del cliente de firma optimizada para Java 6.
MEDIA_ j6_afirma5_coreV3.jar[.pack.gz]	Construcción MEDIA del núcleo del cliente de firma optimizada para Java 6.
COMPLETA_ j6_afirma5_coreV3.jar[.pack.gz]	Construcción COMPLETA del núcleo del cliente de firma optimizada para Java 6.
LITE_afirma.jnlp	Configuración para el despliegue vía JNLP de la construcción LITE del Cliente.
MEDIA_afirma.jnlp	Configuración para el despliegue vía JNLP de la construcción MEDIA del Cliente.
COMPLETA_afirma.jnlp	Configuración para el despliegue vía JNLP de la construcción COMPLETA del Cliente.
afirma_5_java_5.jar.pack.gz	Colección de librerías *endorsed* para el manejo de firmas XML desde JAVA 5.
xalan.zip	Bibliotecas XALAN necesarias para las firmas XML usando Java 5.
mscapi.zip / mscapix64.zip	Bibliotecas nativas necesarias para acceder al repositorio de certificados de Windows en entornos

	Java 5 y Java 6 (x64), respectivamente.
sunmscapi.jar	Biblioteca Java necesaria para acceder al repositorio de Windows en Java 5.
msvcr71.zip	Biblioteca nativa de la que depende MSCAPI.
version.properties	Almacena la versión publicada del cliente @firma, que deberá ser la misma para todas las distribuciones.

5.4 Instalación del cliente

La instalación del cliente de firma, al igual que su carga, se realiza mediante el componente *BootLoader*. Este módulo de instalación ofrece al integrador la posibilidad de comprobar si el usuario dispone de la construcción requería del cliente para el uso de su aplicación e instalarla si no la tuviese.

La instalación del cliente se realiza por defecto en el directorio afirma.5, dentro del directorio de usuario actual. Puede modificarse este directorio por otro, también dentro del directorio del usuario, mediante la constante JavaScript "installDirectory" del fichero *constantes.js*.

Para usar las funcionalidades del BootLoader es necesario crear una página HTML en la que se importan una serie de bibliotecas JavaScript, situadas todas ellas en el directorio de @firma en el servidor:

- common-js/deployJava.js
- common-js/instalador.js
- common-js/time.js
- constantes.js

La forma habitual de usar estas bibliotecas es la mostrada en la página de ejemplo "*demoInstalador.html*", consistente en importar las bibliotecas en la cabecera de la página y declarar funciones que ejecuten la lógica deseada y que puedan ser llamadas frente a las actuaciones del usuario:

```
<head>
	<script type="text/javascript" language="javascript" src="common-
js/deployJava.js"></script>
	<script type="text/javascript" language="javascript" src="common-
js/instalador.js"></script>
	<script type="text/javascript" language="javascript" src="common-js/time.js"></script>
	<script type="text/javascript" language="javascript" src="constantes.js"></script>
	<script language="javascript">

		function instalarCliente(build) {
			if(!instalar(build))
				alert('Ocurrio un error durante el proceso de instalacion');
		}

	</script>
</head>
```

Estas bibliotecas son también necesarias para llevar a cabo los procesos de instalación y desinstalación del cliente. Para más información acuda a los apartados Actualización del Cliente y Desinstalar el Cliente.

Todas las bibliotecas JavaScript están documentadas, por lo que para una información más avanzada puede remitirse a ellas.

Los métodos JavaScript que se ofrecen para llevar a cabo la instalación son:

- ***instalar()***: Instala la construcción LITE del cliente.
- ***instalar(build)***: Instala la construcción indicada del cliente.

Los valores aceptados del parámetro build son:

- **LITE**: Instala la construcción LITE del cliente.
- **MEDIA**: Instala la construcción MEDIA del cliente.
- **COMPLETA**: Instala la construcción COMPLETA del cliente.

Puede encontrar ejemplos sencillos de uso de esta función así como del resto de llamadas relacionadas con la instalación del Cliente en la página Web de ejemplo "*demoInstalador.html*".

Una vez se solicita la instalación del cliente se notifica al usuario mediante un mensaje como el siguiente:

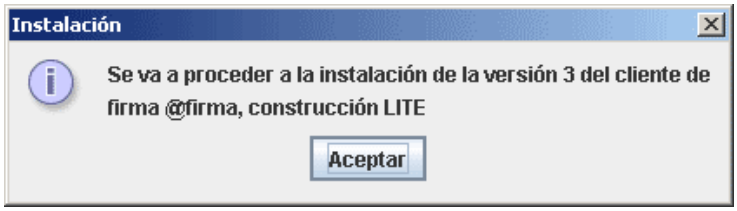


Figura 1: Notificación de instalación

Tras aceptar el mensaje se mostrará el acuerdo de licencia que el usuario deberá aceptar. En caso de no hacerlo, la instalación del cliente quedará suspendida y cualquier otra operación sobre él llevará a error.

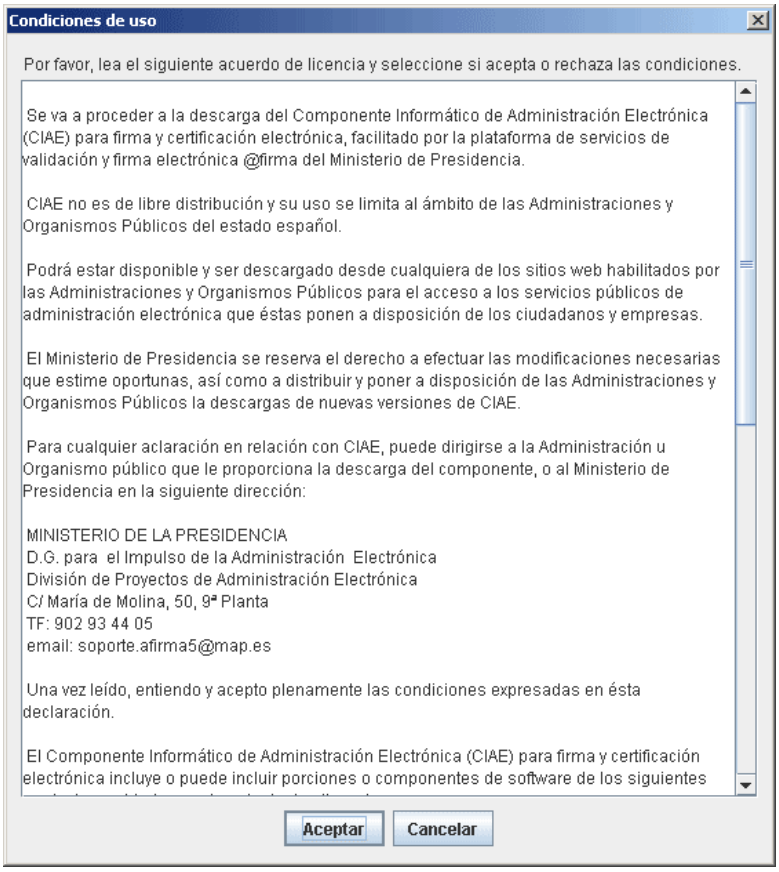


Figura 2: Acuerdo de licencia

Cuando se acepta el acuerdo de uso y una vez finalizada correctamente la instalación, en caso de encontrar versiones antiguas del cliente (versiones 2.4 y anteriores instaladas en el directorio "*.clienteFirmaArrobaFirma5*"), el *BootLoader* informará al usuario y dará la posibilidad de eliminarlas. Si se hace uso de las clases JavaScript de despliegue puede modificarse este comportamiento mediante la variable "`oldVersionsAction`" del fichero "*constantes.js*". Los valores posibles son:

- ***1***: Preguntar al usuario si desea eliminar las versiones anteriores del cliente (por defecto).
- ***2***: No eliminar versiones anteriores del cliente.
- ***3***: Eliminar versiones anteriores del cliente sin preguntar al usuario.

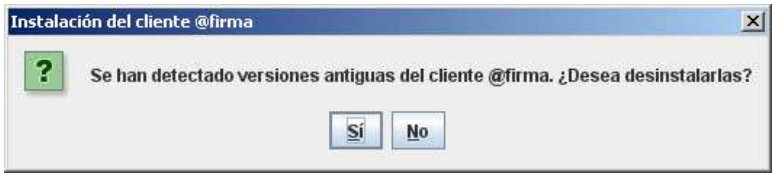


Figura 3: Borrar versiones antiguas

Puede encontrar información adicional de esta funcionalidad en la propia documentación de la biblioteca JavaScript.

La correcta instalación del cliente se notificará con mensajes del siguiente tipo:

Figura 4: Confirmación de la instalación del cliente

Puede verificarse si el cliente está correctamente instalado en el sistema del usuario mediante los métodos JavaScript:

- ***`isInstalado()`***: Comprueba que esté instalada la construcción LITE del cliente o una superior.
- ***`isInstalado(build)`***: Comprueba que esté instalada la construcción indicada del cliente o una superior.

5.5 Actualización del cliente

Se ha incluido en el API de carga del cliente un método para la actualización del mismo. El *BootLoader* es capaz de comprobar que versión del Cliente hay instalada en sistema del usuario y cuál es la disponible para su instalación en el servidor (identificada en el fichero "*versión.properties*"). Siempre que se encuentre una versión nueva del cliente en el sistema servidor se llevará a cabo la actualización del mismo.

Las propiedades del fichero "*version.properties*" son:

- **version.mayor:** Identificador de primer nivel de la versión del cliente publicado.
- **version.minor:** Identificador de segundo nivel de la versión del cliente publicado.
- **version.build:** Identificador de tercer nivel de la versión del cliente publicado.

En el fichero de ejemplo "*demoInstalador.html*" puede verse un ejemplo de uso de esta funcionalidad.

El método JavaScript para llevar a cabo la actualización es:

- ***actualizar()***: Actualiza a la última versión disponible en el servidor de la construcción del cliente que se tenga instalada en el sistema.

La actualización del cliente se informa con un mensaje del tipo:

Figura 5: Notificación de la actualización del cliente

Una vez finalizada la instalación, se informará del resultado de la operación con una ventana tal como la siguiente:

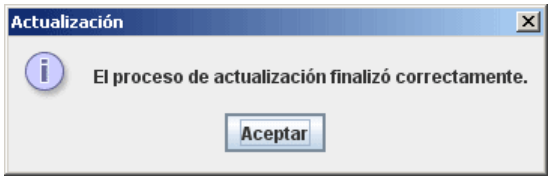


Figura 6: Resultado de la actualización

Tras la carga del cliente siempre se realiza una comprobación de versión. Si el usuario tuviese instalada una versión del cliente anterior a la publicada, se le ofrecería a éste la posibilidad de actualizar.

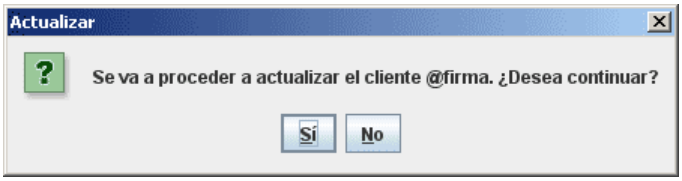


Figura 7: Confirmar actualización

Para comprobar en cualquier momento si el cliente está actualizado pueden usarse el método JavaScript **`isActualizado()`**.

También puede consultarse la versión que se tiene instalada del cliente, para lo que se utiliza el método JavaScript **`getVersionCliente()`**.

Para comprobar la versión del *BootLoader* puede hacerse uso del método JavaScript **`getVersion()`**. El *BootLoader* no se instala en el sistema cliente, así que este método es puramente informativo.

5.6 Desinstalación del cliente

El Cliente ofrece la opción de desinstalación a través del mismo *BootLoader* encargado de su instalación. La desinstalación del cliente consiste en la eliminación del directorio de instalación del cliente junto con todas las librerías y ficheros que allí se copiasen durante el proceso de instalación. Este directorio, por defecto "*afirma.5",* es el indicado mediante la variable "`installDirectory`" del fichero "*constantes.js*" y se encuentra en la carpeta del usuario activo en el sistema.

El proceso de desinstalación evita la eliminación de librerías que puedan ser utilizadas por otras aplicaciones, al igual que mantiene las API ENDORSED de Java. Este proceso, en ningún caso, eliminará, si existiese, el almacén de claves de cifrado del usuario.

Es posible desinstalar el cliente a través de los siguientes métodos:

- **`desinstalar()`**: Desinstala el cliente al completo.
- **`desinstalarAntiguas()`**: Desinstala las versiones antiguas del cliente (versiones 2.4 y anteriores) eliminando su directorio base.

Puede consultar el fichero de ejemplo "*demoInstalador.html*" para ver un ejemplo de las funcionalidades de desinstalación del cliente @firma.

Adicionalmente, se puede hacer uso del método del *BootLoader* **`isInstalado()`** para comprobar si el cliente aun se encuentra instalado o si su desinstalación se llevó a cabo.

Si previamente a la desinstalación del cliente se ha hecho uso del mismo, es muy posible que el navegador haya bloqueado el fichero JAR del núcleo y/o el propio directorio de instalación, con lo cual el proceso de desinstalación devolverá un error aun cuando se hayan desinstalado el resto de dependencias. Para conseguir una desinstalación completa y exitosa del cliente, es recomendable cerrar todas las instancias del navegador y volverlo a abrir, desinstalando el cliente antes de que se cargue.

La desinstalación del cliente de firma se confirma al usuario mediante el siguiente mensaje:

Figura 8: Desinstalación del cliente de firma

5.7 Obtener el directorio de instalación del Cliente

Para obtener el directorio de instalación del Cliente, el instalador tiene un método llamado "getInstallationDirectory()" que devuelve una cadena con la ruta al directorio de instalación.

Este método no puede invocarse antes de que el Instalador termine de cargarse. En el fichero JavaScript "*instalador.js*" se incluye una función llamada **getDirectorioInstalacion()** que espera a que esto suceda antes de invocarlo.

Por ejemplo:

```
<html>
<head>
  <script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
  <script type="text/javascript" language="javascript" src="common-js/appletHelper.js"></script>
  <script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
  <script type="text/javascript" language="javascript" src="common-js/time.js"></script>
  <script type="text/javascript" language="javascript" src="constantes.js"></script>
  […]
</head>
<body onload="cargarAppletInstalador()">
  […]
  <a href="#" onclick="alert('Directorio de instalación: '+getDirectorioInstalacion(); return
  false;">Directorio de instalación del cliente</a>
  <br/>
  […]
</body>
```

```
</html>
```

6 Uso del Cliente de Firma como Applet de Java

6.1 Carga del Cliente

Para cargar el Cliente de Firma en una página WEB debemos invocar la función JavaScript "**cargarAppletFirma()**" incluida en el fichero "***instalador.js***". Esta función:

1. Comprueba si nuestra versión de JRE es compatible con el despliegue JNLP.
    a. Si no es compatible, lanza el *BootLoader* del cliente y comprueba si el Cliente está instalado.
        i. Si no lo está, lo instala según se ha descrito.
2. Carga el Applet de firma (Cliente).

El cliente de firma queda cargado en memoria y puede accederse a las funcionalidades que implementa por medio de la variable JavaScript "clienteFirma", localizada en el fichero "***constantes.js***".

Este método puede ser utilizado también con uno o dos parámetros. El primero de ellos indicaría la construcción mínima que es necesaria para el correcto funcionamiento de la aplicación. Los valores soportados son:

- **LITE**: Comprueba que esté instalada la construcción LITE del cliente o una superior, en caso afirmativo la carga y en caso negativo instala la construcción LITE. Este es el comportamiento estándar cuando no se indica una construcción por parámetro.
- **MEDIA**: Comprueba que esté instalada la construcción MEDIA del cliente o una superior, en caso afirmativo la carga y en caso negativo instala la construcción MEDIA.
- **COMPLETA**: Comprueba que esté instalada la construcción COMPLETA del cliente, en caso afirmativo la carga y en caso negativo instala la construcción COMPLETA.

El segundo parámetro, sólo disponible para las versiones 3.1 y superiores del Cliente, permite obligar a que se ejecute una versión instalada del Cliente @firma y que se instale previamente si no lo estuviese ya. En caso de indicar true como segundo parámetro el cliente se instalará siempre. Si no se indica nada, o se indica cualquier otro valor, el cliente se desplegará vía JNLP si se ejecuta sobre la JRE 6u10 o superior, o se instalará normalmente en caso de versiones anteriores y soportadas.

En caso de encontrarse instalado el cliente, en el sistema del usuario sólo podrá haber una construcción instalada en cada momento (para un directorio de instalación concreto) así que siempre se cargará esta construcción. Es decir, si tenemos instalada una construcción superior a la solicitada se cargará esa construcción superior. Por ejemplo, si tenemos instalada la construcción COMPLETA y se nos pide cargar el cliente con la sentencia JavaScript **cargarAppletFirma('LITE'),** se comprobará que tenemos instalada una construcción igual o superior a la solicitada y se usará esa construcción de tal forma que tendríamos acceso a las funcionalidades de la construcción COMPLETA.

El indicar una construcción en el método de carga sólo sirve cuando no tenemos instalado el cliente (que siempre es el caso de JNLP) o tenemos una construcción inferior en cuyo caso se instalaría la indicada como parámetro del método. Si no se indica la construcción, se tomará la construcción por defecto del fichero "***constantes.js***".

La carga del cliente en una página puede realizarse con sólo introducir un componente JavaScript en el propio cuerpo de la página que se encargue de invocar al método de carga.

Por ejemplo, si sólo vamos a usar las funcionalidades de firma CAdES usaríamos:

```
<html>
  <head>
    <script type="text/javascript" language="javascript" src="constantes.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
    […]
   </head>
   <body>
     <script type="text/javascript">
        cargarAppletFirma();  // Esto carga la construcción por defecto
     </script>
    […]
   </body>
</html>
```

En cambio, si, por ejemplo, quisiésemos realizar firmas PDF tendríamos que realizar:

```
<html>
  <head>
    <script type="text/javascript" language="javascript" src="constantes.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
    […]
  </head>
   <body>
```

```
    <script type="text/javascript">
      cargarAppletFirma('COMPLETA');
    </script>
    […]
  </body>
</html>
```

Si a lo largo de nuestra aplicación se tuviese que utilizar en varias ocasiones el cliente y en cada una de ellas se utilizasen distintas funciones, deberemos utilizar siempre como parámetro del método de carga el identificador de la construcción más completa que se requiera. Esto evitaría casos en los que el usuario se tuviese que instalar una construcción del cliente para seguidamente requerir y necesitar una construcción superior.

NOTA IMPORTANTE:

El Cliente @firma v3 utiliza la biblioteca JavaScript "deployJava.js" de Sun Microsystems para realizar la carga de los Applets, y esta biblioteca exige que la carga de los Applets (la escritura dinámica de las etiquetas que declaran el Applet) se realice antes de que finalice completamente la carga de la página.

El método onLoad() del cuerpo de las páginas HTML se invoca automáticamente justo después de finalizar completamente la carga de estas, por lo que no es un lugar válido para realizar la llamada. Cualquier otro punto no relacionado con eventos de carga es válido para situar la llamada.

En los ejemplos HTML incluidos con el Cliente puede verse una situación correcta, justo tras la etiqueta HTML de inicio del cuerpo de la página:

```
<html>
        <head>…</head>
        <body>
                <script type="text/javascript">  cargarAppletFirma();  </script>
                …
        </body>
</html>
```

6.2 Tratamiento de errores

Es posible tratar todos los errores que se hayan producido durante la operación del cliente mediante JavaScript. El cliente siempre almacena si la última operación criptográfica que realizó finalizó correctamente o no. Es posible consultar este resultado mediante el método del cliente **`isError()`**. En caso de producirse un error además, se podrá obtener la descripción del mismo mediante el método

`getErrorMessage()`. De esta forma pueden elaborarse mecanismos JavaScript capaces de detectar y mostrar los errores pertinentes al usuario.

Un ejemplo que ilustra este sistema de tratamiento de errores es:

```
var fichero= document.getElementById("fichero");
  clienteFirma.initialize();
  clienteFirma.setFileuri(fichero.value);
  firmar();

  if(!clienteFirma.isError()) {
   var firmaB64 = document.getElementById("firmaB64");
   firmaB64.value = clienteFirma.getSignatureBase64Encoded();
   return true; // Enviar
  }
  else {
   alert("No se ha podido firmar: "+clienteFirma.getErrorMessage());
   return false;
  }
}
```

También es posible dejar la tarea de notificación de los errores directamente al cliente. En caso de hacerlo, el cliente mostrará un mensaje de error mediante un dialogo Java por cada error de operación detectado (salvo en multifirmas masivas en donde estas notificaciones harían inviable un uso eficiente del cliente y en donde, por el contrario, se generan trazas de log).

