INFORME FINAL DE TRABAJO DE GRADO
MAESTRÍA EN INGENIERÍA ELECTRÓNICA

TÍTULO:

Arquitectura de Computación Distribuida para la Web 3D

AUTOR:
Jorge Andrés Zaccaro Valverde

DIRECTOR:
Juan Pablo Garzón Ruiz

ASESOR:
Luis Carlos Trujillo Arboleda

Departamento de Ingeniería Electrónica
Pontificia Universidad Javeriana
Bogotá D.C., Colombia
Octubre 1 de 2012

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE SEGMENTOS DE CÓDIGO

33. Inicio de la ejecución de un Servidor de Zona en Zone.js

34. Solicitud de zonificación de un entorno virtual en Zone.js

35. Manejo de los mensajes de zonificación recibidos del Servidor Web Coordinador

36. Página de despliegue con la información de contacto de la capa de aplicación

37. Establecimiento de la conexión WebSocket con los datos de la página de despliegue

38. Recepción del modelo de un entorno virtual en un Cliente Web

39. Manejo del evento 'onkeydown' en un Cliente Web de Zone.js

40. Verificación del alcance del rango visible de un avatar

41. Recepción de una solicitud de elementos visibles en un Servidor de Zona vecino

42. Recepción de los elementos visibles solicitados a un Servidor de Zona vecino

43. Acciones de un transmisor JAMP

44. Acciones de un receptor JAMP

45. Manejo de los mensajes de 'bookIn' y 'checkIn' en un Servidor de Zona

46. Archivo del modelo de prueba del sistema ('model.json')

47. Carga del modelo de prueba en el Servidor de Zona 0

48. Iniciación del Servidor de Zona 0

49. Solicitud de zonificación del Servidor de Zona 0

50. Iniciación de los Servidores de Zona 1, 2, 3 y 4 sin proporcionar un modelo

51. Arreglo de Servidores de Zona disponibles para unirse a una cuadrícula de zonificación

52. Página de despliegue con la información de contacto del Servidor de Zona 1

53. Establecimiento de la conexión WebSocket con el Servidor de Zona 1

LISTA DE ECUACIONES

LISTA DE TABLAS

LISTA DE ABREVIATURAS

1.	**3D**	Tridimensional; Tres dimensiones
2.	**HTML5**	HyperText Markup Language, Versión 5
3.	**WebGL**	Web Graphics Library
4.	**OpenGL**	Open Graphics Library
5.	**NPM**	Node Package Manager
6.	**HTTP**	HyperText Transfer Protocol
7.	**API**	Application Programming Interface
8.	**AJAX**	Asynchronous JavaScript and XML
9.	**XML**	Extended Markup Language
10.	**MVC**	Model-View-Controller
11.	**UML**	Unified Modeling Language
12.	**CSS**	Cascading Style Sheets
13.	**PHP**	Hypertext Preprocessor
14.	**XAMPP**	(X=multi-plataforma) Apache, MySQL, PHP and Perl
15.	**JAMP**	JavaScript Asset Migration Protocol
16.	**JSON**	JavaScript Object Notation
17.	**FTP**	File Transfer Protocol

1. INTRODUCCION

Internet ha revolucionado la forma en que los seres humanos nos comunicamos y compartimos información. En particular, la *World Wide Web* ha facilitado el intercambio de información al incorporar cada vez más formas de contenido (e.g. texto, imágenes, video) y mecanismos de interacción (e.g. blogs, foros, wikis).

Actualmente, uno de los retos en la evolución de la Web consiste en proporcionar soporte de contenido 3D e interacciones en tiempo real sin la instalación de programas adicionales o *plug-ins* como Adobe Flash, por lo que organizaciones como el *Consorcio World Wide Web* y el *Grupo Khronos* se encuentran trabajando en estándares y tecnologías como HTML5 [1] y WebGL [2], que ya ofrecen herramientas para la presentación, manipulación y comunicación de contenido 3D en la Web utilizando el lenguaje JavaScript [3].

Si bien componentes de estas tecnologías como el elemento `<canvas>` de HTML5 facilitan la incorporación de contenido 3D en el lado del cliente, aún no existe una solución estándar en el lado del servidor que utilice tecnologías como el protocolo de comunicaciones *WebSocket* [4] para coordinar las interacciones de los usuarios en tiempo real, ni una arquitectura de computación distribuida que resuelva los problemas típicos de los entornos virtuales de gran escala que harán parte de la Web 3D.

Uno de estos problemas es la concurrencia, pues entornos virtuales de gran popularidad como *World of Warcraft* deben manejar la carga de procesamiento de cientos de miles de usuarios conectados simultáneamente, que al no poder ser asignados a un solo servidor físico suelen ser repartidos a servidores que contienen copias del entorno virtual original (Figura 1), limitando inevitablemente el número de usuarios que pueden interactuar en un mismo entorno.

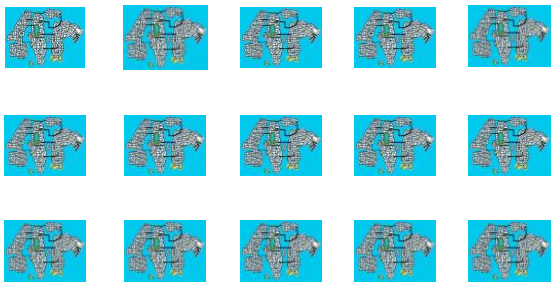

Figura 1a. Ciudad virtual replicada [5].

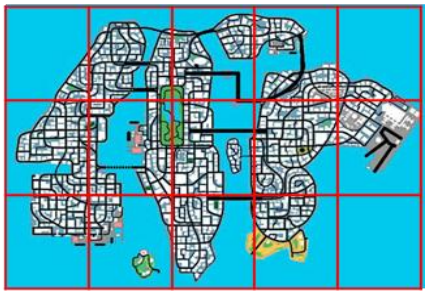

Figura 1b. Ciudad virtual zonificada [5].

Una solución alternativa a la replicación de un entorno virtual es la subdivisión del mismo en zonas espaciales adyacentes (Figura 2), asignando la carga de procesamiento de cada zona a un servidor diferente (Figura 3) y repartiendo la carga de procesamiento de los usuarios a los servidores de acuerdo a la zona en que se encuentren. Así, en vez de repartir los usuarios de una ciudad virtual a réplicas de la misma con usuarios limitados (Figura 1), es posible aumentar el número de usuarios permitidos en una misma instancia de la ciudad con el mismo número de servidores al segmentar la carga de procesamiento por zonas.

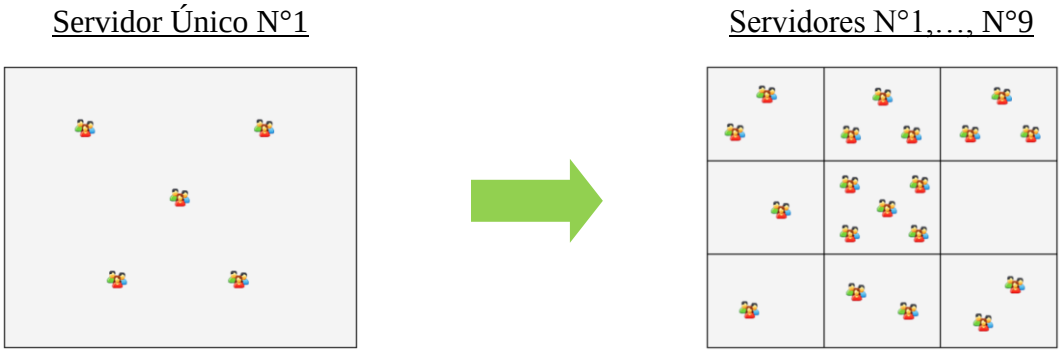

Figura 2. Aumento del número de usuarios permitidos al zonificar un entorno virtual [6].

Naturalmente, el problema de la concurrencia se puede presentar nuevamente en un entorno virtual zonificado si todos los usuarios son ubicados en la misma zona al ingresar al entorno, o si un gran número de usuarios se aglomera en una zona e impone una carga de procesamiento superior a la que puede manejar el servidor encargado de dicha zona.

Si bien este problema podría ser abordado por un trabajo de investigación futuro mediante la zonificación dinámica de un entorno dependiendo de la distribución geográfica de su carga, para el presente trabajo basta con recomendar una asignación aleatoria de usuarios a zonas en el proceso de ingreso al entorno, y sugerir que al diseñar un entorno y determinar el número de zonas en las que será dividido se tengan en cuenta los posibles sitios de mayor interés o popularidad.

Dejando de lado el tema de la concurrencia por ingreso concentrado o sitios de interés en entornos ya zonificados, se debe aclarar que el problema central que se desea abordar en este trabajo de investigación es el cruce de fronteras entre zonas, pues existe evidencia (como la reportada en *Second Life* [7]) de que estos cruces suelen manifestarse como discontinuidades espaciales (cambios abruptos de posición o velocidad) e inconsistencias lógicas (comportamientos inesperados) cuando no son manejados apropiadamente, que deben ser minimizadas para que la creación y operación de este tipo de entornos sea viable.

Dado que dichas discontinuidades e inconsistencias se generan principalmente porque los comandos de un usuario dado no son procesados por ningún servidor durante el tiempo que tarda la transferencia de su carga, resulta necesario utilizar una estrategia de transferencia que minimice dichos tiempos durante los cuales un usuario no es manejado por ningún servidor, junto a una política de cálculos predictivos en el cliente que compense los retardos de red y los haga visualmente imperceptibles.

De este modo, el trabajo de investigación que se expone en este documento busca satisfacer la necesidad de un sistema que garantice que, así como los usuarios de una red de telefonía celular son transferidos de una antena a otra al desplazarse por un entorno real sin sufrir interrupciones audibles en la llamada (Figura 4), los usuarios que se desplacen por un entorno virtual zonificado (Figura 5) sean transferidos de un servidor a otro sin experimentar las discontinuidades e inconsistencias lógicas mencionadas.

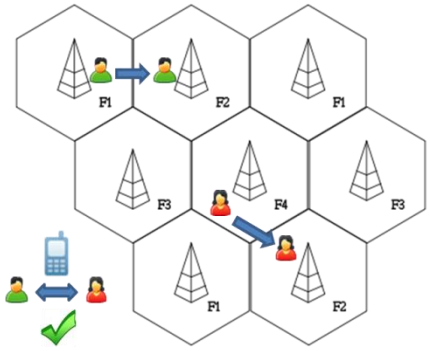

Figura 3a. Desplazamiento de los usuarios en una red de telefonía celular [6][8].

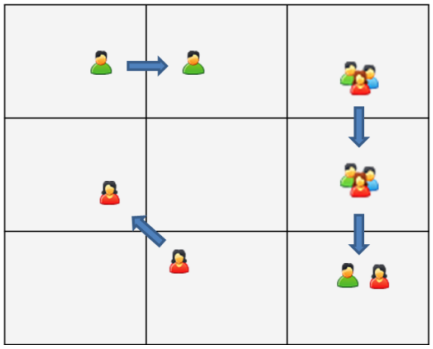

Figura 3b. Desplazamiento de los usuarios (avatars) en un entorno virtual zonificado [6].

Por otro lado, este trabajo busca garantizar que así como los usuarios de una red de telefonía celular pueden realizar llamadas a otros usuarios que se encuentran en su región de cobertura (Figura 6), los usuarios de un entorno virtual zonificado puedan observar todos los elementos que se encuentren en su región de visibilidad, incluso cuando estos estén en lados opuestos de las fronteras y su información deba ser recolectada de otros servidores (Figura 7).

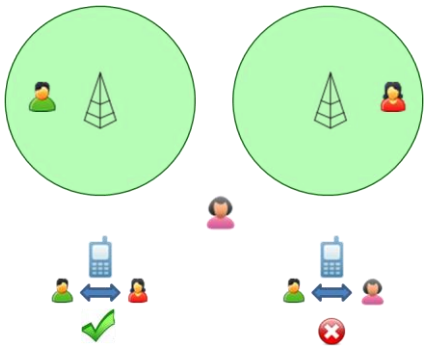

Figura 4a. Cobertura en la telefonía celular [6].

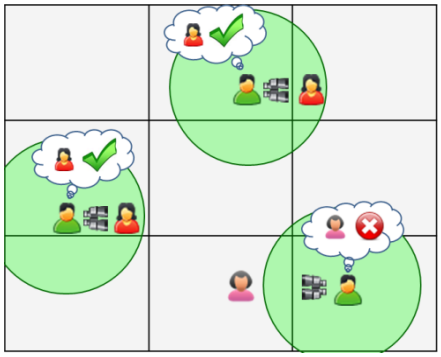

Figura 4b. Visibilidad en entornos zonificados [6]

Finalmente, esta investigación se propone entregar como resultado el diseño de una arquitectura de computación distribuida para la zonificación de entornos virtuales de gran escala y su correspondiente implementación como una plataforma parametrizable para la creación de videojuegos en línea masivos multi-jugador y mundos virtuales en la Web utilizando tecnologías y estándares abiertos.

- TRABAJO RELACIONADO

Diversas publicaciones en el área de los Juegos en Línea Multijugador (MMOG) y los entornos virtuales zonificados han abordado temas como el manejo del área de interés o visibilidad [24], la agrupación de jugadores para el balance de carga [25], la provisión dinámica de recursos [26], esquemas de middleware [27] y arquitecturas para internet [28].

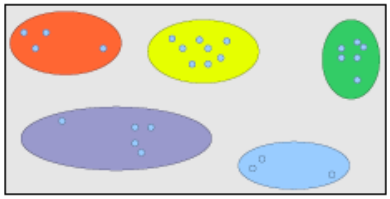

Figura 5. Agrupación de jugadores por proximidad [25]

En [26], por ejemplo, se discuten los enfoques de zonificación (división en zonas), espejos sincronizados (zonas ejecutadas en varios servidores) e instanciación (replicación del entorno) como alternativas para el manejo de la concurrencia (Figura 6), de los cuales se escogió el primer enfoque para el diseño y la implementación de Zone.js.

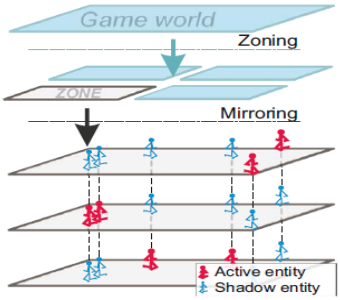

Figura 6. Zonificación, espejo e instanciación de entornos virtuales [26]

Por otro lado, publicaciones como [24] proponen el uso de puntos de check-in y check-out en las fronteras para eliminar los cruces múltiples como se muestra en la Figura 7, los cuales fueron utilizados como punto de partida para desarrollar la estrategia de transferencia de datos y el protocolo para el manejo de los cruces de frontera.`

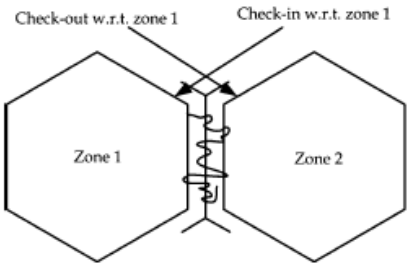

Figura 7. Puntos de check-in y check-out en una frontera [14]

Basado en algunas de las propuestas encontradas en estas publicaciones, en este trabajo se diseñó una arquitectura y se implementó una plataforma específicamente para el contexto de la Web 3D y a partir de tecnologías y estándares abiertos.

2. OBJETIVOS

Teniendo en cuenta la proyección de la Web hacia un entorno interactivo 3D, y los problemas derivados de la zonificación de los entornos virtuales 3D de gran escala que harán parte de esta, se desea diseñar una arquitectura de computación distribuida e implementar una plataforma de software que permita zonificar dichos entornos y manejar las operaciones de transferencia de carga y predicción de cruces de frontera de forma transparente y fluida, utilizando un enfoque abierto y basado en estándares que proporcione herramientas de middleware y la infraestructura necesaria para desarrollar aplicaciones interactivas 3D en la Web divididas en zonas.

2.1 Objetivo General

Diseñar e implementar una arquitectura de computación distribuida para el desarrollo de entornos virtuales zonificados en la Web 3D.

2.2 Objetivos Específicos

- Diseñar una estrategia de transferencia de carga de usuarios entre zonas de un entorno virtual que reduzca las discontinuidades espaciales e inconsistencias lógicas presentadas durante el cruce de fronteras en entornos como *Second Life*.
- Diseñar una estrategia de comunicación entre servidores que les permita compartir oportunamente los datos del entorno cercano a las fronteras entre zonas y construir el entorno visible de los usuarios que se acerquen a las fronteras.
- Desarrollar una plataforma web distribuida que ofrezca una interfaz de programación transparente en cuanto a distribución y migración para la zonificación de videojuegos en línea y mundos virtuales 3D de alta escalabilidad y concurrencia.

3. MARCO TEÓRICO

En esta sección se presentan las bases teóricas necesarias para discutir el diseño y la implementación de la plataforma propuesta sobre la plataforma Node.js, la comunicación cliente-servidor a través de WebSockets, la generación de gráficas 2D y 3D en navegadores web con Canvas y WebGL, y los patrones arquitectónicos a utilizar en el diseño.

3.1 JAVASCRIPT EN EL SERVIDOR CON NODE.JS

Node.js es una plataforma que busca facilitar el desarrollo de aplicaciones de red escalables y de tiempo real como la que se desarrollará en este trabajo [9][10]. A continuación se introducirá el modelo de programación y el sistema de módulos de esta plataforma, junto a los módulos nativos que serán utilizados en este trabajo.

3.1.1 Programación Dirigida por Eventos con Node.js

Tradicionalmente, muchas plataformas y lenguajes de programación han utilizado un enfoque basado en hilos de ejecución y entrada/salida bloqueante para el manejo de la concurrencia en sus aplicaciones. A pesar de sus ventajas en temas como la separación de responsabilidades y el aprovechamiento de múltiples procesadores, dicho enfoque tiene desventajas como el alto consumo de memoria, la carga impuesta por los cambios de contexto y la dificultad en el modelo de programación.

Alternativamente, plataformas como Node.js utilizan un enfoque asíncrono dirigido por eventos en el que los hilos de ejecución son reemplazados por un lazo de eventos (*event loop*) y las operaciones bloqueantes por funciones de devolución de llamada (*callback functions*), el cual reduce el consumo de memoria y elimina la necesidad de los cambios de contexto mientras proporciona un modelo de programación más simple como el empleado por JavaScript en los navegadores web.

Para ejemplificar la diferencia entre el modelo de programación síncrono y el asíncrono dirigido por eventos utilizado por Node.js, los Segmentos de Código 1 y 2 muestran una consulta bloqueante y una no bloqueante a una base de datos, respectivamente. En el caso bloqueante, cuando el programa ejecuta la sentencia `query(…)` envía la consulta a la base de datos y debe esperar hasta que esta responda con el resultado (paralizando todo el programa), por lo que la función `doSomethingElse()` también tendrá que esperar.

```
result = query('SELECT * FROM
                  table');
useResult(result);
doSomethingElse();
```

Código 1. Consulta bloqueante a base de datos.

```
query('SELECT * FROM table',
   function(result)      {
      useResult(result); });
doSomethingElse();
```

Código 2. Consulta no bloqueante a base de datos.

En cambio, en el caso no bloqueante (como Node.js), cuando el programa ejecuta la sentencia `query(…)` envía la consulta a la base de datos pero no se queda esperando la respuesta (i.e. no se bloquea), sino que proporciona una función *callback* para que sea invocada con el parámetro `result` cuando la consulta haya terminado.

Así, en este último caso la función `doSomethingElse()` podrá ser ejecutada casi de inmediato y "en paralelo" mientras el lazo de eventos detecta el evento de 'fin de consulta' para invocar la función *callback* proporcionada como parámetro, lo cual mejorará la utilización del tiempo de procesamiento del sistema al reemplazar las esperas innecesarias por funciones *callback* asociadas a eventos y asignarlo a las partes del programa que ya estén listas para ser ejecutadas.

Node.js propone que el flujo de los programas se dé a través de fases de suscripción, generación, y detección y manejo de eventos. En primer lugar, para que un evento pueda ser manejado, un elemento debe suscribirse a dicho evento y proporcionar una función *callback* para que sea ejecutada cuando el evento ocurra. En segundo lugar, otro elemento debe generar una instancia de dicho evento para que sea insertado en el lazo de eventos y pueda llegar a todos sus oyentes. Finalmente, el lazo de eventos debe invocar las funciones *callback* de todos los oyentes suscritos al evento para que estos puedan actuar en respuesta.

En el Segmento de Código 3, por ejemplo, la sentencia `query(…)` se suscribe al evento 'fin de consulta' de la base de datos, y dicha base de datos debe emitir una instancia de dicho evento con el parámetro `result` cuando termine la consulta. Cuando el lazo de eventos de Node.js detecte el evento 'fin de consulta' asociado a la sentencia `query(…)`, ejecutará la función *callback* proporcionada como parámetro de dicha sentencia para que el resultado de la consulta pueda ser utilizado por la función `useResult(result)`.

En general, la generación de eventos en Node.js se realiza a través de emisores de eventos (*event emitters*) utilizando el método `emit(…)` (Segmento de Código 3) heredado del objeto `EventEmitter`, mientras que la suscripción a eventos se realiza a través de oyentes de eventos (*event listeners*) al especificar el nombre del evento y una función *callback* en el método `on(…)` ó `addListener(…)` (Segmento de Código 4).

```
function callPhone() {
   telephone.emit('ring');
}
function endCall() {
   telephone.emit('hangUp');
}
```

Código 3. Emisión de eventos en Node.js.

```
telephone.on('ring',
  function() { answerPhone(); }
);
telephone.addListener('hangUp',
  function() { hangUpPhone(); }
);
```

Código 4. Suscripción a eventos en Node.js

Finalmente, Node.js posee una valiosa ventaja en el contexto del desarrollo de aplicaciones interactivas 3D en la Web, pues al ser una plataforma para la ejecución de programas escritos en JavaScript en el lado del servidor facilita la compartición de código entre el cliente y el servidor, y resulta bastante familiar para los desarrolladores web por emplear un modelo de programación asíncrona dirigida por eventos similar al utilizado en los navegadores web para la detección de las acciones de los usuarios sobre las páginas.

3.1.2 Creación y Utilización de Módulos en Node.js

Otra de las ventajas de la plataforma Node.js radica en su sistema de módulos, pues este permite desarrollar aplicaciones basadas en módulos como se realizará la implementación de este trabajo. Según su procedencia, estos módulos pueden ser de 3 tipos:

- **Módulos nativos**: Módulos propios de Node.js, como `http` (ver Sección 5.3)
- **Módulos NPM**: Módulos desarrollados para Node.js y publicados e instalados a través de NPM (Node Package Manager), como `socket.io` (ver Sección 3.3).
- **Módulos propios**: Módulos desarrollados para Node.js que no han sido instalados con NPM (típicamente sub-módulos de un módulo a publicar).

Para utilizar un módulo nativo de Node.js simplemente se debe utilizar la función `require` proporcionando como parámetro el nombre del módulo (Segmento de Código 5). Para utilizar un módulo NPM, primero este debe ser instalado utilizando el comando de consola `npm install moduleName` (con NPM previamente [11]), y luego debe ser solicitado como en el Segmento de Código 5. Por último, para el caso de los módulos propios se debe utilizar la función `require` con la ruta relativa (‘./’) de acceso al módulo deseado como se muestra en el Segmento de Código 6.

```
var moduleName =
  require('moduleName')
```

Código 5. Uso de módulos nativos de Node.js o módulos instalados con NPM

```
var moduleName =
require('./path/to/moduleName')
```

Código 6. Uso de módulos o sub-módulos propios no instalados con NPM

La creación de módulos para Node.js es supremamente sencilla, pues solo es necesario crear un archivo JavaScript (extensión ‘.js’) y utilizar el objeto `exports` para exponer las propiedades y funcionalidad deseadas para la interfaz del módulo como se muestra en el Segmento de Código 7.

```
exports.someProperty = someValue;
exports.someMethod = function(someParameter) {...}
exports...

var anotherProperty = anotherValue;
var anotherMethod = function(anotherParameter) {...}
...
```

Código 7. Creación de módulos en Node.js agregando propiedades individualmente

Cabe resaltar que los usuario del módulo del Segmento de Código 7 solo podrán acceder directamente a aquellas propiedades y métodos asignadas al objeto `exports` (`someProperty`, `someMethod`,…), y que todas las que no sean asignadas a este (`anotherProperty`, `anotherMethod`,…) quedarán encapsuladas para ser utilizadas únicamente por el módulo mismo.

Una forma alternativa de exponer la funcionalidad de un módulo de forma compacta se muestra en el Segmento de Código 8, la cual es útil principalmente cuando solo se desea exponer funciones o métodos que serán definidos posteriormente para mejorar la organización del archivo. Sin embargo, se puede apreciar una ligera diferencia en la notación al utilizar `module.exports` en vez de `exports`, la cual no es relevante dado que `exports` es el objeto de paso con el que se poblará finalmente `module.exports` si a este último no se le ha asignado nada como en el Segmento de Código 7.

```
module.exports = {
      someMethod: someMethod;
      ...
};

function someMethod(someParameter){...};
...
```

Código 8. Creación de módulos en Node.js exportando un objeto definido en bloque

Adicionalmente, es posible crear un módulo como una carpeta `moduleName` que contenta un archivo JavaScript nombrado 'index.js' (y otros sub-módulos si es necesario), pues la función `require` de Node.js aplica similarmente para carpetas utilizando rutas relativas. Sin embargo, si se desea utilizar un nombre diferente a 'index.js' es necesario proporcionar un archivo de descripción 'package.json' dentro de la carpeta del módulo para indicarle a Node.js cuál es el archivo principal (Segmento de Código 9).

```
{
  "main": "mainFile.js"
}
```

Código 9. Ejemplo de un archivo de descripción de módulo 'package.json'

Finalmente, es posible crear módulos compatibles tanto con un navegador web como con Node.js sin realizar modificaciones, siempre y cuando no se utilice la función `require` que no es soportada de forma nativa en los navegadores. El Segmento de Código 10 muestra que al rodear el código de un módulo con un *closure* (paréntesis anaranjados) se puede determinar si el módulo debe exponer su funcionalidad a través de la variable `module.exports` (en Node.js) o la variable `this` (en un navegador web).

```
(function(exports) {
var MODULE = exports;

MODULE.someProperty = someValue;
MODULE.someMethod = function() {...};

})('object' === typeof module ? module.exports : this);
```

Código 10. Creación de módulos comunes para compartir código entre el cliente y el servidor

3.1.3 Módulos Nativos de Node.js a Utilizar

- **File System**

El módulo `fs` de Node.js ofrece métodos para el acceso al sistema de archivos del sistema operativo subyacente como `readFile(…)` y `writeFile(…)`, que fueron utilizados en Zone.js para la lectura y escritura de archivos, respectivamente. El Segmento de Código 11 muestra que el método `readFile` permite leer un archivo `file` con una codificación `encoding`, y proporciona los datos del archivo en el parámetro `data` de la función *callback* si no ocurrió ningún error `err` en la operación; posteriormente, muestra que el método `writeFile` permite escribir los datos deseados `data` con codificación `encoding` al archivo `file2`, e informa a la función callback si ocurrió un error `err` o no.

```
var fs = require('fs');
var file = "path/to/file", encoding = "someEncoding";
fs.readFile(file, encoding, function(err, data) { if (!err)
      fs.writeFile(file2, data, encoding, function(err) {
            if(!err) console.log("File written.");
      });
});
```

Código 11. Uso del módulo File System de Node.js para la lectura y escritura de archivos

- **Net**

El módulo `net` de Node.js ofrece métodos para la creación de conexiones TCP como `createServer(…)` y `createConnection(…)`, que fueron utilizados en Zone.js para la creación de servidores TCP y el establecimiento de conexiones entre Servidores de Zona, respectivamente. El Segmento de Código 12 muestra que el método `createServer` permite crear un servidor TCP en el puerto `port` y proporcionar una función *callback* a ejecutar cada vez que un cliente solicite una conexión, a la que proporciona como parámetro el objeto `socket` que representa la conexión con el cliente y permite enviar y recibir mensajes o eventos utilizando los métodos `socket.write(data)` y `socket.on('data')`, respectivamente.

```
var net = require('net');
var server = net.createServer(function(socket) {
      socket.write("Welcome");
      socket.on('data', function(data) {
            doSomethingWith(data);
      });
}).listen(port);
```

Código 12. Uso del módulo Net de Node.js para la creación de un servidor TCP

Por otro lado, el Segmento de Código 13 muestra que el método `createConnection` permite establecer una conexión TCP a un servidor en la dirección IP o dominio `host` y puerto `port`, y retorna un objeto `client` que representa la conexión con el servidor y permite enviar y recibir mensajes o eventos utilizando los métodos `client.write(data)` y `client.on('data')`, respectivamente.

```
var client = net.createConnection(host, port, function() {
      console.log("Connection established.");
});
client.on('data', function(data) {
      client.write(responseTo(data));
});
```

Código 13. Uso del módulo Net de Node.js para la creación de un cliente TCP

- **HTTP**

Finalmente, el módulo `http` de Node.js ofrece métodos para la creación de conexiones HTTP como `createServer(…)` y `get(…)` que fueron utilizados en Zone.js para la creación de servidores HTTP y la solicitud de archivos entre Servidores de Zona, respectivamente. El Segmento de Código 14 muestra que el método `createServer` permite crear un servidor HTTP en el puerto `port` y proporcionar una función *callback* a ejecutar cada vez que un cliente solicite una conexión, a la que proporciona como parámetro los objetos `request` y `response` que representan la conexión con el cliente y permiten obtener información de la solicitud HTTP y enviar la respuesta `data1` y `data2` utilizando los métodos `response.write(data1)` y `response.end(data2)`.

```
var http = require('http');
var server = http.createServer(function(request, response) {
    response.write(data1); response.end(data2);
}).listen(port);
```

Código 14. Uso del módulo HTTP de Node.js para la creación de un servidor HTTP

En correspondencia, el Segmento de Código 15 muestra que el método `get` permite realizar una solicitud HTTP de la ruta `path` a un servidor en la dirección IP o dominio `host` y puerto `port`, y proporcionar una función *callback* a ejecutar cuando se apruebe la solicitud, a la que proporciona como parámetro el objeto `response` que representa la conexión con el servidor y permite recibir la respuesta a la solicitud utilizando los métodos `response.on('data', …)` y `response.on('end', …)`.

```
http.get({hostname: host, port: port, path: path},
function(response) {
      response.on('data', ...); response.on('end', ...);
});
```

Código 15. Uso del módulo HTTP de Node.js para la realización de una solicitud HTTP

3.2 COMUNICACIÓN CLIENTE-SERVIDOR CON WEBSOCKETS

Entornos interactivos 3D en línea como los videojuegos multi-jugador y los mundos virtuales son en esencia aplicaciones de red que siguen el modelo cliente-servidor (Figura 8) para procesar los comandos de los usuarios, sincronizar las acciones y reacciones, y enviar las respuestas y actualizaciones del entorno.

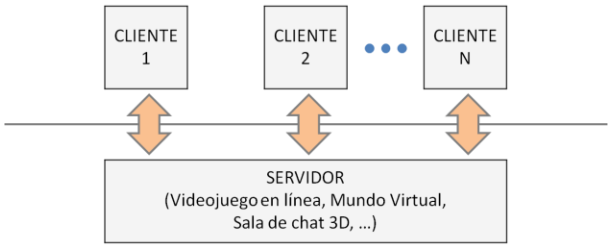

Figura 8. Modelo Cliente-Servidor en aplicaciones interactivas multiusuario

Dado que la calidad y consistencia de estas aplicaciones depende en gran medida de que los comandos y las actualizaciones del entorno se envíen y reciban en tiempo real, no es posible utilizar tecnologías de comunicación basadas en el protocolo HTTP como *long-polling y Comet* para construir entornos interactivos 3D en la Web, pues su sobrecarga y latencia les impiden cumplir los estrictos requerimientos de tiempo de estas simulaciones.

Afortunadamente, una de las tecnologías desarrolladas como parte del reciente estándar HTML5 [1] proporciona una excelente alternativa para la comunicación cliente-servidor en la Web y las actualizaciones dirigidas por el servidor: el protocolo WebSocket [4]. Esta tecnología ofrece herramientas para establecer conexiones TCP bidireccionales entre un navegador web y un servidor a través de una interfaz de programación en JavaScript [12], y permite enviar y recibir mensajes sin la sobrecarga introducida por los encabezados del protocolo HTTP (Figura 9) ni la latencia introducida por las solicitudes adicionales del cliente (Figura 10), superando así al protocolo HTTP con relaciones de 3:1 en latencia y 500:1 en sobrecarga [13].

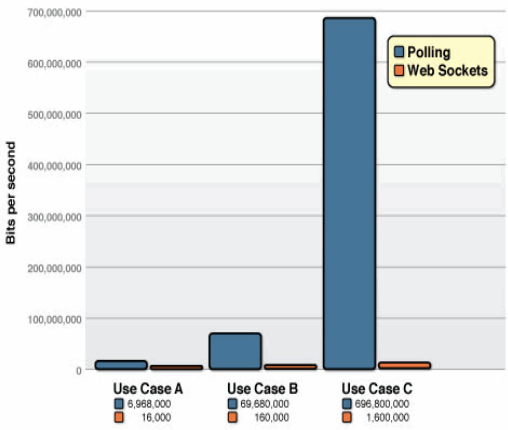

Figura 9. Comparación de la sobrecarga de los protocolos HTTP (*polling*) y WebSocket [13].

