Proyecto de Fin de Carrera

Pizarra Web Compartida

Albert Llop

29 de noviembre de 2008

Índice general

Capítulo 1

Desarrollo

Según se ha establecido mediante el estudio previo, se han diferenciado dos partes muy claras. Incluso así, es posible modularizar el desarrollo de ambas partes en elementos más sencillos, que se irán uniendo con el tiempo.

Simplificando, antes de que la interfaz en Javascript pueda interactuar con el servidor, es necesario que el sistema se haya preparado y sepa como responder a las consultas. Pero para entender totalmente qué tipo de consultas deberá ser capaz de atender, se debe comprender cuál será la implementación de la interfaz, qué elementos básicos se podrán utilizar (líneas, cuadrados, texto, etc.), para así implementar las respuestas adecuadas. Y a la inversa, la interfaz necesita saber cómo serán atendidas sus respuestas para poder tratarlas correctamente.

Es necesario, por tanto, realizar una serialización del trabajo que permita entender qué se necesita de la otra parte, antes de empezar a implementarla. No solamente porque unas partes sean requisito de las otras para que funcionen, sino porque la falta de experiencia con estas dos tecnologías hace que sea necesario un proceso de comprensión previa antes de implementar completamente.

Así pues, los pasos que se han seguido son los siguientes:

1. Interfaz Javascript. Implementación de los módulos de renderizado y dibujo.
2. Sistema. Implementación de los "modelos" de las Pizarras y sus Elementos, con sus controladores y vistas correspondientes, para permitir la introducción, modificación y consulta de nuevas figuras dentro de una Pizarra, y ser consultadas por la interfaz.
3. Interfaz Javascript. Implementación del módulo de comunicación, que utiliza las vistas implementadas anteriormente de forma adecuada.
4. Sistema. Extensión del sistema para manejar Usuarios, Grupos y Pizarras. Se añaden las vistas necesarias para que la interfaz pueda realizar consultas sobre los usuarios que participan en una Pizarra.
5. Interfaz Javascript. Implementación del módulo de usuarios, utilizando las vistas implementadas anteriormente para poder manejar los permisos de los distintos usuarios, así como mantener a los participantes informados de quién está participando.
6. Proceso de diseño de la web e integrado en el sistema.
7. Funcionalidades extra: Impresión de documentos.

1.1. Javascript: Consideraciones previas

Debido a las razones explicadas anteriormente, se deduce que Javascript es un lenguaje un tanto peculiar. Su sintaxis en la misma en todos los navegadores, pero no su DOM (Document Object Model), el cual suele cambiar sutilmente de uno a otro. Estas diferencias han hecho que tradicionalmente, saber programar en Javascript signifique conocer en profundidad dichas diferencias, y la forma en que evitarlas.

Los navegadores "proveen" a Javascript con dicho DOM, que no es más que una serie de objetos, a modo de librería, con los que poder consultar o modificar cualquier cosa. Por ejemplo, en cualquier momento se puede acceder a los objetos `navigator, document, window, location, screen`, etc, los cuales contienen una serie de atributos y métodos que se pueden consultar o ejecutar. Por ejemplo, el atributo `appName` del objeto `navigator` devuelve una cadena de texto con el identificador del navegador. Así, una forma sencilla de diferenciar los navegadores IE del resto sería mediante una línea como la siguiente:

```
if (navigator.appName == "Microsoft Internet Explorer")
```

Otro objeto muy importante es el `document`, a partir del cual se puede acceder y modificar todo el código html. Por ejemplo, mediante la línea siguiente se podría obtener un objeto que representaría la etiqueta `<svg id=‘mainElement‘>` :

```
var mainElem = document.getElementById("mainElement");
```

La mayor parte del DOM es similar en todos los navegadores, haciendo las tareas típicas muy simples. Pero es en esas pequeñas diferencias en que Javascript se convierte en algo tedioso y problemático. Con el tiempo Javascript se ha ido popularizando y por lo tanto un mayor número de programadores han dedicado sus esfuerzos al lenguaje, incrementando la cantidad de documentación y librerías disponibles en proporción a dicha popularidad. Era solo cuestión de tiempo que se encontrara una solución a los problemas que se han planteado.

La solución ha aparecido en forma de librerías que permiten hacer las acciones comunes de forma unificada, encargándose ella de distinguir entre navegadores y versiones, y hacer que siempre se ejecuten los comandos apropiados. Existen diferentes librerías para esto, y en este caso se ha decidido usar jQuery. Tomando el ejemplo de querer añadir un evento a un elemento, este es el código típico que se debería ejecutar:

```
var elem = document.getElementById("body");
if(naigator.appName == "Microsoft Internet Explorer") {
   elem.attachEvent("onmousedown", doSomething);
} else {
   elem.addEvent("mousedown", false, doSomething);
}
```

No solo es diferente el nombre de la función para añadir un evento, sino que en IE no se permite definir la política de bubbling, y además los nombres de los eventos son diferentes (onmousedown y mousedown).

El mismo código con jQuery incluida, quedaría así:

```
$("body").attach("mousedown",doSomething);
```

Aunque aparentemente se reduzca el número de líneas, a la hora de ejecutar se tienen que hacer el mismo número de comprobaciones, quedando por tanto en un rendimiento similar, sino peor. Sin embargo, dejando todo este número de comprobaciones “rutinarias” de manos de una librería hace que se produzcan menos errores, pues al ser una librería pública usada por un gran número de personas, se puede asumir hayan tenido en cuenta un mayor número de factores de los que uno es capaz trabajando independientemente. Una vez más, debido a una mayor sencillez del código, es posible realizar un trabajo más limpio y libre de errores.

No solo eso, sino que es posible añadir dicha librería directamente de la página de jquery de la siguiente forma:

```
<script src="http://jquery.com/jquery-latest.js"></script>
```

Puesto que un gran número de páginas usan estas librerías hoy en día, la mayoría de la gente ya habrá cargado este archivo anteriormente, reduciendo el tiempo de carga de nuestras páginas para una gran parte de nuestros visitantes, puesto que el código usando jQuery es mucho más reducido, y el cliente ya tendrá en su disco duro dicha librería.

El único problema de estas librerías es que tienen una limitación en cuanto a navegadores soportados. Lamentablemente al estar usando esta librería es posible que se reduzca el número de navegadores soportados por la aplicación, pero se considera que los beneficios de usarla sobrepasan los inconvenientes. Los navegadores soportados son más que razonables, y solamente una persona con un software extremadamente desactualizado tendría algún problema.

