UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

VERIFICACIÓN DE FIRMAS DIGITALES EN DOCUMENTOS XML DE TAMAÑO ARBITRARIO

Autor: Raúl Benito García

Tutor: José Luis Pedraza Domínguez

Boadilla del Monte, diciembre 2006

Tabla de Contenido

1. INTRODUCCIÓN 1

1.1. XML 2
1.2. FIRMAS DIGITALES 4
1.3. FIRMAS XML 6
1.4. RENDIMIENTO DE LOS SISTEMAS DE FIRMA DIGITAL 9
1.5. ESTRUCTURA DEL DOCUMENTO 10

2. XML 13

2.1. SINTAXIS 13
2.2. ESTRUCTURA SEMÁNTICA 16
2.3. DTD 17
2.4. NAMESPACES 20
2.5. XML SCHEMA 24
2.6. PROCESAMIENTO XML 25
1.1.1. Ejemplo de lectura DOM: 26
1.1.2. Ejemplo de lectura SAX 30
1.1.3. Ejemplo de lectura STAX 33

3. FIRMAS XML 37

3.1. CONVERSIÓN A FORMA CÁNONICA 37
3.2. FIRMAS XML 39
3.3. EJEMPLO DE FIRMA 44
3.4. EJEMPLO DE VERIFICACIÓN 46
3.5. PREGUNTAS/ATAQUES 48
3.6. TIPOS DE FIRMA 49

4. FIRMA DE DOCUMENTOS TAMAÑO ARBITRARIO 53

4.1. VERIFICADOR STREAM 56
4.2. VERIFICADOR DE FIRMAS STAX 57
4.3. TRABAJADORES Y OBSERVADORES. 60
4.4. EJEMPLO DE VERIFICACIÓN STREAM 61

5. IMPLEMENTACIÓN DE UN VERIFICADOR STREAM. 67

6. RESULTADOS 77

6.1. VERIFICACIÓN DE 10.000 FIRMAS DE 1322 BYTES 81

6.2. VERIFICACIÓN DE 1.000 FIRMAS DE 42 KB 83
6.3. VERIFICACIÓN DE 10 FIRMAS DE 1 MB 85
6.4. VERIFICACIÓN DE UNA FIRMA DE 34 MB 89
6.5. VERIFICACIÓN DE UNA FIRMA DE 103 MB 94

7. CONCLUSIONES 97

8. BIBLIOGRAFÍA 99

9. APÉNDICE: MEJORAS DE RENDIMIENTO EN LA LIBRERÍA DE FIRMAS DOM. 101

1. Introducción

El *Extensible Markup Language* [4] (XML lenguaje extensible de etiquetas), desarrollado por el w3c[1] se ha constituido en la lengua franca tanto para el intercambio de ficheros entre programas informáticos, como de datos entre ordenadores. Ya ha sustituido al formato simple delimitado por comas (CSV) y a casi todos los formatos binarios específicos para cada dominio.

Cuando el XML se usa como protocolo de comunicaciones, sería deseable tener una manera estándar de verificar:

1. Que el mensaje proviene del remitente (Autenticidad).
2. Que el mensaje no ha sido modificado por una entidad externa o por las partes después de haberse firmado (Inviolabilidad).
3. Que el remitente no pueda negar su emisión (No repudio).

Todas estas características son las comunes de un sistema de firma digital estándar, donde al texto que se ha firmado se le concatena un número que es el resultado de una operación matemática que solo el remitente puede realizar. Pero como veremos más adelante, este sistema no produce un fichero XML válido.

Es por eso que el w3c estandarizó un proceso de firma y verificación de ficheros XML.

Este proyecto fin de carrera (TFC) refleja el trabajo realizado por el autor para mejorar el rendimiento de la liberaría de firmas XML [13] de la *Apache Software Foundation (ASF)* [14] de la cual es uno de los contribuyentes principales. En particular es responsable de las versiones 1.2, 1.3 y actualmente de la 1.4. También es miembro

[1] El Consorcio World Wide Web (W3C) es una asociación internacional formada por organizaciones miembro del consorcio, personal y el público en general, que trabajan conjuntamente para desarrollar estándares Web.

del *Project Management Council* del proyecto Santuario (http://santuario.apache.org/) de la ASF.

La librería de firmas que aquí se describe va ser incorporada en la versión 1.6 de la *Java Virtual Machine* como componente estándar para el firmado/verificado de firmas XML [12].

1.1. XML

El XML es un formato de texto que intenta representar estructuras de árbol (de la misma manera que el CSV intenta representar estructuras de tabla).

La sintaxis de XML es bastante sencilla. Un nodo y su contenido (incluidos otros nodos hijos suyos) están delimitados por dos etiquetas (*tags*): la de inicio llamada *start tag* y la de final llamada *stop tag*. Estas etiquetas contienen el nombre del nodo entre los símbolos de mayor y menor y la de parada lleva una barra antepuesta al nombre.

Por ejemplo, para un nodo de nombre *element* la etiqueta de inicio sería `<element>` y la de final `</element>`. Entre estas dos etiquetas puede encontrarse texto simple (dentro de ese texto no pueden aparecer los caracteres mayor o menor). Si se quieren representar esos caracteres se deberán usar entidades (como veremos más adelante) o más nodos.

Un ejemplo de un fichero XML válido, es el siguiente:

```
<Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
</Padre>
```

En el XML de arriba, el árbol representado tiene como raíz el nodo (elemento en la terminología XML) llamado `Padre` que contiene el texto (`Texto que pertenece al padre.`) y un hijo `Hijo` que a su vez contiene otro texto.

El XML obliga a que sólo haya un elemento raíz en un documento. No sería válido poner otro elemento al final del nodo `Padre`. Por ejemplo, el siguiente fichero no es un XML válido:

```
<Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
</Padre>
<OtroPadre>
 ...
</OtroPadre>
```

Esta restricción no es muy limitante porque en caso de que se necesite tener varios elementos siempre se puede inventar un nodo raíz que los englobe. Por ejemplo el fichero de arriba se puede volver válido con este simple cambio:

```
<Padres>
 <Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
 </Padre>
 <OtroPadre>
   ...
 </OtroPadre>
</Padres>
```

La indentación de los nodos no es necesaria y solo la hacemos por aumentar la claridad.

Como se ve, esta sintaxis simple hace del XML un formato inteligible por las personas, no ambiguo y "fácilmente" analizable por las máquinas.

Como principales ventajas del XML podríamos señalar:

1. La legibilidad por parte de personas y máquinas.

2. El uso de estructuras de árbol que permite representar fácilmente otras estructuras típicamente utilizadas en informática como listas, registros, tablas.

3. El uso de texto plano UTF8 que le hace fácilmente internacionalizable e independiente de plataforma. El UTF8 es un estándar derivado del ASCII que permite codificar caracteres *Unicode* sin que ningún *byte* de la cadena sea un 0 binario. Esta característica hace que APIs y programas antiguos (sobre todo escritos en C) que no estuvieran preparados para aceptar un 0

binario en su entrada (y por lo tanto otras codificaciones de *Unicode*), funcionen correctamente con caracteres UTF8 (mientras no necesiten contar el número de letras en una palabra que es distinto que el número de *bytes*).

4. Es un estándar internacionalmente definido por el w3c y portado a múltiples sistemas operativos y arquitecturas hardware.

Más adelante se describirá de forma más detallada el XML

1.2. Firmas Digitales

Las firmas en el mundo físico sirven para dar autenticidad y conformidad a un documento. Su fundamento se basa en que se puede verificar con bastante fiabilidad que una firma (dibujo) ha sido realizada por una determinada persona (a través de una prueba pericial) y que el documento en el que está incluida no ha sufrido modificaciones después de su firma (que se podrían detectar por inspección física). Con lo que la firma y el documento en el mundo físico son un todo indivisible y da las siguientes características:

1. Autenticidad
2. Inviolabilidad
3. No Repudio

Las firmas digitales deben tener las mismas características para los documentos digitales. Para conseguir esto se basan en sistemas de criptografía de clave pública.

En estos sistemas el firmante tiene dos claves, una clave pública (muy distribuida y que debe tener cualquiera y sobre todo el receptor para verificar la autenticidad del mensaje) y una privada (que el firmante debe mantener secreta y segura y es con la que va a firmar el mensaje). El par de claves, por supuesto, tiene una relación algorítmica entre ambas aunque debe ser imposible o muy difícil (es decir computacionalmente costoso) deducir la clave privada a partir de la pública (Si eso fuera posible cualquier persona con la clave pública del firmante podría impersonarle y el sistema fallaría).

Todos los sistemas de firma digital especifican tres algoritmos:

1. Un algoritmo de generación de claves: Que específica como se crea el par de clave privada/clave pública (esta creación es habitual que necesite un componente aleatorio).
2. Un algoritmo de firma: Que especifique cómo dado un texto y el par de claves se da como resultado un número resumen que sólo se pueda haber generado con esa clave y ese texto (aunque al ser un número resumen de longitud fija y finita existe la posibilidad de colisión pero ésta debe asegurarse que es sólo con textos muy diferentes y no con pequeñas variaciones del texto original y nunca con otra clave y con el mismo texto).
3. Un algoritmo verificador: Donde dado un texto, el número resumen y una clave pública determine si el mensaje ha sido firmado efectivamente con esa clave pública.

Con este proceso se pueden conseguir unas características parecidas a las de la firma física.

El sistema descrito no resuelve cómo el verificador consigue la clave pública del firmante. Simplemente se asume que el verificador la ha obtenido por medios confiables y que ha certificado que la clave pública pertenece al firmante (el equivalente físico de ver al firmante realizar la forma en el papel). Tampoco especifica qué hacer si al firmante le roban la clave privada (posible impersonamiento por otras personas) o si el firmante destruye la clave (con lo que imposibilita el repudio y permite la cancelación de contratos).

Todos estos problemas se suelen resolver recurriendo a un tercero en el que tanto el firmante como el receptor tengan confianza. Esta entidad certifica que la clave pública pertenece al firmante (normalmente firmándole su clave pública). Si al firmante le roban la clave privada, éste notifica a la entidad certificadora su pérdida, y ésta revocará la firma (esto implica que el verificador tiene que contactar siempre con la entidad certificadora para comprobar la validez de una clave pública). También

soluciona el problema del repudio porque el verificador tiene la certificación de que la clave pertenece (o ha pertenecido) al firmante y éste no puede hacer nada para negarlo.

Para ver un ejemplo de firma digital podemos firmar con PGP (*Pretty Good Privacy*), uno de los algoritmos de firmas más usados, el documento XML de arriba (como es texto no se tiene que hacer nada especial).

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

<Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
</Padre>

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.5 (Cygwin)

iD8DBQFFXNVTt3ozLmAUGPQRAqzDAKCXJNBuKa5RqXY2rodYRRkwFLbX+gCfduV2
QvOFw94P+UeoFAfw7bdjDBQ=
=Nscg
-----END PGP SIGNATURE-----
```

Cualquier persona que tuviera la clave pública del firmante podría verificar la validez de este documento.

El problema es, como se puede ver, que el documento de firmado no es un XML válido (no tiene un elemento raíz que lo englobe) y eso impide que pueda ser tratado como un XML por los programas.

Este problema es importante si queremos tener un protocolo XML que tenga características de firma digital, ya que, como hemos visto con los algoritmos típicos, no se puede firmar un XML y que el resultado sea un XML válido.

1.3. Firmas XML

El documento de arriba firmado se podría convertir en XML fácilmente insertándolo dentro de un elemento raíz:

```
<Firma>
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

<Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
</Padre>

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.5 (Cygwin)

iD8DBQFFXNVTt3ozLmAUGPQRAqzDAKCXJNBuKa5RqXY2rodYRRkwFLbX+gCfduV2
QvOFw94P+UeoFAfw7bdjDBQ=
=Nscg
-----END PGP SIGNATURE-----
</Firma>
```

Esto sería un documento XML válido, pero obligaría al software que lo trata a entender dos tipos de estructuras: la de XML y la de PGP (que delimita los campos con "----"). Una solución más elegante sería transformar la estructura PGP también a XML:

```
<Firma>
<Mensaje>
<Hash>SHA1</Hash>

<Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
</Padre>
</Mensaje>
<DatosFirma>
<Versión>GnuPG v1.4.5 (Cygwin)</Versión>

iD8DBQFFXNVTt3ozLmAUGPQRAqzDAKCXJNBuKa5RqXY2rodYRRkwFLbX+gCfduV2
QvOFw94P+UeoFAfw7bdjDBQ=
=Nscg
</DatosFirma>
</Firma>
```

Esto básicamente es una firma digital XML. Pero como se puede ver ésta es una estructura XML arbitraria elegida por nosotros, otra persona podría haber elegido mover el elemento `Hash` del Mensaje a los datos de la firma, o cambiar los nombres de los elementos. O decidir este otro tipo de estructura:

```
<Padre>
  Texto que pertenece al padre.
  <Hijo>
    Texto del hijo
  </Hijo>
  <Firma>
    <DatosFirma>
     <Hash>SHA1</Hash>
     <Versión>GnuPG v1.4.5 (Cygwin)</Versión>
iD8DBQFFXNVTt3ozLmAUGPQRAqzDAKCXJNBuKa5RqXY2rodYRRkwFLbX+gCfduV2
QvOFw94P+UeoFAfw7bdjDBQ=
=Nscg
    </DatosFirma>
  </Firma>
</Padre>
```

En este ejemplo, los datos de la firma se han incluido dentro del elemento firmando. Este formato/opción, tiene varias ventajas respecto al anterior.

1. La estructura del XML respeta mucho más la estructura original.
2. Se podrían unificar en un mismo documento distintos elementos firmados. Aunque en ese caso sería deseable que en los Datos de la firma hubiera una referencia a la clave pública usada para firmar ese elemento. De esta manera se podría tener en un documento recopilaciones de elementos firmados por distintos firmantes.

Pero como contrapartida, el mecanismo de verificación no es tan sencillo como quitar el elemento raíz y volver a cambiar los delimitadores XML a PGP, sino que hay que buscar los elementos `DatosFirma` (que pueden estar arbitrariamente colocados) y extraerlos ya que no estaban en el documento original.

Cualquiera de las opciones para crear una firma digital es válida. Pero se ve la necesidad de un estándar que determine qué estructura XML (es decir, cómo tiene que ser el árbol, como se llaman las etiquetas) de las firmas XML.

De esta manera los diseñadores de protocolos no tienen que pensar y documentar su estructura. Se puede crear un mercado de librerías que faciliten la creación y verificación de firmas XML, reduciendo así el tiempo de desarrollo de soluciones que las necesiten.

El estándar de firmas XML más usado en estos momentos fue definido por el w3c [1]. Y además de las funcionalidades que hemos esbozado, tiene otras muchas características que veremos más adelante.

1.4. Rendimiento de los sistemas de firma digital

El estándar de librerías de firmas digitales XML, es interesante y flexible, y se ha usado como base para otros estándares de seguridad como el SAML [6] o *Liberty* ID-FF [7]. Cuando el autor empezó a trabajar con estos últimos sobre el 2002 por motivos laborables, la implementación de dominio público más famosa (y la única) era la implementación de Apache para Java. Pero tras integrarla en nuestros desarrollos, los resultados obtenidos respecto al rendimiento dejaban mucho que desear (1 firma por segundo), los documentos a firmar eran peticiones *Liberty* de 1,6 KB de tamaño, y los requisitos mínimos de rendimiento eran de unas 60 firmas/segundo.

Investigando el porqué de esos resultados pobres, el autor descubrió varios cuellos de botella en la implementación fácilmente corregibles (lecturas del fichero de configuración varias veces por firma). Después de enviar parches para solucionarlos y con los problemas obvios subsanados se aumentó el rendimiento a 4 firmas por segundo.

Pero una vez atacados los problemas sencillos las mejoras vinieron por el cambio de algoritmos. La reescritura de varios de ellos para convertirlos de recursivos a iterativos, y el rediseño de algunas estructuras de datos, mejoró de forma considerable la velocidad (10 firmas por segundo en los ordenadores de la época, 2002).

Al mismo tiempo acometimos un proceso de reducción de la memoria consumida en el proceso de firma. Esta reducción fue perseguida por la mejora de rendimiento que experimentan los servidores que sufren un elevado número de firmas/verificaciones simultáneas, al reducir la presión sobre el *garbage collector*. Esto se trata de forma extensiva en el capítulo de resultados.

Pero esta reducción además de mejorar la ejecución permitió firmar documentos más grandes, la implementación original se colapsaba con documentos de 512 KB, las

siguientes versiones trabajaban sin problemas con firmas de hasta 10 MB y luego 100 MB.

Sobre el 2004, hubo en las listas de distribución de la librería varias muestras de interés por superar el límite de los 100 MB (por ejemplo el gobierno de Hong Kong tenía necesidades de firma de documentos de >250 MB y creciendo).

Este problema no se puede resolver sin cambiar los programas que hacen uso, de la librería debido a que la limitación de tamaño no viene impuesta por el diseño interno de la librería de firmas, sino por la librería usada para la lectura de documentos XML (y esta librería condiciona la arquitectura de los clientes como veremos más adelante)

Para quitar esa limitación el autor creó un parche experimental [15] basado en una tecnología de lectura de documentos XML llamada SAX, distinta de la que usaba la librería de firmas en ese momento (DOM [11]). Este parche está actualmente incluido en productos comerciales y es usado como ventaja competitiva. El principal inconveniente de esta solución es que el API SAX es de utilización compleja.

La investigación posterior con ese parche experimental que permite la firma de documentos arbitrariamente grandes y su adecuación a un API de procesado XML más amigable que el SAX es el principal objetivo que aborda este trabajo de fin de carrera.

Como nota final, todo el código aquí expuesto está disponible en la web de Apache Software Foundation para cualquier persona interesada.

1.5. Estructura del Documento

Este documento esta dividido en tres grandes bloques.

1. La primera parte explicará los conceptos y tecnologías necesarios para entender la solución dada al problema. Estos son los capítulos **XML** y **Firmas XML**.
2. La segunda parte presenta las distintas soluciones diseñadas al problema de firma de documentos de tamaño arbitrario. También explica el diseño e implementación de la solución actual. Estos son los capítulos **Firma de**

documentos tamaño arbitrario e **Implementación de un verificador Stream.**

3. La tercera parte muestra los datos y conclusiones que han llevado las soluciones presentados en el apartado anterior. Esto son los capítulos **Resultados** y **Conclusiones**.

2. XML

El XML surgió como estándar del w3c en el 1998 como un sustituto del HTML cuando se empezaron a ver las limitaciones de éste para la representación de documentos complejos. Los dos son subconjuntos de un sistema de *markup* más general denominado SGML, estandarizado por ISO en 1986.

Pero aun heredando del mismo padre, las diferencias entre los dos son grandes.

Mientras el HTML eligió una serie de *tags* (etiquetas) y les dio significado y representación a estos. El XML decidió no definir ningún *tag* y dejar que el usuario defina los mismos y su representación visual (si fuera necesario). En SGML los usuarios pueden definir nuevos *tags* como en XML, pero también tiene una serie de *tags* definidos estándar a la manera del HTML.

Otra diferencia es la sintaxis y estructura: En el XML la estructura es más estricta que la del HTML y SGML. Mientras estos últimos permiten abrir y cerrar los *tags* sin seguir un orden fijo (i.e. `<b><a></b></a>`). En el XML esa combinación de *tags* esta prohibida porque no define ninguna estructura arbórea, condición necesaria para que sea un documento XML valido.

Esta última característica permite que en SGML y HTML sean correctos documentos multi-raíz (aunque el SGML y el HTML no tienen por que definir árboles), mientras que en XML sea imposible.

Como resumen, se puede decir que tanto el XML como el HTML son subconjuntos del SGML. Es decir todo documento XML valido es también SGML valido, pero a la inversa no tiene por que suceder.

2.1. Sintaxis

Un documento XML tiene tanto una estructura física como lógica. Físicamente podemos definir las siguientes entidades (elementos léxicos):

1. Texto:

   En XML cualquier secuencia de caracteres UTF8 que no ha sido

clasificado como otra entidad se le denomina Texto. Es importante notar que la existencia de algunos caracteres, como los símbolos < o >, hacen que una secuencia no sea reconocida como texto y sea presentado como un error sintáctico o simplemente confundido como otra entidad. Si el texto necesita representar estos caracteres tiene que sustituirlos por la cadenas > y <. Al usar el carácter & para definir estos términos también obliga a "escapar" este carácter. Si se quiere usar solo se tendría que sustituir por la secuencia &
Por ejemplo, la manera de codificar la entidad de texto a > b & c < d en un texto XML, seria:

```
a &gt; b &amp; c &gt; d
```

2. Comentarios:

   Un comentario en XML es texto rodeado de la secuencia de caracteres de inicio <!-- y la secuencia de caracteres final --> . Por compatibilidad con SGML en el texto del comentario no puede aparecer la secuencia de caracteres -- pero si puede incluirse caracteres como <,> o & sin que estos sean interpretados como caracteres especiales.

```
<!-- Esto es un comentario a>b & c<d -->
```

3. Instrucciones de Procesado:

   Esto elementos se mantienen por compatibilidad con SGML, su función original era dar información al programa de cómo procesar el documento aunque actualmente esta en desuso.

```
<? Procesing Instruction ?>
```

4. CDATA

   Un elemento CDATA es un texto rodeado por la secuencia de caracteres de inicio <![CDATA[y de la de fin]]>. Los elementos CDATA son una forma alternativa de escribir texto con la ventaja de que los caracteres especiales <,> y & no tienen que sustituirse por otras cadenas dentro de ella. Pero esto a su vez es un inconveniente porque no se puede definir ningún subelemento dentro de ellas.

```
<![CDATA[a>b & c<d ]]>
```

5. Elementos

   Un elemento consta de dos etiquetas: la de inicio y la etiqueta de fin, posiblemente envolviendo texto y otros elementos. La etiqueta de inicio consiste en un nombre (sin espacios) flanqueado por los símbolos de <,>. Y la etiqueta de fin es el mismo nombre antepuesto el símbolo / y también flanqueado de los símbolos de desigualdad. El contenido del elemento es cualquier texto u otros elementos que se contengan entre las dos etiquetas. Además de contenido los elementos también pueden tener atributos. Los atributos son pares atributo, valor que se encuentran dentro de la etiqueta de inicio separada del nombre por espacios. El valor del atributo tiene que estar entre comillas y su nombre ser único en ese elemento (es decir no se pueden repetir atributos dentro de un elemento). Por supuesto si el valor del atributo tuviera que contener el carácter " este se tendrá que sustituir por la cadena `"`.
   Una peculiaridad es que los elementos sin ningún contenido en vez de repetir inmediatamente la etiqueta de final, esta se puede obviar concatenando una barra al nombre antes del símbolo de mayor que (`<ejemplo/>` es equivalente `<ejemplo></ejemplo>`)

```
<elemento atributo1="Valor1" atributo2="2">
Texto contenido en el elemento <subelement/>
</elemento>
```

Estas son todas las entidades que pueden aparecer en un documento XML. Ahora bien estas entidades no pueden aparecer de cualquier manera, si no que un documento léxicamente valido también tiene que ser sintácticamente valido.

Las restricciones sintácticas del XML:

1. El texto o las secciones CDATA siempre tiene que estar contenido en algún elemento.

2. El orden de las apertura (etiquetas de inicio) y cierre (etiquetas de fin) tiene que ser el orden estricto de anidamiento, es decir `<a><b></a></b>` no se permite porque no mantiene el anidamiento. Esta regla es idéntica a la regla de apertura y cierre de paréntesis en la notación matemática (Ej. ([ )] es incorrecto ([ ]) es correcto).
3. Solo puede existir un elemento raíz.
4. Los comentarios y elementos de procesamiento pueden aparecer en cualquier parte.

Si un documento cumple todas las restricciones léxica y a la vez las sintácticas se dice que es un documento XML bien formado.

2.2. Estructura Semántica

El estándar XML no determina ninguna semántica a los elementos ni ninguna estructura fija. Esta flexibilidad es a veces indeseable: muchos sistemas necesitan definir una estructura fija, unos datos mínimos que tienen que existir, una semántica de posición, etc.

Por ejemplo, un sistema de base de datos electrónica de alumnos: trabaja con un registro de alumnos que tiene los datos del nombre y los apellidos del mismo.

```
<Alumno>
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
</Alumno>
```

Si ahora alguien intentara introducir un campo como por ejemplo una calificación:

```
<Alumno>
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
<Nota tipo=”media”>7</Nota>
</Alumno>
```

El sistema posiblemente fallara ya que no entiende el concepto de `Nota`.

Ahora bien si el sistema verificara el nombre de los elementos y solo aceptara documentos con elementos reconocidos, es posible que tuviera también problemas con el siguiente documento:

```
<Alumno>
<Nombre>Raul</Nombre>
<Nombre>Roberto</Nombre>
<Apellidos>Benito Garcia</Apellidos>
</Alumno>
```

Aquí aunque todos los elementos son conocidos, la manera de codificar nombre compuesto le puede ser extraña. Pero incluso seguro que lo tendría más difícil con el siguiente documento:

```
<Alumno>
<Nombre>Raul
  <Apellidos>Benito Garcia</Apellidos>
</Nombre>
</Alumno>
```

Aquí todos los elementos son conocidos pero su estructura es distinta (lógica en si misma, pero distinta).

Lo más conveniente para evitar todos estos problemas es que los programas puedan especificar los elementos que aceptan y si estos pueden contener o no otros elementos. De esta manera se podrían rechazar los documentos XML que no cumpliera esa estructura.

El estándar XML plantea dos maneras de definir estructuras, una heredada del SGML, y cada vez más en desuso, llamada DTD y otra nueva llamada XML Schema [5]. A continuación las vemos con más detalle.

Una ventaja añadida de que XML permita definir la estructura de datos, es que los generadores de documentos pueden verificar su validez con herramientas estándar XML de generación de documentos sin necesidad de conectarse al sistema de procesamiento.

2.3. DTD

El *Document Type Definition* separa la definición de la estructura de los documentos en dos partes diferenciadas: declaración de elementos, y declaración de lista de atributos.

- Las declaraciones de elementos definen el conjunto de elementos válidos en el documento y específica si estos pueden contener subelementos y cómo los contiene, si son opcionales, si puede haber más de uno, etc. En

DTD los nodos de textos se representan como el elemento #PCDATA, por lo que si dentro de los subelementos permitidos aparece este elemento se permite texto en esa posición.

