UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA TELEMÁTICA

Trabajo Fin de Grado

Estudio y desarrollo de una aplicación web para la generación y tratamiento de datos del simulador MATSim.

Luis de la Cruz Piris
2013

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA TELEMÁTICA

Trabajo Fin de Grado

Estudio y desarrollo de una aplicación web para la generación y tratamiento de datos del simulador MATSim.

Autor: Luis de la Cruz Piris
Director: Enrique de la Hoz de la Hoz

TRIBUNAL:

Presidente: D. Iván Marsá Maestre

Vocal 1º: D. Miguel Ángel López Carmona

Vocal 2º: D. Enrique de la Hoz de la Hoz

CALIFICACIÓN:..

FECHA:..

A mi familia.

Agradecimientos

Es complicado ser capaz de recordar a todo el mundo al que tienes que agradecer haber llegado hasta aquí. Tampoco han sido todo ayudas o siempre han habido las mejores circunstancias, pero es cierto que al final se suele compensar lo bueno y lo malo.

GRACIAS A TODOS.

Resumen

Este Trabajo Fin de Grado se centra en crear una aplicación web que facilite el sistema de entrada, salida, e interpretación de los datos necesarios para trabajar con el simulador MATSim. Tras una búsqueda previa, se ha elegido el sistema de información geográfica de libre distribución OpenStreetMap para obtener los datos fuente a partir de los cuales podamos crear los escenarios de simulación. Interactuaremos con él, de forma visual, utilizando la librería JavaScript OpenLayers. Se ha optado por mostrar las animaciones de los resultados obtenidos mediante el elemento CANVAS de HTML5, lo que proporciona un alto grado de compatibilidad con cualquier sistema que utilice un navegador web moderno. La aplicación ha sido implementada utilizando funciones JavaScript en la parte del cliente, y Servlet y funciones Java en la parte del servidor.

Palabras clave: MATSim, OpenStreetMap, OpenLayer, HTML5, Java.

Abstract

This Bachelor's Degree Final Project is focused on the creation of a web application to simplify the system of input, output and the interpretation of data necessary in order to work with the MATSim simulator After a previous search, the free geographic information system OpenStreetMap has been selected to obtain the source data, from which we would be able to create simulation scenarios. We will interact with the system, in a visual way, by using the JavaScript OpenLayers library. It has been chosen to show the animations of the results obtained by the HTML5 CANVAS element, which provides a high degree of compatibility with any system using a modern web browser. The application has been implemented using JavaScript functions at the client-side and both Servlet and Java functions at the server-side.

Keywords: MATSim, OpenStreetMap, OpenLayer, HTML5, Java.

Resumen extendido

MATSim es un simulador de tráfico de vehículos desarrollado en Java y distribuido bajo Licencia Pública General de GNU (GNU GPL [1]) permitiendo su libre distribución, uso y modificación. Posibilita el poner en práctica simulaciones sobre grandes escenarios, definiendo los agentes de transporte que intervienen, e indicando características específicas que pudieran afectar al tráfico. Está diseñado de manera modular, ofreciendo la posibilidad de elegir la combinación deseada de estos módulos para obtener el resultado buscado.

En anteriores trabajos se ha estudiado en profundidad porqué elegir MATSim frente a las alternativas existentes [1], como adaptarlo a las necesidades requeridas [2], o diferentes enfoques a la hora coordinar rutas [3] [4]. Para iniciar la simulación se requieren básicamente tres tipos de fuentes de datos: el escenario a simular, los planes de los agentes que intervienen (posición origen y destino dentro del escenario, horarios, tipo de agentes,...), y los datos de configuración de la simulación. Todos ellos se deben proporcionar mediante archivos de tipo XML con las estructuras prefijadas. De igual forma los resultados obtenidos, se nos devuelven mediante archivos XML (Figura 1).

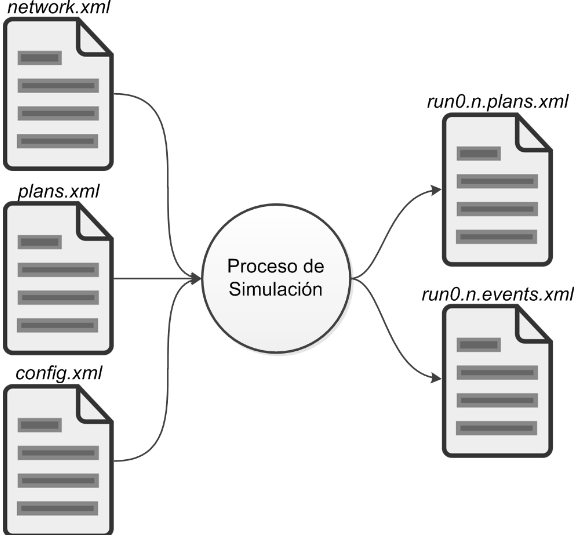


Figura 1: Sistema de archivos de entrada y salida del simulador.

[1] `http://www.gnu.org/licenses/gpl.html`

Uno de los mayores problemas que se presentaba a la hora de trabajar con este simulador, era la complejidad que existía tanto en obtener y crear los archivos de entrada, como al interpretar los archivos XML de salida.

Este trabajo tiene como objetivo general el facilitar estas tareas a través de una aplicación web que permita crear los archivos que contienen los datos de entrada de manera rápida, eficaz, y sencilla. De igual forma, a través de la aplicación, podremos interpretar los resultados obtenidos del proceso de simulación mediante la visualización de diferentes vistas obtenidas a partir de los archivos de salida. En el capítulos 1 se explican en profundidad las carencias encontradas, los objetivos fijados y el sistema final a desarrollar (aplicación web desarrollada utilizando JavaScript en el lado del cliente, y Servlet y funciones de Java en el lado del servidor).

Uno de los temas a resolver era encontrar una fuente de datos geográficos a partir de los cuales pudiéramos crear nuestros escenarios de simulación. En el capítulo 2, se dan los motivos que nos han llevado a elegir OpenStreetMap, su funcionamiento, licencia de uso y características, además hablaremos sobre la librería JavaScript OpenLayers, la cual nos permitirá interactuar desde nuestra aplicación con estos mapas de manera visual y sencilla. Describiremos el desarrollo de esta sección de de nuestra aplicación, obteniendo como resultado la parte que nos permite "navegar" por el mapa, realizar búsquedas por nombre de población y elegir el área deseada, pudiendo exportarla y convertirla a un archivo con el formato correcto para se utilización en el simulador MATSim. (Fig. 2).

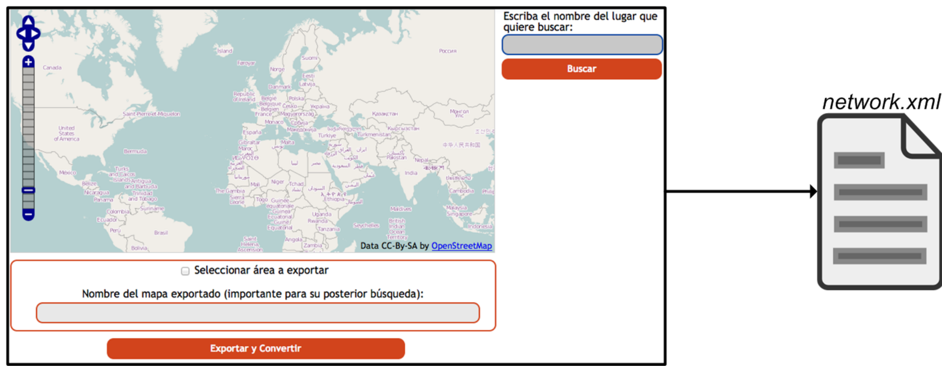


Figura 2: Creación del archivo de entrada network.xml de forma visual.

Llegados a este punto necesitamos completar los archivos de entrada requeridos para lanzar una simulación, centrándonos en el capítulo 3 en qué es necesario definir dentro de un archivo de este tipo (número de agentes, horario de inicio de rutas, tipo de rutas, zonas origen y destino,. . .) y cómo crear con ellos el archivo de planes. El usuario proporcionará todos estos valores a través de nuestra aplicación web de forma sencilla e intuitiva (Fig. 3), y nuestro sistema le devolverá el archivos de planes con el formato deseado. Analizaremos las funciones que se han desarrollado en el lado del servidor para "repartir" a los agentes por las zonas marcada de manera coherente y el desarrollo necesario para obtener el buscado archivo *"plans.xml"*. Adicionalmente se generará un segundo archivo junto al convencional de planes, que contendrá toda la información sobre los agentes y las rutas que realizan, con la intención de facilitar la búsqueda de un determinado trayecto al usuario final.

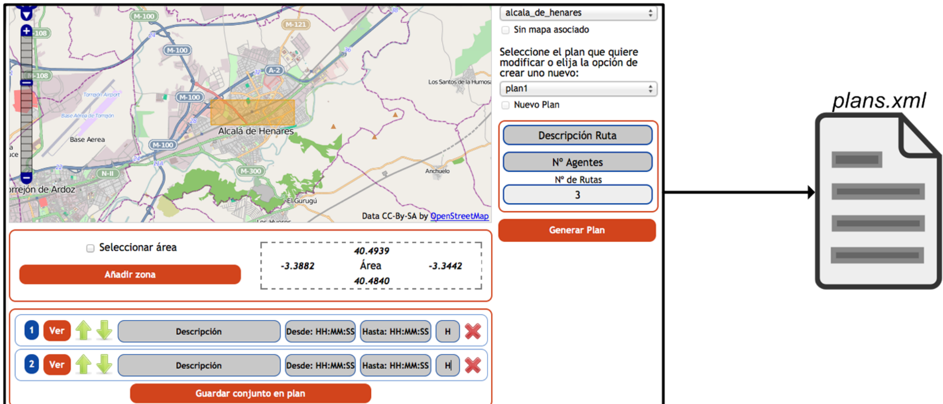


Figura 3: Elección visual de la descripción de los planes.

Todos los archivos que se han ido generando permanecen almacenados en el servidor, siendo accesible desde nuestra aplicación en cualquier momento para poder ser descargados sin necesidad de tener que repetir su proceso de generación.

Dado que dependiendo de la potencia del servidor donde se ejecute esta aplicación y del tamaño de los escenario o planes a generar, el tiempo de ejecución de las tareas puede variar, se ha dotado a aplicación con la posibilidad de visualizar el estado de cada proceso, permitiéndonos seguir realizando nuevas tareas mientras se finalizan las pendientes.

Tras obtener nuestros archivos de entrada y lanzar la simulación en MATSim, queda interpretar los resultados obtenidos. En el capítulo 4, se trata sobre este tema y la razones de compatibilidad para poder acceder a nuestra aplicación desde la mayor parte de los navegadores modernos, que nos llevaron a optar por mostrar los resultados utilizando el elementos CANVAS de HTML5. (Fig. 4)

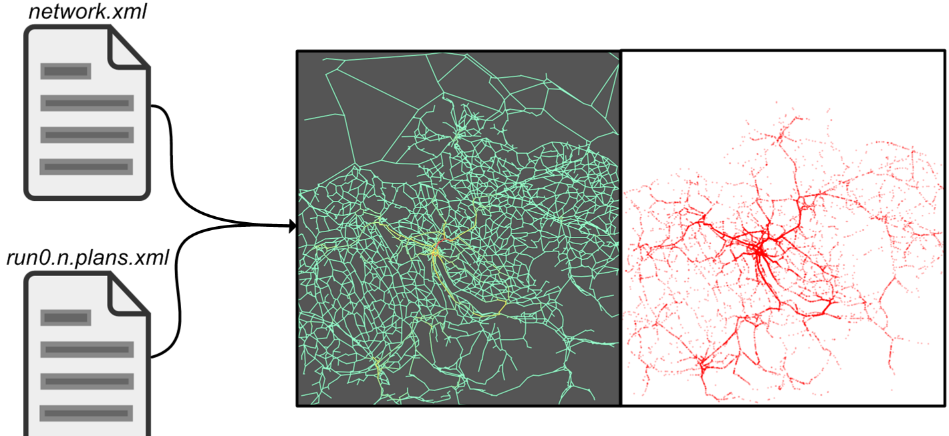


Figura 4: Visualización de los resultado a través de HTML5.

El tema de la visualización de los datos es abordado desde dos puntos de vistas, además de la parte técnica de utilización del elemento CANVAS. Por un lado ver

qué información es relevante para ser extraída de los archivos y almacenada en la base de datos, y por otro lado, una vez tengamos esa información, como tratarla y obtener representaciones realistas y útiles.

Con los datos del archivo del escenario (también es posible utilizar directamente archivos de tipo OSM-OpenStreetMap), generamos un CANVAS que representa las vías y puntos de unión de nuestra red, y sobre este primer CANVAS sobreponemos un segundo con la representación de los agentes que se desplazan en cada instante de tiempo por nuestra simulación. A partir de este concepto, existe la posibilidad de que el usuario final elija el horario que quiere visualizar, cada cuanto quiere se tomen instantáneas de lo agentes, diferentes tipos de visualización...

Aunque en este punto tenemos finalizada nuestra aplicación web, cabe destacar el apéndice A, donde se explican los conceptos básico del uso de sistemas de referencia geográfica, necesarios para entender con claridad determinadas partes de este trabajo.

Por último, se incluye un manual de usuario (apend. B) de la aplicación web donde, a través de ejemplos, se muestran todas las tareas que podemos realizar.

Índice general

Resumen iii

Abstract v

Resumen extendido vii

Lista de figuras xv

1. Introducción 1

1.1. Qué es MATSim y cómo funciona 1

1.2. Lanzar una simulación e interpretar los datos 6

1.3. Dificultades detectadas 8

1.4. Objetivos del proyecto 10

1.5. Descripción del sistema 11

2. Obtención de datos geográficos y conversión 15

2.1. Introducción 15

2.2. OpenStreetMap 18

2.2.1. Introducción 18

2.2.2. ¿Sólo mapas? 18

2.2.3. Licencia y restricciones de uso 19

2.2.4. Formato de los datos 20
2.2.5. Servicios y acceso a ellos 24
2.3. OpenLayer 27
2.3.1. Introducción 27
2.3.2. Cómo funciona 28
2.3.3. Cómo se utilizar 30
2.3.4. Cómo extraer datos 33
2.4. Funciones y comunicación entre ellas 37
2.4.1. Desarrollo de la aplicación 37

3. Generación de la población del escenario 53

3.1. Introducción 53
3.2. Funciones y comunicación entre ellas 56
3.2.1. Desarrollo de la aplicación 56

4. Procesamiento de resultados y visualización 69

4.1. Introducción 69
4.2. Funciones y comunicación entre ellas 71
4.2.1. Leer archivos xml en Java 71
4.2.2. Representación de la información 75
4.2.3. Desarrollo de la aplicación 79

5. Conclusiones finales 87

6. Futuras líneas de trabajo 89

Apéndices 93

A. Sistemas de referencia. **95**

A.1. Introducción. 95

A.2. Sistema de coordenadas WGS84 97

A.3. Proyección de Mercator 97

A.4. Correspondencia 97

B. Manual de usuario **99**

B.1. Introducción. 99

B.2. Exportar y convertir. 99

B.2.1. Nuevo desde OSM 100

B.2.2. Nuevo desde archivo 102

B.2.3. Pendientes 102

B.2.4. Convertidos 103

B.3. Generación de planes. 104

B.4. Generación de planes. 104

B.4.1. Nuevo Plan/Zona 105

B.4.2. Ver/Modificar Plan 108

B.4.3. Pendientes 109

B.4.4. Finalizados 109

B.5. Visualización de los resultados de simulación 110

B.5.1. Nuevo 110

B.5.2. Pendientes 112

B.5.3. Finalizadas 112

Índice de figuras

1. Sistema de archivos de entrada y salida del simulador. vii

2. Creación del archivo de entrada network.xml de forma visual. viii

3. Elección visual de la descripción de los planes. ix

4. Visualización de los resultado a través de HTML5. ix

1.1. Esquema Entrada/Salida MATSim. 2

1.2. Ejemplo de representación mapas/redes. 8

1.3. Visualización de datos en GoogleEarth. 9

1.4. Descripción general del sistema y principales vías de comunicación. . 12

2.1. Pasos que realiza el usuario para exportar un mapa. 17

2.2. Diagrama de la arquitectura básica de OpenStreetMap. 19

2.3. Representación a partir de los datos geográficos. 24

2.4. Ejemplo básico de uso de OpenLayer. 27

2.5. Diferentes representaciones de los mismos datos geográficos. 28

2.6. Ejemplo de uso de los *tiles*. 30

2.7. Efecto del uso de las propiedades del mapa. 33

2.8. Rectángulo dibujado una vez seleccionada un área. 36

2.9. Esquema de la funcionalidad de la pestaña “EXPORTAR”. 37

2.10. Diagrama proceso de descarga y conversión de mapas. 41

3.1. Esquema de los pasos para crear un archivos de planes. 54
3.2. Esquema de la funcionalidad de la pestaña “PLANES”. 56
3.3. Organización de las zonas dentro de una misma ruta. 57
3.4. Comunicación entre las funciones básica para generar planes. 59
3.5. Distribución de agentes sobre un área. 61

4.1. Pasos que realiza el usuario para crear una visualización. 70
4.2. Modificación de datos geográficos para variar su visualización. 75
4.3. Pasos que realiza el usuario para crear una visualización. 79
4.4. Funciones que intervienen para generar una vista y sus comunicaciones. 80
4.5. Diferentes tipos de representación a partir de los mismo datos. . . . 81
4.6. Diferentes tipos de representación de los agentes. 82

A.1. Ejemplo proyección. 96
A.2. Paso de sistema de coordenadas a proyección. 98

B.1. Contenido pestaña *Exportación*. 100
B.2. Contenido pestaña *Exportación*. 104
B.3. Contenido pestaña *Exportación*. 105
B.4. Contenido pestaña *Vista general pestaña Visualización*. 110

Capítulo 1

Introducción

Este capítulo de introducción trata sobre qué es MATSim y cómo funciona su sistema de entrada y salida de datos, el estado del arte del uso del simulador anterior a la realización de este proyecto, y de las carencias que se habían detectado en anteriores trabajos. Con todo ello, concretaremos cúales han sido los objetivos fijados, cómo se van a llevar a cabo, y cúal va ha ser el esquema general del sistema de nuestra solución.

1.1. Qué es MATSim y cómo funciona

MATSim (*Multi-Agent Transport Simulator*), es un simulador de transporte vehicular modular, capaz de trabajar con grandes escenarios y miles de agentes. Los diferentes módulos en los que está compuesto, pueden ser usados de manera individual o combinados para adaptarse a una necesidad específica. Para llevar a cabo las simulaciones, MATSim obtiene la información necesaria a través de una serie de archivos de entrada que contienen los datos básicos (el escenario y los agentes que transitan por él). Además, se pueden incluir otros archivos que aporten más datos para que el resultado final sea más fiel a la realidad. MATSim actuará siguiendo las pautas que le hayamos indicado en el archivo de configuración, y nos proporcionará a la salida los resultados a través de archivos xml que contengas los datos, imágenes, e incluso archivos específicos de determinadas aplicaciones (p.e. *.kml de Google Earth).

Al estar desarrollado íntegramente en Java el simulador se puede ejecutar en la mayoría de los sistemas operativos actuales y al distribuirse bajo la Licencia Pública General de GNU (GNU GPL [1]) permite su libre distribución, uso y modificación. Esto, unido a su carácter modular, nos facilita el poder modificar el funcionamiento interno del módulo que deseemos con nuevas implementaciones para que se comporte de la manera que nos hayamos propuesto, sin necesidad de reescribir todo el sistema.

[1] `http://www.gnu.org/licenses/gpl.html`

En proyectos anteriores [1], además de analizar en profundidad las ventajas e inconvenientes que ofrece MATSim respecto a otras soluciones (el citado diseño modular, el estar programado en Java, tipo de licencia de uso, . . .), se ha explicado el funcionamiento del simulador y de los módulos que lo componen. En nuestro caso, vamos a centrarnos en los archivos que necesita el simulador a su entrada y los que nos ofrece a la salida con los datos obtenidos, considerando todo el núcleo del simulador como una caja negra (Fig. 1.1).

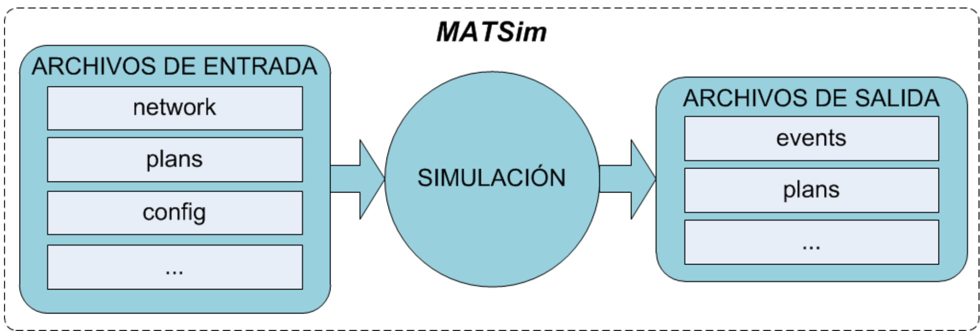


Figura 1.1: Esquema Entrada/Salida MATSim.

En la figura anterior podemos ver los archivos básico que necesita el simulador para trabajar:

- **“network.xml”**: Archivo de tipo entrada que contiene la información sobre el escenario que vamos a simular. Esta información está divida en dos grupos de tipo *“nodes”*, para definir puntos en el área de trabajo, y *“links”*, qué indica uniones entre dos puntos con datos sobre la longitud del enlace, la velocidad de circulación por el mismo, y su capacidad. Con esta dos datos, podemos ver que la información que recibe MATSim de este archivo es un grafo orientado con pesos. Ejemplo:

```
1  <network name="VISUM export national network 2007-11-28">
2   <nodes>
3    <node id="1000" x="730065.3125" y="220415.9531" type="2"
         origid="1000" />
4    <node id="1001" x="731010.5" y="220146.2969" type="2"
        origid="1001" />
5    <node id="10012" x="644902.625" y="234228.0" type="2"
        origid="10012" />
6   </nodes>
7  <!-- ==================================== -->
8   <links capperiod="10:00:00">
9    <link id="100365" from="1000" to="10012" length="921.0"
        freespeed="33.3333333333333" capacity="56000.0"
        permlanes="2" oneway="1" origid="183" type="10" />
10   <link id="100366" from="1001" to="1000" length="921.0"
        freespeed="33.3333333333333" capacity="56000.0"
        permlanes="2" oneway="1" origid="183" type="10" />
11  </links>
12 </network>
```

- **“plans.xml”**: Archivo de tipo entrada en el cual se indica el número de agentes que van a participar en la simulación con la etiqueta *“person”*, el itinerario que va a realizar (*“plan”* – indicando puntos de destino, y el momento en el que parte de ellos). Además, se podrían especificar diferentes medios de transporte (*“leg”*). Ejemplo:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE plans SYSTEM "http://www.matsim.org/files/dtd/
      plans_v4.dtd">
3  <plans>
4  <!-- ==================================== -->
5   <person id="1" employed="no">
6    <plan selected="yes">
7     <act type="origen" x="-3.343295175676936" y="
         40.518178869545736" end_time="09:25:47" />
8     <leg mode="car">
9     </leg>
10    <act type="destino" x="-3.3629627001879836" y="
         40.48728788725905" />
11   </plan>
12  </person>
13 <!-- ==================================== -->
14 </plans>
```

- **“config.xml”**: Contiene la información relativa a la manera de comportarse del simulador. En él se especifican todos aquellos parámetros que configurables tales como, el numero de iteraciones que se van a realizar, nombre de los archivos y directores de entrada y de salida, formatos especiales de salida,. . . .

MATSim, realiza un número de iteraciones definidas en el archivo de configuración, de tal forma qué en cada iteración toma como referencia los resultados de la simulación obtenidos en la anterior y trata de mejorarlos. Por tanto, una vez finalizado el proceso, obtenemos varios grupos de archivos con los resultados de cada *“intento”* organizados en carpetas (it.0, it.1,...).

De entre todos los archivos que contiene cada carpeta correspondiente a una iteración, nos centraremos en uno de los archivos, el cual contiene la información del resultado de la simulación (el resto de archivos, son datos complementarios o interpretaciones de este):

- **“runX.YY.plans.xml”**: Conserva la misma estructura del archivo *“plans.xml”* utilizado a la entrada, pero en el nombre se indica a qué ejecución pertenece (*“X”*), y a qué iteración corresponde (*“XX”*). Además se ha añadido una nueva etiqueta dentro de la estructura de datos llamada (*“route”*) qué indica la sucesión de nodos por los que pasa el agente durante su recorrido (y otros datos importantes como la distancia y el tiempo transcurrido). Si observamos el mismo archivo en diferentes iteraciones, vamos qué los datos de las rutas pasadas no se borran, sino se quedan almacenados en el archivo y el propio simulador marca la propiedad (*“selected”*) de la etiqueta (*“plan”*)

con (*"yes"*) o (*"no"*), dependiendo de si el coste de esa ruta es el menor o no en relación a las otras posibles (es decir, puede ocurrir que en las nuevas iteraciones no se mejores las rutas anteriores). Ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE plans SYSTEM "http://www.matsim.org/files/dtd/plans_v4.dtd">
<plans>
<!-- ==================================== -->
 <person id="0" age="59" employed="yes">
  <plan score="211.06700022162894" selected="yes">
   <act type="h9" link="124605" facility="1521" x="640577.3428467353" y="225985.38934860702" end_time="09:10:15" />
   <leg mode="car" dep_time="09:10:15" trav_time="00:05:52" arr_time="09:16:07">
    <route dist="5885.0" trav_time="00:05:52">
     9628 9629 5182 9630 10082 11968 9626
    </route>
   </leg>
   <act type="l1" link="132405" facility="30" x="644467.1792797182" y="225602.43888185138" end_time="09:34:55" />
   <leg mode="car" dep_time="09:34:55" trav_time="00:07:44" arr_time="09:42:39">
    <route dist="8362.0" trav_time="00:07:44">
     11968 9626 5321 5180 5112 4985 5111 5316
    </route>
   </leg>
   <act type="w5" link="124649" facility="16" x="641950.7971429207" y="219033.26969951618" end_time="16:00:58" />
   <leg mode="car" dep_time="16:00:58" trav_time="00:08:10" arr_time="16:09:08">
    <route dist="9510.0" trav_time="00:08:10">
     9643 5316 5111 4985 4986 4987 5181 5182 5183
    </route>
   </leg>
   <act type="h9" link="124605" facility="1521" x="640577.3428467353" y="225985.38934860702" />
  </plan>
  <plan score="211.06700022162894" selected="no">
   <act type="h9" link="124605" facility="1521" x="640577.3428467353" y="225985.38934860702" end_time="09:10:15" />
   <leg mode="car" dep_time="09:10:15" trav_time="00:05:55" arr_time="09:16:10">
    <route dist="5885.0" trav_time="00:05:55">
     9628 9629 5182 9630 10082 11968 9626
    </route>
   </leg>
   <act type="l1" link="132405" facility="30" x="644467.1792797182" y="225602.43888185138" end_time="09:34:55" />
   <leg mode="car" dep_time="09:34:55" trav_time="00:07:47
```

```
        " arr_time="09:42:42">
36      <route dist="8362.0" trav_time="00:07:47">
37       11968 9626 5321 5180 5112 4985 5111 5316
38      </route>
39     </leg>
40     <act type="w5" link="124649" facility="16" x="
        641950.7971429207" y="219033.26969951618" end_time="
        16:00:58" />
41     <leg mode="car" dep_time="16:00:58" trav_time="00:08:15
        " arr_time="16:09:13">
42      <route dist="9510.0" trav_time="00:08:15">
43       9643 5316 5111 4985 4986 4987 5181 5182 5183
44      </route>
45     </leg>
46     <act type="h9" link="124605" facility="1521" x="
        640577.3428467353" y="225985.38934860702" />
47    </plan>
48   </person>
49  <!-- ==================================== -->
50  </plans>
```

1.2. Lanzar una simulación e interpretar los datos

Sabiendo el funcionamiento de MATSim, siempre que queramos realizar una simulación deberemos de prepara a la entrada los archivos *"network.xml"* y *"plans.xml"*. Según se indica en la documentación de MATSim, el proceso sería el siguiente:

1. Obtener un archivo con los datos geográficos del escenario donde pretendemos realizar la simulación en uno de los formatos compatibles con MATSim (por ejemplo, acudiendo a OpenStreetMap [2], seleccionando el área, exportandola a una archivo tipo osm, y almacenándolo en local).