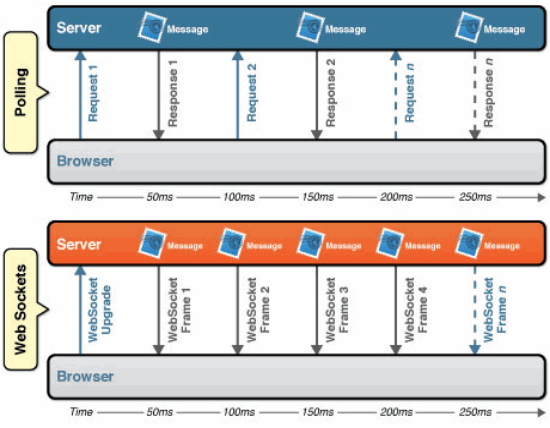

Figura 10. Comparación de la latencia de los protocolos HTTP (*polling*) y WebSocket [13].

Node.js, además de estar orientado al desarrollo de aplicaciones de red escalables y de tiempo real, cuenta con el paquete 'socket.io' [14] que implementa la interfaz (API) de WebSocket de forma uniforme para diferentes navegadores web y proporciona transportes de respaldo como AJAX para navegadores que no soportan el protocolo.

Para manejar conexiones WebSocket en el lado del servidor con Socket.io, los programas deben solicitar el módulo 'socket.io' y utilizar el método `io.listen(portNumber)` para empezar a recibir conexiones en el puerto `portNumber` (Segmento de Código 16). Luego deben proporcionar una función *callback* al método `io.sockets.on('connection', callback)` para que sea ejecutada cada vez que llegue una nueva solicitud de conexión, declarar manejadores de mensajes o eventos con el método `on(…)` y enviar mensajes o emitir eventos utilizando el método `emit(…)`.

```
var io = require('socket.io'); io.listen(portNumber);
io.sockets.on('connection', function(socket) {
  socket.emit('connected', welcomeMessage);
  socket.on('messageType', function(messageContent){
    socket.emit('responseType', responseMessage);
  });
});
```

Código 16. Uso de WebSockets con Socket.io en el lado del servidor.

Por otro lado, para manejar conexiones WebSocket en el lado del cliente, los programas deben solicitar al servidor la librería 'socket.io.js' y conectarse al servidor `targetHost` en el puerto `targetPort` utilizando el método `io.connect("targetHost:targetPort")` (Segmento de Código 17). Posteriormente, deben utilizar los métodos `emit(…)` y `on(…)` para enviar y recibir mensajes o eventos hacia y desde el servidor, respectivamente.

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect("targetHost:targetPort");
  socket.on('connected', function(message) {
    console.log("Welcome message: "+message);
    socket.emit('messageType', messageContent);
  });
  socket.on('responseType', function(responseMessage) {
    console.log("Response: "+ responseMessage);
  });
</script>
```

Código 17. Uso de WebSockets con Socket.io en el lado del cliente.

De este modo, teniendo en cuenta las mejoras significativas en latencia y sobrecarga del protocolo WebSocket y su facilidad uso en Node.js con el módulo 'socket.io', se puede justificar el uso de WebSockets en la implementación de la plataforma de este trabajo.

3.3 GRÁFICAS EN NAVEGADORES WEB CON CANVAS Y WEBGL

Si bien el estándar HTML5 introdujo numerosas mejoras en cuanto a la semántica de las etiquetas de marcado en comparación con la etiqueta general `<div>` (e.g. `<header>`, `<section>`), uno de los mayores aportes se realizó en el área de multimedia con la introducción del elemento `<canvas>` para la generación de gráficas utilizando JavaScript.

Este último aspecto resulta de particular interés para este trabajo de investigación, pues es a través de este elemento que se puede realizar el despliegue gráfico de aplicaciones interactivas en la Web de forma nativa como video juegos en línea y mundos virtuales. Por esta razón, a continuación se presenta una breve discusión sobre el uso del elemento `<canvas>` para dibujar gráficas 2D y 3D en navegadores Web usando JavaScript.

- **Gráficas 2D en el elemento Canvas**

Para hacer uso de la funcionalidad ofrecida por el elemento `<canvas>` de HTML5, primero se debe insertar la etiqueta `<canvas>` en un documento HTML como se muestra en el Segmento de Código 18, indicando el ancho (`width`) y el alto (`height`) deseados.

```
<html>...<body>
  <canvas id="someID" width="1400" height="200"></canvas>
</body></html>
```

Código 18. Inserción del elemento `<canvas>` en un documento HTML

Luego se debe obtener el contexto deseado según las gráficas que se van a dibujar (e.g. '2d') y proceder a utilizar los métodos ofrecidos para el dibujo de líneas (`moveTo(…)`, `lineTo(…)`, `stroke()`), rectángulos (`fillRect(…)`), círculos (`arc(…)`) e imágenes (`drawImage(…)`) como se muestra en el Segmento de Código 19.

```
var canvas = document.getElementById('someID');
var context = canvas.getContext('2d'), ctx = context;
ctx.moveTo(150,50); ctx.lineTo(150,150);
ctx.strokeRect(400,50,100,100);
ctx.moveTo(750,100); ctx.arc(700,100,50,0,Math.PI*2,true);
var image = new Image(); image.src = "user.png";
     image.onload = function() {
     ctx.drawImage(image, 900, 50, 100, 100); }
```

Código 19. Métodos del elemento `<canvas>` para el dibujo de líneas e imágenes

En correspondencia, la Figura 11 ilustra el resultado gráfico del Segmento de Código 19:

Figura 11. Línea, cuadrado, círculo e imagen dibujados en un elemento `<canvas>`

- **Gráficas 3D con WebGL**

Actualmente, las gráficas 3D pueden ser generadas de forma nativa en navegadores web utilizando WebGL [2], una tecnología basada en el acogido estándar OpenGL (versión ES 2.0) que busca facilitar la creación de contenido e interacciones 3D en la web al ofrecer una interfaz de programación de aplicaciones de bajo nivel para la generación de gráficas 3D aceleradas por hardware utilizando JavaScript.

Desde el lanzamiento de la versión estable de esta tecnología en la Conferencia de Desarrolladores de Videojuegos de San Francisco (Marzo 3 de 2011) [15], numerosas bibliotecas y motores 3D en JavaScript como CopperLicht, GLGE, O3D, Three.js y X3DOM han sido desarrolladas para facilitar el uso de WebGL y eliminar la necesidad de utilizar *plug-ins* para generar gráficas 3D en la web.

En particular, la biblioteca *Three.js* [16] ha tenido gran acogida por su baja complejidad, su facilidad para crear elementos 3D a partir de primitivas y su utilización del formato JSON para exportar e importar modelos. Three.js permite crear entornos 3D a partir de cuatro elementos fundamentales: una escena, una cámara, un procesador de gráficas (*renderer*), y un conjunto de objetos 3D. El proceso básico para crear una escena con un cubo giratorio se ilustra en los Segmentos de Código 20 y 21.

```
scene = new THREE.Scene();
camera = new THREE.Camera(…);
scene.add(camera);

cube = new THREE.Mesh(
 new THREE.CubeGeometry( 200,
             200, 200 ),
 new TREE.MeshNormalMaterial()
);
Scene.add(cube);

renderer = new
        THREE.WebGLRenderer();
```

Código 20. Creación de escena 3D con Three.js.

```
function animate() {
render();
requestAnimationFrame(animate);
}

function render() {
 cube.rotation.x += 0.01;
 cube.rotation.y += 0.02;
 cube.rotation.z += 0.03;

