Sistemas Distribuidos

Introducción a Ajax

Tabla de Contenidos

1.- Introducción

El término Ajax se presentó por primera vez en el artículo “Ajax: A New Approach to Web Applications” publicado por Jesse James Garrett el 18 de Febrero de 2005 [5]. Hasta ese momento, no existía un término normalizado que hiciera referencia a un nuevo tipo de aplicación web que estaba apareciendo. En realidad, el término Ajax es un acrónimo de Asynchronous JavaScript + XML, que se puede traducir como “JavaScript asíncrono + XML”.

El artículo define Ajax de la siguiente forma:

> “Ajax no es una tecnología en sí misma. En realidad, se trata de varias tecnologías independientes que se unen de formas nuevas y sorprendentes.”

En el artículo original, se indica que las tecnologías que forman AJAX son:

- XHTML (eXtensible Hypertext Markup Language) y CSS (Cascade Style Sheets), para crear una presentación basada en estándares.
- DOM (Document Object Model) y BOM (Browser Object Model), para la interacción y manipulación dinámica de la presentación, tanto en lo referido al contenido de un documento (DOM) como en lo referido al navegador o *browser* del documento (BOM).
- XML (eXtensible Markup Language), XSLT (Extensible Stylesheet Language Transformations) y JSON (JavaScript Object Notation), para el intercambio y la manipulación de información.
- XMLHttpRequest, para el intercambio asíncrono de información.
- JavaScript, para unir todas las demás tecnologías.

Y se grafica esquemáticamente la relación entre estas tecnologías tal como se muestra en la Figura 1.

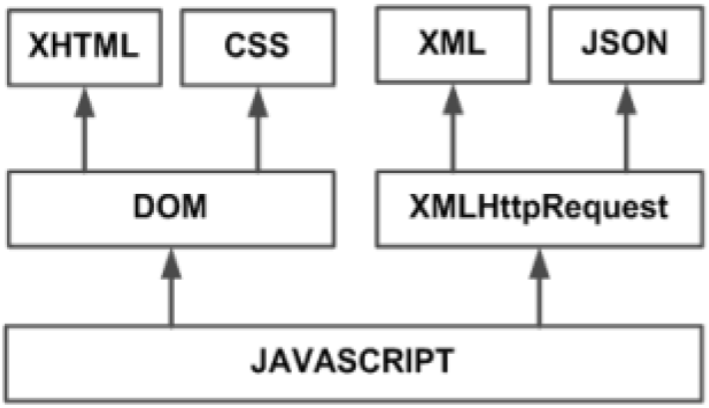

Figura 1. Tecnologías que componen Ajax.

Por otra parte, se puede apreciar la comparación propuesta por Garrett entre la tecnología tradicional Web y el uso de Ajax en lo referente a procesamiento y en el cliente y en el servidor tal como lo muestra la Figura 2. Es interesante notar que en la Figura 2 no se pone énfasis en las comunicaciones ni en la utilización del protocolo HTTP sino en el aumento de las capacidades de procesamiento del navegador, dado que tanto el servidor como la interacción a nivel de comunicaciones se muestran igual.

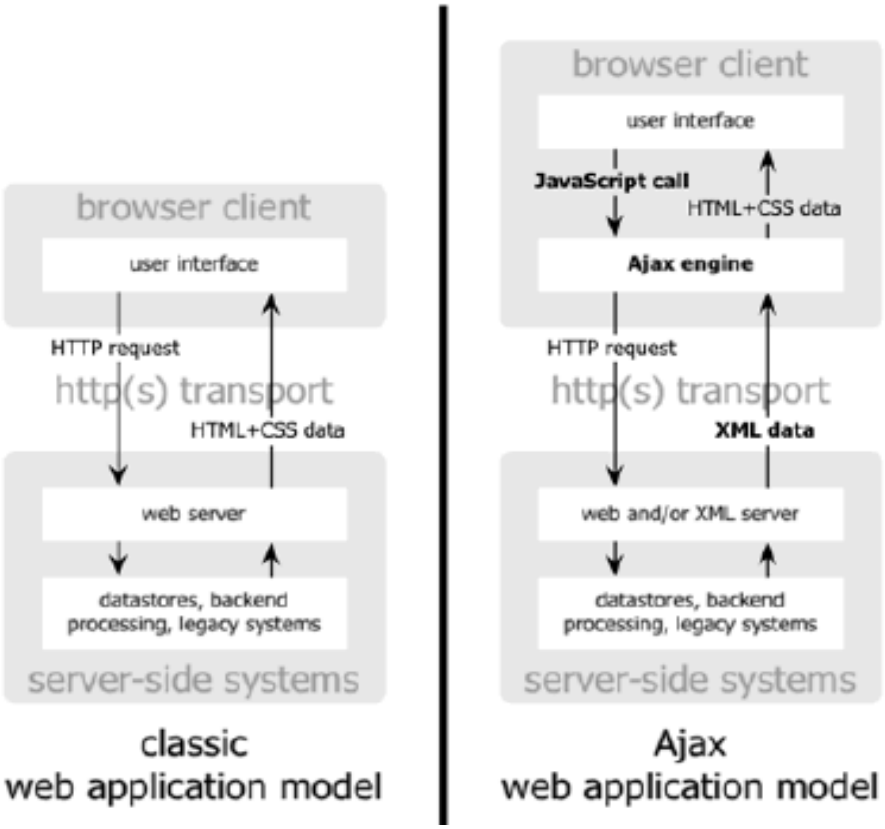

Figura 2. Comparación gráfica del modelo tradicional de aplicación web y del nuevo modelo propuesto por Ajax.

Las aplicaciones construidas con AJAX intentan reducir la recarga constante de páginas mediante la creación de un elemento intermedio entre el usuario y el servidor. La nueva capa intermedia de Ajax intenta mejorar la respuesta de la aplicación, para que el usuario nunca se encuentre con una ventana del navegador vacía esperando la respuesta del servidor. El siguiente esquema de la Figura 3, muestra la diferencia más importante entre una aplicación web tradicional y lo que se espera de una aplicación web creada con Ajax.

La imagen superior de la Figura 3 muestra la interacción sincrónica propia de las aplicaciones web tradicionales. La imagen inferior de la Figura 3 muestra la comunicación asincrónica que se espera de las aplicaciones creadas con Ajax, donde sería posible que cada interacción con el servidor se resuelva más rápidamente que en un esquema básico de requerimiento-respuesta de HTTP.

Podría afirmarse que ésta es la descripción de AJAX en su forma más *elaborada* o *completa* en el momento en que se escribe el artículo mencionado y también en cierto modo la más *optimista*. Desde un punto de vista conceptual, quizás lo más importante está dado por:

- Posibilidad de enviar un programa o código ejecutable JavaScript en vez de contenido HTML a mostrar. Esto implica un aumento cualitativo de las capacidades del navegador: no solamente se muestran documentos HTML sino que, además, se interpreta/ejecuta un lenguaje de programación.

- gador, mplica ambio

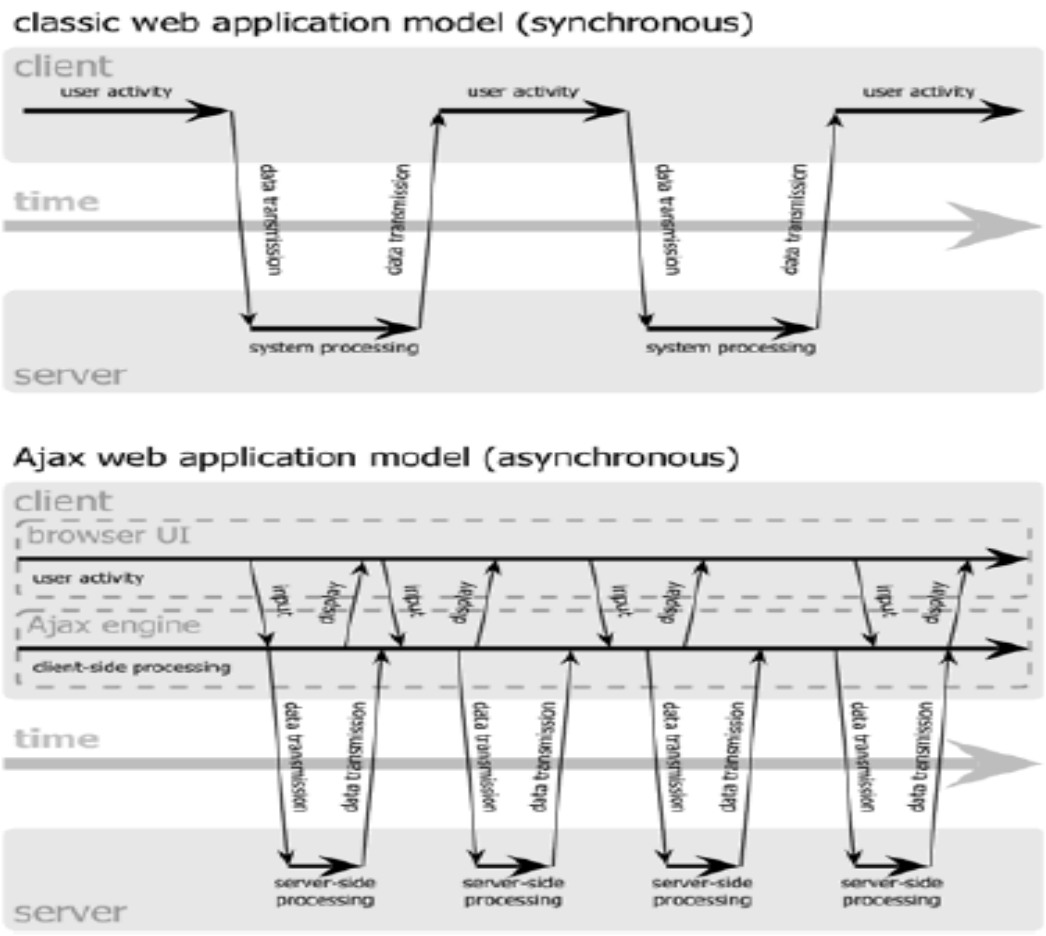

Figura 3. Comparación de las comunicaciones.

- Posibilidad de manejar de manera asíncrona y programática la interacción con el servidor web, es decir independientemente (en tiempo relativo) de los eventos que produce el usuario y se capturan en el navegador.

Sin entrar en todos los detalles más complejos, a continuación se verán justamente estas características. Se comenzará con lo más sencillo en cuanto a código JavaScript incluido en documentos HTML, que acceden al DOM para cambiar el contenido del documento que muestra el navegador y se avanzará hacia la interacción asincrónica con el servidor HTTP.

2.- JavaScript (ECMAScript)

Aunque es conocido popularmente como JavaScript, el lenguaje que la gran mayoría de los navegadores es capaz de ejecutar y que se utiliza también en la gran mayoría de las aplicaciones web actuales es ECMAScript. La organización encargada de definirlo [2] reconoce esta realidad oficialmente en [3], donde se aclara que en realidad JavaScript es una definición de la empresa Netscape. Por supuesto no es dificil identificar la relación entre los lenguajes y la influencia de JavaScript-Netscape sobre el "actual JavaScript", ECMAScript. Por razones de popularidad, se seguirán utilizando como sinónimos JavaScript y ECMAScript.

Sin entrar en los detalles de caracterización de JavaScript como lenguaje (scripting, sintaxis, aspectos dinámicos, aspectos de orientación a objetos, etc.), se avanzará directamente hacia el uso de JavaScript en páginas web dinámicas, comenzando por el acceso al DOM. En cierta forma, el acceso al DOM es lo más inmediato que se puede ver y probar con JavaScript, dado que el resultado puede ser directamente visible en un documento HTML que se muestra en un navegador con JavaScript habilitado. Entre las primeras características interesantes de JavaScript se pueden mencionar:

- El código JavaScript puede ser directamente incluido en un documento HTML. De hecho esto lo hace directamente código movil (movilidad *débil*).
- El código JavaScript es ejecutado por el navegador que interpreta y muestra un documento HTML.
- No es necesario declarar los tipos de variables que van a utilizarse (*loose typing*).
- Las referencias a objetos se comprueban en tiempo de ejecución.
- No puede escribir automáticamente al disco duro.

El fragmento de página web que se muestra en la Figura 4 es un ejemplo de cómo se puede incluir código JavaScript en un documento HTML. Todo código JavaScript incluido en un documento HTML estará identificado entre las etiquetas *<script type="text/javascript">* y *</script>*.

```
<script type="text/javascript">
    document.write("Hello World!");
</script>
```

Figura 4. Fragmento de Código JavaScript a incluir en un Documento HTML.

En la Figura 4 se tiene un ejemplo (que suele ser el primero) de acceso al DOM con código JavaScript:

- Se referencia al objeto *document*, que es la forma de indicar el documento que el navegador está mostrando en el momento de la ejecución del código JavaScript.
- Se invoca el método *write()* de *document* que, justamente, produce la salida de un texto (*Hello World!*, en este caso) en el documento.

También en la Figura 4 queda más claro por qué se suele utilizar el nombre JavaScript en vez de ECMAScript para este lenguaje: en el código HTML se identifica directamente con el tipo de script "*text/javascript*" y es lo que normalmente los diseñadores y programadores tienden a recordar.

Si bien hay varias partes alternativas para incluir código JavaScript en un documento

HTML, en principio puede ponerse directamente en el *body*, como en el caso de ejemplos tan sencillos como el de la Figura 4. La Figura 5 muestra lo que podría ser un documento HTML completo, que un navegador con ejecución de JavaScript habilitada podría mostrar. Si el texto de la Figura 4 se pone como contenido un documento HTML tal como index.html, un navegador lo mostraría tal como en la Figura 6.

```
<html>
<body>
    <script type="text/javascript">
        document.write("Hello World!");
    </script>
</body>
</html>
```

Figura 5. Código JavaScript en el body de un Documento HTML.

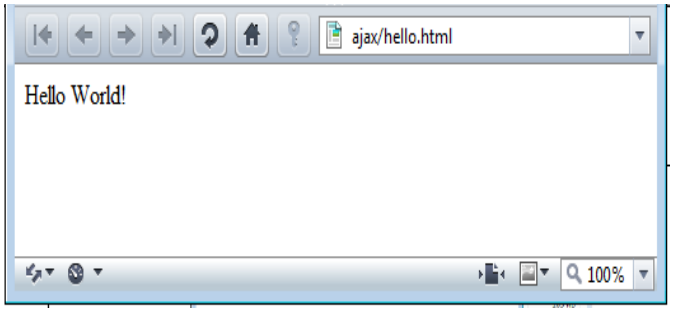

Figura 6. Documento HTML con JavaScript Mostrado por un Navegador.