Para activar este mecanismo de notificación de errores es necesario configurar a **true** la constante **showErrors** del fichero JavaScript "*constantes.js*" y establecerla antes de cada operación mediante la función **initialize()** de "*firma.js*" o "*cripto.js*", según se vayan a realizar operaciones de firma o cifrado/ensobrado, respectivamente. Por defecto, esta opción está configurada a **false**.

6.3 Firma web

6.3.1 ¿Qué es la firma Web?

En el proceso de firma Web una parte de una página Web (como un formulario o la página entera) puede ser firmada digitalmente. Para ello,

1. Se compone un HTML por medio de JavaScript
2. Se muestran al usuario los datos a firmar
3. Se le solicita permiso para firmarlo

4. Se selecciona un certificado con el que firmar
5. Se solicita (si es necesario) la contraseña para acceder tanto al repositorio de certificados como a la clave privada del certificado
6. Se firma el HTML generado

6.3.2 ¿Qué puede firmar el componente firma Web?

Se puede firmar digitalmente cualquier elemento de un documento HTML (o el documento mismo).

En los campos modificables por el usuario, se firman los valores seleccionados por el mismo. Esto incluye también adjuntos, que son firmados por el Cliente. A la hora de firmarse, se muestra al usuario la página Web resultante que le permite verificar lo que realmente va a firmar.

La firma Web del cliente sigue la política WYSIWYS (What You See Is What You Sign), es decir, lo que el usuario ve es lo que firma.

Para llevar a cabo la firma Web es imprescindible que la página generada para la firma esté bien formada.

A continuación se muestra un ejemplo de formulario Web y de su previsualización para la confirmación de firma:

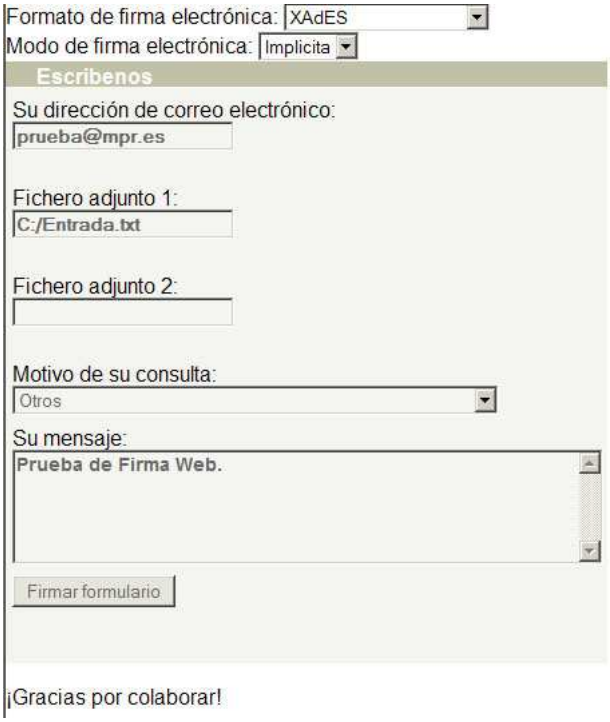


Figura 9: Formulario Web para firmar

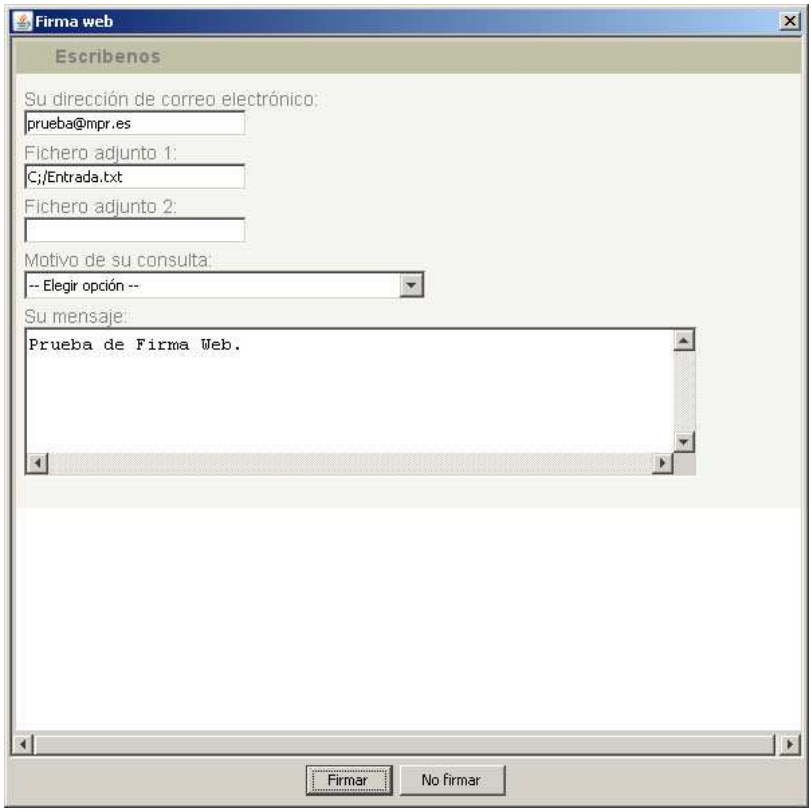


Figura 10: Solicitud de Firma Web

6.3.3 ¿Qué no firma el Cliente en la firma Web?

El cliente no firma las imágenes. Esto hay que tenerlo en cuenta a la hora de diseñar la parte del documento que se ha de firmar, pues es la que se mostrará al usuario (sin imágenes).

El cliente recoge todos los estilos *CSS* del documento que se definan mediante LINK o STYLE. Sin embargo, aquellas hojas de estilo que no se enlacen directamente (sino que se importen mediante la directiva @import) no se incluirán en el HTML si el usuario utiliza Mozilla Firefox. Por lo tanto, los estilos necesarios para mostrar correctamente la parte a firmar deben ir incluidos mediante STYLE o referenciados directamente mediante LINK, nunca mediante @import.

6.3.4 ¿Cómo hacer una firma Web?

Para hacer una firma Web, se puede pasar un HTML al Cliente para que muestre al usuario y éste decida si firmarlo mediante el método **webSign** del Cliente. Este método recibe una cadena HTML como parámetro.

Este método:

1. Muestra el documento al usuario. Se visualizará la página HTML indicada tal y como se ha especificado. Esto quiere decir que en el visualizador los campos aparecerán habilitados si es así como estaban definidos en el documento original. Así pues, cualquier modificación de los datos contenidos en esta ventana se verá reflejada en la posterior firma.
2. Solicita permiso para firmar el documento mostrado.
3. Solicita (si procede) la contraseña del repositorio de certificados.
4. Si no se ha establecido previamente un certificado para firma, muestra al usuario los certificados disponibles para firmarlo, y le solicita que elija uno.
5. En caso de que esté protegido por contraseña se la solicita al usuario
6. Firma el documento.

Una vez invocado:

1. Si el método **isError** del cliente devuelve **false**
    i. El valor devuelto (por la función JavaScript **firmaWeb** o el método del Cliente **webSign**) es la ruta de un fichero (local) que contiene la firma del HTML [en el formato por defecto: CMS (explícito, es decir, que no contiene los datos firmados) y con los algoritmos por defecto: RSA y SHA1 codificada en base 64. En el apartado relativo a la configuración del Cliente se verá cómo cambiar estos parámetros. El contenido del fichero puede ser leído como texto (p. ej. firma XAdES) con el método getTextFileContent del cliente (el valor devuelto por este método puede variar dependiendo de la codificación original del texto) o, si es binario (p. ej. firma CAdES), codificado en base 64 con el método getFileBase64Encoded del cliente. Ambas funciones están descritas en el apartado "Otras funcionalidades" de este documento. La comunicación con el servidor de firma queda relegada a la aplicación donde se integra el cliente, así pues, el encargado de crear un método para enviar los ficheros devueltos por el cliente de firma al servidor es el propio integrador, ya que el cliente de firma en ningún momento se conecta con el servidor de firma, es un proceso independiente.
2. Si el método **isError** devuelve **true** el método **getErrorMessage** devuelve una cadena con el mensaje de error.

Por ejemplo:

```
clienteFirma.initialize();
var rutaFicheroFirma= clienteFirma.webSign(html);
if(!clienteFirma.isError()) {
  document.body.formulario.inputFicheroFirma.value= rutaFicheroFirma;
} else {
  alert("Se ha producido un error: "+ clienteFirma.getErrorMessage());
}
```

Se provee una función JavaScript llamada **firmaWeb** en el fichero "***firmaWeb.js***" que recibe como parámetros un elemento HTML y un documento HTML y compone un HTML y lo firma como se ha descrito (y devuelve la ruta al fichero que contiene la firma). En resumen, este método genera un HTML a partir de un elemento HTML con los valores actuales de los campos, incluyendo el contenido de los ficheros, y lo firma. Esto ahorra al integrador generar un HTML *ad-hoc* con los datos introducidos por el usuario para su firma web. El fichero "***firmaWeb.js***" depende de otros, como vemos a continuación en un ejemplo práctico:

```
<script type="text/javascript" language="javascript" src="constantes.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/firma.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/htmlEscape.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/utils.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/styles.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/firmaWeb.js"></script>
<script type="text/javascript" language="javascript" src="../common-js/instalador.js"></script>
[…]

<script type="text/javascript" language="javascript">
  function enviar()
  {
    clienteFirma.initialize();

    var formulario = document.getElementById("formulario");
    var ruta = webSign(formulario, document);
    if(!clienteFirma.isError())
    {
      var fichero = document.getElementById("fichero");
      fichero.value = ruta;
      return true; // Enviar
    }
    else
    {
      alert("No se ha podido firmar: "+clienteFirma.getErrorMessage());
      return false;
    }
```

```
}
</script>
[…]

<form id="formulario" action="/enviarFirma">
  <input type="text" id="fichero" style="visibility: hidden; display: none;" value="">
  DNI: <input type="text"><br>
  <input type="submit" onclick="return enviar();">
</form>
```

NOTA IMPORTANTE:

La firma Web en formato XMLDSig Enveloped o XAdES Enveloped solo es posible realizarla cuando la página Web a firmar se encuentra en un formato compatible estrictamente con XML, como por ejemplo XHTML. Así mismos, estos formatos exigen que la firma se realice en modo implícito (**IMPLICIT**).

6.4 Firma electrónica

El proceso de firma electrónica es análogo al de firma Web, con la diferencia que los datos no tienen por qué ser HTML. El proceso varía pues, fundamentalmente, en la entrada de datos.

Se permiten diferentes tipos de datos a firmar (solo se puede firmar un tipo cada vez):

- **un fichero**: se establece qué fichero firmar mediante el método **setFileuri**, que recibe como parámetro de entrada una cadena con la ruta al fichero a firmar. Este método no comprueba en ningún momento la existencia de un fichero en la ruta indicada. Si el fichero no existiese se produciría un error durante la operación en cuestión.
- **datos**: se establecen mediante el método **setData**, que recibe una cadena con los datos codificados en base 64.
- **un hash**: se establece mediante el método **setHash**, que recibe una cadena con el hash codificado en base 64.
- Si no se invoca ninguno de estos métodos, el Cliente solicitará al usuario un fichero para firmar

En las firmas XML (XAdES y XMLsSig), en el caso de que los datos insertados estén en base 64 (ya sea mediante el **setFileuri** y un fichero de texto que contenga el base 64 de los datos o a través del **setData** y una cadena doblemente codificada en base 64), no se realizará la codificación interna en base 64 que requiere la firma XML para ficheros binarios. Así obtenemos que se firma la codificación base 64 de los datos y no una doble codificación en base 64 de estos. Este mismo comportamiento lo podemos

obtener mediante el método **setFileuriBase64** que establece como datos de entrada para las firmas electrónicas el contenido descodificado de un fichero en base 64.

Mientras que indicar con **setFileuri** un fichero con datos codificados en base 64 sólo aplica a las firmas XAdES y XMLdSig, el método **setFileuriBase64** funciona con todos los formatos de firma. Esto permite indicar los datos a firmar a través de un fichero que los contiene en base 64.

Previamente a la realización de la firma, es aconsejable la inicialización del cliente y su configuración con los parámetros preestablecidos. Esto podemos realizarlo con las funciones JavaScript **initialize()** y **configuraFirmar()**, que configura los siguientes parámetros según las variables indicadas del fichero *constantes.js"*:

- **Algoritmo de firma**: Determinado por la variable **signatureAlgorithm**. Por defecto, SHA1withRSA.
- **Formato de firma**: Determinado por la variable **signatureFormat**. Por defecto, CMS.
- **Filtro de certificados**: Determinado por la variable **certFilter**. Por defecto, ninguno.

Por ejemplo:

```
<script type="text/javascript" language="javascript">
function enviar()     {
  var fichero= document.getElementById("fichero");
  initialize();
  configurarFirma();
  clienteFirma.setFileuri(fichero.value);
  firmar();

  if(!clienteFirma.isError()) {
    var firmaB64 = document.getElementById("firmaB64");
    firmaB64.value = clienteFirma.getSignatureBase64Encoded();
    return true; // Enviar
  }
  else {
    alert("No se ha podido firmar: "+clienteFirma.getErrorMessage());
    return false;
  }
}
</script>
[…]
```

```
<form id="formulario" action="/enviarFirma">
  <input type="hidden" id="firmaB64"><br>
  Fichero a firmar: <input type="file" id="fichero">
  <input type="submit" onclick="return enviar();">
</form>
```

Pueden ejecutarse operaciones de firma, así como de cofirma y contrafirma desde el HTML de prueba *demoMultifirma.html*.

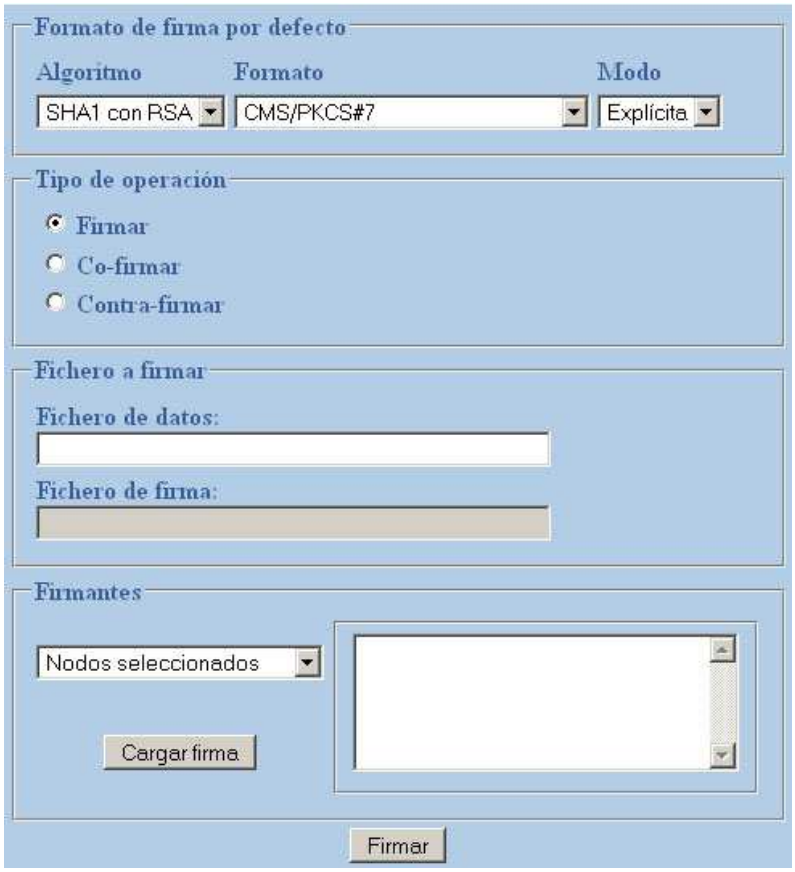


Figura 11: HTML de prueba demoMultifirma.html

6.5 Co-firma (*co-sign*)

La *co-firma* permite a varios usuarios firmar un mismo documento.

Una cofirma siempre firma los datos que se le indican, nunca se aplica ni depende de otra de las firmas del documento.

El caso de la *co-firma* es igual al de la firma simple, pero además de los datos hay que pasar al Cliente la firma electrónica de los demás firmantes. Esto se puede hacer de diferentes maneras:

- Mediante un fichero que contenga la firma electrónica, con el método **setElectronicSignatureFile()**, que recibe como parámetro una cadena con la ruta al fichero.
- Introduciendo directamente la firma, con el método **setElectronicSignature()** que recibe como parámetro una cadena con la firma en base 64.
- Si no se especifica, se pedirá al usuario que seleccione el fichero de firma.

Una vez especificados los parámetros necesarios, se invoca al método **coSign()**. La salida es análoga a la de la operación de firma.

Pueden ejecutarse operaciones de cofirma, así como de firma y contrafirma desde el HTML de prueba "*demoMultifirma.html*".

6.6 Contra-firma (*counter-sign*)

La contra-firma permite a un usuario firmar las firmas de otros usuarios.

El caso de la contra-firma es similar a los anteriores, pero sólo es necesario indicar la firma que deseamos contrafirma (no son necesarios los datos) y, según la operación concreta, puede ser necesario conocer la estructura de firmantes que contiene.

Para conocer la estructura de firmantes de una firma el Cliente dispone del método **getSignersStructure().** Este método devuelve una cadena que contiene los nombres de los firmantes separados por un retorno de carro ("**\n**" en JavaScript). Al comienzo del nombre hay tantos tabulados ("**\t**") como nivel ocupe el firmante en el documento. Por ejemplo, si A y B co-firman un documento y C contra-firma la firma de A, entonces la cadena devuelta sería "**A\n\tC\nB**".

La firma que deseamos contrafirmar se especifica mediante el método **setElectronicSignature()** o **setElectronicSignatureFile()**, que reciben la firma en base 64 y la ruta del fichero de firma, respectivamente.

En el fichero ***demoMultifirma.html*** se puede ver un ejemplo de cómo tratar esta cadena.

Se puede especificar qué firmas se desean firmar de diferentes maneras:

- Todas las firmas hojas (firmas no contra-firmadas): invocando el método **counterSignLeafs().**
- Todas las firmas: invocando el método **counterSignTree().**
- Todas las firmas de un firmante: configurando los firmantes con el método **setSignersToCounterSign()** que recibe como parámetro una cadena con los nombres de los firmantes separados por "**\n**" e invocando el método **counterSignSigners().**
- Firmas concretas: con el método **setSignersToCounterSign()** indicamos que firmas deseamos contrafirmar a partir de su posición (partiendo de 0) según el orden de aparición en la estructura devuelta por **getSignersStructure**(). Las posiciones se indican con números separados por "**\n**". Por ejemplo, "***0\n3\n4***" indica que se contrafirmen las firmas de las posiciones 0, 3 y 4. Se invoca con el método **counterSignSigners().**

La salida es análoga a la de la firma digital.

Téngase en cuenta que las contrafirmas siempre aplican a una firma y se colocan bajo esta en el árbol de firmas, al contrario que las cofirmas, que siempre se colocan como un nodo dependiente de los datos.

Pueden ejecutarse operaciones de contrafirma, así como de firma y cofirma desde el HTML de prueba "*demoMultifirma.html*".

NOTA IMPORTANTE: Dado que las contrafirmas se aplican sobre las firmas previas y no sobre los propios datos, no es posible (no es conceptualmente correcto) realizar contrafirmas multi-fase, es decir, las huellas digitales se calculan al vuelo siempre (no se admiten huellas digitales pre-generadas externamente), ya que estas se generan en base a las firmas, no a los datos.

6.7 Firma y Multifirma Masiva

6.7.1 Consideraciones previas

Un aspecto importante que debe tenerse en cuenta en todas las operaciones de firma y multifirma masiva es que los procesos no son interactivos a nivel de operación individual, es decir, que no se requiere intervención del usuario y este no recibe información ni notificaciones hasta que finaliza el proceso completo, tanto si han ocurrido errores durante su desarrollo como si transcurrió sin incidencias.

Este modo de operar permite que, si por ejemplo se inicia un proceso de 2.000 firmas, el usuario pueda despreocuparse hasta su finalización, y que este no se detendrá en una firma aunque ocurriese un error (sea cual sea este). Siguiendo el ejemplo, si el usuario iniciase la firma de los 2.000 ficheros y desatendiese el proceso pensando que este tardará una o dos horas, el proceso no se habrá detenido porque el dichero número 3 estuviese corrupto, sino que se firmarían los 1.999 restantes y en el informe final de operación se marcarán las incidencias ocurridas.