2. Convertir el archivo de tipo osm, al formato de trabajo de MATSim. (para ello sería necesario llamar al grupo de funciones específicas del API de MATSim, pasando como parámetro el archivo obtenido en el paso anterior).

3. Definir el archivo el contenido del archivo *"plans.xml"*

   a) Obteniendo datos estadísticos reales, que deben de ser tratados para adaptarse al formato especificado.

   b) De manera manual

Tras indicar el nombre de estos archivos, junto con el resto de parámetros necesarios en el archivo de configuración, se llamaría en la línea de comandos a la maquina virtual de Java del sistema con todos los parámetros necesarios y daría comienzo el proceso de simulación.

Una vez finalizado, en ruta de salida que hayamos indicado obtendríamos una serie de directorios y archivos. En especial nos fijaremos en el directorio *"ITERS"*, el cual contiene a su vez tantos directorios como iteraciones se hayan realizado.

Para analizar los datos obtenidos tenemos las siguiente vías:

1. De manera manual, podemos acceder las archivos de salida, y buscar datos de referencia que nos permitan valorar el resultado obtenido. MATSim, genera además de los archivos de texto una serie de imágenes a modo de resúmenes para tratar de facilitar la taréa.

2. La visualización de los datos de la simulación, es la forma más sencilla y rápida para valorar los resultados. Incluidos en MATSim tenemos varias alternativas:

   a) **OFTVis**: Muestra el resultado final de la simulación (únicamente la última iteración), sobre la interpretación del archivo "network.xml".

   b) **Google Earth**: A partir de los datos de salida de MATSim, existe la posibilidad de crear archivos compatibles con Google Earth.

[2] `http://www.openstreetmap.org`

3. Además existen otras soluciones comerciales simulaciones (software propietario) como la que ofrece Senozon [3], empresa privada cuyos integrantes participan activamente en el desarrollo de MATSim, y ofrecen soluciones específicas para tratar y visualizar los datos.

[3] http://senozon.com/

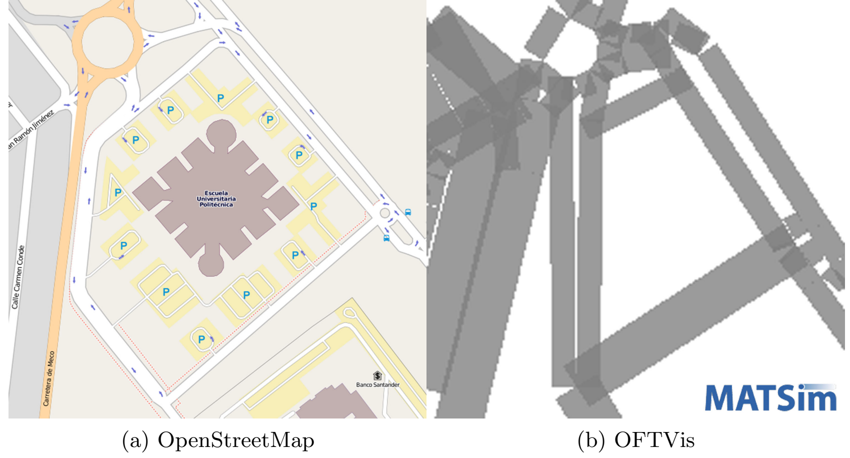
1.3. Dificultades detectadas

Una vez hemos mostrado el funcionamiento general del sistema de entrada y salida del simulador, debemos centrarnos en los carencias o dificultades detectas a la hora de trabajar con él.

Al utilizar un simulador como MATSim, uno de los objetivo principales es mejorar los resultados que obtenemos a la salida, es decir, qué podemos hacer para que partiendo de unas condiciones iniciales prefijadas, la valoración final de la mejor simulación sea mejor. Para ello hay diferentes líneas de trabajo, pero todas cuentan con unas necesidades comunes como poder especificar las condiciones iniciales e interpretar los resultado obtenidos de la manera lo fidedigna y rápida posible.

Según lo visto anteriormente, para definir el escenario inicial hay que realizar gran parte del trabajo de forma manual, con la dedicación de tiempo que conlleva y la posibilidad de cometer errores.

Los sistemas para representar la información con los que cuenta MATSim tienen algunas carencias como podemos observar en la figura 1.2b, donde podemos ver la representación que realizar el módulo OFTVis de los datos de una simulación, la cual comparada con otras representaciones resalta poco realista.



(a) OpenStreetMap (b) OFTVis

Figura 1.2: Ejemplo de representación mapas/redes.

Otra opción para visualizar datos es indicar en el archivo de configuración que se genere una salida específica de tipo *"kml"* para poder ser interpretada por Google Earth[4]. En la figura 1.3 podemos observar un ejemplo, y aunque debido a que los datos se visualizar encima de una capa con la vista satélite de la zona que hemos simulado, el resultado no termina de ser lo suficientemente intuitivo ni interactivo (se tratan de datos estáticos que debido a las conversiones del sistema de coordenadas no terminan de encajar correctamente.

[4]`http://www.google.com/earth/index.html`

Figura 1.3: Visualización de datos en GoogleEarth.

Además del aspecto visual, existen otro tipo de limitaciones tales como la falta de control del resultado que se quiere visionar (no es posible comparar varias iteraciones o simulaciones diferentes del mismo escenario), lo que deriva en una mayor dificultad a la hora de comparar diferentes estrategias o métodos para decir cual ofrece mejores resultados.

1.4. Objetivos del proyecto

Llegados a este punto han quedado claro la necesidad de cubrir determinadas carencias, o mejoras algunas características del simulador MATSim para facilitar la forma de crear un escenario de simulación, y la visualización e interpretación posterior de los resultados. Por tanto los objetivos que pretende alcanzar este proyecto, y la motivación que los origina son:

1. **Automatizar la creación del archivo *"network.xml"***, de tal forma que el usuario final pueda realizar en pocos pasos la búsqueda de una determinada área, su selección y conversión al formato necesario. Persigue el evitar que el usuario final tenga que acudir a diferentes fuentes, y tratar los datos de una manera no estandarizada para conseguir la red de simulación.

2. **Generación del archivo *"plans.xml"*** a partir de directivas dadas según áreas seleccionadas. Debido a la dificultad de encontrar datos estadísticos reales que nos permitirían definir la posición de los agentes en cada momento y sus destinos, debemos de crear una herramienta a la que simplemente indicando zonas orígenes y destinos del escenario de simulación donde especifiquemos el número de agentes que se reúnen en ellas, sea capaz de definir la posición de todos los agentes y emparejarlos. De esta forma, conseguiríamos definir el contenido del archivo de planes en pocos minutos, tarea que de llevarse a cabo manualmente puede llevarnos horas.

3. **Interpretación y visualización de los resultados** de forma realista, y permitiendo la comparativa entre diferentes fuentes. Dotar a nuestra herramientas de los mecanismo para tratar la información de la red de simulación y de los resultados, generando su propia visualización ajustada a una representación realista, que permita al usuario interpretar de manera rápida los resultados obtenidos.

4. **Desarrollar un conjunto de herramientas** que no sólo cumplan con los tres objetivos anteriores, sino que además sean extensibles y fácilmente modificables para permitir posteriores mejoras.

5. **Sencillez de la solución final** a la hora de utilizarla, distribuirla y acceder a ella.

Al final con todos estos objetivos, lo que se pretende es conseguir una solución que por un lado facilite la creación de los archivos de entrada al simulador, evitando en la medida de lo posible que el usuario final tenga que realizar pasos repetitivos. Y por otro lado, permita visualizar e interpretar los resultados sin las restricciones de los sistemas actuales.

1.5. Descripción del sistema

Tras definir cuales son nuestro objetivos, debemos elegir el formato del sistema que nos va a llegar a conseguirlo.

Antes de decidirnos por un camino u otro, debemos de tener en cuanta varios factores:

- El simulador MATSim está desarrollado en Java y necesitaremos utilizar determinados funciones que ya tiene implementas para facilitar nuestro trabajo.
- Entre los objetivos que nos hemos propuesto está el de que la solución sea fácil de distribuir y acceder a ella.
- El sistema debe de ser extensible, y por tanto modular.

Con estas premisas se han tenido en cuenta varias opciones, aunque la que más ventajas ha presentado siempre ha sido el modelo cliente-servidor web, para que en el lado del cliente seamos capaces de presentar la información que recibe del servidor, y enviar las peticiones que realiza el usuario, y el servidor de aplicaciones web además de gestionar las peticiones del servidor, interprete y modifique la información según nuestras necesidades.

Desde el punto de vista del acceso al sistema por parte de los usuarios, una aplicación web presenta muchas ventajas, ya que el usuario final no tiene que instalar en su equipo ningún software especial pudiendo acceder a ella desde su navegador web. Por otro lado, y aunque hay que tener presente que no todos los navegadores web se comportan de la misma forma, es una solución muy genérica y casi cualquier dispositivo con los que trabajamos actualmente cuentan con navegadores web modernos que nos permitan utilizarla.

En el lado del cliente, contamos con un navegador web que recibe del servidor el esqueleto HTML y un conjunto de funciones en JavaScript, para que su propio motor JavaScript puede realizar gran parte de las tareas de manera autónoma.

En el lado del servidor, además de un servidor Web, necesitaremos un servidor de aplicaciones (en nuestro caso Tomcat) que contenga los Servlets que permitan recibir y responder las peticiones que genera el cliente. Contaremos también, con un servicio de base de datos que nos permita gestionar los datos que tratemos con mayor velocidad.

Como veremos en profundidad en cada uno de los capítulos correspondientes, junto con las funciones implementadas de manera especifica para este proyecto se han uso de otros recursos externos como las librerías JavaScript OpenLayers[5] y jQuery[6], que nos facilitaras ciertas tares.

[5] `http://openlayers.org/`
[6] `http://jquery.com/`

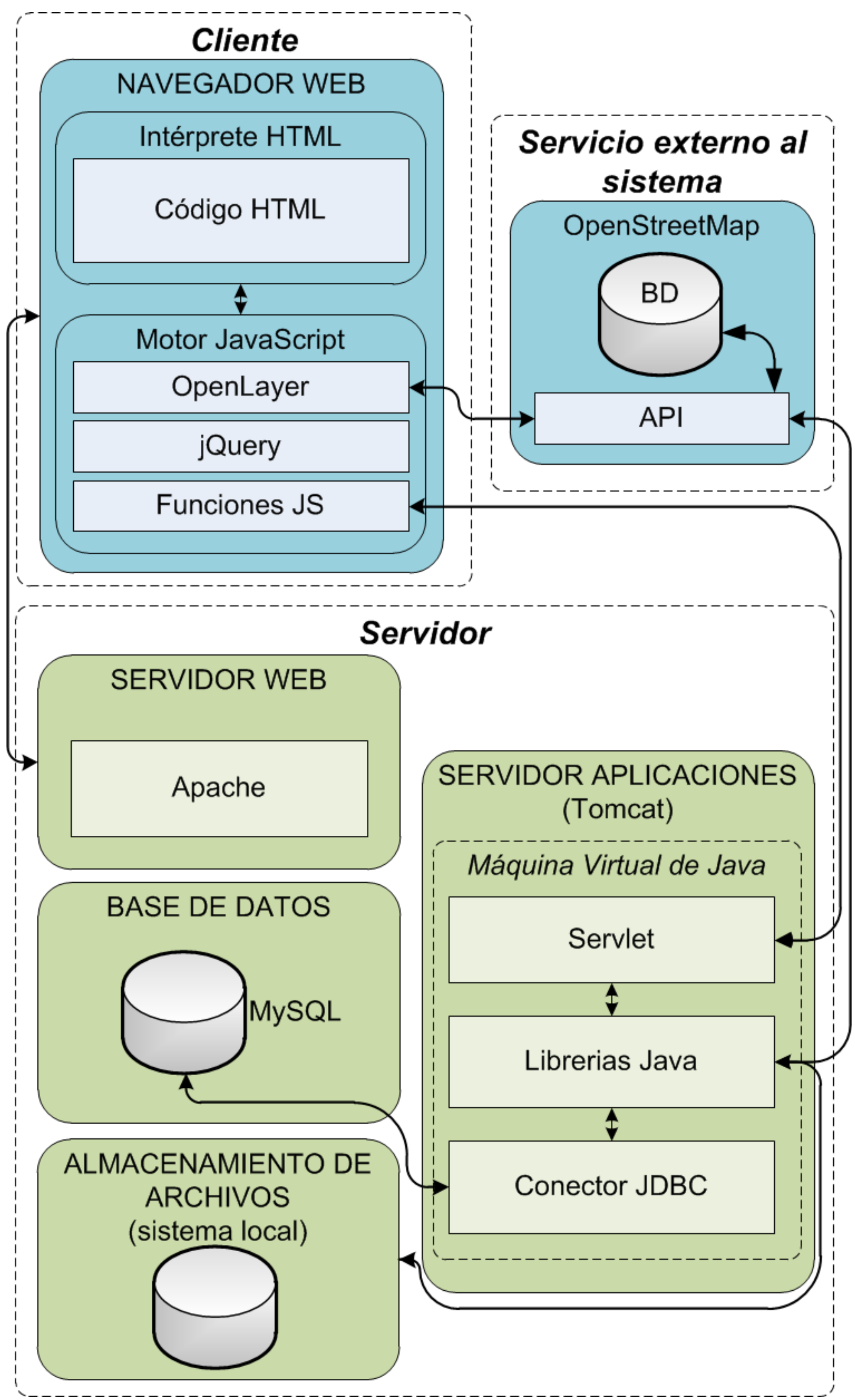


Figura 1.4: Descripción general del sistema y principales vías de comunicación.

Por último, y para cumplir otro de los objetivos fijados, debemos de incluir en nuestro sistema los mecanismo necesarios para comunicarnos con otros sistemas externos necesarios como el ofrecido por OpenStreetMap.

Con todo ello nuestro sistema tendría la forma del esquema de la figura 1.4, donde podemos ver las diferentes partes que lo componen y las principales vías de comunicación que existen entre ellas.

En este esquema podemos apreciar como las distintas partes del sistema están completamente separadas, y son capaces de funcionar de manera autónoma, permitiendo su funcionamiento en otro tipo de sistemas (por ejemplo, si sustituimos al el lado del cliente el navegador web por una aplicación en un dispositivo móvil, no sería necesario modificar la implementación del lado del servidor).

En los siguientes capítulos se explicara con detallas cada una de las partes y se ampliará la información sobre las comunicaciones que se producen entre ellas.

Capítulo 2

Obtención de datos geográficos y conversión

En este capítulo vamos a centrarnos[1] en como cubrir la necesidad de crear escenarios de simulación a partir de datos geográficos reales, de manera rápida y sencilla. Además de las funciones propias de esta parte, hablaremos del sistema de información geográfica OpenStreetMap (su API y licencia de uso), y de la librería de libre distribución OpenLayers, la cual utilizaremos en otras partes del sistema.

2.1. Introducción

Tal y como hemos visto en el capitulo 1, la creación del escenario de simulación puede llegar a ser una tarea relativamente tediosa y complicada.

En determinadas ocasiones es imprescindible rellenar los datos de manera manual, por ejemplo para realizar pruebas muy acotadas, o comprobar alguna funcionalidad específica. Es estos casos, el escenario a simular no suele ser de un tamaño excesivamente grande y por tanto el tiempo que se dedica ha esta tarea es relativamente reducido.

En la mayoría de los casos, vamos a necesitar escenarios que se correspondan con datos geográficos para poder realizar simulaciones que traten de reproducir y anticipar a situaciones reales.

En la documentación de MATSim, se especifican determinadas funciones que permiten la creación del archivo *"network.xml"*, las cuales utilizaremos durante el proceso de conversión. Es necesario contar con una fuente de datos que contenga la

[1]Se ha optado por dividir las diferentes partes del desarrollo de la solución según su funcionalidad final. Debido a esto, hay determinadas secciones del código que se utilizaran en diferentes momentos. En este caso, la primera vez que se aparezcan se explicaran en profundidad, y el resto de las ocasiones simplemente se mencionarán o se remarcan el uso que se hacen de ellas, y se hará referencia al capítulo o sección donde aparecen.

información geográfica y nos permita acceder a ella. En nuestro caso vamos a utilizar OpenStreetMap, del cual hablaremos en la siguiente sección.

Por otro lado a lo largo del proyecto tendremos como uno de los objetivos marcados, el crear un entorno *amigable* de cara al usuario y por ello se hacia imprescindible encontrar una solución para poder mostrar e interactuar con las fuentes de datos geográficos. De las opciones disponibles se ha elegido la librería de libre distribución OpenLayers, la cual entre sus funcionalidades no solo está el mostrar la representación de los datos, sino además nos permite superponer información que vayamos creando de manera dinámica en diferentes capas, para adaptarnos a las necesidades de cada momento.

La información geográfica es almacenada y mostrada utilizando sistemas de coordenadas. Existen multitud de estándares para estos sistemas, y dependiendo de la aplicación con la que trabajemos y de donde sea el usuario final, se utilizan unos u otros. Por ello será necesario contar con funciones específicas como las definidas en la sección 2.4 de este capítulo para poder cambiar el sistema de coordenadas manteniendo la coherencia de los datos. Además en el apéndice A nos centraremos en los sistemas de referencia utilizados en las diferentes partes de la solución y la equivalencia que existe entre ellos para comprender su funcionamiento.

Hablando de cómo están referenciados los datos geográficos, nos encontramos con el problema de que cuando un usuario quiero buscar una determinado lugar a partir del cual crear su escenario, necesitará hacer su búsqueda mediante un método más intuitivo que las coordenadas de esa área. Con ello surge la necesidad de implementar alguna funcionalidad que permita realizar búsquedas cruzadas entre denominaciones de situaciones y su correspondencia a coordenadas. En el apartado de OpenStreetMap hablaremos del servicio *Nominatim* que proporciona y como lo utilizaremos.

Con lo expuesto hasta el momento, y desde el punto de vista del usuario final que utilizará está solución, podemos ver en la figura 2.1, los pasos que seguiría para exportar un escenario al formato de MATSim de manera intuitiva, y por tanto son los pasos en los que nos hemos basado para realizar la implementación de estar parte del sistema.

Hasta el momento, únicamente hemos menciona la obtención de datos geográficos como un concepto abstracto de un sistema externo que nos proporciona esa información. En la siguiente sección, cuando veamos las condiciones de uso del API de OpenStreetMap, comentaremos porque hemos optado por utilizar de manera predefina la petición y descarga de estos datos directamente al servicio web que nos proporciona, aunque también indicaremos la forma de proceder en los casos donde esta opción no sea posible.

Por último, en la sección dedicada las funciones empleadas en esta parte, y dado que dentro de la memoria será la primera vez que expliquemos el código que se ha implementado para la solución, realizaremos una breve explicación de porqué hemos elegido JavaScript para ejecutar funciones en el lado del cliente, el uso de Servlet para gestionar las peticiones y respuestas en el servidor, y la utilización

de MySQL como base de datos del sistema.

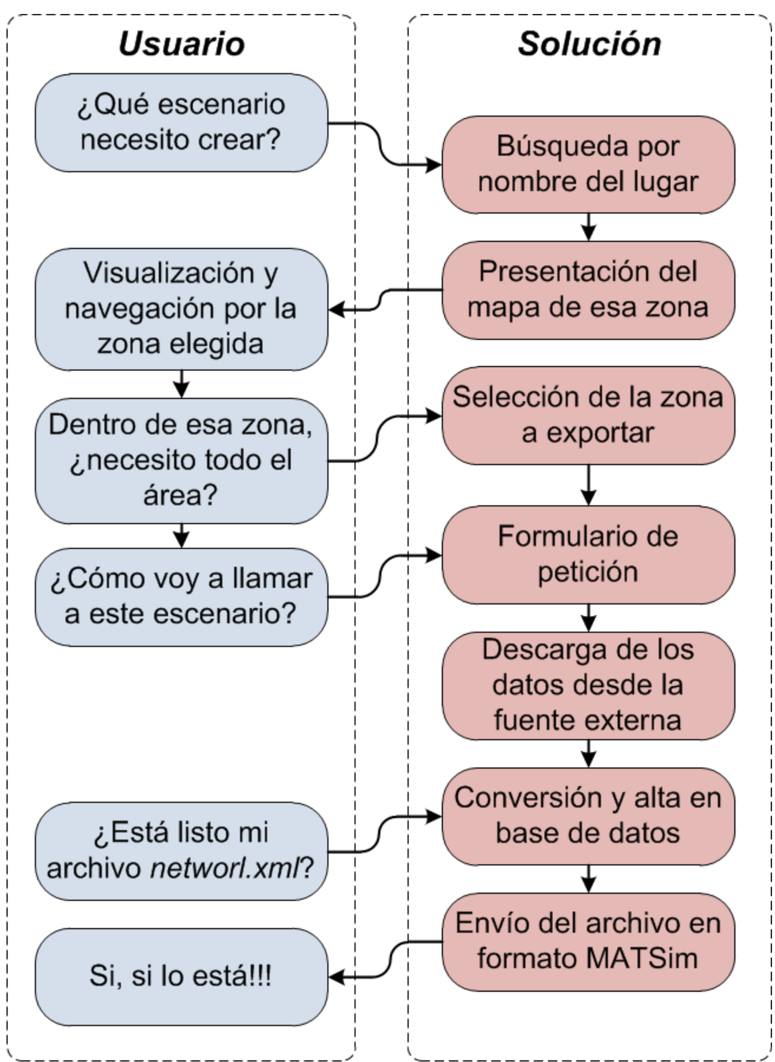


Figura 2.1: Pasos que realiza el usuario para exportar un mapa.

2.2. OpenStreetMap

Ya que OpenStreetMap va a ser nuestra fuentes de datos geográficos, debemos de conocer no sólo su funcionamiento y los servicios que nos puede llegar a proporcionar, sino también las condiciones de uso que nos impone su utilización.

2.2.1. Introducción

Se definen a sí mismo como *the project that creates and distributes free geographic data for the world*, es decir, como un proyecto para crear y distribuir información geográfica del mundo gratuita.

En los últimos años, sobre todo desde al auge del uso de dispositivos GPS, han aparecido multitud de empresas que ofrecen servicios e información relacionada con datos geográficos. En la mayoría de los casos, e incluso en los cosas donde el usuario final no tenga que realizar un pago como tal, estos datos han sido recopilados por las propias empresas y protegidos por licencias de tipo propietario, lo que impide que terceros hagan uso de ellos sin el consentimiento expreso de sus propietarios.

El proyecto de OpenStreetMap nace con el objetivo de crear una base de datos que recopile información geográfica, a través de la colaboración de los propios usuarios, y siendo esta de libre distribución a través de la licencia ODbL [2].

OpenStreetMap recibe información a través de tres vías:

- Comunidad de usuario, los cuales recolectan la información de manera individual mediante dispositivos GPS y posteriormente la vuelcan a través de la API desarrollada para tal fin.
- Datos de carácter público. La mayoría de los paises, en mayor o menor medida, cuentan con datos cartográficos, que publican y pueden ser utilizados.
- Empresas privadas, qué ceden total o parcialmente los recursos de información geográfica con los que cuentan. En la mayoría de los cosas esta participación tiene como objetivo mejorar OpenStreetMap para luego poder utilizarlo.

2.2.2. ¿Sólo mapas?

Cuando hablamos de manera genérica de OpenStreetMap, la característica más destacada es la información geográfica final que nos proporciona, pero esto puede llevarnos al engaño de pensar que esta es su único característica.

Para lograr el objetivo la comunidad que soporta el proyecto OSM (OpenStreetMap), ha tenido que desarrollar una arquitectura de software que permitiera

[2] `http://opendatacommons.org/licenses/odbl/`

tanto la recolección de los datos, como la manera de distribuirlos posteriormente.

Debido a la naturaleza del proyecto, esta arquitectura está formada por soluciones Open Source (SO Linux, servidor web Apache, arquitectura web Ruby on Rails, base de datos PostgreSQL, ...).