Como se puede ver en el ejemplo, el texto queda como parte del documento y de hecho es lo único que hay/se muestra en el cuerpo del documento. Equivaldría a un HTML con un único párrafo que se muestra como texto *estándar*. Pero se tienen múltiples alternativas, entre ellas poner en el documento texto con tags de HTML, que el navegador interpretará y mostrará como si fueran parte del documento HTML. La Figura 7 muestra un ejemplo, donde al ejemplo anterior se le agrega un título de nivel 1.

```
<html>
<body>
    <script type="text/javascript">
        document.write("Hello World!");
        document.write("<h1>Title</h1>");
    </script>
</body>
</html>
```

Figura 7. Código JavaScript que Escribe con tags de HTML.

La Figura 8 muestra cómo se interpretan los tags de la manera usual, como si estuvieran incluidos normalmente en un documento HTML estático.

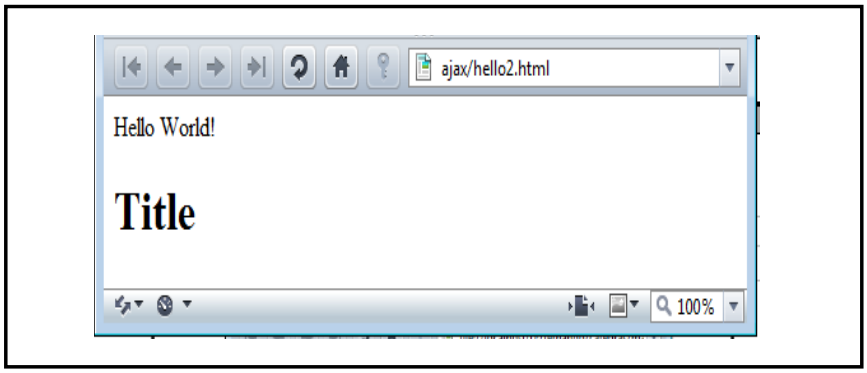

Figura 8. JavaScript con tags de HTML Interpretados.

En el siguiente capítulo se mostrarán más detalles útiles del DOM para ser utilizados con JavaScript y AJAX, en lo que sigue se darán algunos detalles más propios de JavaScript como son las declaraciones de variables, utilización de clases predefinidas, etc. En la descripción que sigue, se asumirá que JavaScript es un lenguaje orientado a objetos básicamente debido a la extensa literatura disponible que lo presenta como tal. La discusión conceptual de JavaScript como lenguaje orientado a objetos se considera fuera del alcance de este apunte.

2.1 Funciones, Variables, Comentarios, etc.

El ejemplo de la Figura 9 tiene varios detalles incluidos que muestran características específicas de JavaScript como lenguaje de programación. La gran mayoría de los detalles incluidos en la Figura 9 son completamente conocidos para los programadores tanto en sintaxis como en semántica. Como suele pasar con la mayoría (sino todos) los lenguajes de programación con tipos y/o clases predefinidos, el aprovechamiento completo depende, justamente, de conocer lo que se tiene disponible de manera predefinida. En el ejemplo de la Figura 9 se crea y se utiliza un objeto *Date* y en función del valor de la hora obtenida a partir del objeto *Date*, con *getHours()*, se muestra un mensaje u otro. Detalles completos de los objetos *Date* se pueden encontrar en [19].

Aunque hasta el momento se ha incluído código JavaScript en el *body* de un documento HTML, también se puede incluir en el *head* del mismo. En [20] se dan diferencias básicas en cuanto a la sección del documento HTML en el que se incluye código JavaScript que no son completamente correctas desde el punto de vista técnico, aunque podrían tomarse como principios útiles para documentos con JavaScript:

- El código JavaScript incluído en el *body* siempre se ejecuta a menos que esté dentro de una función que no se llama en tiempo de ejecución.
- El código JavaScript incluído en el *body* se ejecuta mientras el documento HTML es cargado.
- El código JavaScript incluído en el *head* debe estar incluido en una función y se ejecuta cuando la función es invocada en tiempo de ejecución.
- Normalmente se utilizan las funciones JavaScript incluidas en el *head* una vez que

que se ha asegurado que todo el documento HTML está cargado.

```
<html>
    <body>
<script type="text/javascript">
    //Este es un ejemplo de comentario de una línea

    //Si es muy "temprano", muestra una cosa
    //sino, muestra otra

    //Declaración de variables
    //Utilización de clases y métodos predefinidos
    var fecha = new Date();
    var hora = fecha.getHours();
    if (hora < 7)
      { document.write("Es muy temprano..."); }
    else
      { document.write("Listo para empezar..."); }
</script>
    </body>
</html>
```

Figura 9. Detalles del Lenguaje JavaScript.

Sin embargo, en el ejemplo de la Figura 10 se puede ver que hay código JavaScript tanto en la sección *head* como en la sección *body* del documento HTML y en ningún caso hay funciones definidas.

```
<html>
   <head>
      <title>Ejemplo de JavaScript en head y body</title>
      <script type="text/javascript">
         document.write("El titulo original es: " + document.title + "<br>");
         document.title = "Titulo asignado con JavaScript";
      </script>
   </head>
   <body>
      <script type="text/javascript">
         document.write("El titulo en el body es: " + document.title + "<br>");
      </script>
   </body>
</html>
```

Figura 10. Código JavaScript en el heady y en el body de un Documento HTML.

La Figura 11 muestra el resultado de la ejecución del código JavaScript que se produce en un navegador. Es interesante notar que, tal como parece intuitivo, el código de la sección *head* no solamente se ejecuta sino que se ejecuta antes que el código de la sección *body*.

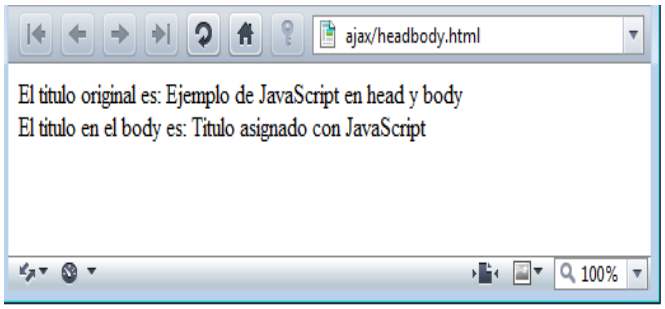

Figura 11. Ejecución de JavaScript en las Secciones head y body.

Algunos otros detalles de la ejecución de JavaScript se dan en [12] [9]. Una de las formas más sencillas de ilustrar tanto la sintaxis y ejecución de funciones así como también el uso de JavaScript para manejar o reaccionar frente a determinados eventos es la que se muestra en la Figura 12.

```
<html>
    <head>
<script type="text/javascript">
     function paraelboton()
     {
        document.write("<p>Cambia el documento</p>");
        alert("Esto aparece como un alert");
     }
</script>
    </head>
    <body>
<form>
    <input type = "button"
            value = "Click para ver"
            onclick ="paraelboton()"
</form>
    </body>
</html>
```

Figura 12. Una Función JavaScript en la Sección head.

Quizás este ejemplo también ilustra parte de la capacidad de JavaScript para generar contenido dinámico y asincrónico en una página web. Se genera contenido dinámico dado que el contenido del documento que se muestra en el navegador cambia en función de una acción del usuario y se lleva a cabo localmente en el navegador, sin ningún intercambio con el servidor web. La Figura 13 muestra el documento HTML ni bien se carga en el navegador (a la izquierda) y posteriormente a hacer click sobre el botón del formulario (a la derecha), donde aparece el *alert()* y el contenido del documento ya no tiene el formulario sino el texto que se escribe en la función JavaScript *paraelboton()*.

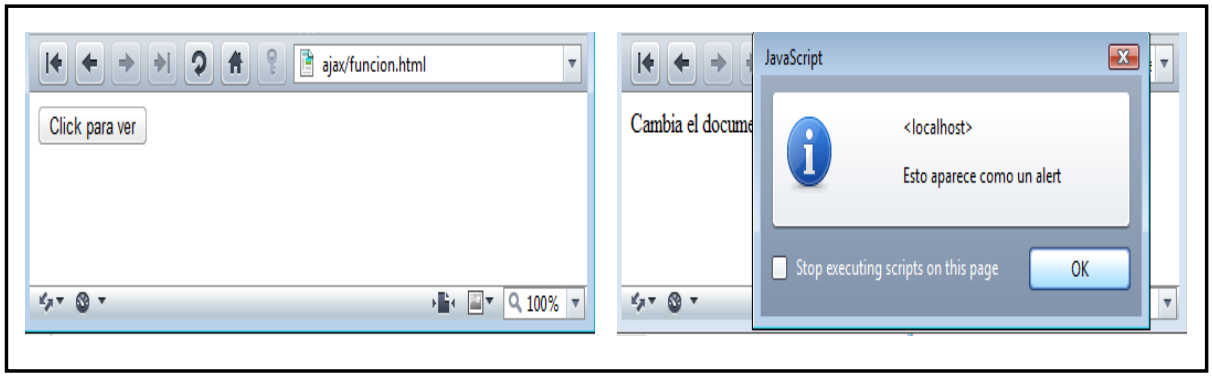

Figura 13. Navegador Antes y Después de la Ejecución de la Función JavaScript.

Otras de las características que también quedan ejemplificadas en este último ejemplo de código JavaScript son:

- La capacidad de asociar explícitamente código JavaScript a determinados eventos que se producen y se capturan en el navegador.
- La combinación en un documento de texto HTML con sus tags asociados, específicamente en la sección *body*, con código JavaScript en la sección *head*. Es importante notar que la asociación evento-código JavaScript a ejecutar es similar en cierta forma a lo que se tiene en un formulario con el atributo *action* de un *form*, donde se indica explícitamente el script CGI a ejecutar en el servidor.

Si bien hay muchos detalles de JavaScript que no se han mostrado, se puede considerar que la descripción dada hasta aquí es suficiente para continuar completando los conceptos necesarios para aplicaciones web con Ajax. En las secciones siguientes se agregarán detalles del DOM así como también de los aspectos más relevantes de la comunicación asincrónica con el servidor web.

3.- DOM (Document Objet Model)

DOM es una definición que permite a las aplicaciones contar con una interfaz para acceder a la información y estructura de documentos, proporcionando un conjunto estándar de objetos para representar documentos HTML y XML. Es un modelo estándar sobre cómo pueden combinarse los componentes de los documentos representados como objetos, y una interfaz estándar para acceder a ellos y manipularlos. Si bien las utilidades fueron específicamente diseñadas para manipular documentos XML, por extensión DOM también se puede utilizar para manipular documentos XHTML y HTML. Técnicamente, DOM es una API de funciones que se pueden utilizar para manipular las páginas XHTML de forma rápida y eficiente.

Antes de poder utilizar las funciones del DOM, quienes son encargados de interpretar y manejar los documentos (navegadores, por ejemplo) transforman internamente el archivo XML original en una estructura más fácil de manejar formada por una jerarquía de nodos. De esta forma, se transforma el código XML en una serie de nodos interconectados en forma de árbol. El árbol generado no sólo representa los contenidos del archivo original (mediante los nodos del árbol) sino que también representa sus relaciones (mediante las ramas del árbol que conectan los nodos). Aunque en ocasiones DOM se asocia con la programación web y con JavaScript, la API de DOM es independiente de cualquier lenguaje de programación. De hecho, DOM está disponible en la mayoría de lenguajes de programación comúnmente empleados.

En el contexto de las aplicaciones web, es importante comprender que es responsabilidad del navegador implementar la interfaz del DOM. En definitiva es el navegador quien provee/implementa la interfaz del DOM. Si se considera el documento HTML sencillo de la Figura 14, Antes de poder utilizar las funciones del DOM, los navegadores convierten de manera automática el documento en la estructura de árbol de nodos que se muestra en la Figura 15. La ventaja de emplear DOM es que permite a los programadores disponer de un control muy preciso sobre la estructura del documento HTML o XML que están manipulando. Las funciones que proporciona DOM permiten añadir, eliminar, modificar y reemplazar cualquier nodo de cualquier documento de forma sencilla.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
   Transitional//EN" "http://www.w3.org/TR/
   xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="Content-Type" content="text/html;
         charset=iso-8859-1" />
      <title>Página sencilla</title>
   </head>
   <body>
      <p>Esta página es <strong>muy sencilla</strong></p>
   </body>
</html>
```

Figura 14. Ejemplo de Documento HTML.

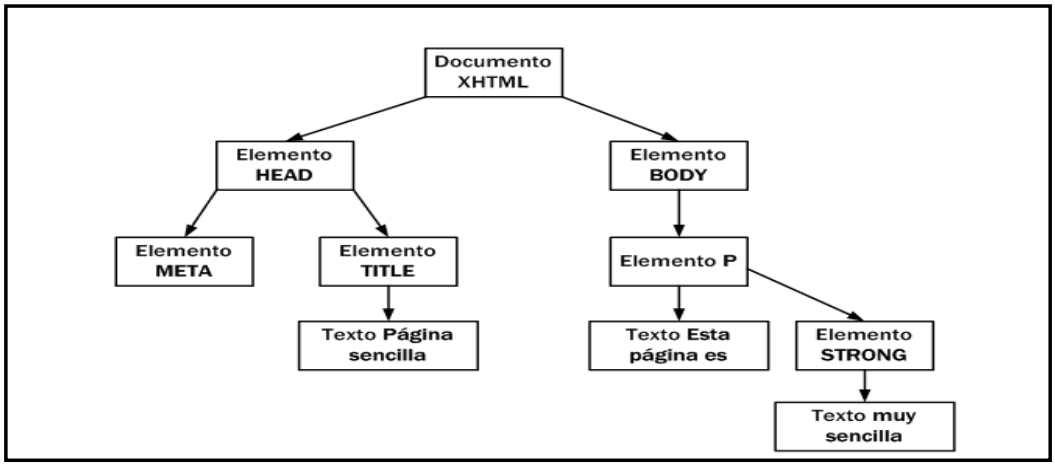

Figura 15. Representación en forma de árbol de la página HTML de ejemplo

Se puede ver otro ejemplo en [11]. El objeto document, que es la raíz del árbol del DOM tiene definidos numerosos atributos y funciones/métodos, de los que se ha usado el write(), tal como se describe en [13] [14] [15] [8] [10] [1] (las referencias están dadas en orden de manera tal que las más técnicamente completas y menos explicadas están al principio las siguientes avanzan en nivel de explicación pero también recortan partes de la especificación).

3.1 Tipos de Nodos

Los documentos XML y HTML tratados por quienes construyen el DOM se convierten en una jerarquía de nodos. Los nodos que representan los documentos pueden ser de diferentes tipos. A continuación se detallan los tipos más importantes:

- **Document**: es el nodo raíz de todos los documentos HTML y XML. Todos los demás nodos derivan de él.
- **DocumentType**: es el nodo que contiene la representación del DTD (Document Type Definition) empleado en la página (indicado mediante el DOCTYPE).
- **Element**: representa el contenido definido por un par de etiquetas de apertura y cierre (<etiqueta>...</etiqueta>) o de una etiqueta abreviada que se abre y se cierra a la vez (<etiqueta/>). Es el único nodo que puede tener tanto nodos hijos como atributos.
- **Attr**: representa el par nombre-de-atributo/valor.
- **Text**: almacena el contenido del texto que se encuentra entre una etiqueta de apertura y una de cierre. También almacena el contenido de una sección de tipo CDATA.
- **CDataSection**: es el nodo que representa una sección de tipo <![CDATA[ ]]>.
- **Comment**: representa un comentario de XML.

3.2 La Interfaz NODE

Una vez que se ha creado de forma automática el árbol completo de nodos del DOM de la página, ya es posible utilizar las funciones del DOM para obtener información sobre los nodos o manipular su contenido. En JavaScript se crea el objeto Node para definir las propiedades y métodos

necesarios para procesar y manipular los documentos.

En primer lugar, el objeto Node define las constantes que se muestran en la Figura 16 para la identificación de los distintos tipos de nodos.