1.2. Javascript: Renderizado

Se ha establecido que se utilizará VML para renderizar los elementos en navegadores Internet Explorer, y SVG en el resto, pues es la forma de conseguir llegar al mayor porcentaje de personas de forma que no necesiten instalar nada. Estas son las versiones de los navegadores que soportan dichas tecnologías:

Navegador	Versión
Internet Explorer	5.0
Mozilla Firefox	1.5
Safari	3.0
Opera	8.0

En principio podría considerarse que cualquier persona con dichos navegadores debería ser capaz de participar en una pizarra, al menos, como espectador, pero estas tecnologías no son las únicas que pueden limitar el rango de navegadores que puedan funcionar. Ya se ha comentado que la librería jQuery tiene unos requerimientos más restrictivos.

Navegador	Versión
Internet Explorer	6.0
Mozilla Firefox	2
Safari	3.1
Opera	9.0

Es imposible dar un porcentaje exacto de uso de los diferentes navegadores, pues varía dependiendo del grupo de usuarios que tiendan a visitar este tipo de webs, y por tanto, la única forma sería tener estadísticas de dicha web a lo largo de un periodo de tiempo. Sin embargo, tomando el ejemplo de una de las pocas webs que publica sus estadísticas[1], los navegadores mencionados comprenden más del 98.5 % del total.

El objetivo de este módulo es que sea capaz de realizar los dibujos de las figuras independientemente del navegador que se esté utilizando. Para ello deberán ejecutarse comandos distintos se esté en Internet Explorer (`IE` a partir de ahora) o en otro. Se pretende, por lo tanto, proveer una serie de funciones como la siguiente:

```
function createLine(here, x1, y1, x2, y2, color, thick, fill)
```

Dicha función creará una línea que vaya de las coordenadas `x1,y1` a `x2,y2`, de color `color`, con un grueso `thick`, y con una transparencia del `fill` %. Dicho elemento se anidará al elemento padre `here`.

Los elementos implementados son: línea, polilínea (trazo que pasa por una serie de puntos, como sería el resultado de un trazo a mano alzada), círculo, cuadrado, texto e imagen. Se puede encontrar la especificación de dichas funciones en el anexo, pero son generalmente suficientes para crear, modificar y eliminar satisfactoriamente todos los elementos.

1.2.1. Requisitos

Por desgracia, para que dicho módulo funcione existen una serie de requisitos. Primero, es necesario que el `HTML` sea el adecuado.

`VML` necesita dos líneas para que el navegador sepa cómo tiene que representar las directrices.

[1] `http://www.w3schools.com/browsers/browsers_stats.asp`

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:v="urn:schemas-microsoft-com:vml">

<style>v\:* { behavior: url(#default#VML);}</style>
```

La primera línea tiene el formato típico de un documento `XML` formal. Se definen dos espacios de nombres, el princial siendo el de xhtml definido pr el `W3C`, y el segundo el de la especificación de `VML` por parte de microsoft, al cual se le añade el prefijo `v:` . Gracias a esta línea se consigue que se puedan incluir los elementos directamente en el documento, siempre y cuando empiecen por el prefijo `v:` .

La segunda línea es necesaria para que se puedan aplicar correctamente los estilos a los diferentes elementos. Dicha línea puede añadirse en cualquier punto de la cabecera (entre las etiquetas de `head`).

En cuanto a `SVG`, en cualquiera de los navegadores, es necesario que el documento sea de tipo XHTML (el código debe ser XHTML estricto, y que cuando el servidor transmita el documento, el campo `content-type` debe ser `application/xhtml+xml`. Para esto basta con cambiar la extensión del archivo a .xhtml, o modificar dicho atributo antes de ser enviado (ruby permite cambiar dicho campo, así como cualquier lenguaje de scripting como podría ser PHP).

A diferencia de VML, SVG necesita de una etiqueta principal dentro de la cual colgarán los elementos. Debería tener una estructura como la siguiente:

```
<svg id="mainElement" xmlns="http://www.w3.org/2000/svg"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   preserveAspectRatio="none" height="600px" width="800px">
</svg>
```

El atributo id permitirá acceder fácilmente a este elemento mediante javascript. Los dos atributos xmlns son también típicos de un documento `XML`, y añaden las definiciones correspondientes a los elementos `SVG` y de `xlink`. Xlink se utiliza para referenciar elementos exteriores al documento, como podrían ser imágenes, y se explicará con más detalle en secciones posteriores. El atributo `preserveAspectRatio` ayuda a que no se deforme la imagen con posibles redimensionamientos de la ventana, y `height/width` simplemente indican el tamaño de la imagen SVG (no hay que olvidar, que al fin y al cabo estamos generando una imagen de tipo SVG).

Por último, para todo tipo de navegadores es necesario definir dos variables de Javascript, una que apunte al elemento principal del cual colgarán las etiquetas y otro que apunte al elemento exterior que servirá de referencia a la hora de calcular la posición exacta del ratón respecto al documento. En VML ambos elementos pueden ser el mismo, por lo tanto teniendo un div sería suficiente. Para SVG, sin embargo es necesario que la etiqueta SVG esté contenida dentro de una etiqueta DIV.

La razón por la cual se necesitan dos variables distintas es a causa de las diferencias en cuanto a la estructura proporcionada por el DOM. Objetos *clásicos* del `HTML` como un DIV traen una serie de funciones y propiedades que el propio navegador proporciona, como son los atributos `offsetLeft` y `offsetTop`, los cuales sirven para saber la posición del ratón de forma precisa. La etiqueta SVG, sin embargo, no trae dichos atributos, con lo cual es necesario que esté ubicada dentro de un elemento estructural de HTML.

En el caso de VML esto no es necesario puesto que es posible añadir elementos VML en cualquier parte del código, y solo es necesario tener un DIV para contener todos los elementos bien ordenados y posicionados respecto al marco de lo que sería la pizarra. En SVG, al estar embediendo un documento SVG dentro de un documento XHTML, es necesario que mantenga toda su estructura típica, que es con un elemento base SVG.

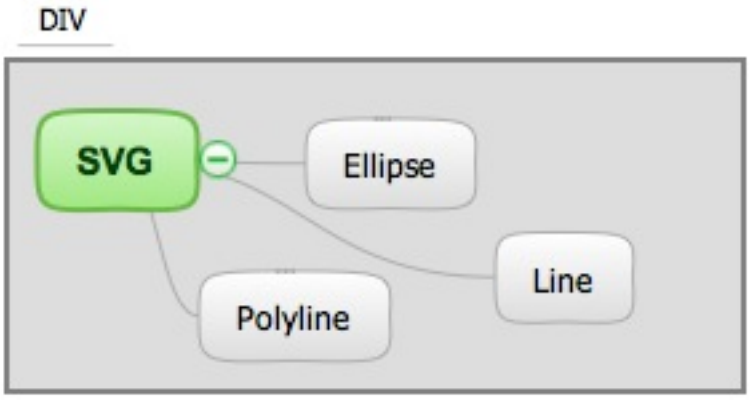

Figura 1.1: Esquema del concepto de SVG dentro de un DIV

Las dos variables a definir son `mainElem` y `container`.

1.2.2. Implementando

Todas las funciones de renderizado funcionan de la misma manera. Si se añadiera al código la línea siguiente:

```
<div id="container">
  <v:line from="0,0" to="200,200">
    <v:stroke color="#000000" weight="1" opacity="0.8"/>
  </v:line>
</div>
```

se obtendría una línea que iría del punto 0,0 al 200,200 dentro del div contenedor, de color negro (`#000000`), de 1 pixel de grosor y una opacidad del 80 %.

Lo que se pretende lograr con el módulo en javascript, es que se pueda ejecutar la línea siguiente:

```
line = new Line(document.getElementById("mainElement"),0,0,200,200,"#000000",1,0.8);
```

en cualquier momento mediante Javascript, y que automáticamente se cree ese nodo `<v:line>`. Gracias a que es posible ejecutar código javascript en cualquier evento, es posible crear una línea cuando se pulsa el ratón, e ir modificando sus coordenadas mientras se mueve. No solo eso, sinó que si se recibe alguna información, por ejemplo, mediante Ajax, es posible dibujar dinámicamente elementos a la vez que el servidor los va mandando, sin tener que recargar la página. Es posible usar, por tanto, estas funciones del motor de renderizado tanto para simular el hecho de estar dibujando (motor de dibujo), como para recibir los dibujos que realizan el resto de participantes de una pizarra (módulo de comunicaciones).

Javascript tiene todas las funciones necesarias para modificar el DOM tanto como se desee. Se ha visto que es posible obtener un elemento del código mediante el `document.getElementById`, y una vez se tiene ese objeto, existen funciones como `appendChild` para ir modificándolo.

Gracias a jQuery, y a que Javascript es un lenguaje muy relajado sintácticamente, se han podido salvar las distancias entre VML y SVG de forma bastante sencilla. Al final siempre se tiene que implementar cada función dos veces, puesto que la sintaxis de VML y SVG es diferente (hay elementos iguales, pero la sintaxis para crearlos no es la misma), pero estas funciones hacen transparente este proceso.

Una vez entendido que existe la posibilidad de recrear gráficos vectoriales mediante javascript, SVG y VML, es necesario recordar los requisitos que se establecieron, y considerar cuáles de los gráficos que se querrán renderizar serán posibles y cuales no. Repasando individualmente las herramientas especificadas inicialmente:

Lápiz En ambos SVG y VML existe el elemento `polyline`, cuya representación es la de una serie de puntos en la superficie, pudiéndose representar por tanto un trazo libre como el de la herramienta lápiz.

Líneas rectas Las líneas rectas se corresponden fácilmente con el elemento `line` de SVG y de VML.

Cajas y elipses Para SVG se dispone de los elementos `ellipse` y `rect`; para VML se dispone de `oval` y de `rect`.

Selección de color, grosor y transparencia (subrallador) Se requería poder personalizar estos tres parámetros. Considerando los usos para los que esta librería está concebida no será necesaria una gran capacidad de personalización. Se considera importante poder especificar estos tres parámetros a la hora de la creación, pero no necesariamente a la hora de la modificación de elementos ya creados. En cuanto a la transparencia, para líneas rectas y trazos de lápiz, la línea en si será la que haga la transparencia, para cajas y elipses, solo el contenido. En el caso de SVG, esto es posible de conseguir mediante CSS, utilizando las reglas `stroke` (color del trazo), `stroke-width` (grosor), `stroke-opacity` (transparencia del trazo), `fill` (transparencia del relleno) y `fill-color` (color del relleno). En cuanto a VML, este proceso es un tanto más complejo, por la necesidad de anidar los elementos `stroke` (para las características del trazo) o `fill` (características del relleno) dentro de cada elemento VML, pero totalmente posible.

Texto El elemento texto, a diferencia del resto, no necesita de SVG o VML para poder representarse. Mediante HTML y CSS es posible posicionar cualquier texto en cualquier posición, y mucho más sencillo de implementar que utilizando los elementos `text` de SVG o `textbox` de VML. Esta función se implementará mediante un DIV posicionado en el inicio de las coordenadas, con un texto siempre en el mismo tamaño de fuente y tipo de letra, adecuado al uso de la aplicación. Este elemento no necesitará de función de modificación, puesto que no se va a generar dinamicamente como el resto, sino que se introducirá el texto, y luego se añadirá permanentemente al documento.

Goma de borrar Al estar enfocadas las funciones de forma orientada a objetos, simplemente con destruir los objetos o descolgarlos del DIV contenedor (`parentElement.removeChild();`) basta para hacerlos desaparecer. Será tarea del motor de dibujo el encontrar qué elementos desean ser borrados, y actuar en consecuencia.

Imágenes Este elemento es fácilmente implementable por ser similar al Texto en cuanto a que es posibleme implementarlo mediante HTML y CSS.

Se ve que todo es posible, y finalmente tras la experiencia, tan solo cuestión de saber sobreponer las minúsculas diferencias de los navegadores a la hora de interpretar javascript, de interactuar con jQuery, y de la gran falta de herramientas para depurar Javascript disponibles.

1.3. Javascript: Dibujo

La sictuación actual es de la disposición de una serie de funciones con las que generar gráficos vectoriales y modificarlos, funcionando en la gran mayoría de exploradores. Proceder a implementar unas herramientas de dibujo mediante esta librería es ahora posible. En esta sección no se pretende describir el proceso de creación de una herramienta de dibujo genérica, sino las peculiaridades de Javascript respecto a la forma en que trata los eventos, las diferencias entre navegadores, y la influencia de la estructura del documento (HTML y CSS).

El reto con el que uno se enfrenta inicialmente es el de conseguir capturar los eventos de ratón de forma adecuada. En los usos típicos de las herramientas deseadas, y exceptuando la introducción de texto, toda la interacción se hace mediante el ratón. Hay que recordar también, el entorno en el que estas acciones se ejecutarán: habrá un marco en la página web (un elemento DIV) dentro de el cual se podrá dibujar, y una serie de elementos externos a este marco. Esto es importante puesto que es posible que los elementos exteriores interfieran de alguna forma. Dichos elementos exteriores deberán ser los siguientes:

Elementos estáticos El nombre del documento, el link de vuelta a la web *normal*, así como cualquier otro elemento estático.

Barra de herramientas Debe haber alguna forma, lo más sencilla posible, de elegir las herramientas a usar, y de configurarlas (elegir el color, grosor y transparencia). Estos elementos pueden complicar más la tarea pues deben ser dinámicos, y controlar también eventos de ratón y/o teclado.

Lista de usuarios Esta lista deberá ser dinámica también, puesto que los usuarios podrán entrar y salir de la pizarra, y deberá informarse al resto de usuarios activos de alguna forma. No influirá con el ratón, pero si ejecutará operaciones periódicamente, y ejercerá cambios sobre el código de la página, si bien no sobre la pizarra en si.

Selector de páginas De la misma forma que la lista de usuarios, éste influirá poco en cuanto a eventos se refiero, pero será actualizado dinámicamente, y debe permitir flexibilidad por si el usuario no tiene permiso para cambiar de página.

En este momento, para dibujar, es solo necesario considerar la barra de herramientas. Más adelante, cuando se implementen más funciones, se deberá considerar cómo implementarlas, y de qué forma influencian a lo que ya se tiene.

1.3.1. Barra de herramientas

La opción de selección de herramientas es relativamente sencilla, puesto que solo necesita detectar cuando se clica en el elemento representativo de la herramienta y defina que esa es la herramienta activa. En javascript existen dos formas básicas de capturar eventos. La primera es utilizando el propio código HTML, con una línea como la siguiente:

```
<div id="line-tool" onClick="setTool('line');">Línea</div>
```

Todos los navegadores entienden estas definiciones, aunque no se consideran elegantes, y son bastante restrictivas por diversas razones.

Una segunda manera sería, mediante jQuery, con una línea como la siguiente (suponiendo que se quiere definir la herramienta línea al hacer click sobre el elemento con id `line-tool`):

```
$("#line-tool").bind("click", function(){setTool('line');});
```

Aunque aparentemente pueda parecer más complicada, es mucho más limpia en cuanto a que se separa lo que es el contenido de la web (HTML) con lo que es su comportamiento, de la misma forma que se separa su diseño utilizando CSS, y se considera poco elegante y práctico añadir estilos directamente en el código de los elementos.

En cuanto al problema de la configuración de las herramientas, existen múltiples y variadas soluciones. No es posible encontrar de forma nativa una manera de seleccionar un color, o de tener una barra de desplazamiento. Las únicas herrmientas de las que se dispone en HTML para introducción de datos, son campos de texto o listas. Esto, aunque suficiente, es poco útil para el usuario, pues es más sencillo seleccionar un color de una paleta de colores que escribiendo su código RGB. De la misma forma, es más intuitivo usar una barra de desplazamiento para elegir el grosor o la transparencia, que escribiendo sus valores directamente. En otro tipo de herramientas sería interesante poder hacer estas cosas a mano, si se necesitara de más precisión, pero en esta aplicación es más prioritaria la agilidad de uso.

A pesar de no existir nada directamente en HTML, existen múltiples opciones que, mediante javascript, simulan estos elementos.

Colores

Para esta opción se ha decidido crear un mini-panel con ocho posibles colores, en vez de tener un selector de colores completo como los típicos de los programas de dibujo. No se cree necesario elegir una grandísima variedad de colores y con mucha precisión, sino ofrecer una pequeña cantidad de colores muy diferentes, vivos y fácilmente identificables.

Figura 1.2: Selector de colores

El código para este selector es extremadamente sencillo y no consta más que de elementos sobre los que clicar para seleccionar el color, de la misma forma que se escogen herramientas.

Grosor y transparencia

Mediante jQuery UI [2], una extensión a jQuery, es posible generar los llamados Sliders. Con un par de líneas de código se puede generar un Slider que permita seleccionar un valor de entre un rango, y que ejecute código personalizado al mover la manilla y al soltarla:

```
$("#fill-dialog").slider({min: 20, max:100, startValue: 80, change: function(e,ui){
  fill = (ui.value/100);
}, slide: function(e,ui){
  $("#fill").html(ui.value);
}});
```

Esta función crea un slider en el elemento con id `#fill-dialog`, con posibles valores entre el 20 y el 100, puesto de forma pre-definida a 80, que al cambiar actualiza el contenido del elemento con id `#fill` (campo de texto mostrando el valor actual), y que al soltarse actualiza la variable `fill` con el valor seleccionado.

[2] jQuery UI - http://ui.jquery.com/

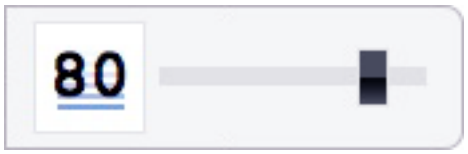

Figura 1.3: Slider selector de relleno (transparencia)

1.3.2. Eventos de ratón

Los tres eventos básicos de un ratón son los de pulsar un botón (`mousedown`), moverse (`mousemove`) y soltar el botón (`mouseup`). Los tres son necesarios, puesto que son, en ese orden, cuando se empieza a dibujar un elemento, cuando se va modificando, y cuando finalmente se *suelta*, quedando gravado definitivamente hasta que es borrado (a excepción del texto, que es un tanto más peculiar, y se explica en otro apartado).

El que solo se quiera poder dibujar dentro del marco seleccionado para ello, hace que se planteen una serie de problemas.

- Cuando se empieza a dibujar dentro del marco, pero se mueve el ratón fuera de él.
- Cuando se suelta el botón fuera del marco.
- O peor aún, cuando se suelta el botón fuera de la ventana.

¿Cómo funcionan los eventos en Javascript?

En Javascript es posible capturar eventos en cualquier elemento, y siempre sucederá de forma jerárgica según la visibilidad que se tiene de los distintos elementos. Es decir, si se tiene un trozo de código como el siguiente:

```
<div>
  <p>Hola <a id="albert-link" href="/users/albert">Albert</a>, qué tal estás?</p>
</div>
```

y se hace click en la palabra Albert, saltará un evento en el elemento `a`, cuando acabe de hacer lo que sea que debe hacer, y pasará a ejecutar el evento del elemento `p`, seguirá con el del `div`, y acabará con el del elemento `body`, que en cada documento HTML es el contenedor de todo lo que se ve en pantalla. Por tanto, si se quisiera capturar cualquier movimiento de ratón en la pantalla, se deberían capturar los eventos `mousemove` del elemento `body`.

Por otra lado, el ejemplo anterior tiene un problema, y es que el evento `click` de todo elemento `a` lo que hace es mandar al usuario a la página que éste linka. En caso de querer, por ejemplo, en vez de mandar al usuario a la página con los detalles personales de Albert, que está en `/users/albert`, si no que queremos hacerlo más dinámico y mostrarlo directamente mediajte Javascript, se debería capturar el evento click de dicho elemento, y realizar lo necesario (una llamada por ajax, creación o modificación de otro elemento para mostrar la información, etc). Pero el comportamiento de los eventos en Javascript, por defecto, al acabar de atender el evento capturado por el usuario, sigue ejecutando el comportamiento normal de dicho elemento (ir a la página enlazada), a menos que la función ejecutada devuelva un valor falso.

Es decir, el código javascript debería ser semejante al siguiente:

```
$("#albert-link").bind("click",function(){
  // Obtener información y mostrarla dinámicamente
```

```
    . . .
    return false;
});
```

Ésto es considerado una buena técnica puesto que en caso de estar ante un usuario sin soporte para Javascript, el sistema mostrará la información de todas maneras, aunque no de una forma tan dinámica. En el caso que interesa a este proyecto, hay que comprender este comportamiento para que, por ejemplo, se pueda seguir dibujando aun estando fuera del marco, o para tener varias cosas que estén encima del marco, poder trabajar con ellas mediante eventos, y que éstos no influyan al proceso de dibujo.

¿Cómo capturarlos?

En este caso, por el comportamiento que se le quiere dar a la aplicación, es necesario estar siempre al tanto de los siguientes eventos:

- Cuando se pulsa un botón del ratón dentro del marco. Si se pulsa el ratón fuera del marco, no debe afectar al dibujo, más que en el caso de las herramientas o el selector de páginas, pero estos trabajan de forma independiente.
- Debemos saber en todo momento cuándo se está moviendo el ratón, y cuándo se suelta el botón, debido a los casos explicados anteriormente, si va dibujando hasta soltar el ratón fuera del marco, o de la ventana. Si se ignoraran los eventos fuera del marco, se producirían comportamientos incómodos, como que se deje de mover una línea por rozar un borde, o que, al no enterarse de que se ha soltado el botón, se siga dibujando aún cuando no se está pulsando.

Sabiendo, como se ha comentado antes, que siempre se tendrá un elemento con id `#container`, que será el `marco`, el código para capturar los eventos es el siguiente:

```
$(document).ready(function(){
  $(container).bind("mousedown", mouseDown);
  $("body").bind(   "mousemove", mouseMove);
  $("body").bind(   "mouseup",   mouseUp);
});
```

En este caso, en vez de implementar las tres funciones directamente, se pasan como parámetro. Puesto que Javascript entiende las funciones también como objetos, es posible definirlas con anterioridad de la misma forma que se definiría una variable, para luego usarla como parámetro. jQuery automáticamente pasa a estas funciones la **variable de evento**, que es un descriptor del entorno en que ha sucedido el evento. Cosas como la posición del ratón, el elemento al que se estaba apuntando, qué botón se ha pulsado, se pueden consultar en este objeto.

¿Dónde está el ratón?

No hay que olvidar, que lo que se está haciendo es, en cierta medida, *incrustar* un elemento SVG, o un conjunto de elementos VML, dentro de otro elemento DIV. A la hora de crear elementos, por tanto, es necesario especificar las posiciones como pixels respecto del inicio del marco. Mediante jQuery es posible obtener la posición absoluta del ratón, respecto del total de la ventana del navegador, pero para este caso es necesario algo más preciso, que tenga en cuenta también la posición del DIV, y o si se ha hecho scroll o no.

Los atributos que se pueden consultar mediante la **variable de estado** (`event` a partir de ahora) del evento, y el resto del DOM, y que son útilies para posicionar elementos, son las siguientes.

`event.clientX` **y** `event.clientY` Estos atributos devuelven la posición del ratón respecto a lo que ahora se ve en el navegador. Esto quiere decir, aunque se haya hecho scroll, si se clica en la esquina superior izquierda, ambos valdrán 0.

`container.offsetLeft` **y** `container.offsetTop` Recordando que `container` es el DIV contenedor dentro del cual se anidarán los elementos, estos dos atributos devuelven su posición respecto a los bordes absolutos de la página, ignorando si se ha hecho scroll o no.

`document.body.scrollLeft document.body.scrollTop window.pageXOffset window.pageYOffset` Este es un claro ejemplo de las diferencias comentadas entre el DOM ofrecido por diferentes navegadores, y que generalmente jQuery ayudaría a resolver. Los dos primeros funcionarían en `IE`, y devuelven la cantidad de pixels que se ha hecho scroll, tanto lateral como verticalmente, y los segundos son sus equivalentes para todo el resto de navegadores.

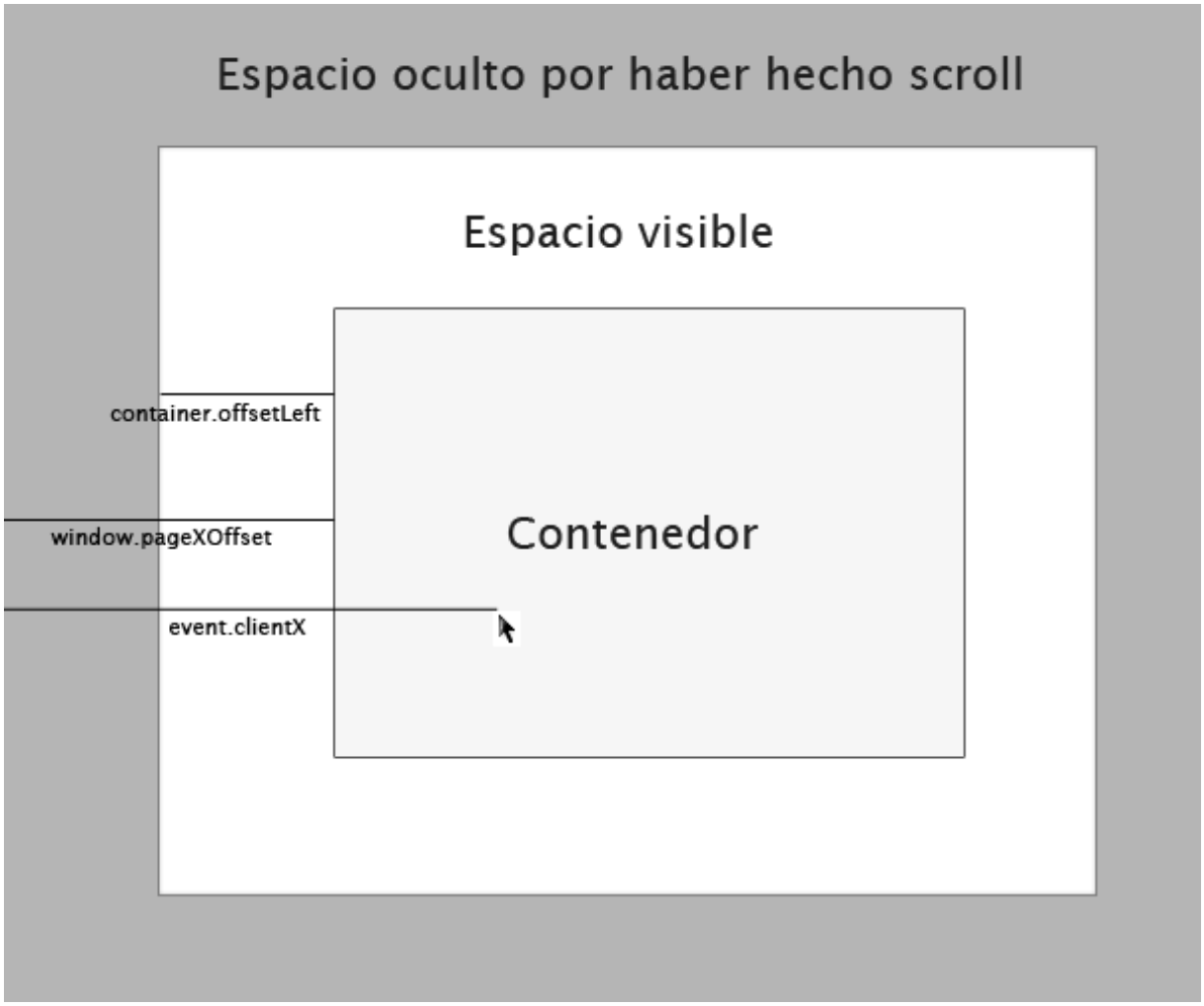

Figura 1.4: Esquema de distancias de los distintos elementos

La figura 1.4 muestra un pequeño esquema de estas distancias. Mediante estos tres tipos atributos es posible averiguar la posición del ratón respecto al DIV contenedor, sea cual sea la situación de la página, con estas instrucciones.

```
//IE
x = e.clientX - container.offsetLeft + document.body.scrollLeft;
y = e.clientY - container.offsetTop + document.body.scrollTop;
//RESTO
x = e.clientX - container.offsetLeft + window.pageXOffset;
y = e.clientY - container.offsetTop + window.pageYOffset;
```

Con estos conocimientos ya es posible programar las herramientas de línea, trazo de lápiz, elipse y caja, que son la base de lo que este proyecto pretendía. El resto del trabajo necesario para programar estas herramientas es de puesta en práctica de estos apartados, y no se considera importante destacar nada más. El código de la implementación se puede encontrar en el anexo.

1.3.3. Introducción de texto

El método tradicional para introducir texto en páginas web, es mediante los elementos de formulario `input` de tipo `text` o los llamados `textarea`. Para esta situación el más adecuado será el `textarea` puesto que el otro es simplemente una línea, y los `textarea` se pueden redimensionar a voluntad.

El efecto que se quiere lograr es el de crear un pequeño recuadro donde escribir, y al hacer click fuera o pulsar tabulador, este texto pase a formar parte del documento. Existe un requisito importante, y es que la posición del dexto mientras se escribe, y cuando se introduzca *definitivamente*, debe ser la misma, para así no confundir al usuario.

Este proceso difiere del resto de herramientas por necesitar de dos pasos distintos, uno para crear un `textarea` en alguna posición, y otro pra acabar agregándolo finalmente.

Generación de un textarea

En este caso, la creación del elemento no será en el evento `mousedown`, puesto que los usuarios están acostumbrados que las cosas suceden al soltar el botón del ratón, no al pulsarlo. Al detectar un evento de tipo `mouseup` con la herramienta de texto seleccionada, y siempre y cuando se haya clicado antes dentro del contenedor, se crea diámicamente eset textarea, posicionado mediante CSS con origen en la posición del ratón.

Introducción del texto

Para poder averiguar cuando el usuario quiere dejar de escribir e introducir el texto en el documento, hay que simular de alguna manera las distintas posibilidades por las que un usuario puede pretender hacer esto. Lo más intuitivo es que esto suceda cuando el elemento pierda el `focus`, es decir, cuando se clique fuera o se pulse tabulador. La forma de asegurarse que esto sucede es capturando los eventos `change` y `blur`, además de mediante los eventos que ya se capturan de mousedown y mouseup.

Al detectar este suceso, se genera un nuevo DIV, que estará anidado dentro de del contenedor igual que el resto de elementos, y posicionado mediante CSS en el mismo lugar en el que estaba el `textarea`, con la misma fuente, tamaño y color.

1.3.4. Goma de borrar

Existen dos vertientes diferentes a la hora de plantear la goma de borrar. La funcionalidad típica a la que se está acostumbrado en los programas de dibujo, es a la de una goma que, de forma inversa a lo que sería el lápiz, borra solamente el trozo por el que se pasa por encima. Esta vertiente, por el tipo de datos con los que se está trabajando, sería extremadamente difícil de implementar, pues al pasar por en medio de un elemento, se debería detectar en qué punto ha pasado, y dividir los elementos en múltiples trazos (si son líneas o trazos de lápiz), o prácticamente imposible de representar si se tratara de una caja o una elipse.

La otra vertiente, más usada en los programas de pizarra compartida por la sencillez, tanto para el programador, como para el usuario, es la una goma de borrar que no elimina solo aquellas zonas por las que pasa, sino que elimina el elemento entero. Situándose en la posición del usuario que está utilizando esta pizarra para acompañar sus explicaciones, la situación clásica es la de estar resaltando distintos elementos de la página, que luego querrá eliminar para seguir su explicación. Tener la facilidad de clicar en el elemento, y que este desaparezca por completo, sería muy práctico, y posiblemente la mejor solución.

Existen múltiples formas de enfocar este problema, puesto que es posible capturar eventos del ratón en cualquiera de los elementos que se generan, no solamente en el contenedor. Sin embargo, se considera que esto es excesivamente complejo, y que gracias al atributo `target` de la **variable de estado** del evento, es posible solucionar todos los problemas con los eventos que ya se capturaban con anterioridad.

Suponiendo que se tiene la herramienta de goma de borrar seleccionada, y que se quiere borrar algo que está en un punto donde coincide, por ejemplo, un círculo y una línea. El círculo se ha dibujado después de la línea, por lo tanto está por encima, así que lo más lógico es que se borrara primero. Al hacer click, el evento se generará en el elemento círculo, pues es el que está visible en ese momento, pero tal y como se ha explicado en la sección **¿Cómo capturarlos?** 1.3.2, este evento irá saltando de superior al inferior, llegando, en efecto, a los eventos que se capturan en el contenedor o en la etiqueta `body`.

Una vez más, mediante la `variable de estado` es muy fácil saber cuál fue el objeto que generó el evento originalmente, a pesar de que en ese momento se esté tratando el evento de un elemento *inferior*. Y una vez más, existe un conflicto entre los atributos generados por `IE`, y los atributos generados por el resto de navegadores.