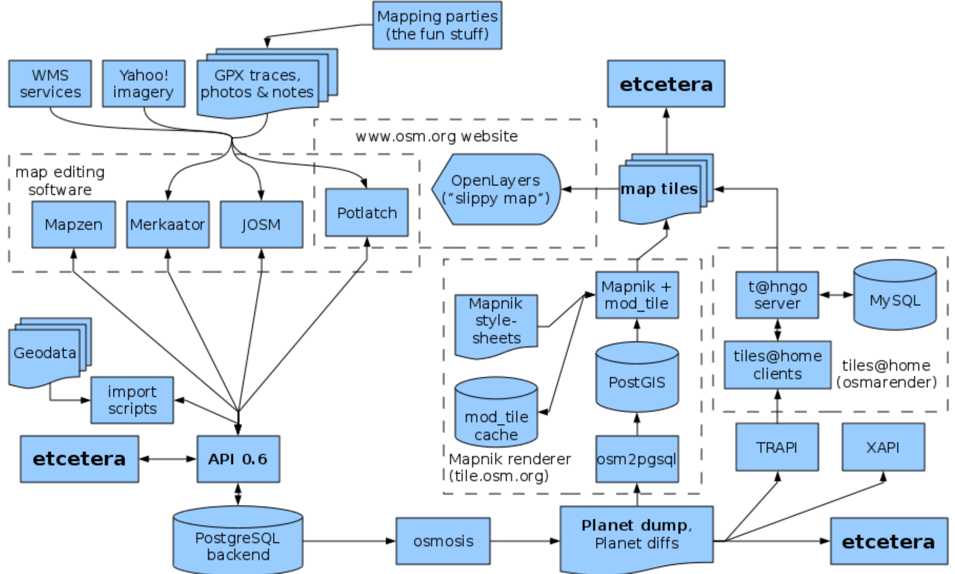


Figura 2.2: Diagrama de la arquitectura básica de OpenStreetMap.

En la figura 2.2 podemos ver los elementos principales que componen el sistema, los cuales podemos dividir en:

- Software de edición que gestiona la entrada de nuevos datos.
- Base de datos donde de almacenan la información.
- Sistema para convertir (renderizar) esa información en una representación real (imágenes que luego pueden ser utilizadas cuando se requieran y su propia base de datos para gestionarlo).
- Sitio Web para acceder a los diferentes servicios que ofrece.

2.2.3. Licencia y restricciones de uso

Hemos comenzado esta sección remarcando el carácter *"gratuito"* en el que se basa OpenStreetMap. Desde Abril de 2012, su licencia[3] de uso a cambiado de Creative Commons By Atribution ShareAlike (CCBySA[4]) a Open Database Licence (ODbL) la cual ha sido específicamente redacta para bases de datos y por tanto

[3] http://www.osmfoundation.org/wiki/License
[4] http://creativecommons.org/licenses/by-sa/2.0/

recoge mejor determinados aspectos. En ambos casos se obligaba a que en caso de realizar mejoras o ampliaciones sobre los datos, estas fueran puestas a disposición bajo el mismo tipo de licencia (Con CCBySA, también ocurría igual con las renderización especificas que hiciéramos a partir de los datos).

En este punto tenemos que tener claro que el uso de los datos es totalmente libre, pero existen determinadas restricciones a la hora de utilizar los servicios que proporciona. En nuestro caso nos vamos a centrar en tres de ellos:

- **Datos geográficos**: Están disponibles bajo petición expresa de zonas en concreto a través de su API (tiene una limitación en cuanto al tamaño de los datos que se pueden solicitar, unos 50.000 nodos), y en archivos de mayor tamaño divididos por áreas geográficas extensas (regiones, países, o continentes) que luego podemos tratar para extraer la porción que nos interese.

- **Renderización *artística* de esos datos**: Esta es una de la tareas que más tiempo de procesa requiere en cualquier sistema. OSM realiza de manera periódica la renderización de diferentes vistas de los datos, según el zoom que corresponda (los zooms más lejanos y por tanto los más utilizados se calculan de manera periódica, mientras a partir de un determinado nivel de detalle se hacen bajo petición expresa, quedando todos ellos quedan almacenados en caché). Esta representación está divida en porciones denominadas *"tile"*, las cuales son solicitadas por las aplicaciones para poder visualizar los datos de manera gráfica.

  Dado el carácter gratuito del sistema, una de sus premisas de funcionamiento es dar servicio a todos sus potenciales usuarios con unos recursos limitados (fondos provenientes de donaciones), y por ello habla siempre en las condiciones de acceso a sus servicios de un ***uso responsable***. En caso de detectar un uso desmesurado proveniente de un único usuario, se especifica claramente que podrán bloquearle el acceso a sus servicios[5].

- **Servido de consulta de localizaciones por nombre *"Nominatim"***: Nos permite realizar consultas indicando un término concreto, y nos devuelve un listado de los lugares encontrados que contengas ese término y sus coordenadas. De manera análoga al punto anterior, un número desmedido de peticiones desde un mismo usuario puede llevar a su bloqueo para garantizar el servicio a la comunidad.

2.2.4. Formato de los datos

Para poder tratar posteriormente la información geográfica, necesitamos saber conocer tanto el tipo de datos como la sintaxis con la que se son guardados.

Desde el punto de vista físico, OpenStreetMap somete a los datos reales a un proceso de modelización para poder definir un número concreto de elementos y la relación que existe entre ellos. Es muy importante resaltar que todos los datos que

[5] `http://wiki.openstreetmap.org/wiki/Tile_usage_policy`

almacena están referenciados mediante el sistema de coordenadas WGS84(Sistema Geodésico Mundial 1984[6] utilizando la proyección de Mercator. En el apéndice A tratamos en profundidad la importancia de este dato. Estos elementos son escritos en los archivos mediante una sintaxis XML. A continuación podemos ver las principales reglas marcadas por su esquema DTD:

```
<!ELEMENT osm (user|preferences|gpx_file|api|changeset|(node|
    way|relation)+)>
<!ATTLIST osm version            CDATA #FIXED "0.6">
<!ATTLIST osm generator          CDATA #IMPLIED>

<!--response to request message api/0.6/user/details -->
<!ELEMENT user (home?)>
<!ATTLIST user display_name      CDATA #REQUIRED>
<!ATTLIST user account_created   CDATA #REQUIRED>

<!ELEMENT home EMPTY>
<!ATTLIST home lat               CDATA #REQUIRED>
<!ATTLIST home lon               CDATA #REQUIRED>
<!ATTLIST home zoom              CDATA #REQUIRED>

<!--response to request message api/0.6/user/preferences -->
<!ELEMENT preferences (tag*)>

<!--response to request message api/0.6/user/gpx -->
<!ELEMENT gpx_file EMPTY>
<!ATTLIST gpx_file id            CDATA #REQUIRED>
<!ATTLIST gpx_file name          CDATA #REQUIRED>
<!ATTLIST gpx_file lat           CDATA #REQUIRED>
<!ATTLIST gpx_file lon           CDATA #REQUIRED>
<!ATTLIST gpx_file user          CDATA #REQUIRED>
<!ATTLIST gpx_file public        (true|false) "false">
<!ATTLIST gpx_file pending       (true|false) "false">
<!ATTLIST gpx_file timestamp     CDATA #REQUIRED>

<!--response to request message api/capabilities -->
<!ELEMENT api (version, area, tracepoints, waynodes)>

<!ELEMENT version EMPTY>
<!ATTLIST version minimum        CDATA #REQUIRED>
<!ATTLIST version maximum        CDATA #REQUIRED>

<!ELEMENT area EMPTY>
<!ATTLIST area maximum           CDATA #REQUIRED>

<!ELEMENT tracepoints EMPTY>
<!ATTLIST tracepoints per_page   CDATA #REQUIRED>

<!ELEMENT waynodes EMPTY>
<!ATTLIST waynodes maximum       CDATA #REQUIRED>

<!--response to request message api/0.6/changeset/*tbd* -->
<!ELEMENT changeset (tag*)>
```

[6] http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html

```
47
48    <!--response to various request messages api/0.6/(create|
         delete|update)/ *tbd* -->
49    <!ELEMENT node (tag*)>
50    <!ATTLIST node id                  CDATA #REQUIRED>
51    <!ATTLIST node lat                 CDATA #REQUIRED>
52    <!ATTLIST node lon                 CDATA #REQUIRED>
53    <!ATTLIST node changeset           CDATA #IMPLIED>
54    <!ATTLIST node visible             (true|false) #REQUIRED>
55    <!ATTLIST node user                CDATA #IMPLIED>
56    <!ATTLIST node timestamp           CDATA #IMPLIED>
57
58    <!ELEMENT way (tag*,nd,tag*,nd,(tag|nd)*)>
59    <!ATTLIST way id                   CDATA #REQUIRED>
60    <!ATTLIST way changeset            CDATA #IMPLIED>
61    <!ATTLIST way visible              (true|false) #REQUIRED>
62    <!ATTLIST way user                 CDATA #IMPLIED>
63    <!ATTLIST way timestamp            CDATA #IMPLIED>
64
65    <!ELEMENT nd EMPTY>
66    <!ATTLIST nd ref                   CDATA #REQUIRED>
67
68    <!ELEMENT relation ((tag|member)*)>
69    <!ATTLIST relation id              CDATA #REQUIRED>
70    <!ATTLIST relation changeset       CDATA #IMPLIED>
71    <!ATTLIST relation visible         CDATA #IMPLIED>
72    <!ATTLIST relation user            CDATA #IMPLIED>
73    <!ATTLIST relation timestamp       CDATA #IMPLIED>
74
75    <!ELEMENT member EMPTY>
76    <!ATTLIST member type              (way|node|relation) #REQUIRED
         >
77    <!ATTLIST member ref               CDATA  #REQUIRED>
78    <!ATTLIST member role              CDATA  #IMPLIED>
79
80    <!ELEMENT tag EMPTY>
81    <!ATTLIST tag  k                   CDATA #REQUIRED>
82    <!ATTLIST tag  v                   CDATA #REQUIRED>
```

Además de la información básica que contiene (tipo de documento, versión de la API que lo creó, usuarios, . . .), vamos a centrarnos en la explicación de las etiquetas que nos proporcionan información relevante sobre datos geográficos.

Las dos etiquetas principales son:

- ***nodes***: Especifican un punto único punto geográfico en concreto. Se utilizan principalmente como referencia para los caminos que se definen en la siguiente etiqueta.

- ***ways***: Recogen una lista ordena de nodos (contenido cada uno de ellos en la etiqueta *nd*). Se pueden utilizar para representar sucesiones de líneas, o polígonos.

Y a ellas se le pueden unir:

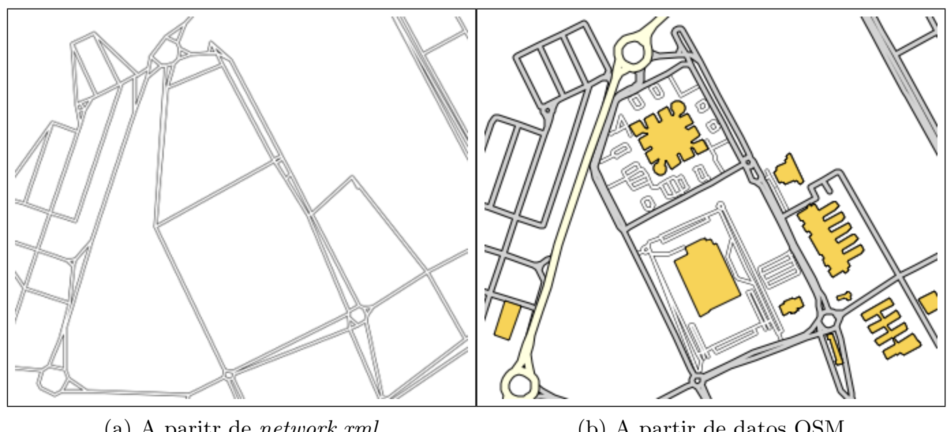
- ***relations***: Permiten indicar propiedades comunes a un conjunto de elementos determinado.
- ***tags***: Definidas por el atributo *key*, definen el tipo de nodos o camino que lo contiene.

A continuación podemos ver un fragmento del uso de esta sintaxis:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="40.5095200" minlon="-3.3464100" maxlat="
    40.5134400" maxlon="-3.3423600"/>
 <node id="637775843" lat="40.5112105" lon="-3.3440059" user="
    mor" uid="220932" visible="true" version="1" changeset="
    3860323" timestamp="2010-02-12T21:09:33Z"/>
 <node id="637776005" lat="40.5133559" lon="-3.3439970" user="
    mor" uid="220932" visible="true" version="1" changeset="
    3860323" timestamp="2010-02-12T21:09:41Z"/>
    .
    .
    .
 <way id="50170623" user="mor" uid="220932" visible="true"
    version="2" changeset="3864196" timestamp="2010-02-13T12
    :32:47Z">
  <nd ref="637776017"/>
    .
    .
    .
  <nd ref="637776017"/>
  <tag k="leisure" v="track"/>
  <tag k="name" v="Pista de atletismo"/>
  <tag k="sport" v="athletics"/>
 </way>
 <relation id="2036415" user="gpesquero" uid="3584" visible="
    true" version="1" changeset="10731209" timestamp="
    2012-02-19T15:30:59Z">
  <member type="node" ref="599016156" role="stop"/>
  <member type="node" ref="599016157" role="stop"/>
  <member type="way" ref="32918620" role="forward"/>
  <member type="way" ref="45896980" role="forward"/>
  <member type="way" ref="46929202" role="forward"/>
  <tag k="ref" v="227"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
 </relation>
</osm>
```

A diferencia del archivo *network.xml* que utiliza MATSim el cual sólo contiene la información básica para realizar la simulación, en los archivos de datos de OSM se incluye mucha información adicional de los nodos y los caminos (tipo de camino,

nombre, ...), además de añadir otros elementos nuevos que permiten definir edificios, puentes, pistas deportivas, universidades... qué aunque en nuestro caso no son necesario para la simulación, consiguen que la representación de los datos sea mucho más realista como podemos ver en la figura 2.3



(a) A paritr de *network.xml* (b) A partir de datos OSM

Figura 2.3: Representación a partir de los datos geográficos.

2.2.5. Servicios y acceso a ellos

OpenStreetMap proporciona numerosos servicios que van desde herramientas para la creación, modificación y volcado de datos geográficos, hasta la propia arquitectura que utiliza para soportar su sistema, por si quisiéramos montar nuestro propio servicio de mapas particular.

En este caso de estudio, obviaremos la forma en que estos servicios se prestan en a través de su entorno web, y nos centraremos en cómo nuestra solución podrá tener acceso a ellos. Para cubrir nuestras necesidades utilizaremos dos de sus servicios más demandados: la obtención de datos, y la relación entre nombre de lugares y coordenadas.

En cuanto al la obtención de datos, siguiente la propia documentación[7] de OpenSreetMap, contamos con varias vías:

- **Descargado en local el conjunto de datos** y separando posteriormente por nuestra cuenta la porción de información que necesitemos. Con cierta periodicidad se ponen a las disposición de los usuario, archivos que contiene la información de áreas geográficas extensas [8], que pueden ir desde países, hasta un único archivo con toda la información que contiene su base de datos (actualmente el archivo *planet.osm* tiene un tamaño de 24 GB comprimido).

  Este método presente claros inconvenientes como el no trabajar con datos totalmente actualizados, el tener que manejar archivos de gran tamaño (existe la

[7] `http://wiki.openstreetmap.org/wiki/Getting_Data`
[8] Sitio web donde están disponible: `http://planet.openstreetmap.org/`

posibilidad de descargar únicamente las variaciones que han surgido desde la última actualización, pero esto requeriría volcar la información de los archivos en nuestra propia base de datos, para poder actualizar sólo una parte), y tener que analizar y extraer por nuestro propios medios la porción de datos buscada utilizando herramientas como *Osmosis*[9], proceso que según las pruebas realizadas puede llegar a ser extremadamente lento[10].

- **Realizar la petición a la API de OSM** del recuadro de información que necesitemos. En un principio, este es el método más sencillo y efectivo, ya que la tarea de más costosa para el sistema de extraer la información y formas con ella el archivo final, recae sobre la infraestructura de OpenStreetMap. Además nos aseguramos de que los datos cuenten con la última actualización en ese momento.

  También presenta algún inconveniente, y es que al igual que con otros servicios, OSM limita en cierta medida el uso que se puede hacer de este tipo de solicitudes para no colapsar su sistema. Estas limitación son que el área a exportar no puede superar los 50.000 nodos, lo cual no suele ser demasiado problema ya que normalmente los escenarios que son objetos de simulación no suelen alcanzar esa cifra (por ejemplo ciudades del tamaño de Alcalá de Henares o Móstoles, se quedan muy lejos de esa cifra límite). Además, en las condiciones de uso de la API nos vuelven a recordar el uso racional de las peticiones, y la posibilidad de denegarnos el acceso en caso de un uso inapropiado.

Por ello, el camino que vamos a utilizar en la mayoría de los casos es realizar peticiones directamente a la API de OSM, y estas deben de seguir la siguiente sintaxis:

```
http://api.openstreetmap.org/api/0.6/map?bbox=left,bottom,right,top
```

Donde left, bottom, right, top (min long, min lat, max long, max lat) se refieren a las coordenadas del rectángulo que queramos obtener. El API de OSM, tras recibir nuestra petición, extraerá de su base de datos las información recogida es esa área, y formara un archivo denominado *map.osm* que nos devolverá.

Por último debemos de comentar el funcionamiento del servicio de consulta de coordenadas mediante nombres, *Nominatim*[11]. Las peticiones admiten diferentes parámetros donde indicaremos la cadena a buscar, y el formato de la respuesta entre html, xml, o json. Si queremos realizar una consulta escribiendo nosotros mismos la url, esta deberá tener este formato:

```
http://nominatim.openstreetmap.org/search?q=NOMBRE_A_BUSCAR&format=FORMATO
```

[9] http://wiki.openstreetmap.org/wiki/Osmosis

[10] Prueba realizada en equipo Intel i5 a 2'3 GHz , 4 GB de RAM y MacOS 10.7.5, para extraer la porción de información contenido en el rectángulo de coordenadas top=40.516 left=-3.354 bottom=40.506 right=-3.340 (área que ocupa parte del campus externo de la Universidad de Alcalá), a partir del archivo *spain.osm* ya descomprimido, llevó un tiempo total de 14 minutos y 13 segundos.

[11] http://wiki.openstreetmap.org/wiki/Nominatim

A continuación mostraremos una petición concreta y la información que nos ha devuelto el servicio:

`http://nominatim.openstreetmap.org/search?q=mazarulleque&format=xml`


```xml
<searchresults timestamp="Wed, 12 Dec 12 00:10:59 +0000" attribution="Data OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright" querystring="mazarulleque" polygon="false" exclude_place_ids="16893579" more_url="http://nominatim.openstreetmap.org/search?format=xml&exclude_place_ids=16893579&accept-language=es-ES,es;q=0.8&q=mazarulleque">
<place place_id="16893579" osm_type="node" osm_id="1480358064" place_rank="19" boundingbox="40.1846237182617,40.184627532959,-2.77009320259094,-2.77009296417236" lat="40.1846251" lon="-2.7700931" display_name="Mazarulleque, Cuenca, Castilla-La Mancha, Espana" class="place" type="hamlet" icon="http://nominatim.openstreetmap.org/images/mapicons/poi_place_village.p.20.png"/>
</searchresults>
```


Aunque nosotros no llegaremos a usarlo, Nominatim permite realizar también búsquedas inversas, es decir, indicándole unas coordenadas concretas nos devolverá la los datos de la denominación que le corresponde.

2.3. OpenLayer

Antes de comenzar con nuestra propia solución, se nos plantea el problema de cómo crear un entorno amigable para el usuario que sea capaz de mostrar la información geográfica, y nos permita interactuar con ella.

Todos tenemos en la mente determinados servicios Web dedicados a mapas, en donde a partir de muchas imágenes juntas se muestras como si fueran una sola ofreciendo la sensación de que estamos *"navegando"* sobre un mapa.

Tras probar varias opciones se ha optado por el uso de la librería JavaScript de libre distribución OpenLayer, debido a la gran cantidad de funciones de las que podemos hacer uso y su sencilla integración en entornos Web.

2.3.1. Introducción

Podemos comenzar por el final para hacernos una idea rápida de lo podemos conseguir utilizando esta librería. En la figura 2.4, vemos el resultado de incluirla y llamarla mediante funciones JavaScript desde el esqueleto HTML de una página Web[12].

Figura 2.4: Ejemplo básico de uso de OpenLayer.

Según la propia definición que se nos da en la documentación, OpenLayer en una librería JavaScript diseñada para mostrar datos de mapas en los navegadores web modernos, de manera independiente al servidor donde se aloje nuestra aplicación Web. Permite crear visualizaciones complejas de datos geográficos, de forma similar a otras soluciones como Googles Maps o el servicio de mapas de Microsoft, pero mediante código abierto.

A la hora de trabajar con OpenLayer, debemos de tener claros determinados

[12]Extraído de `http://openlayers.org/`.

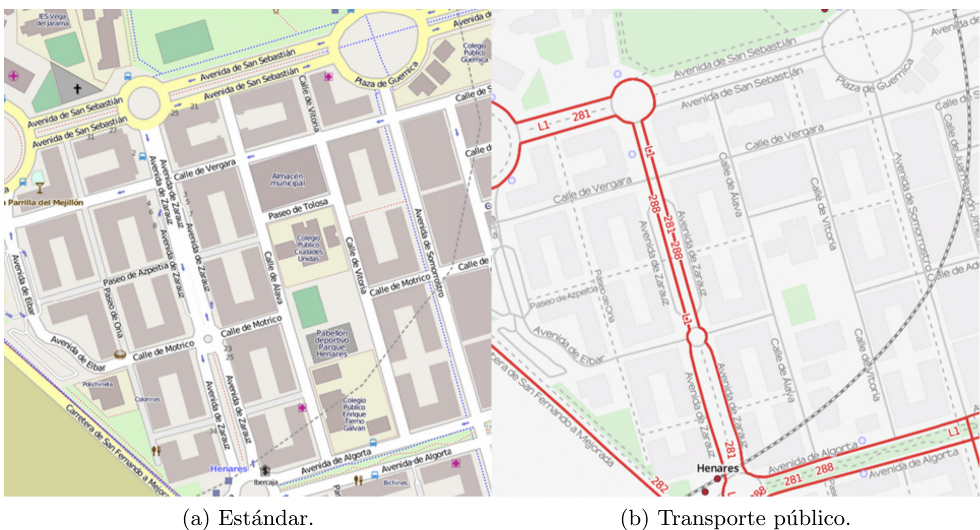
conceptos sobre programación de JavaScript y el funcionamiento de los sistemas de información geográfica, ya que para realizar nuestra personalización, no bastará con utilizar la plantilla por defecto, sino que será necesario realizar modificaciones y llamadas específicas a determinas funciones de la librería.

2.3.2. Cómo funciona

Nos centraremos en dos temas para comprender el funcionamiento de esta librería. Por un lado la manera en que los servicios de información geografía ofrecen el acceso a sus datos, y por otro el funcionamiento en sí de la librería y la organización de sus funciones.

OpenLayers es compatible con varios sistema de información geográfico tales como Google Maps[13], WMS[14], OpenStreetMap,... a los cuales es capaz de realizar peticiones y gestionar sus respuestas.

Como ya hemos visto en el apartado **??**, la información geografica se almacena una vez modelizada, a través de unos parámetros que definen la estructura de los elementos que la componen. Cuando visualizamos un mapa, lo que estamos viendo es una representación *artística* de esos datos, es decir, un dibujo realizado siguiendo los diferentes tipos de etiquetas y los datos que contienen.



(a) Estándar. (b) Transporte público.

Figura 2.5: Diferentes representaciones de los mismos datos geográficos.

Existen diferentes soluciones para realizar esta conversión. OpenStreetMap, en su última actualización realiza esta tarea con el conjunto de herramientas denominadas Mapnik[15]. Es posible realizar diferentes visualización a partir del mismo

[13] `http://maps.google.es/`

[14] servicio Web Map Service definido por el OGC (Open Geospatial Consortium) `http://www.opengeospatial.org/standards/wms`

[15] `http://mapnik.org/`

conjunto de datos, dependiendo de que etiquetas elijamos interpretar como podemos ver en la figura 2.5. Por otro lado, debemos destacar que para cada nivel de zoom que queramos visualizar de debe realizar una renderización específica, ya que aunque partamos de los mismo datos, la imagen mostrada no es la misma para una vista general de una ciudad, que de una determinada zona. Ningún sistema realiza la renderización de todos sus datos, vistas, y zooms posibles debido a que conlleva un gran gasto por parte del sistema donde se ejecuta tanto a nivel de procesamiento como de tiempo.

El sistema que se utiliza para gestionar el uso eficiente de los recursos es dividir la imagen en fragmentos denominados *"tile"*, típicamente de 256x256 pixeles. De esta forma cuando se realiza la renderización de una zona para un nivel de zoom determinado, todos los *"tiles"* que componen la imagen quedan almacenadas y en sucesivas peticiones, el sistema no vuelve a formar dibujo, sino simplemente envía al usuario que lo solicitó los *"tiles"* correspondientes a esa área, y es el lado del cliente el encargado de colocar y mostrar el conjunto de *"tiles"* que ha recibido.

A continuación podemos ver las peticiones que llevan a cabo en un ejemplo sencillo para poder visualizar la figura 2.4 :

```
GET http://c.tile.openstreetmap.org/16/32158/24688.png [HTTP/1.1 200 OK 110ms]
GET http://c.tile.openstreetmap.org/16/32158/24687.png [HTTP/1.1 200 OK 47ms]
GET http://a.tile.openstreetmap.org/16/32157/24688.png [HTTP/1.1 200 OK 102ms]
GET http://b.tile.openstreetmap.org/16/32157/24687.png [HTTP/1.1 200 OK 61ms]
GET http://c.tile.openstreetmap.org/16/32159/24688.png [HTTP/1.1 200 OK 104ms]
GET http://b.tile.openstreetmap.org/16/32159/24687.png [HTTP/1.1 200 OK 56ms]
```

Cada petición se corresponde con la solicitud de una imagen de tamaño 256x256 pixeles. Estas imágenes son organizadas y colocadas para dar la sensación de continuidad. En la figura 2.6a tenemos la imagen obtenida de una de las peticiones anteriores, y en la figura 2.6b la posición que ocupa. Como podemos ver, el *tile* solo se muestra parcialmente y está desplazada, debido a la necesidad de adaptar las imágenes genérica con las que cuenta el servicio de información geográfica, a nuestra petición particular.

Este sistema nos proporciona grandes ventajas ya que en nuestra aplicación web, estas peticiones la realizará directamente el cliente a OpenStreetMap sin pasar por nuestro servidor, con el consiguiente ahorro de recursos por nuestra parte. Presenta un leve problema, ya que OpenStreetMap realiza la renderización de las vistas más utilizadas de manera periódica, por lo que cuando se incluye nueva información geográfica puede llegar a tardar varios días en aparecer reflejada en las imágenes que nos descarguemos (para niveles de zooms con mucho detalle, las renderizaciones son bajo demanda, por lo que esto no ocurre).

Todo este proceso es transparente no solo al usuario final, sino también a nuestra aplicación, ya que una vez definido las configuraciones necesarias que veremos en la siguiente sección, las diferentes funciones de la librería OpenLayers son las que se encargan de comunicarse con la API del sistema de mapas que hayamos elegido, y mostrar la información.

(a) Tile.

(b) Posición dentro del div.

Figura 2.6: Ejemplo de uso de los *tiles*.

Contamos con numerosa documentación para conocer todas las funcionalidades de esta librería y de cómo poder llegar a hacer uso de ellas(libros[6], documentación web[16], y multitud de ejemplo de uso[17])

2.3.3. Cómo se utilizar

Básicamente, lo único que necesitamos es incluir las librerías js en nuestra aplicación web, definir un DIV dentro del esqueleto HTML el cual vaya a contener el mapa, y llamar a la función correspondiente para iniciar el proceso.

OpenLayers es un conjunto de librerías, entra las que cabe destacar Map.js, Layer.js, Control.js, y Handler.js, las cuales realizan las tareas más importantes.

A continuación mostramos un esqueleto básico de HTML, con los elementos necesarios para albergar el mapa:

```
<!DOCTYPE html>
<html lang=es>
<meta charset=utf-8>
<head>
  <title>Mapa Basico</title>
  <script type="text/javascript" src="openlayers/lib/OpenLayers
     .js"></script>
  <script src="mapa_basico.js"></script>
  <style>
    div.olMap {
      cursor: default;
      margin: 0 auto;
```

[16] http://dev.openlayers.org/releases/OpenLayers-2.12/doc/apidocs/files/OpenLayers-js.html

[17] http://openlayers.org/dev/examples/

```
12          padding: 0 !important;
13        }
14        .smallmap {
15          border: 1px solid #CCCCCC;
16          width:600px;
17          height: 350px;
18          padding : 0px;
19        }
20      </style>
21    </head>
22      <body onload="initializeMap();">
23      <div id="map" class="smallmap"></div>
24      </body>
25    </html>
```

- En la línea 6, incluimos la librería JavaScript OpenLayers.
- En la línea 7 la librería que contiene la función *initializeMap();* que permitirá cargar nuestro mapa.
- Entre las líneas 8 y 20 modificamos algunos atributos de los elementos de las hojas de estilo para personalizarlas según nuestras necesidades.
- En la 23, definimos un nuevo DIV y le asignamos un *"id"* al que recurriremos para cargar el mapa. Es importante añadir *class="smallmap"*, ya que OpenLayers trae definidas unas hojas de estilo muy extensas que siempre tomaremos como base.
- Por último, en la línea 22 indicamos que al cargar la págine web llamemos a la función que inicia el proceso para visualizar el mapa.

A partir de este punto, vamos a comenzar a utilizar realmente las funciones de la librería OpenLayers. Veamos el archivo que contiene la función *initializeMap();*, para después analizarlo.