```
Node.ELEMENT_NODE = 1                          Node.PROCESSING_INSTRUCTION_NODE = 7
Node.ATTRIBUTE_NODE = 2                        Node.COMMENT_NODE = 8
Node.TEXT_NODE = 3                             Node.DOCUMENT_NODE = 9
Node.CDATA_SECTION_NODE = 4                    Node.DOCUMENT_TYPE_NODE = 10
Node.ENTITY_REFERENCE_NODE = 5                 Node.DOCUMENT_FRAGMENT_NODE = 11
Node.ENTITY_NODE = 6                           Node.NOTATION_NODE =  12
```

Figura 16: Constantes para Identificación de Titpos de Nodos del DOM.

Para los objetos Node del DOM están definidas las propiedades y métodos que se muestran en la Figura 17.

Propiedad/Método	Valor devuelto	Descripción
nodeName	String	El nombre del nodo (no está definido para algunos tipos de nodo)
nodeValue	String	El valor del nodo (no está definido para algunos tipos de nodo)
nodeType	Number	Una de las 12 constantes definidas anteriormente
ownerDocument	Document	Referencia del documento al que pertenece el nodo
firstChild	Node	Referencia del primer nodo de la lista childNodes
lastChild	Node	Referencia del último nodo de la lista childNodes
childNodes	NodeList	Lista de todos los nodos hijo del nodo actual
previousSibling	Node	Referencia del nodo hermano anterior o null si este nodo es el primer hermano
nextSibling	Node	Referencia del nodo hermano siguiente o null si este nodo es el último hermano
hasChildNodes()	Bolean	Devuelve true si el nodo actual tiene uno o más nodos hijo
attributes	NamedNodeMap	Se emplea con nodos de tipo Element. Contiene objetos de tipo Attr que definen todos los atributos del elemento
appendChild(nodo)	Node	Añade un nuevo nodo al final de la lista childNodes
removeChild(nodo)	Node	Elimina un nodo de la lista childNodes
replaceChild(nuevoNodo, anteriorNodo)	Node	Reemplaza el nodo anteriorNodo por el nodo nuevoNodo
insertBefore(nuevoNodo, anteriorNodo)	Node	Inserta el nodo nuevoNodo antes que la posición del nodo anteriorNodo dentro de la lista childNodes

Figura 17. Propiedades y métodos de Node.

3.3 HTML, DOM y JavaScript

Hasta este punto, se han mostrado detalles importantes de la representación del DOM para un documento HTML y algunas características de los nodos del DOM en cuanto a propiedades y funciones de acceso disponibles definidas por el mismo DOM. En las aplicaciones web es necesario, entonces, manipular estos elementos del DOM para llevar a cabo lo relativo a las propias aplicaciones.

3.3.1 Acceso Relativo a los Nodos

La operación básica para acceder a los nodos o componentes de un documento HTML con JavaScript consiste en obtener el objeto que representa el elemento raíz de la página:

```
var objeto_html = document.documentElement;
```

Se debe notar que *documentElement* es una propiedad del objeto *document*, y después de ejecutar esta instrucción, la variable *objeto_html* contiene un objeto de tipo *HTMLElement* que representa el elemento <html> de la página web. Según el árbol de nodos DOM, desde el nodo <html> derivan dos nodos del mismo nivel jerárquico: <head> y <body>.

Utilizando los métodos proporcionados por DOM, es sencillo obtener los elementos <head> y <body>. En primer lugar, los dos nodos se pueden obtener como el primer y el último nodo hijo del elemento <html>:

```
var objeto_head = objeto_html.firstChild;
var objeto_body = objeto_html.lastChild;
```

Otra forma directa de obtener los dos nodos consiste en utilizar la propiedad *childNodes* del elemento <html>:

```
var objeto_head = objeto_html.childNodes[0];
var objeto_body = objeto_html.childNodes[1];
```

Si se desconoce el número de nodos hijo que dispone un nodo, se puede emplear la propiedad *length* de childNodes:

```
var numDescendientes = objeto_html.childNodes.length;
```

Además, el DOM de HTML permite acceder directamente al elemento <body> utilizando document.body:

```
var objeto_body = document.body;
```

También existen otras propiedades como *previousSibling* y *parentNode* que se pueden utilizar para acceder a un nodo a partir de otro. Utilizando estas propiedades, se pueden comprobar las siguientes igualdades:

```
objeto_head.parentNode == objeto_html
objeto_body.parentNode == objeto_html
objeto_body.previousSibling == objeto_head
objeto_head.nextSibling == objeto_body
objeto_head.ownerDocument == document
```

3.3.2 Atributos

Además del tipo de etiqueta HTML y su contenido de texto, la definición permite el acceso directo a todos los atributos de cada etiqueta. Para ello, los nodos de tipo *Element* contienen la propiedad *attributes*, que permite acceder a todos los atributos de cada elemento. Aunque técnicamente la propiedad *attributes* es de tipo NamedNodeMap, sus elementos se pueden acceder como si fueran elementos de un array. Por medio del DOM se proporcionan diversos métodos para tratar con los atributos:

- **getNamedItem**(nombre), devuelve el nodo cuya propiedad nodeName contenga el valor nombre.
- **removeNamedItem**(nombre), elimina el nodo cuya propiedad nodeName coincida con el valor nombre.
- **setNamedItem**(nodo), añade el nodo a la lista attributes, indexándolo según su propiedad nodeName.
- **item**(posicion), devuelve el nodo que se encuentra en la posición indicada por el valor numérico posición.

Los métodos anteriores devuelven un nodo de tipo *Attr*, por lo que no devuelven directamente el valor del atributo. Utilizando estos métodos, es posible procesar y modificar fácilmente los atributos de los elementos HTML, tal como se muestra en el ejemplo de la Figura 18.

La Figura 19 muestra el resultado del procesamiento del documento HTML de la Figura 18 por parte de un navegador. Sin embargo, también el DOM proporciona otros métodos que permiten el acceso y la modificación de los atributos de forma más directa:

- **getAttribute**(nombre), es equivalente a attributes.getNamedItem(nombre).
- **setAttribute**(nombre, valor) equivalente a attributes.getNamedItem(nombre).value = valor.
- **removeAttribute**(nombre), equivalente a attributes.removeNamedItem(nombre).

Utilizando estos métodos, el ejemplo anterior se puede reescribir también de manera más simple el ejemplo de la Figura 17, porque en vez de seguir la “secuencia”

```
p.attributes.getNamedItem("id").nodeValue = "otraintroduccion";
```

se puede utilizar *directamente*:

```
p.setAttribute("id", "preintroduccion");
```

```
<html>
  <body>
    <p id="introduccion" style="color: blue">Párrafo de prueba</p>

    <script type="text/javascript">
       var p = document.getElementById("introduccion");
       var elId = p.attributes.getNamedItem("id").nodeValue; // elId = "introduccion"
       document.write(p.attributes.getNamedItem("id").nodeValue + "<br>");

       var elId = p.attributes.item(0).nodeValue; // elId = "introduccion"
       document.write(p.attributes.item(0).nodeValue + "<br>");

       p.attributes.getNamedItem("id").nodeValue = "otraintroduccion";
       document.write(p.attributes.item(0).nodeValue + "<br>");

       var atributo = document.createAttribute("lang");
       atributo.nodeValue = "es";
       p.attributes.setNamedItem(atributo);
       document.write("El valor de lang es: " + atributo.nodeValue);
    </script>

    </body>
</html>
```

Figura 18. Procesamiento de los Atributos de Elementos HTML.

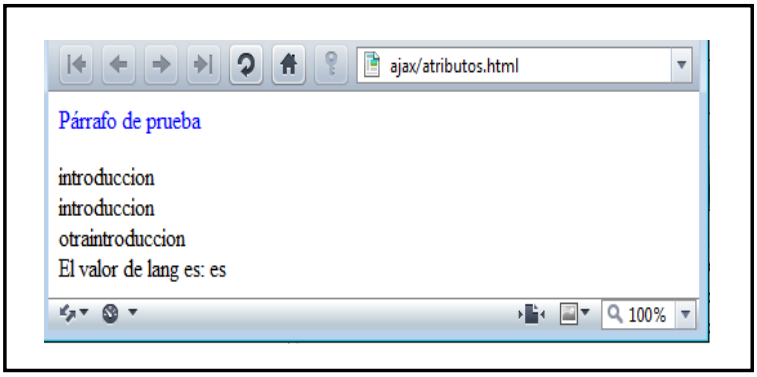

Figura 19. Resultado del Procesamiento de Atributos con JavaScript.

3.3.3 Acceso Directo a los Nodos

Los métodos presentados hasta el momento permiten acceder a cualquier nodo del árbol de nodos DOM y a todos sus atributos. Sin embargo, las funciones que proporciona DOM para acceder a un nodo a través de sus padres obligan a acceder al nodo raíz de la página y después a sus nodos hijos y a los nodos hijos de esos hijos y así sucesivamente hasta el último nodo de la rama terminada por el nodo buscado. Cuando se trabaja con una página web real, el árbol DOM tiene miles de nodos de todos los tipos. Por este motivo, no es eficiente acceder a un nodo descendiendo a través de todos los ascendentes de ese nodo. Para solucionar este problema, DOM proporciona una serie de

métodos para acceder de forma directa a los nodos deseados. Los métodos disponibles son:

getElementsByTagName()

La función getElementsByTagName() obtiene todos los elementos de la página XHTML cuya etiqueta sea igual que el parámetro que se le pasa a la función.

El valor que devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. En realidad, el valor devuelto no es de tipo array normal, sino que es un objeto de tipo NodeList. De este modo, el primer párrafo de la página se puede obtener de la siguiente manera:

```
var parrafos = document.getElementsByTagName("p");

var primerParrafo = parrafos[0];
```

De la misma forma, se pueden recorrer todos los párrafos de la página recorriendo el *array* de nodos devuelto por la función:

```
var parrafos = document.getElementsByTagName("p");

for(var i=0; i<parrafos.length; i++) {
        var parrafo = parrafos[i];
}
```

La función getElementsByTagName() se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función. En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
var parrafos = document.getElementsByTagName("p");

var primerParrafo = parrafos[0];

var enlaces = primerParrafo.getElementsByTagName("a");
```

getElementsByName()

La función getElementsByName() obtiene todos los elementos de la página XHTML cuyo atributo name coincida con el parámetro que se le pasa a la función. Normalmente el atributo name es único para los elementos HTML que lo incluyen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML *radiobutton*, el atributo name es común a todos *radiobutton* que estén relacionados, por lo que la función devuelve una colección de elementos. Internet Explorer 7 y sus versiones anteriores no implementan de forma correcta esta función, ya que también devuelven los elementos cuyo atributo id sea igual al parámetro de la función.

```
var parrafoEspecial = document.getElementsByName("especial");

<p name="prueba">...</p>

<p name="especial">...</p>

<p>...</p>
```

getElementById().

La función getElementById() es de la funciones más utilizadas cuando se desarrollan aplicaciones web dinámicas. Se trata de la función preferida para acceder directamente a un nodo y para leer o modificar sus propiedades. La función getElementById() devuelve el elemento XHTML cuyo atributo id coincide con el parámetro indicado en la función. Como el atributo id debe ser único para cada elemento de una misma página, la función devuelve únicamente el nodo deseado.

```
var cabecera = document.getElementById("cabecera");

<div id="cabecera">
<a href="/" id="logo">...</a>
</div>
```

3.4 Crear, Modificar y Eliminar Nodos

Los métodos disponibles por el DOM para la creación de nuevos nodos son los que se muestran en la Figura 20.

Método	Descripción
`createAttribute(nombre)`	Crea un nodo de tipo atributo con el nombre indicado
`createCDataSection(texto)`	Crea una sección CDATA con un nodo hijo de tipo texto que contiene el valor indicado
`createComment(texto)`	Crea un nodo de tipo comentario que contiene el valor indicado
`createDocumentFragment()`	Crea un nodo de tipo `DocumentFragment`
`createElement(nombre_etiqueta)`	Crea un elemento del tipo indicado en el parámetro `nombre_etiqueta`
`createEntityReference(nombre)`	Crea un nodo de tipo `EntityReference`
`createProcessingInstruction(objetivo, datos)`	Crea un nodo de tipo `ProcessingInstruction`
`createTextNode(texto)`	Crea un nodo de tipo texto con el valor indicado como parámetro

Figura 20. Métodos de la interfaz para crear nodos DOM.

Es importante recordar que las modificaciones en el árbol de nodos DOM sólo se pueden realizar cuando toda la página web se ha cargado en el navegador. El motivo es que los

navegadores construyen el árbol de nodos DOM una vez que se ha cargado completamente la página web. Cuando una página no ha terminado de cargarse, su árbol no está construido y por tanto no se pueden utilizar las funciones DOM relativas a la modificación del árbol. Además de crear, eliminar y sustituir nodos, las funciones del DOM permiten insertar nuevos nodos antes o después de otros nodos ya existentes. Si se quiere insertar un nodo después de otro, se emplea la función **appendChild()**, que está definido para todos los diferentes tipos de nodos y se encarga de añadir un nodo al final de la lista *childNodes* de otro nodo.