```
<!ELEMENT Alumno (Nombre, Apellidos, Genero?, Notas*)>
<!ELEMENT Nombre (#PCDATA)>
<!ELEMENT Apellidos (#PCDATA)>
<!ELEMENT Genero (#PCDATA)>
<!ELEMENT Notas (#PCDATA)>
```

Este DTD especifica una estructura muy parecida a los ejemplos anteriores, donde un elemento `Alumno` tiene que tener obligatoriamente un elemento `Nombre` (que solo puede contener texto y nada más) seguido de un elemento `Apellido` (con idénticas restricciones), puede contener un elemento `Genero` opcional y finalmente pueden aparecer 0 o más elementos `Notas`.

El siguiente documento es correcto.

```
<Alumno>
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
</Alumno>
```

Mientras que este otro es incorrecto porque no sigue el orden de aparición de los elementos, no sigue lo especificado (`Apellidos` aparece antes que `Nombre`).

```
<Alumno>
<Apellidos>Benito Garcia</Apellidos>
<Nombre>Raul</Nombre>
</Alumno>
```

- Las declaraciones de atributos definen el nombre de los atributos permitidos para cada elemento definido. También especifica el subconjunto de valores posibles, los valores por defecto, y si son obligatorios o no.

```
<!ATTLIST Notas
        tipo    (media|primerCiclo|segundoCiclo) #REQUIRED>
```

En este ejemplo decimos que el elemento `Notas` tiene un atributo (y solo uno) llamado `tipo` que solo puede tener los valores: `media`, `primerCiclo`, `segundoCiclo` y que es obligatorio

El siguiente ejemplo muestra un elemento válido.

```
<Alumno>
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
<Notas tipo=”primerCiclo”>5</Notas>
<Notas tipo=” primerCiclo”>5</Notas>
</Alumno>
```

Mientras que este documento no es valido porque el elemento Notas no tiene el atributo obligatorio tipo.

```
<Alumno>
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
<Notas>5</Notas>
</Alumno>
```

Puede ser que un sistema acepte más de una definición al mismo tiempo (por ejemplo implementa un DTD actual y otro de versiones anteriores). Para acomodar estos escenarios en el documento XML se especifica que DTD cumple:

```
<?xml version="1.0"?>
<!DOCTYPE alumno SYSTEM "alumnoV11.dtd">
<Alumno>
<Apellidos>Benito Garcia</Apellidos>
<Nombre>Raul</Nombre>
<Notas>5</Notas>
</Alumno>
```

La librería XML analizará el documento, comprobará si es válido respecto al DTD, y se lo pasará al programa especificando qué DTD ha usado (en este caso el alumnoV11.dtd). Es en este momento cuando el programa puede decidir cómo tratar el documento, sabiendo, eso sí, que la estructura ha sido verificada y es acorde a lo que él ha reclamado.

Normalmente, a la vez que se define un DTD, se suele explicar (informalmente) la semántica de las estructuras (es decir, qué significa una entrada Alumno, cómo se van a interpretar las notas, etc.). En este caso es trivial, pero en algunos otros como el caso del estándar de firmas digitales, las estructuras son más abstractas y su significado depende de otras estructuras, con lo que se hace necesario un documento explicando la estructura y su significado.

Cuando se define una estructura XML (por un DTD u otros medios) y su semántica, se dice que se ha definido un lenguaje XML.

2.4. Namespaces

Hasta ahora hemos supuesto que teníamos el poribilidad de definir un lenguaje XML de la nada. Pero lo más común es que el lenguaje esté ya definido y que incluso tengamos que usar más de uno. Esto plantea un problema, y es que habría que utilizar estructuras de ambos DTTS.

La solución más sencilla a este problema consistiría en utilizar los DTDs que necesitemos y concatenarlos en solo DTD.

Sigamos con el ejemplo de nuestro sistema de Alumnos. Imaginemos que el sistema crece y queremos incluir un enlace con la biblioteca para tener controlados también los prestamos del alumno.

```
<?xml version="1.0"?>
<!DOCTYPE libro SYSTEM "libro.dtd">
<Libro>
<Titulo>The Art of Programming Vol.I</Titulo>
<Autor>
<Nombre>
  Donald <Apellidos>Knuth</Apellidos>
</Nombre>
</Autor>
</Libro>
```

La biblioteca ya tiene su lenguaje XML que define un libro. Seria deseable que nuestro sistema lo entendiera para así poder reutilizar los datos ya existentes en el sistema de biblioteca.

Imaginemos que el DTD de la biblioteca es el siguiente:

```
<!ELEMENT Libro (Titulo, Autor*)>
<!ELEMENT Titulo (#PCDATA)>
<!ELEMENT Autor (Nombre)>
<!ELEMENT Nombre (#PCDATA, Apellidos)>
<!ELEMENT Apellidos (#PCDATA)>
```

El problema surge cuando intentamos mezclarlo con el DTD ya definido de Alumnos, ya que hay una colisión con el elemento `Nombre`. La estructura de la biblioteca engloba los `Apellidos` en el elemento `Nombre` mientras que el DTD de alumno separara los dos y los engloba en el elemento `Alumno`.

Estas dos decisiones arbitrarias pero lógicas, impiden que podamos hacer la integración de los dos esquemas simplemente concatenándolos.

Una posible solución sería cambiar nuestra estructura para que `Nombre` en el lenguaje de Alumnos tuviera la misma estructura que `Nombre` en el lenguaje de Biblioteca. Pero este cambio es muchas veces imposible por coste o por razones legales.

Otra solución razonable sería prefijar cada elemento con el lenguaje que lo define

```
<!-- Sistema de Biblioteca -->
<!ELEMENT BibliotecaLibro (BibliotecaTitulo, BibliotecaAutor*)>
<!ELEMENT BibliotecaTitulo (#PCDATA)>
<!ELEMENT BibliotecaAutor (BibliotecaNombre)>
<!ELEMENT BibliotecaNombre (#PCDATA, BibliotecaApellidos)>
<!ELEMENT BibliotecaApellidos (#PCDATA)>
<!-- Sistema de Alumno -->
<!ELEMENT AlumnoAlumno (AlumnoNombre, AlumnoApellidos,
AlumnoGenero?, AlumnoNotas*)>
<!ELEMENT AlumnoNombre (#PCDATA)>
<!ELEMENT AlumnoApellidos (#PCDATA)>
<!ELEMENT AlumnoGenero (#PCDATA)>
<!ELEMENT AlumnoNotas (#PCDATA)>
```

De esta manera no habría ambigüedades. Las estructuras de Alumno y Biblioteca podrían evolucionar por separado. Pero el problema de modificar los sistemas y todos sus documentos XML se mantiene. Además sigue existiendo el peligro de que otro día necesitemos integrarnos con un tercer lenguaje que no haya seguido esta recomendación.

Para solucionar definitivamente esto podríamos pensar en ampliar el estándar XML para que, además del nombre del elemento, estuviera prefijado el DTD del que proviene:

```
<"alumno.dtd":Alumno>
<"alumno.dtd":Apellidos>Benito Garcia</"alumno.dtd":Apellidos>
<"alumno.dtd":Nombre>Raul</"alumno.dtd":Nombre>
<"alumno.dtd":Notas>5</"alumno.dtd":Notas>
<"alumno.dtd":LibrosPrestados>
<"libros.dtd":Libro>
<"libros.dtd":Titulo>The Art of Programming
Vol.I</"libros.dtd":Titulo>
<"libros.dtd":Autor>
<"libros.dtd":Nombre>
  Donald <"libros.dtd":Apellidos>Knuth</"libros.dtd":Apellidos>
</"libros.dtd":Nombre>
</"libros.dtd":Autor>
```

```
</"libros.dtd":Libro>
</"alumno.dtd":LibrosPrestados>
</"alumno.dtd":Alumno>
```

De esta manera hemos eliminado la ambigüedad y podemos mezclar distintos lenguajes sin modificarlos. Pero a cambio hemos aumentado el tamaño del documento, y hemos reducido su legibilidad para las personas. Este es un problema importante porque uno de los puntos fuertes del XML es su legibilidad.

Para solucionar este problema, el lenguaje permite definir prefijos que luego se expanden a un DTD. La manera de definir prefijos es como atributos en el elemento que se quiera usar el DTD. Para que no haya colisión con otros atributos que pudiera tener ya definidos el usuario, estos prefijos se etiquetan con la cadena `xmlns:`. Una definición de prefijo es válida en todos los subelementos contenidos en este elemento.

```
<a:Alumno xmlns:a="alumno.dtd" xmlns:l="libros.dtd">
<a:Apellidos>Benito Garcia</a:Apellidos>
<a:Nombre>Raul</a:Nombre>
<a:Notas>5</a:Notas>
<a:LibrosPrestados>
<l:Libro>
<l:Titulo>The Art of Programming Vol.I</l:Titulo>
<l:Autor>
<l:Nombre>
  Donald <l:Apellidos>Knuth</l:Apellidos>
</l:Nombre>
</l:Autor>
</l:Libro>
</a:LibrosPrestados>
</a:Alumno>
```

Como se puede ver, el documento se ha vuelto más legible y compacto. Aunque todavía se puede volver un poco más compacto y más parecido al original si usamos el prefijo por defecto. Es decir una manera de especificar a qué DTD pertenecen los elementos sin prefijo. Para hacer esto, se asigna simplemente el DTD por defecto al atributo `xmlns` (esto obliga a no usar nunca el nombre `xmlns` en un atributo para otra cosa distinta, pero parece un problema pequeño para las ventajas obtenidas) y este tendrá validez en este elemento y en todos sus hijos. Usando los prefijos por defecto el ejemplo quedaría:

```
<Alumno xmlns="alumno.dtd" >
<Apellidos>Benito Garcia</Apellidos>
<Nombre>Raul</Nombre>
<Notas>5</Notas>
<LibrosPrestados>
<Libro xmlns="libros.dtd">
<Titulo>The Art of Programming Vol.I</Titulo>
<Autor>
<Nombre>
  Donald <Apellidos>Knuth</Apellidos>
</Nombre>
</Autor>
</Libro>
</LibrosPrestados>
</Alumno>
```

Que es casi igual al documento original sin prefijos.

Cada uno de los prefijos que definen un lenguaje XML, es llamado *namespace* (espacio de nombres).

Ahora imaginemos que es necesario hacer un sistema que interactúe con las distintas bibliotecas de las universidades del país. El problema aquí es que la probabilidad de que algunas bibliotecas hayan elegido el mismo nombre de DTD es alta. Si eso sucediera, el sistema no podría resolver la ambigüedad, y sería inviable.

Para solucionar ese problema se podría pensar en un sistema centralizado que dé nombres de DTD (a la manera del IANA con los OID de SNMP o LDAP). Pero estos sistemas no tienen una buena escalabilidad, y debido a los trámites burocráticos lo más probable es que la gente terminara usando números repetidos, volviendo al problema de partida.

En cambio, a los diseñadores del XML *Namespace* se les ocurrió una idea mejor. Como casi todos los lenguajes XML son utilizados en Internet y la empresa u organismo que los diseña/especifica normalmente tiene ya un dominio Internet, los identificadores, en vez de ser nombres de DTD o números dados por una entidad central, son URL de Internet con *host* el nombre del dominio de quien los diseña. De esta manera se puede asegurar la unicidad de los *Namespaces*.

```
<a:Alumno xmlns:a="http://fi.upm.es/alumnos/1/0"
xmlns:l="http://fi.upm.es/libros/1/0">
…
```

```
</a:Alumno>
```

Estos URL son lógicos y no tienen por qué apuntar a ninguna pagina en un servidor Web (aunque es deseable que el URL sea el DTD de la especificación).

2.5. XML Schema

El XML Schema tiene como objetivo solucionar los mismos problemas que los DTDs, entonces ¿por qué tiene XML dos maneras de solucionar lo mismo?

El principal inconveniente del DTD es ser el mismo mecanismo de definición que usa SGML para definir sus estructuras. Y que su estructura no encaja muy bien con XML, aunque esta definida (más bien heredada) por el estándar. No sigue una estructura de árbol como suelen seguir todos los documentos XML. Pero su mayor problema es que no puede ser ampliado para que entienda el concepto de *Namespaces* manteniendo la compatibilidad hacia atrás con SGML.

Un DTD solo puede definir los componentes de un *Namespace* y no tiene ninguna manera de decir que un elemento tiene subelementos definidos en otro *namespace*. Esta limitación es importante e impide que podamos usar el DTD para definir lenguajes XML de integración.

Es por eso que el w3c diseñó una nueva manera definir estructuras que solucionara el problema de los *namespaces*, de la definición en árbol y ampliara la expresividad de las restricciones respecto al DTD. Está nueva manera de lenguaje XML se denomino XML Schema. A continuación se incluye el ejemplo anterior de `Alumno` pero expresado con XML Schema.

```
<xs:schema targetNamespace="http://fi.upm.es/alumnos/1/0"
 elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:a="http://fi.upm.es/alumnos/1/0">
<xs:element name="Alumno">
 <xs:complexType>
   <xs:sequence>
    <xs:element name="Nombre"  type="xs:string"/>
    <xs:element name="Apellidos" type="xs:string"/>
    <xs:element name="Genero" type="xs:string" minOccurs="0"/>
    <xs:element name="Nota"  maxOccurs="unbounded">
      <xs:complexType>
       <xs:attribute name="tipo" use="required">
```

```
        <xs:simpleType>
            <xs:restriction base="xs:string">
            <xs:enumeration value="media"/>
            <xs:enumeration value="primerCiclo"/>
            <xs:enumeration value="segundoCiclo"/>
          </xs:restriction>
        </xs:simpleType>
         </xs:attribute>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Como se puede ver este XSD (XML Schema Definiton) es equivalente al DTD de `Alumno`. La propia definición es XML y se puede manipular y leer con las herramientas que lean y manipulen XML. No hay que aprender una nueva sintaxis además del XML. Pero como problema, la definición ha aumentado de tamaño.

2.6. Procesamiento XML

Hay diferentes librerías para procesar XML, cada una de ellas asociada o definiendo un "Application Programming Interface" o API de procesamiento. Estos se pueden dividir en dos grandes familias:

1. Centrados en Documento. Esta familia de APIS lee todo documento y presenta al programador un árbol en memoria por el que puede navegar a voluntad. De esta familia el más extendido y estandarizado por w3c es el DOM (*Document Object Model*).

2. Léxicos: Esta familia de APIS va notificando al usuario de cada elemento encontrado en la entrada según va realizando la lectura. Es trabajo de la aplicación de generar la estructura de árbol del documento, si fuera necesario. De esta familia el más extendido es el SAX (*Simple Api for XML*), aunque el STAX (*Streaming API for XML*) esta irrumpiendo con fuerza.

Para ver entre diferencia de los mismos vamos a mostrar varios ejemplos en Java que interprete el documento de `Alumno` con los distintos APIS.

1.1.1. Ejemplo de lectura DOM:

Lo primero que hay que hacer para leer un documento XML con el API de DOM es conseguir una implementación del interfaz `DocumentBuilder`, para eso usamos las instrucciones:

```
DocumentBuilderFactory fac=DocumentBuilderFactory.newInstance();
DocumentBuilder builder=fac.newDocumentBuilder();
```

Estas líneas son necesarias porque en java hay varias implementaciones de distintos proveedores del interfaz DOM. El programador no se tiene que preocupar por qué implementación está instalada en la JVM que está utilizando. Esto se especifica en una variable de configuración del instalador de la máquina virtual. De esta manera, el administrador puede cambiar la implementación de DOM sin tener que recompilar los programas. Pero a cambio el programador debe usar las dos líneas de arriba.

Una vez obtenido el `DocumentBuilder` obtenemos la representación de un fichero en memoria usando esta instrucción:

```
Document doc=builder.parse(new File("alumno.xml"));
```

Después de ejecutar esta línea, el contenido del fichero `alumno.xml` es interpretado por la librería DOM. Y su representación en forma de árbol se puede referenciar por la variable local de tipo `Document` llamada `doc`.

A partir de aquí ya se puede navegar por el árbol DOM a voluntad. Simplemente comentar que la estructura de objetos DOM es análoga a la estructura de objetos XML, todos los objetos heredan de `Node`, y a todos se les puede pedir su siguiente hermano, o se le puede pedir su padre o su primer hijo. Por ejemplo el fichero `alumno.xml`

```
<Alumno xmlns="http://fi.upm.es/alumnos/1/0" >
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
<Nota tipo="media">5</Nota>
</Alumno>
```

Se representa gráficamente en nodos DOM de la siguiente manera:

```
[#document: null] {
```

```
  [Alumno: null] {
    [#text: \r]
    [Nombre: null] {
      [#text: Raul]
    }
    [#text: \r]
    [Apellidos: null] {
      [#text: Benito Garcia]
    }
    [#text: \r]
    [Nota: null] {
      [#text: 5]
    }
    [#text: \r]
  }
}
```

Aquí se ve que el nodo `#document` solo tiene un hijo llamado Alumno que a su vez contiene los demás elementos. Y que los retornos de carro no son ignorados y forman parte del árbol DOM (los *parsers* de Microsoft suelen no representarlo,s en teoría para ayudar al programador, pero el estándar de DOM obliga a su representación y si no estuvieran no se podría implementar por ejemplo el estándar de "canonicalización", que explicaremos más adelante).

```
Node alumnoNode=doc.getFirstChild();
Node nombreNode=alumnoNode.getFirstChild().getNextSibling();
System.out.println("Nombre:"+nombreNode.getTextContent());
Node apellidosNode=nombreNode.getNextSibling().getNextSibling();
System.out.println("Apellidos:"+apellidosNode.getTextContent());
```

En el código de arriba primero se obtiene el nodo `Alumno`. Este es el primer hijo del documento `doc` y se obtiene con `doc.getFirstChild()`. Seguidamente se obtiene el nodo `Nombre`, que es el siguiente hermano del primer hijo del nodo `Alumno` (`getNextSibling().getFirstChild()`). El elemento `Apellidos` se obtiene sabiendo que es el segundo hermano del nodo `Nombre`.

Ahora si el fichero `alumno.xml` cambiara un poco y la añadiéramos comentarios:

```
<!-- Comentario -->

<Alumno xmlns="http://fi.upm.es/alumnos/1/0" >
<Nombre>Raul</Nombre>
<!-- Otro Comentario -->
```

```
<Apellidos>Benito Garcia</Apellidos>
<Nota tipo="media">5</Nota>
</Alumno>
```

El árbol DOM resultante sería distinto y el programa de arriba no obtendría los datos adecuados (más bien fallaría con una excepción porque un nodo comentario no puede contener hijos).

```
[#document: null] {
  [#comment:  Comentario ]
  [Alumno: null] {
    [#text: \r]
    [Nombre: null] {
      [#text: Raul]
    }
    [#text: \r]
    [#comment:  Otro Comentario ]
    [#text: \r]
    [Apellidos: null] {
      [#text: Benito Garcia]
    }
    [#text: \r]
    [Nota: null] {
      [#text: 5]
    }
    [#text: \r]
  }
}
```

Lo primero que muestra la estructura de arriba, es la importancia del nodo `#document` para mantener la estructura de árbol cuando existen comentarios fuera del elemento raíz. También muestra cómo los retornos de carro fuera del elemento raíz se ignoran. Como se ve, el estándar DOM intenta reflejar la estructura sintáctica y léxica (los comentarios, los retornos de carro) a la vez. Pero el coste de mantener cierta fidelidad con el original es que el análisis del documento se hace un poco más complejo.

La manera de ignorar los comentarios contenidos en el nodo #documento, es usar el método de la clase `Document` `getDocumentElement()`, este método dará el elemento raíz. El problema es que no existe un equivalente para los elementos, que nos dé el primer hijo *elemento* o el siguiente hermano *elemento*. Pero se puede hacer un método auxiliar que solucione ese problema.

```
static Node getNextSiblingElement(Node n) {
```

```
      while ((n!=null) && (n.getNodeType()!=Node.ELEMENT_NODE) ) {
            n=n.getNextSibling();
      }
      return n;
}
   …
   Node alumnoNode=doc.getDocumentElement();
   Node nombreNode=getNextSiblingElement(alumnoNode.getFirstChild());
   System.out.println("Nombre:"+nombreNode.getTextContent());
   Node apellidosNode=getNextSiblingElement(nombreNode);
   System.out.println("Apellidos:"+apellidosNode.getTextContent());
```

Este código es mucho más robusto que el anterior (puede leer cualquier documento válido respecto a la estructura definida) y también hemos mejorado un poco su legibilidad.

El ejemplo siguiente es una clase ejecutable de java que implementa la misma funcionalidad que el código presentado anteriormente, pero de una manera ligeramente distinta.

```
package lectura;

import java.io.File;

import javax.xml.XMLConstants;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class LecturaDom {
      final static String ALUMNO_NS="http://fi.upm.es/alumnos/1/0";
      final static String LIBRO_NS="http://fi.upm.es/libros/1/0";
  public static void main(String[] args) throws Exception {
        DocumentBuilderFactory
fac=DocumentBuilderFactory.newInstance();
        fac.setNamespaceAware(true);
        SchemaFactory schemaFactory =

SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Source schemaSource =
                 new StreamSource(new File("alumno.xsd"));
        Schema schema = schemaFactory.newSchema(schemaSource);
        fac.setSchema(schema);
        DocumentBuilder builder=fac.newDocumentBuilder();
        Document doc=builder.parse(new File("alumno.xml"));
```

```
        NodeList alumnos=doc.getElementsByTagNameNS(ALUMNO_NS,
"Alumno");
        for (int i=0;i<alumnos.getLength();i++) {
              Element alumno=(Element) alumnos.item(i);
              String
nombre=alumno.getElementsByTagNameNS(ALUMNO_NS,"Nombre").item(0).getTe
xtContent();
              String
apellidos=alumno.getElementsByTagNameNS(ALUMNO_NS,"Apellidos").item(0)
.getTextContent();
              System.out.println("Nombre:"+nombre+ "
Apellidos:"+apellidos);
              System.out.println("Libros
prestados:"+alumno.getElementsByTagNameNS(LIBRO_NS,"Libro").getLength(
));
        }
  }
}
```

Una diferencia a recalcar es que en ésta implementación estamos obligando la verificación respecto a un esquema, las líneas que indican al API de procesado eso son las siguientes:

```
        fac.setNamespaceAware(true);
        SchemaFactory schemaFactory =

SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Source schemaSource =
                 new StreamSource(new File("alumno.xsd"));
        Schema schema = schemaFactory.newSchema(schemaSource);
        fac.setSchema(schema);
```

La necesidad del `SchemaFactory` es que el API de DOM esta preparado para el uso de otros lenguajes de representación de estructuras como el DTD o para adecuarse a nuevos lenguajes cuando estos se definan.

1.1.2. Ejemplo de lectura SAX

El siguiente API que vamos a ver tiene un enfoque radicalmente distinto al DOM. Mientras el DOM estaba más orientado a la estructura del documento XML, el SAX va notificando al usuario sobre la aparición de elementos léxicos del XML (apertura de una etiqueta de inicio, cierre de un elemento, etc.). Otra diferencia importante es que en el DOM el usuario dirigía la ejecución. En el SAX la propia librería controla la ejecución e invoca a métodos implementados por el usuario para notificarle de la aparición de los elementos léxicos.

El código siguiente genera la misma salida que el equivalente DOM. Pero su explicación es un poco más compleja.

```
package lectura;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.*;
import org.xml.sax.helpers.*;


public class LecturaSax {
      static class AlunnoSaxReader extends DefaultHandler {
        final static String ALUMNO_NS="http://fi.upm.es/alumnos/1/0";
        final static String LIBRO_NS="http://fi.upm.es/libros/1/0";

        String nombre;
        String apellidos;
        boolean leyendoNombre=false;
        boolean leyendoApellidos=false;

        public void startElement(String namespaceURI, String
localName,
                                String qName, Attributes atts) {
          if (ALUMNO_NS.equals(namespaceURI) &&
"Nombre".equals(localName)) {
             leyendoNombre=true;
          }
          if (ALUMNO_NS.equals(namespaceURI) &&
"Apellidos".equals(localName)) {
                   leyendoApellidos=true;
            }
        }
        public void endElement(String uri, String localName, String
name)
                  throws SAXException {
            leyendoApellidos=false;
            leyendoNombre=false;
        }
        public void characters(char[] ch, int start, int length)
                  throws SAXException {
              if (leyendoApellidos)
                    apellidos=new String(ch,start,length);
              if (leyendoNombre)
                    nombre=new String(ch,start,length);
        }
      }

      public static void main(String[] args) throws Exception {
          AlunnoSaxReader al=new AlunnoSaxReader();
          SAXParserFactory factory = SAXParserFactory.newInstance();
          factory.setNamespaceAware(true);
          SAXParser  saxParser = factory.newSAXParser();
          saxParser.parse("alumno.xml",al);
```

```
            System.out.println("Nombre:" + al.nombre+ "
Apellidos:"+al.apellidos);
        }
}
```

Primero la librería de SAX necesita un objeto de una clase que implemente un interfaz obligado por ella. Ese interfaz define entre otros los siguientes métodos:

- `public void startElement(String namespaceURI, String localName, String qName, Attributes atts)`

  La librería se compromete a llamar a este método siempre que se encuentre una etiqueta de inicio de elemento. En nuestro caso lo usamos para detectar cuando comienza el elemento `Nombre` o el elemento `Apellidos`.