```
1   var map;
2
3   function initializeMap() {
4   // configuramos mapa
5     map = new OpenLayers.Map( 'map' , {
6       controls: [
7         new OpenLayers.Control.Navigation(),
8         new OpenLayers.Control.Attribution(),
9         new OpenLayers.Control.PanZoomBar()]});
10
11      //creamos capa base y la anyadimos
12      var capa = new OpenLayers.Layer.OSM('Capa OSM',{'
            attribution': 'Provided by OSGeo'});
13      map.addLayers([capa]);
14
15      // calculamos coordenada
16      var lonLat = WGS84ToSphericalMercator(-3.7170, 40.4951);
```

```
17	    //centramos mapa
18	    map.setCenter(lonLat , 1);
19	  }
20	
21	function WGS84ToSphericalMercator(lon, lat) {
22	  return new OpenLayers.LonLat(lon, lat)
23	    .transform(
24	      new OpenLayers.Projection("EPSG:4326"), // de WGS 1984
25	      map.getProjectionObject() // a Mercator Esferico
26	    );
27	}
```

Para analizar este código, nos detendremos más que con el ejemplo anterior. El proceso que debemos de seguir es siempre, crear el objeto de tipo *OpenLayers.Map* que nos permitirá comenzar a hacer uso de todas las funcionalidades, configurarlo según nuestras necesidades, y añadir la capas que queramos que se muestren en él.

Para la definición del nuevo objeto se utiliza la sentencia:

```
new OpenLayers.Map( div, options)
```

- **div**: Nombre del id de la etiqueta DIV de nuestro esquelto HTML que va a contener al mapa.
- **options**: Opciones de configuración de nuestro mapa. Se pueden añadir propiedades independientes, o grupos de elementos que están agrupados según funcionalidad (por ejemplo *Controls*)

En este ejemplo, creamos la variable de nombre *map*, y le asignamos en la línea 5 un nuevo objeto de tipo textitOpenLayers.Map al que le pasamos como argumentos el nombre del DIV, y como opciones creamos un grupo de controles para poder interactuar con el mapa. En la figura 2.7 podemos ver la conseguimos que conseguimos con cada control.

El siguiente paso es crear una nueva capa y añadir al objeto tipo Map, que hemos creado al comienzo. En la línea 12, creamos una nueva capa de tipo OSM, a la que llamamos *'Capa OSM'*, y en la línea 13 la añadimos a map.

En este punto, ya podríamos visualizar el mapa y nos aparecería con el resto de opciones por defecto, pero vamos a personalizarlo un poco más. Fijaremos el centro del mapa y el nivel del zoom. Los objetos de tipo *OpenLayers.Map* cuentan con la función *setCenter*, que nos permite realizar las dos tareas y que sigue el siguiente prototipo:

```
OpenLayers.Map.setCenter( lonlat, zoom)
```

- **lonlat**: Indica el punto de coordenadas donde queremos centrar el mapa.

- **zoom**: El nivel del zoom inicial.

En nuestro ejemplo llamamos a esta función en la línea 18, pero si nos fijamos, le pasamos como coordenadas la variable lonLat que hemos creado en la línea 16 llamando a otra función. Esto es debido a dos razones:

1. El atributo lonlat de la función setCenter es un Array definido de tipo OpenLayers.LonLat, por lo que no podemos pasarle directamente los valores numéricos de las coordenadas.
2. Como comentamos en el capitulo anterior, OpenStreetMap trabaja con el sistema de coordenads WGS84 utilizando la proyección de Mercator. Para hacer peticiones a OpenLayers **SIEMPRE** debemos de hacerle llegar los datos en el sistema de referencia de la proyección, ya que es este sistema el que permite representar la información en 2 dimensiones y dibujarla, y por tanto con el que trabaja. Aunque trabajamos con el sistema de coordenadas debido a qué estamos más acostumbrados a él, haremos uso de las propias funciones que nos proporciona para realizar la conversión (podemos ver este proceso entre las líneas de código 21 y 26).

El resultado que obtenemos es el siguiente:

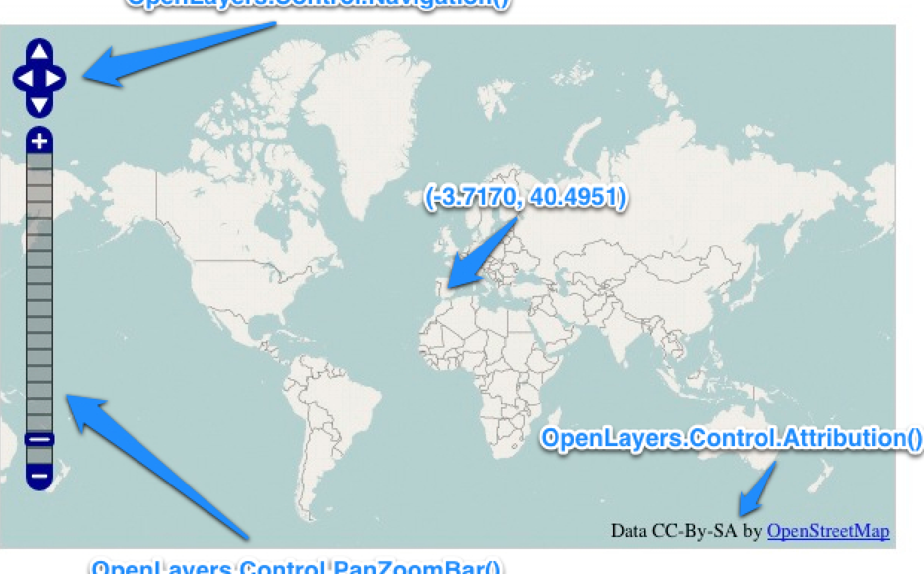


Figura 2.7: Efecto del uso de las propiedades del mapa.

2.3.4. Cómo extraer datos

En el apartado anterior hemos visto como podemos mostrar en nuestro sitio Web un mapa, pero aún necesitamos que OpenLayers nos ayude en algo más. Una

vez tengamos visualicemos los mapas, tendremos que interactuar con ellos, pudiendo seleccionar áreas y recuperando sus coordenadas para luego poder utilizar esos datos en nuestra aplicación. Para ellos vamos a explicar como manejar determinados eventos claves, y el uso de varias capas para superponer información.

Ya sabemos como incluir una capa en nuestro mapa con la información geográfica. De igual forma podemos crear capas vectoriales que se superpongan a la anterior, pudiendo realizar trazos sobre ellas. Para nuestro caso, para interactuar con el mapa vamos a utilizar el ratón o puntero de nuestro sistema, por lo que debemos aprender a interceptar los eventos que provoque el ratón al pulsar y moverse por el mapa. Para ello modificaremos el ejemplo anterior de tal forma:

```
function initializeMap() {
  // configuramos mapa
  map = new OpenLayers.Map( 'map' , {
    controls: [
      new OpenLayers.Control.Navigation(),
            new OpenLayers.Control.Attribution(),
            new OpenLayers.Control.PanZoomBar()]});
  //anyadimos capas base
  var capa = new OpenLayers.Layer.OSM('OSM');
  map.addLayers([capa]);

  vectorLayer = new OpenLayers.Layer.Vector("Zona a exportar");
  map.addLayers([vectorLayer]);

  control = new OpenLayers.Control();
  OpenLayers.Util.extend(control, {
    draw: function () {
      this.box = new OpenLayers.Handler.Box( control,
      {"done": this.notice});
    },
    notice: function (bounds) {
      vectorLayer.removeAllFeatures();
      //recibimos coordenades de pixel
      //y las pasamos a coordenadas del mapa
      var left_top = map.getLonLatFromPixel(new OpenLayers.
         Pixel(bounds.left, bounds.top));
      var right_bottom = map.getLonLatFromPixel(new OpenLayers.
         Pixel(bounds.right, bounds.bottom));

      var newPoint = new OpenLayers.Geometry.Point(left_top.lon
         .toFixed(4),left_top.lat.toFixed(4));
      pointList.push(newPoint);
      var newPoint = new OpenLayers.Geometry.Point(right_bottom
         .lon.toFixed(4),left_top.lat.toFixed(4));
      pointList.push(newPoint);
      var newPoint = new OpenLayers.Geometry.Point(right_bottom
         .lon.toFixed(4),right_bottom.lat.toFixed(4));
      pointList.push(newPoint);
      var newPoint = new OpenLayers.Geometry.Point(left_top.lon
         .toFixed(4),right_bottom.lat.toFixed(4));
      pointList.push(newPoint);

```

```
37        //dibujamos un rectangulo en la capa vectorial
38        pintarCuadrado(pointList,vectorLayer);
39        //vaciamos la variable
40        pointList.splice(0,4);
41        selecArea("salir");
42          }
43    });
44      map.addControl(control);
45      // calculamos coordenada
46      var lonLat = WGS84ToSphericalMercator(-3.7170, 40.4951);
47      //centramos mapa
48      map.setCenter(lonLat, 2);
49  }
50
51  function pintarCuadrado(listapuntos,capaOL){
52    var linearRing = new OpenLayers.Geometry.LinearRing(
         listapuntos);
53    var polygonFeature = new OpenLayers.Feature.Vector(
54      new OpenLayers.Geometry.Polygon([linearRing]));
55    capaOL.addFeatures([polygonFeature]);
56  }
57
58  function selecArea(accion){
59    if (estado_selec.checked)
60      control.box.activate();
61    else
62      control.box.deactivate();
63  }
```

- Líneas 12-13: Creamos y añadimos una nueva capa de tipo Vector a nuestro mapa.
- Línea 16: Iniciamos un nuevo control, para poder manejar los eventos que nos interesen de los que se producen sobre el mapa (es importarte que la variable que almacene este objeto esté declara de manera global, para que se accesible desde las diferentes funciones).
- Línea 17: Recogemos el evento *draw*, cuando mantenemos pulsado el ratón sobre el área de trabajo y creamos un elemento de tipo box, que al finalizar llamará a *notice*.
- Línea 21: Recibe un objeto que contiene dos coordenadas correspondientes a la esquina superior izquierda, y a la esquina inferior derecha, del rectángulo que hayamos dibujos con el manejador box definido en la línea 17.

En este punto hemos conseguido obtener las coordenadas de dos puntos, pero en el mapa no quedará reflejada nuestra selección una vez soltemos el click del ratón. Por ello debemos de añadir una figura con esos datos a la capa vectorial.

- Línea 22: Lo primero que hacemos en borrar posibles dibujos anteriores.

- Líneas 25 a 35: Tratamos esos datos para adaptarlos a nuestras necesidades y al formato que requiere el siguiente paso.
- Línea 38: Llamamos a la función que hemos implementado para generar la figura geométrica definidas por sus coordenadas, y la añadimos a la capa vectorial.

Aunque ya es totalmente funcional, debemos de tener en cuenta otro aspecto. Por defecto, cuando nos posicionamos encima del mapa, hacemos click y desplazamos el puntero nos permite movernos por el mapa. Si este control que acabamos de añadir estuviera siempre activo, perderíamos la opción de movernos por el mapa. Por este motivo se ha añadido entre las líneas 58 y 63 una función que puede interactuar con elementos de HTML para activar y desactivar el control.

Figura 2.8: Rectángulo dibujado una vez seleccionada un área.

Debemos de hacer una ultima consideración. En general utilizamos los valores por defecto de OpenLayers, y en ocasiones puede ocurrir que determinadas actualizaciones no se vean reflejadas durante un tiempo. En este caso, si nos fijamos en el valor del control "Attribution" para OpenStreetMap sigue apareciendo como licencia para los datos CC-by-SA cuando en la sección 2.2.3 hemos señalado que desde Abril de 2012 el tipo de licencia ha variado. Es posible cambiar ese texto de manera manual, pero a cambio tendremos que indicar la url del servicio OSM también a mano, lo cual no es demasiado práctico, si esta variara por cualquier motivo, los mapas de nuestra aplicación dejarían de funcionar. Para este caso creo que la mejor elección es esperar a que en las próxima actualización de OpenLayers se corrija.

De cualquier modo, si se quisiera realizar este cambio se debería de cambiar la forma de cambiar la capa base que se incluye en el mapa utilizando `new OpenLayers.Layer.OSM('name',url,options)`, y dentro de options `{'attribution': 'Nombre y enlace a mostrar'}`.

2.4. Funciones y comunicación entre ellas

En los apartados anteriores hemos establecido las bases de cómo va ha ser la arquitectura de nuestra aplicación, y de cómo hacer uso en ella de algunos recursos externos que nos permitan obtener un mejor resultado final.

Esta es la primera vez que vamos a comentar parte de las funciones que se han desarrollado para implementar la aplicación web en la que se centra el proyecto, y por ello haremos algunos consideraciones previas.

Primero mostraremos un esquema de las funciones que intervienen para cumplir cada objetivos que nos hemos propuesto anteriormente. Se detallarán las principales líneas de código, explicando su funcionalidad y comentando en el caso de que sea necesario, las posibles dificultades que surgieron durante su implementación y cómo se solucionaron.

En el apéndice B, tenemos el Manual de Usuario de la aplicación, por lo que a lo largo de la memoria no explicaremos de forma explicita cómo utilizarla, ya que esa información está detallada en la parte final.

2.4.1. Desarrollo de la aplicación

Dentro de nuestra aplicación web, en este apartado nos centraremos en la pestaña *"EXPORTAR"*, desde la cual podremos crear nuestros escenarios de simulación en el formato de MATSim. Vamos a comenzar por comentar la funcionalidad básico con la que se le ha dotado para este fin. En la figura 2.9 tenemos una visión del sistema desde el punto de vista del usuario (en cada parte de la aplicación, que se puede hacer).

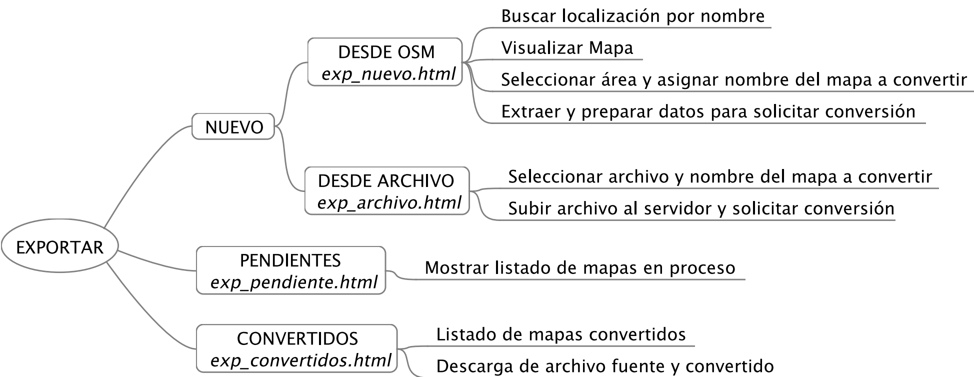


Figura 2.9: Esquema de la funcionalidad de la pestaña "EXPORTAR".

A partir del anterior esquema, pasamos a detallar algunos de los servicios más importantes que aparecen y el código que está detrás de ellos.

Visualización del Mapa

Al cargar la página web que contiene la aplicación se hace uso de la librería `/recursos/mapa_ol.js` la cual contiene la funcionalidad de OpenLayers explicada en la sección 2.3.3 que permite visualizar el mapa en el DIV de HTML que hemos definido a tal efecto.

Búsqueda por nombre de localización

La aplicación permite que seleccionemos directamente un porción del mapa moviéndonos sobre él con la barra de zoom y el cursor del ratón, pero además se han incluido las funciones necesario para poder buscar por nombres de posiciones geográficas y facilitar así el trabajo al usuario. Dentro de la librería `/recursos/buscar_nominatim.js` tenemos la función JavaScript `search()` como podemos ver a continuación:

```
1  function search() {
2    var search_string = document.getElementById("search_string").
         value;
3    var url = "http://nominatim.openstreetmap.org/search";
4    $.getJSON(url, {format: "json", q: search_string}, function(
         data) {
5          if (data.length == 0) {
6            alert("No se encontraron resultados");
7        document.getElementById("search_string").value="";
8          }
9          else {
10       var contenedor = document.getElementById('resultados');
11       var i=0;
12       //anyadimos div para cerrar resultados sin aceptar ninguno
13       var nuevodiv = '<div id="cerrar_busqueda" class="cerrar"
            onclick="cerrar()">Cerrar<\div>';
14       contenedor.innerHTML += nuevodiv;
15
16       for (i=0;i<data.length;i++)
17       {
18         if(i<10)
19         {
20           var nuevodiv = '<div class="cuadro" id="busqueda'+i+'
                " onclick="cargarMap('+data[i].boundingbox[0]+','+
                data[i].boundingbox[1]+','+data[i].boundingbox[2]+
                ','+data[i].boundingbox[3]+');cerrar();"><p>'+data
                [i].display_name+' ('+data[i].class+')</p></div>';
21           contenedor.innerHTML += nuevodiv;
22         }
23         else
24         {
25           var nuevodiv = '<div id="error_10" class="
                cerrar_error" onclick="cerrar()">Hay mas de 10
                resultado, realice una busqueda mas esfecifica
                ...<\div>';
```

```
26              contenedor.innerHTML += nuevodiv;
27              break;
28            }
29          }
30              var maskHeight = $(document).height();
31              var maskWidth = $(window).width();
32              $('#fondo').css({'width':maskWidth,'height':
                   maskHeight});
33              $('#fondo').fadeTo("slow",0.5);
34              var winH = $(window).height();
35              var winW = $(window).width();
36
37              $('#resultados').css('top',  winH/2-$('#resultados').
                   height()/2);
38              $('#resultados').css('left', winW/2-$('#resultados').
                   width()/2);
39              $('#resultados').fadeIn(2000);
40          }
41        });
42  }
43
44  //Cierra ventana modal de la busqueda
45  function cerrar() {
46    document.getElementById("resultados").innerHTML="";
47    $('#resultados').fadeOut(250, function() {
48      $('#fondo').fadeOut(250);
49    });
50    document.getElementById("search_string").value="";
51  }
```

- En la línea 2, recuperamos el texto que el usuario a escrito para buscar.
- En la línea 3, almacenamos la url del servicio Nominatim que ofrece OpenStreetMap.
- En la línea 4, utilizamos al función .getJSON de la librería jQuery, para realizar una petición Ajax a la url anterior, indicando como parámetro que el formato de los datos que nos devuelvan sea JSON.
- Entre las líneas 5-14, tratamos la respuesta recibida y preparamos los datos para ser añadidos a unos nuevos DIV que iremos creando.
- Entre las líneas 16-29, creamos tantos elementos DIV como número de registros hayamos obtenido en la consulta. Cada DIV contiene una llamada a la función js *cargarMap* con las coordenadas de cada registro, para que al pulsar, se cargue esa ubicación en el mapa principal Estos DIV se insertan dentro de otro DIV que actuará a modo de ventana modal.
- Por último entre las líneas 30-51, manipulamos el DIV que actúa de ventana modal para adaptarlo al tamaño de la pantalla y poder cerrarlo.

Seleccionar área

A la funcionalidad básica que tenemos en `/recursos/mapa_ol.js` para trabajar con el mapa, añadimos dos funciones nuevas para poder gestionar cuando queremos seleecionar un área, y cuando queremos movernos por el mapa con el ratón y para poder recuperar las coordenadas de rectángulo que seleccionemos.