Una excepción a esta regla es el uso de dispositivos de firma que requieren la introducción de un PIN / contraseña o una confirmación para cada una de las operaciones de firma (como el DNIe). Aunque los mensajes y diálogos de aplicación se pospondrán a la finalización total de las tareas, esta confirmación o introducción de PIN no puede ser omitida, por lo que el usuario debe realizarla por cada operación individual.

Consulte el punto "**¡Error! No se encuentra el origen de la referencia.**" para más información sobre cómo se muestran los errores en los procesos de firma y multifirma masiva.

6.7.2 Modo de operación basado en ficheros

La firma/multifirma masiva (firma masiva en adelante) consiste en la firma o multifirma de varios ficheros en una única operación.

La firma masiva sólo permite firmar a partir de ficheros, para lo que será necesario establecer un directorio que contenga todos los ficheros que se deseen firmar. El tipo de operación a realizar se especificará mediante **setMassiveOperation,** lo que nos permitirá realizar una firma masiva simple (**FIRMAR**), cofirmar (**COFIRMAR**) o contrafirmar todas las firmas al completo que encontremos (**CONTRAFIRMAR_ARBOL**) o tan sólo las firmas hoja (**CONTRAFIRMAR_HOJAS**). La operación se ejecutará mediante el método **signDirectory** del cliente y, en caso de no haber especificado ningún directorio, se mostrará la pantalla para su selección.

Los ficheros que se firmaran durante la operación pueden ser filtrados por extensión. Para esto se usará el método **setInIncludeExtensions** que recibe las extensiones de los ficheros que se deben procesar separadas por comas (",").

Por ejemplo:

```
clienteFirma.setInIncludeExtensions("txt,xml,p7s");
```

También es posible indicar que se desean procesar los ficheros de los subdirectorios del directorio indicado. Esto se hace con **setInRecursiveDirectorySign**.

En el caso de las operaciones de multifirma es muy recomendable utilizar el mismo formato de firma del que ya dispusiese la firma original. Para indicar que se desea respetar este formato debe usarse el método **setOriginalFormat**. En caso de tratarse de una operación de firma masiva o no desear respetar el formato original del fichero de firma, se realizará una operación de firma conforme la configuración establecida mediante el mecanismo tradicional.

Por tema de eficiencia la firma masiva siempre se realizará en modo explícito (aunque se indique lo contrario), por lo que será necesario indicar un formato por defecto que admita este modo de firma.

Según el tipo de operación masiva que se haya solicitado y el tipo de fichero que se encuentre durante la misma se realizará una u otra acción:

- **Firma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se firmará con la configuración de firma establecida.
- **Cofirma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se extraerán, siempre que sea posible, los datos implícitos de la firma y se cofirmará el fichero.
- **Contrafirma:**
  - **Fichero binario:** Se ignorará.
  - **Fichero de firma:** Se contrafirmará completamente o sólo las firmas hoja según tipo de operación (**CONTRAFIRMAR_ARBOL** o **CONTRAFIRMAR_HOJAS**).

En cada caso, se entenderá como fichero de firma todo aquel que sea una firma en el formato configurado, o en cualquier formato si se ha solicitado mantener el formato original. El resto de ficheros son considerados ficheros binarios.

Las firmas resultado de esta operación se almacenarán en el directorio establecido con el método **setOutputDirectoryToSign**. El método creará los ficheros de firma con el mismo nombre que el fichero original (extensión incluida) y la extensión apropiada según el formato de la firma. En el caso de la cofirma y contrafirma se insertarán las partículas "*.cosign*" y "*.countersign*", respectivamente, antes de la extensión de firma. En caso de no indicar un directorio de salida se tomará el mismo directorio en donde se encuentren los ficheros de entrada.

En el mismo directorio de salida se creará un fichero de log (***result.log***) en donde se registrará el resultado de cada una de las acciones realizadas durante la operación masiva.

En caso de producirse uno o más errores durante el proceso el método **signDirectory** devolverá **false**, pero no se detendrá hasta haber finalizado la operación. Para conocer con más detalle la causa de los errores que puedan producirse será necesario consultar el fichero de log.

Un ejemplo del uso de esta funcionalidad es:

```
clienteFirma.initialize();

clienteFirma.setSignatureFormat("CADES");
clienteFirma.setSignatureAlgorithm("SHA1withRSA");
clienteFirma.setInputDirectoryToSign("C:/ficheros");
clienteFirma.setOutputDirectoryToSign("C:/firmas");
clienteFirma.setInIncludeExtensions("csig");
clienteFirma.setInRecursiveDirectorySign(true);
clienteFirma.setMassiveOperation("CONTRAFIRMAR_HOJAS");

clienteFirma.signDirectory();

if(!clienteFirma.isError()) {
  alert("La operacion finalizo con exito");
}
else {
  alert("Se detectaron errores durante el proceso de firma consulte el log de error para más
información");
}
```

Puede verse el funcionamiento de la multifirma masiva basada en ficheros en el HTML de prueba demoFirmaDirectorios.html.

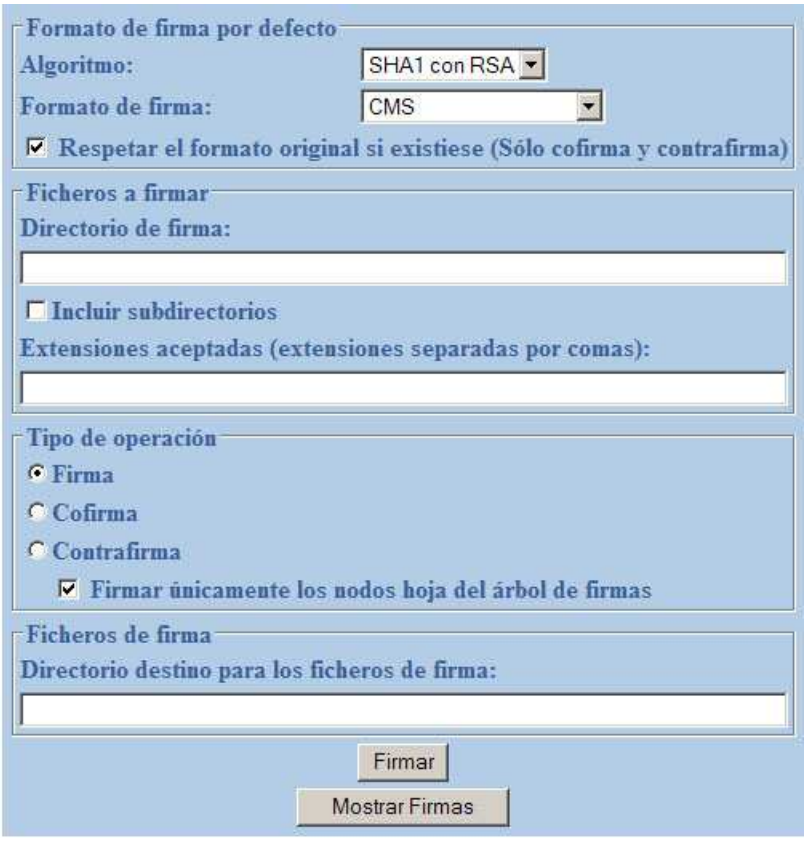


Figura 12: HTML de prueba demoFirmaDirectorios.html

6.7.3 Modo de operación programática

Adicionalmente a la metodología de firma/multifirma masiva ya comentada, se ha desarrollado un nuevo procedimiento para la firma independiente de datos, ficheros y hashes en base a una configuración única de firma.

El procedimiento a seguir para realizar esta operación es el siguiente:

1. Configuración del cliente.
2. Inicialización de la operación masiva.
3. Firma masiva de los datos.
4. Finalización de la operación.

Configuración del cliente

Los aspectos configurables del cliente que afectan a la operación masiva son:

- **Operación masiva a realizar** (Firma, cofirma y contrafirma de nodos hoja o del árbol completo de firma).

- **Algoritmo de firma** (SHA1withRSA, MD5withRSA, SHA512withRSA, etc.).
- **Formato** con el que realizar las firmas (CMS, XAdES Detached, PDF, ODF…).
- Si se debe **respetar el formato original** que, en el caso de las operaciones cofirma y contrafirma, significa detectar el formato de las firmas introducidas para multifirmar con el mismo formato.
- **Modo de firma** (Implícita o explícita).
- **Certificado con el que firmar**.

La configuración de estos parámetros se realiza mediante los métodos:

- setMassiveOperation(String)
- setSignatureAlgorithm(String)
- setSignatureFormat(String)
- setOriginalFormat(boolean)
- setSignatureMode(String)
- setSelectedCertificateAlias(String)

Fuera de los métodos tradicionalmente utilizados en el applet, se encuentran:

- **setMassiveOperation(String),** que configura el tipo de operación masiva y puede recibir los parámetros **FIRMAR** (firmar datos), **COFIRMAR** (cofirmar datos a partir de un objeto de firma que mantenga una referencia válida a esos datos), **CONTRAFIRMAR_ARBOL** (contrafirmar todas las firmas de un objeto de firma) y **CONTRAFIRMAR_HOJAS** (contrafirmar tan sólo las firmas hoja).
- **setOriginalFormat(boolean)**, que recibe el parámetro **true** o **false** según se desee respetar o no el formato original durante las operaciones de cofirma y contrafirma. El comportamiento de esta opción es el siguiente:
    - Si la opción está activada: Se identificará el formato de la firma original y se multifirmará en este formato.
    - Si no está activada la opción: Se comprobará el formato de la firma, si es compatible con el formato indicado se firmará en ese formato y si no es compatible se indicará al usuario y no se realizará la multifirma.

Es decir, que si está activada esta función y, por ejemplo, indicamos que se cofirme en CAdES una firma CMS, se ignorará el formato indicado y se cofirmará en CMS (el formato original). Si fuese el caso

contrario, firma CAdES y se solicita una cofirma CMS, se ignoraría el CMS y se firmaría en CAdES. Si la opción "respetar el formato original" estuviese desactivada (*setOriginalFormat(false)*) se multifirmaría siempre en el formato indicado, o se informaría mediante un mensaje de error que el fichero indicado no es un fichero de datos o firma compatible si el formato indicado no lo soportase.

El mantener activa esta opción es útil cuando no se conozca el formato en el que fuesen originalmente firmados los datos o queramos evitarnos el seleccionarlo para cada elemento de firma, mientras que el desactivarlo evita que se realice el proceso de búsqueda del formato original y, que de seleccionar un formato equivocado, se nos informe.

Inicialización de la operación masiva

El proceso de inicialización configura los parámetros ya comentados en el módulo de firma masiva y reinicia el registro de mensajes (*log*) del módulo. Desde el momento de la inicialización y hasta que se finalice el proceso de firma masiva estos parámetros, a **excepción del tipo de operación** (**setMassiveOperation(String)**), permanecen inalterados a lo largo de las operaciones realizadas por el módulo, aunque sí afectarán los cambios de configuración al resto de funcionalidades del cliente.

En caso de que no se hubiesen establecido todas las propiedades necesarias para la configuración de la firma masiva se tomarán los valores por defecto establecidos por el cliente. Estos son:

- **Operación:** Firma.
- **Algoritmo:** SHA1 con RSA.
- **Formato:** CMS.
- **Respetar formato original:** Activado.
- **Modo:** Explícito.

En el caso del alias, si no se ha establecido ninguno, se mostrará un diálogo para permitir seleccionar el certificado de firma al inicializar el proceso de firma masiva.

La inicialización del proceso de firma masiva se realiza mediante el método **initMassiveSignature()**.

Firma masiva de los datos

Existen 3 métodos para firmar, cofirmar o contrafirmar (nodos u hojas), según sea la operación configurada para el proceso:

- massiveSignatureData(String)
- massiveSignatureFile(String)
- massiveSignatureHash(String)

El método **massiveSignatureData(String)** realiza la operación configurada sobre los datos que recibe en forma de cadena de texto en base 64; **massiveSignatureFile(String)** ejecuta la operación sobre el fichero cuya ruta recibe cómo parámetro y **massiveSignatureHash(String)** lo hace sobre un hash en base 64.

A diferencia del resto de parámetros de la configuración de la firma masiva, es posible modificar el tipo de operación que se desea en cualquier momento durante su desarrollo. Para esto sólo es necesario utilizar el método **setMassiveSignatureOperation(String)** en el momento en el que se desee modificar la configuración.

El comportamiento de cada una de las operaciones simples podrá variar según el tipo de fichero que se les proporcione:

- **Firma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se firmará con la configuración de firma establecida.
- **Cofirma:**
  - **Fichero binario:** Se firmará con la configuración de firma establecida.
  - **Fichero de firma:** Se extraerán, siempre que sea posible, los datos implícitos de la firma y se cofirmará el fichero.
- **Contrafirma:**
  - **Fichero binario:** Se ignorará.

Fichero de firma: Se contrafirmará completamente o sólo las firmas hoja según tipo de operación (**CONTRAFIRMAR_ARBOL** o **CONTRAFIRMAR_HOJAS**).

IMPORTANTE: Téngase en cuenta las siguientes consideraciones:

- Las operaciones de cofirma y contrafirma no pueden realizarse sobre hashes ya que desde estos no pueden obtenerse los datos originales.

- Determinados formatos de firma pueden exigir que sea necesario firmar sobre los datos o un fichero, no siendo posible firmar hashes. Por ejemplo, los formatos XML enveloped, ODF y PDF.
- La operación de firma recibe los datos (mediante cualquiera de los 3 métodos comentados) mientras que la cofirma y las contrafirmas reciben una firma implícita con formato previamente generada.
- La contrafirma se aplica sobre firmas y es indiferente que estas almacenen datos implícitos o no, pero la cofirma requiere los datos originales para ser firmados por lo que es obligatorio que se proporcione una firma con los datos implícitos o, al menos, una explícita realizada con el mismo algoritmo de firma con el que se solicita la cofirma, para así poder reutilizar el hash que almacena.
- En el caso de firmas con formato especial orientado a documentos concretos en donde la firma va incrustada en el propio documento (firmas PDF, ODF…) la operación de cofirma comúnmente supondrá el agregar nueva información de firma al documento.

Las operaciones masivas devuelven su resultado en forma de cadena en base 64. En caso de producirse algún error se devolverá **null** y en ningún caso se lanzará una excepción, permitiendo al integrador obviar la captura de éstas, o se interrumpirá el proceso.

Cada operación individual de la firma masiva realizada generará una entrada en el registro de mensajes (log). En el caso de finalizar la operación correctamente esta simplemente lo indicará, mientras que en el caso de error la entrada explicará el error producido.

Finalización de la operación

La finalización de la operación elimina la configuración de operación masiva establecida por lo que ya no es posible continuar operando hasta que se vuelva a inicializar. Tras ser finalizada la operación, la nueva inicialización podría tomar una nueva configuración de firma establecida.

El método para llevar a cabo la finalización de la operación masiva es **endMassiveSignature()**.

El finalizar la operación no elimina los mensajes de registro (*log*) generados durante la misma, por lo que es posible seguir accediendo a ellos. Sí, en cambio, los eliminará el iniciar una nueva operación de firma masiva.

Registro de mensajes de la operación masiva

Por cada operación individual de firma/multifirma realizada durante el proceso masivo se genera una entrada en el registro de mensajes. Para obtener, tras una operación individual, el mensaje generado se debe utilizar el método **getMassiveSignatureCurrentLog()**. La forma de este registro será:

- Operación sobre TIPO_DATO: MENSAJE.

En donde TIPO_DATO será la palabra "datos", "fichero" o "hash" según el método utilizado para la operación (massiveSignatureData, massiveSignatureFile o massiveSignatureHash respectivamente); y MENSAJE será el mensaje obtenido, "Correcta" en el caso de que la operación finalizase correctamente o la explicación del error en caso de que se produjese.

Puede obtenerse todo el log generado hasta el momento para su proceso mediante el método **getMassiveSignatureLog()**. El texto que devuelve este método se compone de todas las entradas del mismo con el formato indicado separadas por un retorno de carro ("\r\n").

Puede almacenarse este mismo log en disco mediante la función **saveMassiveSignatureLog()**, que lo almacenará en la ruta indicada con el método **setOutFilePath(String)**. Si no se ha establecido ningún fichero de salida se mostrará un diálogo de guardado para seleccionar en donde se desea almacenar el fichero.

El registro de mensajes permanecerá aun cuando se finalice la operación masiva, pero se reiniciará en cada nueva inicialización del proceso.

Guardado de firmas en disco

Las firmas resultantes de la operación de firma masiva se devuelven en base 64 por cada operación de firma individual (realizadas con massiveSignatureData(String), massiveSignatureFile(String) o massiveSignatureHash(String)), por lo cual el cliente no las almacena internamente como hace con las operaciones de firma simple. Por este motivo, el simple uso del método de guardado de firma del cliente no aplica a esta situación, en su lugar se puede utilizar la siguiente sucesión de llamadas a métodos:

- **setElectronicSignature(String)**: Recibe como parámetro la firma en base64 y la guarda internamente.
- **setOutFilePath(String)**: Establece el fichero de salida. Para permitir al usuario que seleccione el nombre y directorio de salida para cada fichero firmado, se le pasará el parámetro **null**.

- **saveSignToFile()**: Almacena la firma en el directorio de salida indicado.

Ejemplo Java de operación masiva

```
// Creamos una instancia del applet (innecesario para su uso en Web)
SignApplet clienteFirma = new SignApplet();

// Configuramos la operación que deseamos
clienteFirma.setMassiveOperation("FIRMAR");
clienteFirma.setSignatureFormat("CMS");
clienteFirma.setSignatureMode("IMPLICIT");

// Inicializamos la operación (en este momento se nos pedirá seleccionar un
// certificado de firma)
clienteFirma.initMassiveSignature();

// Una vez inicializada la operación, cualquier cambio en el algoritmo,
formato,
// tipo de operación, etc. no será tenido en cuenta para la operación masiva,
// aunque sí para el resto de operaciones del cliente

// Vector en el que almacenar los resultados en base 64
Vector<String> firmasB64 = new Vector<String>();

// Firmamos y almacenamos los datos

// Firma de ficheros
firmasB64.add( clienteFirma.massiveSignatureFile("C:\\Fichero.txt") );
firmasB64.add( clienteFirma.massiveSignatureFile("C:\\Fichero.xml") );
firmasB64.add( clienteFirma.massiveSignatureFile("C:\\Fichero.odt") );

// Firma de datos
firmasB64.add( clienteFirma.massiveSignatureData(
     clienteFirma.getFileBase64Encoded("C:\\Fichero.txt", true)
));
firmasB64.add( clienteFirma.massiveSignatureData(
     clienteFirma.getFileBase64Encoded("C:\\Fichero.xml", true)
));
firmasB64.add( clienteFirma.massiveSignatureData(
```

```
     clienteFirma.getFileBase64Encoded("C:\\Fichero.odt", true)
));

// Firma de hashes
clienteFirma.setFileuri("C:\\Fichero.txt");
firmasB64.add( clienteFirma.massiveSignatureHash(
     clienteFirma.getFileHashBase64Encoded(true)
));
clienteFirma.setFileuri("C:\\Fichero.xml");
firmasB64.add( clienteFirma.massiveSignatureHash(
     clienteFirma.getFileHashBase64Encoded(true)
));
clienteFirma.setFileuri("C:\\Fichero.odt");
firmasB64.add( clienteFirma.massiveSignatureHash(
     clienteFirma.getFileHashBase64Encoded(true)
));

// Finalizamos la operación
clienteFirma.endMassiveSignature();

// Almacenamos el log preguntando al usuario donde lo desea almacenar
clienteFirma.saveMassiveSignatureLog();

// Además de almacenarlas en un vector queremos guardarlas en disco (en este
caso no mantenemos referencias a los ficheros originales)
for(int i=0; i<firmasB64.size(); i++) {
     if(firmasB64.get(i) != null) {
        clienteFirma.setElectronicSignature(firmasB64.get(i));
        clienteFirma.setOutFilePath("firma"+i+".p7s");
        clienteFirma.saveSignToFile();
     }
}

// Mostramos un mensaje de error al usuario por cada error obtenido
String[] mensajes =
clienteFirma.getMassiveSignatureLog().trim().split("\r\n");
for(int i=0; i<firmasB64.size(); i++) {
      if(firmasB64.get(i) == null) {
            JOptionPane.showMessageDialog(
```

```
                clienteFirma, mensajes[i], "Error", JOptionPane.ERROR_MESSAGE
            );
        }
}
```

Puede verse el funcionamiento de la multifirma masiva basada en ficheros en el HTML de prueba demoFirmaDirectorios.html.