Para eliminar cualquier nodo, se emplea la función **removeChild()**, que toma como argumento la referencia al nodo que se quiere eliminar. La función **removeChild()** se debe invocar sobre el nodo padre del nodo que se va a eliminar.

Cuando la página está formada por miles de nodos, puede ser costoso acceder hasta el nodo padre del nodo que se quiere eliminar. En estos casos, se puede utilizar la propiedad **parentNode**, que siempre hace referencia al nodo padre de un nodo.

```
var p = document.getElementsByTagName("p")[0];
p.parentNode.removeChild(p);
```

Además de crear y eliminar nodos, las funciones DOM también permiten reemplazar un nodo por otro. Utilizando las funciones DOM de JavaScript, se crea el nuevo párrafo que se va a mostrar en la página, se obtiene la referencia al nodo original y se emplea la función **replaceChild()** para intercambiar un nodo por otro:

```
var nuevoP = document.createElement("p");
var texto = document.createTextNode("Este parrafo se ha creado
dinámicamente y sustituye al parrafo original");
nuevoP.appendChild(texto);

var anteriorP = document.body.getElementsByTagName("p")[0];
anteriorP.parentNode.replaceChild(nuevoP, anteriorP);
```

Por último, es importante mencionar que DOM proporciona métodos específicos para trabajar con tablas. Si se utilizan los métodos tradicionales, crear una tabla es una tarea tediosa, por la gran cantidad de nodos de tipo elemento y de tipo texto que se deben crear. Sin embargo, la versión de DOM para HTML incluye varias propiedades y métodos para crear tablas, filas y columnas de forma sencilla. Propiedades y métodos de <table> se muestran en la Figura 21.

Propiedad/Método	Descripción
rows	Devuelve un array con las filas de la tabla
tBodies	Devuelve un array con todos los <tbody> de la tabla
insertRow(posicion)	Inserta una nueva fila en la posición indicada dentro del array de filas de la tabla
deleteRow(posicion)	Elimina la fila de la posición indicada

Figura 21. Propiedades y Métodos de <table>.

Propiedades y métodos de <tr> se muestran en la Figura 22.

Propiedad/Método	Descripción
cells	Devuelve un array con las columnas de la fila seleccionada
insertCell(posicion)	Inserta una nueva columna en la posición indicada dentro del array de columnas de la fila
deleteCell(posicion)	Elimina la columna de la posición indicada

Figura 22. Propiedades y Métodos de <tr>.

Se puede ver una lista completa de métodos asociados a nodos en [7], donde también se muestran muchos ejemplos asociados a cada uno de ellos.

4.- BOM (Browser Objet Model)

Las últimas versiones de los navegadores introdujeron el concepto de Browser Object Model o BOM, que permite acceder y modificar las propiedades de las ventanas del propio navegador. En general, podría afirmarse que el BOM es el DOM de los navegadores, dado que mediante DOM se accede/procesa el documento y gracias al BOM se procesa/accede al propio navegador. Mediante BOM, es posible re-dimensionar y mover la ventana del navegador, modificar el texto que se muestra en la barra de estado y realizar muchas otras manipulaciones no relacionadas con el contenido de la página HTML sino con el navegador mismo. El mayor inconveniente de BOM es que, al contrario de lo que sucede con DOM, ninguna entidad se encarga de estandarizarlo o definir unos mínimos de inter-operabilidad entre navegadores. Algunos de los elementos que forman el BOM son los siguientes:

- Crear, mover, redimensionar y cerrar ventanas de navegador.
- Obtener información sobre el propio navegador.
- Propiedades de la página actual y de la pantalla del usuario.
- Gestión de cookies.
- Objetos ActiveX en Internet Explorer.

La Figura 23 muestra la jerarquía de objetos que forman el BOM, donde se incluye el DOM a partir del nodo *document*.

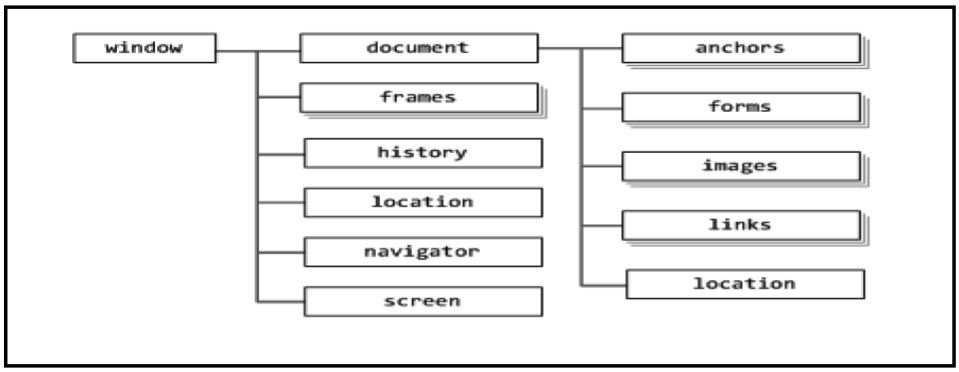

Figura 23. Jerarquía de Objetos que Forman el BOM.

En el esquema anterior, los objetos mostrados con varios recuadros superpuestos son arrays. El resto de objetos, representados por un rectángulo individual, son objetos simples. En cualquier caso, todos los objetos derivan del objeto *window*., lo cual es similar al DOM (comparar con la Figura 15).

5.- Eventos

En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio a fin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es el de la programación basada en eventos. Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JavaScript permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos. Los eventos de JavaScript permiten la interacción entre las aplicaciones JavaScript y los usuarios. Cada vez que se pulsa un botón, se produce un evento. Cada vez que se pulsa una tecla, también se produce un evento. No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que por ejemplo, cada vez que se carga una página, también se produce un evento.

Recién en el año 2004, el estándar DOM publicó la especificación completa de eventos soportados por el estándar para JavaScript, más de diez años después de que los primeros navegadores incluyeran los eventos [4]. Por este motivo, muchas de las propiedades y métodos actuales relacionados con los eventos son incompatibles con los de DOM. De hecho, navegadores como Internet Explorer tratan los eventos siguiendo su propio modelo incompatible con el estándar. El conjunto total de eventos que se pueden generar en un navegador se puede dividir en cuatro grandes grupos. La especificación de DOM define los siguientes grupos:

- Eventos de ratón: se originan cuando el usuario emplea el ratón para realizar algunas acciones.
- Eventos de teclado: se originan cuando el usuario pulsa sobre cualquier tecla de su teclado.
- Eventos HTML: se originan cuando se producen cambios en la ventana del navegador o cuando se producen ciertas interacciones entre el cliente y el servidor.
- Eventos DOM: se originan cuando se produce un cambio en la estructura DOM de la página. También se denominan "eventos de mutación".

En la programación orientada a eventos, las aplicaciones esperan a que se produzcan los eventos. Una vez que se produce un evento, la aplicación responde ejecutando cierto código especialmente preparado. Este tipo de código se denomina "manejador de eventos" y las funciones externas que se definen para responder a los eventos se suelen denominar "funciones manejadoras". La forma más sencilla de incluir un manejador de evento es mediante un atributo de [X]HTML. El siguiente ejemplo muestra un mensaje cuando el usuario presiona un botón:

```
<input type="button" value="Presiona y verás"
onclick="alert('Gracias por Clickear');" />
```

De hecho, ya se ha introducido la idea tanto de evento como de manejador de un evento en el código de la Figura 12, en el contexto de funciones JavaScript.

Debe notarse que tanto la idea de eventos como la de funciones para manejarlos es en cierta forma *natural* cuando se trata de generación dinámica de contenido e incluso de

comunicación **asincrónica**. En la Figura 24 se muestra un resumen de los eventos más utilizados.

Evento	Descripción	Elementos para los que está definido
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
ondblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla y no soltarla	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón *"sale"* del elemento	Todos los elementos
onmouseover	El ratón *"entra"* en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos
onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

Figura 24. Resumen de Eventos Definidos.

6.- XMLHttpRequest

También referido como XMLHTTP (Extensible Markup Language / Hypertext Transfer Protocol), básicamente es una interfaz empleada para realizar peticiones HTTP y HTTPS a servidores Web [17]. Para transferir los datos se usa cualquier codificación basada en texto, incluyendo: texto plano, XML, JSON, HTML y, dependiendo de la implementación, codificaciones particulares específicas. La interfaz se presenta como una clase de la que una aplicación cliente puede generar tantas instancias como necesite para manejar el diálogo con el servidor. Tal como se indica en [16] XMLHttpRequest fue originalmente diseñado para Internet Explorer, luego lo adoptó Mozilla y actualmente está siendo estandarizado por la organización W3C.

Todas las aplicaciones realizadas con Ajax deben instanciar en primer lugar el objeto XMLHttpRequest, que es el objeto clave que permite realizar comunicaciones con el servidor en segundo plano, sin necesidad de recargar las páginas. Dado que aún no todos los navegadores incorporan esta funcionalidad de la misma manera, la implementación del objeto XMLHttpRequest depende de cada navegador, por lo que es necesario emplear una discriminación sencilla en función del navegador en el que se está ejecutando el código:

```
if (window.XMLHttpRequest)    // Navegadores que siguen los estándares
{
 peticion_http = new XMLHttpRequest();
}
else if(window.ActiveXObject)  // Navegadores obsoletos
{
 peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Nótese que para instanciar un objeto de esta clase es necesario acceder al BOM vía *window*... (es una propiedad del navegador), lo cual coincide con lo que se muestra en la Figura 23. Los navegadores que siguen los estándares (Firefox, Safari, Opera, Internet Explorer 7 y 8) implementan el objeto XMLHttpRequest de forma nativa, por lo que se puede obtener a través del objeto window. Los navegadores obsoletos (Internet Explorer 6 y anteriores) implementan el objeto XMLHttpRequest como un objeto de tipo ActiveX. La Figura 25 muestra las **propiedades** definidas para un objeto XMLHttpRequest.

Propiedad	Descripción
readyState	Valor numérico (entero) que almacena el estado de la petición
responseText	El contenido de la respuesta del servidor en forma de cadena de texto
responseXML	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM
status	El código de estado HTTP devuelto por el servidor (200 para una respuesta correcta, 404 para "No encontrado", 500 para un error de servidor, etc.)
statusText	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.

Figura 25. Propiedades de XMLHttpRequest.

Los valores definidos para la propiedad **readyState** (una de las propiedades de los objetos XMLHttpRequest) son los que se muestran en la Figura 26.

Valor	Descripción
0	No inicializado (objeto creado, pero no se ha invocado el método open)
1	Cargando (objeto creado, pero no se ha invocado el método send)
2	Cargado (se ha invocado el método send, pero el servidor aún no ha respondido)
3	Interactivo (se han recibido algunos datos, aunque no se puede emplear la propiedad responseText)
4	Completo (se han recibido todos los datos de la respuesta del servidor)

Figura 26. Valores Posibles para readyState.

Los **métodos** disponibles para un objeto XMLHttpRequest son los que se muestran en la Figura 27.

Método	Descripción
abort()	Detiene la petición actual
getAllResponseHeaders()	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor
getResponseHeader("cabecera")	Devuelve una cadena de texto con el contenido de la cabecera solicitada
onreadystatechange	Responsable de manejar los eventos que se producen. Se invoca cada vez que se produce un cambio en el estado de la petición HTTP. Normalmente es una referencia a una función JavaScript
open("metodo", "url")	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino (puede indicarse de forma absoluta o relativa)
send(contenido)	Realiza la petición HTTP al servidor
setRequestHeader("cabecera", "valor")	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar el método open() antes que setRequestHeader()

Figura 27. Métodos de XMLHttpRequest.

El método ***open*()** requiere dos parámetros (método HTTP y URL) y acepta de forma opcional otros tres parámetros. La definición formal del método *open*() sería:

open(string metodo, string URL [,boolean asincrono, string usuario, string password]);

Por defecto, las peticiones realizadas son asíncronas. Si se indica un valor *false* al tercer parámetro, la petición se realiza de forma sincrónica, esto es, se detiene la ejecución de la aplicación hasta que se recibe de forma completa la respuesta del servidor. No obstante,

las peticiones sincrónicas son justamente contrarias a la filosofía de Ajax. El motivo es que una petición sincrónica congela el navegador y no permite al usuario realizar ninguna acción hasta que no se haya recibido la respuesta completa del servidor. La sensación que provoca es que el navegador se ha detenido o ha fallado, por lo que no se recomienda el uso de peticiones sincrónicas salvo que sea imprescindible. Los últimos dos parámetros opcionales permiten indicar un nombre de usuario y una contraseña válidos para acceder al recurso solicitado.

Por otra parte, el método ***send*()** requiere de un parámetro que indica la información que se va a enviar al servidor junto con la petición HTTP. Si no se envían datos, se debe indicar un valor igual a *null*. En otro caso, se puede indicar como parámetro una cadena de texto, un array de bytes o un **objeto XML DOM**.

6.1 Un Primer Ejemplo

Inicialmente, se puede analizar un ejemplo sencillo con un requerimiento al servidor sin uso de parámetros para poder comprender y tener un panorama del conjunto de conceptos a utilizar. El ejemplo, basado en uno dado en [6], recupera el contenido de un archivo del servidor y reemplaza una sección del documento HTML por lo recibido desde el servidor. Con este objetivo, se podría tener el siguiente *body* de una página html:

```
<body>

  <h1>Recuperando un archivo del servidor con Ajax</h1>

  <form>
    <input type = "button" value = "Muestra Mensaje"
      onclick = "getData('http://localhost/ejerciciosAjax/data.txt', 'targetDiv')">
  </form>

  <div id="targetDiv">
    <p>La información recuperada se reemplazará aquí</p>
  </div>

 </body>
```

que, con lo definido hasta hasta aquí, es totalmente conocido:

- un *body* de un documento HMTL
- dentro del *body* se tiene un *form* y un *div*
- dentro del *form* se define un botón, para el cual se asigna al evento *onclick* una función JavaScript, ***getData(...)***
- la función JavaScript tiene como parámetros dos cadenas de caracteres: una URL y la identificación de un *div* del mismo documento HTML
- el *div* tiene un único párrafo y una identificación que se utiliza en el *form* como parámetro de la función JavaScript

Es importante notar que el único párrafo que es el contenido del *div* es el que será reemplazado sin necesidad de recargar toda la página. El nuevo contenido será el recibido desde el servidor con URL *http://localhost/ejerciciosAjax/data.txt.* Para lograr esto, se

otorga el código que sigue, en JavaScript, que obviamente debe tener definida como mínimo la función que maneja el evento ***onclik*: *getData*(...)**. Este ejemplo tiene algunos aspectos ya definidos y otros que aún no se han detallado. Tanto la creación del objeto *XMLHttpRequest* (líneas 1 a 6) como la utilización de las propiedades *readyState* y *status* (líneas 13 y 14) y los métodos *open(...)* y *send(...)* (líneas 18 y 19), no presentarían mayores inconvenientes a partir de las definiciones dadas anteriormente. Aunque podría tomarse como intuitiva, la asignación de *onreadystatechange* no necesariamente es completamente entendible a partir del código JavaScript del ejemplo, especialmente para quienes se están iniciando en la programación con JavaScript. Más específicamente, tal como se describe antes, para *onreadystatechange* debe darse una función (también denominada función *callback*) que será invocada cada vez que haya un cambio en el estado de la petición HTTP. La función definida y asignada a partir de línea 11 en el ejemplo (y hasta la línea 17) es de las denominadas funciones anónimas (no tiene un nombre asignado) y tal como se puede notar utiliza las variables definidas dentro del alcance de la función *getData(...)*. Tal como era de esperar, la función que se ejecuta en cada cambio de estado del requerimiento HTTP:

- Verifica que el requerimiento haya efectivamente finalizado de manera satisfactoria (readyState=4).
- Cambia el contenido de texto del único párrafo identificado como lo indica el contenido de la variable ***divID*** (que es "*targetDiv*") una vez que se ha recibido la respuesta, que está asignada en ***responseText***.