- `public void endElement(String uri, String localName, String name) throws SAXException`

  La librería se compromete a llamar a este método cada vez que se encuentre una etiqueta de fin de elemento. En nuestro caso lo usamos para marcar como terminado `Nombre` y `Apellidos` (como no están anidados se pueden cerrar a la vez. Pero si el esquema los hubiera definido como anidados, este método necesitaría mas lógica de tratamiento).

- `public void characters(char[] ch, int start, int length) throws SAXException`

  La librería se compromete a llamar a este método siempre que se encuentre texto en la entrada. Este método es el que realmente realiza el trabajo en nuestro ejemplo. Así si estuviéramos en el elemento `Nombre` (por que la variable booleana `leyendoNombre` es cierta) almacenamos los caracteres en el campo del objeto nombre. Análogamente hacemos el mismo procesamiento con el elemento `Apellidos`.

Una vez que hemos codificado una clase que implemente estos métodos hay que decirle a la librería de SAX qué documento leer y a qué clase notificar los eventos. Esto se hace con las líneas:

```
AlunnoSaxReader al=new AlunnoSaxReader();
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser  saxParser = factory.newSAXParser();
saxParser.parse("alumno.xml",al);
```

Aquí hemos creado el objeto que implementa el interfaz (`al`) y hemos creado un analizador SAX siguiendo pasos análogos a los que hicimos en el caso DOM. Y por último le hemos dicho qué fichero analizar y qué objeto debe recibir la notificación de los eventos.

Como se puede ver, el equivalente SAX es más grande y complicado que el equivalente DOM.

Esto es debido a que, al ser notificado por eventos, el programador no puede confiar ni en la pila ni en el contador de programa para llevar la cuenta de lo que lleva realizado hasta ahora. Cada función se invoca de nuevo y si necesita un comportamiento distinto dependiendo de lo que haya pasado anteriormente en la ejecución tiene que guardar variables globales de estado, para que estas modifiquen el comportamiento de futuras llamadas.

Esta manera de programación es complicada y propensa a errores, por lo que el uso de SAX para análisis de ficheros XML está poco extendido, y solo es usado en dominios donde el uso de memoria o la velocidad de proceso son primordiales. Por esto último, actualmente el SAX esta siendo muy utilizado para la programación de teléfonos móviles que tienen la máquina virtual java edición móviles (j2me).

1.1.3. Ejemplo de lectura STAX

Mientras el SAX tiene unas características de memoria y velocidad interesantes su API es tan incomodo que lo invalida para cualquier aplicación que realmente no tenga otro remedio que pagar el precio de la legibilidad y mantenibilidad por esas características.

El STAX se diseñó como un compromiso entre la "facilidad" de programación del DOM y la velocidad del SAX. Usa también la idea de eventos léxicos, pero en vez de ser la librería la que notifica a la aplicación, es la aplicación la que pide los eventos (a la manera que hacen los compiladores con un analizador léxico cuando le piden el siguiente *token*).

Otra analogía parecida sería la de cursores. El STAX da un cursor de solo avance en el documento XML y es la aplicación la responsable de decir cuándo se avanza este cursor.

El código siguiente es análogo al mostrado en la creación de un analizador en DOM y SAX, obtiene una implementación de STAX a la que indicamos que el fichero a analizar es `alumno.xml`

```
XMLInputFactory im=XMLInputFactory.newInstance();
im.setProperty("javax.xml.stream.supportDTD", new Boolean(false));
XMLStreamReader reader=im.createXMLStreamReader(
      new FileInputStream("alumno.xml"));
```

En este momento la variable `reader` se comporta como un curso y que se inicializa en el principio del documento.

En el código siguiente se pregunta al cursor sobre el tipo de evento al que está apuntando (llamando al método `getEventType()`). Si este es de tipo inicio de un elemento, se le pregunta a que *namespace* pertenece (`getNamespaceURI()`). Si este es distinto del de alumnos se ignora. Si sucede que es igual, se le pregunta el nombre del elemento (`getLocalName()`). Si éste es `Nombre` se hace avanzar el cursor (`next()`) y simplemente se le pide el contenido del nodo texto y se guarda en la variable previamente definida `nombre`.

```
if (reader.getEventType()==XMLStreamReader.START_ELEMENT) {
  if (!ALUMNO_NS.equals(reader.getNamespaceURI()))
      continue;
   if ("Nombre".equals(reader.getLocalName())) {
     reader.next();
     nombre=reader.getText();
    continue;
 }
```

El ejemplo completo de lectura del elemento se puede ver arriba.

```
package lectura;
```

```
import java.io.FileInputStream;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;


public class LecturaStax {
   final static String ALUMNO_NS="http://fi.upm.es/alumnos/1/0";
   final static String LIBRO_NS="http://fi.upm.es/libros/1/0";
   public static void main(String[] args) throws Exception {
         XMLInputFactory im=XMLInputFactory.newInstance();
         im.setProperty("javax.xml.stream.supportDTD", new
Boolean(false));
         XMLStreamReader reader=im.createXMLStreamReader(new
FileInputStream("alumno.xml"));
         String nombre=null;
         String apellidos=null;
         do {
            if (reader.getEventType()==XMLStreamReader.START_ELEMENT) {
                  if (!ALUMNO_NS.equals(reader.getNamespaceURI()))
                              continue;
                  if ("Nombre".equals(reader.getLocalName())) {
                              reader.next();
                              nombre=reader.getText();
                              continue;
                  }
                  if ("Apellidos".equals(reader.getLocalName())) {
                              reader.next();
                              apellidos=reader.getText();
                              continue;
                  }
            }
            reader.next();
      } while ((reader.getEventType())!=XMLStreamReader.END_DOCUMENT);
      System.out.println("Nombre: "+nombre+" Apellidos:"+apellidos);
   }
}
```

Salta a la vista que el código de este ejemplo es más sencillo que el código equivalente de SAX.

Pero sigue manteniendo las características de SAX de rapidez y frugalidad en el consumo de memoria.

3. Firmas XML

3.1. Conversión a forma cánonica

Como ya hemos dicho, el sistema de firma XML se basa en el sistema de firma plana. Imaginemos que queremos firmar este documento

```
<prueba/>
```

Su firma digital sería la siguiente:


```
iD8DBQFFXyDmt3ozLmAUGPQRAlA+AKCGb2FTpO2SDDWe/6g5np8+FpVOjwCgnSAr
RLiwZJTtsSFXEGfTIty84BU=
=IqiI
```


Pero hemos visto que ese documento también se podría escribir como:

```
<prueba></prueba>
```

Y la firma digital del mismo sería ahora:


```
iD8DBQFFXyFvt3ozLmAUGPQRAnAHAKCE/rWDxk/O3SD4sfQbQekYIUU/xgCfXSwX
rLFI9DyC9gyfIo6rTmxsXi8=
=DD6p
```


Como se ve, la firma es distinta. Esto es razonable ya que estamos usando un firmado textual y el primer y segundo textos son diferentes. Pero a nivel XML representan lo mismo y deberían dar el mismo resultado.

Sobre estas premisas se define la conversión a forma canónica o "canonicalización[2]" de XML. Esta consiste en dar unas determinadas reglas para definir entre todas las representaciones textuales posibles de un árbol XML una única que es la que se firma. Esto es análogo a la manera de definir una base de vectores canónica en el álgebra, donde las reglas son que el producto escalar dos a dos sea 0 y que la longitud de cada uno de ellos sea 1.

Las reglas más importantes son las siguientes:

---

[2] Vocablo no aceptado por la RAE pero ampliamente usado en el contexto de lenguajes XML. Proviene del inglés "*canonicalization*" y se suele encontrar abreviado como C14N.

1. Manejo de los espacios: los espacios dentro de las etiquetas de inicio o fin no aportan diferencias al documento. Por lo tanto todos los espacios dentro de las etiquetas se suprimen salvo el que separa (y solo uno) la etiqueta de los atributos y los atributos entre ellos. Consideremos otro ejemplo:

```
<  prueba  arg   =  "2"   arg1="3">  E s to es < / prueba >
```

Su representación canónica es:

```
<prueba arg="2" arg1="3">  E s to es </prueba>
```

Nótese como los espacios del texto no son modificados ya que tienen significado.

2. Etiquetas: El orden de los atributos dentro de una etiqueta no es significativo por lo tanto se elige el orden lexicográfico de los nombre de atributos (expandiendo los prefijos de los *namespace* a su URL si lo tuviera) y se escriben en ese orden. Las definiciones de *namespaces* van antes que los atributos normales y por orden lexicográfico de su URL. La forma compacta de elementos sin contenido se expande a la forma normal de etiqueta de inicio, etiqueta de fin. Véase el siguiente ejemplo:

```
<doc>
  <prueba nombre="4" id="3"/>
  <prueba2 a:attr="1" b:attr="2" xmlns:a="http://www.w3.org"
xmlns:b="http://www.ietf.org"/>
</doc>
```

Su representación canónica es:

```
<doc>
  <prueba id="3" nombre="4"></prueba>
  <prueba2 xmlns:b="http://www.ietf.org"
xmlns:a="http://www.w3.org" b:attr="2" a:attr="1"/>
</doc>
```

3. La repetición de la información de *namespaces* no es importante y se puede obviar. Por ejemplo:

```
<a:Alumno xmlns:a="http://fi.upm.es/alumnos/1/0" xmlns:l="
http://fi.upm.es/libros/1/0">
<a:Apellidos/>
<a:LibrosPrestados xmlns:a="http://fi.upm.es/alumnos/1/0" />
<l:Libro xmlns:l="http://fi.upm.es/libros/1/0"/>
</a:Alumno>
```

Es equivalente a este convertido en su forma canónica:

```
<a:Alumno xmlns:a="http://fi.upm.es/alumnos/1/0" xmlns:l="
http://fi.upm.es/libros/1/0">
<a:Apellidos/>
<a:LibrosPrestados/>
<l:Libro/>
</a:Alumno>
```

Y también es equivalente a esta otra versión:

```
<a:Alumno xmlns:a="http://fi.upm.es/alumnos/1/0">
<a:Apellidos/>
<a:LibrosPrestados/>
<l:Libro xmlns:l="http://fi.upm.es/libros/1/0"/>
</a:Alumno>
```

En la primera transformación lo regla que se sigue es: si un *namespace* es definido en algún padre, este no tiene que volverse a repetir en un hijo.

En la segunda conversión la regla seguida es un poco más compleja, la definición de *namespaces* solo se puede dar en un nodo donde se utilice esa definición pero siempre que no este definida ya en un nodo padre. En el caso de arriba la definición de *namespace* `xmlns:l` no puede estar en el elemento `a:Alumno` por que éste no lo utiliza. En cambio estará en el elemento `l:Libro` pero no se volverá a repetir en ninguno de sus hijos (si los tuviera).

Esta diferencia es importante y ha dado lugar a dos estándares de normalización ("canonicalización") que difieren solamente en este punto. Al primer comportamiento primero se le llama canonicalización inclusiva [2] (y fue la primera estandarizada), por contraste con el segundo, llamado canonicalización exclusiva [3] donde los prefijos no utilizados son excluidos y no representados.

Más adelante veremos por que no era suficiente la canonicalización inclusiva y tuvo que definirse la exclusiva.

3.2. Firmas XML

Como hemos visto el XML es más que texto, es una estructura, y se pueden construir lenguajes sobre él. Por ello no sería suficiente un calco directo de las firmas digitales textuales para muchos dominios. Por eso el estándar es más complicado que la simple aproximación que presentamos en la introducción.

La primera mejora respecto a las firmas digitales tradicionales es que se permite firmar parte del documento (los algoritmos tradicionales trabajan solo el documento completo). Pero esta selección se lleva incluso a refinamientos que permiten desde elegir un elemento, a elegir solo los elementos o atributos que cumplan una determinada condición (como que tengan el atributo x con valor z, o no tengan hijos, o sean hijos de elementos con tal o cual condición).

Esta flexibilidad permite que convivan varias firmas en el mismo documento (con firmantes distintos y sobre partes distintas o a veces sobre la misma).

Técnicamente hablando el estándar de firmas define un lenguaje XML, con una estructura y una semántica de procesamiento.

Lo que sigue es una descripción de los puntos más importantes del estándar orientándola desde la estructura del XML.

La taxonomía de una firma digital es la siguiente:

```
<Signature>
  <SignedInfo>
    <SignatureMethod/>
    <CanonicalizationMethod/>
    <Reference>
       <Transforms/>
       <DigestMethod/>
       <DigestValue/>
    </Reference>
    <Reference/>*
  </SignedInfo>
  <SignatureValue/>
  <KeyInfo>?
  <Object>*
</Signature>
```

A continuación se explica cada uno de los elementos/componentes de esa firma, en un orden adecuado para su comprensión:

1. `Reference`

El elemento `Reference` o referencia es la parte más importante de la firma, determina lo que se está firmando y cómo se está haciendo. Dentro del elemento `Signature` puede haber más de una referencia.

El elemento referencia tiene un atributo URI, que es donde se especifica el elemento que se va firmar. Este URI puede apuntar fuera del documento, con lo que tendrá que ser un URI absoluto.

```
    <Reference URI="http://www.w3.org/2000/06/interop-
pressrelease">
    </Reference>
```

O puede ser simplemente el propio documento, con lo que será simplemente la cadena vacía.

```
<doc>
<Reference URI=""/>
</doc>
```

También puede ser un fragmento (un elemento) del documento actual, con lo que empezara con un carácter #.

```
<doc>
  <Signature>
   …
   <Reference URI="#id">
   </Reference>
  </Signature>
  <elemen id="id"/>
</doc>
```

Dentro del elemento `Reference` se encuentra el elemento `Transforms`, que engloba operaciones que se van a ejecutar sobre el documento o fragmento seleccionado por el atributo URI.

```
<doc>
…
<Reference URI="">
<Transforms>
   <Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"/>
   <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315"/>
</Transforms>
</Reference>
…
</doc>
```

En el ejemplo anterior se ha selecciona todo el documento (`URI=""`), y se va a transformar primero quitándole la firma (el `Algorithm= “http://www.w3.org/2000/09/xmldsig#enveloped-`

`signature"`). El resultado de esa transformación se normalizará con el método inclusivo (`Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"`).

Estas transformaciones son las más comunes y necesarias. Pero el estándar define algunas más, y las aplicaciones pueden ampliarlas con transformaciones propietarias.

El siguiente campo de la `Reference` es el `DigestMethod`. Este campo determina cómo se va a hacer el resumen con lo transformado hasta ahora. Este resumen es una función *hash* de calidad criptográfica[3] que dará lugar a un número resumen de longitud finita (actualmente 1024bits, aunque es previsible que aumente en el futuro). El resultado de esta transformación se guardará en el siguiente campo `DigestValue` codificado en *Base64*[4]

```
<doc>
…
<Reference URI="">
<Transforms>
   <Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"/>
   <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315"/>
</Transforms>
  <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
</Reference>
…
</doc>
```

2. `SignedInfo`

---

[3] Una función Hash de calidad criptográfica tiene unas restricciones más fuertes respecto a las colisiones que una función Hash normal: la transformación de dos elementos cercanos debe generar claves que no sean cercanas.

[4] El Base64 es una codificación que transforma cualquier grupo de 3 bytes en 4 caracteres ASCII imprimibles (a-z a-Z 0-9,+,=).

El `SignedInfo` contiene uno o varios elementos referencia, y es la parte realmente firmada de la firma digital XML (ya que las referencias solo se resumen).

Dentro de ella tiene elementos para especificar cómo se firma: cómo se normaliza (c14n) el elemento `SignedInfo` (dentro del elemento `CanonicalizationMethod`), con qué algoritmo se firma lo normalizado (c14n) (dentro del elemento `SignatureMethod`).

3. SignedValue

El resultado de las operaciones anteriores se encuentra codificado en *Base64* fuera del elemento `SignedInfo` y dentro del elemento llamado `SignatureValue`.

```
<doc>
…
<Signature>
<SignedInfo>
  <CanonicalizationMethod
 Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
 Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
  <Reference URI="">
       <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-
xml-c14n-20010315"/>
       </Transforms>
       <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
       <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
   </Reference>
</SignedInfo>
<SignatureValue>MC0=……</SignatureValue>
</Signature>
…
</doc>
```

Los demás elementos del elemento `Signature` contienen información adicional para el programa/usuario que va a procesar la firma, pero no son necesarios para la creación o verificación de la misma.

Uno de estos campos, el `KeyInfo` suele rellenarse y contiene la clave pública (o el certificado) usada para realizar la firma. El verificador debe tener alguna manera de confiar en esa clave pública, o conocer la de antemano y comprobar que es igual.

Si lo que se incluye es un certificado, el problema de confiar en él sigue siendo también del receptor, aunque por lo menos tiene la ayuda de confiar en alguna de las entidades certificadoras que avalan la procedencia de la firma.

3.3. Ejemplo de Firma

Como ejemplo de firmado imaginemos que queremos firmar totalmente el siguiente documento:

```
<doc >
 Quiero ser firmado.
 <Signature/>
</doc>
```

La referencia que vamos a crear será con URI="", y como transformaciones, la normalización (c14n) inclusiva y el descartado de la firma (porque la firma esta incluida dentro del documento). El algoritmo de resumen no nos importa por lo que elegiremos un estándar *sha1* [17]. La firma se encontrará emplazada donde se encuentra el elemento vacío firma.

Como parámetros para el elemento `SignedInfo` elegimos el algoritmo de firmado *rsa-sha1* [18] y la normalización inclusiva.

Con estos datos, la clave privada y el documento anterior se lo indicamos a la librería de firmas para que cree la firma XML.

Lo primero que hace la librería de firmas es calcular el resumen de todas las referencias. Para eso busca el documento a resumir, en este caso el de arriba. Y se lo pasa una a una a las transformaciones.

La primera transformación quita la firma que estamos procesando, en este ejemplo solo hemos añadido el esqueleto de la firma al documento con lo que solo tiene que quitarlo del documento. Con ello el resultado hasta ahora es el siguiente:

```
<doc >
 Quiero ser firmado.
```

```
</doc>
```

Este resultado se lo pasamos a la siguiente transformación que es una normalización (c14n) inclusiva (en este caso da igual el tipo de normalización (c14n) elegida porque no hay ningún *namespace* en juego). El resultado después de la transformación es el siguiente:

```
<doc>
 Quiero ser firmado.
</doc>
```

Este texto se manda a la función *hash,* que dará un número de 1024 bytes que será convertido a *Base64*. Con esto tenemos todos los datos para generar el elemento `Reference`, cuyo contenido es:

```
  <Reference URI="">
       <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-
xml-c14n-20010315"/>
       </Transforms>
       <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
       <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
</Reference>
```

Como es la última referencia, la introducimos con los demás datos proporcionados por el usuario en la estructura `SignedInfo`

```
<SignedInfo>
  <CanonicalizationMethod
 Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
 Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
  <Reference URI="">
       <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-
xml-c14n-20010315"/>
       </Transforms>
       <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
       <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
   </Reference>
</SignedInfo>
```

Ahora tenemos toda la información para generar la firma. Este elemento `SignedInfo` se normaliza (c14n) con el algoritmo elegido (en este momento debemos tener en cuenta en qué documento y en qué parte de este se va a incluir la firma para obtener los *namespaces* definidos en él).

La normalización (c14n) en este ejemplo es igual al documento de entrada. La salida se le pasa al firmador con la clave privada, éste lo firmará y generará un número resumen que será codificado en Base64.

Ahora el documento con su firma XML incluida quedará:

```
<doc >
 Quiero ser firmado.
 <Signature>
<SignedInfo>
  <CanonicalizationMethod
 Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
 Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
  <Reference URI="”>
       <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature”/>
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-
xml-c14n-20010315"/>
       </Transforms>
       <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
       <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
   </Reference>
</SignedInfo>
<SignatureValue>MC0=……</SignatureValue>
</Signature>
</doc>
```

Este es el resultado firmado que le enviamos al receptor del documento.

3.4. Ejemplo de Verificación

Imaginemos que un destinatario recibe la firma del ejemplo anterior, que ya posee la clave pública del firmante y que quiere verificar la autenticidad.

Lo primero que busca el verificador es el elemento `Signature` que pasará a la librería de firmas con la clave pública del firmante. La librería de firmas primero comprobará que las referencias son correctas.

Para eso, empezará obteniendo el documento o fragmento referenciado por el atributo URI, que en este caso coincide con todo el documento. Este se lo pasará a las transformaciones para que hagan su trabajo.

La primera transformación quitará la firma actualmente analizada, con lo que el documento se convierte en:

```
<doc >
 Quiero ser firmado.
</doc>
```

Aquí se ve la importancia de esta transformación. Si no se hubiera especificado el documento incluiría la firma, y habría una referencia circular: se intentaría verificar la firma de un documento que está modificado por la propia firma.

La siguiente transformación normalizada (c14n) el documento XML de arriba, quedando:

```
<doc>
 Quiero ser firmado.
</doc>
```

Esto se resumirá con el algoritmo especificado en el `DigestMethod`, y su resultado se comprobará con el que el firmante ha mandado en el elemento `DigestValue`. Si es igual, la referencia es correcta y se sigue verificando la siguiente referencia. Si no es igual se da un error al usuario informándole de que la firma es incorrecta. En nuestro caso, como se puede ver, el resultado de las transformaciones es idéntico al que calculó el firmante, con lo que la referencia es correcta.

Si se ha acabado con las referencias, se sigue verificando el elemento `SignedInfo`. En este caso el `SignedInfo` se normaliza (c14n) como se especifica en el `CanonicalizationMethod`, con lo que se obtendría lo siguiente:

```
<SignedInfo>
  <CanonicalizationMethod
 Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
 Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
  <Reference URI="">
       <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/
xmldsig#enveloped-signature"/>
```

```
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-
xml-c14n-20010315"/>
        </Transforms>
        <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
    </Reference>
</SignedInfo>
```

Este texto, el número resumen que está codificado en *Base64* en el elemento `SignatureValue` y la clave pública, se le pasan al algoritmo de firmas especificado en `SigntaureMethod`. El algoritmo de verificación dirá si el mensaje es auténtico. En este caso dirá que es correcto, y la librería de firmas devolverá el control al usuario diciéndole que la firma es auténtica y las referencias no han sido modificadas.

3.5. Preguntas/Ataques

Como se ve, el estándar de firmas XML no firma las referencias sino que solo las resume. Lo único que está realmente firmado es el `SignedInfo`. ¿Podría ser esto un agujero de seguridad?

Al principio podría parecer que se podrían cambiar las referencias, calcular un nuevo *hash*, modificar el `DigestValue` con el nuevo valor, y esto confundiría a la librería de firmas, que diría que las referencias son correctas aunque han sido modificadas. Lo que pasa es que ese cambio del `DigestValue` hará que la firma del `SignedInfo` falle (ya que el `DigestValue` está dentro del `SignedInfo`). Con ello este ataque es imposible (hablando con más precisión improbable).

Lo que si puede pasar con el estándar es que al permitir transformaciones, lo que realmente se esté firmando no sea tan inclusivo como lo que el remitente o el receptor piensan. Y un atacante puede explotar esto introduciendo nuevos campos que las transformaciones hacen ignorar, pero que la lógica de aplicación interpreta. Esto se puede solucionar presentándole al remitente las referencias después de las transformaciones o usando transformaciones más sencillas y fáciles de comprobar.

3.6. Tipos de firma

Como hemos visto, una firma XML y sus referencias pueden pertenecer a documentos distintos. Pero si la firma y la referencia están en el mismo documento, la relación espacial entre ellas es relevante.