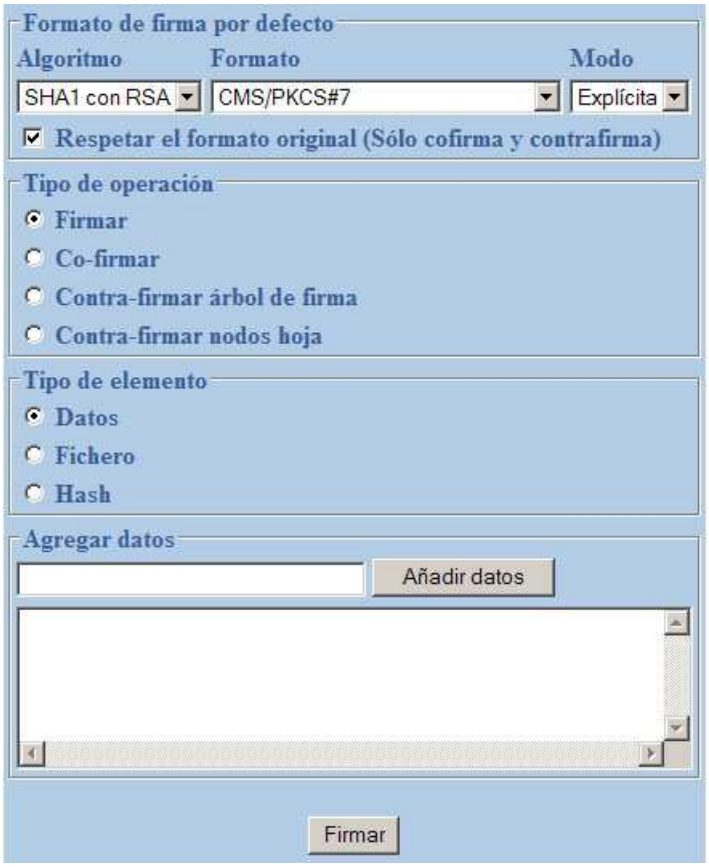


Figura 13: HTML de prueba demoMultifirmaMasiva.html

6.8 Cifrado de datos

Conviene reinicializar la configuración el cliente antes de invocar las funciones de cifrado debido a que comparte recursos con los procesos de firma y podría haber incompatibilidad en la entrada de datos. Para esta tarea puede utilizarse el método "initialize()" de la biblioteca JavaScript llamada "***cripto.js***" que reinicia las propiedades del cliente a sus valores por defecto.

Para iniciar el proceso de cifrado habrá que introducir previamente los datos a cifrar. Es posible especificar los datos a cifrar de diferentes formas:

- **datos o texto en plano**: se especifica cuál es la cadena a cifrar mediante el método **setPlainData**. Internamente se realizará las codificaciones correspondientes para garantizar la fiabilidad del cifrado y posterior descifrado.
- **fichero**: es posible especificar que los datos a cifrar provienen de un archivo indicándole la ruta a la llamada del cifrador (**cipherFile**), o bien, usando la función **setFileuri** para especificar dicho archivo. En ambos casos habrá que usar la ruta absoluta del fichero.

Por defecto el cliente de cifrado define como algoritmo de cifrado AES y generación automática de clave, aunque posteriormente veremos las posibilidades de configuración de estos parámetros. Tras indicar la configuración del cifrador y los datos a cifrar podemos realizar la llamada al método ***cipherData*** o ***cipherFile*** del Applet, o la correspondiente función JavaScript **cifrar** (en ***cripto.js***). El comportamiento de la llamada es análogo al resto de llamadas al Applet, indicando si la ejecución se ha llevado a cabo de forma correcta o los errores en caso negativo.

Los datos cifrados se podrán obtener una vez haya finalizado mediante la llamada al método **getCipherData**, devolviendo este una cadena codificada en formato Base 64. Es posible almacenar los datos cifrados en un archivo mediante la función **saveCipherDataToFile**, a la cual le pasaremos la ruta absoluta del archivo destino (atención, el archivo destino será sobrescrito para evitar problemas a la hora de descifrar). El contenido del archivo destino no se codifica, por lo que no se recomienda su edición, ya que pudiera alterar gravemente el contenido plano del mensaje cifrado o incluso destruirlo.

Un ejemplo de aplicación de lo anterior para un proceso completo de cifrado sería el siguiente:

```
<html>
  <head>
    <script type="text/javascript" language="javascript" src="constantes.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/cripto.js"></script>
    <script type="text/javascript" language="javascript">
      function cifrar()
      {
        var texto= document.getElementById("campo1").value;
        clienteFirma.initialize();
        clienteFirma.setKeyMode(“GENERATEKEY”);
        clienteFirma.setCipherAlgorithm(“AES”);
```

```
        clienteFirma.setPlainData(texto);
        clienteFirma.setShowErrors(false);
        cifrarDatos();
        if(!clienteFirma.isError()){
          var datosCifrados = clienteFirma.getCipherData();
          var campoCifrado =document.getElementById("campo2");
          campoCifrado.value = datosCifrados;
          return true;
        }else{
          alert("No se ha podido cifrar los datos: "+clienteFirma.getErrorMessage());
          return false;
        }
       }
    </script>
    […]

</head>
  <body>
    <script type="text/javascript">
      cargarAppletFirma();
    </script>
    […]
      <label>Datos planos</label><br/>
      <textarea id="campo1" cols="20" rows="5" nowrap>Introduzca texto plano aquí</textarea>
      <br/><br/><input type="button" value="Cifrar" onClick="cifrar();"/><br/><br/>
      <label>Datos cifrados</label><br/>
      <textarea id="campo2" cols="20" rows="5" nowrap readonly></textarea>
     […]
  </body>
</html>
```

Este ejemplo básico captura el texto introducido en un área de texto, la cifra con generación automática de clave y el algoritmo AES y la muestra en un segundo área de texto tras pulsar un botón. Para más información, consultar el ejemplo incluido y la documentación adicional.

Puede verse las distintas configuraciones de cifrado y descifrado de datos en el HTML de ejemplo demoCifrado.html.

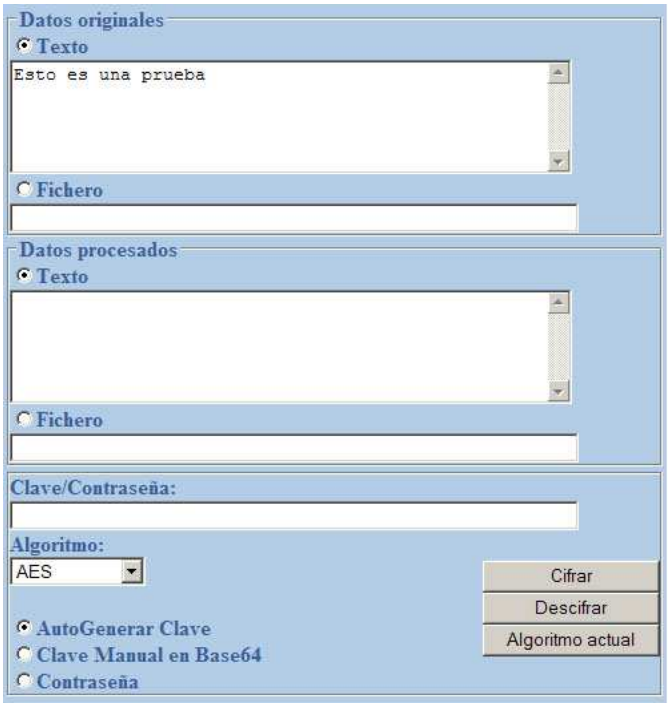


Figura 14: HTML de ejemplo demoCifrado.html

6.9 Descifrado de datos

De manera similar al cifrado, deberemos especificar cuáles son los datos a descifrar, y al igual que antes podremos especificar los datos cifrados mediante dos métodos distintos:

- **datos o texto cifrado**: se especifica cuál es la cadena a descifrar mediante el método **setCipherData**. Los datos de entrada estarán en base 64 (igual que la salida del algoritmo de cifrado) para evitar la aparición de caracteres extraños o no imprimibles. Internamente estos datos se decodificaran a la base apropiada y se descifrarán.
- **fichero**: también es posible especificar que los datos a cifrar provienen de un archivo indicándole la ruta (**decipherFile**), o usando la función **setFileuri** para especificarla. También aquí se deberá especificar la ruta absoluta del fichero.

Evidentemente para descifrar datos no podremos auto generar una clave, sino que tendremos que especificarle una siempre. En caso que se intente iniciar el método de descifrado sin especificar la clave supondrá un fallo automático. Los datos descifrados se pueden recuperar mediante la llamada a la función **getPlainData**. También tenemos un método para escribir estos datos recuperados a un archivo mediante la llamada a **savePlainDataToFile** y pasándole la ruta absoluta del archivo destino.

Un ejemplo básico para descifrar sería el siguiente:

```
<html>
  <head>
    <script type="text/javascript" language="javascript" src="constantes.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/deployJava.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/instalador.js"></script>
    <script type="text/javascript" language="javascript" src="common-js/cripto.js"></script>
    <script type="text/javascript" language="javascript">
        function descifrar()
        {
           var textoCifrado= document.getElementById("campo1").value;
           var clave=document.getElementById("clave").value;
           var archivoOrigen=document.getElementById("origen").value;
           clienteFirma.initialize();
           clienteFirma.setKey(clave);
           clienteFirma.setKeyMode("USERINPUT");
           clienteFirma.setCipherAlgorithm("AES");
           descifrarArchivo(archivoOrigen);
           if(!clienteFirma.isError()){
             var datosPlanos = clienteFirma.getPlainData();
             var campoPlano=document.getElementById("campo2");
             campoPlano.value = datosCifrados;
             var archivoDestino=document.getElementById("destino").value;
             clienteFirma.savePlainDataToFile(archivoDestino);
             return true;
           }else{
             alert("No se ha podido descifrar los datos: "+clienteFirma.getErrorMessage());
             return false;
           }
        }
    </script>
    […]
  </head>
  <body>
    <script type="text/javascript">
       cargarAppletFirma();
    </script>
  […]
    <label>Fichero cifrado:</label>
    <input type="file" id="origen"/>
    <label>Fichero plano (introduzca URI):</label>
    <input type="text" id="destino" value=""/>
    <br/><br/><input type="button" value="Descifrar" onClick="descifrar();"/><br/><br/>
    <label>Datos descifrados</label><br/>
    <textarea id="campo2" cols="20" rows="5" nowrap readonly></textarea>
  […]
```

```
  </body>
</html>
```

Puede verse las distintas configuraciones de cifrado y descifrado de datos en el HTML de ejemplo demoCifrado.html.

6.10 Creación de estructuras CMS cifradas (sobres digitales)

El cliente permite la posibilidad de incluir datos cifrados en un mensaje CMS. Se contemplan tres estructuras distintas que se describirán más detenidamente posteriormente. Estas son:

- CMS encriptado (*EncryptedCMS*).
- CMS envuelto (sobre digital, *EnvelopedCMS*).
- PKCS#7 firmado y envuelto (*Signed&Enveloped*).

Las dos primeras opciones están dentro de RFC 3852:"*Cryptographic Message Syntax*" correspondiente a las versión 3 de CMS, pero debido a sus características, CMS envuelto no es compatibles con PKCS#7.

6.10.1 CMS encriptado

Esta estructura está basada en un mensaje criptográfico que sólo contiene el texto cifrado y opcionalmente el algoritmo utilizado para el cifrado. No contiene ninguna información sobre la clave, emisor o receptor. La metodología de creación es:

1. Se establece los datos a incluir en el mensaje mediante una llamada a **setData**, pasándole en base 64 los datos que se desean incluir en el mensaje, o **setFileuri**, para incluir un fichero. Opcionalmente se definen el algoritmo de cifrado, la clave y el modo de clave.
2. Se realiza una llamada a **buildCMSEncrypted**.
3. El CMS generado se puede recuperar como un String codificado en base 64 mediante el método **getB64Data** o guardarla en un archivo con la operación **saveDataToFile**.

6.10.2 CMS envuelto

Mediante la creación de un CMS envuelto obtenemos un sobre digital en el cual podremos incluir contenido cifrado sólo visible por los receptores que le indiquemos. Posteriormente veremos la estructura generada y comentaremos algunos detalles sobre ella.

El procedimiento de creación es el siguiente:

1. Definimos los datos a incluir en el sobre digital de igual manera que en el apartado anterior, indicando los datos en base 64 mediante **setData** o un fichero mediante **setFileuri**. Definimos también el resto de parámetros opcionales.

2. Establecemos los receptores válidos para el mensaje mediante una llamada a la función **setRecipientsToCMS** especificándole como parámetros una cadena con los diferentes archivos con la clave pública de los diferentes sujetos separados por retornos de carro ("\n"). Estos ficheros deberán indicar su ruta completa y pueden ser formato CER o DER. Pueden eliminarse los receptores indicados llamando a este método con el parámetro **null**. Para la generación del sobre será necesario indicar al menos un receptor válido.

   De forma independiente a los receptores indicados mediante el método **setRecipientsToCMS**, es posible configurar receptores adicionales mediante el método **addRecipientToCMS** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **removeRecipientToCMS**.

3. Hacemos la llamada al método **buildCMSEnveloped**. Tras la llamada nos solicitará que indiquemos el emisor del mensaje mediante la selección de nuestro certificado digital, aunque es opcional indicar el emisor es recomendable.

4. Una vez concluida, podremos obtener el resultado mediante la operación **getB64Data** o guardarla en un archivo con la operación **saveDataToFile**.

6.10.3 PKCS#7 firmado y envuelto

Similar al CMS envuelto, pero los datos además de cifrarse son firmados por el emisor. El procedimiento es el siguiente:

1. Definimos los datos a incluir en el sobre digital de igual manera que en el apartado anterior, indicando los datos en base 64 mediante **setData**. Para incluir un fichero le indicaremos la dirección absoluta del fichero en la llamada **setFileuri**.

2. Establecemos los receptores válidos para el mensaje mediante una llamada a la función **setRecipientsToCMS** especificándole como parámetros una cadena con los diferentes archivos con la clave pública de los diferentes sujetos separados por retornos de carro ("\n"). Estos ficheros deberán indicar su ruta completa y pueden ser formato CER o DER. Pueden

eliminarse los receptores indicados llamando a este método con el parámetro **null.** Para la generación del sobre será necesario indicar al menos un receptor válido.

De forma independiente a los receptores indicados mediante el método **setRecipientsToCMS**, es posible configurar receptores adicionales mediante el método **addRecipientToCMS** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **removeRecipientToCMS**.

3. Hacemos la llamada al método **signAndPack** si hemos especificado datos o al método **signAndPackFile** utilizando la ruta del fichero. Tras la llamada nos solicitará que indiquemos el emisor del mensaje mediante la selección de nuestro certificado digital, en esta ocasión es obligatoria para firmar los datos.

Una vez concluida, podremos obtener el resultado mediante la operación **getB64Data** o guardarla en un archivo con la operación **saveDataToFile**.

6.10.4 CMS autenticado

Similar al PKCS#7 firmado y envuelto. El ensobrado firmado y envuelto contaba con una vulnerabilidad que hacía posible que se eliminase la firma del sobre y se sustituyese por otra, sin posibilidad de advertir el cambio. El ensobrado CMS autenticado corrige este problema haciendo que parte de la información del firmante quede registrada en el propio sobre.

El procedimiento para generar un sobre CMS autenticado es el siguiente:

1. Definimos los datos a incluir en el sobre digital indicándolos en base 64 mediante **setData**. Para incluir un fichero le indicaríamos la dirección absoluta del fichero en la llamada **setFileuri**.
2. Establecemos los receptores válidos para el mensaje mediante una llamada a la función **setRecipientsToCMS** especificándole como parámetros una cadena con los diferentes archivos con la clave pública de los diferentes sujetos separados por retornos de carro (“\n”). Estos ficheros deberán indicar su ruta completa y pueden ser formato CER o DER. Pueden eliminarse los receptores indicados llamando a este método con el parámetro null. Para la generación del sobre será necesario indicar al menos un receptor válido.

   De forma independiente a los receptores indicados mediante el método **setRecipientsToCMS**, es posible configurar receptores adicionales mediante el método

addRecipientToCMS que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **removeRecipientToCMS**.

3. Hacemos la llamada al método **buildCMSAuthenticated**. Tras la llamada nos solicitará que indiquemos el emisor del mensaje mediante la selección de nuestro certificado digital, obligatorio para poder autenticar los datos.
4. Una vez concluida, podremos obtener el resultado mediante la operación **getB64Data** o guardarla en un archivo con la operación **saveDataToFile**.

7 Configuración del Cliente

7.1 Inicialización de las operaciones

Antes de iniciar una operación criptográfica se debe invocar el método **`initialize()`** del Cliente, que borra las entradas y salidas de operaciones anteriores.

En las bibliotecas JavaScript "***firma.js***" y "constantes.js" se incluye un método **`initialize()`** que invoca al **`initialize()`** del cliente y configura diversos parámetros.

7.2 Cambio de almacén de certificados

Al ejecutar el cliente @firma como applet se configura por defecto el almacén de certificados del navegador o sistema operativo sobre el que se ejecuta. Según la configuración navegador/sistema operativo el almacén de certificados por defecto será:

	Internet Explorer	**Mozilla Firefox**	**Google Chrome**	**Apple Safari**
Windows	Almacén Windows (CAPI)	Almacén Mozilla	Almacén Windows (CAPI)	Almacén Windows (CAPI)
Linux / Solaris		Almacén Mozilla	Almacén Mozilla	
Mac OS X		Llavero Mac OS X	Llavero Mac OS X	Llavero Mac OS X

Cuando se ejecuta el cliente como biblioteca y no desde un navegador, se utilizará por defecto el almacén de Windows, aunque se recomienda configurarlo explícitamente.

El cliente @firma, sin embargo, permite la configuración de este almacén de certificados de tal forma que es posible indicar de qué almacén deben extraerse los certificados. Esta configuración se establece mediante el método **`setKeystore(String path, String pass, String type).`**

Este método recibe, por orden:

- **path**: La ruta al almacén de certificados que se desea utilizar (sólo para almacenes en disco). Si es necesaria para el tipo de almacén seleccionado y no se indica, se le mostrará un diálogo al usuario para que lo seleccione.
- **pass**: La contraseña para abrir el almacén. Aplica a cualquier almacén que pueda estar protegido por contraseña (PKCS#12/PFX, Mozilla Firefox configurado con clave maestra,…). Si no se indica y es necesaria se le mostrará un diálogo al usuario para que la inserte.
- **type**: Tipo de almacén de certificados. Los distintos parámetros admitidos son:
  - **WINDOWS**: Repositorio de Microsoft Windows (MSCAPI).
  - **APPLE**: Repositorio de Apple Macintosh (Llavero o KeyChain).
  - **MOZILLA**: Repositorio Mozilla. Para su uso en Windows es obligatorio tener instalado Mozilla Firefox. En el caso de Mac OS X, no es posible utilizar el almacén de Mozilla por lo que siempre se utilizará el del sistema operativo.
  - **P11**: Repositorio de tipo PKCS#11 accesible desde una biblioteca nativa del sistema. No es recomendable el uso directo de este tipo de almacén, en su lugar debería aconsejarse al usuario que instale el dispositivo y acceda a él a través del almacén de certificados de su navegador. Si no se indica, se le solicitará al usuario la ruta y contraseña de la biblioteca.
  - **P12**: Repositorios en disco en formato PKCS#12 o PFX. Si no se indica, se le solicitará al usuario la ruta y contraseña del almacén. Si se indica en la llamada al método la contraseña del almacén, se utilizará esta también para la selección de los certificados.
  - **JKS**: Repositorios en disco en formato JKS. Si no se indica, se le solicitará al usuario la ruta y contraseña del almacén. Si se indica en la llamada al método la contraseña del almacén, se utilizará esta también para la selección de los certificados.

- **SINGLE**: Certificado suelto en disco. Estos certificados sólo disponen de clave pública, por lo que no son aptos para firmar. Si no se indica, se le solicitará al usuario la ruta del certificado.
- **JAVACE**: Repositorios en disco en formato Java Case Exact. Si no se indica, se le solicitará al usuario la ruta y contraseña del almacén. Si se indica en la llamada al método la contraseña del almacén, se utilizará esta también para la selección de los certificados.
- **WINADDRESSBOOK**: Repositorio de Certificados de Otras Personas de Windows. Este almacén no contiene certificados personales de firma, por lo que no se recomienda su uso para tal fin.
- **WINDOWS-CA**: Repositorio de Certificados de Autoridades de Certificación de Windows. Este almacén no contiene certificados personales de firma, por lo que no se recomienda su uso para tal fin.
- **WINDOWS-ROOT**: Repositorio de Certificados Raíz de Windows. Este almacén no contiene certificados personales de firma, por lo que no se recomienda su uso para tal fin.

En caso de seleccionar un almacén no válido (el almacén de Apple en Windows, por ejemplo) u ocurrir un error durante su inicialización, el cliente se reconfigurará al almacén que se tuviese configurado en ese momento.