```
   <script type = "text/javascript">
      function getData(dataSource, divID)
      {
(1)       var XMLHttpRequestObject = false;
(2)
(3)       if (window.XMLHttpRequest) {
(4)         XMLHttpRequestObject = new XMLHttpRequest();
(5)       } else if (window.ActiveXObject) {
(6)         XMLHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
(7)       }
(8)
(9)       if(XMLHttpRequestObject) {
(10)              var obj = document.getElementById(divID);
(11)              XMLHttpRequestObject.onreadystatechange = function()
(12)              {
(13)                    if (XMLHttpRequestObject.readyState == 4 &&
(14)                    XMLHttpRequestObject.status == 200) {
(15)                 obj.innerHTML = XMLHttpRequestObject.responseText;
(16)                    }
(17)              }
(18)              XMLHttpRequestObject.open("GET", dataSource);
(19)              XMLHttpRequestObject.send(null);
(20)       }
(21)      }
   </script>
```

El ejemplo completo se da en el Anexo 1, que no es más que un documento HTML con las

partes definidas para *head* y *body* tal como se han comentado. En el Anexo 2 se da un ejemplo con la misma funcionalidad, pero donde se definieron dentro de funciones tanto la creación del objeto *XMLHttpRequest* como el manejo de los eventos del requerimiento HTTP. Estas funciones tienen como objetivo clarificar qué es cada una de las tareas, específicamente la de la función *callback*, dado que en principio puede haber varios requerimientos pendientes y además separar las funciones correspondientes por claridad y también para el caso en que haya que cambiarles el comportamiento/implementación.

Como comentario final se podría decir que, aunque sencillo, este ejemplo muestra muchas de las características de las aplicaciones Ajax:

- Codificación con JavaScript del lado del cliente.
- Acceso al/utilización del DOM del navegador.
- Acceso al/utilización del BOM.
- Codificación/programación de una conexión/requerimiento HTTP desde el cliente hacia el servidor.
- **Manejo de eventos producidos tanto por el usuario (onclick) como por la conexión con el servidor (onreadystatechange).**
- Específicamente con respecto al punto anterior: las funciones de callback y específicamente lo que se ejecute para *onreadystatechange desacopla* o aporta la ejecución asincrónica necesaria respecto de lo que el usuario genera para tener un control casi completo de toda la aplicación, tanto del lado del servidor (lo que es usual) como del lado del cliente.

Pero aunque se tiene un panorama en cierta forma *completo*, no se ha mostrado aún toda la potencialidad de Ajax, específicamente en términos de la comunicación asincrónica con el servidor y cómo se puede aprovechar para mejorar la respuesta al usuario evitando comunicaciones con el servidor y recargas de documentos HTML completos. En este punto se puede volver a la referencia de la Figura 3, quedando totalmente claro que Ajax provee las herramientas/definiciones para mejorar la respuesta al usuario, cayendo necesariamente en los diseñadores y programadores de aplicaciones que esto sea realidad.

6.2 Interacción con el Servidor

Hasta ahora, el objeto XMLHttpRequest se ha empleado para realizar peticiones HTTP sencillas. Sin embargo, las posibilidades que ofrece el objeto XMLHttpRequest son muy amplias, ya que también permite el envío de parámetros junto con la petición HTTP. El objeto XMLHttpRequest puede enviar parámetros tanto con el método GET como con el método POST de HTTP. En ambos casos, los parámetros se envían como una serie de pares clave/valor concatenados por símbolos &. El siguiente ejemplo muestra una URL con parámetros al servidor mediante el método GET:

```
http://localhost/aplicacion?parametro1=valor1&parametro2=valor2&parametro3=valor3
```

La principal diferencia entre ambos métodos es que mediante el método POST los parámetros se envían en el cuerpo de la petición HTTP. El método GET se suele utilizar cuando se accede a un recurso que depende de la información relativamente acotada proporcionada por el usuario. El método POST se puede utilizar en operaciones más generales, que crean, borran o actualizan información. Técnicamente, el método GET tiene un límite en la cantidad de datos que se pueden enviar. En navegadores antiguos, si

se intenta enviar más de 512 bytes mediante el método GET, el servidor devuelve un error con código 414 y mensaje Request-URI Too Long ("La URI de la petición es demasiado larga").

Es importante destacar que a partir de la posibilidad de programar el lado del cliente en las aplicaciones web con Ajax y específicamente con el objeto XMLHttpRequest, se tiene conceptualmente un sistema de RPC (Remote Procedure Call) por el propio protocolo de requerimiento-respuesta que es HTTP. Si a esto se le suma que la programación del cliente es automática, en cuanto a que el código JavaScript le llega en los *propios* (o por medio de) documentos HTML, se puede comprender que el ambiente de ejecución de las **aplicaciones web** es muy poderoso:

- Se tiene el control casi total (o total, directamente) de lo que se visualiza en el lado del cliente por el acceso al DOM e incluso al BOM.
- Se tiene el esquema básico de cliente/servidor al menos para la descarga inicial de documentos HTML.
- Se tiene la capacidad de enriquecer (en los términos de aplicaciones ricas o RIA: *Rich Internet Applications*) el lado del cliente con el esquema de RPC.
- La programación de los clientes se hace por la descarga/movilidad de código (JavaScript) con lo que la instalación de los clientes se podría afirmar que es *automática*.
- Se tiene la capacidad de ejecutar código asincrónico y por demanda (cuando se tienen los datos necesarios, por ejemplo) gracias a las funciones de *callback*.

Sin embargo, el manejo de los parámetros con XMLHttpRequest se diferencia a cuando se utiliza un elemento <form> de HTML, donde al pulsar sobre el botón de envío del formulario, se crea automáticamente la cadena de texto que contiene todos los parámetros que se envían al servidor. Esto es porque el objeto XMLHttpRequest no dispone de esa posibilidad y la cadena que contiene los parámetros se debe construir manualmente.

6.3 Parámetros Mediante XML

La flexibilidad del objeto XMLHttpRequest permite el envío de los parámetros por otros medios alternativos al tradicional *query string*. De esta forma, si la aplicación del servidor así lo requiere, es posible realizar una petición al servidor enviando los parámetros en formato XML [18]. A continuación se muestra un ejemplo para enviar los datos del usuario en forma de documento XML. En primer lugar, se debe tener en cuenta que la petición ahora debe identificar que contiene XML y por supuesto los parámetros deben tener el formato XML:

```
 function armaPeticion() {
peticion_http = inicializa_xhr();
if (peticion_http) {
        peticion_http.onreadystatechange = procesaRespuesta;
        peticion_http.open("POST", "http://localhost/validaDatos.php", true);
        var parametros_xml = crea_xml();
        peticion_http.setRequestHeader("Content-Type",
                "application/x-www-form-urlencoded");
```

```
        peticion_http.send(parametros_xml);
}
 }
```

La función crea_xml() es la que específicamente se encarga del formato en que deben incluirse los parámetros de la petición y es la que debe construir el documento XML que contiene los parámetros enviados al servidor:

```
 function crea_xml() {
var fecha = document.getElementById("fecha_nacimiento");
var cp = document.getElementById("codigo_postal");
var telefono = document.getElementById("telefono");
var xml = "<parametros>";
xml = xml + "<fecha_nacimiento>" + fecha.value + "<\/fecha_nacimiento>";
xml = xml + "<codigo_postal>" + cp.value + "<\/codigo_postal>";
xml = xml + "<telefono>" + telefono.value + "<\/telefono>";
xml = xml + "<\/parametros>";
return xml;
 }
```

El código de la función anterior emplea el carácter \ en el cierre de todas las etiquetas XML. El motivo es que las etiquetas de cierre XML y HTML (al contrario que las etiquetas de apertura) se interpretan en el mismo lugar en el que se encuentran, por lo que si no se incluyen esos caracteres \ el código no validaría siguiendo el estándar XHTML de forma estricta.

El método *send*() del objeto XMLHttpRequest permite el envío de una cadena de texto y de un documento XML. Sin embargo, en el ejemplo anterior se ha optado por una solución intermedia: una cadena de texto que representa un documento XML. El motivo es que no existe un método robusto y que se pueda emplear en la mayoría de navegadores para la creación de documentos XML completos. En el Anexo 3 se puede ver un ejemplo completo del procesamiento de parámetros y respuestas mediante el uso de XML.

6.4 Parámetros JSON

Aunque el formato XML está soportado por casi todos los lenguajes de programación, por muchas aplicaciones y es una tecnología madura y probada, en algunas ocasiones es más útil intercambiar información con el servidor en formato JSON (JavaScript Object Notation) [21]. JSON es un formato mucho más compacto que XML y además sin la sobrecarga de procesamiento que implica utilizar XML. Por definición, es mucho más fácil de procesar en el navegador del usuario, que normalmente ejecuta código JavaScript. Cada vez existen más utilidades para procesar y generar el formato JSON en los diferentes lenguajes de programación del servidor (PHP, Java, C#, etc.).

7.- Chat: Ajax + CGI + ...

Teniendo la posibilidad de programar tanto el lado del servidor como el del cliente y, además, el manejo de eventos asincrónicos, se podría diseñar una versión simplificada de una “sala de chat”. Esta versión simplificada de chat podría definirse como una pizarra de anuncios compartida que se actualiza en todos los clientes cuando uno de ellos incluye un mensaje o nota. Un mensaje o nota se puede definir directamente como una cadena de texto ingresada por un usuario en un form de un documento HTML.

Quizás es importante aclarar en este punto que se se planteará una implementación de chat bajo la arquitectura cliente/servidor, y esta elección puede no ser la más apropiada desde el punto de vista técnico. Al menos, se tendrían que analizar dos alternativas que parecen como mínimo interesantes y que pueden llegar a ser mejores que la planteada aquí

- Con procesos pares que se comunican utilizando mensajes broadcast o multicast.
- Con la propuesta denominada *server push*, de la cual al menos se introducirán las ideas más importantes.

Con el fin de simplificar el diseño de la interfaz, se podría definir una aplicación web directamente con una única página web como la que se muestra en la Figura 28, con

- Un título general.
- Una parte dedicada a incluir todos los mensajes de los usuarios, que debe tener la capacidad de desplazar el texto (scroll), dado que es de esperar que haya más texto del que se puede mostrar en una pantalla.
- Una línea para entrada de texto por parte del usuario, que debe pasarse al área común una vez que el usuario presiona la tecla enter.

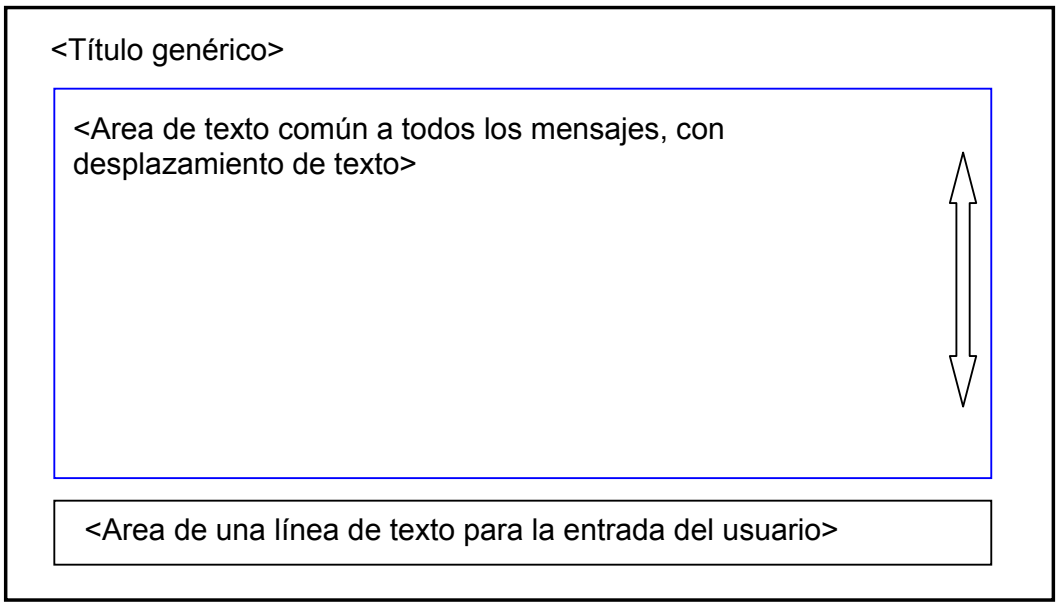

Figura 28. Diseño Simplificado para un Chat.

La parte gráfica del diseño de la Figura 28 correspondería exclusivamente a un documento HTML como el de la Figura 29, donde se tiene cada elemento bien delimitado

y, de hecho, con algunos detalles ya preparados para la aplicación web para la que se diseña, tal como el *id="targetDiv"* para el "destino" del texto común a todos los usuarios. Lo que queda por definir es, evidentemente, el funcionamiento de esta aplicación web, ya que hasta este punto solamente se tiene definido la parte más estática del chat, el propio documento HTML que se puede visualizar como tal utilizando un navegador.