La posición que ocupa la firma respecto al fragmento que se está referenciando es un aspecto importante de una firma (como hemos visto en la introducción). Hasta ahora los ejemplos presentados son de las denominadas firmas *enveloped* (envueltas). Pero hay otros tipos de firmas, tal como se explica a continuación:

- *Enveloped*: La firma es un elemento hijo o nieto, del elemento a firmar. Como hemos visto estas firmas deben de tener siempre la transformación enveloped-signature(`http://www.w3.org/2000/09/xmldsig#enveloped-signature`) para poder ser verificadas.

```
<doc >
 <el id="1">
   <subel>
   <subel>
     <subsubel>
         <Signature>
            <SignedInfo>
               …
              <Reference URI="#1">
                <Transforms>
                 <Transform Algorithm="…enveloped-signature"/>
                </Transforms>
              </Reference>
            </SignedInfo>
        </Signature>
     </subsubel>
   <subel>
 </el id="1">
</doc>
```

- *Enveloping*: Envuelta. La firma es padre o abuelo del elemento a firmar. Para eso, el estándar de firmas ha reservado un elemento `Object` dentro de la firma, que se puede usar tanto para expandir el estándar con nuevos conceptos no previstos a la hora de definir el estándar o para incluir el elemento a firmar en este tipo de firmas.

```
<doc>
```

```
  <Signature>
    <SignedInfo>
      …
    <Reference URI="#1">
     <Transforms>
     </Transforms>
    </Reference>
   </SignedInfo>
   <Object>
     <el id="#1">
     …
     </el>
   </Object>
  </Signature>

</doc>
```

- *Detached*: La firma no contiene ni es contenida por el elemento firmado. La literatura no refina más la especificación de estas firmas, pero como veremos más adelante, para el problema de firmado de documentos arbitrariamente grandes es importante refinar un poco más esta definición. Como ejemplo de firma *detached* tenemos, la siguiente, en que la firma se encuentra antes que el elemento referenciado:

```
<doc>
  <Signature>
    <SignedInfo>
      …
    <Reference URI="#1">
     <Transforms>
     </Transforms>
    </Reference>
   </SignedInfo>
  </Signature>
  <el id="#1">
     …
   </el>
</doc>
```

En el siguiente ejemplo de firma *detached* la firma se encuentra después del elemento referenciado:

```
<doc>
  <el id="#1">
     …
  </el>
  <Signature>
    <SignedInfo>
```

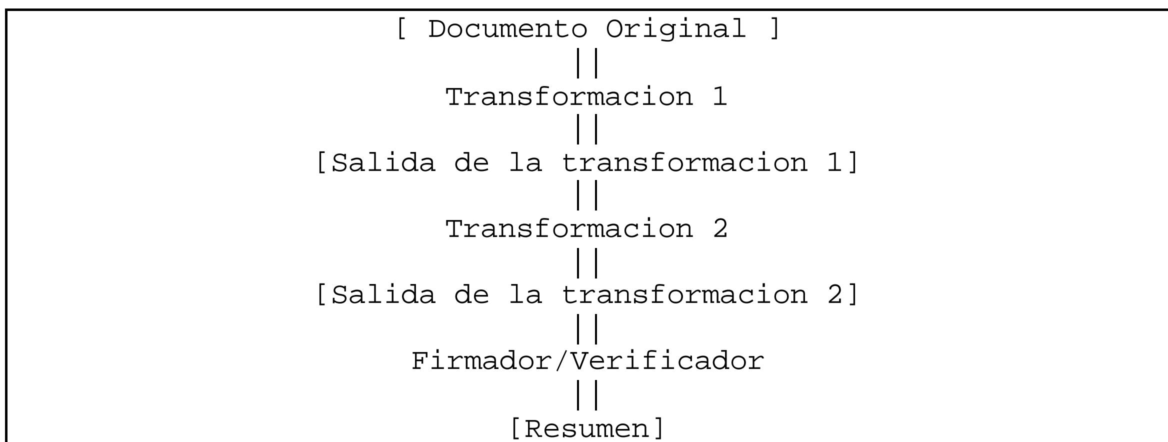
```
      …
    <Reference URI="#1">
     <Transforms>
     </Transforms>
    </Reference>
   </SignedInfo>
  </Signature>
</doc>
```

Esta clasificación será importante cuando expliquemos el verificado en el siguiente capitulo el método *streaming* de verificación.

4. Firma de documentos tamaño arbitrario

Hasta la realización del trabajo descrito en esta memoria todas las implementaciones de librerías de firma digital siguían el flujo de firma/verificación arriba explicado. Este tipo de arquitecturas se denominan *pipeleline* por su similitud con una cadena de montaje industrial (*pipeline* en inglés)

En un *pipeline* se inyecta el documento original al inicio y cada etapa del *pipeline* trabaja sobre la salida de la etapa anterior. La última etapa le pasa la información textual al firmador/verificador.



Esta forma de modelar el proceso de firma es sencillo, extensible y hace las etapas independientes unas de otras, lo que hace que sean más fáciles de programar y enlazar, pero tiene el problema de necesitar mucha memoria: Normalmente tres veces la memoria ocupada por el documento original. Esto es debido a que durante el procesamiento de una etapa cualquiera del *pipeline* hay que mantener el documento original (la aplicación está trabajando con él), la salida de la transformación anterior a la que se está ejecutando, y la memoria de trabajo, que se convertirá en la salida del transformación actual. Si solo hubiera una transformación solo habría que mantener el documento original y la memoria de trabajo, pero es poco común encontrarse firmas con esta característica.

Una posible mejora a este patrón se podría obtener si que las etapas fueran emitiendo resultados intermedios para reducir los requisitos de memoria (que frecuentemente denominaremos "presión de memoria"). Pero para hacer esto habría que

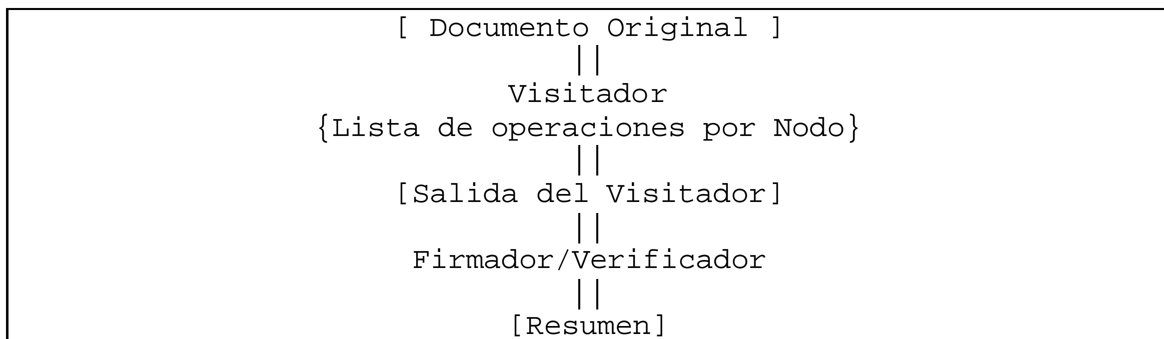
definir un protocolo de comunicación más complicado que el de entrada/salida. Otra desventaja de este método es que si algún verificador debajo de la cadena determina que no le interesa un subárbol completo, seria interesente que todos los elementos anteriores dejaran de trabajar en él, y pasaran al siguiente. Esto complicaría más aún el API de comunicaciones del *pipeline*.

Una forma de solucionar este problema es usar una mezcla entre el patrón de diseño visitador/estrategia.

Primero se analizan todas las transformaciones y se recopila una lista de acciones a realizar con cada nodo y sus hijos. Las acciones son del tipo:

- Ignora este nodo y sus hijos. Este es el tipo de acciones que implementaría el transformador `enveloped-signature`: si el elemento visitado es el nodo de firma actual ignorar de la canonicalización este nodo y sus hijos.
- Ignora este nodo pero no sus hijos. Este tipo de acciones se usan para implementar las transformaciones filtro (como XPath [8] y XPath2 [9]) que define el estándar de y que sirve para firmar únicamente los nodos que cumplen unas propiedades dadas.
- Ignora este atributo del nodo. Igual propósito que las anteriores pero a nivel de atributo dentro del nodo.
- Escribe X en la memoria de Trabajo. Estas son las transformaciones que implementan la normalización (c14n).

De esta manera, la estructura lógica de proceso quedaría:



Como se ve en esta arquitectura software solo se necesita dos veces la memoria del documento original.

Esto es ya una reducción considerable, pero todavía se puede reducir más si tenemos en cuenta que la salida del Firmador/Verificador de pequeño tamaño (normalmente 1024 bits, como mucho 16 kbits). Con ello estamos haciendo un desperdicio enorme si para firmar un documento de 50 MB estamos gastando otros 50 MB solo para calcular 128 bytes.

Si explotamos la propiedad de que todos los verificadores/firmadores presenta modelos de ejecución Stream, solo guardan un estado resumido del anterior y pueden ser actualizados con nuevos datos, la salida del visitador se podría ir pasando al Firmador/Verificador cada Kilobyte (este valor es arbitrario aunque en la implementación de Apache tiene su significado) borrando el anterior.

De esta manera hemos pasado de gastar 3 veces la memoria del documento para verificarlo, a solo a duplicarlo, y luego a gastar solo unos kilobytes. Hemos convertido el proceso en O(1) en términos de memoria.

Además de reducir el espacio gastado para firmar, aumenta el rendimiento del proceso de firmado.

Esto es debido a que la presión sobre la memoria es mucho menor (recordemos que la memoria y sobre todo su escritura es el cuello de botella de los ordenadores actuales). Pero además, mejora la adaptación a *cache* de la aplicación, al no recorrerse vez tras vez el documento como en el modo *pipeline* (que lo más probable que no quepa en la *cache* de L2), sino que trabaja sobre fragmentos pequeños del documento (que es muy probable quepan en la *cache* L1 de datos). Un inconveniente es que la adaptabilidad del código a la cache del código se reduce y los *stalls* de pipeline de procesador aumentarán, al ser un código altamente polimórfico. De todas maneras esta reducción es altamente compensada por la mejora al acceso de datos. En las pruebas realizadas el rendimiento mejoraba entre un 10% para documentos pequeños (<1 KB) y un orden de magnitud en documentos grandes.

Todos estos cambios han sido incorporados por el autor en la versión 1.2 de la librería de firmas de Apache, aunque hay que subrayar que la implementación que usa actualmente la librería de firmas apache es un poco distinta a la presentada aquí.

En apache hay dos caminos de ejecución:

1. uno en el que solo se ha pedido una canonicalización (y a lo mejor una eliminación de la firma tratada), que simplemente se maneja por el normalizador (c14n) indicandole el nodo a ignorar (si lo hubiera).
2. Otro en el que hay transformaciones más complejas y es donde entra en juego el esquema arriba descrito, pero con la salvedad de que el normalizador (c14n) sigue siendo el visitador y preguntará a las estrategias si hay o no que emitir el nodo en curso.

4.1. Verificador Stream

Con la solución anterior, le memoria consumida es todavía igual a lo que ocupa originalmente el documento. Esta restricción es importante porque limita al tamaño de la memoria física (o virtual) de la maquina, la longitud de los documentos. Una solución mejor sería ir procesando el documento a medida que se va leyendo. De esta manera el consumo de memoria sería lineal y se podrían firmar documentos arbitrariamente grandes.

El problema es que para empezar las transformaciones tenemos que saber cuáles aplicar y sobre qué. Esos datos están contenidos en las referencias y solo en las firmas de tipo *enveloping* y en algunas *detached* aprecen antes que lo firmado. En los otros tipos de firma, cuando hayamos leído la firma y sepamos qué hacer será demasiado tarde para empezar a trabajar.

Esta limitación se puede subsanar si el usuario indica al verificador qué elementos se van a verificar y el tipo de transformaciones a aplicar.

Esta restricción parece importante pero, realmente es más cómoda para el usuario del API, que en el caso anterior debería comprobar realmente qué es lo que se está

firmando y cómo. Con el verificador Stream el usuario simplemente especifica qué tipo de firmas acepta.

4.2. Verificador de Firmas STAX

La librería de firmas XML Security de Apache implementa un API basado en DOM, con lo que es inevitable que el documento esté en memoria (y aumentado más o menos 10 veces, es decir un documento de 1 MB ocupará en DOM cerca de 10 MB), por lo que el API y la forma de trabajo tienen que cambiar ligeramente.

En la versión 1.4 de la librería de firmas, que se incluirá con la máquina virtual Java 1.6, está incluido un nuevo API definido en el JSR105 [10], este API es independiente de la representación XML utilizada y es el que vamos a usar para controlar la nueva implementación.

El API de análisis que vamos a usar es STAX. Este API, como ya hemos visto, presenta el Documento XML como si fuera un simple analizador léxico usado en el análisis de lenguajes.

Nuestra librería de firmas se implementará de modo que sea transparente al usuario, actuando como un filtro al análisis de XML para que el usuario pueda dirigir el análisis del documento como si la librería de firmas no existiera.

En el futuro el usuario podrá decidir que la librería le filtre los eventos relacionados con las firmas, pero ahora el usuario los recibe y es responsable de ignorarlos.

La librería se basa en 3 clases importantes:

1. Los observadores. Son informados por el filtro de aperturas de elementos. Si el observador está interesado en tratar ese elemento generará un trabajador que estará asociado a la vida del elemento y recibirá todos los eventos pertenecientes a ese elemento, así como los pertenecientes a todos sus hijos/nietos hasta que el elemento se cierre.
2. Los trabajadores. Están asociados a un elemento y recibirán todos los eventos relacionados con este. Los trabajadores son notificados cuando el

elemento al que están asociados se cierra. En este momento los trabajadores pueden añadir un nuevo observador al filtro.

3. El filtro. Es el controlador de todas las actividades y recibe los eventos del API de STAX. Notifica a los observadores de la apertura de elementos y a los trabajadores activos de todos los eventos informados por el analizador STAX. Cuando se cierra un elemento notifica y borra los trabajadores asociados a ese elemento.

El código del filtro, tiene esta estructura:

```
public boolean accept(XMLStreamReader cur) {
            int eventType = cur.getEventType();
            if (eventType==XMLStreamReader.START_ELEMENT) {
                  //Start element notify all watcher
                  level++;
                  for (StaxWatcher watcher : watchers) {
                        StaxWorker sf=watcher.watch(cur, this);
                        if (sf!=null) {
                              //Add a new worker
                              filters.add(sf);
                              filterStart.add(level);
                        }
                  }
            }
            List<StaxWorker> added=filters;
            //A worker can add new workers. Iterate while there is more
workers to add.
            while (added.size()!=0) {
                  List<StaxWorker> toAdd=new ArrayList<StaxWorker>();
                  List<Integer> toAddStart=new ArrayList<Integer>();

                  for (StaxWorker filter: added) {
                        StaxWorker sf=filter.read(cur);
                        if (sf!=null) {
                              toAdd.add(sf);
                              toAddStart.add(level);
                        }
                  }
                  added=toAdd;
                  filters.addAll(toAdd);
                  filterStart.addAll(toAddStart);
            }
            if (eventType==XMLStreamReader.END_ELEMENT) {
                  //an end element remove any worker attached to this
element
                  do {
                        int i=filterStart.lastIndexOf(level);
                        if (i!=-1) {
                              StaxWatcher
watch=filters.remove(i).remove();
```

```
                              if (watch!=null) {
                                    watchers.add(watch);
                              }
                              filterStart.remove(i);
                        }
                  } while (filterStart.contains(level));
                  level--;
            }
            return true;
      }
```

Este código se ejecuta por cada evento que reporta el STAX antes de llegar a la aplicación del usuario.

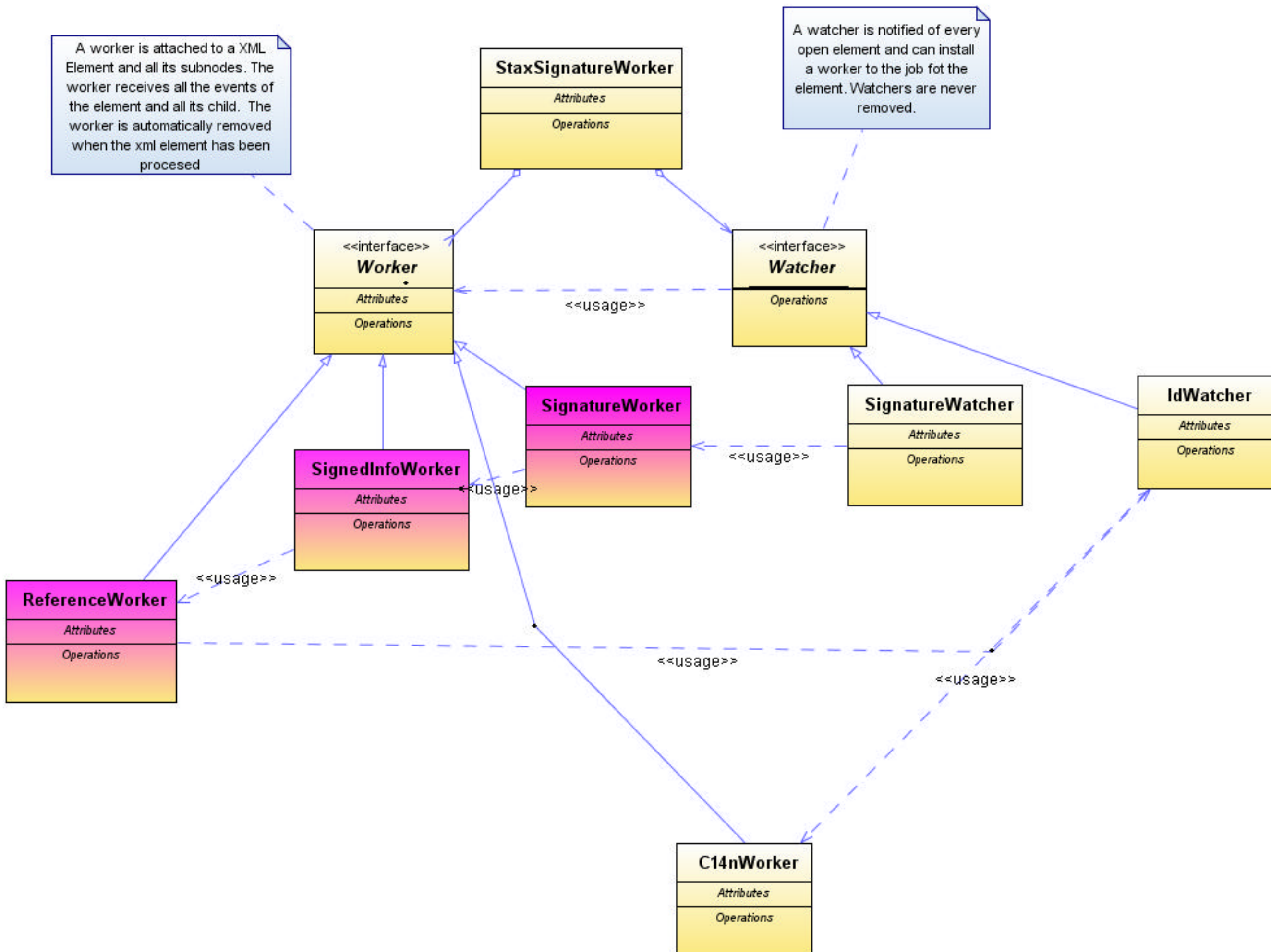
Ahora, eligiendo bien los trabajadores y los observadores podemos verificar sin intervención del usuario si una firma envolvente o una *detached* que aparezca antes que lo referenciado son correctas.

El código para usarlo sería el siguiente:

```
            XMLInputFactory im=XMLInputFactory.newInstance();
            im.setProperty("javax.xml.stream.supportDTD", new
Boolean(false));
            XMLStreamReader reader=im.createXMLStreamReader(<<EL
FICHERO A VERIFICAR>>);
            StaxValidateContext stx =
                  StaxValidateContext.createEnvolopedValidator(reader);
            reader=im.createFilteredReader(reader,
                                    stx.getStreamReader());
            while ((reader.getEventType())
                        !=XMLStreamReader.END_DOCUMENT) {
                  //Hacemos algo con lo que leemos.
                  reader.next();
            }
            XMLSignatureFactory fac=
                   XMLSignatureFactory.getInstance("Stax");
            stx.setSignatureNumber(0);
            XMLSignature sig=fac.unmarshalXMLSignature(stx);
            if (!sig.validate(stx)) {
                  //Firma invalida.
            }
```

Si el usuario quiere verificar otros tipos de firma debería especificar qué elementos quiere verificar con qué transformaciones, etc. Este API todavía no está desarrollado.

4.3. Trabajadores y Observadores.



El esquema de clases que implementa la funcionalidad arriba descrita se desarrolla con los siguientes observadores y trabajadores.

1. `SignatureWatcher`: Está esperando que aparezca el elemento de inicio de la firma digital. Cuando esto surge, crea un `SignatureWorker` para que trabaje sobre la firma.

2. `SignatureWorker`: Analiza la firma y cuando detecta el comienzo del elemento `SignedInfo` crea un `SignedInfoWorker`.

3. `SignedInfoWorker`: Guarda la información del `SignedInfo` y empieza la normalización (c14n) de ese mismo elemento. Si detecta un elemento `Reference` crea un trabajador `ReferenceWorker`.
4. `ReferenceWorker`: El `ReferenceWorker` obtendrá el identificador del fragmento a buscar y las transformaciones a aplicar. Cuando este termine instalará un `IdWatcher` con el fragmento buscado.

Una vez que haya terminado todos los `References` el `SignedInfoWorker` calculará su normalización (c14n).

Una vez que el `SignedInfoWorker` termine, el `SignatureWorker` recibirá el elemento `SignatureValue` este contendrá el número resumen. Este se guarda a la espera de que el usuario nos dé la clave pública con la que verificar. Cuando esto suceda se podrá pedir al verificador que compruebe la validez de la firma.

Pero todavía queda verificar las referencias:

5. El `IdWatcher` espera hasta que aparezca un elemento con el Id especificado. Cuando esto suceda creará un `C14nWorker` (del tipo de normalización pedido por el usuario).
6. El `C14nWorker` trabajará sobre ese elemento, aplicándole todas las transformaciones. Cuando termine, notificará al `ReferenceWorker` que indirectamente lo creó.

Cuando el usuario haya terminado de recorrer el documento le preguntará al Verificador si la firma es válida y le pasará la firma pública que quiere usar. En este momento la librería tendrá todos los elementos para comprobar la firma.

De esta manera hemos verificado una firma sin tener todo el documento en memoria en ningún momento.

Para entender mejor el uso vamos a ver un ejemplo de verificación.

4.4. Ejemplo de Verificación Stream

Imaginemos que tenemos el siguiente documento XML firmado:

```
<RootObject><ds:Signature
xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<ds:SignedInfo>
<ds:CanonicalizationMethod
Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"></ds:CanonicalizationMethod>
<ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-
sha1"></ds:SignatureMethod>
<ds:Reference URI="#1">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"></ds:Transform>
</ds:Transforms>
<ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></ds:DigestMe
thod>
<ds:DigestValue>oMQoFufPA7Un6cfz0GaEOJpE4Z8=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
AhyiFQ6hucykYJOJDBV3wbPBe2TAURXXfCUD7BmSAecT+izT9fHFsxRVez3s+6hY
SgtaVhmeVgbd
ZEOMPFihBGldi1NV73Z/tpXxqNvY+/NwQmmasQp9gzFHxYF2cqi8m7sAHM03BIC1
YoBctxVw/jxV
ClhLJuTSHoKwlzKH24g=
</ds:SignatureValue>
<ds:KeyInfo>
<ds:KeyValue>
<ds:RSAKeyValue>
<ds:Modulus>
skqbW7oBwM1lCWNwC1obkgj4VV58G1AX7ERMWEIrQQlZ8uFdQ3FNkgMdtmx/XUjN
F+zXTDmxe+K/
lne+0KDwLWskqhS6gnkQmxZoR4FUovqRngoqU6bnnn0pM9gF/AI/vcdu7aowbF9S
7TVlSw7IpxIQ
VjevEfohDpn/+oxljm0=
</ds:Modulus>
<ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
</ds:KeyValue>
</ds:KeyInfo>
<ds:Object Id="1">
<Alumno xmlns="http://fi.upm.es/alumnos/1/0" >
<Nombre>Raul</Nombre>
<Apellidos>Benito Garcia</Apellidos>
<Nota tipo="media">5</Nota>
</Alumno>
</ds:Object>
</ds:Signature></RootObject>
```

Hemos incluido en el elemento `KeyInfo` de este documento, que está dentro de la firma, la clave pública usada para firmarlo. De esta manera podemos verificarlo sin necesidad de conseguir la clave por otros medios.

La firma ha sido generada con la librería de firmas DOM de Apache, y como se puede ver es una firma envolvente.

Lo primero que hará el cliente para verificar esta firma es conseguir un objeto que implemente el API STAX como ya hemos explicado anteriormente. Para poder cambiar la implementación sin cambiar el código Java, usa el patrón factoría:

```
XMLInputFactory im=XMLInputFactory.newInstance();
im.setProperty("javax.xml.stream.supportDTD",
      new Boolean(false));
XMLStreamReader reader=im.createXMLStreamReader(
      new FileInputStream(("alumnoEnveloped.xml"));
```

El código siguiente explica cómo el usuario tiene que notificar a la librería de firmas que el cursor STAX de la variable `reader` contiene una firma que se quiere verificar con la clave `key` (el API actualmente no analiza el elemento `KeyInfo` de la firma).

```
StaxValidateContext stx =
StaxValidateContext.createEnvolopedValidator(key,reader);
```

La variable `stx` de tipo `StaxValidateContext` es el contexto de validación que va a usar la librería para referirse a la firma contenida en el documento que apunta el cursor STAX `reader`.

Como hemos comentado, la librería usa un filtro sobre el cursor para enterarse de los eventos que se están leyendo por petición explícita del usuario. Por ejemplo el programa ya desarrollado para la lectura del alumno. Para conseguir eso el usuario no puede usar el cursor conseguido anteriormente, puesto que no tiene ningún filtro asociado. El API habría sido más robusto si el STAX permitiera asociar filtros a `XMLStreamReader` ya creados, pero lamentablemente el API STAX solo permite crearlos desde nuevos cursores con filtro. Esta decisión obliga a que el usuario se acuerde de pedir a la librería de firmas un nuevo cursor y olvidarse de utilizar el antiguo. Las líneas siguientes muestran este comportamiento:

```
StaxValidateContext stx =
```

```
        StaxValidateContext.createEnvelopedValidator(
         checkSignature(in),reader);
        reader=im.createFilteredReader(reader,
          stx.getStreamReader());
```

Después de estas líneas. el usuario puede utilizar el cursor con el filtro para analizar el fichero (como lo haría normalmente si no existiera el verificador de firmas).