7.3 Selección y filtrado de certificados

7.3.1 Selección de los certificados para operaciones criptográficas

Muchas operaciones criptográficas de las soportadas por el cliente @firma requieren que se seleccione un certificado de usuario como, por ejemplo, la firma. Los certificados accesibles por el applet de firma son aquellos disponibles desde el repositorio de certificados del navegador y es posible seleccionar uno de ellos mediante el método del cliente **setSelectedCertificateAlias(String)** al que debe pasarse uno de los certificados recogidos mediante el método **getCertificatesAlias()**.

Es posible permitir al usuario seleccionar un certificado directamente a través de un diálogo de selección de certificados. Podemos mostrar este diálogo a través del método **showCertSelectionDialog()**, que devuelve el alias del certificado. Cuando el usuario selecciona un certificado a través de este método, este queda automáticamente seleccionado, de modo que es posible recuperarlo mediante los métodos

getSignCertificate() y **getSignCertificateBase64Encoded()**, detallados en el apartado 8.2 del manual.

En caso no seleccionarse un certificado, al realizar una operación criptográfica que lo requiera, se solicitará éste automáticamente al usuario mediante el diálogo de selección.

Figura 15: Selección de certificado

Para indicar los receptores de los sobre digitales se deberán introducir las direcciones de sus certificados exportados (ficheros CER o DER). El método utilizado es **setRecipientsToCMS(String)** y recibe una cadena con las rutas de los certificados separadas por el carácter '\n'. Por ejemplo:

```
clienteFirma.setRecipientsToCMS("C:/detinatario1.cer\nC:/destinatario2.cer");
```

De forma independiente a los receptores indicados mediante el método **setRecipientsToCMS**, es posible configurar receptores adicionales de sobre digital mediante el método **addRecipientToCMS** que recibe el certificado del receptor codificado en base 64. Para eliminar alguno de los receptores agregados mediante este método puede utilizarse **removeRecipientToCMS**.

Un punto importante a destacar es que el método **getCertificateAlias()** proporciona los **alias reales** con los que los certificados han sido dados de alta en los almacenes (que son siempre los que deben usarse con **setSelectedCertificateAlias(String)**), pero que el diálogo de selección de certificado solicita la selección en base a un "nombre descriptivo", que se compone a partir del Nombre Común (CN) del titular, el alias real si procede y el nombre de la entidad emisora. Este "nombre descriptivo" se usa porque en muchas ocasiones los alias reales no son realmente descriptivos o están en formatos poco prácticos (como X.500).

7.3.2 Filtros de certificados

El Cliente de firma incorpora una funcionalidad que permite hacer una preselección de los certificados que se muestran para selección al usuario, de forma que se puedan descartar a priori los no aceptados o no apropiados y así disminuir la probabilidad de que el usuario erre en la elección del certificado adecuado.

El establecimiento de los filtros se realiza mediante el método **setCertFilterRFC2254(String, String, Boolean)** que admite tres parámetros:

1. Filtro a aplicar en el campo *Principal* del titular del certificado X.509.
    - Debe proporcionarse una cadena de texto con una condición de filtro según la normativa RFC 2254.
2. Filtro a aplicar en el campo *Principal* del emisor del certificado X.509
    - Debe proporcionarse una cadena de texto con una condición de filtro según la normativa RFC 2254.
3. true si se desea que solo se muestren los certificados aptos para firma electrónica según el campo *KeyUsage* del certificado X.509. false si no se desea hacer distinción por el valor de este campo.

El paso de *null* en cualquiera de los parámetros indica que, por el criterio correspondiente, no se aplicará ningún filtro. Para más información, consulte la documentación JavaDoc.

Ejemplos de uso:

- Selección entre certificados de firma de DNIe:

```
clienteFirma.setCertFilterRFC2254(null, "cn=AC DNIE*", true);
```

- Selección entre cualquier certificado marcado como apto para firma electrónica:

```
clienteFirma.setCertFilterRFC2254(null, null, true);
```

- Selección únicamente entre certificados emitidos por Camerfirma:

```
clienteFirma.setCertFilterRFC2254(null, "o=Camerfirma", false);
```

- Selección con diversos criterios en un mismo Principal:

```
clienteFirma.setCertFilterRFC2254("(OU=Clase 2 persona fisica)(C=ES)",
  null, false);
```

- Selección entre certificados de un titular cuyo número de DNI sea "123456789Z" (funciona con la mayoría de los emisores de certificados, como DNIe, FNMT, etc.):

```
clienteFirma.setCertFilterRFC2254("SERIALNUMBER=123456789Z", null,
  false);
```

- Selección del certificado de firma del DNIe de un titular con número de DNI "123456789Z":

```
clienteFirma.setCertFilterRFC2254("SERIALNUMBER=123456789Z", "cn=AC
  DNIE*", true);
```

Es posible solicitar al cliente que, en caso de que el filtro obtenga un único certificado, este se seleccione automáticamente, sin dar al usuario la posibilidad de elegir. Esto lo haremos mediante el método **setMandatoryCertificateConditionRFC2254(String, String, Boolean)**.

Si el filtro devolviese más de un certificado, este método permitirá al usuario elegir entre los certificados obtenidos.

La marca de certificado apto para firma electrónica en el atributo KeyUsage de un certificado X.509

La inmensa mayoría de los certificados digitales usan el atributo X.509 *KeyUsage* para determinar el uso de un certificado (autenticación, firma electrónica, SSL servidor, etc.), por lo que distinguir por este para la selección del certificado apropiado para las operaciones de firma es en general una buena opción.

No obstante, la mayoría de los certificados emitidos por la FNMT-RCM (CERES, APE, etc.) no siguen las normativas internacionales en este sentido y en el atributo *KeyUsage* no marcan que son adecuados para firma electrónica pese a que se publicitan como aptos para dicho uso. Debido a esta falta de adecuación, si se marca mediante el último parámetro del método anteriormente comentado que solo deben mostrarse certificados aptos para firma, no se mostrará ningún certificado emitido por la FNMT-RCM.

Los certificados del DNIe sin embargo si siguen las normativas internacionales y marcan con los atributos correspondientes el uso, encontrándonos en cada DNIe un certificado apto para firma y otro que no lo es (el de autenticación).

7.4 Configuración de firma

7.4.1 Algoritmos de firma digital

El cliente permite usar distintos algoritmos de firma digital, siempre especificados con el formato A*with*B, donde A es el algoritmo de huella digital y B el de cifrado. Entre los algoritmos soportados encontramos:

- MD2withRSA
- MD5withRSA
- SHA1withRSA (por defecto)
- SHA256withRSA
- SHA384withRSA
- SHA512withRSA (es el más seguro)
- NONEwithRSA (no aplica huella digital, uso limitado)

El algoritmo a utilizar se puede cambiar con el método **setSignatureAlgorithm,** que recibe como parámetro una de las cadenas citadas.

NOTAS IMPORTANTES:

- No todas las operaciones soportan todos los algoritmos:
  - Los formatos de firma XAdES y XMLDSig solo soportan SHA1withRSA en versiones anteriores a Java 6 update 18.
- No todos los algoritmos pueden considerarse seguros:
  - MD2withRSA y MD5withRSA son algoritmos ofrecen un nivel de seguridad inferior al mínimo recomendado y algunos formatos de firma no los admiten.
- No todos los repositorios soportan todos los algoritmos:
  - SHA256withRSA, SHA384withRSA y SHA512withRSA no están soportados en la configuración tradicional del almacén de Windows.
- No todos los datos son aptos para cifrarse con cualquier algoritmo:
  - NONEwithRSA requiere que los datos tengan un tamaño y un formato específico.

- Los formatos de firma no genéricos (ODF, OOXML, PDF) ignorarán cualquier configuración especificada por el integrador que no esté soportada por su correspondiente normativa. Por ejemplo, modo de firma explícito, algoritmos de firma no soportados…
- Las firmas aplicadas con el algoritmo NONEwithRSA pueden considerarse como repudiables en interpretaciones estrictas de las buenas prácticas de firma electrónica.

7.4.2 Formato de firma electrónica

El cliente permite crear firmas digitales en distintos formatos (por defecto CMS).

Globalmente se soportan los siguientes formatos y normativas de firma electrónica:

- CMS: Representado por la cadena "**CMS/PKCS#7**".
- CAdES: Representado por la cadena "**CAdES**".
- XMLDSig Internally Detached: Representado por la cadena "**XMLDSig Detached**".
- XMLDSig Externally Detached (solo firmas, no multifirmas): Representado por la cadena "**XMLDSig Externally Detached**".
- XMLDSig Enveloping: Representado por la cadena "**XMLDSig Enveloping**".
- XMLDSig Enveloped: Representado por la cadena "**XMLDSig Enveloped**".
- XAdES Detached: Representado por la cadena "**XAdES Detached**".
- XAdES Enveloping: Representado por la cadena "**XAdES Enveloping**".
- XAdES Enveloped: Representado por la cadena "**XAdES Enveloped**".
- PDF (Portable Document Format): Representado por la cadena "**Adobe PDF**".
- ODF (Open Document Format): Representado por la cadena "**ODF**".
- OOXML (Office Open XML): Representado por la cadena "**OOXML**".

El formato se puede cambiar con el método **setSignatureFormat** que recibe como parámetro la cadena que representa al formato en cuestión.

Las variantes EPES de los formatos de firma que las soportan se generarán automáticamente al configurar el formato de firma correspondiente y una política de firma (consulte el apartado 7.4.4).

7.4.3 Modos de firma electrónica

Determinados formatos de firma electrónica soportan los llamados, modos de firma. El modo de firma determina si los datos firmados se incorporarán o no junto con la firma electrónica generada. Los modos de firma existentes son:

- Implícito: Representado por la cadena "**implicit**".
- Explícito: Representado por la cadena "**explicit**".

El modo de firma se puede cambiar con el método **setSignatureMode** que recibe como parámetro la cadena que representa al modo en cuestión. Los parámetros son insensibles a mayúsculas y minúsculas.

Los formatos soportados por el cliente @firma que admiten configuración de modo son:

- CMS/PKCS#7
- CAdES
- XMLdSig Detached
- XMLdSig Enveloping
- XAdES Detached
- XAdES Enveloping

Un formato de firma puede definir modos propios válidos para su configuración.

7.4.4 Política de Firma

El cliente permite especificar, para cada firma electrónica, la política a la que esta se restringe. Los formatos de firma, soportados por el cliente, que admiten políticas de firma son CAdES, PDF/PAdES y XAdES (en sus variantes Detached, Enveloping y Enveloped). En el momento de establecer en el cliente la política de firma para una firma CAdES, se generará una firma CAdES-EPES en lugar de la firma CAdES-BES tradicional. De igual manera, al establecer la política de firma para una firma XAdES o PAdES se generará una firma XAdES-EPES o PAdES-EPES, respectivamente.

Las firmas con política generadas por el cliente @firma son de referencia externa. Es decir, la política no se incluye en la propia firma, tan sólo una referencia a la misma.

Es posible establecer una política de firma en el cliente @firma mediante el método **setPolicy** que recibe como parámetros 3 cadenas:

- Identificador: URL en la que se encuentra publicada la política de firma.
- Descripción: Descripción breve de la política de firma.
- OID: Identificación de tipo OID de la política de firma.

7.5 Configuración de sobres digitales

7.5.1 Selección de destinatarios desde LDAP

Además de la posibilidad de seleccionar los destinatarios de un sobre digital a partir de sus certificados de clave pública almacenados en disco, el cliente @firma permite la configuración de un LDAP para seleccionar los certificados que este tenga publicados.

El procedimiento para la selección de estos certificados es la siguiente:

1. Configuración del servidor LDAP al que se desea acceder. Esto lo conseguimos mediante el método **setLdapConfiguration (String address, String port, String root)**. Este método recibe:
   - **address**: Dirección URL del LDAP.
   - **port**: Puerto a través del que se realiza la conexión. Si no se indica se usará el puerto 389, el por defecto para LDAP.
   - **root**: Dirección raíz del LDAP (actualmente sin uso).
2. Selección del certificado que se desea recuperar del LDAP. Para ello se utilizará el método **setLdapCertificatePrincipal**, que recibe como parámetro el *principal* del certificado que deseamos.
3. Recuperación del certificado en base 64 mediante el método **getLdapCertificate**.
4. Configuración del destinatario del sobre indicándolo mediante el método **addRecipientToCMS**, que recibe como parámetro el certificado en base 64 recuperado del LDAP. Pueden agregase más de un destinatario de esta manera. Una vez establecido un destinatario, puede eliminarse mediante el método **removeRecipientToCMS** al que se le pasa como parámetro el mismo certificado en base 64 con el que se estableció.

7.6 Configuración de cifrado

7.6.1 Algoritmos de cifrado

Los algoritmos de cifrado permitidos son los siguientes:

- Cifrado con clave
  - **AES** (por defecto)
  - **ARCFOUR**
  - **Blowfish**
  - **DES**
  - **DESede** (triple DES o 3DES)
  - **RC2**
- Cifrado con contraseña
  - **PBEWithSHA1AndDESede** (basado en DESede/3DES)
  - **PBEWithSHA1AndRC2_40** (basado en RC2)
  - **PBEWithMD5AndDES** (basado en DES)

Para establecer el algoritmo deberemos invocar la función **setCipherAlgorithm** y podemos recuperar el algoritmo actual con el método **getCipherAlgorithm**.

7.6.2 Modo de clave

Definen de qué manera se trata la clave de cifrado. Existen tres posibilidades **GENERATEKEY, USERINPUT** y **PASSWORD.**

- GENERATEKEY: La clave se generará automáticamente.
- USERINPUT: El usuario deberá establecer la clave en base 64.
- PASSWORD: La clave de usuario se generará a partir de una contraseña. Esto requiere el uso de algoritmos de cifrado diseñados con este objetivo (algoritmos PBE).

El modo de clave se establece mediante **setKeyMode** y se recupera con **getKeyMode**.

7.6.3 Clave y contraseña de cifrado

Para obtener la clave que se ha utilizado para el cifrado/descifrado deberemos ejecutar el método **`getKey`**, el cual nos devolverá la clave codificada en base 64. Para fijar una clave para el cifrado o descifrado de datos usaremos **`setKey`**, adjuntando como parámetro la clave deseada en base 64.

En el caso de haber especificado el modo de clave **`PASSWORD`** (consultar apartado 7.6.2), en lugar de una clave de cifrado será necesario especificar una contraseña de cifrado. Para establecer la contraseña de cifrado/descifrado se utilizará el método **`setPassord`**. Para recuperar la contraseña establecida se utilizará el método **`getPassword`**.

ADVERTENCIA: Las contraseñas de cifrado/descifrado no podrán contener caracteres no ASCII.

7.6.4 Almacén de claves de cifrado

El cliente @firma v3.1 y superiores permiten a los usuarios almacenar sus claves de cifrado en un almacén de claves protegido por contraseña.

Es posible configurar el cliente @firma para que, en el momento de autogenerar una clave de cifrado se ofrezca al usuario la posibilidad de almacenarla en su almacén personal de claves. Para esto será necesario configurar el Cliente en modo "GENERATEKEY" tal como se indica en el apartado 7.6.2.

En caso de que el usuario acepte almacenar la clave en su almacén, se comprobará que este ya exista. Si existía, se le solicitará al usuario la contraseña para abrirlo y el alias con el que desea almacenar la clave. Si no existía, se le indicará al usuario y se le dará la posibilidad de crearlo para lo que se le solicitará la contraseña con la que desea protegerlo. Tras crear el almacén se procederá a almacenar la clave tal como ya se indicó.

El integrador también puede permitir al usuario utilizar sus claves ya almacenadas en el almacén para cifrar nuevos datos. Para esto sólo sería necesario configurar el modo de clave del Cliente al valor "USERINPUT" (consultar apartado 7.6.2) y ejecutar la operación de cifrado.

Cuando se desee descifrar un contenido y no se haya indicado directamente la clave para el descifrado, se le preguntará al usuario si desea tomar la clave de su almacén personal. En caso de aceptar, se le pedirá la contraseña del almacén y se le dará a elegir mediante un diálogo modal entre las claves almacenadas (de las que se mostrará el alias asignado y el algoritmo de cifrado para el que fueron

generadas). Si no existiese el almacén de claves o el usuario no quisiera utilizarlo, se le preguntaría directamente por la clave de cifrado.

Cuando se activa el modo de clave para el cifrado/descifrado con contraseñas (modo "PASSW0RD" establecido según el aparatado 7.6.2) el almacén de claves queda inhabilitado.

Queda a elección del integrador la posibilidad de permitir que el usuario pueda o no almacenar la clave de cifrado en su almacén personal de claves o utilizar las almacenadas para cifrar. Esto puede hacerlo mediante el método **setUseCipherKeystore** al que se le puede pasar un `true` o un `false` para permitir o no su uso (por defecto se permitirá almacenarlas). Este método no afecta al descifrado de datos. Si no se indicase la clave para el descifrado y el usuario dispusiese de un almacén de claves, siempre se le dará la posibilidad de descifrar mediante una de las claves almacenadas.

A continuación se muestran algunos ejemplos para el uso del almacén de claves de cifrado:

1. Cifrado con las opciones por defecto (algoritmo de cifrado AES con una clave autogenerada) permitiendo que el usuario almacene la clave en su almacén:

```
…
clienteFirma.setFileuri("fichero_texto");
clienteFirma.cipherData();
var cipheredData = clienteFirma.getCipherData();
…
```

2. Cifrado con las opciones por defecto (algoritmo de cifrado AES con una clave autogenerada) NO permitiendo que el usuario almacene la clave en su almacén:

```
…
clienteFirma.setFileuri("fichero_texto");
clienteFirma.setUseCipherKeyStore(false);
clienteFirma.cipherData();
var cipheredData = clienteFirma.getCipherData();
…
```

3. Cifrado con una clave tomada del almacén del usuario (si no existiese se solicitaría directamente al usuario):

```
…
clienteFirma.setFileuri("fichero_texto");
clienteFirma.setKeyMode("USERINPUT");
clienteFirma.cipherData();
var cipheredData = clienteFirma.getCipherData();
…
```

4. Descifrado con una clave tomada del almacén del usuario:

```
…
clienteFirma.setFileuri("fichero_cifrado");
clienteFirma.decipherData();
var plainData = clienteFirma.getPlainData();
…
```

8 Otras funcionalidades

8.1 Guardar la firma en un fichero

El método **`saveSignToFile`** permite guardar la última firma generada en un fichero. Se puede especificar la ruta al fichero con **`setOutFilePath`**, que recibe una cadena con la ruta al fichero de salida. Si no se especifica, se permitirá elegir al usuario.

Si el fichero ya existe, se pide confirmación:

Figura 16: Diálogo para confirmar la sobreescritura de ficheros

8.2 Obtener el certificado usado para firmar

Es posible recuperar el certificado utilizado en la última operación de firma mediante el método **getSignCertificate**. Este método obtiene una instancia de la clase X509Certificate de Java.

El método **getSignCertificateBase64Encoded** devuelve una cadena de texto con el certificado, codificado en base 64, utilizado en para la última firma realizada. El certificado no estará delimitado por ninguna cadena ASCII ni carácter extra.

8.3 Leer el contenido de un fichero de texto

El método **getTextFileContent** que recibe como parámetro una URI a un fichero devuelve el contenido del mismo como una cadena. Si el fichero está almacenado en local, la URI comenzará por "file:///".

8.4 Leer el contenido de un fichero en Base64

El método **getFileBase64Encoded** que recibe dos parámetros (ruta al fichero y un booleano que indica si mostrar o no gráficamente al usuario el progreso en la lectura del fichero). En caso de producirse un error se devolverá **null**.

8.5 Convertir un texto plano a Base64

El método **getBase64FromText** recibe como parámetro un texto plano y lo codifica a base 64. En caso de producirse un error se devolverá **null**.

8.6 Obtener el hash de un fichero

El método **getFileHashBase64Encoded** devuelve una cadena con el hash de un fichero codificado en base 64. En caso de producirse un error se devolverá **null**.

8.7 Obtener la estructura de un CMS encriptado, envuelto o autenticado

Como método de diagnóstico podemos obtener la estructura de un CMS encriptado, envuelto o autenticado mediante la llamada a los métodos **formatEncryptedCMS** o **formatEnvelopedCMS** (ambos son compatibles con los distintos tipos de sobres), pasándole como parámetros la cadena en base 64 correspondiente al CMS del que queramos obtener su estructura.