 renderer.render(scene,camera);
}
```

Código 21. Animación de escena 3D con Three.js.

Algunos marcos de la animación resultante del Segmento de Código 21 se muestran en la Figura 12:

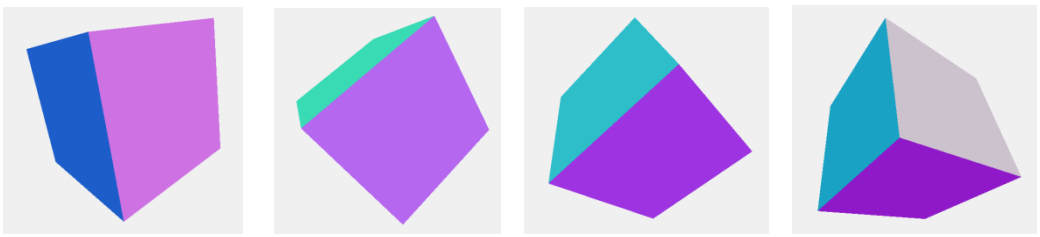

Figura 12. Cubo creado y animado con Three.js

3.4 PATRONES ARQUITECTONICOS DEL DISEÑO

Teniendo en cuenta el modelo de programación asíncrono dirigido por eventos de JavaScript y Node.js, el modelo cliente-servidor utilizado por los entornos interactivos en línea y el modelo de comunicación asíncrona de WebSocket, la arquitectura de computación distribuida del sistema se basará en los patrones arquitectónicos que se acomoden a estas características y separen la aplicación en componentes débilmente acoplados para aumentar su flexibilidad y minimizar las dependencias.

3.4.1 Patrón Multicapas (Multi-tier)

En primer lugar, combinando el modelo cliente-servidor utilizado en la web [17] con la propuesta de subdividir un entorno virtual en zonas y asignarlas a diferentes servidores, la arquitectura general del sistema (Figura 13) se basará en el patrón multicapas para separar los Clientes Web (capa azul), el Servidor Web Coordinador (capa anaranjada) y el Servidor de Aplicación compuesto por los Servidores de Zona (capa canela).

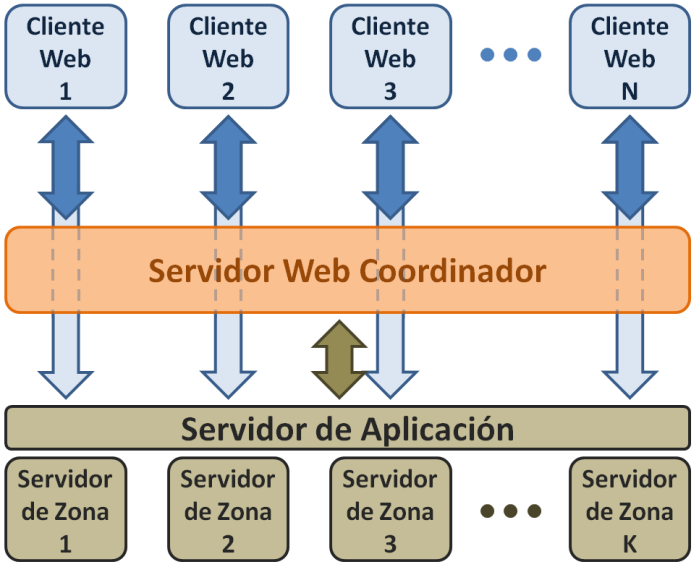

Figura 13. Arquitectura general del sistema

Esta arquitectura propone un proceso de interacción cliente-servidor de dos fases, en el que el Servidor Web Coordinador actúa como una puerta de entrada al Servidor de Aplicación y administra la información necesaria para direccionar a los nuevos usuarios hacia alguno de los Servidores de Zona.

En la primera fase (flechas azul oscuro, Figura 13), los Clientes Web deben realizar una solicitud HTTP al Servidor Web Coordinador para obtener una página web con el elemento `<canvas>` en el que se desplegará el entorno virtual y el código JavaScript necesario para la captura de comandos y cálculos predictivos, la comunicación con los Servidores de Zona del Servidor de Aplicación usando WebSockets y la presentación gráfica del entorno a través del elemento `<canvas>`.

En la segunda fase (flechas azul claro que pasan por detrás del Servidor Web Coordinador, Figura 13), luego de disponer de la página web adecuada para la presentación e interacción con el entorno, los Clientes Web podrán empezar a comunicarse directamente con el Servidor de Aplicación (sin el Servidor Web como intermediario) al contactar a uno de los Servidores de Zona para ingresar al entorno virtual e iniciar la interacción. En esta fase no se realizarán más solicitudes HTTP, pues el envío de comandos y la recepción de actualizaciones de contenido se llevará a cabo utilizando el protocolo WebSocket.

3.4.2 Patrón de Mensajería

El principio de operación de Node.js consiste en utilizar un lazo de eventos para extraer y procesar los elementos de una cola de eventos asíncrona, los cuales han sido depositados a través de emisores de eventos (*event emitters*) y deben ser manejados por las funciones de devolución de llamada (*callback functions*) proporcionadas por sus oyentes.

De forma similar, la interfaz de programación de aplicaciones de WebSocket permite enviar y recibir mensajes de forma asíncrona mediante el depósito y la extracción de elementos de una cola de mensajes que en esencia representan eventos que serán manejados por el motor de JavaScript del navegador web en el lado del cliente o por un programa de comunicaciones en el lado del servidor.

Dado el modelo de programación y las características comunes de estas dos tecnologías de procesamiento y comunicaciones, se puede apreciar que las interacciones soportadas por la arquitectura seguirán el patrón de mensajería, tanto para el manejo de eventos en el lado del servidor con Node.js (Figura 14) como para las comunicaciones entre clientes y servidores con WebSockets (Figura 15).

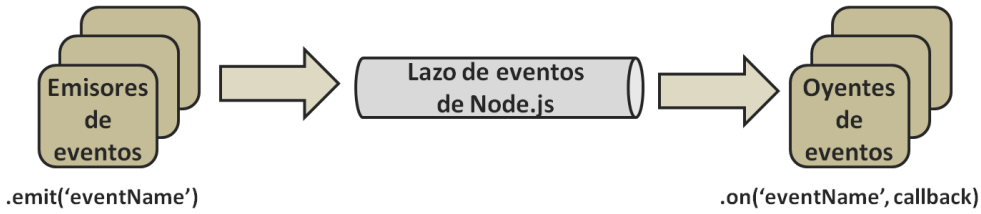

Figura 14. Generación y manejo asíncrono de eventos en Node.js

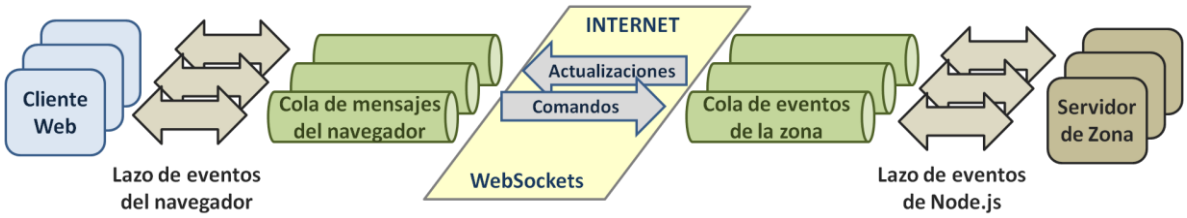

Figura 15. Interacción cliente-servidor con mensajería asíncrona de WebSockets

3.4.3 Patrón de Modelo-Vista-Controlador

Finalmente, resulta conveniente separar las responsabilidades de las capas de la arquitectura con base en el patrón de Modelo-Vista-Controlador, ya que este se puede aplicar fácilmente a las aplicaciones web y a los entornos virtuales interactivos.

Por un lado, en aplicaciones web como las páginas interactivas, este patrón es útil para separar el contenido de las páginas (i.e. modelo), la presentación o apariencia (i.e. vista) y el código JavaScript para el manejo de la interacción (i.e. controlador). Esto facilita la realización de cambios independientes en el diseño de las páginas (e.g. archivos CSS) sin alterar el contenido, cambios en el contenido (e.g. HTML, PHP) sin adaptar el manejo de las interacciones, o cambios en la respuesta a las interacciones (e.g. archivos JavaScript) sin alterar la presentación.

De forma similar, en entornos virtuales interactivos como los videojuegos o los mundos virtuales, este patrón facilita la separación de los objetos que representan al entorno y los usuarios (i.e. modelo), la presentación gráfica de las simulaciones (i.e. vista) y el procesamiento de comandos de los usuarios (i.e. controlador). Así, es posible extender el procesamiento de los comandos (e.g. nuevas acciones) sin alterar la generación de gráficas, mejorar la calidad de las gráficas (e.g. trazado de rayos) sin cambiar el formato de contenido, o modificar las reacciones del entorno (e.g. física y colisiones) sin alterar la interpretación de los comandos.

Esta división de responsabilidades se traduce en un acoplamiento débil entre los componentes y garantiza que la mayoría de modificaciones puedan ser realizadas sin que se propaguen por el resto de la aplicación, facilitando así el mantenimiento del código y la inclusión de versiones especiales de los componentes para diferentes escenarios o configuraciones de visualización (e.g. dispositivos móviles), interacción (e.g. pantallas táctiles) o representación (e.g. formatos de contenido).

Si bien en aplicaciones como videojuegos de un solo jugador existe solo una instancia de cada componente, otras aplicaciones precisan de la existencia de varias instancias del modelo (e.g. entornos independientes), la vista (e.g. varias pantallas) o el controlador (e.g. varios jugadores). Adicionalmente, en aplicaciones interactivas distribuidas como los videojuegos en línea resulta conveniente replicar o repartir parte de los componentes, de modo que se reduzca el impacto de los retardos de red con operaciones predictivas y se mejore así la capacidad de respuesta de la aplicación.

La Figura 16 muestra el patrón de Modelo-Vista-Controlador aplicado a las capas de cliente y aplicación de la arquitectura del sistema. Aunque existe parte de cada componente en ambas capas, las versiones completas del modelo (todo el entorno virtual) y el controlador (interpretación de comandos) se encuentran únicamente en el servidor, mientras que la vista plenamente funcional (generación de gráficas) se encuentra en los clientes. En contraste, el modelo en el cliente solo corresponde a la región visible del usuario y el controlador en el cliente solo ordena cálculo predictivos, mientras que en el servidor la vista está reservada para la supervisión del entorno por parte de los administradores.

El flujo de la interacción inicia cuando el controlador de un cliente contacta al controlador de un servidor de zona y este le retorna el modelo inicial (flecha anaranjada) configurado con los datos de su entorno vecino. Luego, la interacción continúa con envíos de comandos de entrada (flecha verde) y retornos de actualizaciones del modelo (flecha morada) que van acompañadas de predicciones, interpretaciones y actualizaciones en el cliente o el servidor.

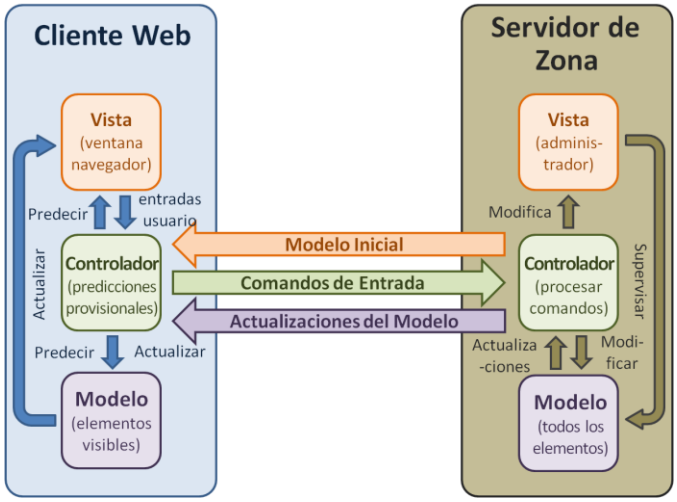

Figura 16. Patrón de Modelo-Vista-Controlador para aplicaciones interactivas distribuidas.

Después de revisar el modelo de programación dirigida por eventos de Node.js junto a su sistema de módulos, las ventajas del protocolo WebSocket y su facilidad de uso con el módulo 'socket.io' de Node.js, el uso del elemento `<canvas>` para la generación de gráficas 2D y 3D con JavaScript, y las características de los patrones arquitectónicos que describirán o constituirán la base del diseño de la plataforma a desarrollar en este trabajo de investigación, se procederá a describir las especificaciones del sistema en la Sección 4.

4. ESPECIFICACIONES

En esta sección se describen las especificaciones de la arquitectura de computación distribuida diseñada y de la plataforma correspondiente implementada en este trabajo de investigación, empezando por los Casos de Uso que deberá soportar del sistema resultante, siguiendo con los Requerimientos Funcionales y No Funcionales, y finalizando con los Diagramas UML para ilustrar las interacciones de los casos de uso más significativos.

4.1 CASOS DE USO

Una de las formas más utilizadas para especificar el comportamiento o la funcionalidad que debe ser ofrecida por un sistema son los Casos de Uso, pues estos permiten listar la secuencia de pasos o interacciones que se deben presentar entre el sistema y un actor dado (e.g. desarrollador, usuario, cliente) para cumplir un objetivo o alcanzar una meta determinada. Aunque no existe una forma única de construir un Caso de Uso, existe un estilo común propuesto por el autor Martin Fowler [18] a partir de 3 elementos:

- **Título:** La meta u objetivo que el Caso de Uso desea lograr.
- **Principal escenario de éxito:** Lista de pasos con sentencias simples de interacción.
- **Extensiones:** Lista de pasos derivados de otras condiciones del escenario principal.

A continuación se presentan los Casos de Uso del sistema a desarrollar, utilizando un párrafo introductorio que luego será desglosado en pasos principales y extensiones.

Pre-condiciones:

- Un Cliente Web se encuentra conectado a Internet y dispone de un navegador web con capacidad HTML5 y soporte de WebGL.
- Un Servidor Web Coordinador está enterado de todos los Servidores de Zona y puede proporcionar la información necesaria para el ingreso a un entorno virtual.
- Un conjunto de Servidores de Zona se encuentran conectados a Internet con una dirección IP fija y contienen una instalación del sistema a desarrollar.

4.1.1 Caso de uso #1: Un desarrollador proporciona los datos de un entorno virtual en un Servidor de Zona.

El desarrollador debe haber construido previamente un entorno virtual mediante la ubicación de elementos 3D sobre una cuadrícula, y proporcionar un archivo de descripción con el arreglo de elementos del modelo y sus respectivos nombres de archivo en la carpeta de elementos, posición (x,y) y escala; también debe indicar el tamaño y las coordenadas del plano sobre el que reposan. Luego, el desarrollador debe proporcionar al Servidor de Zona el archivo mencionado para que este sea agregado a los archivos de configuración de la zona y sus elementos sean cargados en memoria cuando inicie la simulación de la zona.

Suministro de los datos del entorno virtual

Principal Escenario de Éxito:

1) El desarrollador posee un archivo con los datos de un entorno virtual construido a partir de elementos ubicados sobre una cuadrícula.
2) El desarrollador proporciona al servidor el archivo de descripción del entorno virtual.
3) El Servidor de Zona actualiza los archivos de configuración de la zona con los datos del entorno virtual proporcionado.

4.1.2 Caso de uso #2: Un desarrollador inicia un Servidor de Zona

Después de cargar el modelo de un entorno (aunque no sea un requisito), el desarrollador puede iniciar el Servidor de Zona para empezar la simulación del entorno. El Servidor de Zona debe construir el modelo del entorno virtual en memoria utilizando el modelo suministrado previamente por el desarrollador (o un modelo vacío si no lo hizo), iniciar los servidores de soporte necesarios para la comunicación con el Servidor Web Coordinador, otros Servidores de Zona y los Clientes Web, y finalmente iniciar la simulación del entorno virtual.

Simulación del entorno virtual

Principal Escenario de Éxito:

1) El desarrollador inicia el Servidor de Zona.
2) El Servidor de Zona construye el modelo.
3) El Servidor de Zona inicia el controlador (componente MVC).
4) El Servidor de Zona inicia los servidores de soporte.
5) El Servidor de Zona inicia la simulación del modelo.

4.1.3 Caso de uso #3: Un desarrollador solicita la división de un entorno virtual en zonas.

Luego de iniciar el Servidor de Zona, el desarrollador puede solicitar la división del entorno virtual en MxN zonas. El Servidor de Zona debe contactar al Servidor Web Coordinador para que determine si existen otros MxN Servidores de Zona disponibles para hacer parte de la zonificación y para que les solicite que se encarguen de una parte del entorno. Posteriormente, el Servidor Web Coordinador debe clasificar los elementos del entorno virtual en conjuntos disyuntos según su posición en la cuadrícula y el número de zonas especificadas, e indicar a cada Servidor de Zona el conjunto correspondiente junto a su posición en la cuadrícula y sus servidores vecinos.

División del entorno virtual en zonas

Principal Escenario de Éxito:

1) El desarrollador solicita al Servidor de Zona inicial (donde cargó los datos) que divida el entorno virtual en MxN zonas.
2) El Servidor de Zona inicial solicita al Servidor Web Coordinador que determine si existen otros MxN Servidores de Zona disponibles.
3) El Servidor Web Coordinador determina si hay servidores disponibles.
4) El Servidor Web Coordinador invita a los Servidores de Zona disponibles.
5) Los Servidores de Zona invitados responden a la solicitud.
6) El Servidor Web Coordinador clasifica los elementos del entorno en conjuntos disyuntos según su posición en la cuadrícula y el número de zonas.
7) El Servidor Web Coordinador indica a los Servidores de Zona los elementos que deben solicitar al Servidor de Zona inicial.
8) Los Servidores de Zona solicitan los elementos indicados al Servidor de Zona inicial.
9) El Servidor de Zona inicial responde las solicitudes de elementos.

Extensiones:

3.a: No hay suficientes Servidores de Zona disponibles.

.1: El Servidor Web Coordinador rechaza la solicitud de zonificación.

4.1.4 Caso de uso #4: Un usuario solicita la página web de despliegue del entorno

El usuario debe solicitar al Servidor Web Coordinador la página web necesaria para el despliegue e interacción con el entorno. El Servidor Web Coordinador debe retornar una página web con los datos de contacto del Servidor de Zona de destino, el elemento `<canvas>` en el que se desplegará el entorno virtual y una referencia al código JavaScript necesario para el manejo de comandos, la comunicación con los Servidores de Zona usando WebSockets y la presentación gráfica del entorno a través del elemento `<canvas>`. El usuario debe recibir la página de despliegue y prepararse para contactar al Servidor de Zona correspondiente.

Solicitud de la página de despliegue

Principal Escenario de Éxito:

1) El usuario solicita la página de despliegue al Servidor Web Coordinador.
2) El Servidor Web Coordinador retorna una página con los datos de contacto del Servidor de Zona de destino y una referencia al código JavaScript necesario para manejar los comandos, las gráficas y las comunicaciones.
3) El usuario recibe la página de despliegue y se prepara para contactar al servidor de zona.

4.1.5 Caso de uso #5: Un usuario contacta a un servidor de zona para ingresar a su zona del entorno virtual.

Para ingresar a una zona del entorno virtual, el usuario debe realizar una solicitud de conexión a un Servidor de Zona y e identificare para que el servidor sepa a cual avatar corresponde. El Servidor de Zona debe cargar y retornar los datos del avatar del usuario, y luego retornar los elementos abarcados por la región visible del usuario con respecto al punto de partida. El usuario debe recibir los datos del avatar y del entorno circundante, y agregarlos a la escena para desplegarlos en pantalla.

Solicitud de ingreso a una zona

Principal Escenario de Éxito:

1) El usuario contacta al Servidor de Zona de destino.
2) El Servidor de Zona abre una nueva conexión para el usuario.
3) El usuario se identifica para especificar el avatar a ser cargado.
4) El Servidor de Zona retorna los datos del avatar del usuario.
5) El Servidor de Zona retorna los datos del entorno circundante.
6) El usuario recibe los datos y los agrega a la escena.
7) El navegador despliega el entorno en pantalla.

4.1.6 Caso de uso #6: Un usuario se mueve dentro de una misma zona.

Luego de conectarse a un entorno virtual, el usuario puede ingresar comandos de desplazamiento (e.g. flechas) para recorrer el entorno. El Cliente Web debe enviar los comandos al Servidor de Zona asociado y pre-calcular la posición resultante para seguir desplegando la simulación en pantalla sin retardos de red visibles. Si el avatar no se acerca a las fronteras, el Servidor de Zona solo debe procesar los comandos recibidos del Cliente Web y actualizar la posición del avatar en el modelo.

Desplazamiento dentro de una misma zona

Principal Escenario de Éxito:

1) El usuario ingresa un comando de desplazamiento.
2) El Cliente Web envía el comando al servidor.
3) El Cliente Web pre-calcula la posición resultante.
4) El Cliente continúa con el despliegue gráfico de la simulación.
5) El Servidor de Zona recibe y procesa el comando.
6) El Servidor de Zona actualiza la posición del avatar en el modelo.

4.1.7 Caso de uso #7: Un usuario observa elementos ubicados en zonas vecinas.

Cuando un usuario que se desplaza por un entorno virtual se acerca a una frontera entre zonas y su región de visibilidad abarca partes de una o varias zonas vecinas, el Servidor de Zona encargado debe enviar al Cliente Web los datos de del entorno alcanzado por la región de visibilidad del usuario en los servidores vecinos. Aunque la interacción cliente-servidor relacionada con los comandos de desplazamiento es idéntica al caso de uso #6, el Cliente Web ahora debe actualizar los datos del entorno con la información correspondiente a las zonas vecinas y desplegar los nuevos elementos en pantalla.

Visibilidad en las fronteras entre zonas

Principal Escenario de Éxito:

1) El usuario ingresa un comando de desplazamiento.
2) El Cliente Web envía el comando al servidor.
3) El Cliente Web pre-calcula la posición resultante.
4) El Cliente continúa con el despliegue gráfico de la simulación.
5) El Servidor de Zona recibe y procesa el comando.
6) El Servidor de Zona actualiza la posición del avatar en el modelo.
7) El Servidor de Zona percibe que la región de visibilidad excede su frontera.
8) El Servidor de Zona obtiene los datos del entorno alcanzado por la región de visibilidad del usuario de la zona vecina correspondiente
9) El Servidor de Zona retorna los datos de la zona vecina al Cliente Web.
10) El Cliente Web agrega los nuevos elementos al modelo.

4.1.8 Caso de uso #8: Un usuario cruza una frontera entre zonas vecinas.

Cuando un usuario se desplaza por un entorno virtual y se acerca a los límites de una zona para cruzar una frontera, el Servidor de Zona encargado de la zona de partida debe informarle a los Servidores de Zona encargados de las posibles zonas de destino sobre una intención de cruce de frontera, y enviarles anticipadamente los datos del avatar correspondiente para reducir el tiempo de cesión de control y las discontinuidades espaciales e inconsistencias lógicas derivadas de dicho tiempo muerto. Aunque la interacción cliente-servidor relacionada con los comandos es similar al caso de uso #2, el Servidor de Zona de partida deberá informar al Cliente Web sobre el Servidor de Zona de destino cuando su avatar supere los límites de la zona actual (cuando cruce una frontera entre zonas), y deberá ceder el control del avatar al Servidor de Zona de destino correspondiente. Luego, el Cliente Web deberá dejar de comunicarse con el Servidor de Zona de partida y empezar a comunicarse con el Servidor de Zona de llegada.

Cruce de fronteras entre zonas

Principal Escenario de Éxito:

1) El usuario ingresa un comando de desplazamiento.
2) El Cliente Web envía el comando al servidor.
3) El Cliente Web pre-calcula la posición resultante.
4) El Cliente continúa con el despliegue gráfico de la simulación.
5) El Servidor de Zona recibe y procesa el comando.
6) El Servidor de Zona actualiza la posición del avatar en el modelo.
7) El Servidor de Zona detecta una posible intención de cruce de frontera.
8) El Servidor de Zona transfiere anticipadamente los datos del avatar a los posibles Servidores de Zona de destino.
9) El Servidor de Zona detecta que el avatar supera los límites de su zona.
10) El Servidor de Zona de partida cede el control del avatar al Servidor de Zona de destino.
11) El Servidor de Zona de llegada agrega el nuevo avatar a su modelo.
12) El Servidor de Zona de partida informa al Cliente Web sobre el Servidor de Zona de destino.
13) El Cliente Web deja de comunicarse con el Servidor de Zona de partida y empieza a comunicarse con el Servidor de Zona de destino.
14) El Servidor de Zona de destino empieza a recibir los comandos del Cliente Web y a actualizar la información del avatar en su modelo.

4.2 REQUERIMIENTOS FUNCIONALES

Teniendo en cuenta los casos de uso mencionados, y que el objetivo general del sistema a diseñar es manejar las operaciones involucradas en el soporte de entornos virtuales web zonificados, los Requerimientos Funcionales del sistema serán presentados a continuación.

4.2.1 Almacenamiento de datos

El sistema deberá almacenar los datos que representan al entorno virtual y sus usuarios en un formato adecuado para el lenguaje de programación y el motor de gráficas a utilizar.

4.2.2 Zonificación

El sistema deberá ser capaz de dividir un entorno virtual en zonas y alojar cada zona en un servidor diferente.

4.2.3 Simulación

El sistema deberá simular la totalidad del entorno virtual mediante la simulación de cada una de sus zonas en diferentes servidores y el acople de las interacciones en las fronteras entre zonas.

4.2.4 Puerta de acceso

El sistema deberá utilizar un servidor web para proporcionar a cada nuevo usuario una página web inicial que contenga su entorno virtual circundante.

4.2.5 Conexiones

El sistema deberá crear una nueva conexión entre el navegador web de cada nuevo usuario y el servidor de aplicaciones que permita el intercambio de mensajes de forma asíncrona y en tiempo real.

4.2.6 Procesamiento de comandos

El sistema deberá recibir y procesar los comandos de movimiento de los usuarios y enviar los datos relevantes del entorno utilizando comandos ligeros para minimizar la sobrecarga en la comunicación.

4.2.7 Actualizaciones y Visibilidad

El sistema deberá enviar a los usuarios notificaciones y actualizaciones sobre su entorno circundante mediante la recopilación datos de todas las zonas alcanzadas por la región de visibilidad de cada usuario.

4.2.8 Cruce de Fronteras

El sistema deberá manejar los cruces de fronteras que ocurren cuando un usuario se desplaza de una zona a otra mediante la migración de su carga de procesamiento al servidor correspondiente.

4.3 REQUERIMIENTOS NO FUNCIONALES

A su vez, los Requerimientos No Funcionales o cualidades deseadas del sistema se presentan a continuación.

4.3.1 Accesibilidad

Las aplicaciones que utilicen el sistema deberán ser accesibles desde navegadores web con soporte de HTML5 en un computador conectado a Internet.

4.3.2 Uso

El sistema deberá ser utilizable como una biblioteca de funciones o plataforma para el desarrollo de entornos virtuales de gran escala en la web.

4.3.3 Escalabilidad

El sistema deberá ser capaz de aumentar el número de usuarios soportados en un entorno virtual al dividir su carga de procesamiento por zonas y asignarla a diferentes servidores.

4.3.4 Transparencia

El sistema deberá esconder su distribución y sus operaciones de predicción y transferencia de carga involucradas en la operación de un entorno virtual zonificado.

4.3.5 Fluidez

El sistema deberá reducir las discontinuidades espaciales e inconsistencias lógicas presentadas durante los cruces de fronteras de modo que sean imperceptibles para los usuarios.

4.3.6 Interoperabilidad

El sistema deberá estar basado en estándares abiertos y tecnologías web para facilitar su adopción y compatibilidad con otras aplicaciones.

4.3.7 Flexibilidad

El sistema deberá construirse con un enfoque modular y reconfigurable que facilite la realización de cambios y lo haga flexible al ser utilizado por otros desarrolladores.

4.3.8 Desarrollo

El sistema deberá ser programado en JavaScript para facilitar la compartición de código entre el cliente y el servidor.

4.3.9 Integración

El sistema deberá implementado como un modulo o paquete de Node.js y proporcionar interfaces de programación de aplicaciones para el cliente y el servidor.

4.4 DIAGRAMAS UML

El Lenguaje de Modelado Unificado UML (por sus siglas en inglés) es un lenguaje de modelado estándar de propósito general creado por el *Object Management Group* [18]. Utilizando esta convención, a continuación se presentará el diagrama de clases del sistema y los diagramas de secuencia de los casos de uso más significativos.

4.4.1 Diagrama de clases

Dado que los patrones arquitectónicos utilizados separan claramente las responsabilidades del sistema, es posible elaborar su diagrama de clases mediante la asignación de una clase para cada capa de la arquitectura y para cada componente MVC en el lado del cliente y del servidor, junto a algunas clases adicionales para el soporte de aspectos comunes entre ellos (Figura 17).

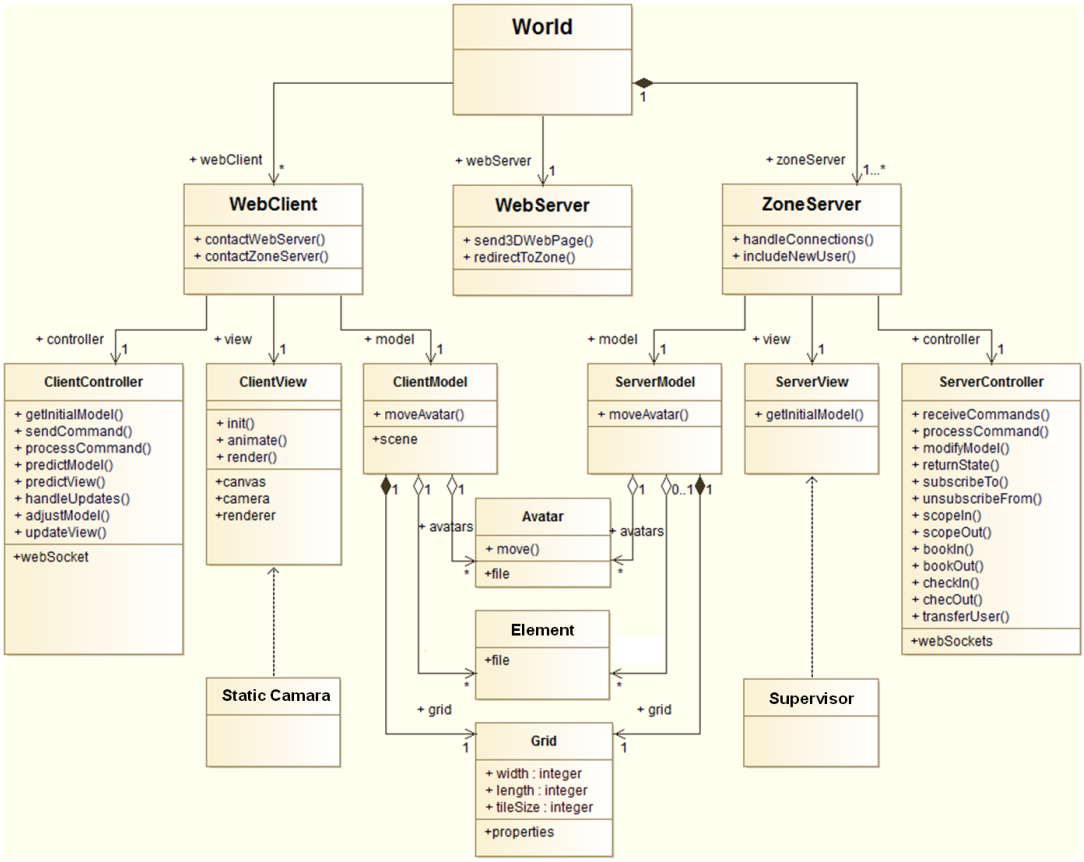

Figura 17. Diagrama de clases de los componentes MVC del sistema.

La clase `World` representa el entorno virtual como una entidad compuesta por un conjunto de servidores de zonas de clase `ZoneServer` que han de ser accedidos a través de un `WebServer` por un conjunto de clientes web de clase `WebClient`. A su vez, las clases `WebClient` y `ZoneServer` poseen una clase para cada componente MVC, donde las clases `ClientController` y `ServerController` manejan las interacciones en el lado del cliente y del servidor, respectivamente.

4.4.2 Diagramas de Secuencia

a) Caso de Uso#6: Un usuario se mueve dentro de una misma zona.

La Figura 18 ilustra el diagrama de secuencia del Caso de Uso #6, en el que se establece que la interacción cliente-servidor se debe presentar a través de los controladores del cliente (`ClientController`) y del servidor (`ServerController`). Cuando un usuario ingresa un comando de desplazamiento (*user input*), el controlador del cliente debe enviarlo al servidor para que calcule la posición resultante mientras que realiza un cálculo predictivo (*predictModel*) para mantener la fluidez de la simulación, y posteriormente debe ajustar el modelo con los datos definitivos del servidor si resulta necesario (*adjustModel*).

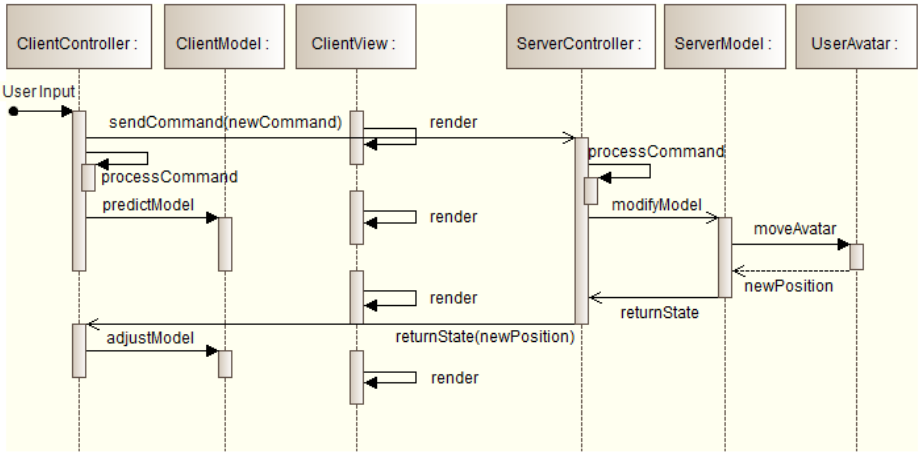

Figura 18. Diagrama de secuencia del caso de uso #6.

b) Caso de Uso #7: Un usuario observa elementos ubicados en zonas vecinas.

La Figura 19 ilustra el diagrama de secuencia del Caso de Uso #7, en el que se establece que para permitir la observación de elementos ubicados en otras zonas de entorno, los controladores de los servidores (`Server1Controller` y `Server2Controller`) deben comunicarse entre sí para compartir los elementos que se encuentren cerca de la frontera y retornarlos al controlador del usuario para que sean agregados dinámicamente a la escena.

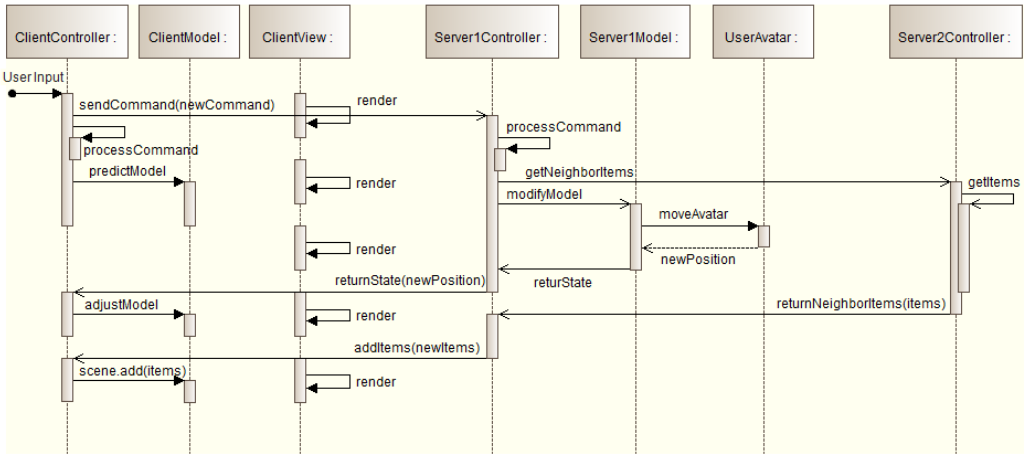

Figura 19. Diagrama de secuencia del caso de uso #7.

c) Caso de Uso #8: Un usuario cruza una frontera entre zonas vecinas.

Finalmente, la Figura 20 ilustra el diagrama de secuencia del Caso de Uso #8, en el que se establece que para manejar los cruces de fronteras entre zonas los controladores de los servidores deben comunicarse entre sí para informarse sobre posible cruces y transferir anticipadamente la carga del usuario (*bookIn(user)* y *bookUser*) o para ceder el control sobre el mismo cuando este llegue a la frontera (*cedeControl(user)* y *takeControl(user)*).

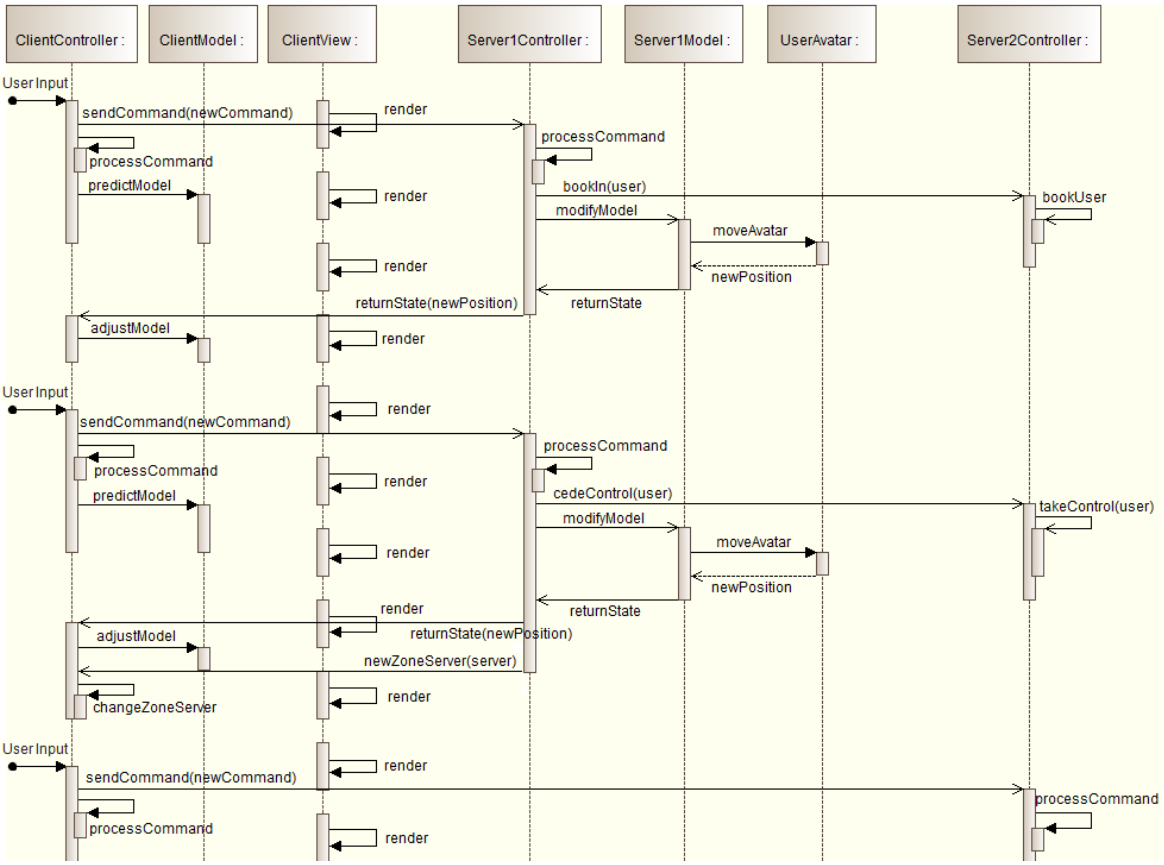

Figura 20. Diagrama de secuencia del caso de uso #8.

Luego de que se cede el control sobre un usuario determinado, se puede apreciar que el controlador del servidor de partida debe informar al controlador del cliente sobre el cambio de zona (*newZoneServer(server)*) para que el cliente se conecte al nuevo servidor (*changeZoneServer*) y empiece a comunicarse con este hasta que ocurra otro cruce de frontera.

Cabe aclarar que todos los casos de uso serán implementados, pero que solo se muestran los diagramas de secuencia de los 3 últimos casos debido a que estos encapsulan el comportamiento global esperado del sistema en operación.

5. DESARROLLOS

Luego de haber discutido las bases teóricas del trabajo en la Sección 3, y enunciado las especificaciones del sistema en la Sección 4, se puede proceder a describir los desarrollos realizados en este trabajo de investigación, empezando por la justificación teórica de las estrategias seleccionadas (i.e. distribución, transferencia anticipada, predicción de cruces), siguiendo con el proceso de diseño de la arquitectura y de dichas estrategias (i.e. anticipación, aplazamiento, zonificación), y finalizando con la implementación de la plataforma como un módulo de Node.js y sus respectivos Casos de Uso.

5.1 JUSTIFICACIÓN TEÓRICA

A continuación se presentará un conjunto de ecuaciones y ejemplos numéricos para justificar la implementación del sistema como un sistema distribuido débilmente acoplado en hardware y no como un cluster, enunciar la ventaja de utilizar una política de transferencia anticipada de carga en los cruces de frontera y plantear la estrategia de predicción de cruces de frontera en términos de un margen de proximidad y los atributos de posición y desplazamiento de los usuarios.

5.1.1 Distribución de Carga por Zonas

Soluciones de infraestructura para entornos virtuales de alta concurrencia como el *cluster* de *EVE Online* [19] utilizan un sistema operativo distribuido para repartir la carga de procesamiento de un entorno virtual entre un conjunto de servidores. Sin embargo, la política de planificación (*scheduling*) de estos sistemas no garantiza que la repartición de carga será óptima con respecto a aspectos como la latencia de las interacciones y la congestión de la red, que son cruciales en aplicaciones interactivas distribuidas.

En contraste, la arquitectura propuesta en este trabajo de investigación está diseñada para tener control sobre la repartición de la carga de procesamiento del entorno, con una política de repartición por zonas que ayuda a minimizar la comunicación entre servidores y la latencia de las interacciones, pues al agrupar a los usuarios de una zona determinada en un mismo servidor, la mayoría del tráfico de sus interacciones (a excepción de las fronterizas) se presentará en memoria principal y no a través de la red, y en consecuencia los tiempos de respuesta serán menores.

La ventaja de utilizar un sistema distribuido con una política de repartición de carga por zonas en vez de un *cluster* con repartición de carga manejada por el sistema operativo puede ser respaldada matemáticamente mediante la comparación del tiempo de interacción entre dos usuarios arbitrarios que se envían mensajes entre sí.

$$T_c \sim \frac{M}{B} + R + D$$

Ecuación 1. Tiempo de transferencia de un mensaje en un *cluster*.

$$T_z \sim \frac{M}{K} + R$$

Ecuación 2. Tiempo de transferencia de un mensaje en una misma zona.

En un *cluster*, dado que dos usuarios cercanos pueden haber sido asignados a servidores diferentes, el tiempo T_c de transferencia de un mensaje entre ellos (Ecuación 1) no solo depende del tamaño *M* del mensaje y el retardo *R* de la atención de la cola de mensajes en la memoria principal, sino de dos variables aleatorias adicionales para el ancho de banda *B* de la red y el retardo *D* derivado a la congestión o el tráfico de la red.

En cambio, en un sistema distribuido con una política de repartición de carga por zonas como en la arquitectura propuesta, el tiempo T_z de transferencia de un mensaje (Ecuación 2) no depende de las características de la red, sino únicamente del retardo *R* de la atención de la cola de mensajes en la memoria principal y del tamaño *M* del mensaje, con una variable *K* que representa ancho de banda de la memoria y *K>>B*.

Las Ecuaciones 3 y 4 muestran un ejemplo numérico de las Ecuaciones 1 y 2, respectivamente, utilizando un tamaño *M* de mensaje de 100 Bytes, un ancho de banda de red *B* de 828 KB/s (valor promedio en San Francisco, Estados Unidos [20]) y un ancho de banda de memoria de 15.2 GB/s (del procesador Intel Core i7-3820 [21]), omitiendo el retardo común *R* y el retardo *D* por ser siempre positivo para resaltar la relación *K>>B*.

$$T_c \sim \frac{M}{B} = \frac{100\ B}{828\ KB/s} = 120\ \mu s$$

Ecuación 3. Ejemplo de tiempo de transferencia de un mensaje en un *cluster*.

$$T_z \sim \frac{M}{K} = \frac{100\ B}{15.2\ GB/s} = 6.57\ ns$$

Ecuación 4. Ejemplo de tiempo de transferencia de un mensaje en una misma zona.

De este modo, y aclarando que la relación $T_z << T_c$ es mucho más significativa en condiciones reales por el retardo de acceso o congestión de red *D*, se puede confirmar que la política de división de la carga de un entorno virtual por zonas asignadas a diferentes servidores contribuye a la reducción de los tiempos de respuesta de un entorno virtual al eliminar las variables aleatorias referentes a la red de las ecuaciones de los tiempos de interacción de usuarios en regiones no fronterizas de una misma zona.

5.1.2 Estrategia de Transferencia Anticipada de Carga

Si bien cada servidor de la capa de aplicación de la arquitectura estará encargado de una zona del entorno virtual y de procesar la carga de los elementos que se encuentren en ella, dichos elementos pueden ser usuarios que se desplazan por el entorno y cuya carga de procesamiento debe ser transferida de un servidor a otro cuando crucen fronteras entre zonas. Dichas transiciones debe ser imperceptibles para los usuarios, de modo que estos recorran el entorno virtual como si fuera manejado por un solo servidor físico y no se percaten de su división en zonas.

Sin embargo, los cruces de fronteras entre zonas pueden ser afectados por discontinuidades espaciales e inconsistencias lógicas que son causadas principalmente porque la transferencia de carga entre servidores no es instantánea, por lo que existe un tiempo muerto durante el cual los comandos de un usuario no son procesados por ningún servidor. Luego, para que dichas discontinuidades e inconsistencias sean imperceptibles para los usuarios, dicho tiempo muerto debe ser minimizado.

La duración del tiempo muerto en las transferencias de carga depende principalmente del ancho de banda de la conexión entre los servidores de zonas, la congestión de la red subyacente y el tamaño de los datos a transferir. En particular, el tiempo *T* de transferencia de carga de un usuario entre servidores de zonas (Ecuación 5) es directamente proporcional al retardo *D* asociado a la congestión de la red y al tamaño de los datos *L* de su avatar (e.g. modelo 3D, código controlador), e inversamente proporcional al ancho de banda *B*.

Dado que *B* y *D* son variables aleatorias que dependen de las características de la red, la única forma de reducir el tiempo de transferencia consiste en reducir el tamaño de los datos a transferir. No obstante, como estos tienen un tamaño fijo, la única forma de "reducir" el tamaño de los datos consiste en utilizar una estrategia anticipativa que transfiera previamente los datos *L* del usuario, de modo que cuando este alcance una frontera entre zonas sus datos ya se encuentren en el servidor de la zona de destino.

De este modo, cuando un usuario se disponga a cruzar la frontera entre la Zona A y la Zona B, por ejemplo, el Servidor A deberá detectar que el usuario tiene la intención de cruzar (e.g. por su aproximación a la frontera), informar al Servidor B sobre un posible cruce de frontera y enviarle los datos de su avatar. Así, cuando realmente ocurra el cruce de frontera, los datos del avatar ya se encontrarán en el Servidor B, y solo restará que el Servidor A ceda el control enviando un mensaje de cesión de control de tamaño *C*<<*L*, lo cual reducirá considerablemente el tiempo efectivo de transferencia a T_c (Ecuación 6).

$$T_L \sim \frac{L}{B} + R + D$$

Ecuación 5. Tiempo de transferencia de la carga de un usuario entre servidores de zonas.

$$T_C \sim \frac{C}{B} + R + D \ll T_L$$

Ecuación 6. Tiempo de cesión de control de un usuario en los cruces de fronteras entre zonas.