```
function selecArea(accion){
  estado_selec=document.getElementById('seleccionarBox');
  if (accion=="salir"){
    estado_selec.checked=0;
    $("cont-coordenadas").slideUp('fast');
  }
  if (estado_selec.checked)
    control.box.activate();
  else
    control.box.deactivate();
}

function rellenar_coord(left_top, right_bottom)
{
  var coor1 = transformToWGS84(left_top.lon, left_top.lat);
  var coor2 = transformToWGS84(right_bottom.lon, right_bottom.lat);

  document.getElementById('lb-top').value=coor1.lat.toFixed(4);
  document.getElementById('lb-bottom').value=coor2.lat.toFixed(4);
  document.getElementById('lb-left').value=coor1.lon.toFixed(4);
  document.getElementById('lb-right').value=coor2.lon.toFixed(4);

  document.getElementById('coordenadas').style.display = 'block';
}
```

- Entre las líneas 1-11, la función `selecArea()` es llamada para activar o desactivar el control que gestiona la selección en el mapa.
- Entre las líneas 13-24, tenemos la función `rellenar_coord`, la cual es llamada cuando OpenLayers detecta el evento correspondiente a soltar el ratón, y además está activo el control de selección. Recibe como parámetro dos esquinas del rectángulo, extrae las cuatro coordenadas límite y muestra un DIV con esa información.

Solicitud de descarga y conversión

Hasta ahora, las anteriores funciones aunque hacían uso de recursos externos, funcionaban sin realizar ninguna comunicación con nuestro servidor, salvo la carga

inicial de la página web y sus recursos asociados.

Como el siguiente proceso será el primero que utilice nuestra solución tanto en el cliente como en el servidor, se explicará de manera más detallada que las sucesivas.

En la figura 2.10 podemos ver el camino que sigue nuestro aplicación y todas las partes que intervienen para conseguir su funcionalidad final.

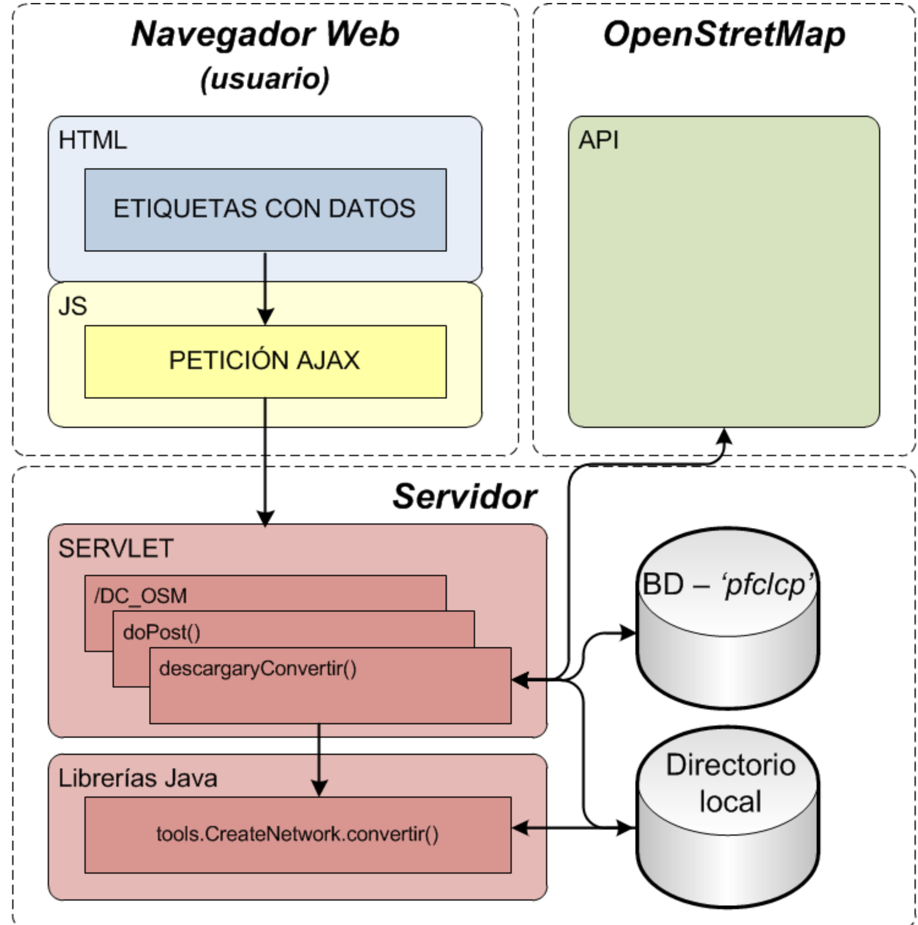


Figura 2.10: Diagrama proceso de descarga y conversión de mapas.

Siguiendo este diagrama, vemos que del lado del usuario, se recopilan los datos y se realiza una petición Ajax a uno de los Servlet que contiene el servidor. Esta petición es la siguiente:

```
function descargarConvertir() {
  var ctop=document.getElementById('lb-top').value;
  var cbottom=document.getElementById('lb-bottom').value;
  var cleft=document.getElementById('lb-left').value;
  var cright=document.getElementById('lb-right').value;

  var type = "osm";

```

```
9     var nombre_mapa=document.getElementById('name_exportar').
         value;
10
11    if ((ctop=='')||(cbottom=='')||(cleft=='')||(cright==''))
12    {
13      alert("Debe seleccionar un area del mapa a exportar.");
14    } else if (nombre_mapa==''){
15      alert("Debe escribir el nombre del mapa a exportar.");
16    } else {
17      var ajax_data = {
18      "user": "invitado",
19      "pass": "invitado",
20      "name": nombre_mapa,
21      "type": type,
22      "top": ctop,
23      "bottom": cbottom,
24      "left": cleft,
25      "right": cright
26    };
27
28    ajax({
29      url: "http://localhost:8080/pfc_servlet/DC_OSM",
30      data: ajax_data,
31      type: "post"
32      });
33    }
34  }
```

- Lineas 1-16, recuperemos los datos necesarios de los diferentes elementos de la página, y avisamos ante posibles errores.
- Lineas 17-25, creamos una variable que contenga los datos de la petición que vamos a realizar. (user y pass, se fuerzan como invitado, ya que aunque no lo utilizamos en nuestra aplicación, las funciones están pensadas para utilizarse en otros sistemas que requieran autenticación para el acceso a los recursos del servidor).
- Líneas 28-32, realizamos la petición Ajax indicando la url del Servlet, los datos que hemos almaceno antes, y tipo de petición.

Ahora llega el turno de ver el funcionamiento del Servlet que atiende esta petición.

Debemos de comenzar comentando una decisión de implementación que se tomó al comenzar las el desarrollo de esta solución. No se han utilizado descriptores de implementación para determinar la forma en que las URL se asignan a los servlets, ya que se ha optado por indicarlo dentro del propio servlet con la sentencia `@WebServlet`, algo posible desde la versión 3.0[18]. En nuestro caso quedaría de la siguiente forma:

[18] http://docs.oracle.com/javaee/6/api/javax/servlet/annotation/WebServlet.html

```
@WebServlet(description = "Descarga y Convierte de OSM a formato MatSIM",
urlPatterns = { "/DC_OSM" })
```

En todos los Servlet que vamos a emplear, hay un denominador común, y es que las peticiones son atendida por los métodos *doGet* o *doPost* dependiendo del tipo de solicitud. Dentro de estos métodos, extraemos los parámetros de la petición de la siguiente forma:

```
1  protected void doPost(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
      IOException {
2    String top = request.getParameter("top");
3    String bottom = request.getParameter("bottom");
4    String left = request.getParameter("left");
5    String right =request.getParameter("right");
6
7    String type = request.getParameter("type");
8    String name = request.getParameter("name");
9
10   String user = request.getParameter("user");
11   String pass = request.getParameter("pass");
12
13   if(type.equals("osm"))
14     descargarYConvertir (user,pass,name,left,bottom,right,top);
15 }
```

- Debemos fijarnos en los atributos *request* y *response*, ya que son los que nos van a permitir interactuar con el cliente que realizó la petición.
- Líneas 2-11, utilizamos el método `getParameter` para recuperar los parámetros de la petición.
- Línea 14, llamamos a la función que realiza la conversión.

Por último, vamos a explicar la implementación de la función que realiza la conversión.

```
1  public class CreateNetwork {
2    /*
3     * Basado en ejemplo www.matsim.org
4     */
5    public String Convertir(String origen_osm,String
        destino_matsim){
6      Config config = ConfigUtils.createConfig();
7        Scenario sc = ScenarioUtils.createScenario(config);
8        Network net = sc.getNetwork();
9        CoordinateTransformation ct =
10         TransformationFactory.getCoordinateTransformation(
11         TransformationFactory.WGS84, TransformationFactory.
             WGS84_UTM33N);
12       OsmNetworkReader onr = new OsmNetworkReader(net,ct);
```

```
13
14          try {
15            onr.parse(origen_osm);
16            new NetworkCleaner().run(net);
17            new NetworkWriter(net).write(destino_matsim);
18            }
19          catch ( Exception e ) {
20            return ("Error Conversion");
21          }
22          return ("OK");
23      }
24  }
```

Al utilizar funciones de la librería de MATSim, estas deben de incluirse y compilarse juntas.

- Líneas 6-8, creamos los objetos necesarios para llegar a tener uno de tipo *Network*, que posteriormente nos permitirá crear la red en el formado de MATSim.
- Líneas 10-11, creamos un objeto de transformación de coordenadas, indicándoles las origen y las de partida. Es importante resaltar que como se puede ver, en la documentación de MATSim utiliza `WGS84_UTM33N`. La zona UTM 33N se corresponde a la parte de Alemanía y Suiza, de donde son los creadores de MATSim. Se podría variar esta zona, pero realmente no aporta demasiado ya que siempre se realizará posteriormente la conversión inversa recuperando las coordenadas reales, y sin embargo el utilizar diferentes zonas UTM puede llevar a errores a la hora de compartir archivos.
- Línea 12, creamos un objeto para leer mapas de tipo OSM.
- Líneas 15-17, analizamos el archivo de tipo OSM que le pasamos, y generemos y almacenamos el nuevo archivo, con la información del escenario en formato de MATSim.

Subir archivos y convertirlos

En el apartado 2.2.5, hablábamos de las posible limitaciones que tenemos a la hora de solicitar fragmentos de mapas a OpenStreetMap. También comentamos que existe la posibilidad de descargar porciones de mapa mucho mayores y extraer de ellas el área que necesitáramos. Aunque se realizaron pruebas, y se llegaron a implementar esta funcionalidad en la solución, los resultado son eran satisfactorios debido a la gran cantidad de tiempo de procesado que era necesario, y a el difícil mantenimiento de archivo de incluso varios cientos de GB, que además debían de ser actualizados de manera periódica.

Por este motivo se ha optado por el camino de incluir en la aplicación la posibilidad de que el usuario pueda subir los archivos de tipo OSM, que en cualquier momento haya podido necesitado generar, y se han convertidos e incluidos en el sistema de manera análoga a cuando son solicitados de manera remota.

En esta parte, debemos destacar como enviar un archivo al servidor, sin necesidad de recargar la página y controlando la respuesta reciba para cerciorarnos de que el archivo se ha subido correctamente.

Incluiremos un elemento **HTML input** de tipe *"file"* tal que así `<input id="archivo" type="file" />`, del que luego recogeremos la información del archivo a subir.

Haremos uso de la intefaz XMLHttpRequest[19], para formar en el cliente la solicitud al servidor, y gestionar su respuesta de la siguiente manera:

```
1  function subirArchivo()
2  {
3    var nameArchivo = document.getElementById("name_convertir").
       value;
4    var archivoaSubir = document.getElementById("archivo").files
       [0];
5
6    if (nameArchivo==""||archivoaSubir=="")
7    {
8      alert("Rellene los dos campos.")
9    }else{
10     var formdata = new FormData();
11
12     formdata.append("user","invitado");
13     formdata.append("pass","invitado");
14     formdata.append("nombre", nameArchivo);
15     formdata.append("archivo", archivoaSubir);
16
17     var xmlhttp;
18     if (window.XMLHttpRequest)
19     {//para los navegadores modernos
20       xmlhttp=new XMLHttpRequest();
21     }
22     else
23     {//si seguimos en la prehistoria
24       xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
25     }
26
27     xmlhttp.onreadystatechange=function()
28     {
29       if (xmlhttp.readyState==4)
30       {
31         if(xmlhttp.status==200){
32           alert("El archivo se ha subido.");
33         }
34         else
35         {
36           alert("Error al subir el archivo.");
37         }
38           document.getElementById("name_convertir").value="";
39           document.getElementById("archivo").files[0]="";
40       }
```

[19] http://www.w3.org/TR/XMLHttpRequest/

```
41        }
42        xmlhttp.open("POST","http://localhost:8080/pfc_servlet/
              SubirOSM", true);
43        xmlhttp.send(formdata);
44      }
45    }
```

- Líneas 3-4, recogemos en dos variables el nombre que le va a dar el usuario al mapa, y el archivo que desea subir.
- Líneas 10-15, tras comprobar que no hay campos vacíos, creamos un objeto de tipo `FormData()` y almacenamos en el los parámetros que vamos a enviar en la petición.
- Líneas 17-25, creamos el objeto `XMLHttpRequest`, asegurándonos su funcionamiento para navegadores antiguos (IE5, IE6).
- Línea 27, llamamos al método `onreadystatechange` del objeto de tipo `XMLHttpRequest` para gestionar los cambio que se produzcan en el mismo.
- Líneas 29-40, cada vez que se produce una cambio en el objeto `XMLHttpRequest`, comprueba de que tipo es, en este caso nos interesa `readyState==4`, ya que indica que la petición se ha completado. Además leeremos el estado de la respuesta y comprobaremos que `status==200` para certificar que servidor nos ha contestado que todo fue correcto.
- Líneas 42-43, son donde realmente iniciamos la solicitud, y enviamos los datos.

```
1   protected void doPost(HttpServletRequest request,
       HttpServletResponse response) throws ServletException,
       IOException {
2   //capturamos posibles excepciones y las mostramos por consola
3     String user="";
4     String pass="";
5     String nombre="";
6     PrintWriter out;
7
8     response.setHeader("Cache-Control","no-store");
9     response.setHeader("Access-Control-Allow-Origin","*");
10    response.setContentType("text/plain");
11
12    String ruta_raiz = "/usr/local/pfc/mapas";
13
14    Usuarios comprobacion_user = new Usuarios();
15
16    try {
17      boolean isMultipart = ServletFileUpload.isMultipartContent(
            request);
18
19      if(isMultipart)
20      {
21        FileItemFactory factory = new DiskFileItemFactory();
```

```
22        ServletFileUpload upload = new ServletFileUpload(factory)
             ;
23        List items = upload.parseRequest(request);
24
25        Iterator iter = items.iterator();
26        while (iter.hasNext())
27        {
28          FileItem item = (FileItem) iter.next();
29          if (item.isFormField())
30            if(item.getFieldName().equals("user")){
31            user=item.getString();
32            }else if (item.getFieldName().equals("pass")){
33            pass=item.getString();
34            }else if (item.getFieldName().equals("nombre")){
35            nombre=item.getString();
36          }else{
37            if(comprobacion_user.comprobarUsuario(user, pass))
38            {
39              //Creamos una carpeta (si no esta ya)
40              Archivos nuevoDir = new Archivos();
41
42              nuevoDir.crearDirectorio(ruta_raiz,user);
43              String ruta_usuario=ruta_raiz+"/"+user;
44
45              //Y creamos otra carpeta con el nombre del mapa
46              if (nuevoDir.crearDirectorio(ruta_usuario, nombre)
                   ==true)
47              {
48                //subimos el archivo
49                String ruta_archivo=ruta_usuario+"/"+nombre;
50                item.write(new File(ruta_archivo+"/"+"map.osm"));
51
52                //Guardamos el registro en la BD
53                AccesoBD nuevo_reg = new AccesoBD();
54                nuevo_reg.altaMapa(user, pass, nombre, "0", "0",
                     "0", "0", ruta_archivo, "0");
55                nuevo_reg.modificacionValor(user, pass, nombre, "
                     estado", "Subido por usuario");
56                nuevo_reg.modificacionValor(user, pass, nombre, "
                     estado", "Convirtiendo");
57
58                CreateNetwork convert = new CreateNetwork();
59                convert.Convertir(ruta_archivo+"/map.osm",
                     ruta_archivo+"/network.xml");
60                nuevo_reg.modificacionValor(user, pass, nombre, "
                     estado", "OK");
61
62                response.setStatus(200);
63                out = response.getWriter();
64                out.println("OK");
65                out.close();
66              }
67              else
68              {
69                response.setStatus(400);
70                out = response.getWriter();
```

```
71                  out.println("Nombre del mapa ya creado");
72                  out.close();
73                }
74                }
75            }
76          }
77        }
78      } catch (Exception e) {
79        e.printStackTrace();
80        response.setStatus(400);
81        out = response.getWriter();
82        out.println(e.toString());
83        out.close();
84      }
85    }
```

- Líneas 8-9, configuramos los parámetros de la cabecera de la respuesta que se le devolverá al cliente. Es un paso importante, ya que si no se hiciera, el cliente no *"reaccionaría"* ante la respuesta.
- Línea 17, comprueba si la petición que le llega contiene información Multipart, es decir si son solo parámetros de texto o contiene archivos.
- Líneas 25-26, en caso de que contenga información de archivos, separamos los diferentes elementos que se envían y los vamos recorriendo uno a uno.
- Línea 29, con el método `.isFormField()`, si nos devuelve `FALSE` ese elemento es un archivo.
- Línea 50, si es un archivo, tras haber preparado el directorio correspondiente, guardamos el archivo recibido con el método `write()`, pasándole como parámetro la ruta local donde queramos guardarlo.

En este punto, una vez guardado el archivo en su directorio, el proceso de conversión es el mismo que para el apartado anterior.

Debemos remarcar, las líneas 62 y 69, donde en cada caso asignamos un valor al estado de la respuestas y un mensaje asociado, para que el cliente pueda saber si la subida del archivo se completó de manera correcta, o en caso de que haya producido un error, ver el mensaje asociado.

Mostrar estado peticiones pendientes

Siempre que realizamos una petición, transcurre un tiempo hasta que recibimos una respuesta. En algunos ocasiones este tiempo es muy reducido, pudiendo avisar al usuario del resultado de su consulta instantes después de su solicitud.

En nuestro caso, hemos visto que se deben de realizar varios pasos en los que intervienen diferentes partes del sistema, y procesar en ocasiones grandes cantidades

de información que producen que el tiempo de procesamiento se alargue demasiada para tener a un usuario pendiente de la respuesta. Por ello, se ha incluido la funcionalidad para que permite al usuario realizar todas las peticiones que necesite, y posteriormente ver en que estado está cada una de ellas.

Para implementarlo se hará la solicitud al servidor, el cual nos devolverá un JSON con el listado de las tareas en cursos y sus estado.

```
function tareas_pend() {
  var url = "http://localhost:8080/pfc_servlet/ConsultaMapasProceso";
  $.getJSON(url, {user: "invitado", pass: "invitado",clase: "pendientes"}, function(data) {
    if (data.length == 0) {
      document.getElementById('mensaje_nohay').style.display = 'block';
    }
    else
    {
      document.getElementById('mensaje_sihay').style.display = 'block';
        var contenedor = document.getElementById('tareas_pendientes');
        var i=0;
      for (i=0;i<data.length;i++)
      {
        var nuevodiv = '<div class="pendiente" id="busqueda'+i+'"><p>'+data[i].nombre+' -> Estado: '+data[i].estado+' <br> Long: '+data[i].arriba+'/'+data[i].abajo+' Lat: '+data[i].izq+'/'+data[i].der+' -> '+data[i].fecha+'</p></div>';
        contenedor.innerHTML += nuevodiv;
      }
    }
  });
}
```

- En la línea 3, se llama a la función de jQuery `$.getJSON`, que realiza la petición Ajax al Servlet `ConsultaMapasProceso`, y a la vez trata el JSON de respuesta, el cual recorremos y mostramos. Si nos fijamos, enviamos el parámetro *clase:"pendientes"*, ya que el mismo Servlet nos puede devolver los mapas que se encuentras en proceso, y los que ya han finalizado.
- En la línea 4, comprobamos si el JSON no tiene registros, para hacer visible el DIV que contiene un mensaje para estos casos.

A continuación tenemos un fragmento del Servlet `ConsultaMapasProceso` que después comentaremos:

```
if(comprobacion_user.comprobarUsuario(user,pass)){
  Connection con=null;
```

```
3    Statement stmt=null;
4
5    try {
6      Class.forName("com.mysql.jdbc.Driver");
7      con = DriverManager.getConnection("jdbc:mysql://localhost/
          pfclcp","root","");
8
9      stmt = con.createStatement();
10   }catch (Exception e){
11     System.out.println("Error al establecer conexion con BD: "+
          e.getMessage());
12   }
13
14   try{
15     String clase = request.getParameter("clase");
16     String texto_clase="";
17     if (clase.equals("pendientes"))
18     {
19       texto_clase="estado!='OK'";
20     }
21     else if (clase.equals("convertidos"))
22     {
23       texto_clase="estado='OK'";
24     }
25
26     ResultSet rs = stmt.executeQuery ("SELECT * FROM mapas
          WHERE usuario='"+user+"' and "+texto_clase);
27
28     JSONArray list = new JSONArray();
29
30     while (rs.next()){
31       JSONObject obj=new JSONObject();
32       obj.put("nombre",rs.getString("nombre"));
33       obj.put("arriba",rs.getString("coord_arriba"));
34       obj.put("abajo",rs.getString("coord_abajo"));
35       obj.put("izq",rs.getString("coord_izq"));
36       obj.put("der",rs.getString("coord_der"));
37       obj.put("fecha",rs.getString("fecha"));
38       obj.put("estado",rs.getString("estado"));
39       obj.put("ruta",rs.getString("ruta"));
40
41       list.add(obj);
42     }
43     rs.close();
44
45     StringWriter out = new StringWriter();
46     list.writeJSONString(out);
47
48     response.setHeader("Cache-Control","no-store");
49     response.setHeader("Access-Control-Allow-Origin","*");
50     response.setContentType("application/json; charset=UTF-8");
51
52     response.getWriter().write(out.toString());
53     response.getWriter().close();
54   }catch (Exception e){
55     System.out.print("Error al recuperar datos de BD:"+e);
```

```
56    }
57  }else{
58    response.setHeader("Cache-Control","no-store");
59    response.setHeader("Access-Control-Allow-Origin","*");
60    response.setContentType("application/json; charset=UTF-8");
61
62    response.getWriter().write("[]");
63    response.getWriter().close();
64  }
```

- Líneas 5-12, tratamos de establecer conexión con la base de datos llamada *“pfclcp”* utilizado el conector JDBC:MYSQL.
- Líneas 17-23, comprobamos si nos solicitan recuperar los datos de los procesas que están pendientes o finalizados.
- Línea 23, realizamos la consulta la base de datos.
- Líneas 28-42, utilizamos los objetos `JSONArray` y `JSONObject` para ir recogiendo los resultado de la consulta en formato JSON.
- Línea 50, indicamos que el contenido que se envía como respuesta es un JSON y el tipo de codificación en el que esta. (si no lo indicáramos, el cliente no tendría por qué saberlo, y a no ser que se forzara la lectura lo omitiría). Es fundamental realizar cualquier asignación de la cabeza del mensaje de respuesta antes de comenzar a enviar datos.
- Línea 52, por último enviamos la estructura JSON que hemos extraído del objeto `JSONArray` en la línea 46.

Ver mapas convertidos y descargar los archivos

Una vez finalizado del proceso de conversión, podremos consultar el listado de igual forma que hacíamos en el apartado anterior para los pendientes, simplemente variando el parámetro de la petición que tiene esa función (*clase: convertido*)

Ahora llega el momento de descargar los archivos convertidos. El sistema va almacenando en su directorio local de manera ordenada los mapas en su formato original y convertido. Por lo que simplemente necesitamos que desde el cliente se solicite cual de los dos quiere descargar, y de qué nombre de mapa, para poder enviárselo.

Para ello, una vez recuperado el listado de añadiremos los enlaces modificados para dato recuperado que siguiendo la siguiente estructura, para lanzarla petición el Servlet *Descargar*: `http://localhost:8080/pfc_servlet/Descargar ?user=invitado&pass=invitado&nombre='NombreDelMapa'&tipo=osm/matsim"`

Donde *tipo* nos permite elegir si queremos descargar el mapa original o en formato MATSim, que tengan asociado el nombre indicado (los archivos originales se envían con el nombre map.osm, y los convertidos con network.xml).

En el lado del servidor, dentro del Servlet *Descargar*, tras recibir y separar los parámetro, y comprobar que el directorio que le correspondería al mapa solicitado existe, abrimos un flujo de lectura del tipo de archivo elegido, lo recorremos y volcamos en el response de vuelta al cliente de la siguiente forma:

```
try{
  FileInputStream fileInputStream = new FileInputStream(ruta+"/
     "+nombre_archivo);

  ServletOutputStream out = response.getOutputStream();
  String mimeType =  new MimetypesFileTypeMap().getContentType(
     ruta+"/"+nombre_archivo);

  response.setContentType(mimeType);
  response.setContentLength(fileInputStream.available());
  response.setHeader("Content-Disposition","attachment;filename
     ="+nombre_archivo);