```
<html>
  <head>
    <title>My chat</title>
  </head>
  <body>
    <h2>Open Chat Site</h2>

    <div id="targetDiv" style="overflow:scroll;width:800px;height:400px;border:1px solid blue;">
    </div>

    <br> <br>

    <form name="forinput">
       <input type="text" id="usrText" style="width:800px">
    </form>
  </body>
</html>
```

Figura 29. Documento HTML para un Chat Simplificado.

El funcionamiento del chat es relativamente sencillo, de acuerdo a la simplificación planteada:

- Los usuarios se conectan a un sitio web que es el que les provee la aplicación web con la interfaz gráfica/documento HTML definido anteriormente.
- Cualquier usuario puede ingresar texto en cualquier momento, asincrónicamente respecto a los demás usuarios.
- Cuando un usuario ingresa un texto, este texto pasa a ser parte del área común de texto de todos los usuarios conectados al servidor.

Con esta definición queda muy claro que la arquitectura del sistema será cliente/servidor. Esta arquitectura está dada (o como mínimo "orientada") casi directamente por la definición misma de las aplicaciones web, que al ser basadas en la navegación web, al menos comienzan su ejecución con una conexión a un servidor HTTP.

La definición del funcionamiento anterior tiende necesariamente a una aplicación web con Ajax o como mínimo con características de manejo de eventos/información asincrónicos. Lo que se produce localmente para un usuario no presenta, en principio, problemas de sincronismo, se podría pensar en el funcionamiento más elemental del form: la generación de una petición al servidor y la recepción de la respuesta del servidor. Sin embargo, toda la actividad de los clientes no locales que necesariamente debe ser recibida no puede ser expresada en términos de requerimiento/respuesta a partir de forms, dado que los usuarios están completamente desacoplados entre sí, sin ningún tipo de comunicación fuera de la estrictamente básica con el servidor. Si bien en este punto se podría utilizar alguna de las propuestas e implementación de server push, al menos para esta aplicación

y con el fin de ejemplificar Ajax, en cierta forma se simulará. La idea básica es elemental: cada cliente debería recibir **siempre** todos los cambios de estado que el servidor identifique para ese cliente en el tiempo en que le llegan datos (información ingresada por otros clientes). En cierta forma, esto genera un inconveniente en cuanto a la información que ingresa localmente cada cliente, dado que queda por definir cuándo esta información del cliente local se envía al servidor (y se refleja en su propia pantalla). En este punto, inicialmente se puede decidir simplemente una frecuencia de envío de requerimientos tal que el usuario no note que se está "postponiendo" el envío de sus datos para que la respuesta llegue en tiempo y forma hacia y desde el servidor.

En las secciones siguientes, se plantea y resuelve la problemática básica de la aplicación como si fuera sincrónica incluyendo algunos comentarios de la parte del servidor web con CGI. Luego de esta explicación, necesariamente se deben incluir las características asincrónicas necesarias para que todos los clientes no solamente puedan intercambiar información sino que se aprovechen todas las características de Ajax para las aplicaciones web.

7.1 Elementos de JavaScript a Utilizar

En esta sección se darán los detalles más importantes de JavaScript relacionados con la Sala de Chat simplificado que se ha presentado. Si bien no será completo, dado que no estará lo que es quizás más importante, que es el comportamiento asincrónico de la aplicación, se considera necesaria esta sección específicamente para explicar los elementos o las características de JavaScript a utilizar. De esta forma, en esta sección específicamente se mostrarán detalles de JavaScript que no se han visto hasta este punto relativos al manejo de eventos e información asociada al DOM y al BOM.

Se partirá de lo dado en la Figura 28, que muestra el diseño de la página web y la Figura 29, que muestra el documento HTML que produce el diseño dado. A partir de aquí se deben empezar a dar las características de funcionalidad de la aplicación, y esto necesariamente sa hará con código JavaScript. Lo primero a agregar necesariamente debe ser el manejo de los datos ingresados por el usuario en el único campo de entrada del único formulario (*form*) del documento HTML. Nótese que el documento HTML de la Figura 29 tiene un formulario pero no produce ningún resultado al ingresar texto y la tecla *enter.* No se ha especificado el atributo *ACTION* para este formulario, por lo tanto, cuando se presiona enter en el campo de texto, los navegadores muestran el documento HTML como si recién se hubiera cargado (con todas las partes de texto "vacías") independientemente de lo que se hubiera ingresado por teclado previo a la tecla *enter*.

En vez de agregarle al formulario un atributo *ACTION*, se le agregará una referencia a una función JavaScript que será la encargada de manejar los datos cuando el usuario ingrese enter en el campo de entrada de texto del formulario. Inicialmente, se puede pensar en una función que solamente ponga en una ventana "alert" (de alerta) lo que el usuario ingresó por teclado. La función JavaScript para realizar esto es sencilla, tal como se muestra en la Figura 30: recibe como parámetro lo necesario para recuperar el texto ingresado por el usuario y genera el alerta. Quedan dos detalles importantes por definir, que no están en el código JavaScript de la Figura 30:

- Cómo se asocia esta función al evento que se genera al presionar enter en el

campo del formulario.

- Dónde se incorpora la propia función dentro del documento HTML.

```
function submitHandler(txtID)
{
   var theText = document.getElementById(txtID);
   alert(theText.value);
}
```

Figura 30. Texto de un Campo en un Alert de JavaScript.

En la Figura 31 se muestra el documento HTML que asocia la función de la Figura 30 al evento producido por el ingreso de información en el formulario, y responde a los interrogantes planteados anteriormente.

```
<html>
  <head>
    <title>My chat</title>
    <script type = "text/javascript">
      function submitHandler(txtID)
      {
        var theText = document.getElementById(txtID);
        alert(theText.value);
      }
    </script>
  </head>
  <body>
    <h2>Open Chat Site</h2>

    <div id="targetDiv" style="overflow:scroll;width:800px;height:400px;border:1px
solid blue;">
    </div>

    <br> <br>

    <form name="forinput" onsubmit="submitHandler('usrText')">
      <input type="text" id="usrText" style="width:800px">
    </form>
  </body>
</html>
```

Figura 31. Documento Completo para Generar un Alert.

Nótese que la función *submitHandler*(..) se incluye en la sección de cabecera del documento HTML, siguiendo las recomendaciones de uso, pero podría estar en el cuerpo del documento y se tendría el mismo resultado. Se debe tener claro que al generarse el alerta desde JavaScript, el navegador espera la entrada correspondiente del usuario para cerrar la ventana correspondiente al alerta y vuelve a cargar el documento HTML ”como

si” se hubiera enviado un requerimiento al servidor y éste retornara el documento HTML *original.*

Otros de los detalles resolver es la incorporación de texto en el área del documento HTML dedicada a tal fin, específicamente en el *div “targetDiv”*. Ya que se tiene el documento HTML de la Figura 31, se puede modificar la función *submitHandler*() para que, justamente, incorpore en el *div “targetDiv”* lo que el usuario ha ingresado en el campo del formulario. La Figura 32 muestra el nuevo documento HTML, con esta funcionalidad incorporada.

```
<html>
  <head>
    <title>My chat</title>
    <script type = "text/javascript">
      function submitHandler(txtID, divID)
      {
        var theText = document.getElementById(txtID);
        var theArea = document.getElementById(divID);
        theArea.innerHTML += "<p>" + theText.value + "</p>";
        theText.value = "";
        theArea.scrollTop = theArea.scrollHeight;
        return false;
      }
    </script>
  </head>
  <body>
    <h2>Open Chat Site</h2>

    <div id="targetDiv" style="overflow:scroll;width:800px;height:400px;border:1px
solid blue;">
    </div>

    <br> <br>

    <form name="forinput" onsubmit="return submitHandler('usrText', 'targetDiv')">
      <input type="text" id="usrText" style="width:800px">
    </form>
  </body>
</html>
```

Figura 32. Manejo de Entrada de Información.

En la Figura 32 aparecen varios detalles relativamente nuevos de JavaScript dedicados específicamente al control de eventos y de modificación del documento que muestra el navegador. La forma de asociar la función al evento de entrada de información es, ahora,

```
<form name="forinput" onsubmit="return submitHandler('usrText', 'targetDiv')">
```

para que se haga una petición al servidor HTTP solamente en el caso en que la función

JavaScript retorne el valor verdadero. En este caso, como al final de *submitHandler()* se ejecuta *return false*, el documento HMTL nunca se recargará como en todos los ejemplos anteriores. Por otro lado,

```
theArea.innerHTML += "<p>" + theText.value + "</p>";
```

modifica el contenido del *div targetDiv* directamente agregando información al final,

```
theText.value = "";
```

deja el campo de entrada de texto sin el valor ingresado, para que aparezca en blanco como debe ser, y

```
theArea.scrollTop = theArea.scrollHeight;
```

desplaza el texto del div targetDiv de forma tal que el contenido final es lo último que se muestra.

Hasta este punto, al cargar el documento HTML en el navegador se debe poner el foco de la entrada en el campo de entrada de texto con el puntero. Para ello, se puede agregar una función para que ni bien se termine de cargar el documento, de manera automática se ponga el navegador directamente a esperar la entrada sobre el campo de texto del formulario. La Figura 33 muestra el agregado de esta función asociada al evento en el documento HTML de la Figura 32.

```
<html>
  <head>
    <title>My chat</title>
    <script type = "text/javascript">
      function submitHandler(txtID, divID)
      {
        ...
      }
      //-------------------------------------------------------------
      window.onload=function()
      {
        document.forms[0][0].focus();
      }
    </script>
  </head>
  <body>
    ...
  </body>
</html>
```

Figura 33. “Enfocando” el Formulario.

Es muy interesante la forma de acceder al único campo (campo 0) del único formulario (formulario 0): como si se tuviera un arreglo de formularios del documento y cada formulario a su vez fuera un arreglo de campos. Este ejemplo también muestra dos funciones independientes dentro de la sección script del *head* del documento HTML: una para manejar el evento de entrada de información en el formulario y la otra para ser ejecutada ni bien el documento se carga en el navegador. También es importante notar que el evento *onload*() corresponde al BOM (se referencia al objeto *window*) y no al DOM. En el Anexo 4 se puede encontrar el documento HTML completo de este ejemplo.

Hasta este punto por supuesto no se tiene un chat ni una aplicación web, aunque se pueden comenzar a asociar algunas de las características vistas con la aplicación final. Como mínimo, cada vez que el usuario ingresa información, se debe enviar al servidor HTTP un requerimiento notificando la entrada de información en vez de mostrarla localmente como se ha hecho en el ejemplo dado en esta sección. Esto significaría que la función *submitHandler*() debería generar un requerimiento al servidor web en vez de mostrar localmente lo ingresado por el usuario. Evidentemente se debe utilizar un objeto XMLHttpRequest y construir el *query string* tal como se explicó en secciones anteriores, por ejemplo, con este fragmento de código en la función *submitHandler*()

```
...
reqObj = getReqObj();
var url="cgi-bin/textchat?newtxt=" + escape("<p>" + theText.value + "</p>");
reqObj.open("GET", url...);
reqObj.send(null);
...
```

donde

- *getReqObj*() es una función que retorna un objeto XMLHttpRequest como la dada en el Anexo 2
- *textchat* es el nombre del programa CGI que será el servidor de chat
- *escape*() es la función JavaScript que codifica los strings que son valores de un *query string* (codifica espacios y demás caracteres “especiales”)
- *theText* es la variable local de la función
- *newtxt* es el nombre del “parámetro” con el que el programa CGI espera recibir el texto ingresado por el usuario

Si el envío es sincrónico, lo que se tendría hasta este punto es el reemplazo del procesamiento estándar de un form por el manejo del envío y de la respuesta con JavaScript. Antes de continuar con el procesamiento del lado del cliente, es conveniente al menos comentar lo más importante del procesamiento en el servidor.

7.2 Primer Servidor CGI para el Chat

La tarea primordial del servidor de Chat es la de mantener y proveer la visión unificada de todo el texto que los usuarios han ingresado. Esto significa que todos los clientes envían información y deben recibir su propio texto más el texto de los demás clientes en el orden cronológico en el que se ha hecho. Dado que es muy difícil asegurar que el texto de los

diferentes clientes está en el orden temporal en el que se envió, se asegurará que está en el orden en que cada uno llegó al servidor de chat.

Una primera versión del servidor de chat con CGI puede ser una aplicación web estrictamente sincrónica, que por supuesto no estaría relacionada con Ajax pero que se hará evolucionar *hacia* Ajax. Esta primera versión no hará más que recibir las peticiones de los clientes que serán de texto plano, tal como las maneja el código JavaScript que se ha mostrado en el ejemplo anterior y poner este texto plano en un archivo donde solamente se agregará información. Específicamente, este archivo que será manejado por el servidor contendrá el texto "unificado" de todos los clientes que debe ser mostrado en la sección *div targetDiv* del documento HTML. Inicialmente, el servidor podría ser tan sencillo como el que se muestra en la Figura 34, teniendo definido que los clientes enviarán la información con un método GET y con un único parámetro conteniendo todo el texto ingresado por el usuario.