Las Ecuaciones 7 y 8 muestran un ejemplo numérico de las Ecuaciones 5 y 6, respectivamente, utilizando un tamaño *L* de los datos de un avatar de 270 KB [22], un tamaño *C* del mensaje de cesión de control de 270 Bytes y un ancho de banda *B* de 828 KB/s, omitiendo los retardos comunes *R* y *D*.

$$T_L \sim \frac{L}{B} = \frac{270\ KB}{828\ KB/s} = 326\ ms$$

Ecuación 7. Ejemplo de tiempo de transferencia de la carga de un usuario entre servidores de zonas.

$$T_C \sim \frac{C}{B} = \frac{270\ B}{828\ KB/s} = 326\ \mu s$$

Ecuación 8. Ejemplo de tiempo de cesión de control de un usuario en los cruces de fronteras entre zonas.

De este modo, se puede confirmar que la política de transferencia anticipada de los datos de un avatar contribuye a la reducción del tiempo muerto durante los cruces de frontera al reducir el tamaño efectivo de los datos a transferir en el momento de la cesión de control a un servidor vecino.

5.1.3 Estrategia de Predicción de Cruces de Frontera

Finalmente, la estrategia de transferencia de carga entre servidores por los cruces de frontera debe contar con que la predicción de un posible cruce se realice con la suficiente anticipación como para que haya tiempo de transferir los datos del usuario al servidor de destino antes de que el usuario llegue a dicha frontera.

Dado que el retardo por congestión de red *D*, el retardo *R* de la cola de mensajes en memoria y el ancho de banda *B* pueden ser estimados, y que el tamaño de los datos *L* es conocido, se puede utilizar la Ecuación 5 para determinar el tiempo mínimo T_{min} necesario para transferir los datos de un avatar desde el servidor actual al posible servidor de destino.

Por otro lado, el tiempo T_A que se tardará un avatar en llegar a una frontera si se aproxima desde una distancia *X* con un ángulo θ y a una velocidad *V* (Figura 21) está dado por la Ecuación 9. Este tiempo T_A debe ser mayor que el tiempo de anticipación mínimo T_{min} necesario para transferir los datos respectivos entre zonas, de modo que estos lleguen al servidor de destino antes que el avatar correspondiente llegue a la frontera.

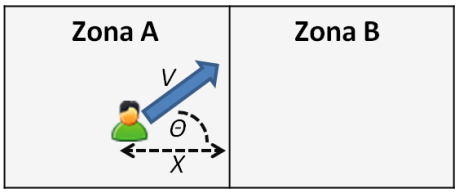

Figura 21. Variables de aproximación de un usuario a una frontera.

$$T_A = \frac{X}{Vcos(\theta)} = \frac{Xsec(\theta)}{V}$$

$$T_A > T_{min}$$

Ecuación 9. Tiempo de anticipación para la transferencia de los datos de un usuario

Luego, combinando las Ecuaciones 9 se puede predecir "cuándo" se debe iniciar la transferencia, o mejor dicho, a qué distancia X_B de la frontera se deben empezar a transferir los datos de un usuario que se aproxima con un ángulo θ y una velocidad *V* (Ecuación 11).

$$X_B = T_A Vcos(\theta) > T_{min} Vcos(\theta)$$

Ecuación 10. Distancia a la que se debe iniciar la transferencia de datos de un usuario.

La Ecuación 11 muestra un ejemplo numérico de la Ecuación 10, utilizando una velocidad *V* de 60 unidades por segundo, un ángulo de aproximación θ de 0 grados, un tiempo T_{min} de 326 ms (avatar de 270 KB transferido a 828 KB/s), con la que se obtiene que la transferencia de datos debe iniciarse a más de 19.56 unidades de la frontera.

$$X_B > T_{min} Vcos(\theta) = (326ms)(60\ x/s)\ (1) = 19.56\ x$$

Ecuación 12. Ejemplo de distancia a la que se debe iniciar la transferencia de datos de un usuario

De este modo, se puede confirmar que la política de transferencia anticipada de los datos de un avatar se puede implementar con una política de predicción de cruces de frontera basada en las variables de aproximación de un avatar a una frontera, pero aclarando que resulta conveniente utilizar márgenes mayores a los calculados para contrarrestar los cambios de velocidad o los retardos de red que se pueden presentar de forma inadvertida.

5.2 DISEÑO DE ZONE.JS

Dado que el objetivo de este trabajo es diseñar e implementar una arquitectura de computación distribuida para la Web 3D, y teniendo en cuenta las tecnologías a utilizar mencionadas en el marco teórico, resulta conveniente organizar los aportes de este trabajo en un módulo de software para la plataforma Node.js.

"Zone.js" ha sido el nombre seleccionado para el módulo entregable de este trabajo de investigación, conformado por la traducción al idioma Inglés de la palabra "Zona" y la extensión correspondiente a los archivos JavaScript. Si bien la implementación detallada de este módulo se abordará en la Sección 5.3, se utilizará esta sección para describir los componentes y estrategias utilizadas para crear esta plataforma.

A continuación se abordarán de forma independiente los componentes de la arquitectura del módulo Zone.js y el sistema general resultante, el diseño de la estrategia de transferencia de datos propuesta para la reducción de las discontinuidades espaciales e inconsistencias lógicas que pueden presentarse durante los cruces de fronteras, y el diseño de la estrategia de zonificación utilizada para dividir un entorno virtual en zonas espaciales manejadas por servidores independientes.

5.2.1 ARQUITECTURA

Retomando la arquitectura general del sistema ilustrada en la Figura 13 (Sección 4.4.1), cabe especificar los 3 tipos de componentes que integrarán el patrón multi-capas del sistema propuesto y listar brevemente sus respectivas responsabilidades:

1) **Servidor de Zona:**
   - Simular la región del entorno virtual que le haya sido designada.
   - Suministrar a los clientes el código necesario para interactuar con el entorno.
   - Procesar los comandos de los usuarios y enviar actualizaciones del entorno.
   - Coordinar los cruces de fronteras con los servidores vecinos.
   - Garantizar la visibilidad en las fronteras.
2) **Servidor Web Coordinador:**
   - Dirigir la zonificación de un entorno virtual.
   - Administrar la información correspondiente a los servidores de zonas.
   - Gestionar el acceso de los usuarios a la capa de aplicación.
   - Proporcionar a los clientes web la página web para el despliegue del entorno.
   - Suministrar la información de contacto necesaria para conectarse a una zona.
3) **Cliente Web:**
   - Contactar al Servidor Web Coordinador para acceder al entorno virtual.
   - Conectarse a uno de los servidores de zona de la capa de aplicación.
   - Enviar los comandos que ingrese el usuario al servidor de zona adecuado.
   - Realizar cálculos predictivos para mantener la fluidez de la simulación.
   - Desplegar gráficamente el entorno virtual en el navegador web del usuario.

- **Arquitectura general de los componentes:**

Además de enunciar las responsabilidades, resulta conveniente describir e ilustrar la arquitectura general de cada uno de los componentes mencionados. En primer lugar, la Figura 22 muestra que un Servidor de Zona es básicamente un servidor Linux con una instalación de Zone.js ejecutándose sobre la plataforma Node.js:

Figura 22. Arquitectura general de un Servidor de Zona

Por otro lado, la Figura 23 muestra que el Servidor Web Coordinador es un servidor Linux con una instalación de la plataforma Node.js, una instalación de un servidor web convencional como XAMPP para suministrar las páginas web necesarias para el acceso a los servidores de zonas, y un sub-módulo de Zone.js (Zone Gridify) encargado de contactar a los servidores de zona disponibles para realizar la zonificación de un entorno virtual:

Figura 23. Arquitectura general del Servidor Web Coordinador

Finalmente, la Figura 24 muestra que un cliente web es cualquier configuración de navegador web con capacidad HTML5 y soporte de WebSockets como Google Chrome:

Figura 24. Arquitectura general de un Cliente Web

- **Arquitectura detallada del sistema:**

Luego, combinando el diagrama general del sistema de la Figura 13 (Sección 3.4.1) y los diagramas generales de los componentes del sistema de las Figuras 22, 23 y 24, es posible ilustrar la arquitectura detallada del sistema como se muestra en la Figura 25:

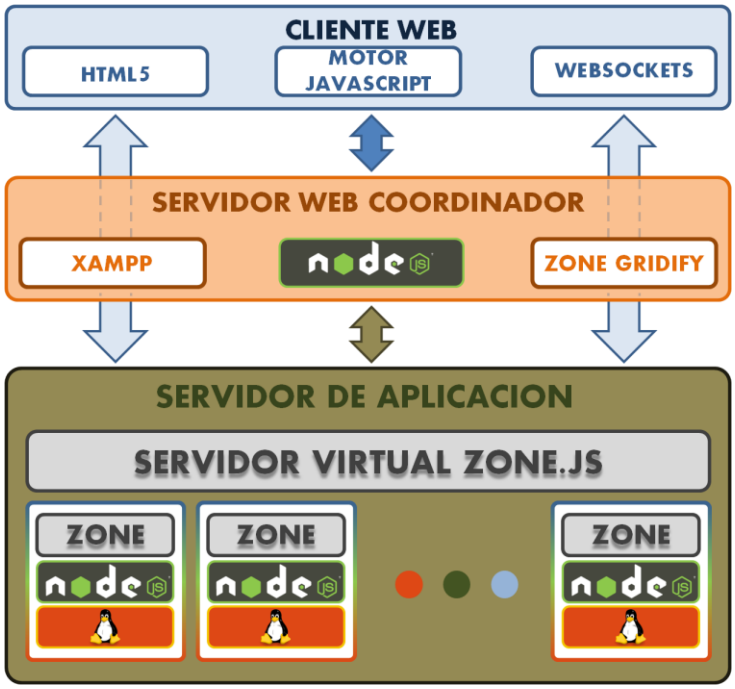

Figura 25. Arquitectura detallada del sistema

En esta figura, la flecha azul oscuro representa la interacción que existe entre la capa azul (Cliente Web) y la capa anaranjada (Servidor Web) para llevar a cabo las operaciones de coordinación de acceso de los usuarios al entorno virtual, la obtención de la página de despliegue y el suministro de la información de contacto del Servidor de Aplicación.

Por otro lado, la flecha café representa la interacción que existe entre el Servidor Web Coordinador y el Servidor de Aplicación para administrar la información de los Servidores de Zona integrantes del sistema, establecer las relaciones entre los servidores que soportarán un entorno virtual determinado y coordinar la zonificación de un entorno virtual.

Las flechas azul claro (que pasan por detrás del Servidor Web Coordinador) representan la interacción que se presenta directamente entre los Clientes Web y los Servidores de Zona de la capa de aplicación (sin el Servidor Web como intermediario) para manejar el procesamiento de comandos, suministrar las actualizaciones pertinentes y ordenar las operaciones necesarias para la comunicación con otros Servidores de Zona en los cruces de fronteras.

Por último, cabe resaltar la importancia del bloque denominado “Servidor Virtual Zone.js”, pues este simboliza la transparencia en cuanto a distribución y migración deseada del sistema, y representa las interacciones que se deben presentar entre los distintos Servidores de Zona para el manejo de los cruces y la visibilidad en las fronteras.

- **Arquitectura del módulo Zone.js:**

Después de revisar la arquitectura detallada del sistema para brindar una visión global de la plataforma, e ilustrar la arquitectura general de cada uno de los 3 tipos de componentes, resulta conveniente desglosar la arquitectura del módulo Zone.js como preámbulo a la descripción detallada de la implementación que se presentará en la Sección 5.3.

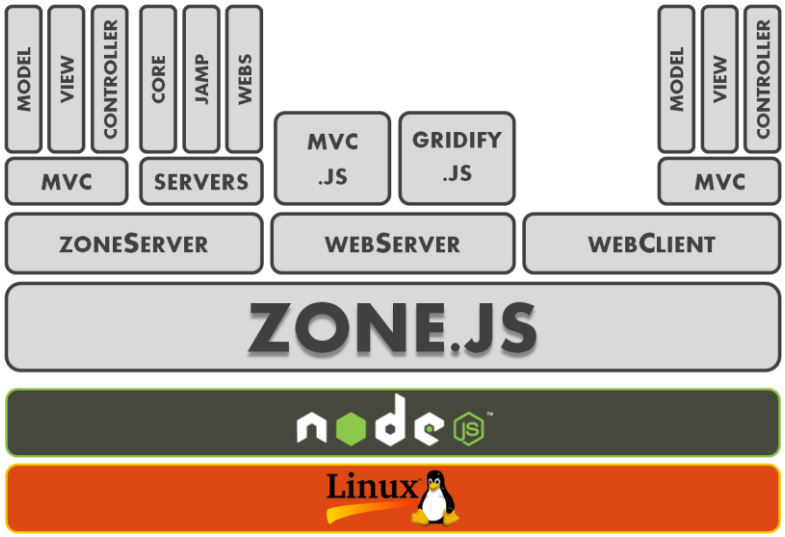

Figura 26. Arquitectura del módulo Zone.js

La Figura 26 ilustra los componentes de la arquitectura del módulo Zone.js agrupados en 3 categorías según su funcionalidad o la capa del sistema a la que pertenecen:

1) **zoneServer**: Componentes encargados de las responsabilidades de un Servidor de Zona.

MVC	*Model*: Almacenar los datos del entorno y ejecutar la simulación.
	View: Proporcionar herramientas de monitoreo para el administrador.
	Controller: Gestionar la interacción con los usuarios, el procesamiento de comandos y el envío de actualizaciones.
SERVERS	*Core*: Procesar las solicitudes del Servidor Web Coordinador.
	JAMP: Coordinar los cruces y la visibilidad en las fronteras.
	WEBS: Administrar las conexiones WebSockets con los clientes.

2) **webServer**: Componentes para ejecutar en el cliente y zonificar el entorno virtual.

MVC	Componente a suministrar a cada Cliente Web para el acceso y la interacción con el entorno virtual.
GRIDIFY	Componente a utilizar por el Servidor Web Coordinador para llevar a cabo la zonificación de un entorno virtual.

3) **webClient**: Componentes utilizados para construir el componente MVC del *webServer*.

MVC	*Model*: Almacenar los datos del entorno cercano y ejecutar la simulación.
	View: Desplegar gráficamente el entorno virtual en el navegador.
	Controller: Realizar cálculos predictivos y comunicarse con los servidores.

5.2.2 ESTRATEGIA DE TRANFERENCIA EN LAS FRONTERAS

El objetivo central de este proyecto de investigación es diseñar una estrategia de transferencia de datos que permita manejar los cruces de frontera presentados durante el desplazamiento de un usuario por un entorno virtual dividido en zonas, y reducir las discontinuidades espaciales e inconsistencias lógicas presentadas durante el salto entre los servidores encargados de las zonas involucradas en dichas fronteras.

Por esto, a continuación se presentará el proceso de análisis y diseño del cual surgió la estrategia de transferencia anticipativa *JAMP*, abreviatura de *JavaScript Asset Migration Protocol*, la cual es utilizada en Zone.js para manejar los cruces de fronteras entre zonas. El protocolo de aplicación JAMP será explicado detalladamente en la Sección 5.3.

- **¿Qué es un cruce de frontera?**

Un cruce de frontera puede definirse como la acción de atravesar la división de dos regiones dadas para pasar del dominio de una al de otra. En la Figura 27, por ejemplo, se muestra un avatar que cruza una frontera entre zonas al desplazarse desde la izquierda hacia la derecha como lo indican los números de secuencia que aparecen sobre el mismo, los cuales representan instantes de tiempo arbitrarios en el proceso de desplazamiento total y están rodeados por un círculo del color de la zona en la que se encuentra el usuario.

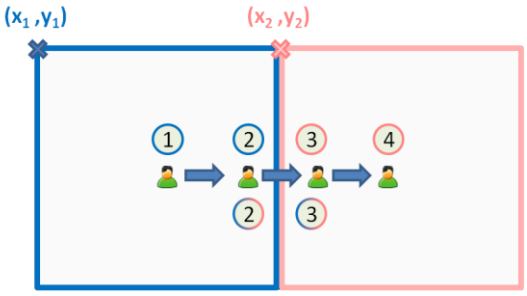

Figura 27. Cruce de frontera de un avatar

Idealmente, cuando el avatar se encuentra en la zona azul, los círculos sobre él deben ser de color azul para representar que el usuario es manejado por esta zona, como sucede con el número 1 y el número 2 de la parte superior; lo mismo aplica para la zona rosada.

Sin embargo, cabe recordar que las discontinuidades espaciales e inconsistencias lógicas que se presentan durante los cruces de fronteras se deben principalmente a que los comandos del usuario en transición no son procesados por ningún servidor mientras se efectúa la transferencia de sus datos y se cede el control sobre sí. Por esto, resulta apropiado considerar algún mecanismo que permita anticipar un cruce de frontera, el cual se traduciría en círculos con coloreado mixto como se ilustra con los números 2 y 3 de la parte inferior.

- **¿Cómo se puede anticipar un cruce de frontera?**

Dado que el objetivo de una plataforma transparente en cuanto a distribución y migración es precisamente que el usuario no se entere de las divisiones subyacentes ni sea participe de la decisión de ser migrado, la única forma de anticipar o intentar predecir un cruce de frontera es a partir de la proximidad de los usuario a las fronteras de una zona.

La Figura 28 muestra que partir de la inserción de una línea punteada vertical de color azul para representar un umbral X_B de proximidad a la frontera, y de la inclusión de dicha información de proximidad en el coloreado mixto del círculo que rodea al número 2, es posible determinar si un avatar debe ser considerado como un candidato para cruce de frontera o no.

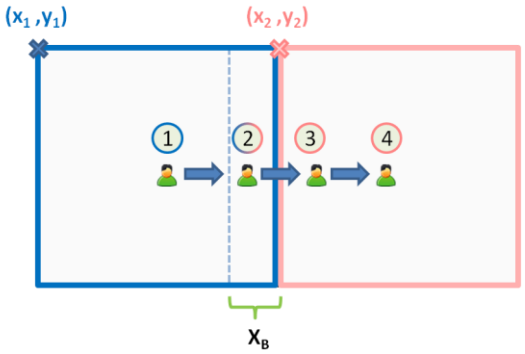

Figura 28. Margen de anticipación al cruce de frontera de un usuario

El umbral X_B puede ser calculado a partir de la Ecuación 11 enunciada en la Sección 5.1.3, tanto de forma estática generalizada utilizando "el peor caso" de velocidad y tamaño de datos de un avatar del sistema, como de forma dinámica e individualizada para cada avatar y cada situación en la que se aproxime a una frontera.

Luego, al utilizar un umbral de proximidad a las fronteras, un Servidor de Zona puede tomar la decisión de informar al Servidor de Zona con el que comparte la frontera sobre un posible cruce, y empezar a transferir de forma anticipada los datos del avatar correspondiente para agilizar la cesión de control en caso de que el usuario efectivamente cruce la frontera.

- **¿Cuándo debe realizarse la cesión de control?**

Bajo la suposición de que los datos del usuario ya se encuentran en el posible servidor de destino, y que el umbral de proximidad calculado fue suficiente para que los datos llegaran al servidor vecino antes que el avatar a la frontera, resta determinar el momento en el que se debe ceder el control del usuario al servidor vecino. La respuesta inmediata es, naturalmente, que la cesión de control se realice cuando el usuario cruce la frontera, pero un ejemplo será suficiente para mostrar porque la solución obvia no es la correcta.

La Figura 29 muestra el recorrido de un avatar que se aproxima perpendicularmente a una frontera, pero que antes de continuar su recorrido perpendicular en la zona vecina realiza un zigzag con el que cruza varias veces la frontera entre las zonas.

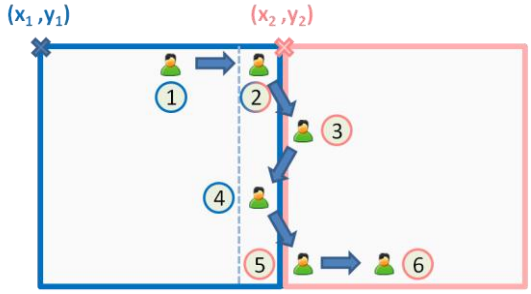

Figura 29. Múltiples cruces de frontera durante el zigzag de un usuario

Como se puede apreciar en el color de los números de secuencia 2-3-4-5 del recorrido del avatar (azul-rosado-azul-rosado), utilizar la frontera como punto de cesión de control genera múltiples operaciones de cesión que van en detrimento de la estrategia anticipativa utilizada con el umbral de proximidad, pues la zona rosada probablemente no alcanzará a reenviar los datos del usuario a la zona azul antes de que el avatar pase del punto 3 al 4, con lo cual muy seguramente se generará el indeseado tiempo muerto en la transferencia del avatar y se producirán las discontinuidades e inconsistencias mencionadas.

- **¿Qué pasa si se utiliza histéresis?**

Una posible solución al problema de las múltiples operaciones de cesión de control en la frontera es utilizar un margen de histéresis, de modo que una vez un avatar haya cruzado la frontera hacia una zona vecina, deba regresarse la misma distancia anticipativa X_B (histéresis simple) para volver a ser registrado en la zona de partida inicial.

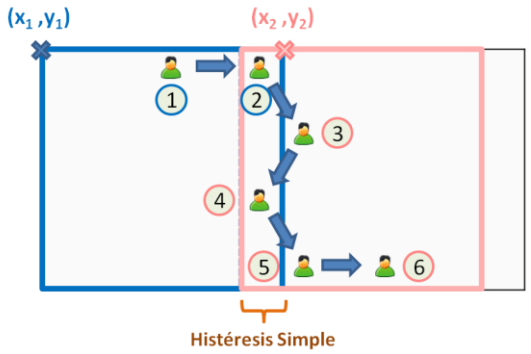

Figura 30. Inconsistencia en las coordenadas de las zonas al utilizar histéresis simple.

Utilizando este enfoque, la Figura 30 muestra que ya no se realizan múltiples operaciones de cesión de control, pues los colores de los números de secuencia no se intercalan. Sin embargo, esta solución genera una inconsistencia en las coordenadas de la zona rosada, pues para que un avatar se encuentre a X_B de la frontera rosada justo cuando cruce la frontera azul, la primera debe encontrarse desplazada la misma distancia hacia la izquierda.

- **¿Qué pasa si se aplaza la cesión de control más allá de la frontera?**

Dado que la histéresis simple no resultó ser suficiente para resolver el problema de cesión de control en los cruces de frontera, resulta lógico pensar que la cesión de control debe realizarse más allá de la frontera, a una distancia que como se puede esperar de la discusión anterior debe ser la misma distancia anticipativa X_B (Figura 31).

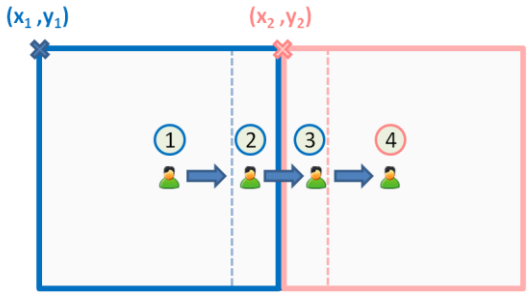

Figura 31. Margen de aplazamiento de la cesión de control en la frontera

De este modo, y aunque parezca que la cesión de control se está realizando tardíamente con respecto al cruce real de la frontera, el Servidor de Zona azul puede ceder el control de un avatar al Servidor de Zona rosada sin preocuparse por que este último le regrese inmediatamente el control del avatar que acaba de cruzar la frontera.

- **¿Resuelve la histéresis doble el problema del zigzag?**

La Figura 32 muestra el mismo recorrido en zigzag introducido en la Figura 29, pero utilizando un margen de histéresis doble para el envío anticipado de los datos y la cesión de control de un avatar determinado.

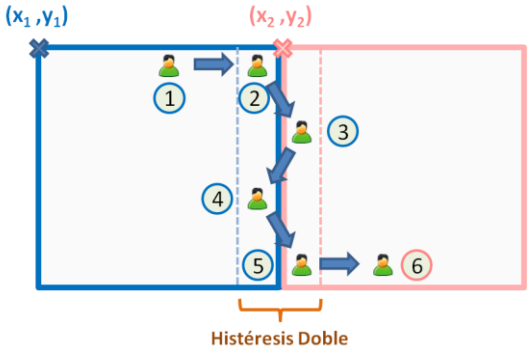

Figura 32. Eliminación de los múltiples cruces de frontera con histéresis doble

Se puede apreciar, según los colores de los números de secuencia, que la cesión de control no se realiza hasta que la posición horizontal del avatar supera la línea punteada rosada de la zona de destino, eliminando así las múltiples operaciones de cesión de control observadas en la Figura 29 y manteniendo la consistencia de las coordenadas de las zonas que se había perdido en la Figura 30.

- **Entonces, ¿dónde o cuándo se deben realizar las reservas y la cesión de control?**

Después de definir los márgenes de anticipación y cesión de control en términos de un margen de histéresis doble, es posible establecer los puntos espaciales o instantes en los que se debe realizar en una zona vecina la reserva o *book-in* de un posible avatar migrante (instantes 2 y 6, Figura 33) y en los que finalmente se debe ceder el control o realizar el *check-in* de dicho avatar en el servidor de destino (instantes 3 y 7, Figura 33):

- *Desplazándose desde la zona azul hacia la rosada*: El Servidor de Zona azul debe detectar la proximidad del avatar a la frontera en el instante 2 e iniciar inmediatamente la transferencia anticipada o *book-in* del avatar en el Servidor de Zona rosado, pero solo debe ceder el control o realizar el *check-in* del avatar cuando este cruce la línea punteada rosada en el instante 3.
- *Desplazándose desde la zona rosada hacia la azul*: El Servidor de Zona rosado debe detectar la proximidad del avatar a la frontera en el instante 6 e iniciar inmediatamente la transferencia anticipada o *book-in* del avatar en el Servidor de Zona azul, pero solo debe ceder el control o realizar el *check-in* del avatar cuando este cruce la línea punteada azul en el instante 7.

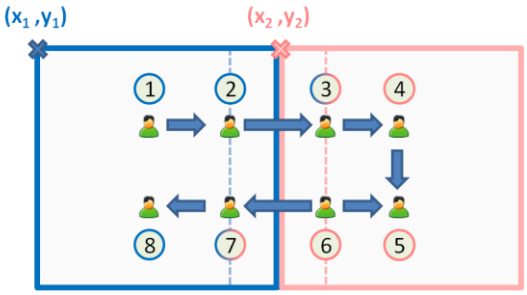

Figura 33. Puntos de book-in (2 y 6) y check-in (3 y 7) durante el cruce de frontera de un usuario

Cabe aclarar, naturalmente, que los usuarios no están obligados a continuar desplazándose después de haber sido reservados en una zona vecina (pues ni siquiera están enterados de dichas operaciones), ni limitados a seguir trayectorias perpendiculares para cruzar fronteras entre zonas (pues ni siquiera están enterados de la subdivisión del entorno).

Por lo tanto, existe la posibilidad de que un usuario se detenga justo antes del punto 3 y se regrese a la zona azul, por lo que no deberá ser cedido a la zona rosada y su reserva deberá ser cancelada si se aleja la distancia o el tiempo suficiente de la frontera. De forma similar, existe la posibilidad de que un usuario se acerque a una de las esquinas de una zona y deba ser reservado en más de una zona, lo cual deberá recibir un tratamiento especial que será discutido en la siguiente sección.

5.2.3 ESTRATEGIA DE ZONIFICACIÓN

Zone.js fue diseñado para ser utilizado como una plataforma para el desarrollo de aplicaciones interactivas multiusuario en la Web que busca proporcionar la abstracción necesaria para que los desarrolladores de estas aplicaciones perciban los entornos virtuales diseñados como la entidad indivisible que se ejemplifica en la capa superior de la Figura 34, a pesar de que las capas subyacentes puedan evidenciar la segmentación del modelo en unidades más pequeñas (zonas) y la distribución de los recursos computacionales utilizados para su simulación (servidores de zonas).

Figura 34. Niveles de abstracción de Zone.js

Para lograr este propósito, resulta imprescindible que la estrategia de transferencia anticipada de datos presentada en la sección anterior vaya acompañada de una estrategia de zonificación que se acople exitosamente a las políticas de reservas y cesión de control estipuladas.

Así, teniendo en cuenta los márgenes de anticipación y cesión de control propuestos como una histéresis doble en la Sección 5.2.2, se presenta a continuación la estrategia de zonificación de la que se obtiene la transparencia en distribución y migración de Zone.js

- **¿Cómo se modela una zona en un entorno virtual manejado con Zone.js?**

Luego de analizar las relaciones e interacciones fronterizas que se presentarán entre dos zonas arbitrarias ubicadas una al lado de la otra, se debe proseguir con la generalización de las relaciones e interacciones de una zona arbitraria, que en una cuadrícula puede llegar a estar rodeada hasta por 8 zonas adyacentes como se muestra en la vista superior de la Figura 35.

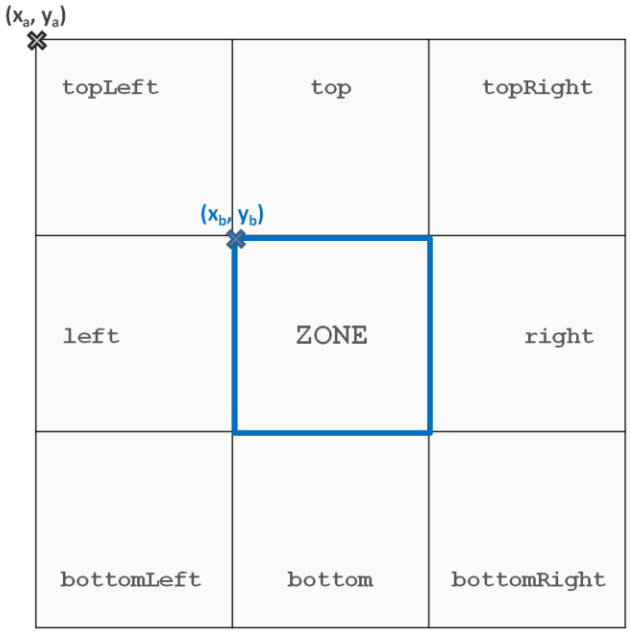

Figura 35. Modelo de vecindario bidimensional para la zonificación en Zone.js

Utilizando el modelo de vecindario bidimensional de la Figura 35, la zona central azul podrá saber si existe una zona vecina en cada una de las 8 posibles posiciones relativas que rodean su casilla en la cuadrícula, y su Servidor de Zona podrá determinar si un avatar debe ser reservado y transferido a otro Servidor de Zona cuando se acerque a una de sus fronteras o si debe detenerse en los límites de su región.

Así, el servidor de la zona azul tendrá vecinos en todas las posiciones relativas posibles (`left`, `right`, `top`, `bottom`, `topLeft`, `topRight`, `bottomLeft`, `bottomRight`); el servidor encargado de la zona ubicada en la casilla `right` tendrá vecinos únicamente en las posiciones relativas `left` (zona azul), `top`, `bottom`, `topLeft`, `bottomLeft`; y el servidor encargado de la zona ubicada en la casilla `bottomLeft` tendrá vecinos únicamente en las posiciones relativas `top`, `right` y `topRight` (zona azul).

Este modelo posibilita la utilización de Zone.js para la zonificación arbitraria de un entorno virtual, sujeto a la limitación inherente de las formas rectangulares utilizadas para representar las zonas, que fueron seleccionadas por la facilidad que ofrecen para representar matemáticamente los límites de una zona y la simplicidad de los cálculos necesarios para determinar la proximidad de cada uno de sus elementos a las fronteras.

- **¿Cómo se ve la política de anticipación y aplazamiento?**

Para que el modelo de vecindario bidimensional propuesto sea compatible con las políticas de anticipación y aplazamiento planteadas en la sección 5.2.2, este debe incluir una versión generalizada de los márgenes de histéresis doble que incorpore la información de proximidad a cada una de las zonas vecinas que rodeen a una zona arbitraria.

La Figura 36 muestra el resultado de combinar los márgenes de histéresis doble de una zona arbitraria con cada una de las 8 zonas que la rodean. Se puede apreciar que los márgenes de anticipación se combinan para formar un cuadrado de líneas punteadas en el interior de la zona, mientras que los márgenes de aplazamiento forman un cuadrado similar pero al exterior de la zona.

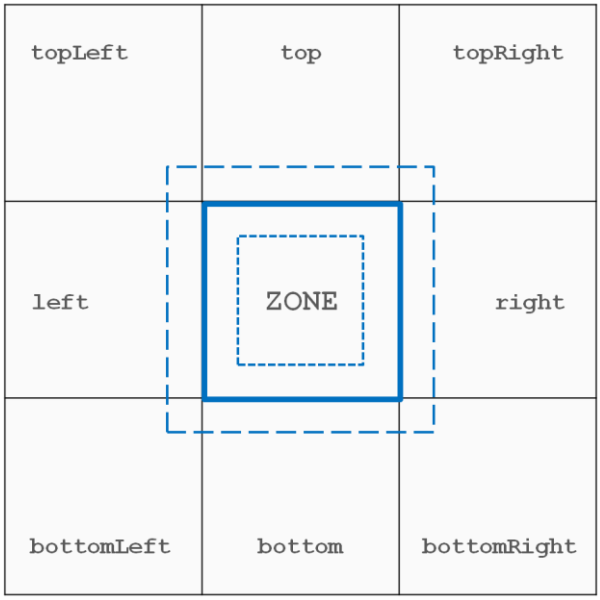

Figura 36. Márgenes de anticipación y aplazamiento de una zona central

De este modo, la versión generalizada del margen de histéresis doble resulta en una especie de anillo rectangular que divide a una zona en 3 regiones o niveles de control según la proximidad bidimensional a las zonas vecinas:

1) La región interna al cuadrado punteado interno (donde se encuentra la palabra `ZONE`), en la que un avatar no deben ser reservados en ningún servidor vecino.
2) La región entre el cuadrado punteado interno y el cuadrado punteado externo, en la que un avatar debe ser reservado en uno o varios servidores vecinos.
3) La región externa al cuadrado punteado externo, en la que el control sobre un avatar dado debe ser cedido al servidor de la zona correspondiente.

- **¿Cómo se ve la superposición de los márgenes de aplazamiento de dos zonas?**

Omitiendo provisionalmente el cuadrado punteado interior resultante de los márgenes de anticipación para continuar con el análisis de la estrategia de zonificación de Zone.js, la Figura 37 agrega los límites y márgenes de aplazamiento de una zona rosada a la derecha de la zona azul de la Figura 36 con el objetivo de ilustrar la superposición de los márgenes de aplazamiento de ambas zonas y el margen de histéresis doble resultante en la frontera.

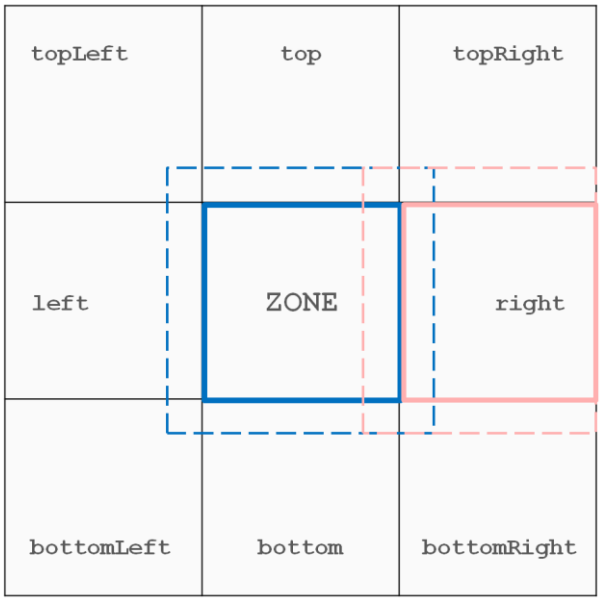

Figura 37. Superposición de los márgenes de aplazamiento de dos zonas

Naturalmente, el margen de aplazamiento no existirá en el lado derecho de la zona rosada pues en esta posición relativa no tiene vecino alguno, de modo que el cuadrado punteado externo se verá truncado por los límites absolutos de la cuadrícula 9x9. Algo similar se podrá apreciar posteriormente con los márgenes de las demás zonas.

- **¿Cómo se ve la superposición de los márgenes de nueve zonas?**

Agregando ahora un cuadrado sólido y uno punteado exterior a cada una de las zonas restantes como se muestra en la Figura 38, se completa la generalización de los márgenes de histéresis doble utilizados en Zone.js para manejar desde los cruces de frontera perpendiculares que involucren 2 zonas hasta los cruces diagonales por las esquinas que involucren 3 o 4 zonas.

Como se mencionó anteriormente, se puede apreciar que los márgenes de las zonas que no tienen vecinos en alguno de sus límites se ven truncados por los límites absolutos de la cuadrícula, y que las diferentes posiciones absolutas en la cuadrícula permiten identificar como deben lucir los márgenes en cada posible ubicación en el vecindario.

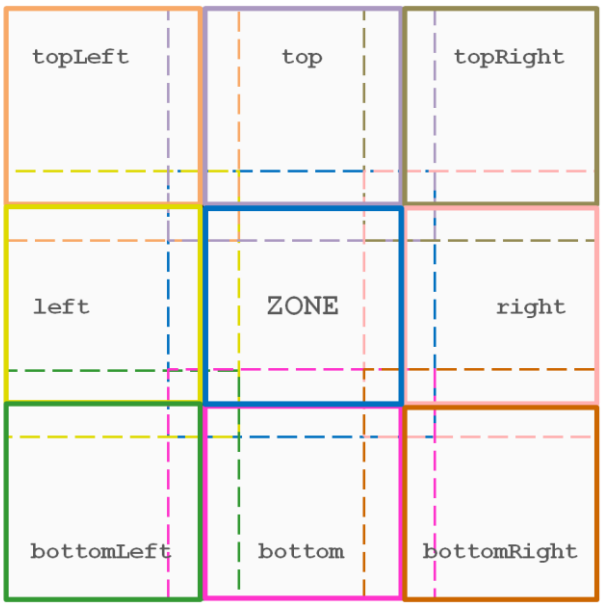

Figura 38. Superposición de los márgenes de aplazamiento de múltiples zonas

- **¿Qué resulta de la superposición de todos estos márgenes?**

El resultado de la superposición de los márgenes de aplazamiento de las zonas que rodean a la zona central azul se muestra en la Figura 39, en la que se puede justificar la omisión provisional del margen de anticipación (cuadrado punteado interno de la Figura 36) al notar que este coincide con el cuadrado punteado interno resultante de la superposición (usando X_B para ambos márgenes), y resaltar la aparición de 25 cuadrantes internos a la línea punteada roja que serán explicados a continuación.

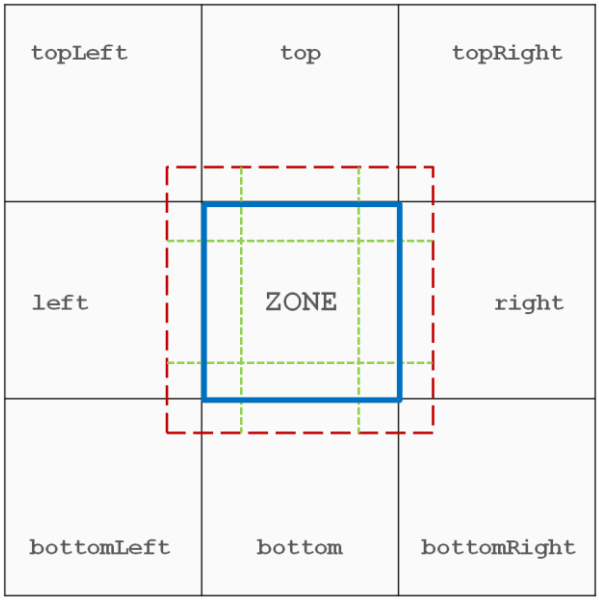

Figura 39. Cuadrantes resultantes de la superposición y los márgenes de anticipación

- **¿Cuáles son los cuadrantes de anticipación o creación de reservas?**

Luego de ilustrar la superposición de los márgenes en la Figura 39, es posible empezar a describir la estrategia de zonificación de Zone.js en términos de cuadrantes. El primer grupo de cuadrantes corresponde a los 9 cuadrantes internos a los límites de una zona (Figura 40), los cuales permiten determinar si un avatar se encuentra dentro de los márgenes de anticipación de la zona (cuadrado punteado interno, Figura 36) o si debe ser reservado en uno o más Servidores de Zona vecinos.

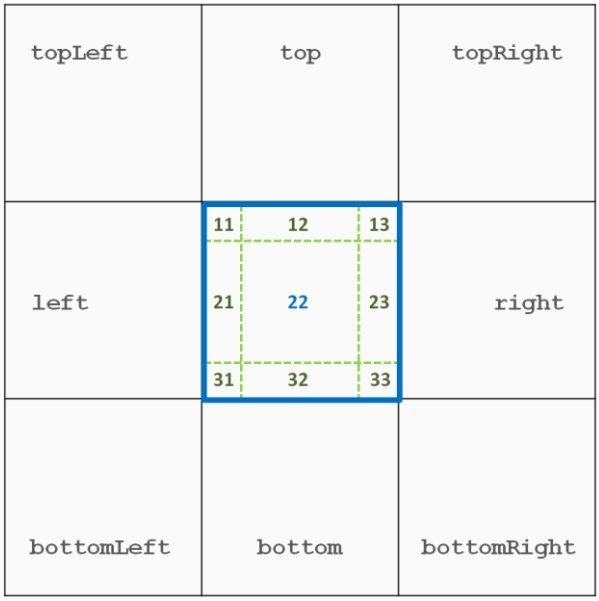

Figura 40. Cuadrantes internos de anticipación

Utilizando la Tabla 1 de equivalencias que se presenta a continuación, un Servidor de Zona podrá determinar a cuál o cuáles Servidores de Zona vecinos deberá transferir anticipadamente los datos de un avatar determinado según del cuadrante al que este ingrese:

Si ingresa al cuadrante:	**Debe realizarse una reserva en los servidores de las zonas vecinas ubicadas en las posiciones relativas:**
11	`left, topLeft, top`
21	`left`
31	`left, bottomLeft, bottom`
32	`bottom`
33	`bottom, bottomRight, right`
23	`right`
13	`right, topRight, top`
12	`top`
22	

Tabla 1. Equivalencia entre los cuadrantes internos y las posiciones relativas a reservar

- **¿Cuáles son los cuadrantes previos a la cesión de control?**

Dado que la política de cesión de control utilizada en Zone.js implica que un Servidor de Zona debe seguir controlando a todo avatar que supere los límites de la zona pero no el margen de aplazamiento, existen trayectoria alternativas para que los usuarios se desplacen a una zona vecina sin pasar por uno de los cuadrantes internos asociados en la Tabla 1.

Un avatar, por ejemplo, puede desplazarse del cuadrante `22` al cuadrante `12` y superar la frontera superior pero no el margen de aplazamiento, y luego desplazarse hacia la izquierda para ingresar a la zona `topLeft`. Sin embargo, como solo fue reservado en la zona `top`, no podrá ser transferido correctamente a la zona deseada, por lo que surge la necesidad de introducir 16 cuadrantes adicionales alrededor de los límites de una zona como se muestra en la Figura 41, y realizar reservas adicionales según las equivalencias de la Tabla 2.

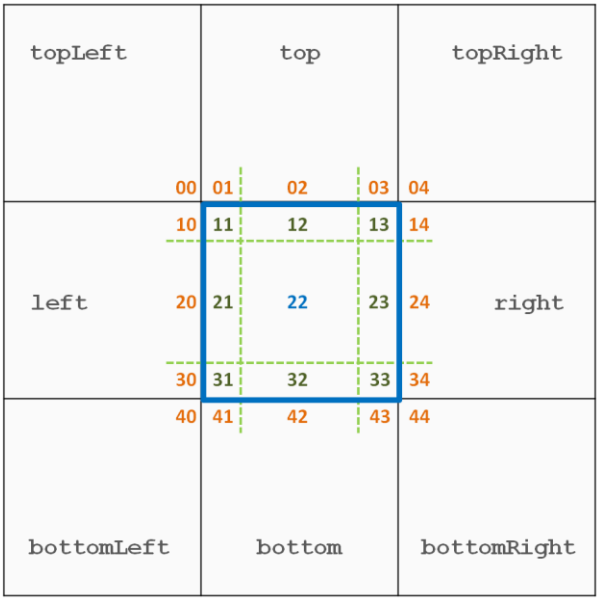

Figura 41. Cuadrantes externos previos a la cesión de control

Si ingresa al cuadrante:	Debe realizarse una reserva adicional en los servidores de las zonas vecinas ubicadas en las posiciones relativas:
01	`topLeft, left`
10	`topLeft, top`
30	`bottomLeft, bottom`
41	`bottomLeft, left`
43	`bottom, bottomRight, right`
34	`right`
14	`right, topRight, top`
03	`top`

Tabla 2. Equivalencias adicionales para las reservas externas a los límites de una zona

- **¿Cuáles son los cuadrantes de cesión de control?**

Finalmente, cuando un avatar supere el margen de aplazamiento indicado por el cuadrado externo punteado azul de la Figura 36 o el cuadrado externo punteado rojo de las Figura 39 y 42, el Servidor de Zona encargado deberá ceder el control del avatar a otro Servidor de Zona dependiendo del cuadrante de color rojo al que haya ingresado al superar el margen de aplazamiento según dictan las equivalencias de la Tabla 3.

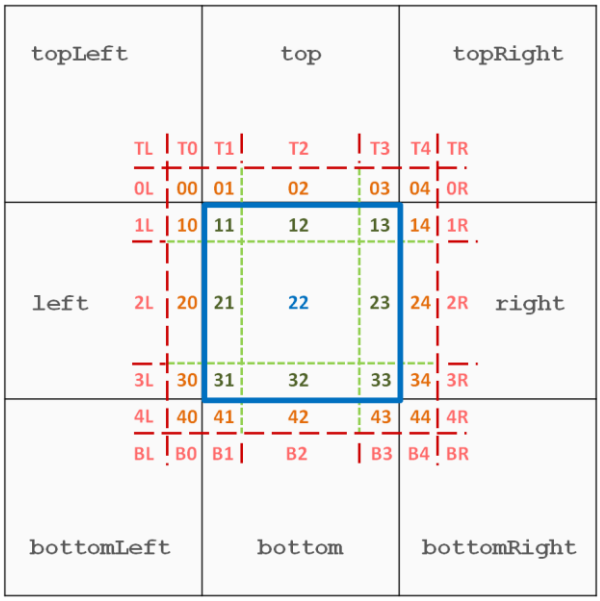

Figura 42. Cuadrantes externos de cesión de control

Así, un avatar podrá desplazarse desde la zona central a la zona de la posición relativa `right` siguiendo diferentes trayectoria, entre las cuales se encuentran:

- `22-23-24-2R` Trayectoria perpendicular básica con reserva única en `right`
- `22-12-13-14-1R` Trayectoria con reservas en `top`, `topRight` y `right`
- `22-32-33-34-3R` Trayectoria con reservas en `bottom`, `bottomRight` y `right`

De forma similar, un avatar podrá desplazarse desde la zona central a la zona de la posición relativa `bottomLeft` siguiendo diferentes trayectorias (aunque todas con reservas en `left`, `bottomLeft` y `bottom`), entre las cuales se encuentran:

- `22-31-40-BL` Trayectoria diagonal básica
- `22-21-20-30-40-B0` Trayectoria centro-izquierda-abajo
- `22-32-42-41-40-4L` Trayectoria centro-abajo-izquierda

Si ingresa al cuadrante:	El avatar debe ser transferido al servidor de la zona vecina ubicada en la posición relativa:
T3	top
T2	top
T1	top
T0	topLeft
TL	topLeft
0L	topLeft
1L	left
2L	left
3L	left
4L	bottomLeft
BL	bottomLeft
B0	bottomLeft
B1	bottom
B2	bottom
B3	bottom
B4	bottomRight
BR	bottomRight
4R	bottomRight
3R	right
2R	right
1R	right
0R	topRight
TR	topRight
T4	topRight

Tabla 3. Equivalencias para la cesión de control de un usuario a un servidor vecino

5.3 IMPLEMENTACIÓN DE ZONE.JS

Luego de abordar el proceso de diseño de Zone.js en la sección 5.2, se puede proceder a exponer la implementación del módulo empezando con su estructura en términos de las carpetas y los archivos derivados de la arquitectura presentada en la Figura 26 (Sección 5.2.1). Posteriormente se abordarán las estrategias de comunicación entre el Servidor Web Coordinador y los Servidores de Zona para la zonificación de un entorno virtual, entre Servidores de Zona utilizando el protocolo de aplicación JAMP para los cruces y la visibilidad en las fronteras, y entre Clientes Web y Servidores de Zona a través de WebSockets. Finalmente, se presentará la implementación de los Casos de Uso.

5.3.1 MÓDULO PROPUESTO

La Figura 43 muestra la estructura general del módulo Zone.js diseñado para la plataforma Node.js en la Sección 5.2. Se puede apreciar la existencia del archivo de descripción de paquete 'package.json' que permite denominar 'zone.js' al archivo principal del módulo, la presencia de una carpeta para cada una de las 3 categorías enunciadas en la Figura 26 ('zoneServer', 'webServer', 'webClient'), la carpeta 'common' en la que se almacenará el código a compartir entre clientes y servidores, y la carpeta 'node_modules' en la que se encontrará el módulo WebSockets no nativo de Node.js 'socket.io'.

Figura 43. Estructura general del módulo Zone.js

A su vez, la Figura 44 ilustra la estructura detallada del módulo Zone.js, en la que se especifican los archivos que integran cada una de las carpetas o sub-módulos del sistema (a excepción de los archivos 'package.json'), y cuyo código fue desarrollado en su totalidad en este trabajo (a excepción del módulo 'socket.io' de la carpeta 'node_modules').

En esta se listan los sub-módulos MVC del servidor ('serverModel', 'serverView', 'serverController') y del cliente ('clientModel', 'clientView', 'clientController'), los servidores de soporte introducidos en la arquitectura del módulo en la Figura 26 ('core.js', jamp.js', 'webs.js'), el archivo 'mvc.js' a enviar a los clientes para la interacción con el entorno virtual (módulo 'webServer'), el archivo 'loop.js' que será uno de los archivos a compartir entre la capa de Clientes Web y el Servidor de Aplicación, el archivo 'gridify.js' que será utilizado por el Servidor Web Coordinador para zonificar un entorno virtual, y el archivo 'grid.js' que representará la información espacial de cada zona (e.g. ancho).

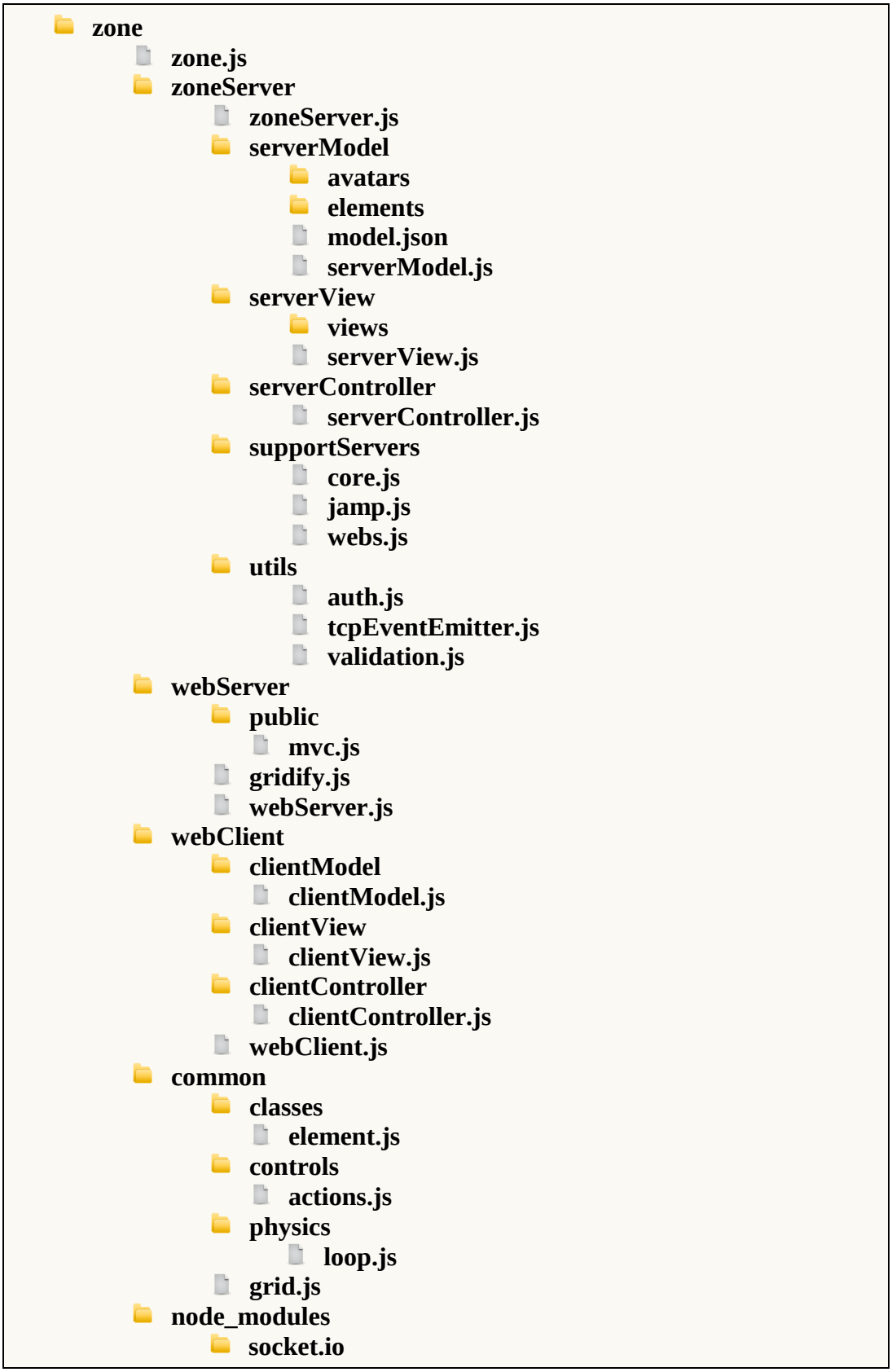

Figura 44. Estructura detallada del módulo Zone.js

- **zone.js**

Para abordar la descripción detallada del módulo Zone.js resulta conveniente empezar con el archivo principal 'zone.js', el cual a pesar de su calificativo de 'principal' solo participa en la iniciación del módulo, pues fue diseñado para actuar como una interfaz de validación y arranque del sub-módulo `zoneServer`, en el que se encuentra realmente toda la lógica de la Capa de Aplicación del sistema.

El Segmento de Código 22 muestra algunas líneas del archivo 'zone.js', en las que se realiza la definición de la interfaz del Zone.js a través del objeto `module.exports`, se solicita el sub-módulo `zoneServer`, y se incluye una utilidad de validación que será empleada para verificar los parámetros suministrados por el usuario del módulo al ejecutar una de las 3 operaciones ofrecidas por esta interfaz: `start(…)` para iniciar la simulación, `loadModel(…)` para cargar el modelo de un entorno virtual y `splitWorld(…)` para dividir un entorno en zonas.