  int c;
  while((c=fileInputStream.read()) != -1){
    out.write(c);
  }
  out.flush();
  out.close();
  fileInputStream.close();
}catch(Exception e){
  e.printStackTrace();
  response.setContentType("text/html");
  PrintWriter resp_error = response.getWriter();
  resp_error.println("Error al descargar archivo:"+e);
  resp_error.flush();
  resp_error.close();
}
```

De este fragmento de código cabe destacar:

- Línea 4, creamos un objeto de tipo `ServletOutputStream` para poder *recoger* byte a byte los datos que vayamos leyendo.
- Línea 5, recuperamos la información del archivo en cuestión, para poder incluirla en la cabeza en la línea 7.
- Líneas 8-9, guardamos en la cabecera el tamaño del archivo y su nombre.
- Líneas 12-14, recorremos el flujo de lectura del archivo y lo escribimos byte a byte en el objeto creado en la línea 4.
- Línea 21, enviamos los datos leídos, con sus cabecera asociada.

Además en cada registro, se ha habilitado un enlace para poder cargar las coordenadas del mapa y poder visualizarlo. Esta opción no funciona con los mapas que hayamos subido de manera manual, ya que en el proceso de conversión no se leen los límites del mapa, por lo que ese dato no se tiene.

Capítulo 3

Generación de la población del escenario

En el capitulo anterior hemos podido ver los mecanismos necesarios para la creación de un escenario de simulación que cumpliera nuestras necesidades.

El segundo conjunto de datos de entrada que tenemos que proporcionar al simulador MATSim para comenzar a trabajar es la información relativa a la población que se *desplaza* por el escenario, es decir, los agentes qué circulas por é, indicando su situación inicial, final y la hora a la que comienza su ruta.

En este capítulo veremos cómo hemos alcanzado este objetivo en nuestra aplicación.

3.1. Introducción

El caso ideal de simulación se produciría si contáramos con absolutamente todos los datos que intervienen en el sistema, desde cualquier tipo de característica del escenario, hasta todos y cada uno de los elementos que interactúan en él. Este supuesto es inviable, primero porque el tiempo y los recursos que necesitaríamos para recopilar esa información serían ingentes, y segundo porque la cantidad de variables con las que tendríamos que tratar haría que el volumen de datos fuera desmesuradamente elevado.

Una solución a este problema, es la modelización del escenario a través de datos reales, con lo que sería posible, eligiendo cuidadosamente el espacio maestral, elaborar una modelo de la población que reside en una determinado zona, situándolos e indicando los desplazamiento típicos que realizan (en la documentación de MATSim, se habla por ejemplo, de un modelo real realizado a partir del censo de población de Suiza del año 2000, y un estudio que recoge desplazamientos diarios de

una muestra de 30000 personas[1]). Este tipo de modelos, resulta muy costoso tanto por tiempo requerido como por los recursos necesario para la obtención de los datos. Además para determinadas simulaciones sobre supuestas situaciones es totalmente inviable por la ausencia de datos reales de contraste.

Sin necesidad de llegar al extremo anterior, si que podemos tener unas nociones de cómo se comporta la población, sabiendo por ejemplo las zonas del escenario que son residenciales, zonas más industrializadas, accesos a autopistas, zonas escolares. De igual forma, existen determinadas horas del día donde se produce más actividad dependiendo del tipo de zona donde nos encontremos. Combinando estas dos características, podemos llegar a describir el comportamiento global de una población de manera bastante precisa.

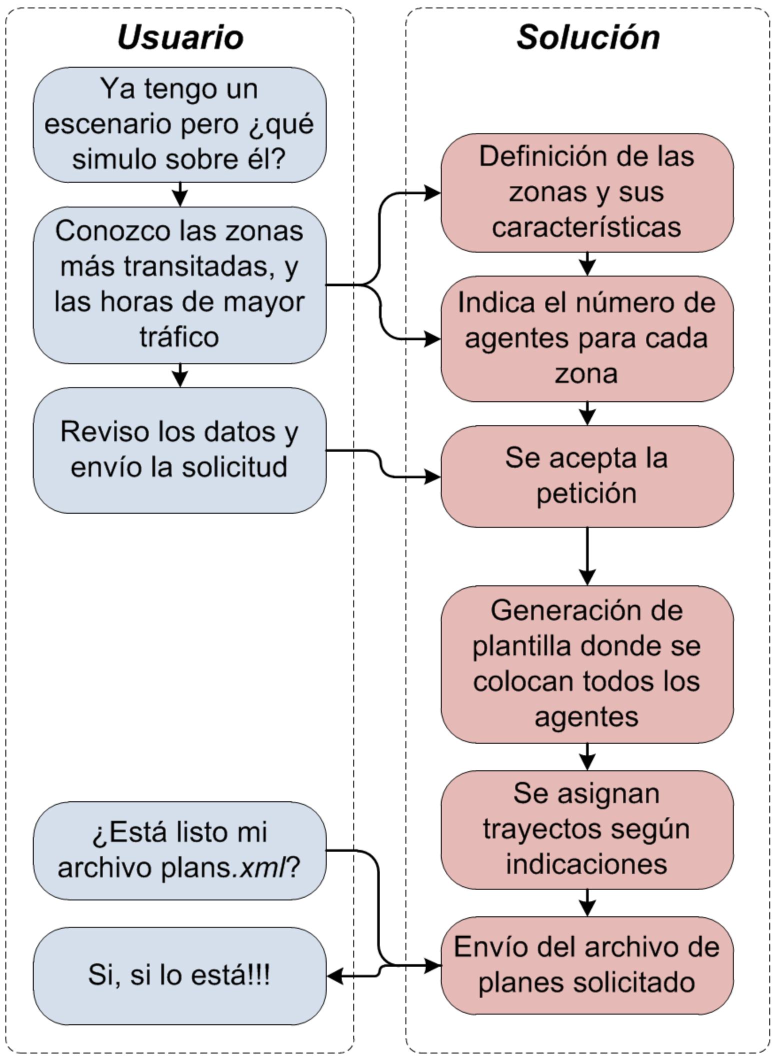


Figura 3.1: Esquema de los pasos para crear un archivos de planes.

El problema surge en como trasladar esos datos empíricos con los que conta-

[1] http://www.matsim.org/node/603

mos, a un listado exacto donde se indique cada integrante de la población, en que punto de escenario se sitia, a que hora inicia su recorrido, los puntos intermedios por los que transita, y en que punto finaliza.

MATSim no incorpora ninguna funcionalidad específica que pueda solventar esta necesidad, y por ello en muchas ocasiones era necesario elaborar estos archivos de planes, que contenían esa descripción, de manera totalmente manual.

Poniéndonos en el lado del usuario, en la figura 3.1 queda reflejado las preguntas iniciales y los pasos que seguiría para crear su archivo de planes personalizado, que contenga la situación, y trayectos que realizan la población "tipo" de su escenario de simulación, frente a las tareas, vista de forma general, que realizaría el servidor.

En la siguiente sección, vamos a explicar el sistema y las funciones implementadas que se han elaborado para automatizar este proceso.

3.2. Funciones y comunicación entre ellas

En el capítulo anterior, hemos explicado por primera vez el tipo de sistema en el que estamos trabajando y cómo hacer llamadas desde el cliente al servidor, para que sean atendidas por los Servlets, y recibamos respuesta de ellos. Hay muchas líneas de código que se deben de repetir en todas las funciones, y otras en las que utilizamos métodos bastantes estandarizados para resolver tareas comunes, pero que aportan poca información sobre las funcionalidades específicas de este trabajo. Por ejemplo, dentro de la sección 2.4.1, hemos visto cómo puede el cliente subir y descargar archivo realizando peticiones al servidor. Este paso se repetirá con relativa frecuencia.

Por tanto, vamos a pasar a explicar las partes del código que más aportan, y que son específicas de la tarea que nos ocupa.

3.2.1. Desarrollo de la aplicación

En la pestaña *"PLANES"*, vamos a contar con todas las herramientas que nos permitirán generar un archivo con datos de población creados a partir de las directrices que el usuario haya indicado. En la figura 3.2 tenemos la estructura y funcionalidad que está disponible para esta tarea desde el punto de vista del usuario.

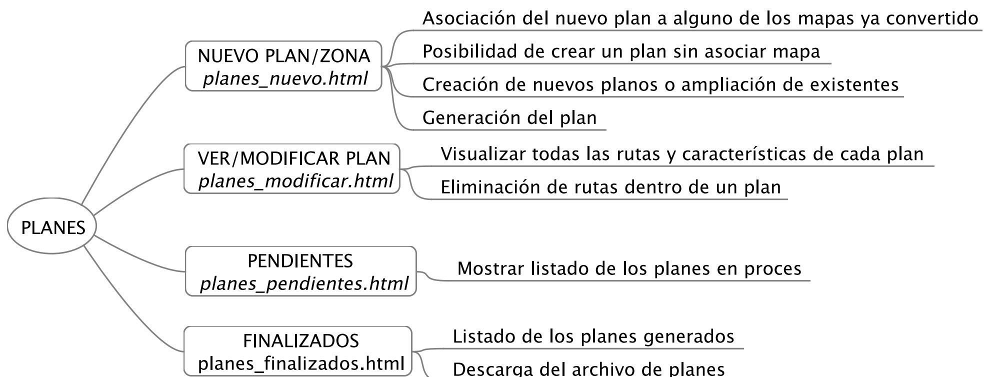


Figura 3.2: Esquema de la funcionalidad de la pestaña "PLANES".

Los servicios disponibles en las secciones *PENDIENTES* y *FINALIZADOS*, están desarrollados siguiendo el mismo proceso que los que veíamos para la pestaña de *EXPORTAR*. Aunque han sido modificados y adaptados, ya que existen ligeras variaciones, no volveremos a explicarlos ya que no presentan ninguna novedad.

Por tanto vamos a centrarnos en comprender el proceso que seguimos para generar los planes, y sus funciones más destacadas.

Generación de planes

En la introducción se ha explicado que aunque no tengamos toda la información sobre como se comporta la población de un determinado escenario, si podemos llegar a hacer una buena aproximación de las zonas más transitadas, y de los horarios donde se producen la mayor parte de los desplazamientos.

El usuario una vez seleccionado el mapa y creado el plan, podrá añadir tantas rutas como necesita. Cada ruta a su vez, podrá estar compuesta por todos las zonas que se deseen incluir haciendo uso de las funcionalidades que hemos visto anteriormente de OpenLayers. Para organizar el trabajo, la aplicación permitirá añadir de una en una las rutas compuestas por las zonas seleccionadas, pudiendo en todo momento añadir o eliminar zonas, y varias el orden y descripción de las mismas (Fig. 3.3).

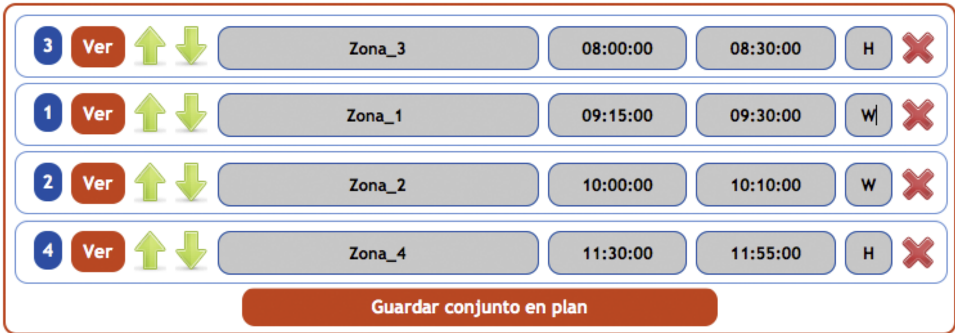


Figura 3.3: Organización de las zonas dentro de una misma ruta.

Una vez el usuario a terminado de introducir, colocar y catalogar las zonas, debe rellenar tanto la información de cada uno de las zonas (palabra que describa la zona, horario de inicio de desplazamiento, y el tipo de actividad que va a realizar), como la información general de la ruta (número de agentes que la componen y la descripción de la ruta). Nuestra aplicación, desde el lado del cliente, recogerá todos estos datos y organizará con ellos cadenas de texto con el siguiente formato:

```
0,350,Ruta1

1,40.5220,40.5070,-3.3534,-3.3347,08:00:00,08:30:00,Zona_3

2,40.5088,40.5043,-3.3380,-3.3311,09:15:00,09:30:00,Zona_1

3,40.4853,40.4759,-3.3761,-3.3558,10:00:00,10:10:00,Zona_2

4,40.4357,40.5137,-3.3854,-3.3658,11:30:00,11:55:00,Zona_4
```

Estas líneas las podemos clasificar en dos grupos, la que se inician con el valor “0”, y el resto. Las primeras contienen los datos de la ruta tal que:

```
0,Nº de Agentes,Descripción de la Ruta
```

El resto de cadenas de texto contienen la información de las zonas que componen la ruta. Están ordenadas comenzando por el punto de origen, y finalizando por el destino siguiendo la siguiente sintaxis:

```
Posición,arriba,abajo,izquierda,derecha,Hora Inicio,Hora Fin,Descripción
```

Todas estas cadenas se unen en una sola, separándolas con el carácter "&" a la hora de preparar los datas para ser enviados al servidor.

El servidor, a través del Servlet ***GuardarNuevaRuta***, recibe la petición del cliente y almacena los datos en la base de datos. Esta petición está formada por los siguiente campos:

- **usuario**, que realiza la petición.
- **contraseña**, del usuario (ya hemos comentado, que aunque nosotros no llegamos a diferenciar entre usuarios, la funciones se han implementado para ser capaces de diferenciar y validar a distintos usuarios).
- **mapa**, seleccionado desde un desplegable que ha recupera todos los mapas que hemos convertido anteriormente (o en su caso la con "Sin mapa asociado").
- **nombre**, del plan al que pertenece la ruta.
- **contenido**, cadena de texto donde el cliente a recopilado la información de todas las zonas definidas por el usuario. Para diferenciar entre diferentes zonas se utiliza el carácter delimitador "`&`", y para diferenciar entre diferentes elementos dentro de una misma zona se utiliza "`,`".

La figura 3.4, es un esquema de las funciones más importantes que hemos desarrollado en este apartado. Además podemos ver donde está ubicada cada una de ellas, y como se comunican entre sí.

La función JavaScript `guardar_nueva_zona()`, recopila todos los datos que el usuario ha ido introduciendo sobre la nueva ruta, los da el formato requerido y los incluye en la petición al Servlet *GuardarNuevaRuta*, el cúal los verifica y almacena dentro de la base de datos.

Cuando pulsamos el botón *Generar Plan*, desde el cliente se llama a la función JavaScript `nuevo_plan()`, que actúa de manera análoga a las otras función en el lado del clientes que hemos visto, recopila la información, enviándola y esperando recibir la respuesta. Por su parte, el Servlet *CrearPlanNuevo* gestiona la interpretación de las peticiones, y la llamada a otras funciones que realizan las tareas pesadas. Por lo que vamos a centrarnos en el funcionamiento de estas otras funciones.

Planes.class

En la librería `utils_lib.jar` hemos reunido las funciones que se han ido implementando, y que podemos necesitar en varios puntos del sistema.

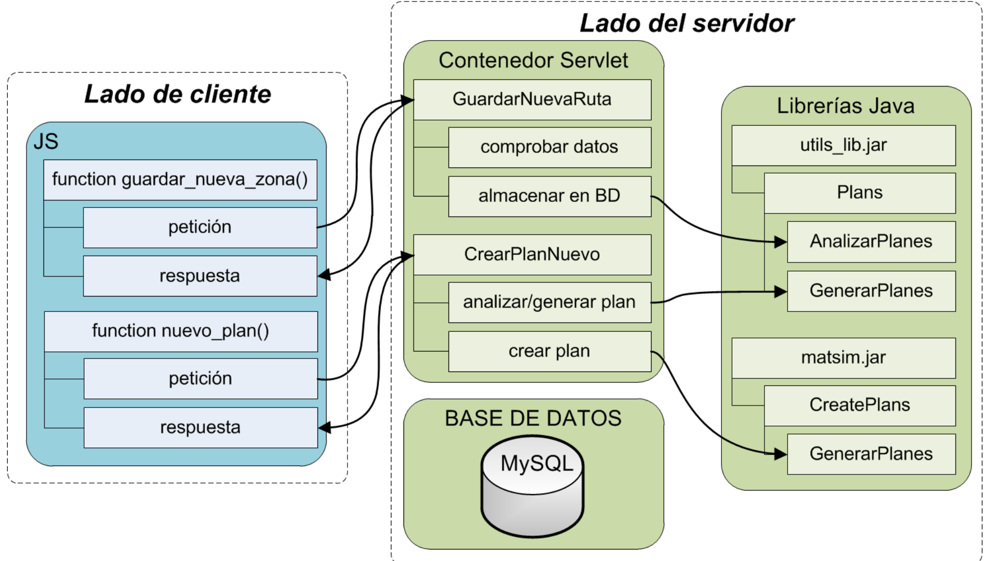


Figura 3.4: Comunicación entre las funciones básica para generar planes.

Dentro de la clase *Planes*, destacan dos de sus métodos:

- **AnalizarPlanes**: Realiza la petición a la base de datos de todas las rutas que han sido almacenadas de un determinado plan. De cada ruta, extrae la información relativa a las zonas que los componen. Con esta información, junto con la del número de agentes que integran cada ruta, llama al método asignar-Coord() de la clase. Tras contar con la posición de todos los agente, guarda los nuevos valores en la base de datos.
- **asignarCoord**: Recibe las coordenadas de un área y el número de agentes que se encuentran en ella, y genera un array bidimensional, de tamaño el número de agentes, con las ubicaciones de todos ellos (contenidas dentro del área).
- **GenerarPlanes**: Con los datos almacenados gracias al método *AnalizarPlanes*, comprueba que los datos han sido guardados correctamente, realiza varias consultas a la base de datos para organizar la información, y por último llama al método contenido en la clase *CreatePlans* para generar el archivo XML de planes final..

Debemos detenernos a analizar el código desarrollado en ciertas zonas de estas funciones, ya que es importante entender su funcionamiento.

El método *AnalizarPlanes*, simplemente gestiona las peticiones que le llegan, los accesos a la base de datos, y las peticiones otros métodos o funciones. Pero el método ***asignarCoord*** es muy importante para poder entender como repartimos a los agentes por el escenario.

A continuación mostramos su código, para comentarlo posteriormente y explicar el sistema que utiliza:

```
double [][] asignarCoord(double arriba, double abajo, double
    izq, double der, int num){
  //tantos elementos como agentes, y cada uno con
  //dos coordenadas lat, y long
  double [][] listaagentes=new double[num][2];

  double distx= Math.abs(izq-der);
  double disty=Math.abs(arriba-abajo);

  double filas=Math.sqrt((num*disty)/distx);
  double columnas=num/filas;

  double separacion=distx/columnas; //o disty/filas

  //tomamos como orgien arriba/izq
  //repartimos de manera homogenea el numero entero de
  //agentes de filas*columnas (siendo enteros)

  //COLUMNAS, distx
  int numcolumnasentero=(int)columnas;
  double posinicialcolumna=(distx/2)-((separacion*
     numcolumnasentero)/2);
  double posinicialx=izq+posinicialcolumna;

  //FILAS, disty
  int numfilasentero=(int)filas;
  double posinicialfila=(disty/2)-((separacion*numfilasentero)
     /2);
  double posinicialy=arriba+posinicialfila;

  int contador=0;
  //equivale a la longitud [][0]
  for (int i=0;i<numcolumnasentero;i++){
    //equivale a la latitud [][1]
    for (int j=0;j<numfilasentero;j++){
      listaagentes[contador][0]=posinicialx+(separacion*i);
      listaagentes[contador][1]=posinicialy+(separacion*j);
      contador++;
    }
  }

  //completamos el numero de agentes añadiendo los que faltan
  //diferencia, entre numfilasentero*numcolumnasentero y num pedido
  //(agentes que por el redondeo no entran en la cuadricula)
  for (;contador<num;contador++){
    int posAleatoria = (int)Math.rint((Math.random()*((
       numfilasentero*numcolumnasentero)-1)));
    listaagentes[contador][0]=listaagentes[posAleatoria][0];
    listaagentes[contador][1]=listaagentes[posAleatoria][1];
  }
  return listaagentes;
}
```

- Líneas 1-2: recibe las coordenadas y el número de agentes, e inicializa el array bidimensional que posteriormente devolverá.
- Líneas 6-12: calcula el longitud horizontal y vertical del área que se le haya pasado, y calculas cuantas filas y columnas debería de haber para repartir todos los agentes.
- Líneas 19-26: cómo no podemos tener columnas o filas parciales, redondeamos al valor entero menor más próximo, y fijamos con ese nuevo número de filas y columnas equidistantes, cual sería su posición (el margen que dejaremos próximo a los bordes del área debido al redondeo).
- Líneas 28-35: vamos recorriendo las posiciones de filas y columnas (latitud y longitud) y se las asignamos a los entradas del array.
- Línea 42-45: debido al redondo, puedo ocurrir que el número de agentes asignados siguiendo el método ordena de filas y columnas, sea menor al número requerido. En estos casos, se completará repitiendo alguna posición de manera aleatorio (es decir, es posible que las coordenadas de algún punto tenga dos agentes).

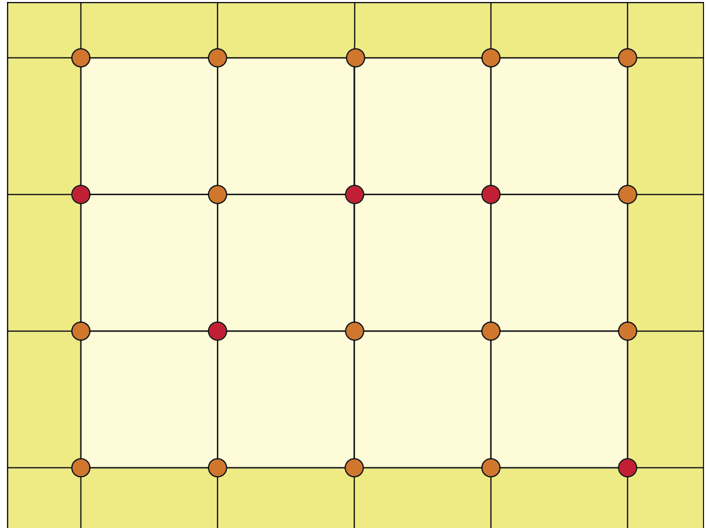


Figura 3.5: Distribución de agentes sobre un área.

En la figura 3.5 tenemos un ejemplo visual de cómo quedarían repartidos 25 agentes un área rectangular determinada. Filas y columnas se sitúan equidistantes en la zona amarilla clara, mientras alrededor hay una zona más oscura que se corresponde con el margen (la suma de la distancia del margen izquierdo y derecho, y superior e inferior, es inferior a la distancia entre líneas). En este caso tenemos 4 filas y 5 columnas, por lo que contamos con 20 posiciones, lo que hace necesario añadir 5 agentes repartidos al azar (puntos naranjas posición simple, puntos rojos posición doble).

Llegados a este punto tenemos para zona de cada ruta, las posiciones donde se sitúan los agentes, es decir, sus coordenadas. Estas coordenadas son devueltas al

método que llamó a la función en formato de un array bidimensional de objetivos de tipos *Double*. Podríamos directamente emparejar agentes de una zona (origen), con los de la siguiente (destino), pero eso haría que todos los desplazamiento entre rutas llevaran el mismo patrón. Para solucionar este tema, y dotar al resultado final de mayor aleatoriedad dentro de los márgenes, se ha implementado la siguiente función:

```
int [] secuenciaAleatorio(int num_min, int num_max){
  int [] cadena;
  if (num_max>num_min){
    int diferencia = num_max - num_min;
    ArrayList<Integer> lista=new ArrayList<Integer>();

    for (int i=0;i<diferencia;i++)
      lista.add(num_min+i);

    for (int i=0;i<diferencia;i++){
      int valor_aleatorio = (int)Math.rint((Math.random()*(
          diferencia-1)));
      int valor_aleatorio2 = (int)Math.rint((Math.random()*(
          diferencia-1)));

      int temp = lista.get(valor_aleatorio);
      lista.set(valor_aleatorio, lista.get(valor_aleatorio2));
      lista.set(valor_aleatorio2, temp);
    }

    cadena=new int[lista.size()];

    for (int i=0;i<lista.size();i++)
      cadena[i]=lista.get(i);
  }else{
    cadena = new int[1];
    cadena[0]=0;
  }
  return cadena;
}
```

Si nos fijamos, esta función nos devuelve un Array de enteros, relleno con los valores comprendidos entre los números que le pasemos como parámetros pero ordenados de manera aleatoria. Es decir, llamamos a la función con los parámetros `num_min=0, num_max=9`, nos devuelve (cada vez que llamemos de nuevo a la función, la secuencia variará):

```
[7 2 4 6 9 1 3 5 0 8]
```

Por tanto, nos serviremos de estas secuencias, para asignar las relaciones entre las posiciones de los agentes en zonas continuas (trayectos origen-¿destino).

Dado que debemos se muy organizados para evitar cualquier tipo de cruce de datos indeseado, se ha creado un nuevo tipo de objeto llamado ***DatoPlan***, el cual contiene todos los datos que debemos recoger para que luego sean grabados en la base de datos.

```
private String desc_ruta;

private String desc_zona;

private int persona;

private Double long_origen;

private Double lat_origen;

private String type;

private int hora_inicio;
```

Además, cremos un ArrayList de este tipo de objeto, con lo que podemos ir añadiendo datos, y después recórrelo por completo para hacer el volcado a la BD.

```
ArrayList<DatoPlan> listado_act=new ArrayList<DatoPlan>();
```

En este punto tenemos organizados en la base de datos todas las coordenadas donde se sitúan los agentes, y cada uno de ellos está identificado con el plan, mapa, ruta, zona, y persona a la que pertenece. Para recopilar esta información, según nos convenga, y terminar formando el archivo de planes, debemos llamar al método *generarPlan*.

"generarPlan", comprueba que los datos que necesitamos se encuentras en la base de datos, y crea un objeto de tipo CreatePlans, para poder llamar a sus métodos y generar el archivo final.

CreatePlans

Ya tenemos el listado de los trayectos que va a realizar nuestra población guardados en la base de datos, pero ahora necesitamos crear un archivo que MATSim pueda interpretar.

Al comienzo de este trabajo hemos comentado que MATSim no proporciona ninguna manera de generar planes, y eso no es del todo cierto. Entre sus librerías, cuenta con métodos los cuales permiten crear una estructura de datos, pasándole todos los datos de cada trayecto, y posteriormente volcar esa estructura en un archivo, lo cual nos asegura que el archivo final sea totalmente compatible con la sintaxis utilizada por MATSim. Como se puede ver, el problema de esta funcionalidad es que requiere que se le indique absolutamente todos los datos para generar el plan, pero llegados a este punto, nosotros ya contamos con esa información en nuestra base de datos.

El método **CreardesdeBD**, lo que hace es realizar una llamada a la base de datos, preguntándole por los trayectos definidos para un determinado nombre de plan, mapa, y usuario. Se realizan varias consultas, la primera con todas las

diferentes rutas con las que cuenta un plan, la segunda, por cada ruta se pregunta por cuantas personas lo formas, y por último, recuperamos la información de cada persona.

Dado que obligatoriamente debemos de consultar y colocar todos los datos para crear la estructura que utiliza MATSim para generar el archivo de planes, aprovechémoslo para crear nuestro propio archivo de texto, que contenga el identificador que va a tener cada persona en MATSim, junto con los datos que hemos ido introduciendo. Este archivo será de texto plano con cadenas de caracteres, separando cada campo con “,”, pensado para que sea fácil de exportar a otro programas como por ejemplo *Microsoft Excel*, y que cualquier persona pueda, en caso de así requerirlo, buscar un determinado desplazamiento de manera rápida y sencilla.

A continuación se muestra el código necesario para generar un *plan* en cada iteración:

```
1
2   FileWriter txt=null;
3   PrintWriter pw=null;
4
5   Config config = ConfigUtils.createConfig();
6   Scenario sc = ScenarioUtils.createScenario(config);
7
8   Network network = sc.getNetwork();
9   Population poblacion = sc.getPopulation();
10
11  PopulationFactory populationFactory = poblacion.getFactory();
12      .
13      .
14      .
15  pw.println("ID_PERSON,MAPA,NOMBRE_PLAN,DESCRIPCION_RUTA,
        RECORRIDO\n");
16
17  //tratamos de conectar con la base de datos
18  try {
19       Class.forName("com.mysql.jdbc.Driver");
20       con = DriverManager.getConnection("jdbc:mysql://localhost/
            pfclcp","root","");
21
22       stmt = con.createStatement();
23  }catch (Exception e){
24      System.out.println("Error al establecer conexion con BD: "+
           e.getMessage());
25  }
26
27  try{
28    ResultSet rs = stmt.executeQuery ("SELECT DISTINCT desc_ruta
         ....");
29
30    int cont_rutas=0;
31
32    while (rs.next()){
33
```

```
34      stmt2 = con.createStatement();
35      ResultSet rs2 = stmt2.executeQuery ("SELECT DISTINCT
            persona ....");
36      while (rs2.next()){
37        stmt3 = con.createStatement();
38        ResultSet rs3 = stmt3.executeQuery ("SELECT * ....");
39
40        String cadena="";
41        Person person = populationFactory.createPerson(sc.
             createId(Integer.toString(cont_agentes)));
42        Plan plan = populationFactory.createPlan();
43        cadena=cadena+Integer.toString(cont_agentes)+","+mapa+","
             +nombre+","+rs.getString("desc_ruta")+",";
44
45        Double long_origen=0.0, lat_origen=0.0;
46        int hora_inicio=0;
47        String d_zona="", tipo="";
48
49        boolean seguir=true;
50        boolean primero=true;
51
52        while (seguir){
53          if (rs3.next()){
54            if (primero){
55              long_origen=rs3.getDouble("long_origen");
56              lat_origen=rs3.getDouble("lat_origen");
57              hora_inicio=rs3.getInt("hora_inicio");
58              d_zona=rs3.getString("desc_zona");
59              tipo=rs3.getString("type");
60
61              primero=false;
62            }else{
63              Coord origen= sc.createCoord(long_origen,
                   lat_origen);
64
65              Activity act_origen = populationFactory.
                   createActivityFromCoord(tipo, origen);
66              act_origen.setEndTime(hora_inicio);
67              Leg leg = populationFactory.createLeg("car");
68              leg.setDepartureTime(hora_inicio);
69
70              plan.addActivity(act_origen);
71              plan.addLeg(leg);
72
73              cadena=cadena+d_zona+"->";
74
75              long_origen=rs3.getDouble("long_origen");
76              lat_origen=rs3.getDouble("lat_origen");
77              hora_inicio=rs3.getInt("hora_inicio");
78              d_zona=rs3.getString("desc_zona");
79              tipo=rs3.getString("type");
80            }
81          }else{
82            Coord origen= sc.createCoord(long_origen, lat_origen)
                ;
83            Activity act_origen = populationFactory.
```