```
#include <stdio.h>     /* printf(), fgets() */
#include <stdlib.h>    /* getenv() */
#include <string.h>    /* strchr() */

// Nombre del archivo con todo el texto del chat
#define chatfname "data/chat.txt"

// Cantidad maxima de caracteres de una linea
#define MAXLINE 500

main(int argc, char *argv[])
{
   char *reqmet, *querystr, linein[MAXLINE];
   FILE *chatfile;

   // Definición del tipo de respuesta...
   printf("Content-type: text/plain%c%c", 10, 10);

   // Variable de ambiente con el Query_String para un GET
   querystr = getenv("QUERY_STRING");

   //  Abrir el archivo
   chatfile = fopen(chatfname, "a+");

   while (linein == fgets(linein, MAXLINE, chatfile))
      printf("%s", linein);
   fprintf(chatfile, "%s\n", strchr(querystr, '=')+1);
   fclose(chatfile);

   printf("%s\n", strchr(querystr, '=')+1);
}
```

Figura 34. Servidor de Chat Básico.

Nótese que tanto para este programa CGI como para el servidor web lo que se responde

a cada requerimiento es de tipo texto, aunque en realidad contiene etiquetas de HTML. Hasta este momento, del lado del servidor se tendrían que responder dos tipos de requerimientos: 1) El “inicial”, que debe responderse con el documento HTML incluyendo el código JavaScript para manejar los eventos y la posterior comunicación con el servidor, y 2) El que se genera desde cada cliente con el ingreso de información por parte del usuario. El requerimiento inicial tiene implícito un detalle importante a decidir: si incluye el texto ya ingresado por otros usuarios al momento del primer requerimiento de un cliente “nuevo”. En caso de que se decida no incluirlo, se debe recordar que este texto de todas maneras estará incluido en todas las respuestas del programa CGI de la Figura 34. Esto significa que si un usuario quiere saber qué se ha ingresado hasta el momento debe generar un requerimiento ingresando algo en el formulario para luego recibir todo lo que sería el contenido del archivo “chat.txt” que para él es transparente, por supuesto. Ya en este ejemplo tan sencillo se llega a esta clase de decisiones, que en realidad está relacionada también con:

- Hay requerimientos que pueden involucrar varias fuentes de información: en este caso, el primer requerimiento podría responderse con texto HTML que es estático (el que incluye el código JavaScript, por ejemplo, que es siempre el mismo) y el contenido del archivo donde se tiene toda la información que los usuarios ha incluido hasta el momento (el archivo chat.txt que aparece en el código de la Figura 34).
- Quizás sea necesario que un programa responda a todos los requerimientos de los usuarios, tal como sería el caso en que se deba responder al primer requerimiento con la combinación de HTML + chat.txt. En este caso se está planteando la primera respuesta con HTML estático y las siguientes con contenido dinámico, generado por el programa de la Figura 34.
- Una sesión por usuario, o mantener un estado de sesión por usuario sería útil para saber cuál es el estado de los clientes y entonces responder a los requerimientos con el mínimo contenido necesario. En este caso, cada requerimiento se responde con todo el contenido de chat.txt en vez de responder con lo que se ha recibido en el servidor desde el último requerimiento que se le respondió al usuario.

7.3 Primera Versión del Chat

En este punto se tendría todo definido (tanto en JavaScript como en CGI) y las decisiones de diseño tomadas para tener la primera versión “rudimentaria” de la aplicación web chat:

- Los usuarios se deben conectar a una URL con un navegador para recibir el primer documento HTML con todas sus partes en blanco, tal como el de la Figura 28.
- Al primer requerimiento enviado por cada usuario se le responde con **todo el texto** almacenado en el servidor de todos los usuarios/requerimientos anteriores. Este texto es incremental desde la perspectiva de la propia aplicación. Específicamente, el archivo que contiene toda la información enviada por los usuarios solamente crece, nunca se inicializa a cero o se recortan partes a partir de la puesta en marcha del sistema.
- Todo el manejo del envío y recepción de información ingresada por los usuarios se hará en JavaScript y de manera sincrónica.

La gran mayoría de estas definiciones son solamente justificadas en la simplificación de esta primera versión de la aplicación web, muchas de ellas no son razonables desde la perspectiva de la funcionalidad del sistema frente a los usuarios.

El documento HTML con el que se responde al primer requerimiento de los clientes sería básicamente el de la Figura 32, con el agregado que se muestra en la Figura 33 y el agregado del manejo de los requerimientos al servidor con un objeto XMLHttpRequest. El documento HTML completo se muestra en la Figura 35.

```
<html>
  <head>
    <title>My chat</title>
    <script type = "text/javascript">
        //---------------------------------------------------------------------
        function getReqObj() {
          var XMLHttpRequestObject;
          if (window.XMLHttpRequest) {
            XMLHttpRequestObject = new XMLHttpRequest();
          } else if (window.ActiveXObject) {
            XMLHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
          }
          return XMLHttpRequestObject;
        }
       //---------------------------------------------------------------------
       function submitHandler(txtID, divID){
          var theText = document.getElementById(txtID);
          var theArea = document.getElementById(divID);
          var reqObj = getReqObj();
          var url="cgi-bin/textchat?newtxt=" + escape("<p>" + theText.value + "</p>");
          reqObj.open("GET", url, false);
          reqObj.send(null);
          theArea.innerHTML = unescape(reqObj.responseText);
          theArea.scrollTop = theArea.scrollHeight;
          theText.value = "";
          return false;
       }
       //---------------------------------------------------------------------
       window.onload=function(){
          document.forms[0][0].focus();
       }
    </script>
  </head>
  <body>
     <h2>Open Chat Site</h2>
     <div id="targetDiv" style="overflow:scroll;width:800px;height:400px;border:1px solid
blue;">
     </div>
     <br> <br>
     <form name="forinput" onsubmit="return submitHandler('usrText', 'targetDiv')">
        <input type="text" id="usrText" style="width:800px">
     </form>
  </body>
</html>
```

Figura 35. Manejo de Entrada de Información.

El documento HTML de la Figura 35 tiene todo lo necesario para comenzar el intercambio de información con el servidor de chat a partir de un sector en blanco en cuanto a contenido de usuarios. Se debe recordar que este sector en blanco aparecerá siempre que se acceda al servidor por primera vez o, simplemente, se haga una recarga de la página web. Esta es una de las características que necesariamente debería ser eliminada en una aplicación más elegante (o más *correcta*) para el usuario. Más específicamente, la función *submitHandler*() está completa y describe el intercambio sincrónico de información o de requerimiento-respuesta con la instrucción

```
reqObj.open("GET", url, false);
```

lo cual significa que un usuario no recibe la información de los demás a menos que envíe algo al servidor, lo cual es, por lo menos, difícil de justificar frente a los usuarios. Además, así como es necesario el uso de la función *escape*(...) para la codificación del texto y el envío de información hacia el servidor, es necesaria también la función *unescape*(...) para su decodificación cuando la misma información es recibida desde el servidor. También en el código de la función submitHandler queda explícito que se se hará un requerimiento HTTP GET al programa CGI *cgi-bin/textchat*.

Por su parte, el servidor de chat sería exactamente el de la Figura 34, aunque habría algunas consideraciones para hacer específicamente en lo relativo a las políticas de seguridad del sistema en el que se ejecute. Una de las características de este servidor de chat (como de una gran mayoría de los servidores) es la necesidad de escribir (específicamente agregar al final) un archivo, con el nombre *data/chat.txt*.

En el caso específico de la distribución Fedora de Linux, se ha incorporado SELinux como un mecanismo extra de seguridad “intra-sistema” y por ello es posible que esto genere problemas de seguridad o más específicamente, no se pueda abrir un archivo para escritura en estas condiciones. Una de las soluciones más rápidas (y como suele suceder también menos segura) es la de directamente manejar este archivo en el directorio /tmp, (cambiando la definición correspondiente de *chatfname*) sobre el que generalmente no se tienen restricciones. Si se quiere mantener el nombre del directorio y archivo tal como están dados en la Figura 34 deberían ejecutarse los siguientes comandos, con el usuario *root* del sistema:

```
# mkdir /var/www/cgi-bin/data
# chcon -t httpd_sys_content_t /var/www/cgi-bin/data
# chown apache.apache /var/www/cgi-bin/data
# chmod 700 /var/www/cgi-bin/data
```

dado que el directorio de trabajo (working directory) del programa cgi es el directorio donde está ubicado el propio ejecutable: */var/www/cgi-bin* y el usuario que efectivamente lo ejecuta es el usuario con el que se ejecuta el propio servidor httpd: *apache*. En [22] [23] [24] [25] [26] se dan algunos detalles del funcionamiento de SELinux y de su relación con el servidor Apache.

En el caso específico de la distribución Kubuntu, Apache2 corre como el usuario **www-data**, debido a ello, deberá otorgarse permiso de escritura del archivo chat.txt a tal usuario. Por otro lado, en las últimas versiones de apache, el directorio por defecto donde deben residir las aplicaciones CGI es: **/usr/lib/cgi-bin**, por ello el ejecutable CGI debe residir en tal directorio y a su vez data/chat.txt debe residir a partir de aquel (si queremos respetar el camino definido en la el archivo HTML presentado anteriormente.

Aunque en este punto podría pensarse que el sistema está completo, aún falta un detalle importante a resolver que está directamente relacionado con una característica específica del servidor apache: **la atención de requerimientos concurrentemente**. Aunque se podría decidir cancelar la atención concurrente (incluso cambiando de servidor de HTTP si fuera necesario) es interesante resolver el problema específicamente en lo referente a la operación concurrente sobre el archivo que se cambia en cada requerimiento generado por el ingreso de información por parte de un usuario. Está claro que no se puede dejar el acceso libre de escritura sobre el archivo chat.txt dado que pueden suceder numerosas condiciones de carrera (*race conditions*) que dejarían el archivo inconsistente a la visión de los usuarios de este archivo. La solución más básica es sencilla: utilizar bloqueos (*locks*). El servidor en realidad no cambiaría mucho: tendría que agregarse una llamada al sistema *flock*(...) antes de comenzar a operar sobre el archivo para que quede bloqueado para los demás procesos y otra llamada al sistema *flock*(...) al terminar de operar sobre el archivo para que los demás procesos puedan también operar sobre el mismo. En el Anexo 5 se da la información completa de esta primera versión de chat: el servidor y el documento HTML.

7.4 Características a Incorporar en la Primera Versión del Chat

Como se aclaró varias veces en la sección anterior, la primera versión de chat presentada es muy limitada tanto en lo referente a la sucesión de información que se le muestra al usuario como en lo referente a rendimiento. No se han aprovechado las capacidades de comunicación asincrónica con el usuario para actualización de información ni tampoco se ha aprovechado el hecho de incorporar la información **de forma incremental**, enviando a cada usuario la parte del archivo chat.txt que no ha recibido aún.

7.4.1 Del Lado del Cliente: Asincronismo y Estado

Para dejar abierta la posibilidad de que el servidor envíe información que recibió de otro cliente, evidentemente se debe enviar un requerimiento que no necesariamente será respondido sincrónicamente, es decir que el envío se debe hacer con

```
reqObj.open("GET", url, true);
```

o, directamente, con

```
reqObj.open("GET", url);
```

para que (en ambos casos) el servidor envíe información cuando haya efectivamente recibido algo y, además, se debe establecer una función de manejo de la respuesta asincrónica del servidor tal como se explicó anteriormente. Por otro lado, esta petición pendiente debe ser tenida en cuenta a la hora de manejar el ingreso de información por parte del usuario local, dado que se debe generar otra petición con el nuevo texto. En este punto se puede utilizar la operación *abort*() del objeto XMLHttpRequest.

Otra alternativa para la implementación del "asincronismo" y la "simulación" del esquema *server push* sería simplemente mantener el requerimiento como sincrónico y utilizar la función JavaScript setInterval(), para que el cliente pida al servidor de manera periódica las posibles actualizaciones que deba presentar al usuario. Si el período de consulta es **suficientemente pequeño** el usuario tendrá disponibles las actualizaciones como si fueran llegando exactamente a medida que se producen. Dos consideraciones son importantes en este punto:

- Mientras más frecuentes sean las peticiones al servidor más "real" o cercana a los propios eventos de los demás clientes se mostrarían de manera local las actualizaciones. Dado que los usuarios interactivos toman varios segundos para producir texto (entrada por teclado), una frecuencia de un requerimiento por segundo, por ejemplo, sería suficiente para "capturar" las actualizaciones de todos los clientes, .
- A mayor frecuencia de actualizaciones es mayor la sobrecarga, y además esta sobrecarga también es proporcional a la cantidad de usuarios. Nótese que tanto en el servidor como en las conexiones hacia el servidor siempre habrá tarea para realizar aunque ninguno de los clientes produzca ninguna actualización del área de texto común. Esta característica por supuesto es completamente contrapuesta a la idea de Ajax y de las aplicaciones web y distribuidas en general.

Otra de las características a incorporar del lado del cliente es la actualización de información del área común en vez de la recarga completa de la información. Es decir que, en vez del código de la Figura 35

```
theArea.innerHTML = unescape(reqObj.responseText);
```

debería haber algo similar a