```
var zoneServer = require('./zoneServer');
var validate = require('./zoneServer/utils/validation');

module.exports = {
      start: start,
      loadModel: loadModel,
      splitWorld: splitWorld
};
...
```

Código 22. Implementación de 'zone.js', el archivo principal del módulo Zone.js

- **zoneServer.js**

Como se mencionó anteriormente, el sub-módulo `zoneServer` se caracteriza por contener la lógica de la Capa de Aplicación del sistema, lo cual se puede apreciar en la inclusión de los sub-módulos de los servidores de soporte (`core`, `jamp`, `webs`) y los sub-módulos de los componentes MVC (`serverModel`, `serverView`, `serverController`) en el Segmento de Código 23.

Adicionalmente, se puede notar la inclusión de los módulos relacionados con responsabilidades de las otras dos capas de la arquitectura: el sub-módulo `webServer` que representa una parte del Servidor Web Coordinador (archivo 'gridify.js' y suministro del archivo 'mvc.js'), y el sub-módulo `webClient` que será el encargado de construir el archivo 'mvc.js' a enviar a los Clientes Web.

Cabe resaltar la homonimia de los métodos ofrecidos por el módulo del archivo 'zone.js' y el módulo `zoneServer` (para recordar la función de interfaz del primero), y el uso de la sentencia `module.ZONE = {…}` que permitirá que el objeto `ZONE` sea visible para los módulos hijos de `zoneServer`.

```
var ZONE = module.ZONE = {…}

ZONE.servers = {
      core: require('./supportServers/core'),
      jamp: require('./supportServers/jamp'),
      webs: require('./supportServers/webs')
};
ZONE.webClient = require('../webClient');
ZONE.webServer = require('../webServer');
ZONE.mvc = {
      model: require('./serverModel'),
      view: require('./serverView'),
      controller: require('./serverController')
};

module.exports = {
      start: start,
      loadModel: loadModel,
      splitWorld: splitWorld
};
...
```

Código 23. Implementación de 'zoneServer.js', la Capa de Aplicación de Zone.js

- **webServer.js**

La responsabilidad principal del módulo `webServer` consiste en suministrar el archivo 'mvc.js' que será construido por el módulo `webClient`, para lo cual ofrece un único método `start(…)` que permite iniciar un servidor web utilizando el módulo nativo de Node.js `http`. En el Segmento de Código 24 se pueden apreciar las propiedades del módulo agrupadas en el objeto **`WEB`**, y destacar el uso de un *closure* para suministrar como parámetro el objeto `ZONE` del módulo padre `zoneServer` (definido en la sentencia `module.ZONE = {…}` del Segmento de Código 23).

```
(function(ZONE) {
var WEB = {
      port: 0,
      server: {}
};

module.exports = {
      start: start
};
...
})(module.parent.ZONE);
```

Código 24. Implementación de 'webServer.js', la Capa del Servidor Web Coordinador de Zone.js

- **webClient.js**

En correspondencia, la responsabilidad principal del módulo `webClient` consiste en construir el archivo 'mvc.js' (método `generateMVC`) que será suministrado por el módulo `webServer` a los Clientes Web para la interacción con el entorno virtual, el cual básicamente consistirá en la concatenación del contenido de los archivos listados en el arreglo `files` del objeto **`WEBC`** del Segmento de Código 25.

```
var WEBC = {
      mvc: {},
      files: ['/../common/classes/element.js',
                  '/../common/physics/loop.js',

                  '/clientModel/clientModel.js',
                  '/clientView/clientView.js',
                  '/clientController/clientController.js' ],
      content: ""
};

module.exports = {
      generateMVC: generateMVC
};
...
```

Código 25. Implementación de 'webClient.js', la Capa de Clientes Web de Zone.js

Cabe resaltar la presencia de los archivos comunes 'element.js' y 'loop.js', pues el primero es una clase que representa un elemento arbitrario del entorno virtual como un avatar, mientras que el segundo contiene las reglas de lógica o física que serán utilizadas para determinar las reacciones del entorno y sus elementos a los comandos de los usuarios, tanto en el servidor de forma autoritaria como en el cliente de forma predictiva.

El Segmento de Código 26 muestra la implementación de la clase 'element.js', que consiste en un constructor de la clase `Element` seguido de dos funciones prototipo para la aplicación y remoción de fuerzas sobre el elemento según los comandos ingresados por el usuario o el entorno.

La clase `Element` contiene propiedades relacionadas con el tamaño y la ubicación de un elemento (`scale`, `position`), el rango de visibilidad y las zonas alcanzadas por este (`visibleRange`, `inRange`), los cuadrantes en que se encuentra (`quadrant`, `visibleQuadrant`), los servidores en los que tiene reservas (`bookedIn`), la ubicación y el tipo de sus datos (`src`, `data`, `dataType`), las teclas que su usuario tiene presionadas y las fuerzas resultantes (`keys`, `forces`), entre otros.

```
(function(exports) {
/*CONSTRUCTOR*/
var Element = function(element) {
      /*IF NODE*/
      if(typeof element.src == 'undefined') return null;
      /*ENDIF NODE*/

      this.serial = 0;
      this.jamped = <boolean>;

      this.name = "";
      this.type = "";
      this.src = element.src;
      this.scale = <{x,y,z}>;
      this.position = <{x,y,z}>;
      this.visibleRange = <number>;
      this.inRange = [];
      this.quadrant = <{x,y}>;
      this.visibleQuadrant = <{x,y}>;
      this.bookedIn = [];
      this.dataType = "";
      this.data = {};
      this.keys = [];
      this.forces = [];
      this.step = <number>;
};

/*PROTOTYPES*/
Element.prototype.applyForce =
  function(direction, magnitude) {
      this.forces[direction] = magnitude;
  }
Element.prototype.removeForce = function(direction) {
      this.forces[direction] = 0;
}

exports.Element = Element;
})('object' === typeof module ? module.exports : this);
```

Código 26. Clase 'element.js', la representación general de los elementos en Zone.js

Finalmente, se puede notar la utilización de un *closure* para que la clase pueda ser exportada (`exports.Element = Element`) e instanciada tanto en el cliente como en el servidor, y la presencia de las líneas de comentario "`/*IF NODE*/`" y "`/*ENDIF NODE*/`" utilizadas por `webClient` durante el pre-procesamiento de los archivos comunes para filtrar las líneas de código que solo deben o pueden ser ejecutadas en Node.js.

5.3.2 MANEJO DEL PROCESO DE ZONIFICACIÓN

Una vez el módulo `zoneServer` reciba la invocación del método `start(…)` con los parámetros apropiados, incluso sin haber recibido previamente la invocación de los métodos `loadModel(…)` y `splitWorld(…)`, procederá inmediatamente a iniciar la ejecución de los componentes MVC `serverModel` y `serverController`, y de los servidores de soporte `core`, `jamp` y `webs` invocando el método `createServer(…)`. El primer servidor a iniciar será `core`, pues este constituye el punto de acceso del Servidor Web Coordinador a cada uno de los Servidores de Zona para llevar a cabo la zonificación de un entorno virtual dado.

- **core.js**

El Segmento de Código 27 muestra algunas líneas de la implementación del módulo `core` en el archivo 'core.js', donde se pueden distinguir las propiedades del módulo y la interfaz expuesta para la creación del servidor, junto a la inclusión de un sub-módulo propio `tcpEventEmitter` que fue desarrollado para proporcionar a las conexiones TCP del módulo `net` una interfaz con soporte de temas y objetos similar a la ofrecida por `socket.io` con los métodos `on('topic', object)` y `emit('topic', object)`, pues originalmente la interfaz de `net` solo ofrece soporte de mensajes *string* sin tema con los métodos `on('data', string)` y `write(string)`.

```
(function(ZONE) {
var tcpEventEmitter = require('../utils/tcpEventEmitter');
var CORE = {
      ports: {…},
      tcpServer: {}
};

module.exports = {
      createServer: createServer
};
...
})(module.parent.ZONE);
```

Código 27. Implementación de 'core.js', el servidor de acceso del Servidor Web Coordinador

Luego de que el módulo `core` haya sido iniciado en un Servidor de Zona, este se encontrará en la capacidad de recibir mensajes del Servidor Web Coordinador para comunicarle su estado (libre u ocupado), unirse a una cuadrícula de zonas para zonificar un entorno virtual determinado, y recibir los elementos correspondientes a su porción del modelo del entorno como se mostrará posteriormente en el Caso de Uso 2 (Sección 5.3.5.2).

Para que un Servidor Web Coordinador realice la zonificación de un entorno virtual en MxN zonas, se deben presentar una serie de interacciones ilustradas en la Figura 45:

1) El Servidor de Zona inicial (en el que se cargó el modelo) debe solicitar al Servidor Web Coordinador que zonifique su entorno virtual.
2) El Servidor Web Coordinador debe contactar a cada uno de los Servidores de Zona disponibles para invitarlos a participar en la zonificación
3) Los Servidores de Zona deben asociar las fronteras con sus vecinos.
4) Los Servidores de Zona deben solicitar al Servidor de Zona inicial los elementos ubicados en la zona de la que están a cargo.
5) El Servidor de Zona inicial debe enviar los elementos solicitados.

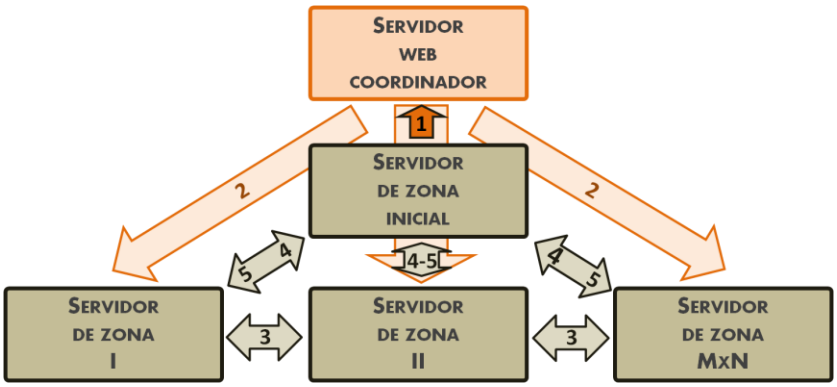

Figura 45. Secuencia de interacciones durante la zonificación de un entorno virtual

Zone.js utiliza una clase llamada `Grid` para administrar la información espacial de una zona obtenida al aceptar una solicitud de unión a una cuadrícula del Servidor Web Coordinador o al cargar un modelo con el método `loadModel(…)`. Esta clase permite almacenar propiedades como el ancho (`gWidth`), el largo (`gLength`), el margen de anticipación o aplazamiento (`margin`) y las coordenadas de una zona (`coords`), y parte de su implementación se muestra en el Segmento de Código 28 (archivo 'grid.js').

```
var Grid = function(newGrid) {
      this.margin = newGrid.margin;
      this.coords = {
            x: newGrid.coords.x,
            y: newGrid.coords.y,
            z: newGrid.coords.z
      };
      this.gWidth = newGrid.widthTiles * newGrid.tileSize;
      this.gLength = newGrid.lengthTiles * newGrid.tileSize;
      this.tileSize = newGrid.tileSize;
      this.neighbors = [];
}
...
```

Código 28. Implementación de 'grid.js', la información espacial de los Servidores de Zona

5.3.3 MANEJO DE LA COMUNICACIÓN ENTRE ZONAS

Una vez iniciado el servidor de soporte `core`, un Servidor de Zona puede proceder a iniciar el servidor de soporte `jamp`, que será el encargado de implementar el protocolo JAMP para llevar a cabo las comunicaciones y operaciones necesarias para la asociación de las fronteras con las zonas vecinas (e.g. `glueBorder`), el manejo de los cruces de frontera (e.g. `bookIn`, `checkIn`) y la construcción del entorno visible en las fronteras (`getVisibleElements`).

- **jamp.js**

El Segmento de Código 29 muestra parte de la implementación del servidor de soporte `jamp` en el archivo 'jamp.js', en el cual se puede apreciar una extensa lista de propiedades agrupadas en el objeto `JAMP`, entre las cuales se destacan los puertos del servidor `jamp` (`ports`), el arreglo de vecinos de la zona (`neighbors`), y el arreglo de usuarios reservados en cada zona vecina (`bookedIn`).

```
(function(ZONE) {
var tcpEventEmitter = require('../utils/tcpEventEmitter');
var JAMP = {
      ports: {…},
      transports: {…},
      neighbors: [],
      sides: […]
      gridRels: […],
      quadrantRels: […],
      bookedIn: […],
      fileKeys: [],
      retryTimeout: <number>
};

module.exports = {
      createServer: createServer,
      glueBorder: glueBorder,
      joinGrid: joinGrid,

      bookIn: bookIn,
      checkIn: checkIn,
      bookOut: bookOut,

      getVisibleElements: getVisibleElements
};
...
})(module.parent.ZONE);
```

Código 29. Implementación de 'jamp.js', el servidor de paso de mensajes entre Servidores de Zona

- **El Protocolo JAMP: JavaScript Asset Migration Protocol**

El aporte teórico de este trabajo de investigación, partiendo del diseño de estrategias de transferencia y zonificación, se condensa entonces en la propuesta de un protocolo de aplicación para la migración de elementos virtuales entre servidores utilizando JavaScript: el protocolo *JAMP*, por las siglas en inglés de *JavaScript Asset Migration Protocol*.

La Figura 46 muestra gráficamente las interacciones del protocolo JAMP durante un cruce de frontera, que han sido mencionadas anteriormente en términos de reservas o transferencias anticipadas de datos (i.e. *book-ins*) y operaciones de cesión de control (i.e. *check-ins*):

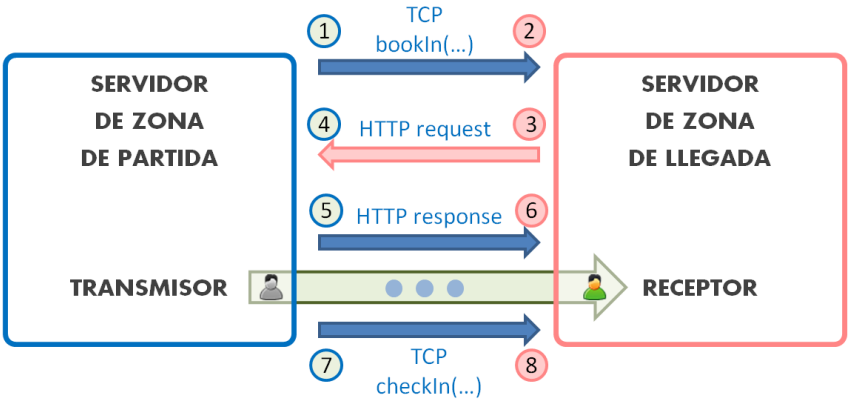

Figura 46. Secuencia de interacciones del protocolo JAMP durante un cruce de frontera

Utilizando los números de secuencia de la Figura 46, resulta conveniente presentar las interacciones del protocolo JAMP como una lista de operaciones efectuadas por los servidores involucrados durante el proceso de cruce de frontera de un avatar:

1) El transmisor envía un mensaje TCP de `bookIn` de un avatar.
2) El receptor detecta el mensaje TCP de `bookIn` del transmisor.
3) El receptor realiza una solicitud HTTP del archivo del avatar a reservar.
4) El transmisor detecta la solicitud HTTP del archivo del avatar.
5) El transmisor envía la respuesta HTTP con el archivo correspondiente.
6) El receptor maneja los segmentos de la repuesta HTTP y escribe el archivo.
7) El transmisor envía un mensaje TCP de `checkIn` cuando el avatar sale de la zona.
8) El receptor detecta el mensaje TCP `checkIn` y empieza a controlar el avatar.

Cabe mencionar que el mensaje de `checkIn`, además de ceder el control de un avatar a al Servidor de Zona de destino, contiene información sobre la posición de ingreso y las teclas que el usuario tiene presionadas junto a las fuerzas resultantes aplicadas en el modelo, y que esta será utilizada por el Servidor de Zona de destino para continuar desplazando al avatar correspondiente en su zona, hasta que el Cliente Web envié nuevos comandos de desplazamiento que modifiquen las fuerzas aplicadas sobre su modelo y en consecuencia su trayectoria.

5.3.4 MANEJO DE LA COMUNICACIÓN CLIENTE-SERVIDOR

Finalmente, con los servidores de soporte `core` y `jamp` en ejecución, un Servidor de Zona puede proceder a iniciar el servidor de soporte `webs`, que será el encargado de manejar las comunicaciones con los Clientes Web a través de conexiones WebSockets utilizando el módulo `socket.io`.

- **webs.js**

El Segmento de Código 30 muestra algunas líneas de la implementación del módulo `webs` en el archivo 'webs.js', en las que se muestra la inclusión del módulo `socket.io` y se expone su funcionalidad a través de los métodos `createServer(…)` y `getSocket(…)`.

```
(function(ZONE) {
var io = require('socket.io');
var WEBS = {
      port: 0,
      nameSpaces: {…},
      socketHistory: [],
      clientsIO: {}
};
module.exports = {
      createServer: createServer,
      getSocket: getSocket
};
...
})(module.parent.ZONE);
```

Código 30. Implementación de 'webs.js', el servidor de comunicaciones a través de WebSockets

Para que un Cliente Web establezca una conexión WebSockets con un Servidor de Zona a través del módulo `webs` debe seguir los siguientes pasos ilustrados en la Figura 47:

1) Enviar una solicitud de la página de despliegue al Servidor Web Coordinador.
2) Recibir la página de despliegue y extraer la información de contacto del Servidor de Zona como se mostrará en el Caso de Uso 5 (Sección 5.3.5.5).
3) Establecer una conexión WebSocket con el Servidor de Zona correspondiente.

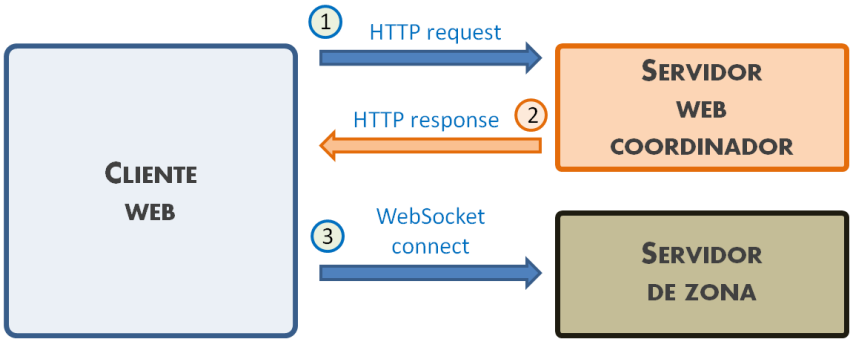

Figura 47. Secuencia de interacciones cliente-servidor para el acceso a un entorno virtual

5.3.5 CASOS DE USO

Luego de haber abordado con detalle la implementación de Zone.js y haber introducido la implementación de sus principales sub-módulos como los servidores de soporte, se puede proceder a describir la implementación de los Casos de Uso a partir de la funcionalidad ofrecida por cada uno de estos.

5.3.5.1 Caso de Uso #1: Carga del modelo

En primer lugar, para realizar la carga del modelo de un entorno virtual en un Servidor de Zona se debe contar con una carpeta ubicada en una ruta `filePath` dada, la cual debe contener un archivo 'model.json' de descripción de modelo como el mostrado en el Segmento de Código 31, y una sub-carpeta `elements` con los archivos de los elementos listados en el arreglo `"elements"` del archivo de descripción.

```
{"name":"ModelName",
  "type":"2D",
  "camera":"static",
  "grid": {
      "margin": 10,
      "coords": { "x":0, "y":0, "z":0 },
      "widthTiles":100,
      "lengthTiles":100,
      "tileSize":1
  },
  "elements":[
      {"src":"source.png",
        "position": {"x":50,"y":50, "z":0},
        "scale": {"x":3,"y":3, "z":1}}
      ...
  ]}
```

Código 31. Archivo 'model.json' de descripción del modelo de un entorno virtual

Luego se puede proceder a invocar el método `loadModel(filePath,…)` del módulo Zone.js (interfaz `zone`) como se muestra en el Segmento de Código 32, suministrando la variable `filePath` como parámetro y una función *callback* opcional. En caso de que la extensión del archivo sea distinta de '.js' y '.json', que el contenido del mismo no siga la notación JSON, o que falte alguno de los campos obligatorios del modelo (`name`, `type`, `camera`, `grid`, `elements`), la solicitud de carga del modelo será rechazada.

```
var zone = require('zone');
var filePath = "path/to/model";

zone.loadModel(filePath, function() {
      // Model has loaded
});
```

Código 32. Carga del modelo de un entorno virtual en Zone.js

5.3.5.2 Caso de Uso #2: Simulación del entorno

Para iniciar la ejecución de un Servidor de Zona o la simulación del entorno virtual correspondiente, se debe invocar el método `start(host, ports,…)` del módulo Zone.js (interfaz `zone`) como se muestra en el Segmento de Código 33, suministrando como parámetros el nombre de dominio o dirección IP del servidor (`host`) y un objeto `ports` con los números de los puertos en los que van a operar el servidor web del archivo 'mvc.js' (`webServer`) y los servidores de soporte `core` (`coreTCP`), `jamp` (`jampTCP`, `jampHTTP`) y `webs` (`webSockets`).

```
var zone = require('zone');

var host = "some.domain.or.ip";
var ports = {
  coreTCP: <number>,
  jampTCP: <number>,
  jampHTTP: <number>,
  webSockets: <number>,
  webServer: <number>
};

zone.start(host, ports, function() {
      // Zone.js has started.
});
```

Código 33. Inicio de la ejecución de un Servidor de Zona en Zone.js

Cabe mencionar que es posible indicar únicamente el puerto `coreTCP` en el objeto `ports`, caso en el que Zone.js asignará los siguientes cuatro puertos consecutivos a los puertos no indicados explícitamente.

Adicionalmente, se debe resaltar que el método `start(…)` puede ser invocado sin haber cargado un modelo previamente con el método `loadModel(…)`, caso en el que el Servidor de Zona resultante contendrá un modelo vacío pero estará en la capacidad de unirse a una cuadrícula de Servidores de Zona e incluir elementos a su modelo cuando el Servidor Web Coordinador se lo indique como se ilustró en la Figura 45 (Sección 5.3.2).

5.3.5.3 Caso de Uso #3: Zonificación de un entorno virtual

Una vez el modelo de un entorno virtual ha sido cargado en un Servidor de Zona se puede solicitar la zonificación del entorno por medio de la invocación del método `splitWorld(rows, cols, cWebServer,…)` como se muestra en el Segmento de Código 34. Este método, el tercero y último del módulo Zone.js (interfaz `zone`), recibe como parámetros el número de filas (`rows`) y columnas (`cols`) de la cuadrícula de zonificación deseada junto a un objeto `cWebServer` que debe contener los datos de contacto del Servidor Web Coordinador.