Una vez analizado el fichero y cuando el cursor llegue al final del documento, el usuario querrá saber si la firma ha sido correcta, para eso usará las siguientes instrucciones:

```
XMLSignatureFactory fac=XMLSignatureFactory.getInstance("Stax");
stx.setSignatureNumber(0);
XMLSignature sig=fac.unmarshalXMLSignature(stx);
if (!sig.validate(stx)) {
  //Firma invalida.
}
```

La primera línea le pide a la factoría de firmas JSR105 un proveedor que entienda STAX.

La siguiente línea le dice a la variable `stx` de tipo `StaxValidateContext` que la firma a verificar es la primera (un documento podría contener más de una firma).

La siguiente instrucción indica que se obtenga la firma requerida. En este momento se valida la firma (reacuérdese que `stx` se creó con la clave que íbamos a utilizar para verificar).

A continuación se presenta el código completo de un programa que analiza un documento firmado de tipo Alumno. La única diferencia entre este código y el que se presentó para leer el documento Alumno, son las líneas en que se obtiene el filtro, y llas últimas líneas en que se verifica la firma.

```
package verificador;

import java.io.FileInputStream;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;


public class VerificadorStax {
   final static String ALUMNO_NS="http://fi.upm.es/alumnos/1/0";
   final static String LIBRO_NS="http://fi.upm.es/libros/1/0";
   static Key key=…;
   public static void main(String[] args) throws Exception {
```

```
        XMLInputFactory im=XMLInputFactory.newInstance();
        im.setProperty("javax.xml.stream.supportDTD", new
Boolean(false));
        XMLStreamReader reader=im.createXMLStreamReader(new
FileInputStream("alumnoEnveloped.xml"));
        StaxValidateContext stx =
       StaxValidateContext.createEnvolopedValidator(key,reader);

           reader=im.createFilteredReader(reader,
stx.getStreamReader());
        String nombre=null;
        String apellidos=null;
        do {
           if (reader.getEventType()==XMLStreamReader.START_ELEMENT) {
                 if (!ALUMNO_NS.equals(reader.getNamespaceURI()))
                             continue;
                 if ("Nombre".equals(reader.getLocalName())) {
                             reader.next();
                             nombre=reader.getText();
                             continue;
                 }
                 if ("Apellidos".equals(reader.getLocalName())) {
                             reader.next();
                             apellidos=reader.getText();
                             continue;
                 }
           }
           reader.next();
      } while ((reader.getEventType())!=XMLStreamReader.END_DOCUMENT);
      System.out.println("Nombre: "+nombre+" Apellidos:"+apellidos);
      XMLSignatureFactory fac=XMLSignatureFactory.getInstance("Stax");
      stx.setSignatureNumber(0);
      XMLSignature sig=fac.unmarshalXMLSignature(stx);
      if (!sig.validate(stx)) {
            //Firma invalida.
            System.out.println("Firma Invalida");
      }
  }
}
```

5. Implementación de un verificador Stream.

Después de ver como un cliente usaría la librería de firmas para verificar una firma, vamos a ver en este capítulo su implementación en Java y una explicación de una ejecución.

La primera clase de la librería con la que interactúa el usuario es la clase `StaxValidatingContext`. Su código simplificado es el siguiente:

```
public class StaxValidateContext implements XMLValidateContext {
	XMLStreamReader reader;
	int signatureNumber=0;
	private StaxSignatureVerifcator sig;
	Key key;
	public static StaxValidateContext createEnvolopedValidator(Key
key, XMLStreamReader reader) {
		return new StaxValidateContext(key,reader);
	}
	public void setSignatureNumber(int number) {
		signatureNumber=number;
	}

	protected StaxValidateContext(Key key,XMLStreamReader reader) {
		this.key=key;
		this.reader=reader;
	}

	public StreamFilter getStreamReader() {
		sig = new StaxSignatureVerificator();
		return sig;
	}

	protected XMLSignature getSignature() {
		return sig.signatures.get(signatureNumber);
	}

}
```

Este objeto mantiene la clave que se va usar para verificar y determina qué firma se devolverá cuando la factoría de firmas pida una. También es responsable de crear el filtro STAX que será el responsable de dirigir la verificación.

A continuación se muestra el código de la clase `StaxSignatureVerificator`:

```
public class StaxSignatureVerificator implements StreamFilter{
	List<XMLSignatureWorker> signatures=new
ArrayList<XMLSignatureWorker>();
	List<StaxWorker> filters=new ArrayList<StaxWorker>();
```

```
	List<Integer> filterStart=new ArrayList<Integer>();
	List<StaxWatcher> watchers=new ArrayList<StaxWatcher>();
	int level=0;
	public StaxSignatureVerificator() {
		watchers.add(new SignatureWatcher());
	}
	public void addSignature(XMLSignatureWorker s) {
		signatures.add(s);

	}
	public void insertWatch(IdWatcher watcher) {
		watchers.add(watcher);

	}
	public boolean accept(XMLStreamReader cur) {
		…
		//This code has be shown before
		…
	}
}
```

`StaxSignatureVerificator` es la clase principal del diseño, mantiene a los observadores y a los trabajadores y les notifica de los eventos importantes. Nota: el código del método `accept` definido en esta clase ya se ha mostrado cuando explicamos el diseño de las clases.

Es importante recalcar las variables miembros de clase:

- `signatures`: Una lista donde se guardan las firmas descubiertas hasta ahora en el documento.
- `watchers`: Una lista de observadores a los que hay notificar de cada inicio de elemento.
- `filters`: Una lista de trabajadores a notificar de todos los eventos.
- `filtersStart`: Una lista ordenada igual que la anterior donde se anota en qué nivel del árbol se registraron los trabajadores para eliminarlos cuando se cierre ese nivel.

En la creación de la clase, la única lista que tiene un elemento es la de `watchers` que se rellena por el constructor con un objeto del tipo `SignatureWatcher`.

Como se ve en el código mostrado a continuación, el método `watch` cuando sea notificado de un elemento con nombre `Signature` y que pertenezca al *namespace* de

Digital Signature, creará un objeto de tipo `XMLSignatureWorker` y lo añadirá a lista de firmas que mantiene el filtro. Este objeto le será devuelto al filtro, que anotará en qué nivel del árbol está, y le notificará de todos los eventos hasta que salga de ese nivel.

```
class SignatureWatcher implements StaxWatcher {
      public StaxWorker watch(XMLStreamReader reader,
StaxSignatureVerificator sig) {
            String name=reader.getLocalName();
            String uri=reader.getNamespaceURI();
            if (name.equals("Signature") &&
                        uri.equals(Constants.DS_URI)) {
                  XMLSignatureWorker s=new XMLSignatureWorker();
                  sig.addSignature(s);
                  return s;
            }

            return null;
      }
}
```

Con el fichero del ejemplo, anterior este observador será notificado de la apertura del elemento `RootObject`, y acto seguido de la del elemento `Signature`. En ese momento desencadenará la creación del trabajador asociado a esa firma. A este observador se le seguirá informando de todos los elementos, aun que en nuestro ejemplo no volverá a encontrar ninguna otra firma.

A continuación se incluye el código del objeto trabajador `XMLSignatureWorker`, que como se ve es responsable de obtener el valor del `SignatureValue` y de crear, cuando se encuentre con la apertura del elemento `SignedInfo`, un `SignedInfoWorker`.

```
public class XMLSignatureWorker implements StaxWorker,XMLSignature {

 SignedInfoWorker si;
 private boolean readSignatureValue=false;
 private byte[] signatureValue;
 public StaxWorker read(XMLStreamReader reader) {
    switch (reader.getEventType()) {
      case XMLStreamReader.START_ELEMENT:
        if (Constants.DS_URI.equals(reader.getNamespaceURI())) {
            String name=reader.getLocalName();
            if (name.equals("SignedInfo") ) {
                  si=new SignedInfoWorker();
                  return si;
```

```
            }
            if (name.equals("SignatureValue")) {
                  readSignatureValue=true;
            }
        }
        break;
      case XMLStreamReader.END_ELEMENT:
        if (Constants.DS_URI.equals(reader.getNamespaceURI())) {
           if (reader.getLocalName().equals("SignatureValue")) {
              readSignatureValue=false;
            }
        }
        break;
      case XMLStreamReader.CHARACTERS:
        if (readSignatureValue) {
         try {
           signatureValue=Base64.decode(reader.getText());
         } catch (Base64DecodingException e) {
            e.printStackTrace();
         }
        }
        break;
    }
    return null;
 }
 public StaxWatcher remove() {
   return null;
 }
 public boolean validate(XMLValidateContext validateContext)
    throws XMLSignatureException {
    StaxValidateContext ctx=(StaxValidateContext) validateContext;
    try {
       for (Reference ref: si.references) {
         if (!ref.validate(ctx))
            return false;
        }
        SignatureAlgorithm sa=new SignatureAlgorithm(
                  si.signatureMethod);
        sa.initVerify(ctx.key);
        sa.update(si.bos.toByteArray());
        return sa.verify(signatureValue);
    } catch (org.apache.xml.security.signature.XMLSignatureException
e) {
            e.printStackTrace();
      }
      return false;
 }
 public SignedInfo getSignedInfo() {
      return si;
 }
}
```

Otro método a destacar es el de `validate`, al que llama el usuario para comprobar la validez de la firma. Para hacer eso valida primero todas la referencias

contenidas en el `SignedInfo`, devolviendo `false` si alguna es inválida. Si todas son correctas, crea un verificador que inicializa con la clave que contiene el `StaxValidateContext`, obtiene los *bytes* normalizados de la `SignedInfo` y actualiza el verificador. A continuación obtiene el número resumen de la firma y devuelve el resultado de la confirmación al usuario.

En nuestro ejemplo el `SignatureWorker` será notificado primero del `SignedInfo` y creará un `SignedInfoWorker` que lo entregará al filtro. Luego seguirá recibiendo eventos pertenecientes al `SignedInfo` hasta que este se cierre y llegue la apertura del `SignatureValue`. Una posible mejora al diseño consistiría en poder insertar trabajadores exclusivos, es decir que un trabajador le pudiera indicar al filtro que el trabajador recién creado es responsable de todos los eventos que estaban dirigidos a él (y que estos deben serle pasados exclusivamente al nuevo trabajador) hasta que el nuevo trabajador haya finalizado.

```
class SignedInfoWorker implements
 StaxWorker, SignedInfo, DigestResultListener {
  ByteArrayOutputStream bos=new ByteArrayOutputStream();
      boolean initial=true;
      C14nWorker c14n=new C14nWorker(this,bos);
      List<ReferenceWorker> references=new
ArrayList<ReferenceWorker>();
      String signatureMethod;
      String c14nMethod;
      public StaxWorker read(XMLStreamReader reader) {
       if (reader.getEventType()==XMLStreamReader.START_ELEMENT &&
         Constants.DS_URI.equals(reader.getNamespaceURI())) {
            String name=reader.getLocalName();
            if (name.equals("Reference") ) {
                  ReferenceWorker r=new ReferenceWorker();
                  references.add(r);
                  return r;
            }
            if (name.equals("SignatureMethod")) {
              signatureMethod=reader.getAttributeValue(null,
           "Algorithm");
            }
            if (name.equals("CanonicalizationMethod")) {
                c14nMethod=reader.getAttributeValue(null,"Algorithm");
            }
      }
      if (initial) {
            initial=false;
            return c14n;
      }
```

```
        return null;
    }

    public StaxWatcher remove() {
        return null;
    }

    public List getReferences() {
        return references;
    }

    public void setResult(byte[] result) {

    }
}
```

El `SignedInfoWorker` analiza toda la información contenida en el elemento `SignedInfo` y generará un trabajador por cada referencia. Una peculiaridad del `SignedInfo` es que tiene que ser normalizado al mismo tiempo que se lee, por eso devuelve un trabajador normalizador nada mas empezar (controlado por la variable booleana `initial`). Sin embargo el método de normalizado no es conocido hasta que ha pasado ya una serie de elementos. La solución a este problema se podría implementar de dos maneras:

1. Implementar una normalización desconocida que simplemente guarde los eventos recibidos en una pila interna. Cuando se conozca la definitiva, simplemente se le pasarían a esta todos los eventos almacenados para ponerla al día. Esta opción fue la elegida en la implementación de la librería de SAX.

2. Tener dos trabajadores normalizando uno de forma inclusiva y otro exclusiva. De esta manera cuando venga el dato de la canonicalización bastará con elegir la correcta. Se puede pensar que este método es más costoso pero la longitud del elemento `SignedInfo` no es muy grande.

Actualmente la implementación de STAX solo maneja normalizaciones exclusivas, con lo que este problema no existe. Pero cuando se expanda a manejar todo el estándar, lo más probable es que se elija la segunda opción (tener dos trabajadores), por ser más fácil de implementar. En este caso vemos que es deseable todos los trabajadores reciban los eventos.

El `ReferenceWorker` es el último de los trabajadores que se encargan de interpretar el elemento `Signature`. Su principal tarea es obtener la información necesaria para preparar la normalización de la referencia, y obtener el valor calculado por el firmante. Como se puede ver en el código incluido a continuación, esta implementación no trata ninguna transformación, y supone que se va normalizar de forma exclusiva. Esta es una simplificación que se tendrá que eliminar en versiones futuras. Pero muchas de las transformaciones representadas en el estándar no se podrán implementar como el XPath por que no hay implementación XPath para STAX. Otras, como la `enveloped-signature`, no tiene sentido intentar tratarlas a este nivel, porque si realmente fuera necesaria, esto se sabría demasiado tarde (ya que lo referenciado ha aparecido antes que la referencia).

```
class ReferenceWorker
  implements StaxWorker, Reference, DigestResultListener {
      boolean readDigestValue=false;
      String uri;
      String c14nType;
      String digestMethod;
      byte[] digestValue;
      byte[] calculateDigestValue;
      boolean correct=false;
      DigesterOutputStream os;
      public StaxWorker read(XMLStreamReader reader) {
        switch (reader.getEventType()) {
        case XMLStreamReader.START_ELEMENT:
            if(Constants.DS_URI.equals(reader.getNamespaceURI())) {
              String name=reader.getLocalName();
              if (name.equals("Reference") ) {
                  uri=reader.getAttributeValue(null,"URI");
              }
              if (name.equals("DigestMethod")) {
               digestMethod=reader.getAttributeValue(null,"Algorithm");
               try {
                  MessageDigest ms = MessageDigest.getInstance(
                        JCEMapper.translateURItoJCEID(digestMethod));
                  os=new DigesterOutputStream(ms);
                } catch (NoSuchAlgorithmException e) {
                  //TODO: Better error handling.
                  e.printStackTrace();
                }
              }
              if (name.equals("DigestValue")) {
                  readDigestValue=true;
              }
            }
            break;
        case XMLStreamReader.END_ELEMENT:
            if (Constants.DS_URI.equals(reader.getNamespaceURI())) {
```

```
              if (reader.getLocalName().equals("DigestValue")) {
                  readDigestValue=false;
              }
            }
            break;
        case XMLStreamReader.CHARACTERS:
            if (readDigestValue) {
                  try {
                        digestValue=Base64.decode(reader.getText());
                  } catch (Base64DecodingException e) {
                        e.printStackTrace();
                  }
              }
              break;
            }
            return null;
      }
      public StaxWatcher remove() {
            return new IdWatcher(uri.substring(1),this,os);
      }
      public void setResult(byte[] result) {
            calculateDigestValue=os.getDigestValue();
            correct=Arrays.equals(calculateDigestValue, digestValue);
      }
      public byte[] getDigestValue() {
            return digestValue;
      }
      public byte[] getCalculatedDigestValue() {
            return calculateDigestValue;
      }
      public boolean validate(XMLValidateContext validateContext)
throws XMLSignatureException {
            return correct;
      }
      public String getURI() {
            return uri;
      }
}
```

Cuando este trabajador se retira porque se ha cerrado el elemento `Reference`, expande un nuevo observador `IdWatcher` encargado de encontrar la referencia a firmar.

Cuando el normalizador de lo referenciado notifique al `ReferenceWorker`, este comprobará que lo normalizado una vez pasado por la función *hash* sea idéntico a lo calculado por el firmante. Si esto es igual, se dará la referencia por correcta, si no se marcará como incorrecta.

El siguiente observador será notificado de todas las aperturas de elementos y buscará uno que tenga en el atributo Id, el valor buscado. Cuando lo encuentre creará un `C14nWorker` para normalizarlo.

```
class IdWatcher implements StaxWatcher {
 String uri;
 DigestResultListener re;
 OutputStream os;
 public IdWatcher(String uri, DigestResultListener reader,
   OutputStream os) {
      this.uri=uri;
      this.re=reader;
      this.os=os;
 }
 public StaxWorker watch(XMLStreamReader reader,
      StaxSignatureVerificator sig) {
      if (uri.equals(reader.getAttributeValue(null, "Id"))) {
            return new C14nWorker(re,os);
      }
      return null;
  }
}
```

El siguiente trabajador simplemente va pasando los eventos a un normalizador (que no tiene estructura de trabajador). Cuando el trabajador se despacha (porque termina el elemento en el que estaba trabajando), el trabajador notifica al elemento interesado en su trabajo que ha terminado (en nuestro caso el `ReferenceWorker` y el `SignedInfoWorker`).

```
public class C14nWorker implements StaxWorker {
      DigestResultListener re;
      C14n c14n;
      public C14nWorker(DigestResultListener re,OutputStream os) {
            c14n=new C14n(new
com.r_bg.stax.c14n.AttributeHandleExclusive(),os);
            this.re=re;
      }

      public StaxWorker read(XMLStreamReader reader) {
            c14n.accept(reader);
            return null;
      }

      public StaxWatcher remove() {
            re.setResult(null);
            return null;
      }
}
```

De esta manera la firma es analizada y sus referencias comprobadas de una pasada y sin intervención de usuario.

6. Resultados

Para comprobar experimentalmente la validez de los diseños, vamos a probar las dos librerías contra una serie de documentos XML firmados con la librería de DOM de distintos tamaños, incluyendo la firma:

- 1 KB El documento mínimo significativo que se puede firmar (el elemento `Signature` con una clave publica ocupa unos 800bytes). La longitud típica de un mensaje SAML (firmado) es de 1.6 KB. Esta prueba intenta ser representativa de protocolos XML/HTTP ligeros (sin mucho intercambio de información por petición).
- 42 KB es el tamaño medio de un documento simple firmado. El uso típico de la librería de firmas para un protocolo XML/HTTP pesado y pequeños documentos XML puros.
- 1 MB Documentos XML pequeños de substitución de formatos específicos tipo WORD, AUTOCAD, etc. Con binarios codificados dentro del XML (gráficos de *bitmaps*, etc.). Por ejemplo este TFC ocupa un poco más de 1 MB
- 34 MB Documentos XML medianos.
- 100 MB El tamaño de documento más grande que puede manejar la librería de firmas en un ordenador normal con 1GB de RAM sin hacer que la paginación sea insoportable.

Como se ha comentado se ha usado la librería de firmas DOM para generar los documentos de distintos tamaños. Estos documentos son una firma de tipo envolvente, con canonicalización exclusiva tanto para la transformación de la referencia, como para el elemento `SignedInfo`. El algoritmo de resumen es `sha1` y el de firma `rsa1-sha1`. El contenido a firmar es una repetición de la línea:

```
<Prueba><SubPrueba>Prueba de firmas
gordas<SubPrueba><SubSubPrueba></SubSubPrueba>Otro textillo por
aqui</SubPrueba></SubPrueba>
</Prueba>
```

Hasta conseguir la longitud de fichero requerida.

El ejemplo de 1 KB es el siguiente (sombreados los aspectos más significativos)