```
                createActivityFromCoord(tipo, origen);
84            plan.addActivity(act_origen);
85            cadena=cadena+d_zona;
86            seguir=false;
87          }
88        }
89        person.addPlan(plan);
90        poblacion.addPerson(person);
91        cont_agentes++;
92        pw.println(cadena);
93      }
94      cont_rutas++;
95    }
96    poblacion.setName(nombre+"_"+mapa+"num_ruta:"+cont_rutas);
97    }catch(Exception e){
98      System.out.println("Error al leer dato: "+e.getMessage());
99    }
100   MatsimWriter popWriter = new org.matsim.api.core.v01.
        population.PopulationWriter(poblacion, network);
101   popWriter.write(ruta+"/plans.xml");
102
103   try {
104     txt.close();
105   } catch (IOException e) {
106     System.out.println("Error al finalizar el archivo auxiliar
          de planes."+e.toString());
107   }
```

- Líneas 1-11, declaramos las variables que vamos a emplear.
- Líneas 15, escribimos la primera línea de nuestro archivo personalizado de texto que indica el orden de aparición de los campos.
- Línea 28, tras hacer las habituales conexiones a la base de datos, realizamos un SELECT DISTINCT del campo `desc_ruta`, para obtener todos los valores diferentes de este campo (siempre para este mapa, nombre y usuario).
- Línea 35, recorremos todos las descripciones de ruta encontradas, y para cada una de ellas obtenemos las personas que transitan por esas rutas.
- Líneas 41-42, creamos los objetos del tipo *Person* y *Plan*, y los inicializamos con los valores que hemos recuperado.
- Línea 43, copiamos esos mismo valores a nuestra cadena te texto, que posteriormente grabaremos en el archivo auxiliar.
- Líneas 53-88, recorremos la última consulta que hicimos, con todos los desplazamientos que realiza cada persona, dentro de una ruta, de un plan determinado. Añadimos los datos a los objetos *Coord, Activity y Leg*, además de añadirlos a nuestra cadena de texto. Controlamos la posición del elemento que estamos leyendo, para al llegar el último (zona de destino final), no añadir los datos que no son necesario (franja horaria de partida,. . .).

- Línea 92, añadimos la cadena de texto a nuestro archivo auxiliar.
- Línea 96, añadimos una descripción al plan, que aparecerá en el archivo *plans.xml*.
- Línea 101, llamamos a la función de MATSim que preparan y genera a partir de esta estructura el archivo final.

Es **IMPRESCINDIBLE** seguir el orden que aparece en la función para crear y guardas los objetos *Person, Plan, Coord, Activity, y Leg*. Si no se hiciera así, las etiquetas en el archivo xml final quedarían desordenadas, y sería ilegible en el proceso de simulación (o lo que sería peor, no actuaría como nosotros esperáramos que lo hiciera).

Planes pendientes y finalizados

En esta ocasión, no hay nada nuevo que destacar. En ambos casos se sigue el mismo procedimiento que para el capítulo anterior, realizando una consulta a la base de datos esperando que nos devuelva el listado de planes pendiente y su estado, y finalizados. En este caso, además se habilitará el enlace para la descargar el archivo `"plans.xml"` y del archivo con la información auxiliar `"plans_info.txt"`.

Capítulo 4

Procesamiento de resultados y visualización

Este capítulo recoge las últimas funcionalidades que hemos implementado para nuestra solución. Veremos como poder cargos los resultados, analizarlos para volcarlos en una base de datos que nos permita trabajar mejor con ellos, y crear nuestra propia visualización. Para este último paso, se ha optado por utilizar el elemento CANVAS de HTML5 para facilitar su visionado en diferentes sistemas sin necesidad de software extra, simplemente utilizando el navegador.

4.1. Introducción

En los dos capítulos anteriores, teníamos mucha más definido el objetivo final que debíamos de cumplir. El caso que abordamos ahora tiene un final mucho más abierto, ya que la interpretación final de los datos de la simulación puede ser algo mucho más subjetivo.

Hay ciertas tareas que sabemos que debemos realizar de cualquiera forma. Debemos ser capaces de subir a nuestro servidor el archivo que contiene el resultado de la simulación, algo que ya hemos afrontado y resuelto en anteriores partes del sistema. Hasta ahora, una vez que subíamos al servidor el archivo, utilizábamos alguna librería que ya estaba desarrollada anteriormente para trabajar con él. En esta ocasión debemos de buscar la forma de no solo subir el archivo, sino además leer e interpretar su contenido.

Al igual que el resto de archivos que necesita o genera MATSim, el archivo es un XML. Por ello, hablaremos de manera general de las técnicas con las que cuenta Java para analizar los archivos xml, y explicaremos porqué nos hemos decantado por una de ellas y como la utilizamos.

Una vez tengamos los datos en nuestra base de datos, debemos interpretarlos

y conseguir una visualización mucha más realista que la que ofrece MATSim, y me manera sencilla para el usuario.

Ya que estamos trabajando con una aplicación Web, tras estudiar varias opciones, se ha optado por hacer uso de CANVAS, uno de los nuevos elementos que han incorporado a HTML5. CANVAS nos permite realizar "dibujos" de manera dinámica, pudiendo interactuar con él a través de funciones JavaScript, lo que nos ayudará mucho al resultado final que buscamos.

En la figura 4.3 tenemos el diagrama se la interacción general del usuario con esta parte de la aplicación.

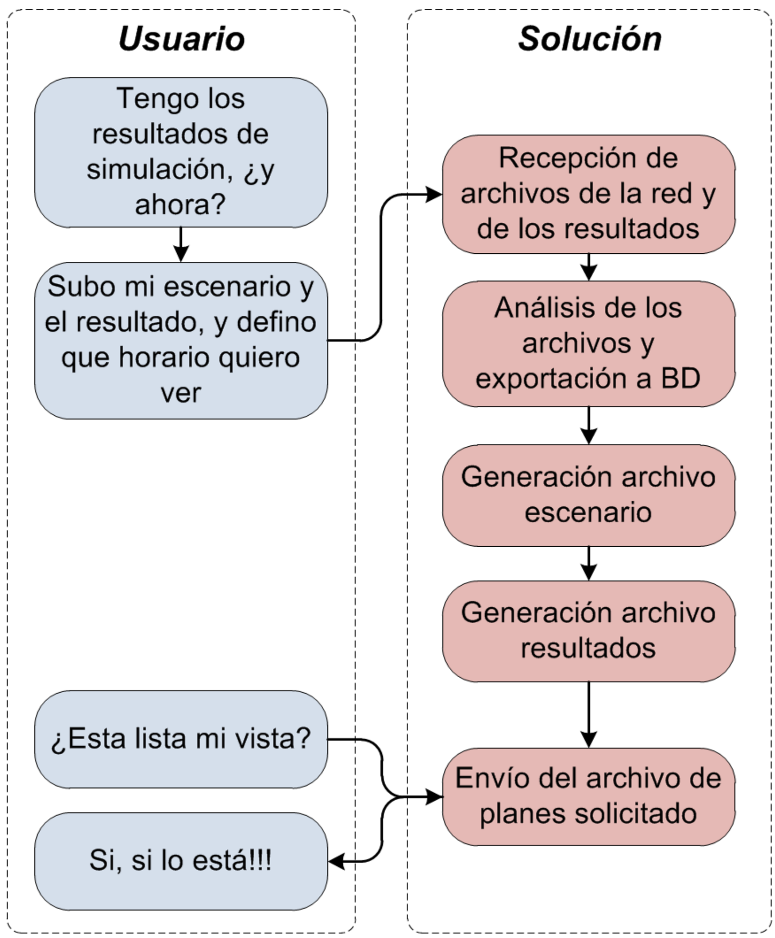


Figura 4.1: Pasos que realiza el usuario para crear una visualización.

El principal reto con que no enfrentamos es la gran cantidad de información con la que deberemos trabajar, y que podemos hacer para tratarla en el menor tiempo posible y con el mejor resultado. En la siguiente sección vamos a ver el esquema de esta parte de la solución y las funciones más importante implementadas para conseguir nuestros objetivos. Explicaremos cómo leer archivos XML de gran tamaño, que es CANVAS, cómo se trabaja con él y como lo vamos a utilizar nosotros.

4.2. Funciones y comunicación entre ellas

En los capítulos anteriores hemos visto ya todas las partes de comunicación entre el cliente y el servidor que vamos a utilizar. Nos centraremos en tres grandes aspectos fundamentales que son necesarios para entender el funcionamiento del sistema de visualización.

- Sistema para leer archivos XML, e interpretar su contenido. Veremos uno de los casos, para reflejar el funcionamiento de todos los demás.
- Elemento CANVAS de HTML5. Al habernos decantado por esta opción, explicaremos porque la hemos elegidos, las nociones básica necesarias para movernos y dibujar en su lienzo.
- Como interpretamos los datos, para generar las vistas.

4.2.1. Leer archivos xml en Java

Ha diferencia de los anteriores capítulos, esta vez necesitamos hacer algunas consideraciones previas antes de entrar con el desarrollo en sí de la solución.

En otras ocasiones, se han obviado las bases de los lenguajes de programación para poder centrarnos en las partes diferenciadoras que hemos implementados. Esta vez, creo que es necesario comentar este tema, ya que existen varias formas de leer estructuras de datos con formato XML, y tras haber probado varias, se ha elegido una de ellas.

De manera general existen dos tipos de analizadores de documentos XML, los denominados analizadores DOM y los analizadores SAX.

- **Analizadores DOM**: Se basan en la representación en memoria del modelo DOM. Construyen el árbol que se desprende del documento XML en memoria, para posteriormente por recorrerlo según nuestras necesidades. Tiene la ventaja de que al disponer de todo el árbol en memoria, es sencillo acceder a los datos de manera ordena. Además nos permite acceder a ellos de manera no ordena, pudiendo hacer consultas según nuestra conveniencia, y no según el orden en que aparecen los datos en el documento leído. También tienen inconvenientes, y es que al cargarse toda la información en memoria, dependiendo del documento, el espacio que consumen puede ser muy elevado.
- **Analizadores SAX**: Utilizan un gestos de eventos, que reacciona antes las diferentes características del documento, avisando a al aplicación cada vez que encuentra un nuevo elemento para que esta decida si utilizarlo o no. Realizan una lectura secuencial del documento XML. Presentan como ventaja que consumen mucha menos memoria que los anteriores, pero a cambio su utilización en nuestros programas en mucho menos intuitiva, y hay que indicar exactamente tipo de etiquetas queremos tratar. Al ser una lectura secuencia, una

vez se ha leído una línea del documento no se vuelve atrás, por lo que si no capturamos un dato que necesitáramos, este se habrá perdido.

Existe mucha documentación [9] sobre el uso de un modelo de analizador u otro. A priori, parece que el analizador DOM es mucho más sencillos de utilizar, y nos ofrece la ventaja de poder realizar un acceso aleatorio al contenido del archivo leído.

En nuestro caso, debemos de tener en cuenta que los archivos XML que vamos a leer son de tamaño muy elevado. Además debemos de procesar el documento entero, ya que contienen un número muy limitado de etiquetas diferentes, pero que se repiten desde el principio hasta el final de archivo y contienen información que debemos recuperar. Se trataron de realizar pruebas utilizando el analizador DOM para poder compararlo frente a SAX, pero fue imposible debido a los fallos que ocasionaba en el sistema la carga en memoria de tal número de registros (como ejemplo, un mapa de una ciudad del tamaño de Alcalá de Henares, tiene sobre 20.000 registros entre nodos y enlaces, y el archivo de resultado de planes puede llegar a ser mucho mayor dependiendo del tamaño de la población que simulemos y las iteraciones que realicemos).

Por lo tanto se ha elegido como método para realizar la lectura, el analizador SAX. A continuación veremos un ejemplo de cómo lo utilizamos en nuestro sistema.

Ejemplo de uso del Analizador SAX

Deberemos de implementar los métodos para leer tanto archivos de escenarios en formato MATSim, como en formato OpenStreetMap, ya que los segundos incorporan más datos a partir de los cuales podemos hacer representaciones propias de escenarios más realistas. Además debemos, poder leer los archivos que contienen información sobre los resultados de la simulación.

MATSim, como ya hemos comentado, realiza un número de iteraciones definidas por el usuario. Como resultado de estas iteraciones, como resultado nos devuelve en otros el archivos plans.xml que le habíamos pasado a la entrada, pero rellenos con los nuevos datos que ha obtenido. Por tanto, para interpretar los resultados de la simulación, el usuario deberá de subir al servidor, el archivos de planes correspondiente la iteración que desea visualizar, y el archivos del escenario, para poder superponer ambos datos.

Dentro de nuestra función, deberemos crear un objeto de para el analizador de la siguiente forma:

```
1 try{
2   SAXParserFactory spf=SAXParserFactory.newInstance();
3   SAXParser sp = spf.newSAXParser();
4   sp.parse(url, new LibraryXMLReader() );
5 }catch(ParserConfigurationException e){
6   System.err.println("Error al Parsear");
```

```
7   }catch(SAXException e2){
8     System.err.println("Error al recorrer con SAX: " + e2.
        getStackTrace());
9   } catch (IOException e3) {
10    System.err.println("Error entrada/salida: " + e3.getMessage()
        );
11  }
```

Cabe destacar que en la línea 4, indicaremos la ruta de archivo que vamos a proceder a leer.

Además necesitamos variar el manejador de eventos, para indicarle como actuar ante determinadas señales.

```
1   private class LibraryXMLReader extends DefaultHandler {
2     public void startDocument() throws SAXException{
3       System.out.println("Estoy al principio del documento");
4       }
5
6     public void endDocument()throws  SAXException{
7       System.out.println("Estoy al final del documento");
8       String cadena=y_min+","+x_min+","+y_max+","+x_max;
9       AccesoBD modificar = new AccesoBD();
10      modificar.modificacionVista(nombre, usuario, "invitado", "
          Leido escenario", cadena);
11    }
12
13    public void startElement(String uri, String localName, String
        name,
14      Attributes attributes) throws SAXException {
15      if (name=="node")
16      {
17        String id="";
18        String x="";
19        String y="";
20
21        for(int i=0;i<attributes.getLength();i++){
22          if (attributes.getQName(i)=="id"){
23            id=attributes.getValue(i);
24          } else if (attributes.getQName(i)=="x"){
25            x=attributes.getValue(i);
26            double temp = Double.parseDouble(attributes.getValue(
                i));
27
28            if (x_min>temp)
29              x_min=temp;
30            if (x_max<temp)
31              x_max=temp;
32          } else if (attributes.getQName(i)=="y"){
33            y=attributes.getValue(i);
34            double temp = Double.parseDouble(attributes.getValue(
                i));
35
36            if (y_min>temp)
```

```
37              y_min=temp;
38            if (y_max<temp)
39              y_max=temp;
40          }
41        }
42        .
43        .
44        .
45      }else if (name=="link"){
46        String id="";
47        String from="";
48        String to="";
49        String length="";
50        String freespeed="";
51
52        for(int i=0;i<attributes.getLength();i++){
53          if (attributes.getQName(i)=="id"){
54            id=attributes.getValue(i);
55          } else if (attributes.getQName(i)=="from"){
56            from=attributes.getValue(i);
57          } else if (attributes.getQName(i)=="to"){
58            to=attributes.getValue(i);
59          } else if (attributes.getQName(i)=="length"){
60            length=attributes.getValue(i);
61          } else if (attributes.getQName(i)=="freespeed"){
62            freespeed=attributes.getValue(i);
63          }
64        }
65        .
66        .
67        .
68      }
69    }
70  }
```

- Línea 2, *startDocument* es llamado una única vez al comienzo del documento. Simplemente avisamos que se ha comenzado con la lectura.
- Línea 6-10, *endDocument*, de igual manera se llama una única vez al finalizar la lectura. Momento que aprovechamos para grabar en la base de datos, que se ha finalizado la lectura, y añadir también la información sobre los valores máximos y mínimos leído, ya que luego no serán de utilidad.
- Línea 13, *startElement*, es llamado cada vez que lee una etiqueta XML, sea del tipo que sea.
- Líneas 15 y 45, comprobamos si la etiqueta leída es de alguno de los tipos que buscamos.
- Líneas 21-39, si es una de las etiquetas que queremos leer, recorremos sus atributos, y comprobamos uno a uno su nombre. En caso de coincidir con el buscado, recuperamos el dato utilizando `.getValue` para luego tratarlo como queramos.

Ya hemos visto, que la utilización de este tipo de analizar, requiere un conocimiento preciso de la estructura del documento XML, ya que de no ser así sería imposible conocer ante qué etiquetas debemos reaccionar.

Nuestras funciones, leerán los archivos he irán almacenando los datos en la base de datos, para luego poder acceder a ellos cuando los necesitemos en el siguiente paso.

4.2.2. Representación de la información

Al llegar el momento de buscar cómo visualizar los datos, se han realizado multitud de pruebas para probar que método se podía emplear obteniendo mejores resultados.

Básicamente estas pruebas podrían reducirse a tres:

- **Renderización de imágenes:** La primera idea surgió tratando de imitar el funcionamiento de los proveedores de mapas existentes. Partiendo de los datos geográficos que componen nuestro escenario, modificarlos según los resultados de las simulaciones para posteriormente realizar nuestra propia interpretación *artística* de ellos utilizando conjuntos de herramientas de libre distribución que generan imágenes a partir de datos geográficos. Entre estas herramientas tenemos entre otras Mapnik [1] y Osmarender [2].

(a) Render con los datos originales

(b) Render modificando el tipo de una de las vías

Figura 4.2: Modificación de datos geográficos para variar su visualización.

Utilizando la segundo obteníamos resultados unos buenos resultado iniciales como podemos ver en la figura 4.2. El problema venía en que el tiempo ne-

[1] `http://mapnik.org/`
[2] `http://wiki.openstreetmap.org/wiki/Osmarender`

cesario para hacer esta renderización era muy elevado, y además únicamente obteníamos imágenes estáticas.

Aunque se vio que este método no llevaba a conseguir nuestro objetivo. Sirvió para entender como funcionaban este tipo de sistemas.

- **Utilizando las funcionalidades de OpenLayers:** Esta librería cuenta con la opción de interpretar archivos de tipo GML[3], los cuales están compuestos por una estructura XML con características geográficas concretas. Al igual que en el caso anterior, los resultado iniciales fueron bueno, pero seguíamos sin solucionar el problema de mostrar esa información en movimiento.
- **Elemento CANVAS de HTML5:** Se utiliza para dibujar gráficos, "al vuelo", utilizando scripts. Después de realizar varias pruebas, se vio que era la mejor opción ya que podíamos superponer varias capas, pudiendo utilizar una de ellas para representar el escenario y otra para la población y sus trayecto.

Elemento CANVAS de HTML5

En si, no se es más que un lienzo en que podemos dibujar, pero debido a que estos dibujos los realizamos mediantes "scripts" (en nuestro caso utilizando JavaScript), cuenta con un grado de versatilidad enorme.

Para poder uso de él, necesitamos incluir en el código del cliente dos porciones de código. La primera el propio elemento con la etiqueta CANVAS, y la segunda, el script que interactuará con se elemento para realizar el dibujo.

En w3school.com[4] tenemos el siguiente ejemplo de uso. Incluimos la definición básica del elemento:

```
<canvas id="myCanvas" width="200" height="100"></canvas>
```

Y el script que dibujará la forma:

```
<<script>

<var c=document.getElementById("myCanvas");

<var ctx=c.getContext("2d");

<ctx.fillStyle="#FF0000";

<ctx.fillRect(0,0,150,75);

<</script>
```

Como vemos el código es muy simple, simplemente recupera el elemento tipo

[3]Geography Markup Language (Lenguaje de Marcado Geográfico)
[4]`http://www.w3schools.com/html/html5_canvas.asp`

CANVAS que hemos creado, inicializa un dibujo en dos dimensiones, y añade características, indicando en este caso la posición de las cuatro esquinas de un rectángulo.

Debemos de tener presenta el sistema de coordenadas en que nos movemos dentro del área definida en el elemento CANVAS para realizar el dibujo. Tomo como origen la esquina superior izquierda, que es referida por las coordenadas (0,0), e incrementa la coordenada X al avanzar hacia la derecha, y la coordenada Y al avanzar hacía abajo.

Ya que nosotros tenemos en este punto en la base de datos las coordenadas de todos los elementos que queremos representar (definidas por la posición de los nodos, y la relación que hay entre ellos), debemos realizar un equivalente entre las coordenadas de los datos almacenados y las coordenadas del lienzo donde ser van a representar.

En el lado del cliente, nos limitaremos a darle ya todo preparado para su visualización, como veremos un poco más adelante, mientras que en el lado del servidor, procesaremos los datos y los adaptaremos para poder ser visualizados en el elemento CANVAS.

Toda las funciones que hemos implementado para crear datos para CANVAS a partir de información volcada en la base de datos comienzan con las siguientes líneas de código:

```
String [] lista=tammano.split(",");

double minlat=Double.parseDouble(lista[0]);
double minlon=Double.parseDouble(lista[1]);
double maxlat=Double.parseDouble(lista[2]);
double maxlon=Double.parseDouble(lista[3]);

double width_canvas=width;
double height_canvas=height;

double dif_lon=maxlon-minlon;
double dif_lat=maxlat-minlat;

double factor_pixel=width_canvas/dif_lon;

double tam_via=height_canvas*0.0035;


if ((factor_pixel*dif_lat)>height_canvas)
  factor_pixel=height_canvas/dif_lat;
```

- Líneas 1-6, recuperamos los límites geográficos que ocupa escenario o el archivo de planes que vamos a representar.
- Líneas 8-9, almacenamos el tamaño del elemento CANVAS.
- Líneas 11-12, hayamos el tamaño vertical y horizontal que ocupan los datos

originales a representar.

- Línea 14, 19 y 20, calculamos el factor de conversión, es decir el valor por que debemos de multiplicar un valor en una de las escalas para obtener su correspondencia en la otra. Es posible que al relación entre el alto y el ancho del lienzo, y de la porción de datos geográficos no sea la misma, por ellos nos quedamos con el factor que más limite, para no perder datos en la visualización.
- Líneas 16, almacenmos un valor obtenido a patir de multiplicar el ancho del CANVAS por un número obtenido a través de pruebas. Este valor se empleara como referencia para obtener el ancho de las vías y de los agentes que circulan sobre ellas (conseguiremos que al aumentar o reducir el tamaño de la representación, también varíe el grosor de los elementos que lo componen).

Una vez sepamos como relacionar los dos sistemas de referencia, el proceso es muy sencillo, ya que simplemente recorreremos todos los elementos que queramos representar, he iremos dibujando las formas.

Existen muchas funciones a las que podemos llamar para dibujar diferentes figuras, pero en nuestro caso, al tener que presentar formas definidas por coordenadas, nos es suficiente con utilizar la que nos permite dibujar líneas.

```
context.moveTo(x1,y1);

context.lineTo(x2,y2);
```

Debemos pensar en que la forma para dibujar sobre un elemento CANVAS, es similar a si lo hiciéramos directamente sobre un papel. En la primera línea, situamos el "lápiz" en un punto de lienzo que tiene coordenadas (x1, y1), y con la segunda línea, dibujamos una línea hasta la coordenada (x2, y2). Por tanto repitiendo este proceso cuantas veces sea necesario, podremos dibujar nuestro mapa.

Una vez dibujada la línea es necesario pintarla para poder visualizarla.

```
context.lineCap = 'round';

context.lineWidth = tam_via;

context.strokeStyle = 'black';

context.lineJoin='round';

context.stroke();
```

Definimos ciertas propiedades como por ejemplo, que los finales de nuestras líneas serán redondeados, su tamaño, color, tipo de uniones, y muy importante el método `.stroke()`, el cual dibujará los datos de la figura que le hemos pasado, interpretándolos como líneas sueltas. Para la representación de escenarios a partir del archivo de redes de MATSim, con esto será suficiente, y que solo contiene información sobre las vías que lo forman. Sin embargo, para escenarios creados a partir de

archivos de tipo OSM, definiremos conjuntos de datos comenzando por la sentencia `.beginPath()`, y finalizando con `closePath()`también utilizaremos el método `.fill()`, y posteriormente llamando a `.fill()`, nos permitirá dibujar forma rellenas (fill unirá el último punto definido y el primero para formar una figura cerrada.

Combinando estás propiedades que hemos visto, y recorriendo los datos geográficos que hemos extraído en el capítulo anterior, obtenemos representaciones como las que tenemos en la figura 2.3.

Por último, usaremos otro tipo de figura en CANVAS para representar a cada elemento de la población.