```
var zone = require('zone');

var rows = <number>, cols = <number>;
var cWebServer = {
      host: "coordinator.domain.or.ip",
      port: <number>
}

zone.splitWorld(rows, cols, cWebServer, function() {
      // Model has been divided in rowsXcols zones
});
```

Código 34. Solicitud de zonificación de un entorno virtual en Zone.js

Luego de que el Servidor de Zona inicial realice la solicitud de zonificación al Servidor Web Coordinador, este último determinará si existen suficientes Servidores de Zona disponibles para efectuar la zonificación solicitada, y en caso afirmativo enviará un mensaje de invitación `'joinGrid'` que será recibido en cada uno de los Servidores de Zona como se muestra en el Segmento de Código 35, los cuales procederán a unirse a la cuadrícula `rowsXcols` y empezarán a asociar sus fronteras con los vecinos indicados.

```
var socket = tcpEventEmitter.bind(tcpSocket);

socket.on('status', function(message) {
      socket.send('status', { available: (ZONE.id==0) });
});
socket.on('joinGrid', function(message) {
      if(typeof message.newID!='undefined' && ZONE.id==0) {
            // 1. Set grid properties
            // 2. Contact neighbors to glue borders
      }
});
socket.on('addElement', function(message) {
      http.get({options}, function() {
            // 1. Write file
            // 2. Add element to model
      });
});
```

Código 35. Manejo de los mensajes de zonificación recibidos del Servidor Web Coordinador

Posteriormente, el Servidor Web Coordinador clasificará los elementos del modelo del Servidor de Zona inicial según su posición en la cuadrícula solicitada, y enviará un mensaje `'addElement'` por cada elemento del modelo al Servidor de Zona responsable del mismo según la zona en la que haya sido clasificado. Al final del proceso de zonificación, cada zona de la cuadrícula será manejada por un Servidor de Zona diferente, y los elementos del modelo inicial se encontrarán repartidos en distintos servidores.

5.3.5.4 Caso de Uso #4: Suministro de la página de despliegue a un usuario

Desde el punto de vista de los Clientes Web, la responsabilidad principal del Servidor Web Coordinador es proporcionarles la página de despliegue y la información de contacto necesaria para empezar a comunicarse con un Servidor de Zona del entorno virtual.

El Segmento de Código 36 muestra un ejemplo de página de despliegue en el que se puede observar la presencia del elemento `<canvas>` que será utilizado para la presentación gráfica del entorno, la referencia `<script>` al código 'mvc.js' necesario para el manejo de comandos, comunicaciones y generación de gráficas, y la inclusión de la información de contacto `iohost` y `ioport` en la etiqueta `<meta>`.

```
<html>
<head>
      <meta iohost="zoneHost" ioport="webSocketsPort" />
      <script src="http://zoneHost:webServerPort/mvc.js">
      </script>
</head>
<body>
      <canvas id="canvasID"></canvas>
</body>
```

Código 36. Página de despliegue con la información de contacto de la capa de aplicación

La página de despliegue debe ser solicitada por los Clientes Web utilizando un navegador web con capacidad HTML5 y soporte WebSocket, y debe ser suministrada por un servidor web convencional como Apache instalado en el Servidor Web Coordinador. Esta página puede ser nombrada como lo desee el desarrollador o administrador del entorno virtual, o almacenada en el archivo 'index.html' o 'index.php' que es retornado por defecto cuando se realizar la solicitud de un nombre de dominio sin especificar una ruta.

5.3.5.5 Caso de Uso #6: Acceso de un usuario a un entorno virtual

Una vez un entorno virtual determinado se encuentre en ejecución podrá empezar a recibir solicitudes de acceso por parte de Clientes Web que ya dispongan de la página de despliegue suministrada por el Servidor web Coordinador y de la información de contacto de la capa de aplicación embebida en la misma.

Cuando el Cliente Web obtenga el archivo 'mvc.js' del componente `webServer` del Servidor de Zona con nombre de dominio o dirección IP `zoneHost`, podrá extraer la información de contacto de la etiqueta `<meta>` (`ioConfig(…)`) y proceder a establecer la conexión WebSocket con el módulo `webs` del Servidor de Zona en el puerto `webSocketsPort` (`ioConnect(…)`) como se muestra en el Segmento de Código 37. Cabe resaltar la utilización de la variable `LOCAL` para el almacenamiento de la conexión WebSocket, pues este será el mecanismo utilizado por Zone.js para intercambiar las conexiones cliente-servidor en los cruces de fronteras.

```
var LOCAL = {};

function ioConfig(callback) {
      var io = document.createElement('script');
      var host = document.getElementsByTagName('meta')[0]
                                  .getAttribute('iohost');
      var port = document.getElementsByTagName('meta')[0]
                                  .getAttribute('ioport');

      io.setAttribute('type', 'text/javascript');
      io.setAttribute('src', "http://"+host+":"+port+
                              +"/socket.io/socket.io.js");
      io.onload = function() { ioConnect(host, port); }
      document.head.appendChild(io);
}

function ioConnect(host, port) {
      LOCAL.commands =
          io.connect("http://"+host+":"+port+"/commands");
      ioHandleCommands(LOCAL.commands, true);
}
```

Código 37. Establecimiento de la conexión WebSocket con los datos de la página de despliegue

Luego de establecer la conexión con el módulo `webs` de un Servidor de Zona, un Cliente Web habrá accedido oficialmente al entorno virtual y deberá recibir las especificaciones del modelo (mensaje `'modelSpecs'`), los datos de su avatar (mensaje `avatar'`) y los elementos del entorno circundante (mensajes `'newElement'`), como se muestra en el Segmento de Código 38. Estos mensajes le permitirán almacenar la conexión inicial (`LOCAL.webSockets[…]`), iniciar el despliegue gráfico (`MVC.view.startRendering`) y empezar la simulación del entorno en el navegador web (`MVC.model.run()`).

```
commands.on('modelSpecs', function(modelSpecs) {
      LOCAL.webSockets[modelSpecs.zoneID+""] = commands;
      MVC.model.setup(modelSpecs);
      MVC.view.startRendering(…);
});
commands.on('avatar', function(avatar) {
      MVC.model.addAvatar(avatar);
      LOCAL.avatar.name = avatar.name;
      MVC.model.run();
});
commands.on('newElement', function(newElement) {
      MVC.model.addElement(newElement);
});
```

Código 38. Recepción del modelo de un entorno virtual en un Cliente Web

5.3.5.6 Caso de Uso #6: Desplazamiento de un usuario por un entorno

Cuando se haya cargado el modelo circundante en el navegador web y los componentes MVC del Cliente Web se encuentren en ejecución, un usuario podrá empezar a ingresar comandos de desplazamiento para recorrer un entorno virtual dado.

Para minimizar el tráfico de las comunicaciones relacionadas con las posiciones de los usuarios, Zone.js únicamente envía mensajes de un Cliente Web a un Servidor de Zona cuando suceden los eventos `onkeydown` y `onkeyup`, evitando así la alternativa de mayor tráfico de enviar la posición del usuario de forma periódica en cada iteración del modelo. Estos comandos podrán ser convertidos posteriormente en fuerzas que al ser procesadas por el modelo se traducirán en el movimiento del avatar en la dirección correspondiente.

El Segmento de Código 39 muestra parte de la implementación del manejo del evento `onkeydown` en Zone.js, en el cual las líneas comentadas representan las operaciones de validación e interpretación de teclas, y las invocaciones `avatar.applyForce(…)` y `LOCAL.commands.emit('keyDown',…)` permiten aplicar la fuerza apropiada sobre el modelo y enviar el comando al Servidor de Zona para que sea procesado, respectivamente.

```
CONTROLLER.onKeyDown = function(key) {
var key = String.fromCharCode(event.keyCode || event.which);
      // 1. Check valid keys
      // 2. Check pressed keys
      // 3. Check movement keys
      avatar.applyForce(LOCAL.movementKeys[key], 1);
      LOCAL.commands.emit('keyDown',
             { key: key, , position: avatar.position}
      );
}
```

Código 39. Manejo del evento 'onkeydown' en un Cliente Web de Zone.js

Cabe recordar que el objetivo de ejecutar la sentencia `avatar.applyForce(…)` consiste en permitirle al modelo (componente MVC) del Cliente Web que realice la predicción de la respuesta del modelo para evitar los retardos de red que serían visibles si se tuviera que esperar la respuesta del Servidor de Zona para responder a los comandos del usuario. Esta predicción, en conjunto con las políticas de anticipación (*book-ins*) y aplazamientos en la cesión de control (*check-ins*) en los cruces de frontera, constituye el fundamento de Zone.js para reducir las discontinuidades espaciales e inconsistencias lógicas mencionadas durante los cruces de fronteras.

Por otro lado, se puede observar que el envío de comandos se realiza a través del método `emit(…)` del objeto `LOCAL.commands`, en el que anteriormente se mencionó que sería almacenada la conexión WebSocket con el Servidor de Zona actual, y cuyo valor será intercambiado con las conexiones almacenadas en el arreglo `LOCAL.webSockets[…]` (Segmento de Código 38) cuando el avatar cambie de zona al realizar cruces de fronteras.

5.3.5.7 Caso de Uso #7: Visibilidad en las fronteras entre zonas

Para realizar la construcción del entorno visible de los avatars que se acerquen a las fronteras de una zona y abarquen regiones de otras zonas con su rango de visibilidad, Zone.js utiliza un mecanismo basado en 9 cuadrantes internos y 8 externos (equivalencias adicionales) similar al presentado en la Figura 41 (Sección 5.2.3) pero suministrando como parámetro el rango de visibilidad del usuario `visibleRange` como se muestra en el Segmento de Código 40.

```
var visibleQuadrant = getQuadrant(element, visibleRange);
// 1. Detection of quadrantChange
if(quadrantChange) {
      switch(quadrant) {
            case …: ZONE.servers.jamp
                        .getVisibleElements(element, quadrant);
      }
}
```

Código 40. Verificación del alcance del rango visible de un avatar

Cabe resaltar el uso del método `getVisibleElements` del servidor de soporte `jamp`, el cual permite enviar un mensaje con tema `'getVisibleElements'` a un Servidor de Zona vecino para que este lo reciba y responda como se muestra en el Segmento de Código 41, en el que se enviarán de vuelta al Servidor de Zona solicitante todos los elementos que cumplan una condición de proximidad usando el método `socketOut.send(…)`.

```
socketIn.on('getVisibleElements', function() {
      ZONE.mvc.model.elemKeys.forEach(function(key) {
            // Condition validation ...
            if(condition)
               socketOut.send('newVisibleElement', element);
      });
});
```

Código 41. Recepción de una solicitud de elementos visibles en un Servidor de Zona vecino

Finalmente, el Servidor de Zona que solicitó los elementos visibles de la frontera a un Servidor de Zona vecino recibirá dichos elementos como se muestra en el Segmento de Código 42, y procederá a escribir el archivo del elemento en el sistema de archivos y enviarlo al Cliente Web correspondiente para que sea agregado al modelo.

```
socketIn.on('newVisibleElement', function() {
      http.get({options}, function() {
            // 1. Write file
            // 2. Send element to client
      });
});
```

Código 42. Recepción de los elementos visibles solicitados a un Servidor de Zona vecino

5.3.5.8 Caso de Uso #8: Cruce de fronteras entre zonas

El último Caso de Uso del sistema corresponde al más importante del problema abordado por este proyecto de investigación: el cruce de fronteras entre zonas. Los Segmentos de Código 43 y 44 muestran la implementación de las interacciones del protocolo JAMP descrito en la Sección 5.3.3 e ilustrado en la Figura 46, en los que se pueden apreciar los mensajes de `'bookIn'` y `'checkIn'` enviados por el transmisor con las sentencias `socket.send('bookIn')` y `socket.send('checkIn')` y manejados por el receptor con las sentencias `socket.on('bookIn')` y `socket.on('checkIn')`.

```
/*1*/ socket.send('bookIn',…)


/*4*/ server.on('request',…){
/*5*/  response.end(avatarFile)
      }
...
/*7*/ socket.send('checkIn')
```

Código 43. Acciones de un transmisor JAMP

```


/*2*/ socket.on('bookIn',…)
/*3*/ http.get(…,avatarName)


/*6*/ response.on(…,avatarFile)
...

/*8*/ socket.on('checkIn',…)
```

Código 44. Acciones de un receptor JAMP

También se puede apreciar la solicitud `http` de un avatar `avatarName` efectuada por el receptor con la sentencia `http.get(…,avatarName)`, la detección de la solicitud `server.on('request',…)` y el envío de la respuesta `http` del archivo con la sentencia `response.end(avatarFile)` en el transmisor, y la detección de este archivo en el receptor con la sentencia `response.on(…,avatarFile)`.

Para terminar, el Segmento de Código 45 muestra una parte de la implementación del manejo de los mensajes de `'bookIn'`, `'bookOut'` y `'checkIn'` en un Servidor de Zona.

```
socket.on('bookIn', function(element) {
      var file = fs.createWriteStream(…);
      http.get({…,path:'/avatars/'+element.key}, function(){
            response.on('data', function(newChunk) {
                  file.write(newChunk);
            });
            response.on('end', function() {
                  ZONE.mvc.model.bookInElement(element); …
            });
      })
});
socket.on('checkIn', function(element) {
      ZONE.mvc.model.checkInElement(element);
});
```

Código 45. Manejo de los mensajes de 'bookIn' y 'checkIn' en un Servidor de Zona

Al culminar la presentación de los desarrollos de este trabajo de investigación, vale la pena recapitular lo discutido en esta sección antes de pasar a analizar los resultados obtenidos en la Sección 6.

En primer lugar, la justificación teórica de las estrategias propuestas fue presentada en la Sub-sección 5.1. Se discutieron las ventajas de implementar la plataforma Zone.js como un sistema distribuido débilmente acoplado en hardware y no para ser ejecutado sobre un *cluster*, y se presentaron ejemplos numéricos que respaldan la estrategia de transferencia anticipada de datos y la política de predicción de cruces de fronteras utilizadas en Zone.js.

Posteriormente, el proceso de diseño de Zone.js fue presentado en la Sub-sección 5.2. Se ilustró la arquitectura detallada del sistema precedida de la arquitectura general de cada uno de sus componentes, seguida de la explicación detallada de la estrategia de transferencia de carga en los cruces de frontera basada en un margen de histéresis doble para realizar reservas anticipadas (*book-ins*) y aplazar las operaciones de cesión de control (*check-ins*), y de la estrategia de zonificación basada en cuadrantes empleada en Zone.js para determinar las zonas de contacto o destino involucradas en la construcción del entorno visible en las fronteras o los cruces de fronteras de los usuarios.

Finalmente, la implementación de Zone.js como un módulo de la plataforma Node.js y sus respectivos Casos de Uso fueron presentados en la Sub-sección 5.3. Se describió la estructura del módulo Zone.js en términos de los archivos y carpetas que lo componen, y se discutió la implementación general de los sub-módulos encargados de participar en la zonificación de un entorno (`webServer`, `gridify`, `core`), de implementar el protocolo JAMP para realizar la comunicación entre Servidores de Zona para la administración de las frontera (`jamp`) y de manejar la comunicación con los Clientes Web (`webs`), para cerrar con la descripción de la implementación de los 8 Casos de Uso descritos en la Sección 4.1.

De este modo, se concluye el núcleo de diseño e implementación de este trabajo de investigación, y se procederá a realizar el análisis de resultados de la implementación de Zone.js en la Sección 6.

6. PRUEBAS

Como preámbulo a los resultados de este trabajo de investigación, en esta sección se presenta el protocolo de pruebas utilizado para verificar el funcionamiento de la arquitectura de computación distribuida diseñada y de la correspondiente implementación en el módulo Zone.js, especificando el modelo del entorno virtual a zonificar, describiendo las características de los servidores utilizados para esta labor, e ilustrando las trayectorias de desplazamiento de un avatar que se recorrerán durante las pruebas.

6.1 MODELO DE PRUEBA

El Segmento de Código 46 muestra el archivo 'model.json' que contiene la información del entorno virtual que será utilizado para comprobar la funcionalidad del módulo Zone.js. Se puede apreciar que el nombre del modelo es "Static2DGrid", que es de tipo 2D, que la cámara a utilizar es estática, que el margen de anticipación es de 30 unidades, que se encuentra ubicado en las coordenadas (0,0,0), que tiene un ancho y un largo de 400, y que está poblado por 16 elementos con sus respectivas posiciones en el entorno.

```
{"name":"Static2DGrid",   "type":"2D",       "camera":"static",
"grid": {"margin": 30,     "coords": {"x":0,"y":0,"z":0},
         "widthTiles":400,"lengthTiles":400, "tileSize":1 },
"elements":[
  {"src":"bug.png",       "position":{"x":175,"y": 30,"z":0},…},
  {"src":"spider.png",    "position":{"x":175,"y":180,"z":0},…},
  {"src":"carGray.png",   "position":{"x": 25,"y":350,"z":0},…},
  {"src":"userYellow.png","position":{"x":225,"y":360,"z":0},…},
  {"src":"userBlue.png",  "position":{"x":175,"y":225,"z":0},…},
  {"src":"userRed.png",   "position":{"x":360,"y":175,"z":0},…},
  {"src":"userPurple.png","position":{"x": 25,"y":175,"z":0},…},
  {"src":"userGray.png",  "position":{"x":225,"y":175,"z":0},…},
  {"src":"policeMan.png", "position":{"x":360,"y":225,"z":0},…},
  {"src":"house1.png",    "position":{"x":350,"y": 25,"z":0},…},
  {"src":"house2.png",    "position":{"x": 25,"y": 25,"z":0},…},
  {"src":"house3.png",    "position":{"x":350,"y":350,"z":0},…},
  {"src":"warehouse.png", "position":{"x":165,"y":350,"z":0},…},
  {"src":"football.png",  "position":{"x":225,"y": 25,"z":0},…},
  {"src":"basketball.png","position":{"x": 25,"y":225,"z":0},…},
  {"src":"baseball.png",  "position":{"x":225,"y":225,"z":0},…}
]}
```

Código 46. Archivo del modelo de prueba del sistema ('model.json')

Este archivo será enviado por el Servidor de Zona solicitante de una zonificación al Servidor Web Coordinador para que este determine si es posible efectuar la división solicitada y distribuya los elementos a diferentes Servidores de Zona según su ubicación final en la cuadrícula de zonas.

A su vez, la Figura 48 muestra la representación gráfica del modelo de prueba del sistema, o mejor dicho, cómo se debe ver el entorno virtual cuando es manejado por un solo Servidor de Zona y accedido utilizando un navegador web con soporte del elemento `<canvas>` de HTML5.

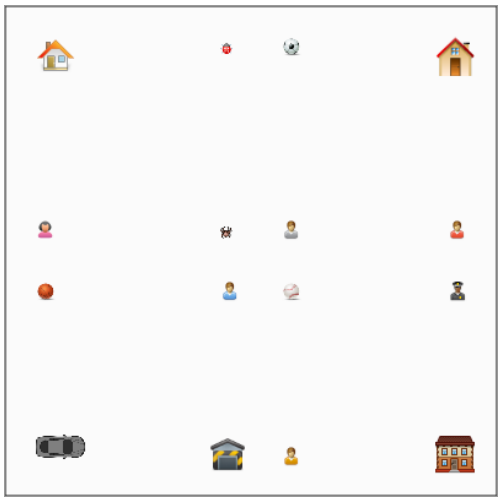

Figura 48. Representación gráfica del modelo de prueba del sistema

Cabe aclarar que, aunque todas las pruebas del sistema se presentarán con el modelo anterior de 16 elementos, Zone.js fue diseñado e implementado para soportar modelos 3D complejos como el que se muestra en la Figura 49, y para demostrarlo se utilizará el archivo del modelo 3D de esta figura (‘ogro-light.js’ de 270 KB) en las operaciones de transferencia de los cruces de frontera, que equivaldrá a emular la migración de avatars 3D entre zonas aunque solo se utilicen imágenes livianas en el despliegue gráfico por su facilidad de presentación en los resultados del documento.

Figura 49. Modelo 3D ‘ogro-light.js’ a transferir en los cruces de fronteras [22]

También resulta conveniente anotar que, gracias al patrón arquitectónico Modelo-Vista-Controlador utilizado en Zone.js, las pruebas que se realizarán al sistema generarían los mismos resultados si se modificara la vista (componente MVC) para realizar un despliegue gráfico 3D, pues la lógica de las reservas anticipadas y operaciones de cesión de control fue implementada con la información espacial de dos dimensiones únicamente (‘x’ y ‘y’), y por ello los cambios en la vista serían irrelevantes para el controlador y el servidor JAMP.

6.2 SERVIDORES UTILIZADOS

Para demostrar la funcionalidad de Zone.js, el modelo presentado en la sección anterior será dividido en 4 zonas del mismo tamaño (2 filas y 2 columnas) utilizando una configuración de 6 servidores organizados según la arquitectura del sistema de la Figura 25:

- Un (1) Servidor Web Coordinador.
- Un (1) Servidor de Zona inicial que cargue los datos y solicite la zonificación.
- Cuatro (4) Servidores de Zona que acepten la solicitud de participación en el proceso de zonificación que será enviada por el Servidor Web Coordinador.

La razón por la que se requieren 5 (MxN+1) Servidores de Zona en vez de 4 (MxN) es que el proceso de zonificación de Zone.js (ilustrado en la Figura 45) fue diseñado para repartir el modelo inicial a Servidores de Zona que hayan sido iniciados sin un modelo propio (solo con el método `start(…)`), por lo que el solicitante de la división del entorno quedará en desuso cuando se lleve a cabo la zonificación.

La Tabla 4 muestra la lista de los servidores a utilizar, donde `webServer` es el Servidor Web Coordinador, `zone0` el Servidor de Zona inicial que solicitará la zonificación del entorno, y `zone1-2-3-4` son los Servidores de Zona que participarán en dicha zonificación. Estos servidores fueron creados en la plataforma de computación en la nube *Rackspace* [23] con sus respectivas direcciones IP, sistemas operativos y valores de memoria RAM; el número de puerto principal (`coreTCP`) se incluye para facilitar la lectura de las salidas de las consolas, y la versión de la plataforma Node.js con un propósito informativo y de compatibilidad.

NOMBRE	DIRECCIÓN IP	PUERTO CoreTCP	SISTEMA OPERATIVO	MEMORIA RAM	VERSIÓN NODE.JS
webServer	50.57.222.231	10000	Ubuntu 12.04	256 MB	0.6.12
zone0	50.57.171.154	54321	Ubuntu 12.04	256 MB	0.6.12
zone1	50.57.222.232	12345	Ubuntu 12.04	256 MB	0.6.12
zone2	50.57.226.31	23456	Ubuntu 12.04	256 MB	0.6.12
zone3	50.57.223.120	34567	Ubuntu 12.04	256 MB	0.6.12
zone4	50.57.223.169	45678	Ubuntu 12.04	256 MB	0.6.12

Tabla 4. Características de los servidores utilizados para las pruebas del sistema

Para acceder de forma remota a estos servidores e iniciar el módulo Zone.js desde la línea de comandos se empleó la utilidad `ssh root@ipaddress` de la consola de Ubuntu Linux 12.04, y para la transferencia del módulo Zone.js al sistema de archivos de cada Servidor de Zona y del archivo ‘gridify.js’ al Servidor Web Coordinador el programa FTP FileZilla. A su vez, en el Servidor Web Coordinador fue instalada la versión 1.7.7 del paquete de servidor web XAMPP para servir la página de despliegue del entorno.

6.3 TRAYECTORIAS DE DESPLAZAMIENTO

Dado que el manejo de los cruces de fronteras entre zonas es el principal objetivo de este trabajo de investigación, resulta conveniente describir el protocolo de pruebas que será utilizado para verificar el funcionamiento de las estrategias diseñadas para resolver los problemas de las fronteras discutidos en secciones anteriores.

La Figura 50 muestra los 4 tipos de trayectorias que serán utilizados para probar el manejo de cruces de fronteras en Zone.js y recorridas a una velocidad de desplazamiento de 60 unidades por segundo, que a una frecuencia de procesamiento del modelo de 30 iteraciones por segundo se recorrerán en saltos de 2 unidades cada 33.3 ms. En el cuadrante superior izquierdo se ilustra la trayectoria básica de cruces perpendiculares de fronteras, con la cual se espera que el usuario sea transferido de una zona a otra siguiendo la secuencia 1-2-4-3 hasta regresar al punto de partida en la zona 1. En el cuadrante superior derecho se ilustra una trayectoria diagonal que generará reservas en las zonas 2 y 3, pero que deberá culminar en la cesión de control del avatar a la zona 4.

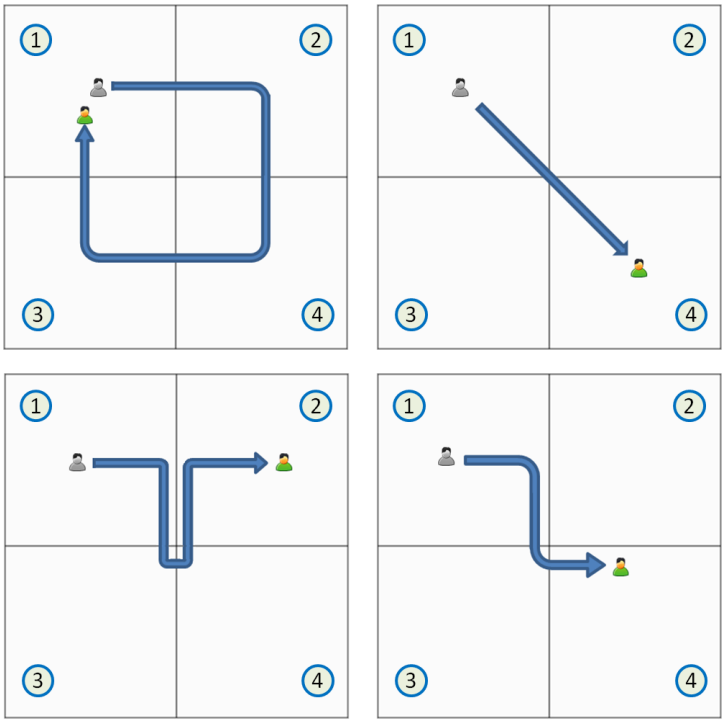

Figura 50. Trayectorias de prueba para los cruces de fronteras

Por otro lado, en la parte inferior izquierda se muestra una trayectoria más compleja que la del primer caso para llegar de la zona 1 a la zona 2, pues se realiza una forma de 'U' con la que generan reservas en las zonas 3 y 4 y recorren partes ellas, pero que solo debe generar la cesión de control de la zona 1 a la zona 2. Finalmente, en la parte inferior derecha se ilustra una trayectoria en forma de escalón que debe generar reservas en las zonas 2 y 3, y que al provocar la cesión de control a la zona 4 debe generar una reserva inmediata de la zona 4 a la zona 2 por la proximidad del avatar a la frontera.

7. RESULTADOS

Luego, teniendo en cuenta las especificaciones de los sistemas y los protocolos de prueba descritos en la Sección 6, a continuación se presentan los resultados obtenidos de la implementación de la arquitectura de computación distribuida diseñada en la Sección 5.2. Se empezará con la respuesta del sistema ante cada uno de los Casos de Uso descritos en la Sección 4.1, seguida de la discusión cualitativa sobre la forma en que se satisficieron los Requerimientos Funcionales y No Funcionales especificados en las Secciones 4.2 y 4.3, para finalizar con el análisis cuantitativo de los resultados.

7.1 CASOS DE USO

7.1.1 Caso de Uso #1: Un desarrollador proporciona los datos de un entorno virtual en un Servidor de Zona

En primer lugar, para proporcionar los datos del entorno virtual 2D presentado en la Sección 6.1 se inició una sesión FTP en el Servidor de Zona 0 (`zone0`) con el programa FileZilla para transferir los archivos que se muestran en el cuadrante izquierdo de la Figura 51: el archivo 'model.json', la carpeta 'elements' con las imágenes del modelo, y el archivo 'Load2DGrid'que será el encargado de cargar el modelo y solicitar su zonificación.

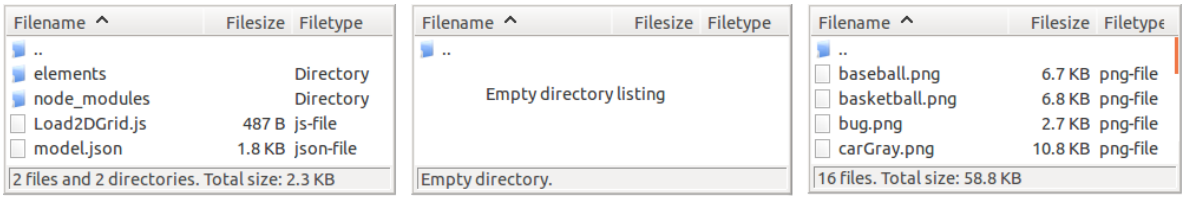

Figura 51. Archivos del Servidor de Zona 0 antes y después de cargar el modelo de prueba

Luego se procedió a acceder de forma remota utilizando el comando de consola `ssh root@50.57.171.154` para ejecutar el archivo 'Load2DGrid' con el comando `node Load2DGrid.js`. En el Segmento de Código 47 se puede apreciar que este archivo solicita el módulo Zone.js y ejecuta el método `loadModel(…)` suministrando como parámetro el directorio actual, donde acertadamente se encuentran la carpeta 'elements' y el archivo 'model.json' requeridos para cargar un modelo.

```
var zone = require("zone");
/* 1. Specify ports and filepath */
var host = "50.57.171.154", ports = { coreTCP: 54321};
var filePath = __dirname;
var webServer = { host: "50.57.222.231", port: 10000};
var rows = 2, cols = 2;
/* 2. Load model to zone */
zone.loadModel(filePath, function() {
      ...
});
```

Código 47. Carga del modelo de prueba en el Servidor de Zona 0

A su vez, en los cuadrantes central y derecho de la Figura 51 se puede apreciar el estado de la carpeta 'elements' del módulo Zone.js antes y después de que se ejecutara el método `loadModel(…)`, el cual copió todos los elementos de la carpeta 'elements' del cuadrante izquierdo de la Figura 51 que estaban señalados en el archivo 'model.json'. En correspondencia, la Figura 52 muestra la salida de la consola remota del Servidor de Zona inicial (`zone0`) en la que se reportó que el modelo fue cargado exitosamente, lo cual confirmó que el Caso de Uso #1 fue satisfecho.

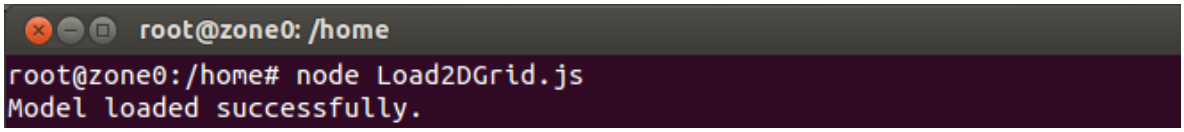

Figura 52. Salida de la consola al cargar el modelo en el Servidor de Zona 0

7.1.2 Caso de Uso #2: Un desarrollador inicia un Servidor de Zona

Luego de cargar los datos del entorno virtual en cuestión, se inició el Servidor de Zona 0 como se muestra en el Segmento de Código 48.

```
...
zone.loadModel(filePath, function() {
      zone.start(host, ports, function() {
         ...
      });
});
```

Código 48. Iniciación del Servidor de Zona 0

La Figura 53 muestra la salida de la consola del Servidor de Zona 0 al ejecutar el archivo 'Load2DGrid' extendido para ejecutar el método `start(…)`, en la que se puede apreciar que el Servidor de Zona inició con el puerto `coreTCP` especificado en el Segmento de Código 47, y que los demás puertos fueron asignados de forma consecutiva al no haber sido especificados como `coreTCP`, con lo cual se confirmó la satisfacción del Caso de Uso #2

Adicionalmente, se puede notar la presencia del mensaje '`info - socket.io started`' que indica que un nuevo servidor de conexiones WebSocket fue creado con el módulo 'socket.io' (en el puerto `WebSockets 54325`).

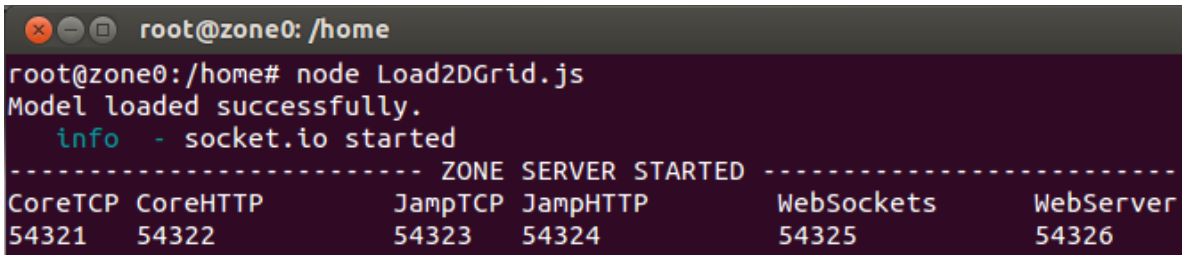

Figura 53. Salida de la consola al iniciar el Servidor de Zona 0

7.1.3 Caso de Uso #3: Un desarrollador solicita la división de un entorno virtual en zonas

Después de iniciar el Servidor de Zona 0 y tener el modelo cargado en memoria y en sus archivos de configuración, se procedió a ejecutar el método `splitWorld(…)` para solicitar al Servidor Web Coordinador la división del entorno virtual en zonas como se muestra en el Segmento de Código 49, donde se suministraron el número de filas (`rows`) y columnas (`cols`) de la cuadrícula 2x2 deseada, junto a la información de contacto del Servidor Web Coordinador en el parámetro `webServer`.