```
theArea.innerHTML += unescape(reqObj.responseText);
```

Es decir que la aplicación debería mantener algo similar a **una sesión y estado del cliente** que hace la petición para que el servidor envíe solamente el contenido del área común que no tiene, que es lo que se ha recibido de otros clientes desde la última respuesta.

7.4.2 Del Lado del Servidor: Estado + Notificación de Cambios

El servidor por supuesto depende de la elección que se haga para el cliente y en particular para la forma en que el cliente recibe las actualizaciones. Si el cliente envía periódicamente requerimientos, el servidor no es muy diferente: siempre recibe el requerimiento y responde de acuerdo a lo que el cliente haya recibido por última vez (el *estado* del cliente) y lo que hayan producido los demás clientes. En algunos de los requerimientos necesariamente se llevarán datos introducidos por el usuario y en otros (quizás la mayoría, dado que son requerimientos periódicos) no habrá datos producidos por el usuario. Esta alternativa es en realidad una simulación de "server push" con el costo de sobrecarga, pero el servidor es realmente sencillo, no más que el presentado para la implementación más sencilla, dado que ya se controla concurrencia. A lo sumo habría que agregar lo que corresponde a estado o verificación de qué información del área común de la sala de chat debería recibir el cliente que hace el requerimiento.

En el caso que se elija que el cliente deje abierto o pendiente un requerimiento hasta que en el servidor se identifique nueva información (una versión más propia de server push y definitivamente asociada a Ajax), evidentemente el servidor es más complejo, Una de las alternativas es dejar pendiente, justamente la respuesta HTTP al requerimiento y que se responda cuando algo cambia en el servidor. Esto implica resolver varios detalles:

- El servidor debe quedar a la espera en caso de no haber nada diferente a lo que el cliente tenga como estado. Esto significa alguna forma de suspensión de proceso o de thread, dependiendo del servidor.
- Dado que pueden haber períodos de inactividad relativamente largos, puede suceder que las conexiones TCP sobre las que se transportan los requerimientos y las respuestas HTTP se cancelen por *timeout* y estos eventos también deben ser identificados tanto del lado del cliente como del lado del servidor.
- Pueden haber múltiples threads o procesos en espera del lado del cliente y esto puede generar problemas para algunos servidores. Es una de las características que suele discutirse ampliamente en las aplicaciones denominadas *comet*.

En lo referente a mantener el estado, evidentemente se puede hacer de varias maneras, como en el contexto de una sesión de una aplicación web. Quizás la forma más tradicional sean las cookies, que son generadas en el servidor y manejadas directamente por los navegadores del lado del cliente. Otra alternativa posible por la disponibilidad de JavaScript del lado del cliente es que el propio código Javascript **mantenga o incluso calcule el estado actual** y se lo envié al servidor en cada requerimiento:

- En el caso de que mantenga el estado, es como un reemplazo de las cookies, pero a nivel del código JavaScript del lado del cliente.
- En el caso de que se calcule el estado, en el caso del chat, lo importante es qué información se tiene en el área de texto común a todos los clientes. Una de las formas más rudimentarias de calcular el estado es la de contar la cantidad de palabras o bytes o párrafos que tiene el cliente y con esta cantidad se debería operar en el servidor para verificas si hay información que deba ser enviada al cliente como respuesta al requerimiento.

Referencias

[1] L. Cárdenas, J. Gracia, El objeto document,
http://www.webestilo.com/javascript/js22.phtml

[2] Ecma International, http://www.ecma-international.org/

[3] Ecma International, TC39 – ECMAScript,
http://www.ecma-international.org/memento/TC39-M.htm

[4] J. Eguíluz Pérez, Introducción a AJAX, Capítulo 6. Eventos,
http://librosweb.es/ajax/capitulo6.html,

[5] J. J. Garret, “Ajax: A New Approach to Web Applications”,
http://www.adaptivepath.com/publications/essays/archives/000385.php

[6] S. Holzner, Ajax For Dummies, Wiley Publishing, Inc., Indianapolis, Indiana, ISBN: 978-0-471-78597-2, 2006.

[7] JavaScript Kit, DOM Element methods,
http://www.javascriptkit.com/domref/elementmethods.shtml

[8] JavaScript Kit – Document Object, http://www.javascriptkit.com/jsref/document.shtml

[9] JavaScript tutorial - Adding JavaScript to a page,
http://www.howtocreate.co.uk/tutorials/javascript/incorporate

[10] JavaScript Tutorials, The Document Object (Part One), That Document Thing,
http://homepage.ntlworld.com/kayseycarvey/document.html

[11] JavaScript tutorial - DOM nodes and tree,
http://www.howtocreate.co.uk/tutorials/javascript/dombasics

[12] JavaScript tutorial - Writing with script,
http://www.howtocreate.co.uk/tutorials/javascript/writing

[13] A. Le Hors, P. Le Hégaret, G. Nicol, L. Wood, M. Champion, S. Byrne, Document Object Model Core, April 2004, http://www.w3.org/TR/DOM-Level-3-Core/core.html

[14] A. Le Hors, G. Nicol, L. Wood, M. Champion, S. Byrne, Document Object Model Core, November 2000, http://www.w3.org/TR/DOM-Level-2-Core/core.html

[15] Mozilla Developer Center / DOM / document,
https://developer.mozilla.org/en/DOM/document

[16] Mozilla Developer Center / XMLHttpRequest,
https://developer.mozilla.org/en/XmlHttpRequest

[17] A. van Kesteren, XMLHttpRequest, W3C Working Draft 19 November 2009,
http://www.w3.org/TR/XMLHttpRequest/

[18] W3C, Extensible Markup Language (XML), http://www.w3.org/XML/

[19] W3Schools Online Web Tutorials, JavaScript Date Object,
http://www.w3schools.com/jsref/jsref_obj_date.asp

[20] W3Schools Online Web Tutorials, JavaScript Where To,
http://www.w3schools.com/js/js_whereto.asp

[21] JSON, http://www.json.org/json-es.html

[22] Fedora Project, Virtual Hosting, CGI scripts and suEXEC,
http://docs.fedoraproject.org/selinux-apache-fc3/sn-suexec-and-cgi-scripts.html

[23] Selinux on FC5
http://aplawrence.com/Linux/basic_selinux.html

[24] Apache and SELinux
http://www.beginlinux.com/server_training/web-server/976-apache-and-selinux

[25] Murray McAllister, Daniel Walsh, Dominick Grift, Eric Paris, James Morris, Fedora 10 Security-Enhanced Linux, User Guide, Edition 1.1, Copyright © 2008 Red Hat, Inc.
http://docs.fedoraproject.org/selinux-user-guide/f10/en-US/

[26] Fedora Project, Disabling SELinux
http://docs.fedoraproject.org/selinux-user-guide/f10/en-US/sect-Security-Enhanced_Linux-Enabling_and_Disabling_SELinux-Disabling_SELinux.html

Anexo 1 – Ejemplo Ajax-1.html

```
<head>
  <script type = "text/javascript">
    function getData(dataSource, divID)
    {
      var XMLHttpRequestObject = false;

      if (window.XMLHttpRequest) {
        XMLHttpRequestObject = new XMLHttpRequest();
      } else if (window.ActiveXObject) {
        XMLHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
      }

      if(XMLHttpRequestObject) {
        var obj = document.getElementById(divID);
        XMLHttpRequestObject.onreadystatechange = function()
        {
          if (XMLHttpRequestObject.readyState == 4 &&
            XMLHttpRequestObject.status == 200) {
              obj.innerHTML = XMLHttpRequestObject.responseText;
          }
        }
        XMLHttpRequestObject.open("GET", dataSource);
        XMLHttpRequestObject.send(null);
      }
    }
  </script>
</head>
<body>
 <h1>Recuperando un archivo del servidor con Ajax</h1>

  <form>
    <input type = "button" value = "Muestra Mensaje"
      onclick = "getData('http://localhost/ejerciciosAjax/data.txt', 'targetDiv')">
  </form>

  <div id="targetDiv">
    <p>La información recuperada se reemplazará aquí</p>
  </div>
 </body>
</html>
```

Anexo 2 – Ejemplo Ajax-2.html

```
<html>
  <head>
    <script type = "text/javascript">
      //---------------------------------------------------------------------
      //-- Returns an XMLHttpRequest object
      function getReqObj() {
        var XMLHttpRequestObject;

        if (window.XMLHttpRequest) {
          XMLHttpRequestObject = new XMLHttpRequest();
        } else if (window.ActiveXObject) {
          XMLHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
        }

        return XMLHttpRequestObject;
      }

      //---------------------------------------------------------------------
      //-- Handles the request statechange event
      function onReqChange(httpReq, divID) {
        if (httpReq.readyState == 4 && httpReq.status == 200) {
          var obj = document.getElementById(divID);
          obj.innerHTML = httpReq.responseText;
        }
      }

      //---------------------------------------------------------------------
      //-- Handles the button-form onclick event
      function getData(dataSource, divID)
      {
        var XMLHttpRequestObject = false;

        XMLHttpRequestObject = getReqObj();

        if (XMLHttpRequestObject) {
          XMLHttpRequestObject.onreadystatechange = function(){
            onReqChange(XMLHttpRequestObject, divID)
          }
        }
        XMLHttpRequestObject.open("GET", dataSource);
        XMLHttpRequestObject.send(null);
      }
    </script>
  </head>
  <body>
   <h1>Recuperando un archivo del servidor con Ajax</h1>

    <form>
      <input type = "button" value = "Muestra Mensaje"
        onclick = "getData('http://localhost/ejerciciosAjax/data.txt', 'targetDiv')">
    </form>

    <div id="targetDiv">
      <p>La información recuperada se reemplazará aquí</p>
    </div>
  </body>
```

```
</html>
```

Anexo 3 - Ejemplo Procesando XML como Respuesta

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejercicio 16 - Listas desplegables encadenadas</title>

<script type="text/javascript">
var peticion = null;

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function muestraProvincias() {
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
      var lista = document.getElementById("provincia");
      var documento_xml = peticion.responseXML;

      var provincias = documento_xml.getElementsByTagName("provincias")[0];
      var lasProvincias = provincias.getElementsByTagName("provincia");
      lista.options[0] = new Option("- selecciona -");

      // Método 1: Crear elementos Option() y añadirlos a la lista
      for(i=0; i<lasProvincias.length; i++) {
        var codigo = lasProvincias[i].getElementsByTagName("codigo")[0].firstChild.nodeValue;
        var nombre = lasProvincias[i].getElementsByTagName("nombre")[0].firstChild.nodeValue;
        lista.options[i+1] = new Option(nombre, codigo);
      }

      // Método 2: crear código HTML de <option value="">...</option> utilizar el innerHTML de la lista

      /*
      var codigo_html = "";
      codigo_html += "<option>- selecciona -<\/option>";
      for(var i=0; i<lasProvincias.length; i++) {
        var codigo = lasProvincias[i].getElementsByTagName("codigo")[0].firstChild.nodeValue;
        var nombre = lasProvincias[i].getElementsByTagName("nombre")[0].firstChild.nodeValue;
        codigo_html += "<option value=\""+codigo+"\">"+nombre+"<\/option>";
      }

      // La separacion siguiente se tiene que hacer por este bug de microsoft:
      // http://support.microsoft.com/default.aspx?scid=kb;en-us;276228
      var esIE = navigator.userAgent.toLowerCase().indexOf('msie')!=-1;
      if(esIE) {
        document.getElementById("provincia").outerHTML =
            "<select id=\"provincia\">"+codigo_html+"</select>";
      }
      else {
```

```
        document.getElementById("provincia").innerHTML = codigo_html;
      }
      */
    }
  }
}

function cargaMunicipios() {
  var lista = document.getElementById("provincia");
  var provincia = lista.options[lista.selectedIndex].value;
  if(!isNaN(provincia)) {
    peticion = inicializa_xhr();
    if (peticion) {
      peticion.onreadystatechange = muestraMunicipios;
      peticion.open("POST",
            "http://localhost/ejerciciosAjax/cargaMunicipiosXML.php?nocache=" + Math.random(), true);
      peticion.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
      peticion.send("provincia=" + provincia);
    }
  }
}

function muestraMunicipios() {
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
      var lista = document.getElementById("municipio");
      var documento_xml = peticion.responseXML;

      var municipios = documento_xml.getElementsByTagName("municipios")[0];
      var losMunicipios = municipios.getElementsByTagName("municipio");

      // Borrar elementos anteriores
      lista.options.length = 0;

      // Se utiliza el método de crear elementos Option() y añadirlos a la lista
      for(i=0; i<losMunicipios.length; i++) {
        var codigo = losMunicipios[i].getElementsByTagName("codigo")[0].firstChild.nodeValue;
        var nombre = losMunicipios[i].getElementsByTagName("nombre")[0].firstChild.nodeValue;
        lista.options[i] = new Option(nombre, codigo);
      }
    }
  }
}

window.onload = function() {
  peticion = inicializa_xhr();
  if(peticion) {
    peticion.onreadystatechange = muestraProvincias;
    peticion.open("GET",
          "http://localhost/ejerciciosAjax/cargaProvinciasXML.php?nocache="+Math.random(), true);
    peticion.send(null);
  }

  document.getElementById("provincia").onchange = cargaMunicipios;
}

</script>
</head>
```

```
<body>
<h1>Listas desplegables encadenadas</h1>

<form>
  <label for="provincia">Provincia</label>
  <select id="provincia">
    <option>Cargando...</option>
  </select>
  <br/><br/>
  <label for="municipio">Municipio</label>
  <select id="municipio">
    <option>- selecciona una provincia -</option>
  </select>
</form>

</body>
</html>
```

Anexo 4 – JavaScript para el Chat

```
<html>
  <head>
    <title>My chat</title>
    <script type = "text/javascript">
      function submitHandler(txtID, divID)
      {
        var theText = document.getElementById(txtID);
        var theArea = document.getElementById(divID);
        theArea.innerHTML += "<p>" + theText.value + "</p>";
        theText.value = "";
        theArea.scrollTop = theArea.scrollHeight;
        return false;
      }
      //---------------------------------------------------------------
      window.onload=function()
      {
        document.forms[0][0].focus();
      }
    </script>
  </head>
  <body>
    <h2>Open Chat Site</h2>

    <div id="targetDiv" style="overflow:scroll;width:800px;height:400px;border:1px solid blue;">
    </div>

    <br> <br>

    <form name="forinput" onsubmit="return submitHandler('usrText','targetDiv')">
       <input type="text" id="usrText" style="width:800px">
    </form>
  </body>
</html>
```

Anexo 5 – Primera Versión del Chat

Servidor CGI

```
#include <stdio.h>     /* printf(), fgets(), fileno() */
#include <stdlib.h>    /* getenv() */
#include <string.h>    /* strchr() */
#include <sys/file.h>  /* flock() & friends */

// Nombre del archivo con todo el texto del chat
#define chatfname "data/chat.txt"

// Cantidad maxima de caracteres de una linea
#define MAXLINE 500

main(int argc, char *argv[])
{
  char *reqmet, *querystr, linein[MAXLINE];
  FILE *chatfile;
  int fd_chatfile;

  // Definicion del tipo de respuesta...
  printf("Content-type: text/plain%c%c", 10, 10);

  // Variable de ambiente con el Query_String para un GET
  querystr = getenv("QUERY_STRING");

  //  Abrir el archivo
  chatfile = fopen(chatfname, "a+");

  if (chatfile == NULL)
    printf("Error opening the file<br>\n");

  // Obtener el fd del archivo y bloquearlo
  fd_chatfile = fileno(chatfile);
  if (flock(fd_chatfile, LOCK_EX) != 0)
    printf("<p> Error on flock()</p>\n");

  while (linein == fgets(linein, MAXLINE, chatfile))
    printf("%s", linein);

  fprintf(chatfile, "%s\n", strchr(querystr, '=')+1);

  // Unlock and close file
  flock(fd_chatfile, LOCK_UN);
  fclose(chatfile);

  printf("%s\n", strchr(querystr, '=')+1);
}
```

Configuración en el Servidor con SELinux

```
# mkdir /var/www/cgi-bin/data
# chcon -t httpd_sys_content_t /var/www/cgi-bin/data
# chown apache.apache /var/www/cgi-bin/data
# chmod 700 /var/www/cgi-bin/data
```

Configuración en el Servidor con Ubuntu

```
# mkdir /usr/lib/cgi-bin/data
```

```
# chown www-data.www-data /usr/lib/cgi-bin/data/chat.txt
```

Documento HTML

```
<html>
  <head>
    <title>My chat</title>
    <script type = "text/javascript">
        //---------------------------------------------------------------
        function getReqObj() {
           var XMLHttpRequestObject;
           if (window.XMLHttpRequest) {
              XMLHttpRequestObject = new XMLHttpRequest();
           } else if (window.ActiveXObject) {
              XMLHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
           }
          return XMLHttpRequestObject;
        }
        //---------------------------------------------------------------
        function submitHandler(txtID, divID)
        {
              var theText = document.getElementById(txtID);
              var theArea = document.getElementById(divID);
              var reqObj = getReqObj();
              var url="cgi-bin/textchat?newtxt=" + escape("<p>" + theText.value + "</p>");
              reqObj.open("GET", url, false);
              reqObj.send(null);
              theArea.innerHTML = unescape(reqObj.responseText);
              theArea.scrollTop = theArea.scrollHeight;
              theText.value = "";
              return false;
       }
       //---------------------------------------------------------------
        window.onload=function()
        {
          document.forms[0][0].focus();
        }
    </script>
</head>

<body>
    <h2>Open Chat Site</h2>

    <div id="targetDiv" style="overflow:scroll;width:800px;height:400px;border:1px solid blue;">
    </div>

    <br> <br>

    <form name="forinput" onsubmit="return submitHandler('usrText', 'targetDiv')">
       <input type="text" id="usrText" style="width:800px">
    </form>
 </body>
</html>
```