```
// IE
event.srcElement
// SVG
event.target
```

1.4. Javascript: Comunicación

En estos momentos ya es posible renderizar gráficos vectoriales, así como poder dibujarlos de forma interactiva. Es, a efectos prácticos, una **Pizarra Web**, faltando solo hacerla compartida. Para conseguir esto se han de conseguir dos funcionalidades básicas:

Persistencia de los elementos Es necesario que al crear y eliminar elementos, estos se refleje en una base de datos de fondo, para que en un futuro, estos elementos siguen ahí, y para que el resto de la gente pueda obtener esos elementos.

Comunicación entre usuarios Es decir, que los elementos generados por un usuario sean vistos por el otro, de forma automática.

La naturaleza de la web hace esto realmente complicado. El flujo interacción entre un úsuario y una página web es el siguiente:

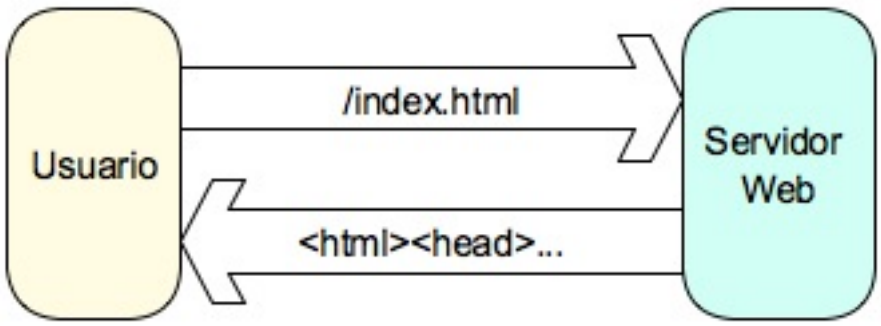

Figura 1.5: Interacción usuario - servidor

Es un una interacción totalmente pasiva por parte del servidor, el cual solo contesta cuando y si se le solicita un documento. No solo eso, sino que típicamente, la forma de navegar por páginas webs es de ir recargando cada página individualmente. Para este proyecto se necesita algo más, pues la interacción entre el usuario y el servidor tiene que ser totalmente transparente, y por supuesto la página no debe de recargarse cada vez que suceda algún cambio.

En el año 2002, y con el navegador Internet Explorer 5 [3], se introdujo la interfaz `XMLHttpRequest`, que permite abrir conexiones HTTP con el servidor de forma dinámica mediante Javascript, de forma transparente al usuario, puesto que no necesita recargar la página. Gracias a esto, es posible establecer un flujo de información entre el usuario y el servidor, con el que no solamente transmitir los cambios que dicho usuario haga al documento, sino que permita recibir información de lo que el resto de gente hace.

Las conexiones `XMLHttpRequest` tienen las mismas características que cualquier otra conexión abierta por un navegador de forma tradicional, con las únicas limitaciones de no poder enviar o recibir archivos de caracter binario. Eso quiere decir que, por ejemplo, es posible pasar parámetros. La diferencia, sin embargo, es que para este tipo de peticiones, no es nada útil que el servidor conteste con documentos que sean páginas web típicas, sino que es necesario que la información sea devuelta en algún formato fácil de entender por un lenguaje de scripting como Javascript. Habitualmente se usa XML, de ahí el nombre de la interfaz, pues es una sintaxis muy adecuada a este tipo de usos, pero es posible recibir cualquier tipo de texto, y de hecho, existen mejores alternativas cuando dicho resultado se va a usar con Javascript. En aplicaciones más

[3] http://es.wikipedia.org/wiki/XMLHttpRequest

modernas se utiliza JSON [4], que es básicamente la misma sintaxis utilizada en Javascript para generar objetos (Arrays y Hashes).

```
{
  "page":1,
  "objets": [{"type":"line","x1":10,"x2":20,"y1":0,"y2":5},
             {"type":"circle","x":20,"y":20,"radius":15}]
}
```

Ese mismo trozo de código, si se escribiera dentro de una pieza de código Javascript, generaría un Hash con dos atributos, `page` y `objects`, y object sería un Array con dos elementos, que a su vez son Hashes. Teniendo ese trozo de texto, en la variable `response` por ejemplo, que se ha consultado mediante XMLHttpRequest, solo hay que ejecutar la función `eval` sobre `response` para generar el objeto correspondiente.

Como era de esperar, la forma de realizar consultas mediante XMLHttpRequest difiere de un navegador a otro, por eso es muy útil utilizar alguna librería, en este caso jQuery, que lo hace tan fácil como usar la función `get(url)` para consultar una página mediante `GET`, o `post(url)` para hacerlo mediante `POST`.