```
...
zone.loadModel(filePath, function() {
      zone.start(host, ports, function() {
         zone.splitWorld(rows, cols, webServer);
      });
});
```

Código 49. Solicitud de zonificación del Servidor de Zona 0

Para habilitar la recepción de solicitudes de zonificación por parte de Servidores de Zona, previamente se suministraron al Servidor Web Coordinador (`webServer`) el archivo de zonificación 'gridify.js' y el descriptor 'servers.json' como se muestra en el cuadrante derecho de la Figura 54.

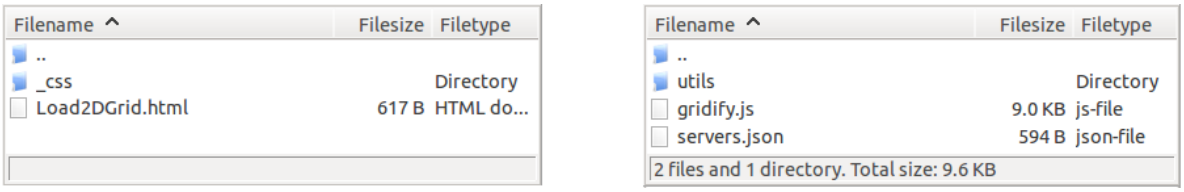

Figura 54. Archivos del Servidor Web Coordinador

De forma similar, para habilitar la recepción de solicitudes de participación en un proceso de zonificación en los servidores `zone1-2-3-4`, previamente se empleó la utilidad `ssh` para acceder de forma remota a cada uno de los servidores y ejecutar el archivo 'StartEmptyZone.js' que se muestra en el Segmento de Código 50 utilizando el comando de consola `node StartEmptyZone.js hostname:coreTCP`. El nombre de dominio o dirección IP `hostname` y el puerto `coreTCP` fueron suministrados como parámetros en la línea de comando y utilizados para iniciar cada servidor `zone1-2-3-4` sin proporcionarles un modelo.

```
var zone = require("zone");
var params = process.argv[2].split(":");
var host = (params[0] == '') ? "localhost" : params[0];
var ports = {coreTCP: parseInt(params[1])};
zone.start(host, ports);
```

Código 50. Iniciación de los Servidores de Zona 1, 2, 3 y 4 sin proporcionar un modelo

Luego, utilizando la información de los Servidores de Zona 1 a 4 enunciada en la Tabla 4 (Sección 6.2), el descriptor 'servers.json' fue configurado como se muestra en el Segmento de Código 51 para indicar que los servidores `zone1-2-3-4` se encontraban disponibles para participar en la zonificación solicitada por `zone0`.

```
{"available": [
      {"host":"50.57.222.232", "ports": {...} }, //zone1
      {"host":"50.57.226.31",  "ports": {...} }, //zone2
      {"host":"50.57.223.120", "ports": {...} }, //zone3
      {"host":"50.57.223.169", "ports": {...} }, //zone4
]}
```

Código 51. Arreglo de Servidores de Zona disponibles para unirse a una cuadrícula de zonificación

De este modo, cuando `zone0` envió la solicitud de zonificación al ejecutar el método `splitWorld(2,2,webServer)`, el Servidor Web Coordinador ya estaba preparado para procesar dicha solicitud con el archivo 'gridify.js' y determinar si existían servidores disponibles para participar según el descriptor 'servers.json', el cual ya hacía referencia a los servidores `zone1-2-3-4` en ejecución.

Así, el Servidor Web Coordinador pudo configurar una cuadrícula 2x2 sobre el modelo inicial, establecer las relaciones entre servidores para formar la cuadrícula, agrupar los elementos del modelo según la zona a la que iban a ser enviados, y contactar a los servidores `zone1-2-3-4` para invitarlos a hacer parte del proceso de zonificación, que a su vez aceptaron la invitación y se dispusieron a unir sus fronteras con los vecinos correspondientes como se muestra en las salidas '`Glue complete:`' de la Figura 55.

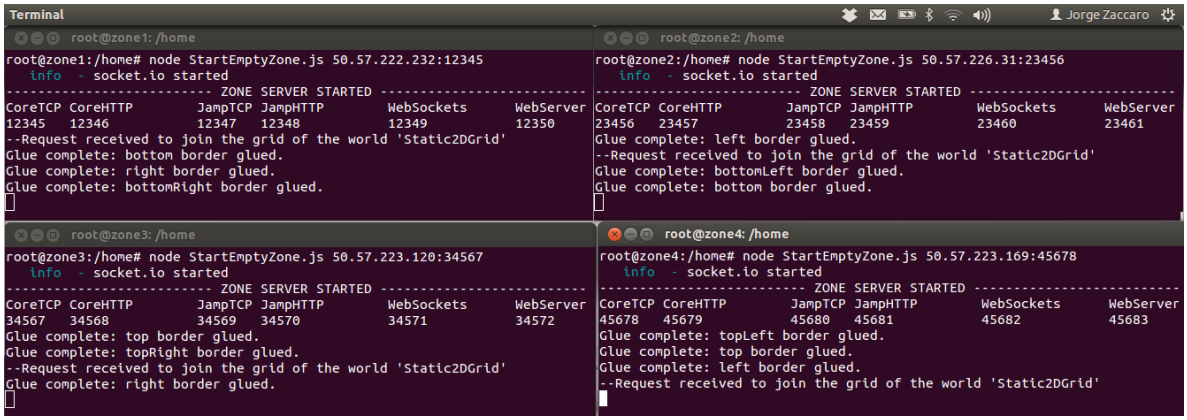

Figura 55. Salida de las consolas de los servidores de zona 1-4 al unirse a la zonificación solicitada

Posteriormente, el Servidor Web Coordinador procedió a notificar a los servidores `zone1-2-3-4` sobre los elementos que debían solicitar a `zone0` según la zonificación efectuada, y estos procedieron a obtener los elementos correspondientes y guardarlos en su sistema de archivos como se muestra en la Figura 56, la cual constituye la prueba maestra de que la zonificación (Caso de Uso #3) se llevo a cabo exitosamente.

Filename	Filesize
..	
bug.png	2.7 KB
house2.png	2.9 KB
spider.png	731 B
userPurple.png	2.1 KB
4 files. Total size: 8.2 KB	

Filename	Filesize
..	
football.png	7.8 KB
house1.png	3.1 KB
userGray.png	1.6 KB
userRed.png	1.6 KB
4 files. Total size: 14.0 KB	

Filename	Filesize
..	
basketball.png	6.8 KB
carGray.png	10.8 KB
userBlue.png	1.6 KB
warehouse.png	4.0 KB
4 files. Total size: 23.1 KB	

Filename	Filesize
..	
baseball.png	6.7 KB
house3.png	3.7 KB
policeMan.png	1.8 KB
userYellow.png	1.5 KB
4 files. Total size: 13.6 KB	

Figura 56. Archivos de los Servidores de Zona 1-4 después de participar en la zonificación

La Figura 57 muestra el resultado gráfico de la zonificación solicitada por el cargador inicial del modelo (`zone0`) y efectuada con los servidores `zone1-2-3-4`, que puede ser utilizada para verificar si los archivos listados en la Figura 56 corresponden a los que especifica la cuadrícula 2x2 resultante.

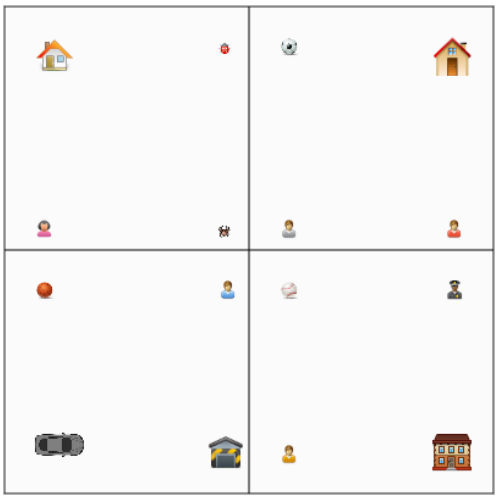

Figura 57. Representación gráfica del modelo después de ser zonificado con una cuadrícula 2x2

7.1.4 Caso de Uso #4: Un usuario solicita la página web de despliegue del entorno

Con el entorno virtual zonificado en ejecución, se procedió a solicitar la página de despliegue del entorno ingresando la ruta '50.57.222.231/Load2DGrid.html' en el navegador web Google Chrome, con lo que se obtuvo el documento HTML que se muestra en el Segmento de Código 52 con la información de contacto del servidor `zone1`, el archivo 'mvc.js' y el elemento `<canvas>` para el despliegue gráfico del entorno.

```
<html>
<head>
  <meta iohost="50.57.222.232" ioport="12349" />
  <script src="http://50.57.222.232:12350/mvc.js"></script>
  <link href="_css/main.css" type="text/css"/>
</head>
<body>...<canvas id="canvas"></canvas>...</body>
</html>
```

Código 52. Página de despliegue con la información de contacto del Servidor de Zona 1

7.1.5 Caso de Uso #5: Un usuario contacta a un servidor de zona para ingresar a su zona del entorno virtual

Una vez se obtuvo la página de despliegue y el archivo 'mvc.js' referenciado en esta, se extrajo la información de contacto del servidor `zone1` embebida en la etiqueta `<meta>` y se inició la configuración de la conexión WebSocket como se muestra en el Segmento de Código 53. A través de estas operaciones, el navegador web pudo contactar al Servidor de Zona 1 para solicitar el archivo 'socket.io.js' e ingresar a su zona del entorno virtual enviando el mensaje `comm.emit('login',…)`.