8.8 Obtener la ruta de un fichero

Para permitir a un usuario obtener la ruta completa de un fichero el cliente dispone del método **loadFilePath(String, String, String).** Este método abre una ventana modal para la selección de un fichero. Es posible configurar el diálogo de selección a través de los parámetros que recibe la función y que son respectivamente:

- El título de la ventana de selección.
- El listado de extensiones permitidas separadas por "$%$".
- La descripción del fichero que se busca.

Todos los parámetros pueden ser nulos.

La salida de este método puede utilizarse para configurar la entrada del cliente mediante el método **setFileuri**.

ADVERTENCIA: Este método bloquea el script desde el que se ejecuta a la espera de que el usuario seleccione un fichero mediante el diálogo mostrado. Este comportamiento puede hacer que el navegador Mozilla Firefox 3.6 muestre al usuario una advertencia informando que el script está ocupado y puede ser dañino, dándole la posibilidad de bloquearlo. En caso de que se desee evitar esta interferencia, es responsabilidad del integrador ejecutar este método de forma asíncrona al resto del script (por ejemplo, mediante AJAX).

8.9 Obtener la ruta de un directorio

Para permitir a un usuario obtener la ruta completa de un directorio puede hacerse uso el método **selectDirectory**. Este método devuelve la ruta absoluta al directorio.

ADVERTENCIA: Este método bloquea el script desde el que se ejecuta a la espera de que el usuario seleccione un directorio mediante el diálogo mostrado. Este comportamiento puede hacer que el navegador Mozilla Firefox 3.6 muestre al usuario una advertencia informando que el script está ocupado y puede ser dañino, dándole la posibilidad de bloquearlo. En caso de que se desee evitar esta interferencia, **es responsabilidad del integrador ejecutar este método de forma asíncrona al resto del script**.

9 Ejemplos de uso

Junto al cliente se distribuyen los siguientes ficheros HTML de ejemplo de uso del cliente:

- ***demoInstalador.html:*** Demostración de las capacidades del *BootLoader*.
- ***demoMultifirma.html*:** Ejemplos de firma, co-firma y contra-firma
- ***demoMultifirmaMasiva.html***: Ejemplo de multifirma masiva programática.
- ***demoFirmaDirectorios.html***: Ejemplo de multifirma masiva sobre directorios.
- ***demoFirmaWeb/demoFirmaWeb-01.html***: Ejemplo de firma web
- ***demoCifrado.html:*** Ejemplo de cifrado.

- ***demoSobreDigital.html***: Ejemplo de CMS encriptado, CMS envuelto y CMS firmado y envuelto.
- ***demoKeyStores.html***: Ejemplo de la funcionalidad de cambio de almacén de certificados.
- ***demoLdap.html***: Ejemplo de la carga de certificados desde LDAP.

10 Buenas prácticas en la integración del cliente

10.1 Localizar el *Bootloader* y el directorio de instalables

Aunque la librería del cliente que facilita el uso del cliente (*instalador.js*) toma como dirección por defecto de los recursos del cliente la del HTML que lo carga, es muy recomendable el establecer estas direcciones explícitamente. En concreto, los parámetros a establecer se encuentran en el fichero *constantes.js* y son:

- **base**: Ruta del directorio en el que se encuentra el Applet de instalación y carga del cliente (*bootloader.jar*). Esta ruta debe apuntar al directorio en donde se encuentra este Applet, no al mismo. Por ejemplo, si la localización del Applet fuese "http://www.mpr.es/clienteAfirma/bootloader.jar" la dirección que se debería establecer sería "http://www.mpr.es/clienteAfirma".
- **baseDownloadUrl**: Ruta del directorio con los ficheros instalables del cliente (biblioteca de compatibilidad con Java 5, librerías nativas, fichero *version.properties* etc.). Esta ruta debe apuntar al directorio en donde se encuentran los instalables, no a un fichero concreto.

Las rutas indicadas pueden ser absolutas o relativas. Las rutas absolutas deben comenzar por "file:///" (nótese la triple barra), "http://" o "https://" (por ejemplo, "file:///C:/ficheros", "http://www.mpr.es/ficheros",...) y las rutas relativas no pueden empezar por "/" (por ejemplo, "afirma/ficheros"). Se debe usar siempre el separador "/", nunca "\".

La configuración de estas rutas, asegura la completa localización del cliente independientemente de la distribución de los HTML de la aplicación Web o de si estos se generan automáticamente. En este último caso sería necesario establecer la ruta absoluta de los directorios.

10.2 Indicar siempre la construcción mínima requerida del cliente

Aunque puede utilizarse el método JavaScript "`cargarAppletFirma()`" para asegurar que se tiene instalada y se carga el cliente @firma, es muy aconsejable el pasarle siempre como parámetro a este método la construcción mínima que requiera nuestra aplicación. Esto es utilizar los parámetros 'LITE' (por defecto), 'MEDIA' o 'COMPLETA'. Esto asegurará que, no sólo está instalado y se carga el cliente, también que la construcción instalada dispone de todas las capacidades requeridas por nuestra aplicación.

No es recomendable cargar siempre el cliente con el parámetro 'COMPLETA' salvo que se requiera de alguna de las capacidades exclusivas de esta construcción, ya que si no se necesitan éstas podemos estar obligando a los usuarios a descargarse una mayor cantidad de datos que no le reportarán beneficio alguno. Por este motivo se debe utilizar siempre como parámetro el identificador de la construcción mínima exigida.

10.3 Reducir las opciones de configuración

Siempre debe ofrecerse al usuario el menor número de opciones de configuración posibles sobre el proceso de firma o cualquier otra operación criptográfica. Son dos los aspectos que llevan a esta decisión:

- El cliente de firma comúnmente se integra en las aplicaciones Web para un fin determinado como puede ser el envío de un formulario Web firmado, por ejemplo, por lo que es el sistema de *backend* el receptor de los datos generados y el que finalmente debe almacenarlos y gestionarlos. En este caso, es lógico que sea el integrador el que decida la configuración y condiciones de la operación.
- La finalidad del usuario, no suele ser el propio uso del cliente, sino el acceso al servicio dado por la aplicación que lo integra. De esta forma, los usuarios no tienen porqué conocer detalles de las operaciones criptográficas que se realizan y ni siquiera conocimientos de los conceptos relacionados con la firma electrónica. En estos casos conviene simplificarle la tarea y no llevarle a dudar acerca de la opción más acertada para su fin concreto.

10.4 Configuración y uso del cliente en operaciones únicas

En el caso de que la ejecución de las operaciones del cliente dependan de una configuración introducida por el usuario o generada en tiempo de ejecución, es recomendable el realizar la configuración y ejecución de la operación criptográfica sin dar posibilidad de alterar el proceso. Por ejemplo, una forma de proceder sería el inicializar y configurar el cliente nada más cargarlo (filtro de certificados, datos obtenidos de una ventana anterior…) y establecer el resto de la configuración a medida que el usuario inserta los datos (formato de firma, datos a firmar, certificado de usuario,…) para, finalmente, sólo ejecutar la operación de firma. Este mecanismo tiene el inconveniente que cualquier interrupción en el cliente puede desechar toda esa información y terminar operando con una configuración por defecto en lugar de la indicada por el usuario.

En su lugar, es recomendable que, una vez se vaya a realizar la operación criptográfica, sea cuando se configure el cliente. Como ejemplo, en una implementación genérica JavaScript de invocación al cliente esto sería:

```
// Inicializamos la configuración para asegurar que no hay preestablecido
// ningún valor de operaciones anteriores
clienteFirma.initialize();

// Configuramos todos los parámetros del cliente, ya sea con datos directorios o
// extraídos de la página (formularios, contexto de la aplicación,…)
clienteFirma.setSignatureFormat("CADES");
clienteFirma.setSignatureAlgorithm("SHA1withRSA");
clienteFirma.setFileuri(document.getElementById("fichero").value);

// Ejecutamos la operación que corresponda
clienteFirma.sign();
```

Este modo de ejecución ayudará a evitar que, por ejemplo, el refrescar la página Web con F5 se pierda la sincronización con la configuración real del cliente con la que pueda verse en un momento determinado en la página Web. El uso de la tecla F5 o el botón “Refrescar Pantalla” debe evitarse siempre cuando nos encontremos a medias de un procedimiento online. En el caso de que el entorno de despliegue pueda detectarlo, incluso es recomendable que se obligue al usuario a reiniciar el procedimiento completo.

11 Funciones y métodos en la interfaz Applet del cliente @firma v3.x añadidos respecto a versiones anteriores

public String getCertificate(final String alias)

Obtiene el certificado X.509 correspondiente al alias proporcionado. El resultado es el certificado en Base64 delimitado por las cadenas ASCII -----BEGIN CERTIFICATE----- y -----END CERTIFICATE-----.

public String getCertificatePublicKey(final String alias)

Obtiene la clave pública del certificado X.509 correspondiente al alias proporcionado. El resultado es una clave RSA en Base64 delimitado por las cadenas ASCII -----BEGIN RSA PUBLIC KEY----- y -----END RSA PUBLIC KEY-----.

public String getCertificates()

Obtiene todos los certificados del almacén actual en una única cadena en donde los elementos se dividen mediante el separador STRING_SEPARATOR definido como constante en el cliente. El formato individual de los certificados es el mismo que el devuelto por el método public String getCertificate(final String alias). También es posible obtener de forma segura un array con los certificados mediante el método JavaScript getCertificates() definido en "firma.js".

public String[] getArrayCertificates()

Obtiene todos los certificados del almacén actual en un array unidimensional, con el mismo formato individual que el devuelto por el método public String getCertificate(final String alias). **El uso de este método no está recomendado** debido a la incompatibilidad existente entre el formato de array de Java 5 y el motor JavaScript de Microsoft Internet Explorer.

public String getCertificatesAlias()

Se ha considerado útil que el integrador, vía JavaScript, pueda obtener los alias del almacén de certificados utilizado por el navegador Web activo. Este método obtiene los alias de los certificados en una única cadena separándolos mediante la constante STRING_SEPARATOR definida en el cliente. También es posible obtener de forma segura un array con los alias de los certificados mediante el método JavaScript getCertificatesAlias() definido en "firma.js". Para más información, consulte la documentación en formato JavaDoc.

public String[] getArrayCertificatesAlias()

Se ha considerado útil que el integrador, vía JavaScript, pueda obtener los alias del almacén de certificados utilizado por el navegador Web activo. Para más información, consulte la información en formato JavaDoc. **El uso de este método no está recomendado** debido a la incompatibilidad existente entre el formato de array de Java 5 y el motor JavaScript de Microsoft Internet Explorer.

public void setSelectedCertificateAlias(String certAlias)

Como complemento al método anterior, se ha considerado útil que el integrador, vía JavaScript, pueda establecer el alias del certificado a utilizar por el Applet en el navegador Web activo. Para más información, consulte la información en formato JavaDoc.

public boolean signDirectory()

Para las funciones de firma masiva, firma todos los archivos de un directorio según la configuración establecida. Para más información, consulte la información en formato JavaDoc.

public void setMassiveOperation(String massiveOperation)

Para las funciones de firma masiva, establece la operación masiva a realizar en el proceso generado por el método signDirectory(). Para más información, consulte la información en formato JavaDoc.

public void setOriginalFormat(boolean originalFormat)

Para las funciones de firma masiva, indica si se debe respetar el formato de firma original para las operaciones de multifirma masiva. Para más información, consulte la información en formato JavaDoc.

public void setOriginalFormat(boolean originalFormat)

Para las funciones de firma masiva, indica si se debe respetar el formato de firma original para las operaciones de multifirma masiva o, si en cambio, se usará la configuración de firma establecida para todas las firmas. Para más información, consulte la información en formato JavaDoc.

public String getInputDirectoryToSign()

Para las funciones de firma masiva, devuelve la ruta absoluta del directorio donde se ubican los ficheros a ser firmados de forma masiva. Para más información, consulte la información en formato JavaDoc.

public void setInputDirectoryToSign(String directory)

Para las funciones de firma masiva, establece el directorio de donde se tomarán los ficheros de firma y datos para la operación de firma masiva. Para más información, consulte la información en formato JavaDoc.

public String getOutputDirectoryToSign()

Para las funciones de firma masiva, devuelve la ruta absoluta del directorio donde se almacenarán las firmas resultado de la operación de firma masiva. Para más información, consulte la información en formato JavaDoc.

public void setOutputDirectoryToSign(String directory)

Para las funciones de firma masiva, establece el directorio donde se depositarán las firmas masivas de los archivos situados en InputDirectoryToSign. Para más información, consulte la información en formato JavaDoc.

public void setInIncludeExtensions(String extensions)

Para las funciones de firma masiva, define las extensiones que se incluirán en la firma de directorios. Para más información, consulte la información en formato JavaDoc.

public void setInRecursiveDirectorySign(boolean recursiveSignDir)

Para las funciones de firma masiva, establece si la firma de directorios se efectuará de forma recursiva o no. Para más información, consulte la información en formato JavaDoc.

public void setFileuriBase64(String uri)

Establece los datos contenidos en el fichero indicado (en donde se encontrarán codificados en base 64), como los datos de entrada para las operaciones criptográficas y establece la ruta introducida como ruta de entrada.

El contenido del fichero se interpretará siempre como datos en base 64 no realizándose la comprobación previa de los mismos.

public String loadFilePath(String title, String exts, String description)

Muestra un diálogo modal para la selección de un fichero del que se recuperará su ruta completa. Para más información, consulte la información en formato JavaDoc.

12 Casos problemáticos de despliegue e integración del cliente

12.1 Despliegue del cliente en servidores Web que requieren identificación de los usuarios mediante certificado cliente

12.1.1 Applets de Java y Autenticación con Certificado Cliente

Durante la instalación del Applet en el sistema local del usuario, el Applet BootLoader establece varias conexiones con el servidor Web para descargarse los ficheros necesarios.

En los servidores en los que se requiere al cliente que se identifique mediante un certificado (autenticación por certificado cliente) cuando solicita datos o una conexión, los Applets Java no son una excepción, por lo que el BootLoader debe identificase de forma independiente del navegador Web, ya que las conexiones que establece son independientes de este.

El Plugin de Java contempla esta posibilidad, y gestiona la autenticación por certificado cliente de forma independiente de los Applets que ejecute, por lo que estos no deben implementar ningún cambio para adaptarse a estos entornos, siendo todo el proceso completamente transparente para ellos.

El almacén que usa el Plugin de Java para seleccionar el certificado en muestra al servidor Web varía según la configuración cliente, pero sigue en todos los casos el mismo proceso:

1. Intenta acceder primero al almacén nativo del navegador Web (MS-CAPI, Apple KeyRing o Mozilla/Firefox NSS).
2. Si por problemas de configuración el almacén nativo no pudiese ser accedido por el Plugin de Java, se selecciona el almacén propio del JRE.
3. Se pide al usuario que seleccione uno de los certificados del almacén finalmente seleccionado, que será el que se envíe al servidor Web.

La clasificación de los navegadores Web por almacén utilizado para los certificados y por sistema operativo es la siguiente:

Almacén de certificados MS-CAPI
- Windows
  - Internet Explorer
  - Google Chrome
  - Apple Safari
  - Opera

Almacén de certificados Apple KeyRing
- Mac OS X
  - Apple Safari
  - Google Chrome
  - Opera
  - Internet Explorer

Almacén de certificados Mozilla / Firefox (NSS)
- Windows
  - Mozilla / Firefox
- Linux / Solaris
  - Mozilla / Firefox
  - Google Chrome
  - Opera

A continuación, detallamos la configuración adicional necesaria en cada uno de los casos para el correcto funcionamiento en servidores Web que soliciten certificado cliente. Esta configuración no es específica para el Cliente @firma, sino que será necesaria para cualquier otro Applet de Java que establezca conexiones independientes del navegador con el servidor Web:

MS-CAPI

No es necesario ningún proceso adicional de configuración. El entorno de ejecución de Java, desde su versión 1.5 accede directamente al almacén CAPI.

Apple KeyRing

No es necesario ningún proceso adicional de configuración. Los entornos de ejecución de Java distribuidos por Apple Computer para su sistema operativo Mac OS X acceden directamente al almacén Apple KeyRing.

Mozilla / Firefox (NSS)

Es necesario instalar previamente en el entorno de ejecución de Java las bibliotecas JSS (Netscape Java Security Services), atendiendo a las siguientes precauciones:

- Ha de seguirse el proceso de instalación exactamente como se describe en la documentación de Java: http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/keystores.html.
  - Las instrucciones publicadas por Sun Microsystems están desactualizadas y no aplican para las últimas versiones de Mozilla Firefox, para las cuales deben seguirse los siguientes pasos:
    1. Copiar el fichero ***jss4.jar*** al directorio de extensiones del entorno de ejecución de Java (JRE) en uso:
       - %JAVA_HOME%\lib\ext en sistemas Windows
       - $JAVA_HOME/lib/ext en sistemas basados en UNIX (Linux, Solaris, Mac OS X)
    2. Copiar el fichero la biblioteca nativa de JSS en el directorio principal de bibliotecas del sistema operativo. Es necesario cerciorarse de que la biblioteca que copiemos corresponda con la versión y arquitectura de nuestro sistema operativo:
       - En sistemas Windows, el fichero ***jss4.dll*** debe copiarse al directorio %SystemRoot%\system32
       - En sistemas Linux y Solaris, el fichero ***jss4.so*** debe copiarse al directorio /lib o al directorio /usr/lib
       - En sistemas Mac OS X, el fichero ***jss4.dylib*** o el fichero ***jss4.so*** (dependiendo de la compilación utilizada) debe copiarse al directorio /lib o al directorio /usr/lib

- Hemos también de asegurarnos de que la versión instalada de JSS sea compatible con nuestra versión de Mozilla / Firefox. Consulte atentamente la documentación de los productos antes de proceder a instalarlos.
  - Firefox 3 solo es completamente compatible con JSS 4.3.2 y superiores: https://developer.mozilla.org/En/JSS/4_3_ReleaseNotes, pero aun presenta ciertos problemas de compatibilidad en sistemas Windows. En cualquier caso, si encuentra dificultades, pruebe a instalar siempre la versión más actualizada.
- JSS puede descargarse de forma libre desde: ftp://ftp.mozilla.org/pub/mozilla.org/security/jss/releases/
  - No obstante, para ciertas versiones de JSS, la Comunidad Mozilla no distribuye binarios, sino únicamente el código fuente.

En el caso concreto de que se esté ejecutando el cliente @firma sobre un sistema con la configuración Mozilla Firefox / Java 5, nos encontramos el problema de Java 5 no es compatible con el modelo de seguridad que implementan las últimas versiones de Mozilla. Ya que Mozilla Firefox 3.6 y superiores requieren de Java 6 update 10, se recomienda la actualización a la última versión de Java 6.

Almacén propio de Java

Como se ha comentado anteriormente, cuando el Plugin del entorno de ejecución de Java (JRE) no puede acceder al almacén del navegador Web, solicita al usuario la selección de un certificado de su propio almacén.

El principal problema de esta opción es que JRE no accede a los módulos PKCS#11 o CSP del sistema, por lo que los certificados residentes en tarjetas inteligentes (incluido DNIe) o dispositivos externos (USB, etc.) no son accesibles.

Para comprobar los certificados existentes en el almacén de Java e importar certificados en él, se pueden seguir los siguientes pasos (sistemas operativos Windows):

1. Abrir la opción de "Java" en el Panel de Control y seleccionar la pestaña "Seguridad":

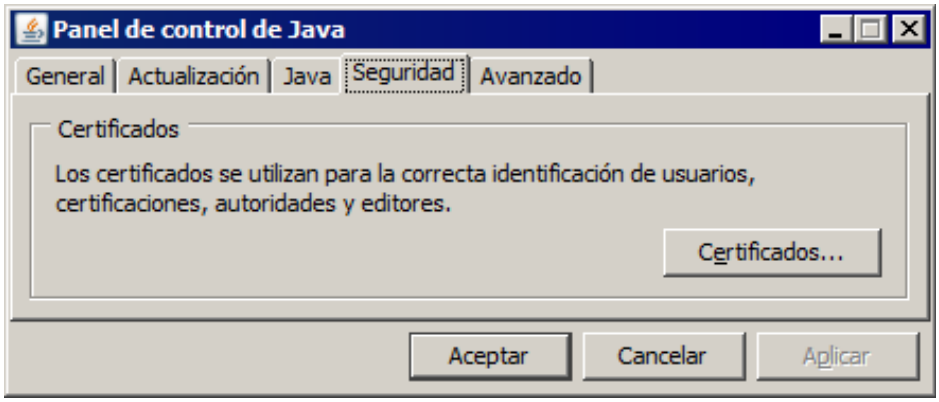


Figura 17: Panel de control de Java

2. Pulsar el botón "Certificados" y seleccionar el tipo de certificado como "Autenticación de cliente", dentro de la pestaña "Usuario".