```
response = $.get("/list_of_elements.json");
object = eval(response);
alert(object.page);
```

Suponiendo que el documento `list_of_elements.json` contiene el trozo de JSON anterior, al realizar un `alert(object.page)`, se mostrará el valor **1**.

De la misma forma, se puede utilizar la versión más completa de las funciones para poder enviar información:

```
$.ajax({
  type: "POST",
  url: "/save_circle",
  data: {x: 1, y: 2, radius: 20}
});
```

este comando crearía una solicitud HTTP como la que se genera cuando se rellena un formulario, mandándola mediante POST, con los atributos especificados en data. Lógicamente es necesario un sistema que sea capaz de entender todas estas solicitudes, pero por ahora este capítulo se centrará en como el Javascript tratará todas las comunicaciones necesarios con el servidor, suponiendo siempre que existe el sistema que actuará de forma adecuada.

[4] http://es.wikipedia.org/wiki/JSON

Crear y destruir elementos

Esta parte es la más sencilla, puesto que solo es necesario comunicar al servidor de los cambios realizados por el usuario en concreto. En todos los casos está muy claro cuando se crea un elemento, y cuando se elimina, por tanto, ese será el momento de comunicarlo.

El primer problemaes decidir de qué forma se va a representar un elemento. Recordando que cada elemento tiene atributos diferentes, y que todo ha de acabar siendo almacenado en una base de datos relacional, es posible que todo pueda llegar a complicarse. Lo más sencillo en este caso sería que todos los elementos, sean del tipo que sean, pudieran ser guardados en una misma tabla de una base de datos, y que por tanto, tubieran los mismos atributos.

Puesto que siempre que se consulten datos al servidor, éste los comunicará en forma de JSON, se ha creído adecuado que ocurra el mismo proceso en sentido inverso. La forma de que todos los objetos puedan estar en la misma tabla, es codificándolos de alguna manera de forma que puedan representarse mediante una cadena de texto. De la misma forma que, con una cadena de texto en formato JSON es posible generar un objeto en Javascript, es posible realizar el proceso opuesto. Suponiendo la existencia de una función `toJSON(object)` que devuelva un string representativo del objeto, el proceso para comunicar al servidor un nuevo elemento, podría ser el siguiente:

```
function save(it){
 var text = toJSON(it);
 $.ajax({
 url: "/add_element",
 data: {attr: text}
 });
}
```

Ésta es una versión simplificada de la función que se ha acabado usando en realidad, pero ya sirve para ver el proceso que se ha seguido. Es necesario añadir una serie de mejoras, que se comentan a continuación.

¿Cómo sabe el sistema a qué documento pertenece este elemento? Es necesario enviar información para que se sepa a qué documento corresponde este elemento. Puesto que habrá una base de datos de fondo, es posible suponer que siempre habrá un identificador para documento, y que es conocido de alguna manera por el javascript, pues se habrá definido al cargar la página.

```
function save(it){
 var text = toJSON(it);
 $.ajax({
 url: "/add_element",
 data: {attr: text, :doc: docId}
 });
}
```

La parte de crear elemento se puede considerar como resuelta. En cuanto eliminar objetos, es necesario saber algo más. ¿Cómo se le comunica al sistema qué elemento se ha eliminado? De la misma forma, es posible suponer que en la base de datos, estos objetos tendrán asignado un identificador como clave primaria. Este identificador se genera al crear dicho elemento, por lo tanto no es posible saberlo hasta que no se ha comunicado al servidor. La solución para esto es modificar la función de guardado de elementos, haciendo que el servidor conteste a la consulta que agrega un elemento, con el id generado para dicho elemento. Aunque una consulta se utilice para transmitir información, el servidor siempre debe contestar algo, por lo tanto es posible aprovechar estos datos:

```
function save(it){
 var text = toJSON(it);
 $.ajax({
   url: "/add_element",
   data: {attr: text, :doc: docId},
   success: function(ret){
     it.element.id=ret;
   }
 });
}
```

Este cambio agrega código personalizado al evento success de esta consulta XMLHttpRequest. jQuery lanza eventos en diferentes puntos de las consultas para que el programador pueda ejecutar distintas acciones en los distintos puntos de la consulta. Los eventos generados son `beforeSend, complete, success` y `error`, todos ellos suficientemente descriptivos.

En este caso, cuando la consulta finaliza y es satisfactoria, y suponiendo que el servidor ha contestado simplemente con el id del elemento creado, se coge el objeto que se había guardado, y se modifica añadiéndole un campo id con dicho valor. De esta forma, los elementos representados en pantalla tienen el id correspondiente a su equivalente de la base de datos, y permiten, por tanto, simplificar la función de borrado a un simple:

```
//IE
$.get("/remove_element/" + event.srcElement.id);
//RESTO
$.get("/remove_element/" + event.target.id);
```

utilizando, como ya se había comentado anteriormente, los atributos `srcElement` y `target` de la **variable de estado**, para saber con qué elementos se estaba tratando.

Sincronización entre usuarios

Uno de los mayores retos a los que se enfrenta esta aplicación, es mantener a todos los usuarios sincronizados. Hasta este momento es posible mantener al servidor informado de la creación y eliminación de elementos, pero no es posible informar al resto de usuarios de dichas acciones. En el caso de aplicaciónes típicas, con un paradigma cliente-servidor en que el servidor pudiera iniciar transferencias de información, lo lógico sería que cuando hubiera algún cambio, el servidor informara al resto de clientes de los cambios.

Sin embargo, en el caso de una página web, esto no es posible. No hay ninguna posibilidad de que el servidor establezca una conexión con el usuario por iniciativa propia, y por tanto la única solución restante es realizar consultas periódicas en busca de nuevos cambios. El tiempo entre consulta y consulta, así como la latencia entre el servidor y el cliente, definirá el tiempo máximo entre la creación de un elemento y el momento en que el resto de usuarios es consciente de los cambios.

$tMax = 2tLatencia + tIntervalo$