```
io = document.createElement('script');
io.setAttribute('type','text/javascript');
io.setAttribute('src',
      "http://50.57.222.232:12349/socket.io/socket.io.js";
io.onload = function() {
  comm = io.connect("http://50.57.222.232:12349/commands");
  comm.on('connect', function() { comm.emit('login',…); …});
};
```

Código 53. Establecimiento de la conexión WebSocket con el Servidor de Zona 1

El cuadrante izquierdo de la Figura 58 muestra el entorno virtual que se obtuvo al utilizar la página de despliegue del Segmento de Código 52 para conectarse al servidor `zone1` del entorno y realizar el despliegue gráfico en pantalla, en la que se puede apreciar que se dispone de un avatar de color verde para recorrer el entorno virtual y de los elementos contenidos en la zona 1 como se esperaba de la Figura 57.

Por otro lado, en el cuadrante derecho de la misma figura se presenta la salida que se obtuvo de la consola del servidor `zone1` cuando se solicito el acceso al entorno virtual con el Segmento de Código 52, en la que se confirmó el establecimiento de la conexión WebSocket con el navegador web ('`info - handshake authorized ...`'), y en consecuencia, la satisfacción del Caso de Uso #5.

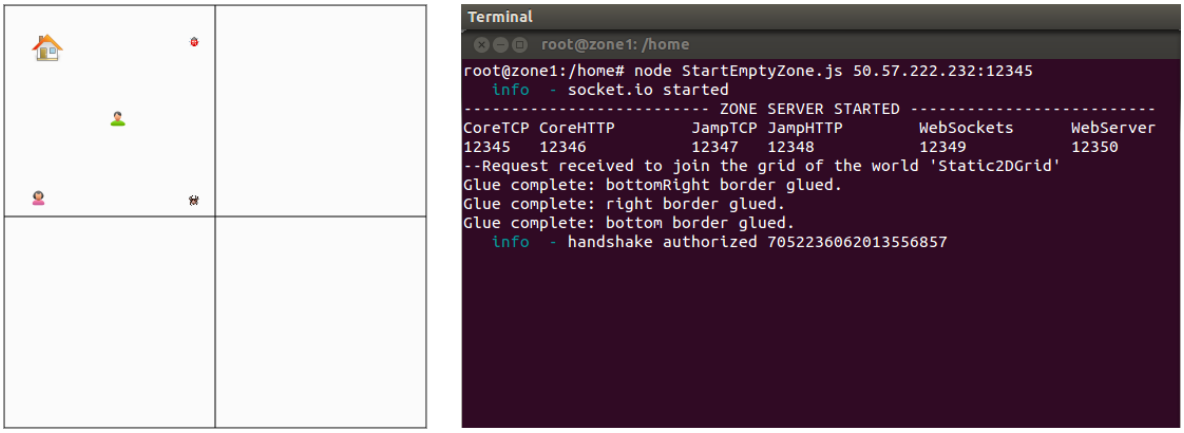

Figura 58. Ingreso de un usuario a la zona 1 del entorno virtual de prueba

7.1.6 Caso de Uso #6: Un usuario se mueve dentro de una misma zona

Después de acceder al entorno virtual, se procedió a ingresar comandos de desplazamiento (i.e. flechas del teclado) para recorrer la zona 1 como se muestra en la Figura 59:

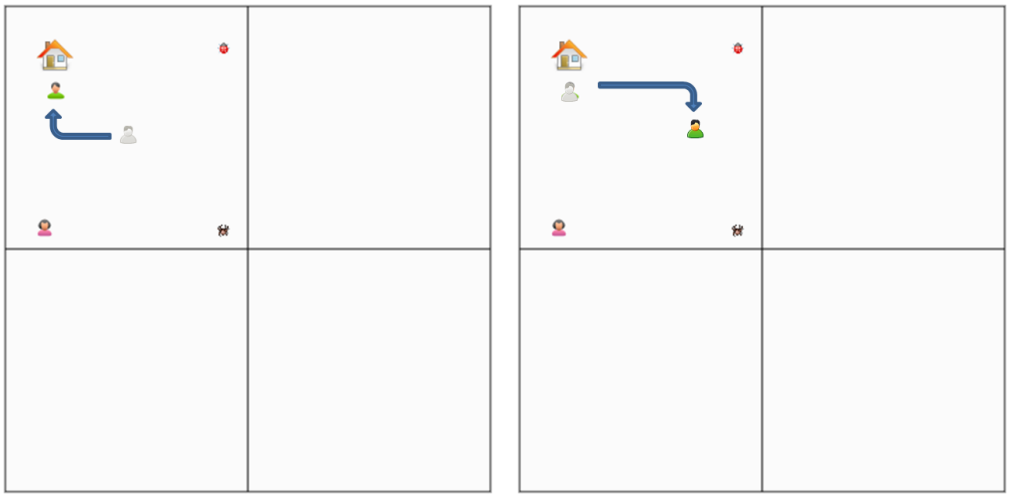

Figura 59. Desplazamiento del avatar en la zona 1

Dado que este el procesamiento de comandos de desplazamiento de los usuarios fue implementado utilizando una política de cálculos predictivos en el Cliente Web para evitar retardos de red visibles, se pudo observar que el despliegue gráfico del entorno se realizó con la continuidad y fluidez esperadas, satisfaciendo así el Caso de Uso #6.

Sin embargo, cabe aclarar que esta política no contempla la realización de correcciones para contrarrestar posibles intentos de fraude de los usuarios, lo cual implica que el sistema confía plenamente en los datos recibidos de los Clientes Web, y que naturalmente habrá que modificarlo para crear aplicaciones en las que esto no sea aceptable.

7.1.7 Caso de Uso #7: Un usuario observa elementos ubicados en zonas vecinas

Continuando con el desplazamiento Caso de Uso #6, se procedió a acercar el avatar a la frontera entre las zonas 1 y 2 como se muestra en el cuadrante izquierdo de la Figura 60, y a acercarlo a la esquina inferior derecha de la zona 1 como se muestra en el cuadrante derecho de la misma figura.

En esta figura se puede apreciar que al llegar el avatar a cierta distancia de la frontera con la zona 2, aparecieron en pantalla dos elementos cercanos a la frontera entre las zonas 1 y 2 (balón de fútbol, avatar gris), y que al acercarse a la esquina inferior derecha de la zona 1 aparecieron otros 3 elementos cercanos a las fronteras con las zonas 3 (balón de baloncesto, avatar azul) y 4 (pelota de beisbol), lo cual verificó el cumplimiento del Caso de Uso #7.

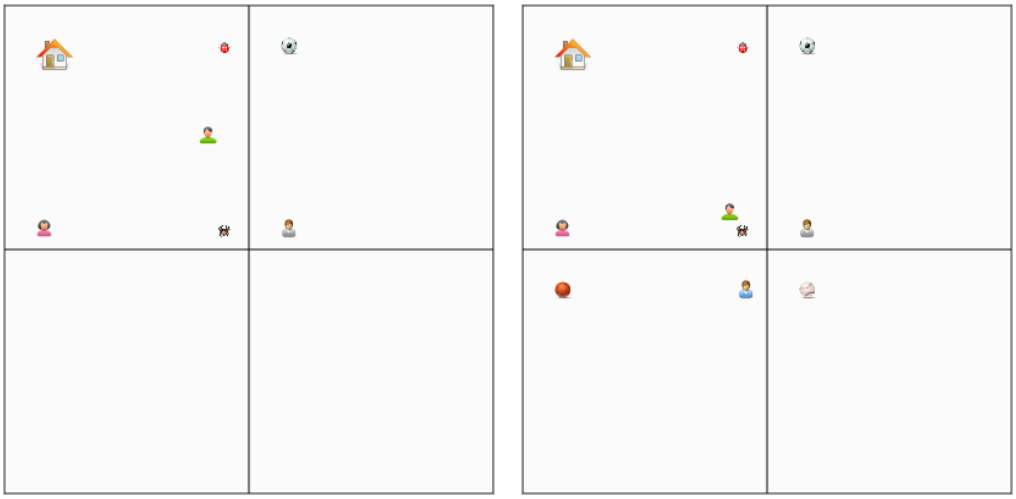

Figura 60. Visibilidad en las fronteras con las zonas 2,3 y 4

7.1.8 Caso de Uso #8: Un usuario cruza una frontera entre zonas vecinas

Finalmente, se procedió a recorrer las trayectorias de desplazamiento ilustradas en la Figura 50 (Sección 6.3) y a observar las salidas correspondientes en las consolas remotas de los Servidores de Zona 1, 2, 3 y 4.

La Figura 61 muestra el recorrido cíclico de cruces perpendiculares de fronteras que se realizó para verificar la ejecución de las operaciones de transferencia anticipada y aplazamiento de la cesión de control de Zone.js utilizando el servidor JAMP. En esta se puede apreciar que, a medida que el avatar se acercó e ingresó a las diferentes zonas, los elementos de las fronteras o zonas respectivas fueron agregados y desplegados en pantalla, por lo que al completar el recorrido 1-2-4-3-1 se consiguió poblar completamente el modelo local con todos los datos del entorno virtual zonificado.

Las salidas de las consolas de los Servidores de Zona 1 al 4 generadas al realizar este recorrido se muestran en la Figura 62, con `zone1` en la consola superior izquierda, `zone2` en la consola superior derecha, `zone3` en la consola inferior izquierda, y `zone4` en la consola inferior derecha.

Se puede apreciar, por ejemplo, que el cruce de la frontera entre la zona 1 y 2 hizo que `zone1` realizara una reserva del avatar en `zone2` (“Booking user ‘greenUser’ to right zone”), que `zone2` reportara el establecimiento de una nueva conexión WebSocket (“info – handshake authorized …”) junto a la nueva reserva y el tiempo de transferencia empleado (“New book-in: … Transfer time: …”), y que 839 milisegundos después de realizar la reserva (“Hysteresis crossing time: …”) `zone1` cediera el control del avatar a `zone2` (“Checking ‘greenUser’ in right neighbor”). Algo similar sucedió con los cruces de frontera de la zona 2 a la zona 3, de la zona 3 a la zona 4 y de la zona 4 a la zona1.

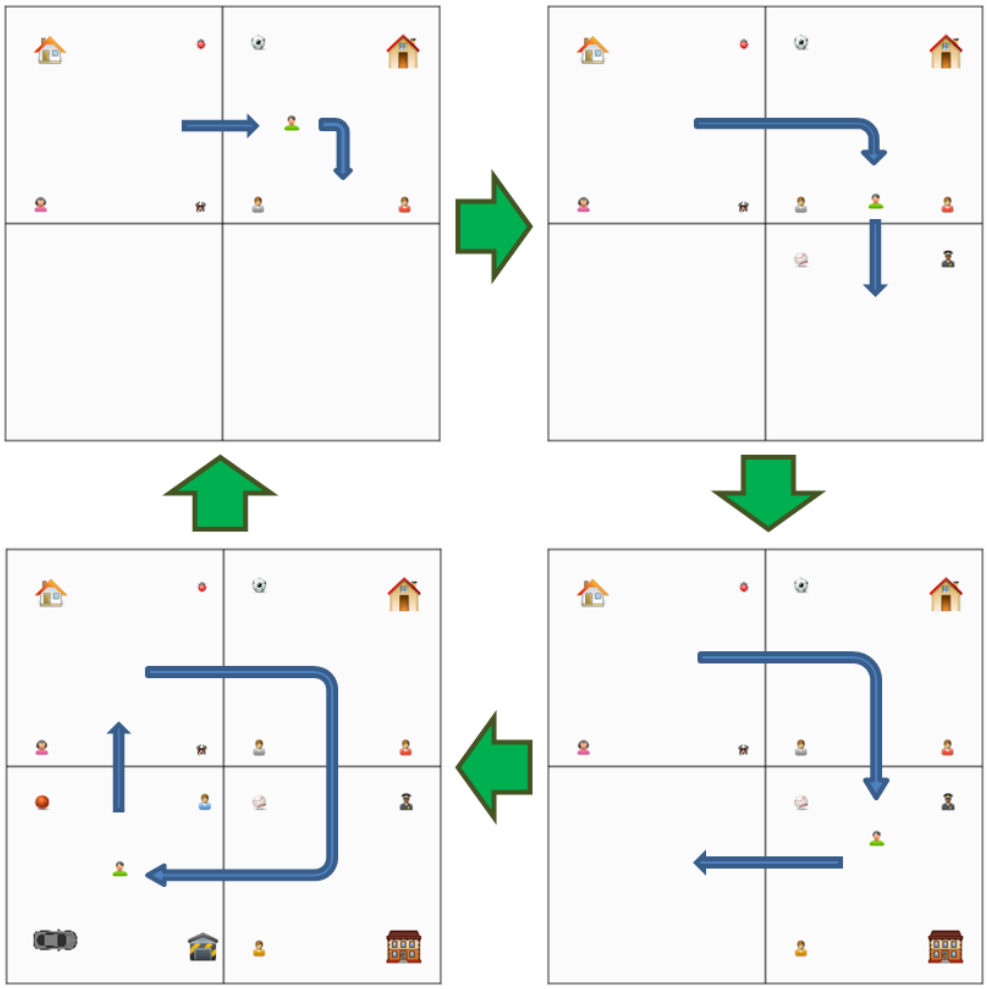

Figura 61. Cruces perpendiculares de fronteras entre zonas según la secuencia 1-2-4-3-1

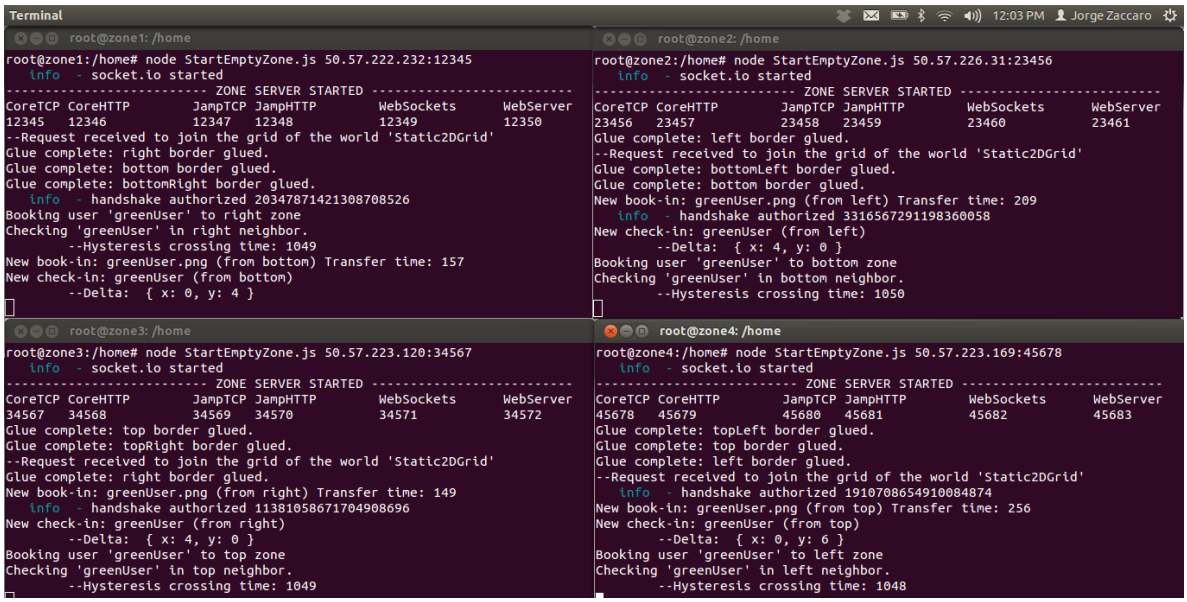

Figura 62. Salidas de las consolas de los Servidores de Zona con la secuencia de cruces 1-2-4-3-1

Por otro lado, la Figura 63 muestra el recorrido diagonal que se realizó para verificar el funcionamiento de las operaciones de transferencia anticipada y aplazamiento de la cesión de control de Zone.js al cruzar una frontera de esquina (entre zonas con relaciones diagonales como “topLeft” o “bottomRight”). En esta se puede apreciar que los elementos de las fronteras de las zonas 2 y 3 fueron agregados y desplegados en pantalla, y que al completar el recorrido 1-4 se consiguió poblar completamente el modelo de la zona 4.

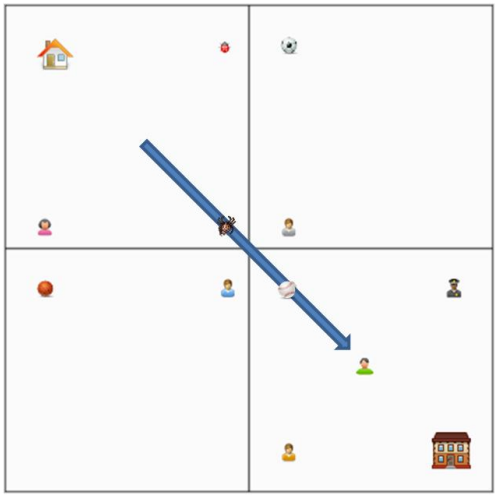

Figura 63. Cruce diagonal de frontera de esquina entre la zona 1 y la zona 4

En correspondencia, la Figura 64 muestra las salidas de las consolas de los servidores `zone1–2-3-4`, en la que se puede apreciar que `zone1` realizó una reserva del avatar en cada uno de los servidores vecinos (“right, bottom, bottomRight”) y que cada uno de ellos (`zone2`, `zone3`, `zone4`) estableció una conexión WebSocket con el navegador web y reportó el tiempo de transferencia empleado, pero que finalmente el control del avatar fue cedido a `zone4` como se confirmó con el mensaje “New check-in: (from topLeft)”

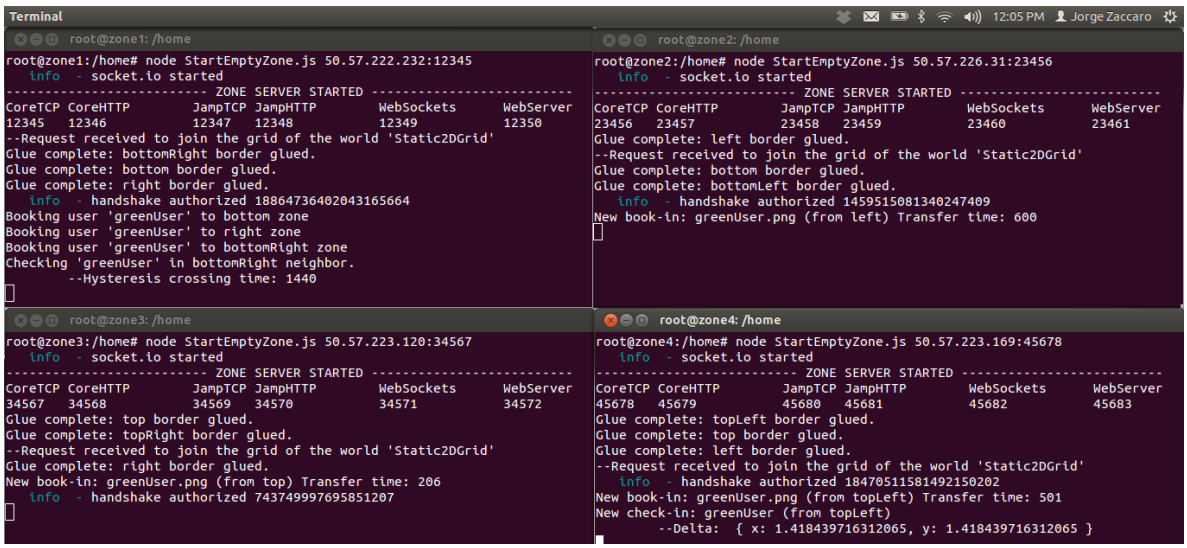

Figura 64. Salidas de las consolas de los Servidores de Zona con el cruce diagonal de 1 a 4

En el caso de la trayectoria en forma de ‘U’ ilustrada en el cuadrante inferior izquierdo de la Figura 60, las salidas de las consolas de los servidores (Figura 65) confirmaron las reservas que se esperaba realizar en `zone3` y `zone4`, y que el control del avatar fue cedido de `zone1` a `zone2` (“New check-in: (from left)”) como se había especificado.

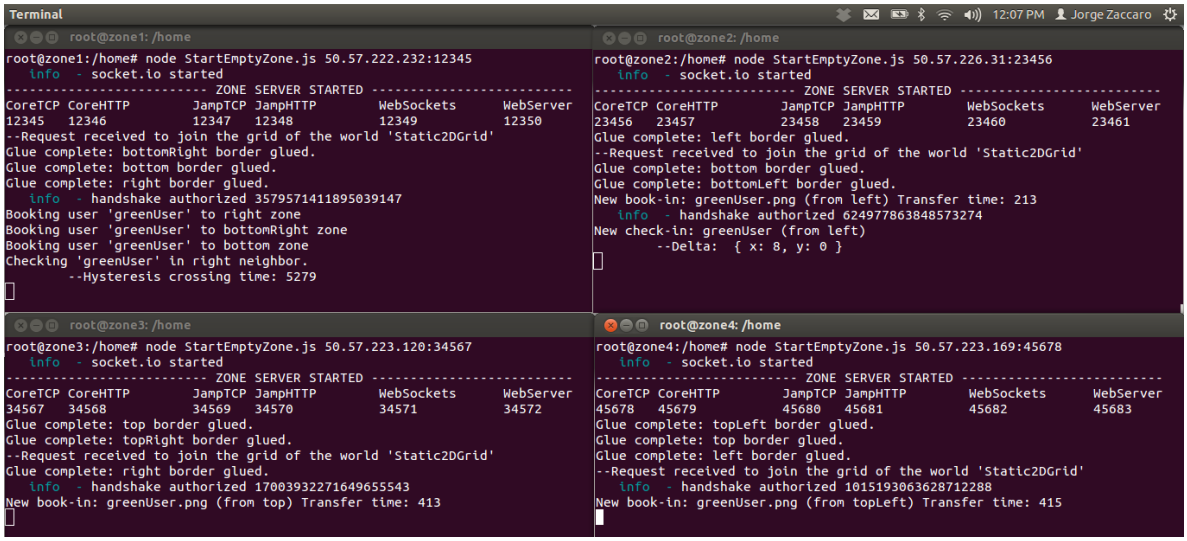

Figura 65. Salidas de las consolas de los Servidores de Zona con el cruce en forma de U de 1 a 2

Para terminar, como última prueba para afirmar con suficiente criterio que se satisfizo el Caso de Uso #8 se recorrió la trayectoria en forma de escalón ilustrada en el cuadrante inferior derecho de la Figura 50, con la que se obtuvieron las salidas de consola mostradas en la Figura 66, en la que se pueden observar las reservas realizadas en todas las zonas como en el caso del cruce diagonal, pero con la diferencia de la reserva del avatar realizada por `zone4` en `zone2` inmediatamente después de recibir el control del mismo de `zone1`.

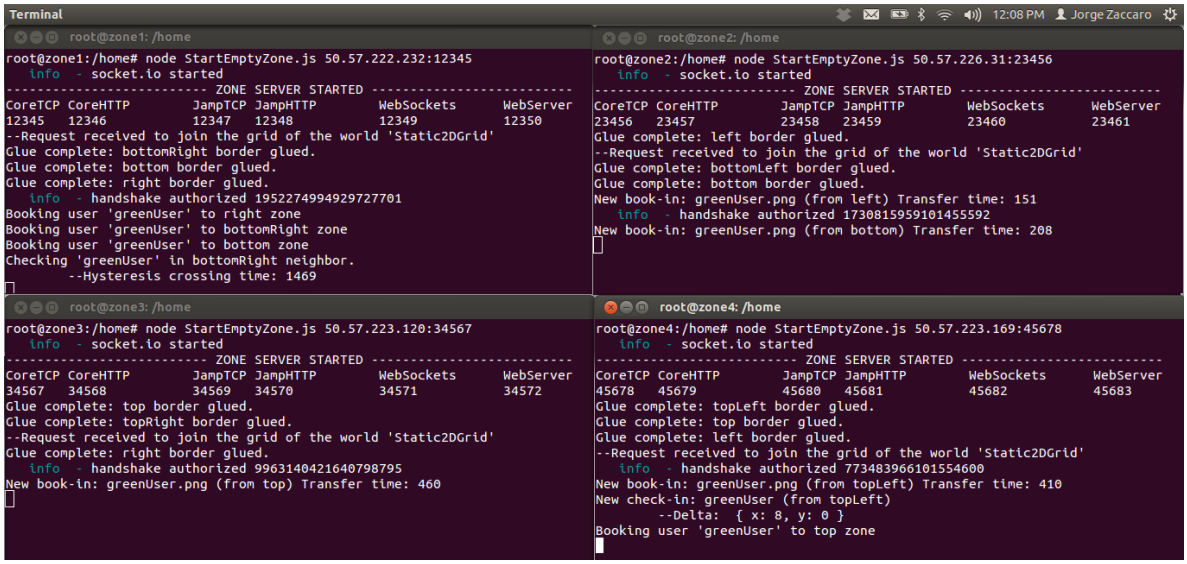

Figura 66. Salidas de las consolas de los Servidores de Zona con el cruce en escalón de 1 a 4

7.2 ANÁLISIS CUANTITATIVO

Luego de verificar la operación del sistema desarrollado desde un enfoque funcional, resulta necesario realizar un análisis cuantitativo de las variables involucradas en las operaciones del sistema para corroborar la utilidad de las estrategias propuestas.

Cabe recordar que en los cruces de frontera, el mensaje de cesión de control (*check-in*) de un avatar enviado por un Servidor de Zona de partida contiene la posición y las fuerzas aplicadas sobre el modelo del avatar, por lo que el Servidor de Zona de destino empezará a desplazar el avatar apenas reciba el control de este, hasta que el Cliente Web correspondiente se registre en la zona e indique la posición en la que se encuentra.

Al recibir la posición de ingreso de un avatar migrante, un Servidor de Zona procederá a actualizar la posición del avatar en el entorno, generando un cambio abrupto de posición que se traducirá en las mencionadas discontinuidades espaciales de los cruces de frontera. Sin embargo, como se mostrará a continuación, la estrategia de transferencia anticipada de datos (*book-ins*) propuesta en este trabajo reduce la magnitud de estas discontinuidades en comparación con los resultados que se habrían obtenido sin una estrategia anticipativa.

En primer lugar, retomando las salidas de las consolas de los Servidores de Zona 1 a 4 de la Figura 62 causadas por la secuencia de cruces de frontera 1-2-4-3-1, se puede notar la presencia de los mensajes “Transfer time: ”, “Hysteresis crossing time: ” y “Delta: {x: …, y: …}” que indican, respectivamente, el tiempo de transferencia anticipada de los datos del avatar, el tiempo entre la reserva y la cesión de control, y la magnitud de la discontinuidad espacial presentada al actualizar la posición del avatar con la recibida del Cliente Web.

La Figura 67 muestra la comparación entre la ubicación temporal y duración de las operaciones de cruce de frontera involucradas en el protocolo JAMP (anticipativas) de la Figura 62, y de las mismas operaciones suponiendo que se realizaran en el momento de cesión de control (no anticipativas), utilizando un avatar con una velocidad de desplazamiento de 60 unidades por segundo y un margen de histéresis de 30x2.

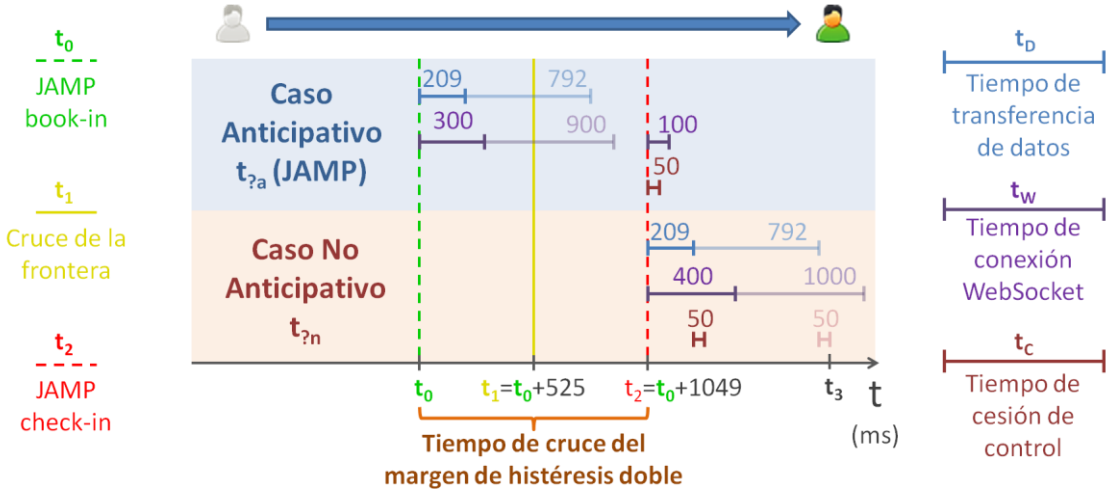

Figura 67. Ventajas en el tiempo de la estrategia anticipativa del sistema

En la parte superior de esta figura (franja azul claro) se muestra el tiempo t_{Da} empleado en la transferencia de datos del avatar de 270 KB (línea azul de 209 ms), el tiempo t_{Wa} supuesto para el establecimiento de la conexión WebSocket (línea morada de 300 ms) y el tiempo t_{Ca} supuesto para la cesión de control al Servidor de Zona de destino (línea vino tinto de 50 ms) en un cruce de frontera de la zona 1 a la zona 2 manejado con el protocolo JAMP de forma anticipativa desde el instante t_0 (30 unidades antes de la frontera).

En contraste, la parte inferior de la misma figura (franja rosada) muestra las mismas medidas de tiempo pero a partir del instante de cesión de control t_2 (30 unidades después de la frontera), resultando en un tiempo t_{Wn} de conexión WebSocket de 400ms (300 ms de creación + 100 ms de conmutación de zona) y en un retraso de t_{Dn} = 209 ms para iniciar la operación de cesión de control de t_{Cn} = 50 ms.

Así, se puede apreciar que en el caso anticipativo la cesión de control se completa en el instante $t_2 + t_{Ca} = t_2 + 50$ ms, mientras que en el caso no anticipativo se completa en el instante $t_2 + t_{Dn} + t_{Cn} = t_2 + 209$ ms $+ 50$ ms, donde es evidente el impacto del tamaño de los datos a transferir en el caso no anticipativo, que porcentualmente sería aún mayor si el tiempo de cesión de control fuera mucho menor a 50 ms.

De forma similar, se puede notar que en el caso anticipativo la conmutación de la conexión WebSocket del Cliente Web entre la zona 1 y la zona 2 se completa en el instante $t_2 + t_{Wa} = t_2 + 100$ ms, mientras que en el caso no anticipativo se completa en el instante $t_2 + t_{Wn} = t_2 + 300$ ms $+ 100$ ms, donde es evidente el impacto del tiempo de creación de la nueva conexión WebSocket (300 ms), pues en el último caso el Cliente Web empezó a crear la conexión con el Servidor de Zona 2 después de iniciada la cesión de control en t_2, mientras que en el primero se empezó a crear en el instante t_0 y se culminó en $t_0 + 300$ ms.

Para resaltar las ventajas de la estrategia anticipativa del protocolo JAMP, la Figura 67 incluye el tiempo de transferencia de datos de un archivo de 1 MB en la línea azul clara de 792 ms y un tiempo de establecimiento de conexión WebSocket menos optimista en la línea morada clara de 900 ms, con las que se desea mostrar que la cesión de control y la conmutación de conexión WebSocket del caso anticipativo seguirían realizándose en los instantes $t_2 + 50$ ms y $t_2 + 100$ ms sin verse impactadas por el aumento de los datos o el retardo de la conexión, mientras que en el caso no anticipativo sí se presentaría un impacto significativo al retrasar estas operaciones a $t_3 = t_2 + 792$ ms $+ 50$ ms y $t_2 + 1000$ ms.

Las ventajas enunciadas del protocolo JAMP en el dominio del tiempo, teniendo en cuenta que un Servidor de Zona de destino continuará desplazando a un avatar migrante hasta que su Cliente Web se registre e indique su posición de ingreso, se traducirán finalmente en una reducción de las discontinuidades espaciales presentadas durante los cruces de fronteras, pues los tiempos muertos de las operaciones de cesión de control serán considerablemente menores que los del caso no anticipativo.

Tomando la magnitud de la discontinuidad presentada en el cruce de frontera de la zona 1 a la zona 2 de la Figura 62 ("{x: 4, y: 0}"), se puede respaldar la suposición de los tiempos de cesión de control de 50 ms y de conmutación de conexión WebSocket de 100 ms (50 ms del mensaje del servidor al cliente, 50ms del cliente al servidor), pues a una velocidad de 60 unidades por segundo (2 unidades cada 33.3 ms) el avatar utilizado debería haber recorrido 6 unidades del eje X en 100 ms, por lo que la diferencia de 4 unidades en la discontinuidad es coherente con lo que pudo haber procesado el Servidor de Destino entre $t_2 + 50$ ms (cesión de control) y $t_2 + 100$ ms (registro del Cliente Web) al realizar una iteración de 33.3 ms (un paso de 2 unidades).

En contraste, dado que la estrategia no anticipativa tomaría el control del avatar en el instante $t_2 + 209$ ms $+ 50$ ms y recibiría el registro del Cliente Web en el instante $t_2 + 400$ ms, solo lograría procesar las fuerzas del avatar durante 141 ms (400 ms – 209 ms – 50 ms) y desplazarlo unas 8 unidades en el eje X (4 iteraciones de 2 pasos cada 33.3 ms), hasta ser informado por el Cliente Web que el avatar debería haberse desplazado 24 unidades en el mismo eje (400 ms a 60 unidades por segundo), lo cual se traduciría en una discontinuidad espacial de 16 unidades y en una reducción del 75% en el caso anticipativo (4 Unidades).

Cabe aclarar que el caso no anticipativo presentaría inconvenientes incluso si tuviera más tiempo para procesar las fuerzas del avatar por recibir el registro del Cliente Web más tarde en el tiempo (e.g. $t_2 + 1000$ ms), pues mientras desplaza al avatar con las fuerzas que portaba en el cruce de frontera el usuario podría haber ingresado comandos de desplazamiento para modificarlas, lo cual no sería notificado hasta el momento de la conmutación de la conexión WebSocket (e.g. $t_2 + 1000$ ms) y causaría una discontinuidad espacial mayor que podría incluir desfases en el otro eje.

De este modo, se puede apreciar que la estrategia de transferencia anticipada de datos del protocolo JAMP propuesto cuenta con numerosas ventajas para el manejo de los cruces de fronteras entre zonas de un entorno virtual, al reducir el tiempo de cesión de control y conmutación de conexión WebSocket y crear un margen de tolerancia para la transferencia de archivos pesados y los posibles retardos en la creación de las conexiones WebSocket con los Servidores de Zona de destino.

El análisis de tiempos y discontinuidades presentado aplica similarmente para los resultados obtenidos en las Figura 64, 66 y 67. Sin embargo, cabe resaltar la diferencia en los tiempos de transferencia de los datos en las reservas, pues se puede notar que en una de las zonas el tiempo es mucho menor que en las otras dos (e.g. Figura 64: zona3=206, zona4=501, zona2=600), lo cual se debe a que el Servidor de Zona de partida debe transferir los datos a más de un Servidor de Zona vecino simultáneamente.

Este hecho, junto a la carga que impondrían múltiples usuarios, debe ser tenido en cuenta al calcular o estimar los márgenes de proximidad de un entorno virtual dado con las ecuaciones de la Sección 5.1 (Justificación Teórica), de modo que los tiempos de anticipación resultantes sean suficientes para realizar exitosamente las operaciones del protocolo JAMP en los cruces de fronteras entre zonas.

7.3 REQUERIMIENTOS FUNCIONALES

Después de confirmar la satisfacción de los Casos de Uso y demostrar cuantitativamente las ventajas de las estrategias diseñadas, vale la pena recapitular brevemente la forma en que se satisfizo cada uno de los requerimientos funcionales del sistema.

7.3.1 Almacenamiento de datos

Mediante el uso del módulo `fs` de Node.js, Zone.js almacena los datos de un entorno virtual en el sistema de archivos subyacente (Caso de Uso #1), escribiendo el archivo de descripción 'model.json' y copiando sus elementos a la carpeta 'elements'.

7.3.2 Zonificación

Con el archivo 'gridify.js' el Servidor Web Coordinador del sistema puede recibir las solicitudes de zonificación (Caso de Uso #3), determinar los Servidores de Zona disponibles con el descriptor 'servers.js' y contactarlos para distribuir un entorno.

7.3.3 Simulación

Por medio de la invocación del método `start(…)` en un Servidor de Zona (Caso de Uso #2) y del método `splitWorld(…)` si se desea zonificar el entorno (Caso de Uso #3), Zone.js inicia la simulación centralizada o distribuida de un entorno virtual.

7.3.4 Puerta de acceso

A través del Servidor Web Coordinador los Clientes Web pueden obtener la página de despliegue del entorno y la información de contacto de las zonas (Caso de Uso #4).

7.3.5 Conexiones

Al contactar al servidor de soporte `webs` de Zone.js (Caso de Uso #5) los Clientes Web pueden establecer conexiones WebSocket con los Servidores de Zona (usando el módulo 'socket.io') para intercambiar mensajes de forma asíncrona y en tiempo real.

7.3.6 Procesamiento de comandos

Mediante el uso del módulo `serverController` de Zone.js un Servidor de Zona puede recibir los comandos enviados por el módulo `clientController` de los Clientes Web (Caso de Uso #6), que son enviados solo cuando suceden eventos de teclado y no de forma periódica para minimizar el tráfico de la interacción.

7.3.7 Actualizaciones y Visibilidad

Con el módulo `jamp` de Zone.js un Servidor de Zona puede obtener los elementos visibles de Servidores de Zona vecinos y enviarlos a los Clientes Web para que sean agregados y desplegados en pantalla (Caso de Uso #7).

7.3.8 Cruce de Fronteras

Al aplicar la estrategia de zonificación basada en cuadrantes del módulo `jamp` un Servidor de Zona puede realizar las operaciones de reservas y cesión de control necesarias para manejar los cruces de fronteras, utilizando el protocolo JAMP como mecanismo de interacción con los Servidores de Zona vecinos (Caso de Uso #8).

7.4 REQUERIMIENTOS NO FUNCIONALES

De forma similar a la sección anterior, resulta necesario discutir si el sistema diseñado e implementado cuenta con las cualidades especificadas en los requerimientos no funcionales.

7.4.1 Accesibilidad

Tal como se mostró en el Caso de Uso #4, el sistema de prueba es accesible utilizando un computador conectado a Internet y un navegador web con soporte HTML5 como Google Chrome.

7.4.2 Uso

La arquitectura de computación distribuida diseñada fue implementada como un módulo parametrizable que puede utilizarse para construir entornos divididos en un número arbitrario de zonas con atributos espaciales parametrizables.

7.4.3 Escalabilidad

Aunque no se realizaron pruebas multiusuario, la distribución del modelo a diferentes servidores se realizó satisfactoriamente, por lo que la carga de procesamiento asociada también se repartió como se esperaba.

7.4.4 Transparencia

Como se mostró en las pruebas de los cruces de fronteras, la distribución del sistema se hizo transparente para los usuarios al utilizar las estrategias de anticipación y aplazamiento condensadas en las interacciones del protocolo JAMP.

7.4.5 Fluidez

Dado que se utilizó una política de cálculos predictivos en los Clientes Web y el protocolo JAMP para los cruces de fronteras, las discontinuidades espaciales fueron minimizadas durante los cruces y resultaron imperceptibles en el despliegue gráfico.

7.4.6 Interoperabilidad

Para la implementación de Zone.js se utilizaron los estándares abiertos y tecnologías web HTML5 y WebSocket, y la plataforma de código abierto Node.js.

7.4.7 Flexibilidad

El módulo Zone.js se construyó igualmente de forma modular y parametrizable para operar con un número arbitrario de zonas, con base en los patrones arquitectónicos Multi-capas y MVC para facilitar su modificación por otros desarrolladores.

7.4.8 Desarrollo

El sistema fue programado en JavaScript, y partes de los componentes MVC (e.g. cálculos de física de ‘loop.js’) fueron compartidas entre el cliente y el servidor.

7.4.9 Integración

El sistema fue implementado como un modulo o paquete de Node.js y expuso su funcionalidad en las interfaces de los componentes MVC del cliente y el servidor.

7.5 VERIFICACIÓN DE LOS OBJETIVOS

Finalmente, resulta conveniente cerrar la Sección de Resultados con la validación del cumplimiento de los objetivos de este trabajo de investigación.

- **Objetivo General: Diseño e implementación de la arquitectura**

Con el proceso de diseño descrito en la Sección 5.2, y su correspondiente implementación en la Sección 5.3, se logró cumplir el objetivo general de este trabajo y concretar sus aportes teóricos y prácticos en el protocolo JAMP y el módulo Zone.js.

- **1. Reducción de las discontinuidades espaciales de los cruces de fronteras:**

Como se demostró cuantitativamente en la Sección 7.2, la estrategia de transferencia anticipada de datos de Zone.js implementada con el protocolo JAMP propuesto permite reducir las discontinuidades espaciales en los cruces de fronteras entre zonas en comparación con una estrategia no anticipativa, al enviar previamente los datos del avatar migrante al Servidor de Zona encargado de la posible zona de destino y ordenar al Cliente Web que empiece a establecer una conexión WebSocket con dicho servidor.

- **2. Construcción del entorno visible en las fronteras:**

Tal como se discutió en la implementación del Caso de Uso #7 en la Sección 5.3.5.7 y como se mostró en las pruebas correspondientes en la Sección 7.1.7, Zone.js permite que un Servidor de Zona construya el entorno visible de usuarios que se acerquen a las fronteras de su zona al contactar a los Servidores de Zona encargados de las zonas vecinas para solicitarles los elementos que se encuentren dentro de un margen de visibilidad determinado en alguna de sus fronteras.

- **3. Transparencia en distribución y migración de la plataforma resultante:**

Al utilizar la configuración de servidores descrita en la Sección 6.2 para realizar las pruebas de los Casos de Uso en la Sección 7.1, se demostró que tanto los desarrolladores de un entorno virtual como los usuarios del mismo no tienen que preocuparse por la distribución de la infraestructura subyacente ni las operaciones de migración involucradas en los cruces de fronteras entre zonas.

En particular, las pruebas del Caso de Uso #3 mostraron que para dividir un entorno virtual en zonas un desarrollador solo debe indicar el número de filas y columnas de la cuadrícula deseada sin preocuparse por nada más, pues de ahí en adelante Zone.js se encargará de solicitar la zonificación indicada al Servidor Web Coordinador, contactar a los Servidores de Zona necesarios y realizar las operaciones de sincronización en las fronteras.

De forma similar, las pruebas del Caso de Uso #8 mostraron que los Clientes Web ni siquiera se enteran de las operaciones de migración realizadas para transferir a sus avatars entre Servidores de Zona en los cruces de frontera, lo cual garantizó una experiencia virtual transparente y fluida como se había especificado en los objetivos del trabajo.

8. CONCLUSIONES

El trabajo de investigación que se presentó en este documento es el producto de un proceso de estudio del problema abordado, de selección de las tecnologías más apropiadas para resolverlo, de diseño e implementación de estrategias de migración y transparencia, y de realización de las pruebas necesarias para demostrar la utilidad de la plataforma desarrollada y respaldar los aportes teóricos y prácticos realizados.

El problema de la concurrencia en entornos virtuales de gran escala como video juegos en línea y mundos virtuales fue abordado utilizando una estrategia de zonificación que propone dividir la carga de procesamiento de un entorno y sus usuarios por zonas espaciales asignadas a diferentes servidores, manejando los cruces y la visibilidad en las fronteras entre zonas para brindar la ilusión de que se trata de un entorno indivisible.

El diseño de la arquitectura se basó en los patrones arquitectónicos Modelo-Vista-Controlador y Multi-capas, los cuales ayudaron a separar claramente los componentes y responsabilidades del sistema en términos de presentación, lógica, datos, coordinación de acceso y procesamiento de comandos, con el objetivo de habilitar el uso de la plataforma para la construcción de aplicaciones interactivas en la Web y la realización de modificaciones localizadas de componentes en trabajos futuros.

La implementación del sistema se realizó sobre la plataforma de código abierto Node.js, con lo cual se aprovecharon muchas de sus ventajas como el modelo de programación dirigido por eventos, que facilita la creación de aplicaciones de red escalables y de tiempo real, su lenguaje de programación JavaScript que posibilitó la compartición de código entre el cliente y el servidor, y su enfoque modular que facilitó la utilización de paquetes adicionales como ‘socket.io’ y la publicación de los aportes realizados en forma compacta.

La comunicación cliente-servidor se realizó utilizando el protocolo WebSocket, que fue seleccionado por sus considerables ventajas en términos de latencia y sobrecarga, y permitió el envío de actualizaciones en tiempo real dirigidas por el servidor que fueron necesarias construir el entorno visible de los usuarios en las fronteras y notificar los cruces de fronteras entre zonas a los Clientes Web.

El despliegue gráfico en el Cliente web se realizó a través del elemento Canvas de HTML5, con lo cual se evitó la instalación de programas adicionales o *plug-ins* como Adobe Flash y se pudo construir una plataforma compatible con los estándares Web más recientes, aspectos esenciales para la adopción de este trabajo por otros desarrolladores.

Los Casos de Uso presentados en las especificaciones del sistema permitieron enunciar de forma clara y funcional las interacciones que debía soportar con los desarrolladores y usuarios, y especificar las respuestas que debía presentar ante cada una de ellas, lo cual ayudó a implementar el sistema en etapas consecutivas y a realizar las pruebas necesarias para verificar su funcionamiento.

Los requerimientos funcionales permitieron listar y verificar el cumplimiento de las responsabilidades del sistema desarrollado en aspectos como el almacenamiento de datos, el proceso de zonificación, el manejo de las conexiones con los clientes, el procesamiento de comandos y el manejo de la visibilidad y los cruces de fronteras, mientras que los requerimientos no funcionales ayudaron a describir y comprobar las cualidades con las que debía contar el sistema en temas como el uso y la accesibilidad Web, la transparencia y fluidez, la flexibilidad e interoperabilidad y el lenguaje de programación y forma de entrega de la plataforma.

Los cálculos realizados en la justificación teórica permitieron respaldar la selección de un sistema distribuido en vez de un *cluster* para implementar dicha plataforma, la estrategia de transferencia anticipada de datos y la política de detección o predicción de los cruces de frontera, y proporcionaron las bases para diseñar la estrategia de zonificación basada en cuadrantes y desarrollar el protocolo JAMP (JavaScript Asset Migration Protocol) a partir de reservas anticipadas y operaciones ligeras de cesión de control.

La estrategia de transferencia de datos en los cruces de frontera se planteó en términos de un margen de proximidad o anticipación y un margen de aplazamiento que formaron un margen de histéresis doble, el cual debe ser calculado según las características de la red subyacente, los atributos de desplazamiento de los avatars y el tamaño de los datos a transferir, y teniendo en cuenta los posibles retardos que introducen las reservas simultaneas en varios servidores y el procesamiento de múltiples usuarios.

La detección o predicción de cruces de frontera se generalizó utilizando un sistema de cuadrantes internos que permitió determinar el momento y las zonas en las que se debía reservar un avatar que se acercara a una frontera, y se complementó con un sistema de cuadrantes externos que permitió determinar el instante y el servidor de destino al que debía ser cedido el control de un avatar que superara el margen de histéresis doble.

La arquitectura del módulo Zone.js resultante se organizó a partir del patrón Multi-Capas para separar las responsabilidades de los Clientes Web, el Servidor Web Coordinador y los Servidores de Zona del Servidor de Aplicación, y el patrón Modelo-Vista-Controlador para separar los datos de un entorno virtual de su despliegue gráfico en la Web y el procesamiento de comandos, con una división adicional de servidores de soporte para manejar las comunicaciones de un Servidor de Zona con el Servidor Web Coordinador y otros Servidores de Zona a través de conexiones TCP y con los Clientes Web a través de conexiones WebSocket.

Uno de los servidores de soporte fue utilizado para condensar los procesos de sincronización de fronteras para el manejo de los cruces y la visibilidad en el protocolo de aplicación JAMP, el cual fue propuesto para realizar la transferencia anticipada de los datos de un avatar migrante en las fronteras a partir de mensajes TCP de reserva (*book-in*) y cesión de control (*check-in*) utilizados para coordinar la solicitud HTTP de los datos involucrados y el registro oficial de la migración.

Para realizar las pruebas del sistema, se configuró un conjunto de seis servidores creados en una plataforma de computación en la nube para desempeñar las responsabilidades de un Servidor Web Coordinador, un Servidor de Zona inicial para cargar el modelo y solicitar una zonificación, y cuatro Servidores de Zona encargados de llevar a cabo dicha zonificación.

Al probar los Casos de Uso en el mismo orden que se enunciaron, se logró cargar un modelo de 16 elementos en el Servidor de Zona inicial, dividir el modelo en una cuadrícula de 2x2 zonas a través del Servidor Web Coordinador, acceder al entorno a través de un navegador web con capacidad HTML5 y recorrerlo con trayectorias perpendiculares y diagonales en las que se comprobó la operación de las estrategias de visibilidad en las fronteras y los cruces de fronteras entre zonas.

Los resultados cuantitativos de las pruebas de los cruces de frontera demostraron las ventajas del protocolo JAMP propuesto para el manejo de los cruces en comparación con una estrategia no anticipativa, al reducir significativamente las discontinuidades espaciales presentadas durante los cruces de frontera y el tiempo de cesión de control de un avatar entre Servidores de Zona mediante la transferencia anticipada de sus datos y el establecimiento previo de la conexión WebSocket entre el Cliente Web y el Servidor de Zona de destino.

Los objetivos generales y específicos planteados fueron satisfechos a cabalidad, pues se consiguió una reducción en las discontinuidades espaciales presentadas durante los cruces de fronteras entre zonas, se garantizó la construcción del entorno visible de los usuarios en las fronteras, y se alcanzó la transparencia en distribución y migración esperadas en la interfaz de programación y la usabilidad de la plataforma web desarrollada.

Finalmente, teniendo en cuenta la verificación de los Casos de Uso, la satisfacción de los requerimientos funcionales y no funcionales, las ventajas cuantitativas de las estrategias propuestas, y el cumplimiento de los objetivos generales y específicos, se puede afirmar que la arquitectura de computación distribuida diseñada y la plataforma web implementada cumplieron con las expectativas que estructuraron este trabajo de investigación, y realizaron aportes concretos con el protocolo de aplicación JAMP y el módulo Zone.js que han de resultar útiles para el desarrollo de entornos interactivos en la Web 3D.

9. TRABAJO FUTURO

La arquitectura de computación distribuida diseñada y la plataforma Zone.js implementada en este trabajo de investigación constituyen un punto de partida para el desarrollo de aplicaciones interactivas 3D en la Web más complejas al proporcionar las herramientas básicas necesarias para construir y operar entornos virtuales zonificados utilizando tecnologías y estándares abiertos como HTML5, WebSockets y Node.js.

Si bien se cumplieron los objetivos propuestos en el trabajo, existen numerosas áreas en las que se debería seguir trabajando si se deseara convertir a Zone.js en un motor de videojuegos distribuido (*distributed game engine*), entre las que se encuentran la adaptación de una librería de física como JigLibJS2 para soportar aspectos como colisiones, la creación de una interfaz WYSIWYG (*What You See Is What You Get*) o un entorno integrado de desarrollo (IDE) en la Web, o la implementación de un sistema de alojamiento Web 3D que permitiera a los desarrolladores de videojuegos y mundos virtuales almacenar y operar sus aplicaciones en la nube.

En cuanto a las pruebas de la plataforma, sería conveniente implementar un sistema de pruebas de estrés que simule la interacción de miles de usuarios en un entorno virtual zonificado para monitorear las respuestas del sistema y determinar los cambios necesarios para hacer que la plataforma sea más robusta y tolerante a fallas.

Con respecto al proceso de zonificación, resultaría de gran utilidad desarrollar una estrategia de dinámica que manejara el problema de la concurrencia en tiempo de ejecución, dividiendo un entorno en más o menos zonas dependiendo de la carga de usuarios que tenga en un momento determinado, para lo cual debería estar en la capacidad de migrar parte de su modelo sin interrumpir la simulación ni las interacciones con el entorno.

Finalmente, lo más provechoso para divulgar los aportes de este trabajo de investigación sería desarrollar un videojuego o mundo virtual distribuido en la Web que permitiera demostrar la funcionalidad ofrecida en forma práctica y aplicada a la industria actual del entretenimiento interactivo y la naciente industria de la navegación Web 3D.

10.BIBLIOGRAFÍA

[1] World Wide Web Consortium, "HTML5 Specification." [Online]. Available: http://dev.w3.org/html5/spec/Overview.html. [Accessed: 29-Feb-2012].

[2] Khronos Group, "WebGL Standard." [Online]. Available: http://www.khronos.org/webgl/. [Accessed: 29-Feb-2012].

[3] Mozilla Developer Network, "JavaScript Language." [Online]. Available: https://developer.mozilla.org/en/JavaScript. [Accessed: 29-Feb-2012].

[4] Internet Engineering Task Force, "The WebSocket Protocol." [Online]. Available: http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17. [Accessed: 29-Feb-2012].

[5] "Grand Theft Auto 4's Liberty City Map." [Online]. Available: http://s2.n4g.com/media/11/news/115000/115112_0.jpg. [Accessed: 29-Feb-2012].

[6] J. Zaccaro, "Elaboración propia con iconos tomados de FreeIconsWeb.com." .

[7] Linden Lab, "Second Life - Sin Crossing Problems." [Online]. Available: http://jira.secondlife.com/browse/SVC-472. [Accessed: 29-Feb-2012].

[8] "Cellular Network Handovers." [Online]. Available: http://upload.wikimedia.org/wikipedia/commons/e/ee/Frequency_reuse.svg. [Accessed: 29-Feb-2012].

[9] R. Dahl, "Node.js." [Online]. Available: http://www.nodejs.org. [Accessed: 29-Feb-2012].

[10] D. Herron, *Node Web development : a practical introduction to Node, the exciting new server-side JavaScript Web development stack*. Birmingham, U.K.: Packt Pub., 2011.

[11] "NPM: Node Package Manager." [Online]. Available: http://www.npmjs.org.

[12] World Wide Web Consortium, "The WebSocket API." [Online]. Available: http://dev.w3.org/html5/websockets/. [Accessed: 29-Feb-2012].

[13] "Web Sockets - A Quantum Leap in Scalability for the Web." .

[14] "Socket.io." [Online]. Available: http://www.socket.io. [Accessed: 29-Feb-2012].

[15] "Game Developers Conference," 2011. [Online]. Available: http://www.gdconf.com/. [Accessed: 29-Feb-2012].

[16] R. Cabello, "Mr. Doob's Three.js JavaScript 3D Engine." [Online]. Available: https://github.com/mrdoob/three.js/. [Accessed: 29-Feb-2012].

[17] I. Gorton, *Essential software architecture*. Berlin; Heidelberg; New York: Springer, 2006.

[18] M. Fowler, *UML distilled applying the standard object modeling language*. Boston, MA: Addison-Wesley, 2004.

[19] "EVE Online: 30,000 Users on One Server Shard," 2006. [Online]. Available: http://www.datacenterknowledge.com/archives/2006/09/13/eve-online-30000-users-on-one-server-shard/. [Accessed: 29-Feb-2012].

[20] Gamers Nexus, "Study: Fastest Average Broadband Speeds in the US by City." [Online]. Available: http://www.gamersnexus.net/guides/796-study-fastest-download-speeds-in-the-us. [Accessed: 25-Sep-2012].

[21] Techspot, "Intel Core i7-3820 Processor Review." [Online]. Available: http://www.techspot.com/review/492-intel-core-i7-3820/page3.html. [Accessed: 25-Sep-2012].

[22] Magarnigal, "WebGL Morph Targets Quake 2 Ogro." [Online]. Available: http://www.webgl.com/2012/04/webgl-demo-morph-targets-quake-2-ogro/. [Accessed: 25-Sep-2012].

[23] "Rackspace Cloud Hosting Pricing." [Online]. Available: http://www.rackspace.com/cloud/cloud_hosting_products/servers/pricing/. [Accessed: 29-Feb-2012].

[24] D. T. Ahmed and S. Shirmohammadi, "Zoning Issues and Area of Interest Management in Massively Multiplayer Online Games," in Handbook of Multimedia for Digital Entertainment and Arts, B. Furht, Ed. Boston, MA: Springer US, 2009, pp. 175–195.

[25] S. Rieche, K. Wehrle, M. Fouquet, H. Niedermayer, T. Teifel, and G. Carle, “Clustering players for load balancing in virtual worlds,” International Journal of Advanced Media and Communication, vol. 2, no. 4, p. 351, 2008.
[26] R. Prodan, V. Nae, T. Fahringer, and H. Jordan, “Dynamic Real-Time Resource Provisioning for Massively Multiplayer Online Games,” in Parallel Computing Technologies, vol. 5698, V. Malyshkin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 98–111.
[27] Tsun-Yu Hsiao and Shyan-Ming Yuan, “Practical Middleware for Massively Multiplayer Online Games,” IEEE Internet Computing, vol. 9, no. 5, pp. 47–54, Sep. 2005.
[28] Wentong Cai, P. Xavier, S. J. Turner, and Bu-Sung Lee, “A scalable architecture for supporting interactive games on the internet,” pp. 54–61.