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<ds:SignedInfo>
<ds:CanonicalizationMethod
Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"></ds:CanonicalizationMethod>
<ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-
sha1"></ds:SignatureMethod>
<ds:Reference URI="#1">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"></ds:Transform>
</ds:Transforms>
<ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></ds:DigestMe
thod>
<ds:DigestValue>xFnPTUHYHptGbXy4Rd/lTzzo8XA=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
QzkKez
KBdj+Nsag+gwevSql8B1eCWecMZH+309KRNqX7MoUFar56fv7a7d4zYCZbU+BslV
nSSFgs
YrbzV1FK3RuelfqyWnede0/JdaXDSvWaEXO3kaedfz/1M/vyRvH4Rq7Yl9SM+zwp
R0VplnZTG+nN
+uqZktSbH0ZS6io5hiE=
</ds:SignatureValue>
<ds:KeyInfo>
<ds:KeyValue>
<ds:RSAKeyValue>
<ds:Modulus>
j8uRbDOpjjuRWw7zVzUXNTmXHXlDy5y7lN8xMVUfHC4v6Agaw+l8AhQV3pH39PU8
RMYV4za9xxPj
PDaNtNBYBu5pdJuAGXYy0xGEdJZhhkuFSukVgnQLW7AYT7ewBzORcOwA7eEn8GN6
Hly35lKcPuGA
xto6Xr7DTEocwjYCYCE=
</ds:Modulus>
<ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
</ds:KeyValue>
</ds:KeyInfo>
<ds:Object Id="1"><Prueba><SubPrueba>Prueba de firmas
gordas<SubPrueba><SubSubPrueba></SubSubPrueba>Otro textillo por
aqui</SubPrueba></SubPrueba>
</Prueba>
</ds:Object>
</ds:Signature>
```

Todas las demás firmas son iguales con la salvedad de las veces que se repite el elemento `Prueba`.

Los documentos se verifican dos veces por cada librería de firmas (DOM o STAX), la primera vez solo se mide el tiempo de ejecución. Y la segunda vez se corre en una máquina virtual java instrumentada para medir el consumo de memoria. En algunas pruebas se vuelve a ejecutar una tercera vez para medir el tiempo gastado en cada método.

Todas las pruebas se realizan bajo la versión 1.5.0_09 de la JVM con las opciones por defecto (excepto que se especifique alguna otra opción dentro de la descripción de la prueba) dentro del entorno de desarrollo *Netbeans* 5.5 de *Sun*. La ejecución instrumentada se realizó dentro del mismo entorno de desarrollo pero con el *plugin* de *Profiler*, que ejecuta la misma máquina virtual java pero con parámetros de *profiling* que modifican ligeramente la ejecución del programa (veremos un ejemplo concreto más tarde).

Los gráficos aquí mostrados son capturas de las pantallas del *Netbeans* presentando los datos de memoria o el tiempo de ejecución.

Antes de pasar a presentar los datos y su análisis, unas consideraciones sobre la memoria y la JVM.

Java es un lenguaje definido con *garbage collector* [16] (recogida de basura) por lo que el programador no tiene que preocuparse de liberar la memoria reservada (reservada con la instrucción `new`), sino que el lenguaje es el encargado de reclamar la memoria reservada que nadie referencia. El lenguaje no especifica qué método se usado para detectar esa situación y cuándo lo hace sino que se deja en manos de la implementación de la máquina virtual.

El primer mecanismo implementado y el más utilizado hasta ahora se llama *mark-and-sweep* y es implementado como un *thread* paralelo a la ejecución del programa que lo detiene en momentos de escasez de memoria o periódicamente. Este *thread* recorre todos los objetos accesibles desde las pilas de programa (una por cada *thread*) y las variables globales del mismo así como todos los objetos accesibles desde estos. Esto se

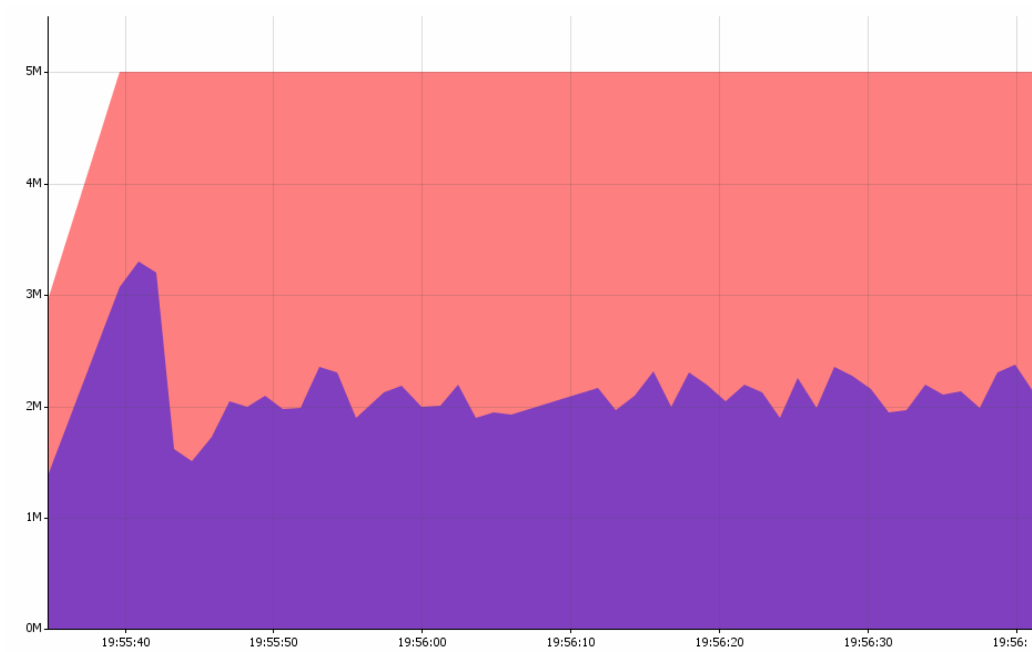
repite hasta que tiene marcados todos los objetos accesibles por el programa. En ese momento procede a copiar los objetos accesibles a una nueva zona y borra la zona de reserva antigua (liberando de esta manera la memoria no utilizada). Seguidamente continúa con la ejecución de la aplicación.

Si se había ejecutado la fase de *garbage collector* por escasez de memoria y después de la compactación/reclamación no hay suficiente memoria libre, se pide un nuevo bloque de memoria al sistema operativo (aumentando en ese momento la memoria residente del proceso). Si se ha sobrepasado un límite de memoria predeterminado se aborta el programa con una excepción de falta de memoria (el límite por defecto en la versión 1.5.0_09 es de 64 MB, este límite se aumenta pasándole el parámetro –Xmx<Tamaño>m en la línea de comandos de invocación de la máquina virtual).

En los gráficos de memoria que vamos a ver a continuación se muestran estos dos conceptos: el de memoria reservada (residente) como sombreado en naranja y el de memoria actualmente ocupada en azul.

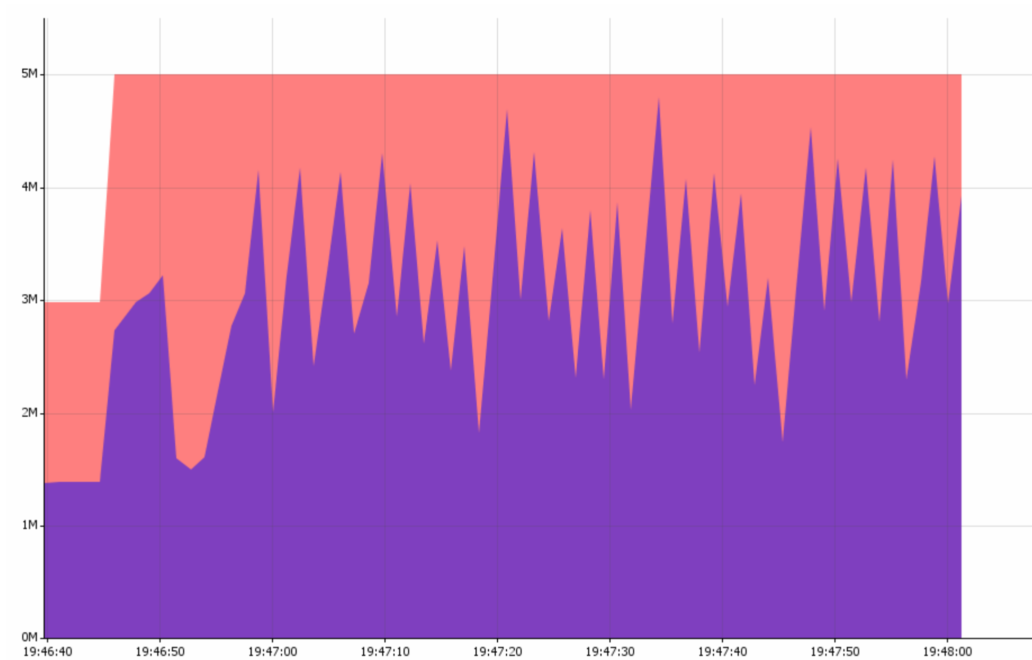
6.1. Verificación de 10.000 Firmas de 1322 bytes

STAX



44.315.738.452ns (44s)

DOM



60.066.677.748ns (60s)

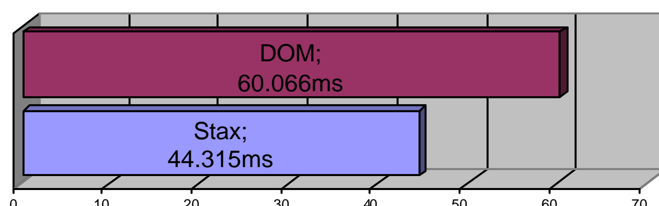
En este ejemplo se decidió verificar 10.000 veces la misma firma para obtener suficientes muestras de consumo de memoria. Con menos firmas los datos son menos significativos.

El comportamiento de la librería STAX para la verificación de 10.000 firmas de tamaño 1 KB, es bastante homogéneo. Empieza con un pico de 3 MB que es lo que consume la librería de firmas DOM en inicializarse. La librería de firmas STAX usa algunos servicios de la librería DOM (como la transformación de nombres de algoritmos de firma en notación w3c a clases java) y eso se lee de un fichero XML por procedimientos DOM.

Una vez gastados esos 3 MB y recogida la basura de la memoria no necesitada, el consumo de memoria se estabiliza en 2 MB con pequeños picos de menos de 200 KB (que es lo que debe de ser el consumo efectivo de verificación de la firma, el resto debe de ser *overhead* necesario de miembros de clase y código, aunque no se puede determinar de manera exacta el consumo efectivo).

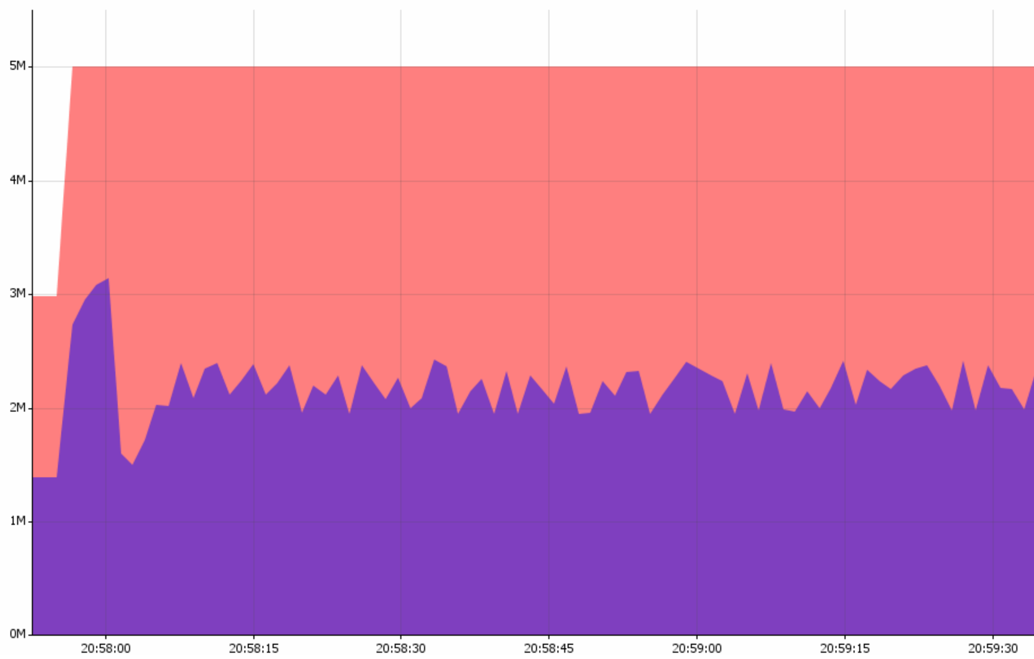
El patrón de memoria de la librería DOM es más caótico. Igual que pasaba con la librería de STAX después de inicializarse, baja su consumo a 1,5 MB pero inmediatamente sube a 4 MB para fluctuar entre 2 MB y 4 MB, con picos de más de 2 MB que debe de ser el consumo efectivo de un documento, aunque tampoco es demostrable.

Respecto al tiempo de ejecución, la librería STAX es un 25% más rápida que la librería DOM. La mejora parece escasa teniendo en cuenta la diferencia de rendimiento entre la lectura pura DOM/STAX pero la librería de firmas DOM está muy optimizada, mientras que la de STAX se ha diseñado para ser conceptualmente simple y no tiene todas las optimizaciones (como veremos más adelante) de la de DOM.



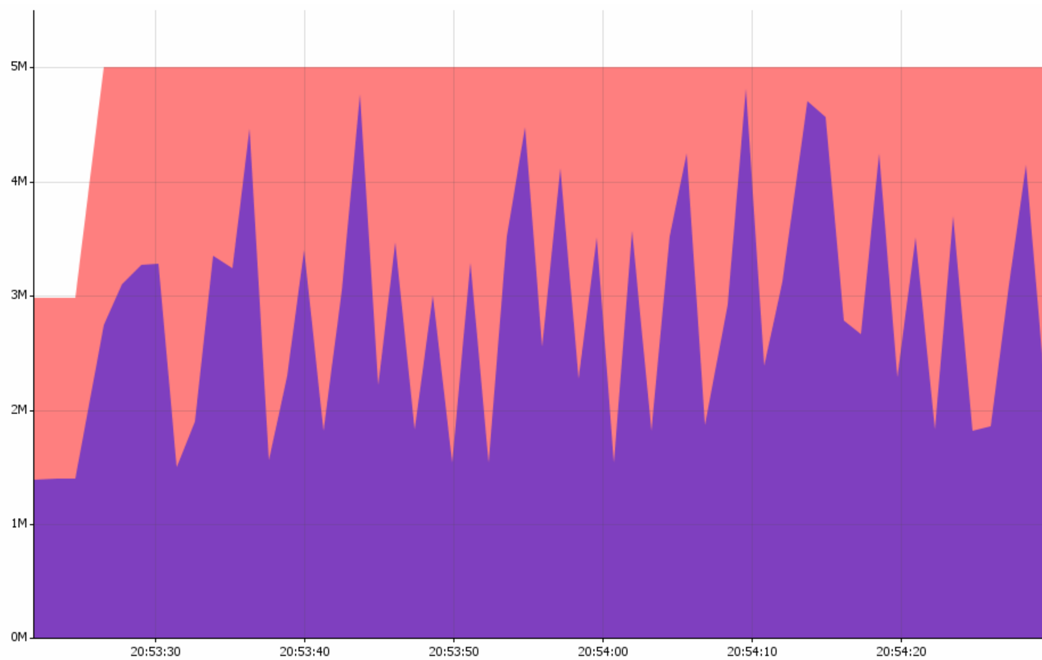
6.2. Verificación de 1.000 Firmas de 42 KB

STAX



56.253.108.426 ns

DOM

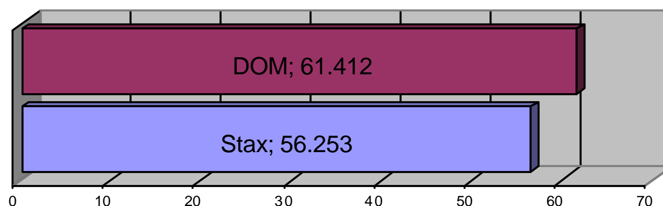


61.412.510.250 ns

El comportamiento de la librería de STAX en firmas de 42 KB es similar al que tenía con firmas de 1 KB, el sistema inicializa la librería gastando 3 MB, se estabiliza en 2 MB con picos de 200 KB igual que antes. La única diferencia significativa es que el número de picos ha aumentado. Esto puede ser debido a que también ha aumentado el tiempo de ejecución y con él el número de veces que entra el *garbage collector*. Aunque también puede ser debido a que al empezar una nueva firma y al ser esta más grande, la presión sobre la memoria sea mayor y se activa el *garbage collector*. En contra de esta última explicación es que el volumen de memoria residente sin gastar es todavía muy alto en los puntos de bajada (que es donde detectamos que entra el *garbage collector).*

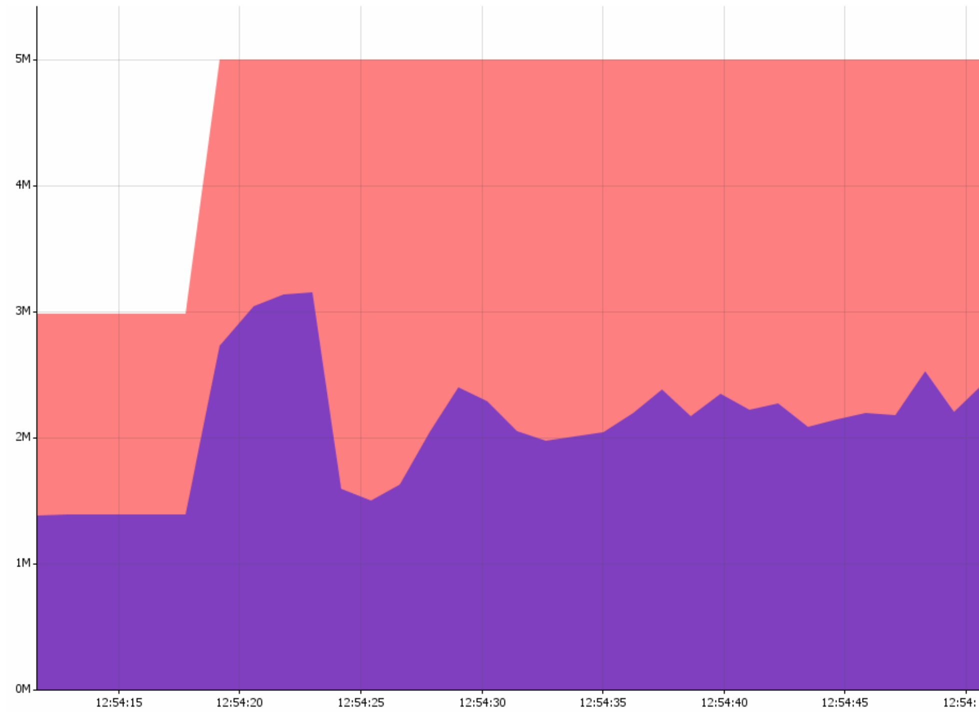
El comportamiento de la librería DOM es también bastante similar al observado con las firmas de 1 KB, incluso parece que hay menos presión sobre la memoria al haber más picos de 3 MB de los que había en el caso anterior. Una explicación posible es que al haber menos firmas y estas verificarse en un tiempo superior a un ciclo de recolección de basura, cuando entra periódicamente el *garbage collector* se encuentra menos memoria que recoger.

Respecto al rendimiento, parece que la diferencia entre las dos implementaciones se va acortando a medida que aumenta el tamaño del documento.



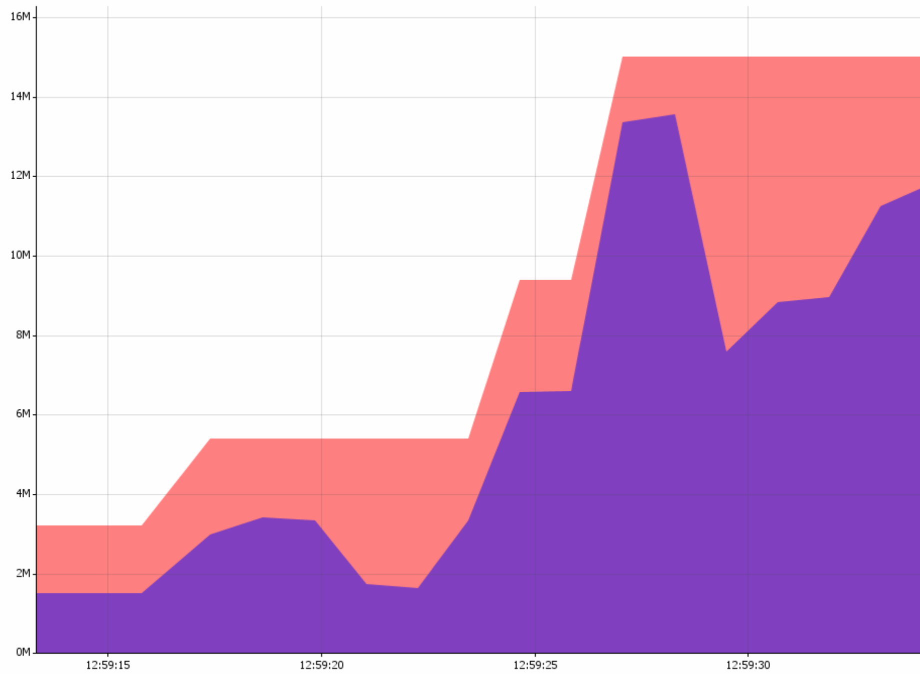
6.3. Verificación de 10 Firmas de 1 MB

STAX



13.177.925.407 ns

DOM

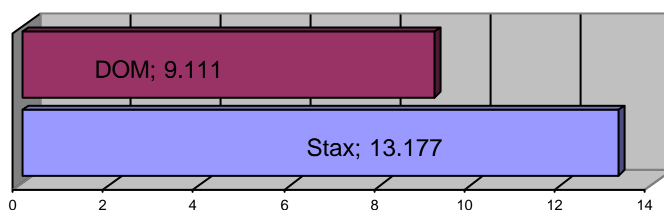


9.111.582.618 ns

El gráfico de consumo de memoria de STAX es idéntico a los anteriores, con menos dientes de sierra porque el programa solo se estuvo ejecutando durante 13 segundos y por tanto el *garbage collector* no pudo entrar tantas veces como en las ejecuciones anteriores.

Esta misma característica puede explicar el gráfico tan extraño que sale con la librería DOM, aunque se puede ver que es el mismo patrón de comportamiento que luego se repite muchas más veces en ejecuciones más largas.

Lo que merece mayor motivo de análisis es que por primera vez la librería DOM es más rápida que la de STAX (un 30%)



Para analizar mejor esto hemos ejecutado otra vez el mismo caso, esta vez diciéndole a la máquina java de *profiling* que estamos interesados en el tiempo de ejecución.

Después de ejecutar el programa, el *plugin* de *profiling* interpreta los resultados y saca un gráfico en árbol mostrando el coste de ejecución relativo al total de cada método invocado (véase página siguiente). En él se puede observar que el 32% (22%+10%) del tiempo de ejecución se lo lleva, por distintos caminos, el método estándar del objeto `String getBytes()`.

Este método se usa para obtener la representación UTF8 de una cadena Unicode de java (que esta codificada en una variante de UTF16). Esta conversión ya se detectó como costosa en la librería DOM y fue cambiada a varios métodos de conversión optimizados, más rápidos que los incluidos en la librería estándar. Incluso la librería DOM mantiene una cache de nombres de elementos para no tener que hacer la conversión más de una vez. Todas estas mejoras están en la versión DOM pero no en la STAX y puede mejorar la balanza del lado DOM.

Call Tree - Method	Time [%]	Time	Invocations
All threads		262280 ms (100%)	1
main		262280 ms (100%)	1
profileingstax.Main.**main** (String[])		262280 ms (100%)	1
profileingstax.Main.**checkSignatureStax** (javax.xml.stream.XMLInputFactory, java.io.InputStream,		259894 ms (99,1%)	10
com.ctc.wstx.stax.FilteredStreamReader.**next** ()		256089 ms (97,6%)	848690
com.r_bg.stax.StaxSignatureVerificator.**accept** (javax.xml.stream.XMLStreamReader)		227630 ms (86,8%)	848690
com.r_bg.stax.C14nWorker.**read** (javax.xml.stream.XMLStreamReader)		165684 ms (63,2%)	848390
com.r_bg.stax.c14n.C14n.**accept** (javax.xml.stream.XMLStreamReader)		164144 ms (62,6%)	848390
com.r_bg.stax.c14n.C14n.**obtainName** (javax.xml.namespace.QName, java.io.C		61929 ms (23,6%)	616980
java.lang.String.**getBytes** ()		57739 ms (22%)	617160
org.apache.xml.security.utils.DigesterOutputStream.**write** (byte[])		3023 ms (1,2%)	616840
Self time		1168 ms (0,4%)	616980
java.io.OutputStream.**write** (byte[])		0.800 ms (0%)	320
java.lang.ClassLoader.**checkPackageAccess** (Class, java.security.Protectic		0.146 ms (0%)	1
java.io.ByteArrayOutputStream.**write** (int)		0.048 ms (0%)	160
org.apache.xml.security.utils.DigesterOutputStream.**write** (int)		0.004 ms (0%)	20
com.r_bg.stax.c14n.AttributeHandleExclusive.**handleAttributes** (javax.xml.stre		37369 ms (14,2%)	308490
java.lang.String.**getBytes** ()		26284 ms (10%)	231410
com.r_bg.stax.c14n.StaxC14nHelper.**push** ()		20084 ms (7,7%)	308490
org.apache.xml.security.utils.DigesterOutputStream.**write** (int)		6602 ms (2,5%)	925230
Self time		3796 ms (1,4%)	848390
org.apache.xml.security.utils.DigesterOutputStream.**write** (byte[])		3069 ms (1,2%)	539710
com.ctc.wstx.stax.FilteredStreamReader.**getName** ()		1638 ms (0,6%)	616980
com.ctc.wstx.stax.FilteredStreamReader.**getText** ()		1521 ms (0,6%)	231410
com.ctc.wstx.stax.FilteredStreamReader.**getEventType** ()		1291 ms (0,5%)	848390
com.r_bg.stax.c14n.StaxC14nHelper.**pop** ()		569 ms (0,2%)	308490
java.io.OutputStream.**write** (byte[])		0.491 ms (0%)	190
java.io.ByteArrayOutputStream.**write** (int)		0.129 ms (0%)	240
Self time		1541 ms (0,6%)	848390
Self time		11917 ms (4,5%)	848690
java.util.AbstractList$Itr.**next** ()		9655 ms (3,7%)	2314540
java.util.ArrayList.**<init>** ()		9141 ms (3,5%)	1697420
com.r_bg.stax.XMLSignatureWorker.**read** (javax.xml.stream.XMLStreamReader)		6905 ms (2,6%)	848680
java.util.ArrayList.**addAll** (java.util.Collection)		6117 ms (2,3%)	1697420
java.util.AbstractList.**iterator** ()		4873 ms (1,9%)	1157270
java.util.AbstractList$Itr.**hasNext** ()		3847 ms (1,5%)	3471810
com.r_bg.stax.SignatureWatcher.**watch** (javax.xml.stream.XMLStreamReader, com.r_bg.		3200 ms (1,2%)	308560
java.util.ArrayList.**contains** (Object)		2120 ms (0,8%)	308570
com.r_bg.stax.IdWatcher.**watch** (javax.xml.stream.XMLStreamReader, com.r_bg.stax.St		1614 ms (0,6%)	308470
com.ctc.wstx.stax.FilteredStreamReader.**getEventType** ()		1308 ms (0,5%)	848690

Otra mejora que incorpora la librería DOM pero no la STAX es el control de *buffers* para firma y resumen. La librería STAX escribe directamente sobre el verificador/digester mientras que la librería DOM guarda esto en un buffer que solo vacía cuando esté lleno. Esto mejora el rendimiento porque el verificador/digester usa funciones optimizadas para varios datos.

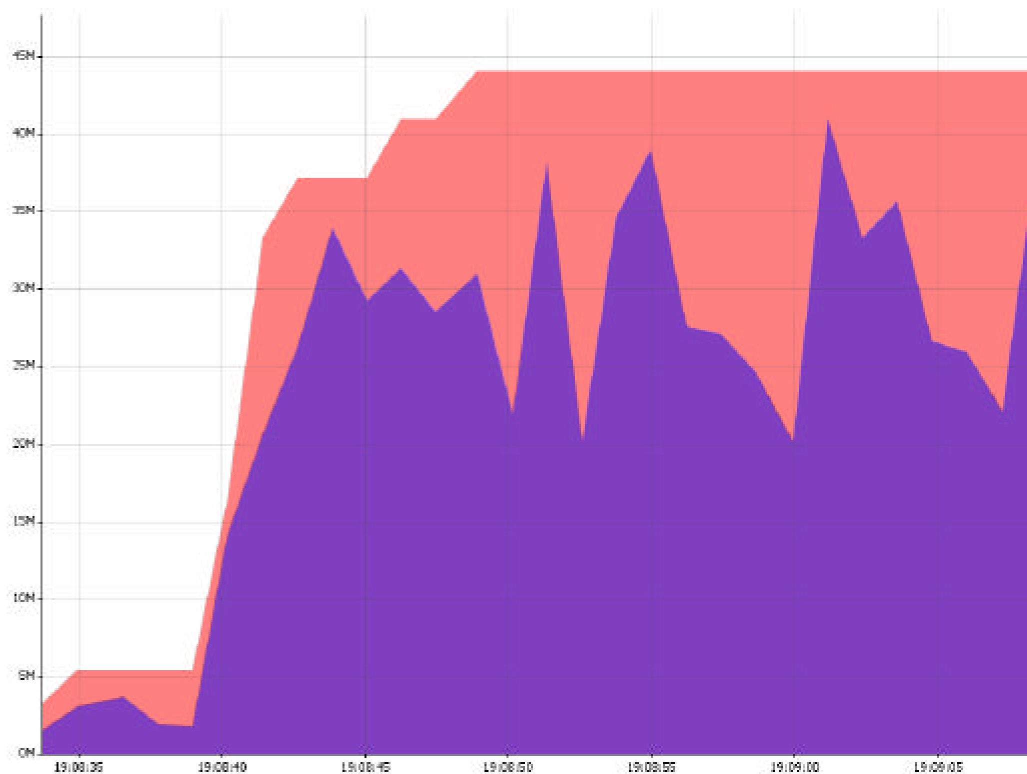
Estas optimizaciones no son exclusivas de la librería DOM y se podrían portar también a la rama de STAX.

Por último vamos a usar este caso para ver la diferencia de la arquitectura *pipeline* con la arquitectura visitador. Para ello vamos a volver a ejecutar el caso DOM en una versión antigua (versión 1.1) de la librería de firmas que tiene implementada dicha arquitectura.

Tristemente las diferencias de versiones (todos los casos hasta ahora han sido ejecutados con la versión 1.4), no se refieren solo el cambio de arquitectura sino que contienen otros muchas mejoras de rendimiento y velocidad. Esto hay que tenerlo en cuenta a la hora de sacar conclusiones de los datos.

Como se puede ver en la figura siguiente los picos de memoria son mucho más altos en la versión 1.1 que en la versión 1.4. Mientras que en la 1.4 los picos eran de 14 MB, en la 1.1 son de 35 MB e incluso de 40 MB. Esto puede ser un dato para confirmar la hipótesis de que el patrón *pipeline* consume el doble (recordemos que solo hay una transformación) de memoria que el patrón visitador.

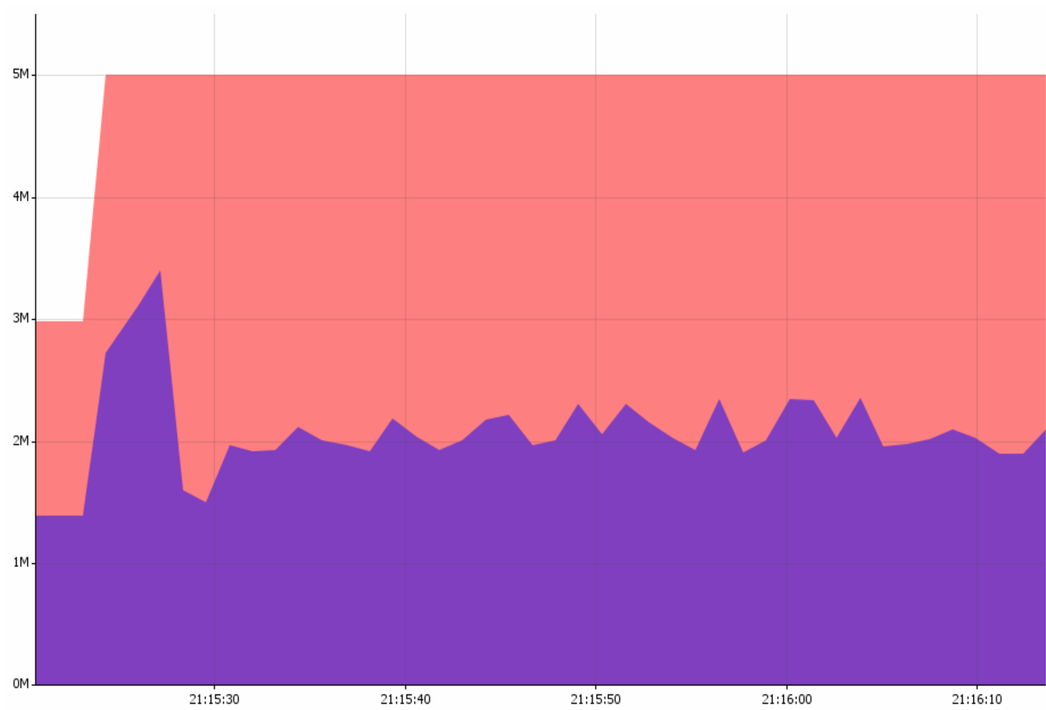
DOM 1.1



De todas maneras el autor no se atreve a extrapolar más datos por las múltiples diferencias que hay entre las versiones de la librería.

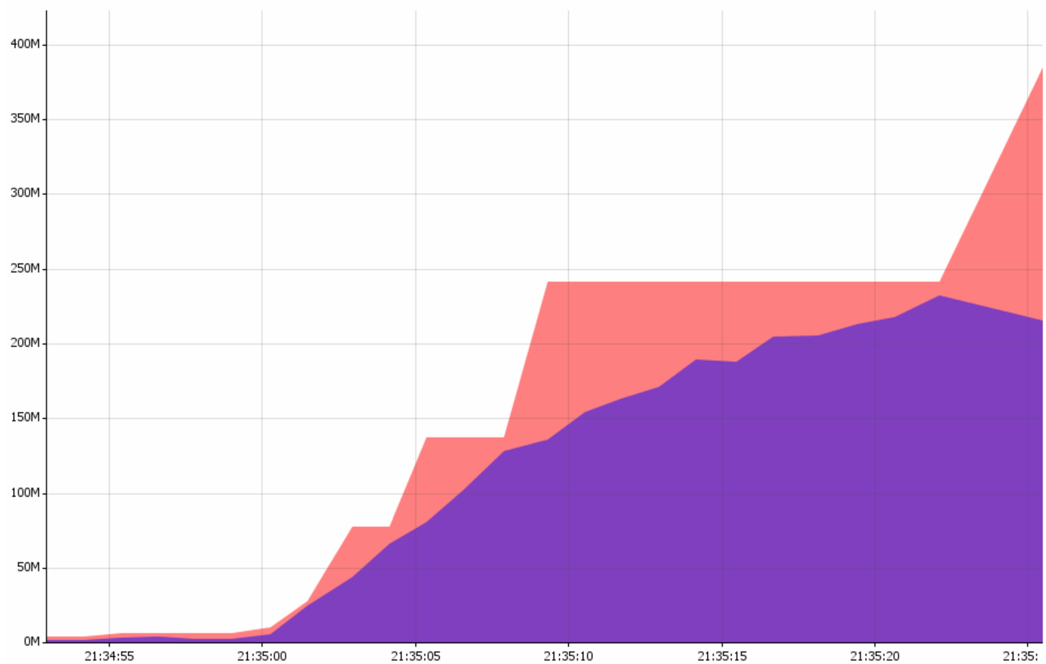
6.4. Verificación de una Firma de 34 MB

STAX



43.673.506.828ns

DOM*



31.355.095.690ns

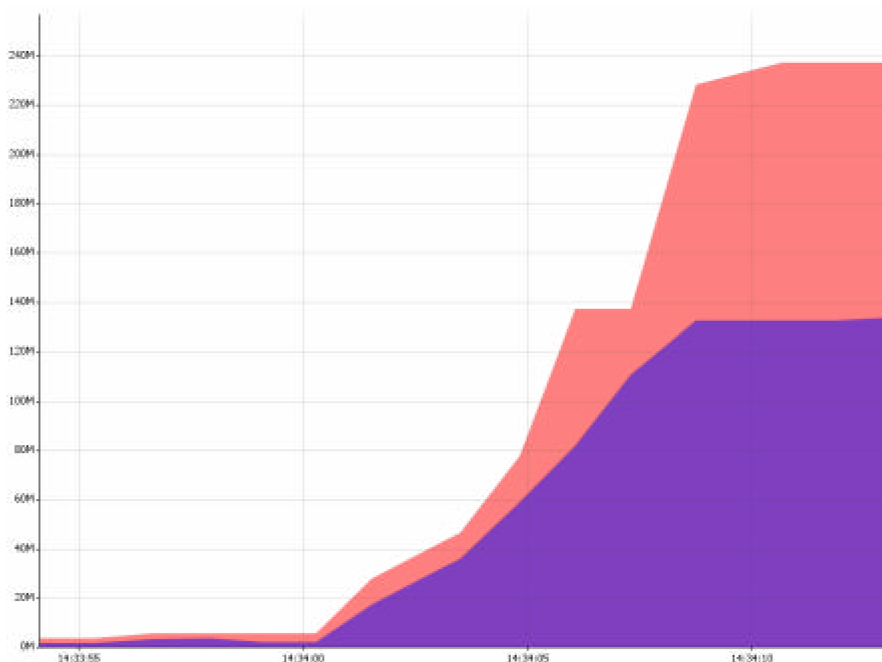
El gráfico de memoria de la librería STAX es un calco de los anteriores.

En cambio la librería DOM ha necesitado que le aumentáramos el tamaño máximo de memoria a 512 MB (java -Xmx512m), para poder terminar la verificación sin lanzar una excepción de falta de memoria. También el gráfico es bastante diferente, esto es debido a dos factores: solo estamos verificando una firma, el *garbage collector* entra en momentos en que toda (o la mayor parte) de la memoria está siendo utilizada.

De todas maneras el consumo de memoria parece desproporcionado: para verificar un fichero de 34 MB requiere una memoria residente de 250 MB (la última reserva de 100 MB de memoria residente fue un fallo del predictor de necesidades de memoria de la JVM).

Para investigar qué parte del consumo de memoria se debe a librería de firmas y cual a la librería de análisis DOM, se midió el consumo del siguiente programa, que simplemente carga el documento `Enveloped34M.xml` en memoria (que es el mismo documento que queremos para verificar).:

```
DocumentBuilderFactory fact=DocumentBuilderFactory.newInstance();
fact.setNamespaceAware(true);
Document doc=fact.newDocumentBuilder().parse("Enveloped34M.xml");
```



Como se ve, la lectura del fichero consume unos 120 MB de memoria residente con lo que la librería debe consumir los otros 130 MB para verificar la firma.

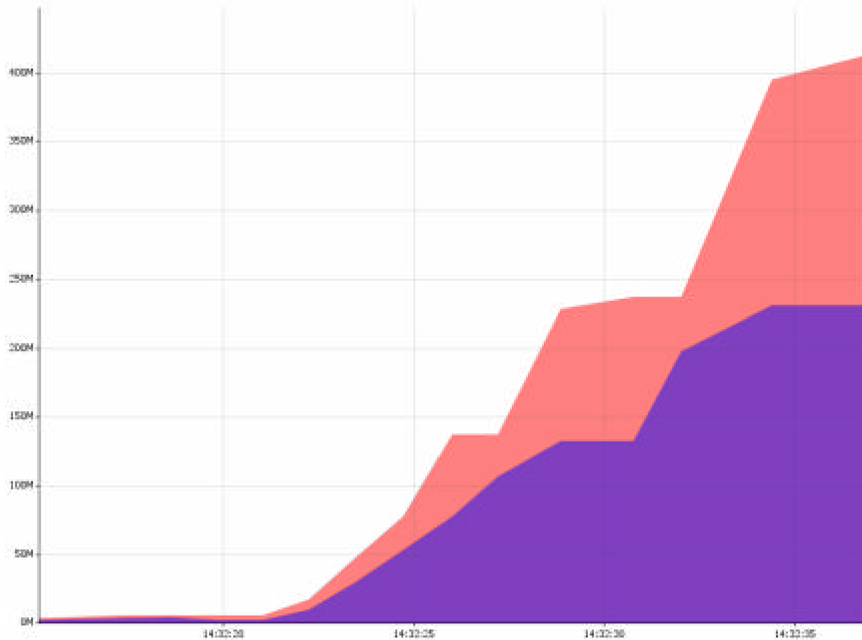
Pero estos datos entran en contradicción con lo expuesto en el capítulo de mejoras de reducción de memoria. Ahí se argumentó que el coste de verificación/firma de un documento es constante, y no depende de la longitud del mismo. Ignorando claro está las caches y tablas auxiliares que la librería utiliza para agilizar el proceso. Por muy grandes que fueran esas tablas no podrían ocupar 100 MB.

La explicación a esto sin introducir la duda sobre lo expuesto en este trabajo, es pensar que la librería de análisis DOM no ha creado todas las estructuras necesarias para el procesamiento del documento cuando ha devuelto el control al programa de usuario (es decir la librería es *lazy* en vez de *greedy*). Para comprobar esta hipótesis se midió el consumo de memoria del siguiente programa, que lo único que hace es recorrer los nodos del árbol DOM sin hacer nada.