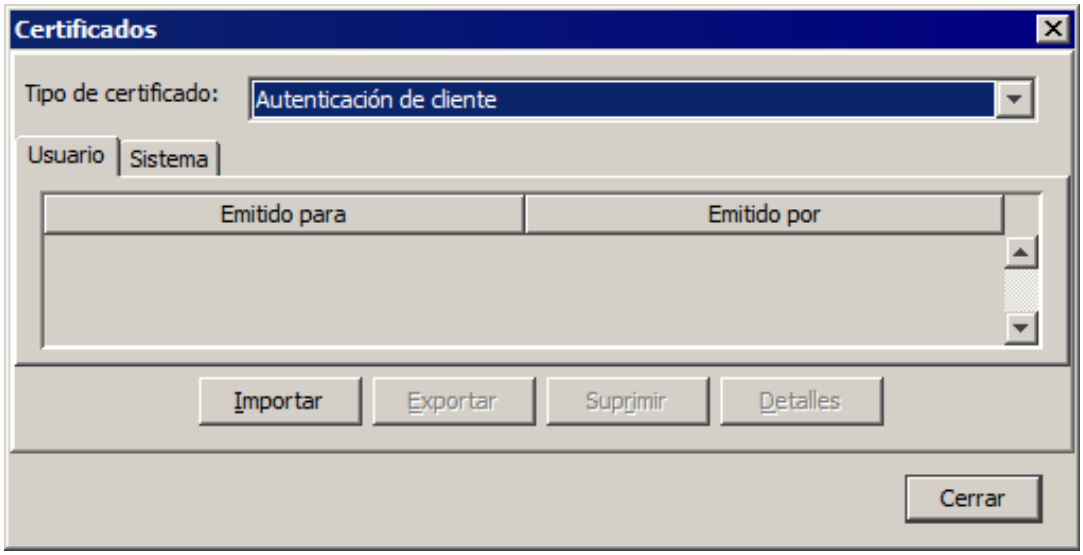


Figura 18: Certificados del almacén de Java

3. Los certificados mostrados en la lista son los disponibles por el Plugin de Java para autenticar a un Applet ante un servidor Web que requiere certificado cliente. Podemos importar nuevos certificados usando el botón "Importar".

Para realizar las mismas comprobaciones en otros sistemas operativos, consulte la documentación del entorno de Java instalado (JRE) y del Java Plugin.

12.1.2 Alternativa de despliegue

Una variante sobre lo arriba expuesto es el no publicar el cliente @firma al completo en un servidor con autenticación con certificado cliente, tan sólo las páginas Web que dan acceso al mismo. Es posible situar, tanto el applet de carga como los ficheros instalables en un segundo servidor (o una ruta del mismo configurada para no pedir certificados). Con esto obtenemos que:

- Páginas Web: En servidor con conexión SSL y autenticación con certificado cliente.
- Binarios Java: En servidor con conexión SSL o sin ella, según se desee.

El principal beneficio de esta alternativa está claro, no es necesario que los binarios java se autentiquen contra el servidor, por lo que no es necesario que el usuario configure ningún certificado en el repositorio de Java.

Para configurar este entorno basta con configurar las siguientes variables del fichero de despliegue "*constantes.js*":

- **base:** Esta variable apunta al directorio en el que se sitúa el applet de carga (bootloader.jar). Deberá establecerse para que apunte a la ruta en la que se sitúa este fichero (en un servidor que no requiere autenticación cliente). Por ejemplo, si la ruta del fichero es "https://www.mpr.es/afirma/bootloader.jar" el valor de la variable deberá ser "https://www.mpr.es/afirma". También es posible establecer una ruta relativa hasta el directorio del aplicativo.
- **baseDownloadURL:** Esta variable apunta al directorio en el que se encuentran los ficheros necesarios para la instalación del cliente. Deberá establecerse para que apunte a la ruta en la que se sitúan estos (en un servidor o un directorio en donde no se requiera autenticación cliente). Por ejemplo, "https://www.mpr.es/afirma/instalables" o "../instalables".

Los ficheros que deberán situarse en el directorio de instalables son:

- version.properties
- mscapi.zip
- sunmscapi.jar
- msvcr71.zip
- xalan.zip

- afirma_5_java_5.jar.pack.gz
- COMPLETA_afirma.jnlp
- COMPLETA_j6_afirma5_coreV3.jar
- COMPLETA_j6_afirma5_coreV3.jar.pack.gz
- MEDIA_afirma.jnlp
- MEDIA_j6_afirma5_coreV3.jar
- MEDIA_j6_afirma5_coreV3.jar.pack.gz
- LITE_afirma.jnlp
- LITE_j6_afirma5_coreV3.jar
- LITE_j6_afirma5_coreV3.jar.pack.gz
- COMPLETA_j5_afirma5_coreV3.jar.pack.gz
- MEDIA_j5_afirma5_coreV3.jar.pack.gz
- LITE_j5_afirma5_coreV3.jar.pack.gz

12.2 Problema con el objeto HTML File en los nuevos navegadores

La nueva generación de navegadores Web (Internet Explorer 8, Firefox 3,…) ha restringido el comportamiento del objeto *File* de HTML por motivos de seguridad.

La finalidad de este componente es únicamente el permitir la carga de ficheros a un servidor. Sin embargo, antes se le permitía obtener a este servidor excesiva información sobre el sistema del usuario ya que, determinados navegadores, mediante JavaScript, proporcionaban la ruta completa en la que estaba almacenado el fichero.

Los nuevos navegadores no permiten obtener más que el nombre del fichero o, a lo sumo, este y una ruta genérica y no descriptiva.

Esto podría inhabilitar la práctica que han seguido muchos integradores del cliente @firma, que utilizaban este componente para, además de la carga del fichero, obtener la ruta del mismo para así utilizar el método setFileuri() y operar con el fichero.

Con objeto de solventar en parte este problema, se ha incluido en el applet cliente el método loadFilePath() que muestra al usuario un diálogo para la selección de un fichero de datos y devuelve la ruta completa de ese fichero. Con esto el usuario es libre de usar la ruta con el método setFileuri() para realizar la firma del fichero y/o mostrársela al usuario en un cuadro de texto, por ejemplo. Puede encontrarse la descripción completa del método en el Javadoc del cliente @firma.

Sin embargo, al igual que no permite obtener la ruta del fichero seleccionado, el objeto *File* no permite establecer de forma externa una ruta de fichero para su posterior subida al servidor, ya que esto posibilitaría a cualquier página Web maliciosa a obtener ficheros del disco duro del usuario sin su consentimiento. Esto imposibilita el tomar la ruta del fichero mediante el método loadFilePath() para luego cargarlo con un objeto *File*.

La solución a este problema puede llevarse a cabo mediante el uso de diferentes mecanismos, cada cual ajustado al entorno, el fin y las tecnologías con las que cuente el integrador del sistema.

Una solución simple sería, por ejemplo, el leer el fichero mediante el método getFileBase64Encoded() y anexar la cadena en base 64 resultante como campo oculto al formulario del usuario (configurar con el método POST). La ruta del fichero se habrá obtenido previamente con la ayuda del método loadFilePath() y especificado al cliente mediante setFileuri().

ADVERTENCIA: Si su sistema requiere o permite que el usuario envíe ficheros mayores de 4 megas de tamaño, consulte el apartado 12.3.

12.3 Procedimiento de carga para ficheros mayores de 4MB

Al ejecutar el Cliente @firma en un entorno con Java 6u10 o superior y el plugin de próxima generación activado (configuración por defecto), nos encontramos con que no es posible convertir ficheros de datos mayores de 4MB a cadenas Base 64. Esta operación es necesaria para posteriormente adjuntar los datos firmados (o la firma implícita generada) al formulario Web a través del cual se enviará la información al servidor. Esta limitación también puede afectar a la generación de firmas XML implícitas de ficheros mayores de 4MB.

Este problema no tiene solución en la versión 3.1 del Cliente @firma pero es posible llevar a cabo algunas prácticas con la que es posible evitarlo.

1. Evalúe si es necesario que su sistema firme los ficheros adjuntos a una transacción o si basta con firmar la propia transacción. Esto podría hacerse mediante un XML en el que se almacenen los

datos de la transacción (identificador, los datos proporcionados por el usuario, nombre de los ficheros adjuntos y su hash,…).

2. Si su sistema realiza firmas de ficheros seleccionados por el usuario y se van a admitir ficheros mayores de 4MB, evalúe el uso de firmas binarias (CAdES) en lugar de firmas XML (XAdES). El problema comentado puede afectar a la generación de firmas XML (XMLdSig / XAdES) de ficheros binarios mayores de 4MB.

3. Si es necesario el envío de ficheros mayores de 4MB al servidor, deberán enviarse mediante el componente File de los formularios HTML. Para esto, tendremos que firmar previamente los datos y obligar a que sea el propio usuario quien seleccione los ficheros de firma generados. Se propone el siguiente modelo de aplicación Web:

    a. Mostrar al usuario el formulario Web con la información que debe rellenar. Esto puede hacerse en una única página Web o en varias si la cantidad de datos lo requiere.

    b. En el punto que corresponda del formulario, se dará la opción al usuario de seleccionar los ficheros que desea adjuntar al mismo. Esto abrirá una nueva ventana en donde se cargará el Cliente @firma y, mediante el método descrito en el apartado 12.2, se dará al usuario la posibilidad de firmar los ficheros. En este caso, en lugar de adjuntar el resultado de la firma al formulario Web, se le permitirá almacenarla en disco, notificándole que esta es la firma electrónica generada que posteriormente se deberá adjuntar al formulario y que, si lo desea, puede conservar como parte del resguardo de la transacción. En este paso se pueden firmar tantos ficheros como se deseen.

    c. De vuelta al formulario principal y al final del mismo se mostrará un botón Aceptar que redirigirá al usuario a una nueva página con el resumen de los datos del formulario para que confirme que son válidos. En esta página se cargará de nuevo el Cliente @firma y se le permitirá al usuario cargar, mediante componentes File de HTML, los ficheros de firma (y de datos en caso de firmas explícitas) que se generaron en el paso anterior.

    d. Tras revisar los datos y seleccionar los ficheros necesarios, el usuario podrá enviar el formulario para finalizar el trámite. Al pulsar el botón Enviar, se firmará la transacción con el Cliente @firma y seguidamente se enviará el formulario.

NOTA: En sistemas con Java 5 la limitación del tamaño de fichero está en torno a los 100MB y en sistemas con Java 6 y el plugin de próxima generación desactivado en torno a los 50MB. Sin embargo, no debe presuponerse que el usuario operará desde alguno de estos entornos.

13 Siglas

CAdES	*CMS Advanced Electronic Signature*
CMS	*Cryptographic Message Standard*
CSP	*Cryptographic Service Provider* (proveedor de servicios criptográficos)
DNI-e	DNI electrónico
JAR	*Java Archive*
JNLP	*Java Networking Launching Protocol*
JRE	*Java Runtime Environment*
MPR	Ministerio de la Presidencia
PDF	*Portable Document Format*
PKCS#1	*Public Key Cryptography Standard number 1* (estándar de criptografía de clave pública nº 1)
PKCS#7	*Public Key Cryptography Standard number 7* (estándar de criptografía de clave pública nº 7)
PKCS#11	*Public Key Cryptography Standard number 11* (estándar de criptografía de clave pública nº 11)
PKCS#12	*Public Key Cryptography Standard number 12* (estándar de criptografía de clave pública nº 12)
PKI	*Public Key Infrastructure*
SHA	*Secure Hash Algorithm*
URI	*Uniform Resource Identifier* (Identificador Uniforme de Recursos)
URL	*Uniform Resource Locator* (Localizador Uniforme de Recursos)
WYSIWYS	*What You See Is What You Sign* (lo que ves es lo que firmas)
XAdES	*XML Advanced Electronic Signature* (firma electrónica avanzada XML)
XML	*eXtensible Markup Language* (Lenguaje de marcas extensible)
XMLDSig	*XML Digital Signature* (firma digital XML)

14 Documentos de Referencia

[JAVADOC]

Documentación de los métodos públicos de los Applets Bootloader y de Firma en la carpeta `javadoc`.

15 Información de contacto

Soporte a la Administración Electrónica

Consejería de Hacienda y Administración Pública

Dirección General de Tecnologías para Hacienda y la Administración Electrónica

soporte.admonelectronica@juntadeandalucia.es

16 Anexo A. Formatos de firma binaria genérica soportados por el cliente

16.1 Matriz de formatos soportados en formatos binarios (PKCS#1, CMS y CAdES)

		Firma Digital		Sobre Digital				
		Implícito	Explícito	Cifrado	Envuelto	Envuelto y Firmado	Autenticado	Autenticado y Envuelto
CMS	Firma							
	Cofirma							
	Contrafirma							
CAdES	Firma							
	Cofirma							
	Contrafirma							
PKCS#1	Firma							

Leyenda:		
		Soportado y acorde a estándar
		Soportado pero con problemas de adecuación a estándar
		No soportado
		No contemplado en el estándar

Adicionalmente, deben observarse las siguientes aclaraciones sobre los formatos:

- Las firmas CMS generadas son compatibles PKCS#7
- Las firmas sin formato (PKCS#1) generadas son compatibles con PKCS#1 1.5, y en versiones anteriores del cliente de firma se denominaban firmas "NONE".
- Las firmas CAdES generadas son compatibles con la especificación CAdES-BES o CAdES-EPES.

16.2 Algoritmos de huella digital

El cliente de firma soporta (con las salvedades indicadas en las notas posteriores) la aplicación de los siguientes algoritmos de huella digital para las firmas binarias:

- MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, NONE (sin huella digital).

NOTAS IMPORTANTES SOBRE LOS ALGORITMOS DE HUELLA DIGITAL:

- NONE
    - Ciertos dispositivos criptográficos (tarjetas inteligentes, dispositivos USB, almacenes software, etc.) no soportan la aplicación de firmas digitales si no mantienen de forma interna el control de la huella digital (por motivos de seguridad), por lo que **no se garantiza el correcto funcionamiento de la firma NONE en estos casos**. Es importante recalcar que **la firma NONE es necesaria en la firma en más de una fase**, ya que en estos casos la huella digital se calcula en origen, por lo que no se debe repetir en destino.

NOTAS IMPORTANTES SOBRE LOS SOBRES DIGITALES:

Ciertos dispositivos de seguridad, incluido el DNIe y las tarjetas CERES de la FNMT-RCM no soportan operaciones de descifrado RSA usando clave privada, por lo que son incapaces de abrir sobres digitales. Debe informarse convenientemente a los usuarios para que no traten de enviar sobres digitales usando claves públicas de DNIe o de certificados CERES cuya clave privada resida en una tarjeta FNMT-RCM CERES. Esta limitación afecta a todas las plataformas (implementada tanto en el controlador MS-CAPI/CSP como en el PKCS#11).

16.3 Notas específicas para la plataforma MS-Windows

Debido al *bug* de Java anteriormente mencionado (http://bugs.sun.com/view_bug.do?bug_id=6578658**), no es posible realizar firmas en formato PKCS#1 v1.5 (firma NONE) en varias fases**, es decir, separando la generación de la huella digital (*message digest o* hash) y el cifrado RSA con clave privada.

Es resto de las funcionalidades son plenamente compatibles con todos los sistemas operativos y plataformas soportadas (Windows, Linux, Solaris, Mac OS X), siempre atendiendo a los almacenes de claves compatibles con cada sistema operativo.

No hay variaciones de compatibilidad en este caso por versión del entorno de ejecución de Java.

16.4 Uso de los parámetros de funcionamiento

El modo de uso del cliente para establecer los parámetros de funcionamiento del Cliente consiste en realizar llamadas a ciertos métodos del Applet indicando cadenas de texto que identifican los valores que queremos establecer.

En paticular, se indican los formatos y sub-formatos (modos) de firma mediante unas cadenas de texto específicas. Además, cada formato o sub-formato introducido va asociado a una extensión de fichero (que define su tipo), extensiones que se usan como filtro de entrada o de salida a la hora de abrir o salvar archivos.

Las cadenas de identificación de formato deben usarse con la llamada JavaScript:

```
clienteFirma.setSignatureFormat(String format)
```

Y las cadenas de sub-formato con la llamada JavaScript:

```
clienteFirma.setSignatureMode(String mode)
```

Ambas funciones están documentadas en el JavaDoc del Applet, Remítase a estos documentos para más información. El orden de llamada de ambos métodos no es significativo.

16.5 Parámetros de funcionamiento

- Cadenas (se ignoran las diferencias entre mayúsculas y minúsculas) de identificación de formato (varias alternativas por cada uno de ellos, por flexibilidad de uso, se muestran separadas por "/"). **Aplica únicamente a firma digital, y no a sobres digitales**:
  - CMS
    - `"CMS" / "PKCS7" / "PKCS#7"`
  - CAdES
    - `"CAdES" / "CAdES-BES"`
  - PKCS#1
    - `"NONE" / "RAW" / "PKCS1" / "PKCS#1"`
- Cadenas de identificación del modo de firma (insensibles a mayúsculas/minúsculas). **Aplica únicamente a firma digital, y no a sobres digitales**:
  - Firma Explícita
    - `"Explicit"`

- Firma Implícita
  - `"Implicit"`
- Ficheros de entrada (todos: CMS y CAdES, Implícitas y Explícitas y PKCS#1)
  - Binarios (*.*)
- Ficheros de salida
  - CMS y CAdES
    - Ficheros de firma ASN.1 (*.csig)
- PKCS#1
  - Binarios (*.sig)

16.6 Cofirmas cruzadas entre CMS y CadES

Las cofirmas de un documento dan como resultado dos firmas sobre este mismo documento que se encuentran a un mismo nivel, es decir, que ninguna envuelve a la otra ni una prevalece sobre la otra.

A nivel de formato interno, esto quiere decir que cuando cofirmamos un documento ya firmado previamente, esta firma previa no se modifica. Si tenemos en cuenta que CAdES es en realidad un subconjunto de CMS, el resultado de una cofirma CAdES sobre un documento firmado previamente con CMS (o viceversa), son dos firmas independientes, una en CAdES y otra en CMS. Dado que todas las firmas CAdES son CMS pero no todas las firmas CMS son CAdES, el resultado global de la firma se adecúa al estándar más amplio, CMS en este caso.

Otro efecto de compatibilidad de formatos de las cofirmas con varios formatos de un único documento es la ruptura de la compatibilidad con PKCS#7, ya que, aunque las firmas generadas por el cliente mediante CMS son compatibles con PKCS#7, las generadas con CAdES no lo son, por lo que, en el momento que se introduzca una estructura CAdES, se perderá la compatibilidad PKCS#7 en el global de la firma.

16.7 Formato CMS de Firma Digital

Al igual que otros elementos CMS que describiremos posteriormente, la estructura de una firma CMS viene definida en la RFC 3852, aunque en este caso en especial, y para mantener la compatibilidad con PKCS#7, el elemento *signedData* especificado en la RFC indicada se encuentra limitado. La estructura empleada por el cliente de firma es la siguiente:

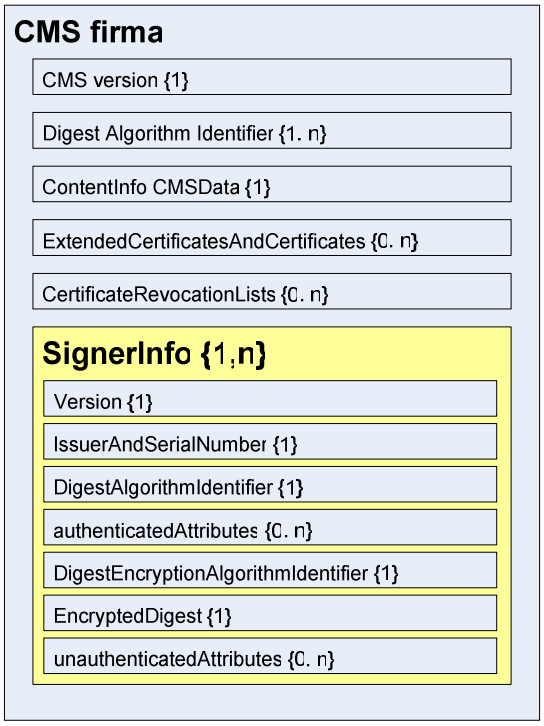


Figura 19: Estructura PKCS#7 SignedData

- **CMS Version**. Indica las diferentes versiones del mensaje. Para que la compatibilidad con PKCS#7 se mantenga debe ser 0.
- **Digest Algorithm Identifier**. Identifica el algoritmo utilizado.
- **ContentInfo**. Secuencia de parámetros que identifican el contenido del mensaje. Comprende el tipo de contenido (*contentType*) que en nuestro caso será *Data* y *content* que se refiere a la secuencia de bytes correspondiente a los datos mismos.
- **Extended Certificates And Certificates**. Opcional. Permite especificar una cadena de certificados para la validación de los distintos certificados firmantes.
- **Certificate revocation Lists**. Opcional. Permite especificar las CRL para los certificados utilizados.
- **Signer Info**. Estructura que especifica la información de los diferentes firmantes del contenido del mensaje. Se subdividen en los siguientes campos:
  - **Versión**. Especifica la versión de esta estructura y será siempre 1 para PKCS#7.
  - **Issuer And Serial Number**. Especifica el certificado usado mediante el emisor y número de serie de éste.
  - **Digest Algorithm Identifier**. Identifica el algoritmo utilizado.
  - **Authenticated Attributes**. Opcional. Secuencia de atributos firmados que especifican ciertos parámetros importantes para la interpretación del contenido. Si

el tipo del contenido fuese distinto de *Data*, sería obligatorio incorporar como atributos el tipo empleado y el hash del contenido, pero en nuestro caso esto no es posible ya que siempre tendremos el *ContentType Data*.

- **Digest Encryption Algorithm**. Describe que algoritmo se ha usado en la encriptación de la firma y resumen del documento.
- **Encrypted Digest**. Hash del mensaje encriptado empleando la clave privada del certificado y el algoritmo especificado antes
- **Unauthenticated Attributes**. Opcional. Atributos no firmados definidos en PKCS#9, como por ejemplo las contrafirmas.

16.8 Formato de sobre digital CMS encriptado

La estructura vendría definida como sigue:

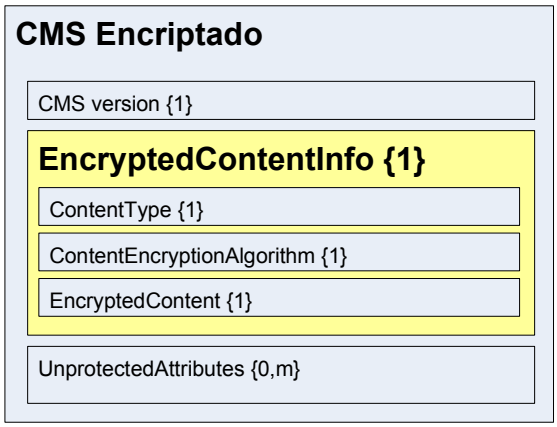


Figura 20: Estructura PKCS#7 EncryptedData

- **CMS Version**. Será 0 si no existen ***UnprotectedAtributes*** o 2 en caso contrario.
- **Encrypted Content Info**. Subestructura que se define mediante los siguientes campos:
  - **Content Type**. Define el tipo de contenido.
  - **Content Encryption Algorithm**. Define el algoritmo utilizado para encriptar el contenido.
  - **Encrypted Content**. Contenido encriptado usando el algoritmo especificado anteriormente.
- **Unprotected Attributes**. Opcional. Secuencia de parámetros auxiliares definidos por otros estándares.

Como se puede apreciar, esta estructura no contiene la clave de cifrado ni ningún método de transmisión de esta, por lo que si se usa como mensaje se debe buscar un método para compartir una clave privada.

16.9 Formato de Sobre Digital CMS Envuelto

Esta estructura se identifica con el sobre digital identificado en la RFC 3852 como **Enveloped CMS** y sigue la siguiente estructura:

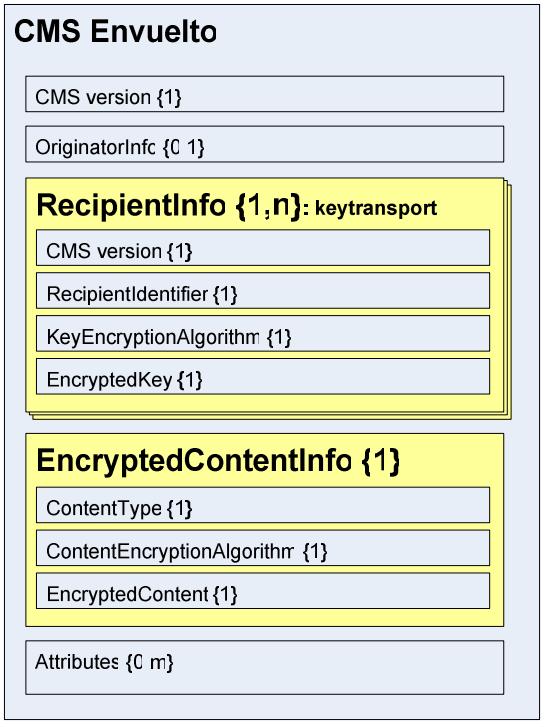


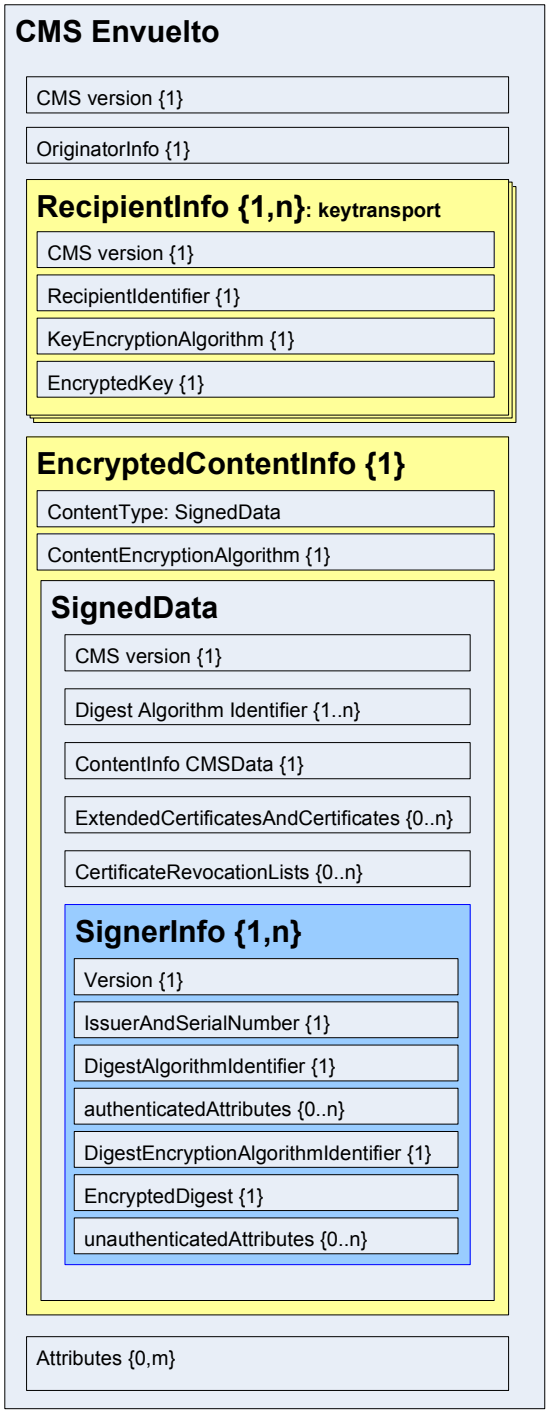
Figura 21: Estructura PKCS#7 EnvelopedData

- **CMS Version**. Viene determinada en función de los parámetros presentes en la estructura generada. Para que fuese compatible con la estructura especificada en PKCS#7 debería ser 0, pero quitaría mucha de las opciones más importantes que incorpora esta solución.
- **Originator Info**. Define el emisor del mensaje. Aunque es opcional, su presencia viene determinada por los algoritmos utilizados internamente.
- **Recipient Info**. Define los receptores válidos para el mensaje actual. Por requerimientos de la tecnología utilizada serán de tipo **keytransport**, ya que necesitamos incorporar la clave simétrica utilizada en el cifrado. Se distinguen los siguientes campos:
  - **Version**. Puede ser 0 o 2 en función de los datos incluidos en **Recipient Identifier**. En nuestro caso será 2.

- **Key Encryption Algorithm**. Define el algoritmo por el cual se ha encriptado la clave simétrica adjunta. Se utilizan algoritmos asimétricos y en nuestro caso RSA.
- **Encrypted Key**. Clave utilizada para encriptar el contenido del mensaje cifrada utilizando el algoritmo anteriormente definido.

- **Encrypted Content Info**. Estructura igual que la contenida en CMS encriptado:
  - **Content Type**. Define el tipo de contenido.
  - **Content Encryption Algorithm**. Define el algoritmo utilizado para encriptar el contenido.
  - **Encrypted Content**. Contenido encriptado usando el algoritmo especificado especificado.
- **Unprotected Attributes**. Conjunto de atributos no cifrados definidos o necesarios por otros estándares.

16.10 Formato de sobre digital CMS Firmado y envuelto



Esta estructura es un atajo para crear un CMS envuelto en cuyo interior se encuentra un mensaje firmado. Esto significa que la única diferencia en cuanto a la estructura es que el *conten type* de la subestructura *Encrypted Content Info* sería un *Signed Data* como el definido en el CMS Firmado.

La diferencia fundamental es que los parámetros a especificar no son tan libres, ya que por ejemplo es obligatorio especificar el emisor, ya que tenemos que firmar el mensaje con su certificado.

Este es un ejemplo de cómo se pueden anidar estructuras CMS. Por ejemplo, podríamos insertar un CMS envuelto en un CMS firmado (obviando la utilidad que pudiese tener o no) simplemente generando el CMS envuelto y especificando el resultado de la salida como datos de entrada para la creación del CMS firmado, y así sucesivamente.

Figura 22: Estructura PKCS#7 SignedAndEnvelopedData

16.11 Formato de sobre digital CMS Autenticado

Esta estructura se identifica con el sobre digital identificado en la RFC 3852 como Authenticated CMS y sigue la siguiente estructura:

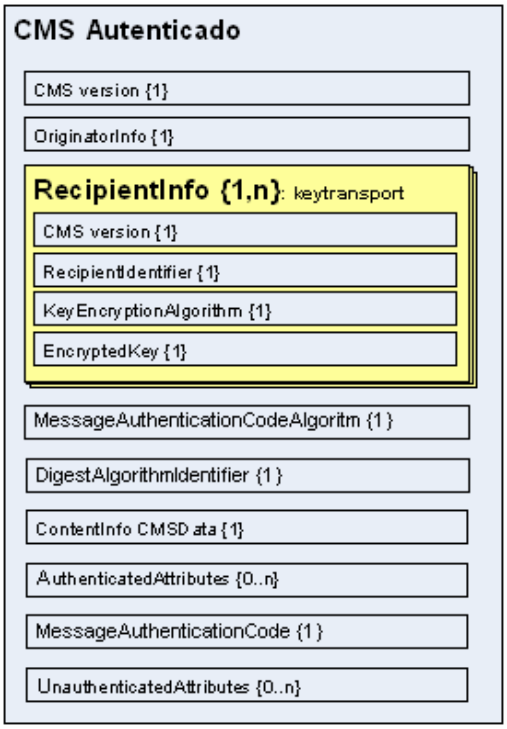


Figura 23: Estructura CMS AuthenticatedData

- **CMS Version.** Viene determinada en función de los parámetros presentes en la estructura generada. Para que fuese compatible con la estructura especificada en PKCS#7 debería ser 0, pero quitaría mucha de las opciones más importantes que incorpora esta solución.
- **Originator Info**. Define el emisor del mensaje. Aunque es opcional, su presencia viene determinada por los algoritmos utilizados internamente.
- **Recipient Info**. Define los receptores válidos para el mensaje actual. Por requerimientos de la tecnología utilizada serán de tipo **keytransport**, ya que necesitamos incorporar la clave simétrica utilizada en el cifrado. Se distinguen los siguientes campos:
  - **Version**. Puede ser 0 o 2 en función de los datos incluidos en **Recipient Identifier**. En nuestro caso será 2.
  - **RecipientIdentifier:** Identifica al usuario al que va dirigido el sobre.

- **Key Encryption Algorithm**. Define el algoritmo por el cual se ha encriptado la clave simétrica adjunta. Se utilizan algoritmos asimétricos y en nuestro caso RSA.
- **Encrypted Key**. Clave utilizada para encriptar el contenido del mensaje cifrada utilizando el algoritmo anteriormente definido.

- **MessageAuthenticationCodeAlgorithm**: Define el algoritmo con el que se creará la MAC.
- **DigestAlgorithmIdentifier**: Identifica el algoritmo utilizado.
- **ContentInfo**: Secuencia de parámetros que identifican el contenido del mensaje. Comprende el tipo de contenido (*contentType*) que en nuestro caso será *Data* y *content* que se refiere a la secuencia de bytes correspondiente a los datos mismos.
- **AuthenticatedAtributes:** Opcional. Secuencia de atributos firmados que especifican ciertos parámetros importantes para la interpretación del contenido. Si el tipo del contenido fuese distinto de *Data*, sería obligatorio incorporar como atributos el tipo empleado y el hash del contenido, pero en nuestro caso esto no es posible ya que siempre tendremos el *ContentType Data*.
- **MessageAuthenticationCode:** Código que autentifica el mensaje.
- **Unauthenticated Attributes**. Opcional. Atributos no firmados definidos en PKCS#9, como por ejemplo las contrafirmas.

17 Anexo B. Configuración específica para el formato CadES

El Cliente de firma genera firmas CAdES compatibles por estructura y atributos tanto con la versión 1.7.3 como con la 1.8.1, pero en ambas versiones, el atributo **Signing Certificate** se puede generar de dos formas distintas, la V1 y la V2.

Por defecto, y para una mayor compatibilidad, este atributo se genera de la forma V2, teniendo que especificar un parámetro adicional desde el Applet para pasar a generar con la forma V2, mediante el método del Applet (que es posible invocar vía JavaScript): SignApplet.addExtraParam(String paramName, String paramValue), y el siguiente uso:

signApplet.addExtraParam("signingCertificateV2", "**true**");

Desde la invocación de este método todas las firmas CAdES que se realicen hasta el reinicio del Applet tendrán la forma V2 del atributo Signing Certificate. Si queremos restablecer el comportamiento normal de generación en la forma V1 debemos invocar el paso de parámetro adicional de este otra forma:

signApplet.addExtraParam("signingCertificateV2", "**false**");

Para las firmas en CAdES que van a sufrir un tratamiento posterior acorde a la versión CAdES 1.8.1 (como por ejemplo sellos de tiempo complejos), se recomienda usar siempre la forma V2 del atributo Signing Certificate.

ADVERTENCIA: Se han detectado problemas en la validación por parte de VALIDe de las firmas CAdES realizadas con algoritmos SHA-2. Así mismo, si se usasen algoritmos de firma SHA-2 se recomienda la activación del parámetro "signingCertificateV2".

18 Anexo C. Evolución del cliente

A continuación se detalla grosso modo las funcionalidades más importantes incorporadas en cada versión del cliente de firma:

Versión 1.1

- Incorporación de funciones de cifrado.
- Incorporación de modos de firma explícito e implícito en formato CMS.
- Añadida compatibilidad entre CMS y PKCS#7.
- Añadidos demostraciones de uso.

Versión 1.2

- Incorporación de los formatos de firma XMLDSignature y XAdES v1.1.1 como plugin.
- Mejoras en el módulo de cifrado. Incorporación de claves alfanuméricas no seguras.
- Corrección de bug de pérdida de foco en diálogos Java.
- Añadida la capacidad de introducir atributos firmados y no firmados.
- Añadidos ejemplos.
- Mejorada la instalación e integración de plugins.

Versión 2.1

- Adaptación del formato de firma XADES a estándar 1.3.2.
- Nueva distribución de elementos de firma XML.
- Corrección de bugs relativos a representación gráfica en Linux.
- Corrección de bugs de generación de firmas XML.
- Mejora del sistema de firmado en bloque.

Versión 2.3

- Corrección de bug de interpretación del árbol de firmas CMS.
- Corrección de contrafirma y cofirma CMS.
- Añadida capacidad de acceso a ficheros remotos.

Versión 2.3.1

- Corrección de bug relativo al filtrado de los certificados.

Versión 2.3.2

- Posibilidad de parametrización de la aparición o no de la ventana que informa al usuario acerca del hash que va a proceder a firmar.

- Aparición del disclaimer del instalador en negrita.

Versión 2.3.3

- Modificación de la DLL y del cliente para adaptación a formato propietario de tarjetas SIEMENS, que no utilizan como alias de certificados cadenas con codificación UTF-8

Versión 2.4

- Compatibilidad con Windows Vista e Internet Explorer 7.
- Testeada compatibilidad con Red Hat v4
- Incorporación del formato de firma CADES-BES.
- Adición de funcionalidad para el formato de firma XAdES (detached, enveloped y enveloping, incorporación de mime types)
- Descifrado de sobres digitales.
- Incorporación del sistema de distribuciones, mediante el cual pueden coexistir diferentes
- Versiones del cliente instaladas en la misma máquina del usuario.
- Se robustece el sistema de instalación y los mensajes de error al usuario.
- Se solventa el problema de Mozilla Firefox cuando el cliente está publicado en entorno seguro HTTPS.
- Se ha modificado el mecanismo de carga de DLLs dinámicas, que corrige el problema de múltiples instancias del cliente en el mismo navegador.
- Se ha creado una clase para obtener los mensajes desde un archivo .properties por cada plugin. Para mantener el nivel de abstracción del cliente se han implementado independientemente por cada parte del cliente. Así existen 6 clases con su correspondiente archivo .properties (instalador, commom, cliente, XAdES, PDF y ODF).
  - Instalador → LanguageInstalador.java → messsages_instalador.properties
  - Common → LanguageCommon.java → messages_common.properties
  - Cliente → LanguageClient.java → messages.properties
  - Plugin XAdES → LanguageXades.java → messages_xades.properties
  - Plugin ODF → LanguageODF.java → messages_odf.properties
  - Plugin PDF → LanguagePDF.java → messages_pdf.properties

Con esta mejora se pretende dar la posibilidad poder cambiar el idioma fácilmente en un futuro, de forma que todos los mensajes que muestra el cliente tienen su referencia en los archivos .properties.

Versión 3.0.2

- Re-implementación completa del Cliente de firma, con los siguientes objetivos:
  - El cliente se basa ahora en la arquitectura de certificación y firma electrónica definida en JCA (*Java Cryptography Architecture*), omitiendo API propietarios siempre que es posible. (DigiDoc, IAIK, etc.).
  - Cese de la compatibilidad con Java 1.4, permite usar ahora las funcionalidades avanzadas de firma electrónica de Java 6 (incluso en Java 5).
  - El soporte de Windows CAPI pasa a ser el estándar de Java, omitiendo bibliotecas a medida, lo cual garantiza la compatibilidad con cualquier versión y arquitectura de Windows soportada por el proveedor de seguridad estándar de Java SunMSCAPI.
  - Eliminación de JSS y uso directo de NSS como dispositivo PKCS#11. Esto proporciona compatibilidad con cualquier versión de Mozilla / Firefox actual y futura.
  - Mejora del proceso de instalación, eliminando por completo la necesidad de permisos de Administrador en JRE 6.
- Actualización de la firma ODF a la versión 3 (y superiores) de OppenOffice.org, que difiere notablemente respecto a la versión 2 (y rompe la retro-compatibilidad).
- Soporte de SHA-2 en firmas PDF.
- Soporte parcial de SHA-2 en firmas binarias.
- Soporte preliminar de SHA-2 en firmas XML (menos ODF).
- Inclusión de la cadena de certificación en las firmas siempre que es posible.
- Adecuación de los diálogos de usuario a las recomendaciones de "Java Look and Feel" y "Windows Power Experience".
- Uso de "Java Logging API" para el registro de errores, mensajes y advertencias.
- Eliminación del sistema de plugins para adecuarse a las necesidades de despliegue de Applets de Java 1.6u17 y superiores.
- Distribución del cliente en 3 configuraciones distintas: LITE, MEDIA y COMPLETA.

Versión 3.1

- Soporte de transformaciones a medidas en firmas XML.
- Soporte de transformaciones Base64 en firmas XML de contenido binario.
- Reducción adicional de tamaño.
- Soporte de Internet Explorer 9 (platform preview).
- Soporte de almacenes de claves externos (PKCS#12, etc.).
- Soporte de LDAP para obtención de claves públicas.
- Soporte de XAdES V1.4.1.
- Soporte de Windows x64 en Internet Explorer (CAPI).
- Nuevo motor de filtrado de certificados según la RFC 2254.
- Paliación de los problemas de "script ocupado" en Firefox 3.x
- Soporte de firmas OOXML para Microsoft Office 2007, 2008 y 2010.
- Soporte de despliegue vía JNLP.
- Soporte del idioma Inglés en los mensajes desde el Applet hacia el usuario (auto-detección del lenguaje del sistema operativo).