```
public static void visitDomStructure(Node n) {
        Node child=n.getFirstChild();
        while (child!=null) {
                visitDomStructure(child);
                child=child.getNextSibling();
                }
}
…
DocumentBuilderFactory fact=DocumentBuilderFactory.newInstance();
fact.setNamespaceAware(true);
Document doc=fact.newDocumentBuilder().parse("Enveloped34M.xml");
System.gc();
Thread.sleep(1000);
System.gc();
visitDomStructure(doc);
System.gc();
Thread.sleep(1000);
System.gc();
```

Se podría pensar que al ser el visitador una función recursiva el consumo de memoria puede ser alto debido solamente a esa recursividad. Pero si recordamos el documento que estamos analizando (que es una copia expandida del presentado arriba) el nivel de anidamiento es pequeño (8 elementos a la sumo) con lo que el consumo por la pila es despreciable. Además el consumo debido a pila no se presenta en los gráficos, que solo muestran el consumo de Heap.

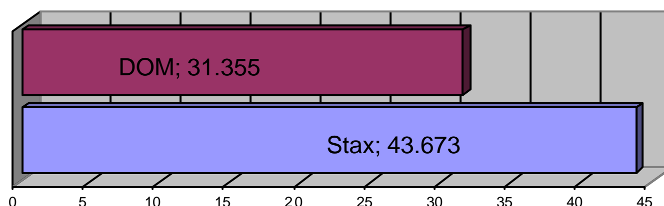
El gráfico del consumo del programa es:



Con lo que la hipótesis es correcta y la implementación de la librería de análisis DOM es *lazy* (cosa que también podíamos haber comprobado mirando la documentación si hubiéramos sabido que la implementación DOM que trae por defecto la máquina virtual java en sus versiones 1.5 es la librería Xerces de Apache).

Otra cosa que se puede deducir viendo el gráfico, es que para la lectura DOM de un documento de 34 MB se necesita 240 MB de memoria residente. Si hemos determinado con el gráfico anterior que la verificación de un documento de 34 MB es de 250 MB podemos determinar que el consumo por parte del verificado es de unos 10 MB (lo más probable es que sea bastante menor, pero con los datos actuales es una buena aproximación).

Respecto a la velocidad de ejecución podemos observar que el DOM sigue siendo un 30% más rápido que el STAX



Para comprobar la hipótesis de que esto es debido a la conversión UTF16 a UTF8 presentada arriba, observemos el árbol de costes de ejecución:

Call Tree - Method	Time [%]	Time	Invocations
All threads		450408 ms (100%)	1
main		450408 ms (100%)	1
profileingstax.Main.**main** (String[])		450408 ms (100%)	1
profileingstax.Main.**checkSignatureStax** (javax.xml.stream.XMLInputFactory, java.io.InputStream,		447546 ms (99.4%)	1
com.ctc.wstx.stax.FilteredStreamReader.**next** ()		442364 ms (98.2%)	2883643
com.r_bg.stax.StaxSignatureVerificator.**accept** (javax.xml.stream.XMLStreamReader)		394793 ms (87.7%)	2883643
com.r_bg.stax.C14nWorker.**read** (javax.xml.stream.XMLStreamReader)		295875 ms (65.7%)	2883613
com.r_bg.stax.c14n.C14n.**accept** (javax.xml.stream.XMLStreamReader)		292496 ms (64.9%)	2883613
com.r_bg.stax.c14n.C14n.**obtainName** (javax.xml.namespace.QName, java.io.C		111742 ms (24.8%)	2097170
java.lang.String.**getBytes** ()		101180 ms (22.5%)	2097188
org.apache.xml.security.utils.DigesterOutputStream.**write** (byte[])		8304 ms (1.8%)	2097156
Self time		2269 ms (0.5%)	2097170
java.lang.ClassLoader.**checkPackageAccess** (Class, java.security.Protectic		0.129 ms (0%)	1
java.io.OutputStream.**write** (byte[])		0.089 ms (0%)	32
java.io.ByteArrayOutputStream.**write** (int)		0.008 ms (0%)	16
org.apache.xml.security.utils.DigesterOutputStream.**write** (int)		0.003 ms (0%)	2
com.r_bg.stax.c14n.AttributeHandleExclusive.**handleAttributes** (javax.xml.stre		58884 ms (13.1%)	1048585
java.lang.String.**getBytes** ()		44668 ms (9.9%)	786443
com.r_bg.stax.c14n.StaxC14nHelper.**push** ()		34906 ms (7.7%)	1048585
org.apache.xml.security.utils.DigesterOutputStream.**write** (int)		12010 ms (2.7%)	3145731
Self time		8409 ms (1.9%)	2883613
org.apache.xml.security.utils.DigesterOutputStream.**write** (byte[])		7527 ms (1.7%)	1835009
com.ctc.wstx.stax.FilteredStreamReader.**getName** ()		4855 ms (1.1%)	2097170
com.ctc.wstx.stax.FilteredStreamReader.**getText** ()		4183 ms (0.9%)	786443
com.ctc.wstx.stax.FilteredStreamReader.**getEventType** ()		3093 ms (0.7%)	2883613
com.r_bg.stax.c14n.StaxC14nHelper.**pop** ()		2259 ms (0.5%)	1048585
java.io.OutputStream.**write** (byte[])		0.093 ms (0%)	19
java.io.ByteArrayOutputStream.**write** (int)		0.070 ms (0%)	24
Self time		3386 ms (0.8%)	2883613
Self time		20373 ms (4.5%)	2883643
java.util.AbstractList$Itr.**next** ()		18065 ms (4%)	7864474
java.util.ArrayList.**<init>** ()		12800 ms (2.8%)	5767290
java.util.ArrayList.**addAll** (java.util.Collection)		10012 ms (2.2%)	5767290
com.r_bg.stax.XMLSignatureWorker.**read** (javax.xml.stream.XMLStreamReader)		9950 ms (2.2%)	2883642
java.util.AbstractList.**iterator** ()		6772 ms (1.5%)	3932237
com.r_bg.stax.SignatureWatcher.**watch** (javax.xml.stream.XMLStreamReader, com.r_bg.		5525 ms (1.2%)	1048592
java.util.AbstractList$Itr.**hasNext** ()		4047 ms (0.9%)	11796711
com.r_bg.stax.IdWatcher.**watch** (javax.xml.stream.XMLStreamReader, com.r_bg.stax.St		3426 ms (0.8%)	1048583
com.ctc.wstx.stax.FilteredStreamReader.**getEventType** ()		[illegible]	[illegible]

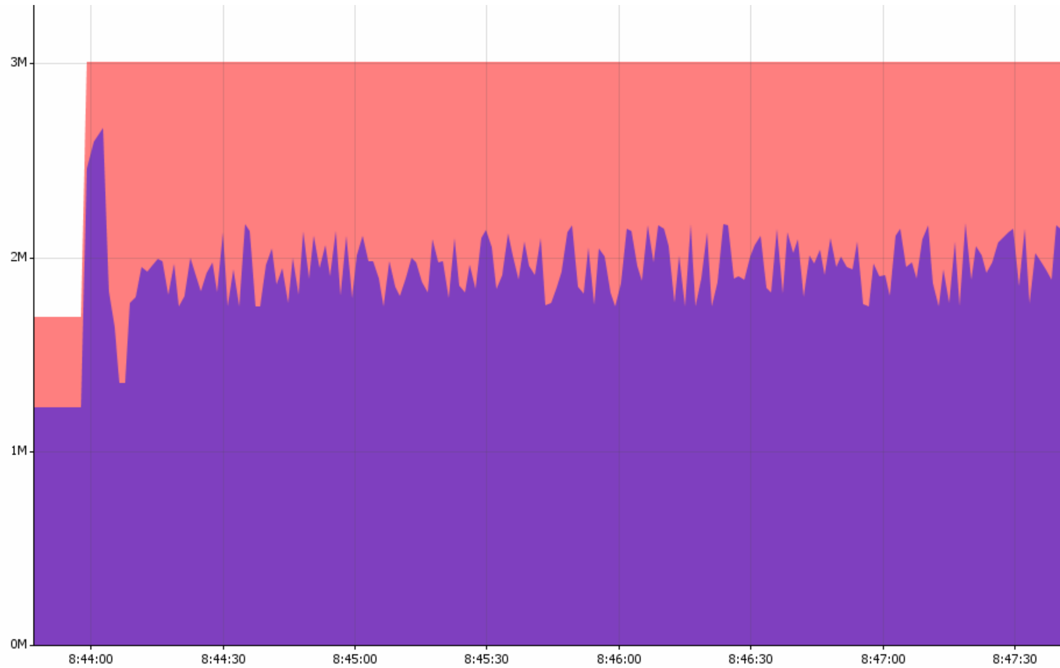
El coste de el método `String.getBytes()` sigue manteniéndose incluso ha aumentado un poco (32,4%).

Para corroborar totalmente la hipótesis, se debería modificar el código de la librería de STAX con las mejoras. Pero este caso no presenta ninguna evidencia empírica en contra de ella.

Como con la firma anterior de 1 MB, también se ha intentado usar esta firma de 34 MB para comprobar la diferencia entre el patrón *pipeline* y el visitador. Tristemente la firma no pudo ser verificada con los 512 MB dados, ni siquiera con 640 MB.

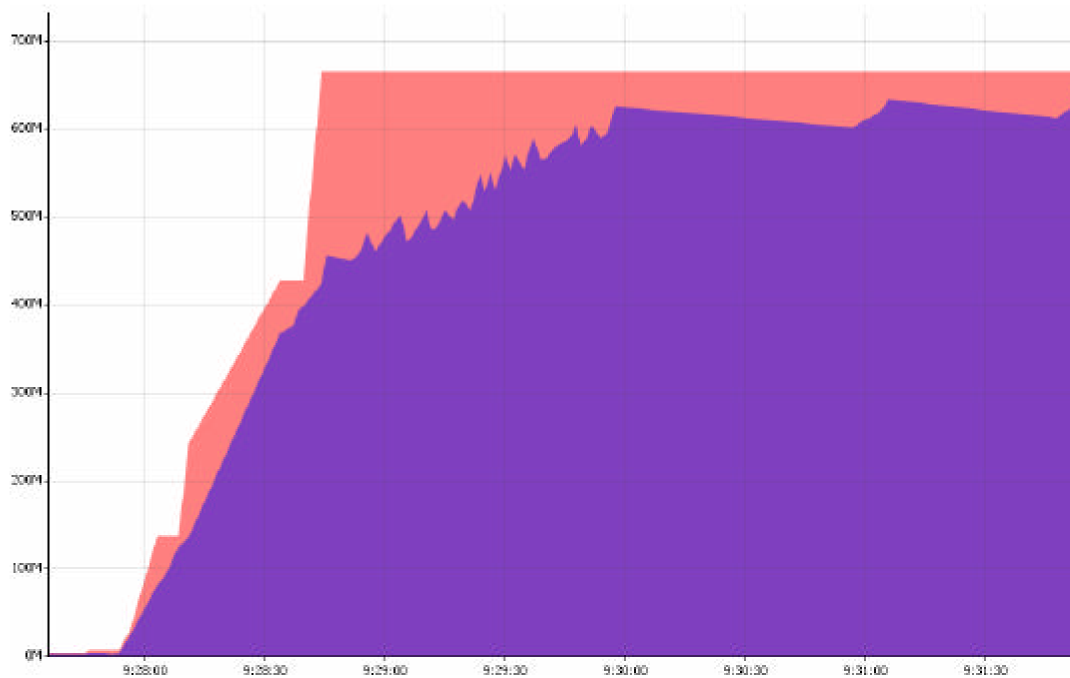
6.5. Verificación de una firma de 103 MB

Stax



128.955.176.985 ns

DOM*

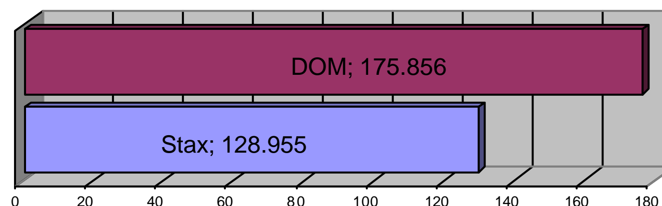


175.856.378.674ns

El gráfico de memoria de la librería STAX es idéntico a los anteriores, y no ha aumentado a medida que aumentaba el tamaño del fichero a verificar. No hay motivos para pensar que el consumo aumentaría si le pidiéramos verificar un fichero de 10GB.

Tristemente no sucede lo mismo con la librería DOM. Aunque la ejecución no instrumentada logró verificar la firma asignándole solo 512 MB de memoria máxima, no pasó lo mismo cuando se ejecutó en la máquina virtual instrumentada. En este caso no terminó, se quedó bloqueada y tuvo que ser finalizado por el usuario con herramientas del sistema operativo. Se intentó ejecutar de nuevo, ampliándole la memoria máxima a 640 MB pero sin éxito. Parece que la máquina virtual instrumentada solo funciona bien con programas que reserven menos de 512 MB.

Respecto a la velocidad en esta prueba la librería, STAX fue la más rápida, pero esto pudo ser debido a las necesidades de memoria de la librería DOM, que obligó al sistema operativo a paginar otros programas para poder cumplirlas, penalizándole en su rendimiento.



7. Conclusiones

Con el código y los datos aquí presentados se puede concluir que es posible realizar una implementación del estándar de firmas XML que sea rápida y a la vez permita la firma de documentos arbitrariamente grandes con un consumo de memoria constante y extremadamente reducido (2-3 MB con código incluido).

Al tener unas necesidades de consumo frugales, este código se podría usar en dominios donde la memoria es escasa, como entornos móviles o empotrados, donde hasta ahora no se consideraban las firmas digitales debido a su coste.

Tristemente, para hacer uso de esta librería, los clientes tendrán que portar el código de sus aplicaciones del API de lectura DOM al API de lectura STAX. Este cambio, aunque costoso, no es tan traumático como si tuvieran que migrar a SAX.

En algunos casos, el impacto en la arquitectura software de producto que impone la librería STAX será tan grande que no merecerá la pena el esfuerzo de intentarlo. Para ese tipo de usuarios el cambio de arquitectura interna de la librería DOM de *pipeline* a visitador les ha permitido incrementar el tamaño máximo de las firmas que pueden manipular.

Aunque el código desarrollado es útil actualmente, para ampliar su uso habría que completarlo resolviendo algunos aspectos. Lo primero que se debería hacer es implementar también la normalización (*canonicalización*) inclusiva (en el código de Apache está ya implementada al 90%, aunque le faltan algunos pequeños detalles para darla por finalizada). Habría que expandir el API para que permitiera la verificación de otros tipos de firmas en los que lo referenciado aparece antes que la firma. La infraestructura para permitirlo está ya desarrollada y solo falta trabajar en el API de usuario para dotarlo de una ergonomía de uso razonable.

Aunque el código implementado solo verifica firmas, de nada sirve poder verificar firmas de tamaño arbitrariamente grande si no existe el mecanismo para crearlas. Por lo tanto, debería implementarse también la parte de creación de firmas. Por suerte, el proceso de firma y verificación es simétrico y solo varía en que en la operación de firma

no existe el elemento Signature, sino que hay que crearlo y rellenarlo con los datos calculados, mientras que en el proceso de verificación se trata de comprobar que los datos calculados coinciden con los existentes. Igual que antes, la mayor complicación vendrá dada por el diseño del API y no por la complicación técnica de la tarea.

Por último, se debe afrontar la optimización de la librería, ya que los resultados muestran que es competitiva para mensajes pequeños, pero es más lenta que la librería de DOM a tamaños medios. Su eficacia con los mensajes de tamaño reducido la hacen principalmente atractiva para el manejo de protocolos XML como SAML, Liberty o WS-Security. La inclusión de las optimizaciones ya comentadas podría convertirla en la librería de firmas más rápida de las existentes (la librería de DOM se considera actualmente la librería de firmas más rápida).

8. Bibliografía

1. XML-Signature Syntax and Processing, W3C Recommendation 12 February 2002 (http://www.w3.org/TR/xmldsig-core/)
2. Canonical XML Version 1.0, W3C Recommendation 15 March 2001 (http://www.w3.org/TR/xml-c14n)
3. Exclusive XML Canonicalization Version 1.0, W3C Recommendation 18 July 2002 (http://www.w3.org/TR/xml-exc-c14n/)
4. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000 (http://www.w3.org/TR/2000/REC-xml-20001006)
5. XML Schema Part 1: Structures, W3C Recommendation 2 May 2001 (http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/)
6. Security Assertion Marckup Language, OASIS (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security)
7. Liberty Identity Federation Framework (ID-FF), Liberty Alliance, http://www.projectliberty.org/resource_center/specifications/liberty_alliance_id_ff_1_2_specifications
8. XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999 (http://www.w3.org/TR/1999/REC-xpath-19991116)
9. XML-Signature XPath Filter 2.0, W3C Recommendation 08 November 2002 (http://www.w3.org/TR/xmldsig-filter2/)
10. Java XML Digital Signature API Specification (JSR 105), JSR, http://download.java.net/jdk7/docs/technotes/guides/security/xmldsig/overview.html
11. Document Object Model (DOM) Level 1 Specification Version 1.0, W3C Recommendation 1 October, 1998 (http://www.w3.org/TR/REC-DOM-Level-1/)

12. Java 1.6 security features
(http://java.sun.com/javase/6/docs/technotes/guides/security/index.html)

13. Apache XML Security project (http://santuario.apache.org o
http://xml.apache.org/security/)

14. Apache Software Foundation (http://www.apache.org/)

15. Experimental single pass SAX XML signature verification
(http://issues.apache.org/bugzilla/show_bug.cgi?id=32657)

16. Tuning Java Garbage Collector
(http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html)

17. RFC 3174 - US Secure Hash Algorithm 1 (SHA1), IETF,
(http://www.faqs.org/rfcs/rfc3174.html)

18. PKCS #1: RSA Cryptography Standard, RSA Security
(http://www.rsasecurity.com/rsalabs/node.asp?id=2125)

9. Apéndice: Mejoras de rendimiento en la librería de firmas DOM.

En este trabajo se ha comparado el rendimiento de la versión 1.4 de la librería de firmas DOM con la librería experimental de STAX. Para tener una visión más completa del escenario en que se ha desarrollado el trabajo se presentan aquí algunas mejoras que el autor ha incorporado a las distintas versiones de la librería DOM. Estas modificaciones han mejorado el rendimiento de la librería y este epígrafe tiene el propósito de resumir el trabajo realizado y su impacto en el rendimiento.

Primero veamos las diferencias de alto nivel de las versiones de la librería DOM:

- **1.0.5D2** Versión base.
- **1.1**
  - Pequeños cambios de una línea, sobre todo para no volver a calcular resultados o datos ya obtenidos en la configuración.
- **1.2.1**
  - Creación de un camino rápido para firmas simples (solo con transformaciones de normalización (c14n) y opcionalmente de extracto de firma (`enveloped-transform`)) y un camino lento para las demás.
  - Implementación del patrón visitador para el camino rápido.
- **1.3**
  - Optimizaciones en estructuras de datos.
  - Manejo de *Buffers* para mandar en bloque datos al resumen y al verificador.
  - Optimización de las transformaciones UTF16 a UTF8.
- **1.4**

- Implementación del patrón visitador para el camino lento.
- Reducción de la creación de objetos auxiliares, pasando de uno por firma a uno solo por *thread*.

Para medir el impacto de estos cambios, vamos a probar la velocidad de firma de los distintos tipos de documentos y tipos de firmas que se han seleccionado para medir la diferencia entre el camino rápido y el lento.

Hemos elegido tres tipos de documentos que vamos a firmar con dos tipos de firmas:

Documentos:

1. TINY Un documento con un simple elemento y sin texto.
2. SAML Una petición de SAML 1.1 de longitud 1,6 KB.
3. 34 KB una petición pseudos SAML ampliada a 34 KB.

Firmas (todas son de tipo `enveloped`, con método de resumen: *sha1* y algoritmo de firmado *rsa-sha1*):

1. No Xpath: Una firma con transformación `enveloped` y normalización exclusiva. La URI de la referencia contiene el identificador del elemento a firmar.
2. Xpath: Una firma con transformación `enveloped`, transformación `XPath` que selecciona el elemento a firmar, y normalización exclusiva.

Para comprobar la diferencia de rendimiento se escribió un programa que firma primero un bloque de 1000 firmas idénticas de tipo No XPath, luego otro bloque de 1000 de tipo Xpath y repite este proceso múltiples veces.

Esta ejecución en bucle doble se hace para determinar el impacto del JIT (Just In Time Compiler) de Java en la ejecución.

El JIT va optimizando los métodos a medida que detecta que son "puntos calientes" de ejecución (*hotspots*), compilándolos a código máquina, haciendo métodos *inline*, etc. Esto repercute en que al principio, la ejecución es más lenta (la compilación

y supervisión es un trabajo extra) pero una vez optimizados los puntos calientes, la ejecución debe ser más rápida. El grado de optimización va cambiando también con el número de ejecuciones: primero se eligen optimizaciones pequeñas y rápidas,para ir aplicando optimizaciones más agresivas a medida que aumenta el número de ejecuciones.

Empíricamente, el número en que se estabilizan las mejoras del orden de 10.000 firmas. Finalmente se eligió el número de 1000 firmas para poder ejecutar las pruebas en las versiones antiguas de la librería en un tiempo razonable (con 1000 firmas la versión 1.0.5D2 tarda unos 13 minutos en ejecutar 4 iteraciones, con 10.000 tardaría unas 2 horas).

En los gráficos que se incluyen a continuación la primera ejecución de tipo `No XPath`, se presenta con la leyenda `No XPath1`, la segunda con `No XPath2`,.de manera análoga se representan las ejecuciones de tipo `XPath`.

En algunos gráficos aparece también una línea horizontal etiquetada como `Plain`. Esta línea representa cuánto tardan las primitivas Java en ejecutar 1000 veces lo siguiente, sin tener ningún *overhead* de normalización (c14n) o procesamiento XML, sino, simplemente ejecutando esas primitivas criptográficas sobre texto sencillo:

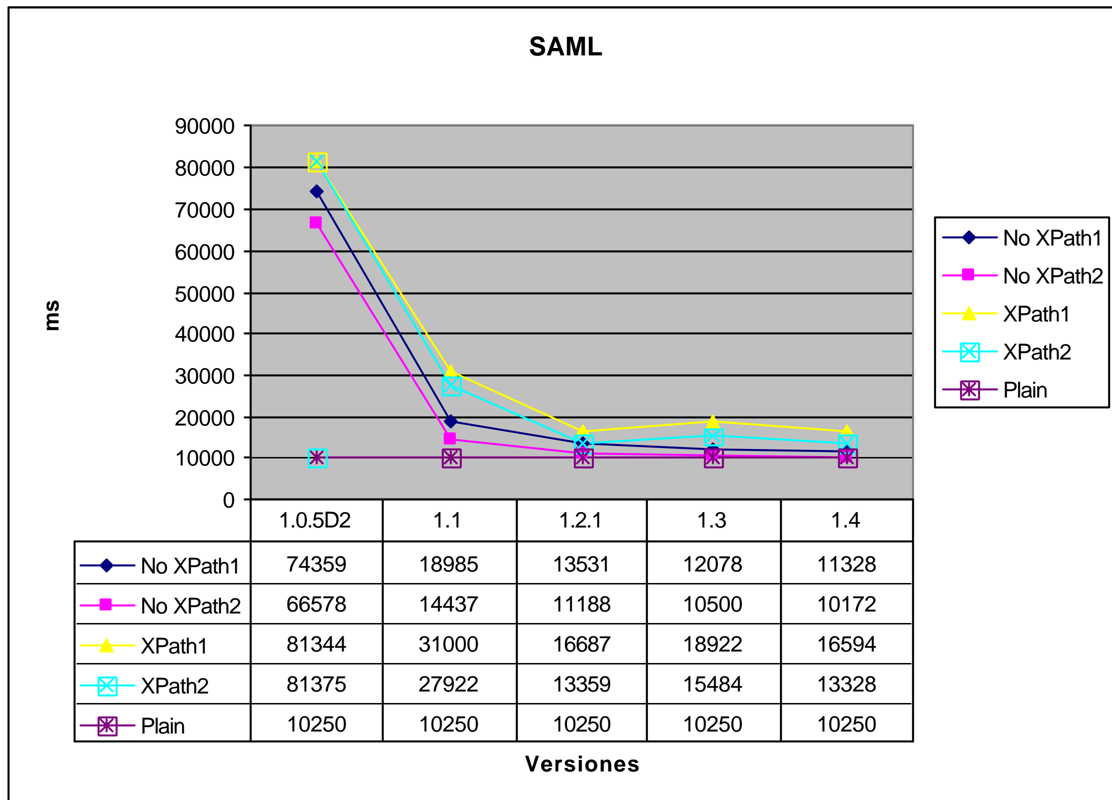
1. firmar los bytes de la firma,
2. hacer un resumen de los bytes de la referencia.

Como la librería DOM usa estas primitivas, se puede pensar que representa el límite inferior de tiempo de ejecución.

SAML es un protocolo de seguridad muy usado y extendido por otros estándares como Liberty o WS-Security. Utiliza como base el estándar de firmas y es un buen ejemplo real en el que probar las optimizaciones (además de ser los mensajes que el producto comercial del autor tenía que firmar).

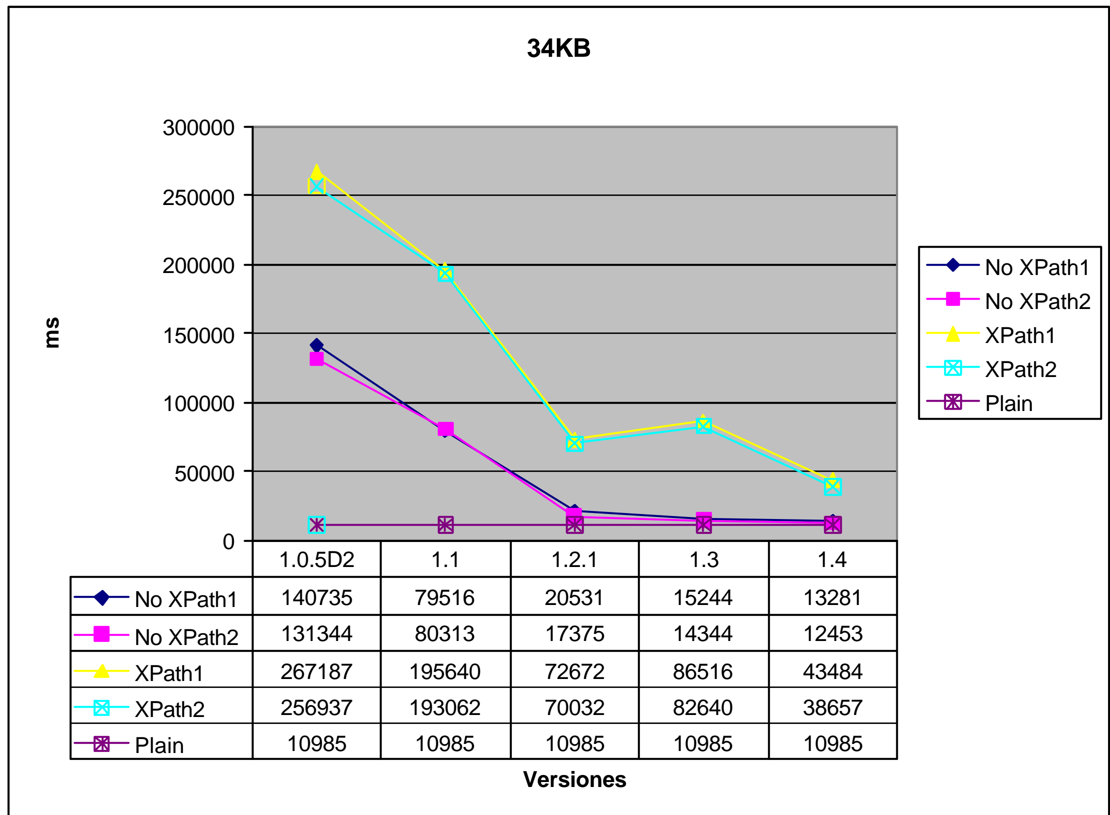
Como se puede observar en la figura siguiente, los cambios de una línea de la versión 1.0.5D2 a 1.1 hicieron la librería 4,6 veces más rápida. La siguiente versión mejoró un 30% el rendimiento. Los demás cambios han sido más modestos (entre un 5-

10% cada versión). Pero hay que observar que la asíntota es el coste de las primitivas de firmas/resumen (aunque parezca que la versión 1.4 en la `No XPath2` es más rápida que las primitivas, recordemos que cuando esta se ejecuta, es ya la vez 3000 que se ejecutarán las primitivas y que el JIT ya habrá hecho bastante trabajo sobre ellas).



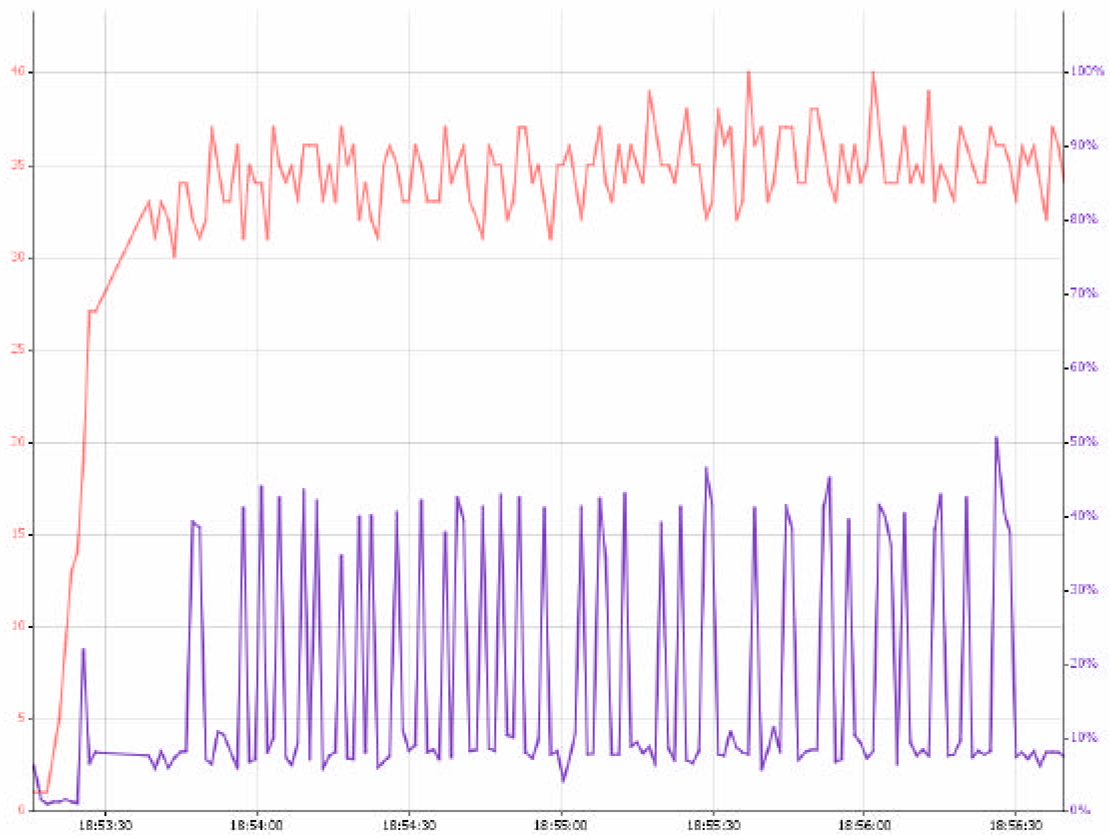
	1.0.5D2	1.1	1.2.1	1.3	1.4
No XPath1	74359	18985	13531	12078	11328
No XPath2	66578	14437	11188	10500	10172
XPath1	81344	31000	16687	18922	16594
XPath2	81375	27922	13359	15484	13328
Plain	10250	10250	10250	10250	10250

En el ejemplo siguiente, con firmas de 34 KB, las diferencias entre versiones son más acusadas. Esto es debido a que al ser el documento más grande, las diferencias en los distintos algoritmos son un poco más visibles. Lo único a remarcar es la diferencia de velocidad de un 50% en las firmas XPath entre la 1.2.1/1.3 y la 1.4. Esto es debido a que en la última versión se ha implementado el patrón visitador en las firmas que ejercitan el camino lento.

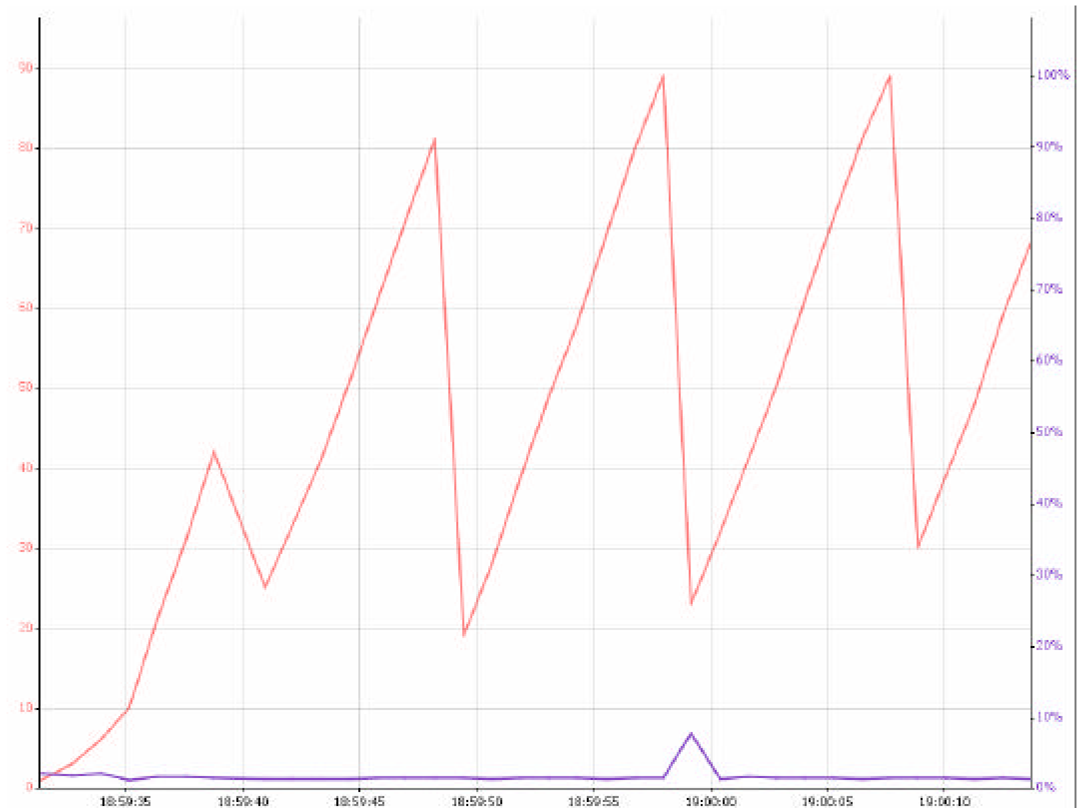


	1.0.5D2	1.1	1.2.1	1.3	1.4
No XPath1	140735	79516	20531	15244	13281
No XPath2	131344	80313	17375	14344	12453
XPath1	267187	195640	72672	86516	43484
XPath2	256937	193062	70032	82640	38657
Plain	10985	10985	10985	10985	10985

Una magnitud que no se muestra en estos gráficos es el tiempo de programa gastado en el recolector de basura. Este dato es importante para el rendimiento porque implica tiempo de ejecución que estamos perdiendo por manejar mal la memoria. En el gráfico siguiente vemos en trazo azul sobre la escala de porcentaje de la derecha el tiempo de CPU que gasta el *garbage collector* en la versión 1.1 al realizar 1000 firmas de 34 KB.El trazo naranja es el número de objetos que sobreviven a una recolección de basura.

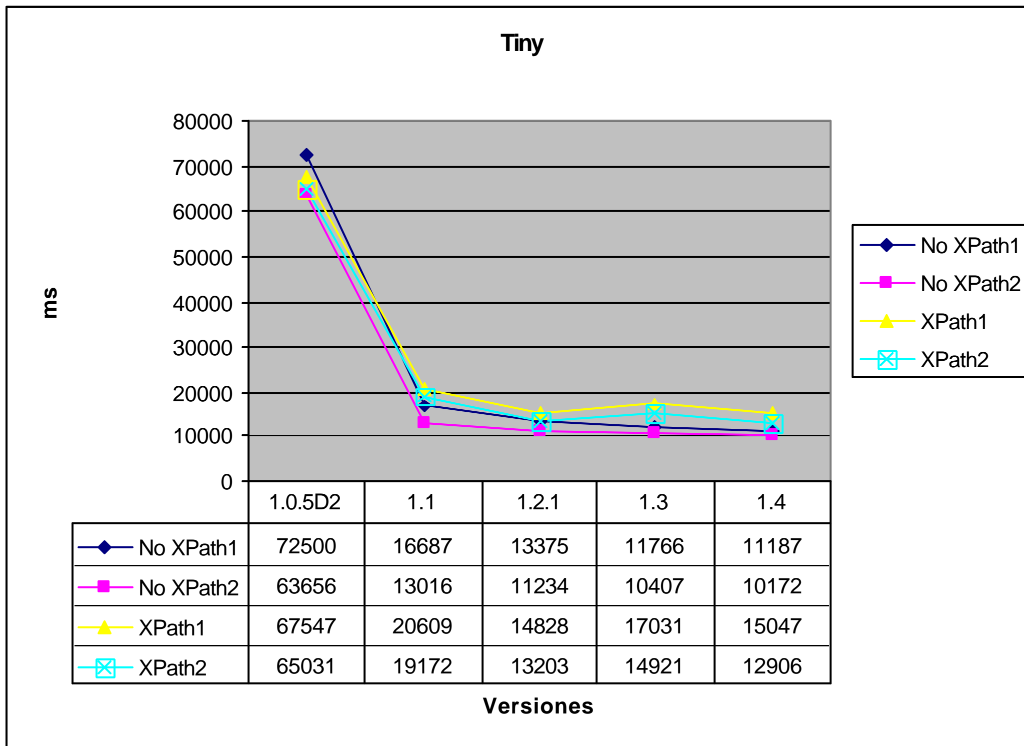


El tiempo gastado en el *garbage collector* llega en bastantes momentos a ser hasta del 40% del tiempo de ejecución. El siguiente gráfico se ha obtenido como resultado de la misma ejecución sobre la versión 1.4:



Los picos del *garbage collector* han desaparecido y solo se mantiene un gasto residual de un 2% de tiempo de ejecución.

Esta reducción es debida sobre todo a la implementación del patrón visitador.



	1.0.5D2	1.1	1.2.1	1.3	1.4
No XPath1	72500	16687	13375	11766	11187
No XPath2	63656	13016	11234	10407	10172
XPath1	67547	20609	14828	17031	15047
XPath2	65031	19172	13203	14921	12906

Este gráfico muestra que a la diferencia entre los distintos tipos de firmas es menos significativa cuanto más pequeño sea el documento.