```
contexto.arc(x,y,tam_via,0,Math.PI*2,true);
```

Con el método `.arc`, podremos dibujar arcos, pero al definirlo como tenemos en la línea anterior conseguiremos dibujar una circunferencia, centrada en el punto (x,y), y con tamaño "`tam_via`".

4.2.3. Desarrollo de la aplicación

Tras ver en los apartado anteriores las nuevas funcionalidades que vamos a emplear, sumado a los visto ya en capítulos anteriores, tenemos cubierta gran parte de lo que utilizaremos para implementar esta parte de la aplicación.

En la pestaña *"VISUALIZACIÓN"*, vamos a contar con todas las herramientas que nos permitirán generar la vista a partir del escenario y los resultados de la simulación que le indiquemos. En la figura 4.3 tenemos la estructura y funcionalidad que está disponible para esta tarea desde el punto de vista del usuario.

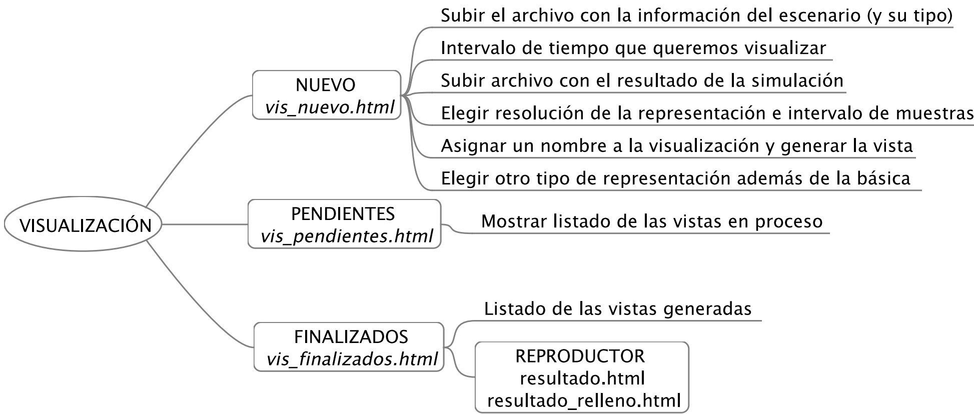


Figura 4.3: Pasos que realiza el usuario para crear una visualización.

Los servicios disponibles en las secciones *PENDIENTES* y *FINALIZADOS*, están desarrollados siguiendo el mismo proceso que los que veíamos para las pestañas de *EXPORTAR* y *PLANES*. Aunque han sido modificados y adaptados, ya que en la sección de *FINALIZADOS* no es necesario descargar ningún archivo, sino se nos

muestras los enlaces de las visualizaciones completadas.

Por tanto, vamos a pasar a explicar las partes del programa que intervienen en el proceso y las comunicaciones que se producen entre ellas.

En la figura 4.4, tenemos el diagrama de las principales funciones intervienen para lograr el objetivo de generar la visualización mostrado. El proceso que seguirá la aplicación será el siguiente:

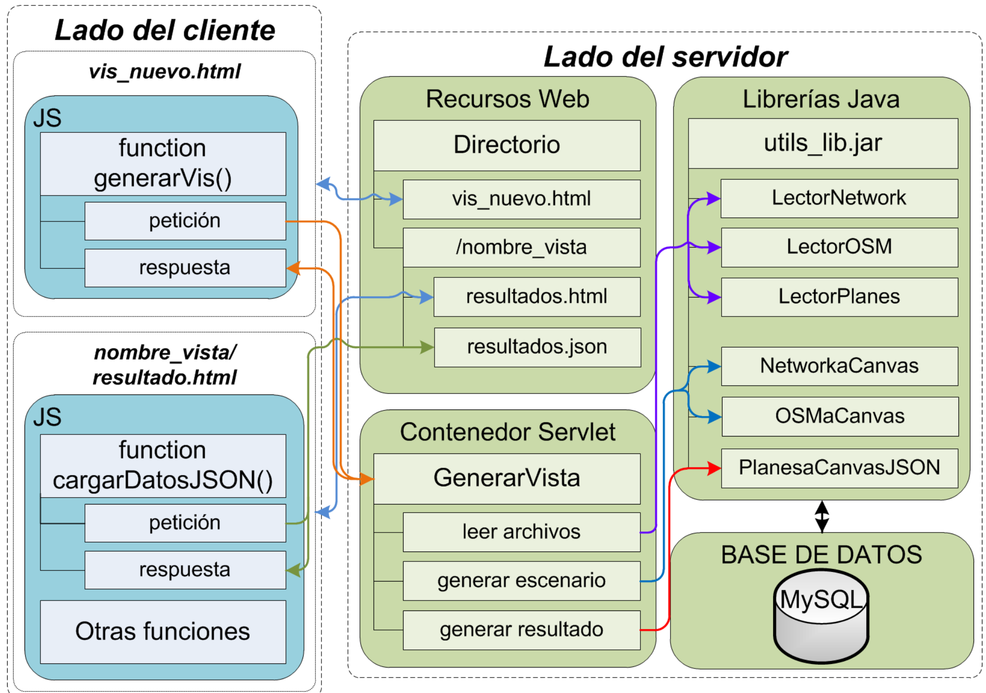


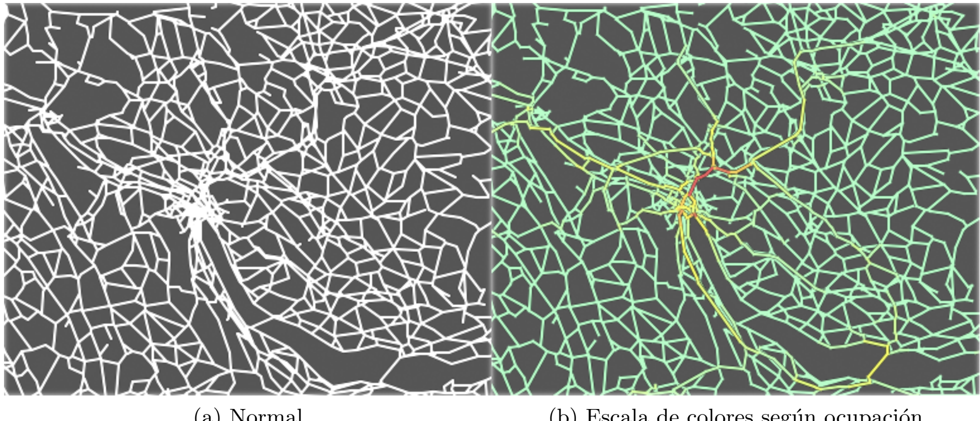
Figura 4.4: Funciones que intervienen para generar una vista y sus comunicaciones.

1. En el lado del servidor, la función JavaScript `generarVis()`, recopilará los datos que ha introducido el usuario, comprobando que haya rellenado todos los campos, y realizará la petición al servidor.

2. En el lado del servidor, el Servlet `GenerarVista`, recibirá los datos del cliente, guardará los archivos en los directorios correspondientes y seguirá el siguiente proceso:

   a) Llamará a las funciones `LectorNetwork o LectorOSM` dependiendiendo del tipo de archivo de escenario recibido, y a la función `LectorPlanes` para procesar los archivos XML y volver la información que contienen en la base de datos.

   b) En caso de haber marcado la opción de “Incluir visualización de vías más transitadas“ se llamara al método *revisarOcupacion* de la clase OcupacionNetwork, el cual recorrerá todos los planes para saber por que vías transitan los agentes, e incrementar un contador asociado a cada vía.



(a) Normal (b) Escala de colores según ocupación

Figura 4.5: Diferentes tipos de representación a partir de los mismo datos.

c) Tras completar el proceso anterior, se llamará a las funciones `NetworkaCanvas o OSMaCanvas`, según el caso, para generar el archivo HTML que servirá para visualizar los resultados(Fig. 4.5a), y que a su vez contendrá la información necesaria para generar la vista del escenario, y a la función `NetworkaCanvasSemaforo` en caso de haberlo solicitado, para generar un fondo alterntativo(Fig. 4.5b).

d) Por último, llamará a la función `PlanesaCanvasJSON`, la cual creará un fichero JSON que contendrá las coordenadas de las posiciones de todo la población de los resultados, junto con su código de tiempo.

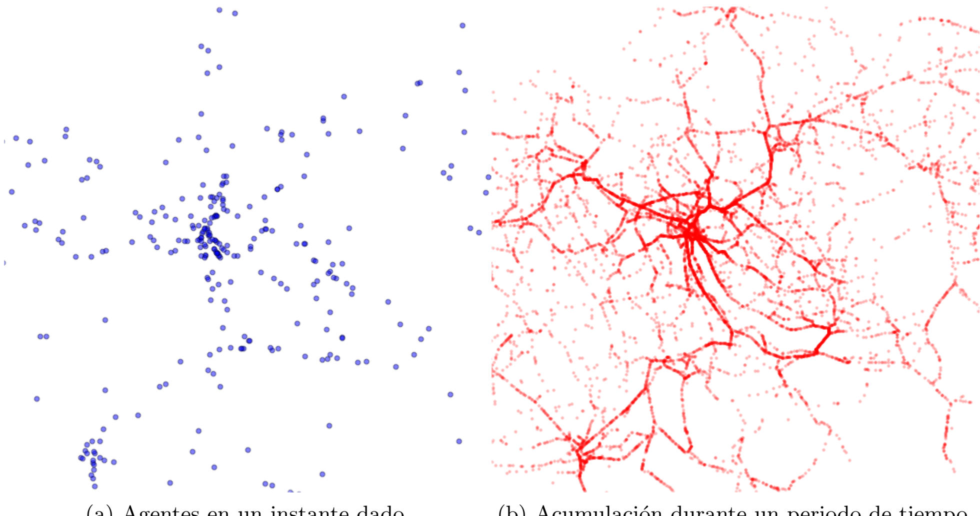
Una vez completado todo el proceso, el usuario tendrá disponible los archivos `resultado.html`, `resultado_relleno.html`, y `resultado.json`.

3. Después de finalizar el proceso anterior, la visualización aparecerá en la pestaña de *FINALIZADOS*, con un enlace donde al pulsar se abrirá una ventana nueva que mostrará `resultado.html` o `resultado_relleno.html`

4. Al cargarse la página del reproductor, se llamará la función JavaScript `cargarDatosJSON` la cual leerá el contenido del archivo `resultado.json` (no se incluyen directamente todos los datos en un archivo, debido al gran tamaño que puede llegar a tener el JSON).

5. Para finalizar, el usuario cuenta con varios botones para poder iniciar o parar la visualización(Fig. 4.6a) (interactuarán con la función JavaScript que recorre los datos extraídos del JSON y lo pinta en pantalla), accionar otra forma de ver los resultados de manera acumulativa(Fig. 4.6b), y para poder exportar a una imagen de tipo PNG lo que está visualizanco en cada momento.

Las funciones Java **LectorNetwork**, **LectorOSM**, y **LectorPlanes**, desarrolladas para extraer la información de los archivos que reciben, utilizan el analizador SAX explicado anteriormente, indicando en cada caso las etiquetas que vayamos a necesitar leer.



(a) Agentes en un instante dado. (b) Acumulación durante un periodo de tiempo.

Figura 4.6: Diferentes tipos de representación de los agentes.

Por otro lado las funciones **NetworkaCanvas** y **OSMaCanvas**, leen los datos almacenados en la base de datos por las funciones anteriores, dibujando cada enlace que aparece a través de la posición de sus dos extremos. A las coordenadas de estos puntos, aplica el `factor_pixel`, explicado en el apartado de CANVAS para transformar entre sistemas de coordenadas.

PlanesaCanvasJSON, recibe además de los datos del resultado de la simulación, el intervalo de tiempo del que queremos visualizar datos (de entre todos los trayectos posibles, solo selecciona los que tengan su hora de salida, comprendida en ese intervalo). Además permite indicar cada cuanto tomamos muestras dentro de ese intervalo (debemos de tener cuidado con este valor, ya que más muestras tomemos, mayor será el tiempo de procesamiento). Con todo ello, genera a su salida un archivo JSON con la siguiente estructura:

```
1  {"coord":[],"time":"2:46:40"}
2
3  {"coord":[{"y":249,"x":437},{"y":275,"x":439},{"y":345,"x"
       :229},{"y":328,"x":238},{"y":310,"x":241}],"time":"4:51:40"}
```

Donde podemos ver en la línea 2 como se almacenan las parejas de coordenadas de los puntos la miembros de la población que están en movimiento en ese instante de tiempo. En la línea 1, vemos una muestra tomada dentro del intervalo de tiempo solicitado donde no hay ningún movimiento (no hay nigún trayecto en curso).

Debemos detenernos un instante en analizar los datos con los que contamos y la forma en que tomamos muestras. Los archivos de resultados nos dan un listado de los nodos por los que pasamos durante un trayecto, la hora de inicio, la longitud del trayecto y la hora de finalización. Es posible, que dado que vamos a tomar muestras de manera periódica, haya algunos agentes que en ese instante de la muestra no

se encuentren justo encima de un nodo, pero en los datos geográficos con los que contamos solo tenemos ese dato. Si no los remediamos de alguna forma, en nuestra visualización no habrá ningún agente situado a lo largo de una vía, y solo se moverán de nodo a nodo.

Para solventar este asunto, implementamos la siguiente función, la cual con las coordenadas origen y destino, y la distancia recorrida (la hemos calculado con el tiempo transcurrido y la velocidad de la vía), nos devuelve el punto exacto donde se encuentra.

```
public static double [] coordconDistancia(double x1, double y1,
    double x2, double y2, double distancia){
  double [] resultado={0,0};

  double angulo=Math.atan((Math.abs(y2-y1))/(Math.abs(x2-x1)));
  double cat_opuesto=distancia*(Math.sin(angulo));
  double cat_contiguo=distancia*(Math.cos(angulo));

  if (x2>x1)
    resultado[0]=(x1+cat_contiguo);
  else
    resultado[0]=(x1-cat_contiguo);

  if(y2>y1)
    resultado[1]=(y1+cat_opuesto);
  else
    resultado[1]=(y1-cat_opuesto);

  return resultado;
}
```

Como vemos, es una simple implementación de una función de geometría.

La última tarea desde el lado del servidor sería la de ubicar los archivos `resultado.html`, `resultado_relleno.html`, y `resultado.json` en el directorio web para que fuera accesible por el usuario (el archivo JSON es común para ambas representaciones).

Por último vamos a comentar la funcionalidad que se implementan dentro del archivo reproductor para que desde el lado cliente, el usuario pueda visualizar el escenario y los movimientos de la población.

```
window.onload = function() {
  fondo_canvas('fondo');
  planes_canvas('animacion');
  cargarDatosJSON();
  step=0;
  stepTodo=0;
}

function playstop(){
  if (stepTodo>0){
```

```
      if (todo==true){
        clearInterval(estadoAccionTodo);
        todo=false;
      }
      can.width = can.width; // limpia el canvas
      stepTodo=0;
    }
    if (PlayoStop==false){
      estadoAccion=setInterval('dibujar()',200);
      PlayoStop=true;
    }else{
      clearInterval(estadoAccion);
      PlayoStop=false;
    }
}

function planes_canvas(nombre_canvas){
    can = document.getElementById(nombre_canvas);
    contexto = can.getContext("2d");
    posX=15;
    posY=100;
    direccion = 0;
    direccionV = 0;
}

function cargarDatosJSON(){
    var url = "urldeldirectorioweb/nombrevista/resultado.json";
    $.getJSON(url, function(data){
datos_json=data;
    });
}

function dibujar() {
    if (datos_json.length == 0) {
alert('Sin datos');
    }
    else{
can.width = can.width; // limpia el canvas

var i=0;
for (i=0;i<datos_json[step].coord.length;i++){
contexto.strokeStyle = "#000000";//color linea exterior
contexto.fillStyle = "blue";//color relleno
contexto.beginPath();
contexto.arc(datos_json[step].coord[i].x,datos_json[step].coord
    [i].y,2,0,Math.PI*2,true);//formamos
    circulo
contexto.globalAlpha=0.5;//grado de transparencia
contexto.closePath();
contexto.stroke();//dibujamos linea
contexto.fill();//dibujamos relleno
}

document.getElementById("tiempo").value=datos_json[step].time;

step=step+1;
```

```
65 if (step==datos_json.length){
66   step=0;
67   playstop('stop');
68 }
69   }
70 }
71
72 function dibujartodo() {
73   if (datos_json.length == 0) {
74     alert('Sin datos');
75   }
76   else{
77     if (fin_todo==true){
78       can.width = can.width; // limpia el canvas
79       fin_todo=false;
80     }
81     var i=0;
82     for (i=0;i<datos_json[stepTodo].coord.length;i++){
83     contexto.strokeStyle = '#000000';//color linea exterior
84     contexto.fillStyle = 'red';//color relleno
85     contexto.beginPath();
86     contexto.arc(datos_json[stepTodo].coord[i].x,datos_json[
         stepTodo].coord[i].y,1.75,0,Math.PI*2,true);
87     contexto.globalAlpha=0.3;//grado de transparencia
88     contexto.closePath();
89     contexto.fill();//dibujamos relleno
90     }
91
92     document.getElementById('tiempo').value=datos_json[stepTodo
         ].time;
93
94     stepTodo=stepTodo+1;
95     if (stepTodo==datos_json.length){
96       stepTodo=0;
97       clearInterval(estadoAccionTodo);
98       fin_todo=true;
99     }
100   }
101 }
102
103 function segAString(segundos){
104   var resto_horas=segundos%3600;
105   var horas=segundos/3
106 }
107
108 function convertCanvasAImagen(elec_canvas) {
109   var dataUrl = elec_canvas.toDataURL("image/png");
110   dataUrl=dataUrl.replace("image/png",'image/octet-stream');
111   document.location.href = dataUrl;
112 }
113
114 function fondo_canvas(nombre_canvas){
115
116   var canvas = document.getElementById(nombre_canvas);
117   var context = canvas.getContext('2d');
118
```

```
119     context.moveTo(3,242);
120     context.lineTo(8,244);
121     .
```

- Líneas 1-6, en la carga de la página, llamamos la funciones que dibujan los dos CANVAS (el del escenario y el de los resultados), cargamos la estructura de datos JSON, e iniciamos la variable step a 0 (esta variable lleva la cuenta del momento que estamos mostrando).
- Líneas 9-25, la función `playstop` nos permite iniciar o parar la visualización de planes. Utilizamos la función `setInterval` para llamar cada cierto tiempo a la función `dibujar()`, y con `clearInterval` detenemos el bucle que habíamos creado anteriormente. Ya que tenemos dos posibles tipos de representación, debemos de tener controla cuando se llama a esta función, que había dibujado en el CANVAS de datas, para actuar de una forma u otra.
- Línea 27, en esta función iniciamos los parámetros necesarios para dibujar en el CANVAS dedicado a ver la población.
- Líneas 36-37, descargamos con jQuery el JSON con los datos y almacenamos su contenido en la variable `datosjson`.
- Líneas 43-70, cada vez que es llamada la función `dibujar()` recorre todos los datos que tenga el array `datosjson` para el "step" actual, extrae las coordenadas y pinta los círculos en esas posiciones dentro de su CANVAS. También carga un cuadro de texto, el valor de la hora que tiene almacenado para esa muestra. Al finalizar todos los datos incremente el "step" en una unidad.
- Líneas 72-101. De manera similar a la función anterior, `dibujartodo` recorre el array con los datos obtenidos del JSON, pero los representa con diferentes propiedades, y sin borrar el CANVAS después de cada representación, obtiendo una representación del resultado acumulado durante el periodo que está activo.

  item Líneas 108-111, convierten el CANVAS que se le pasa como parámetro a imagen y lo prepara para poder ser descargado a través del navegador web.
- Línea 114 y sucesivas, dibujan una única vez el escenario de simulación.

Gracias a este grupo de funciones, hemos creado nuestro propio visualizador de resultados, que permite interactuar al usuario por si quisiera detener la reproducción en un punto determinado, sin qué cada vez que se quiera reproducir un resultado tenga que intervenir servidor (una vez creados los archivos).

Capítulo 5

Conclusiones finales

Durante el desarrollo de este trabajo fin de grado se han podido comprobar las carencias con las que cuenta el simulador MATSim para generar los datos necesarios a la entrada e interpretar los resultados finales.

Este tipo de simuladores tienen multitud de aplicaciones (poder simular planificaciones de ciudades antes de llevarlas a cabo para elegir la mejor opción, trabajar con escenarios simulados para probar alternativas que mejoren problemas concretos de congestión de tráfico, tener calculadas diferentes rutas para ofrecer caminos alternativos, . . .), pero en la mayoría de los casos se requiere realizar muchas simulaciones hasta alcanzar el resultado buscado. Esto implica el tener que preparar sobre un mismo escenario, planes con la información de los agentes que se desplazan sobre él de la forma más detallada posible, y posteriormente interpretar los datos obtenidos, tarea que utilizando la aplicación web que se ha desarrollado en este trabajo se ha simplificado enormemente.

En el apartado 1.4, establecimos los objetivos de este trabajo. A continuación vamos a recordarlos, indicando en cada caso la solución desarrollada para alcanzarlos:

1. **Automatizar la creación del archivo *"network.xml"***: Mediante nuestra nueva aplicación, el usuario puede navegar de manera manual por el mapa que se le muestra, o buscar directamente una localización que será cargada en el mapa, seleccionar el área que desea utilizar para su simulación y tras pulsar el botón para solicitarlo, descargar tanto el archivo *"network.xml"*, como el archivo fuente original de OpenStreetMap (todo ellos de manera totalmente transparente para él, sin tener que preocuparse del proceso de obtención de datos geográficos, ni de la conversión al formato de MATSim).

2. **Generación del archivo *"plans.xml"***: A través de nuestra interfaz, el usuario puede crear nuevos planes asociados a escenarios ya convertidos, o de forma totalmente libre. La aplicación permite no sólo añadir para cada ruta un punto origen y otro destino, sino también todos los puntos intermedios que se requieran. Se ha facilitado la posibilidad de añadir, modificar, o eliminar rutas dentro de un mismo plan para lograr que los planes puedan cambiarse si surgen

nuevos requerimientos, sin necesidad de rehacerlos por completo.

Además, nuestra apliación no sólo genera el archivo de planes utilizado por MATSim, sino que a su vez almacen toda la información relativa al plan en un segundo archivo auxiliar que permite al usuario final localizar facilmente a cualquier agente.

3. **Interpretación y visualización de los resultados**: Teníamos como objetivo conseguir que esta interpretación de los resultado fuera lo más realista posible, y se ha conseguido no sólo el obtener un sistema que nos proporciona nuestra propia vista de un escenario a partir de un archivo XML, sino gracias a la utilización del elemento CANVAS de HTML5 junto con JavaScript, poder mostrar los resultados de manera dinámica.

   Una vez alcanzado este objetivo se ha continua trabajando en este sentido, obteniendo diferente tipos de representaciones para poder mostrar con mayor precisión los resultado.

4. **Desarrollar un conjunto de herramientas**: Todas las funciones, tanto del lado del cliente como del lado del servidor, han sido desarrolladas para que cumplan pequeñas tareas concretas, que usadas de forma conjunta, nos proporcionan soluciones a problemas mucho más complejos. Por ello, estas funciones pueden utilizarse por separa en el futuro para otros trabajos, o añadirse nuevas extensiones a este aplicación, reutilizando gran parte de las funcionalidades implementadas.

5. **Sencillez de la solución final** Todas las soluciones desarrolladas, están integradas en una aplicación web, lo cual permite que sea accesible desde cualquier Navegador Web moderno, facilitando su uso, acceso, y distribución (además de repartir la carga de trabajo, ejecutando algunos procesos en el cliente y otros en el servidor).

Se ha tratado siempre de encontrar la mejor solución para alcanzar los objetivos, aunque ello implicara tener que comenzar desde cero con una determinada tecnología o solución.

Se han utilizado recursos de libre distribución como los mapas y los servicios asociados de OpenStreetMap o la librería OpenLayers para visualizar mapas, que han aportado grandes mejoras al resultado final.

El propio sistema facilita el almacenamiento de los mapas convertidos, planes creados, y vistas solicitadas, por lo que puede utilizase como un repositorio.

Tras haber finalizado la aplicación Web que se desprende de este trabajo, podemos afirmar que tareas que antes requerían horas para ser completadas por los usuarios de MATSim, ahora pueden realizarse en pocos minutos.

Capítulo 6

Futuras líneas de trabajo

En muchas de las partes desarrolladas se ha partido desde cero y por tanto, aunque se han logrado los objetivos buscado, e incluso en alguno de ellos se ha superado con creces lo planificado, es posible continuar añadiendo nuevas funcionalidades a este sistema desde la base que se ha creado.

Entre las posibles futuras líneas de trabajo, podemos destacar:

- **Adaptación de las funcionalidades desarrolladas en este trabajo a los datos obtenidos de otros simuladores diferentes a MATSim**. Buena parte del tiempo que se ha empleado en este trabajo, se ha dedicado a analizar y resolver problemas que son comunes a cualquier tipo de simulador de tráfico de vehículos, por lo que podría ser interesante realizar pequeñas modificaciones en las funciones especificas que trabajan con datos de MATSim, para modificarlas y adaptar la aplicacion a otros entornos.
- **Personalización de las visualizaciones**. Se han implementado las sistema de visualización que desde nuestro punto de vista podían ser más útiles, pero puede suceder que para determinados usos se requieran otros tipos de vista. Habiendo establecido en este trabajo, la forma de interpretar los datos y mostrarlos, el crear nuevas vista a partir de la base establecida, debería ser bastante sencillo, pero muy útil en determinados casos.
- **Mejoras en el rendimiento**. Aunque siempre se ha tenido presente que la aplicación debería de ofrecer una respuesta lo suficientemente rápida para poder interactuar con el usuario, algunas funcionalidades como la de crear una vista con una escala de colores que organice las vías desde las menos a la más transitadas requiere su tiempo de procesado (aunque siempre se ejecuta en segundo plano, permitiendo continuar con el trabajo). Se podría valorar la opción de utilizar otro tipo de base de datos diferente a MySQL que proporcione mejor rendimiento al trabajar con grandes cargas.

Bibliografía

[1] Romero Chica, José Miguel. “*Investigación para la adaptación de simuladores de tráfico para permitir la modificación de rutas en tiempo real en función de datos sobre la red provenientes de módulos externos*”, 2010

[2] Pérez Hernández, Pablo. ”*Adaptación del simulador de tráfico vehicular MATSim para el cálculo cooperativo de rutas*”, 2011

[3] Vedia Velasco, Andrés. ”*Evaluación de estrategias centralizadas de coordinación de rutas mediante el simulador de tráfico vehicular MATSim*, 2012

[4] López Martínez, María. ”*Evaluación de estrategias distribuidas de coordinación de rutas mediante el simulador de tráfico vehicular MATSim*, 2012

[5] MATSim API Specification. `http://www.matsim.org/apidocs/core/0.4.0/`

[6] Hazzard, Erik. ”*OpenLayers 2.10 Beginner's Guide*”, ISBN: 9781849514125

[7] Flanagan, David. ”*JavaScript*”, Editorial Anaya Multimedia, 2007, ISBN: 9788441522022

[8] Gauchat, Juan Diego. ”*El gran libro de html5, CSS3 y Javascript*”, Marcombo, 2011, ISBN: 9788426717702

[9] Parsons, David. ”*Desarrolo de Aplicaciones Web Dinámicas con XML y Java*”, Editorial Anaya Multimedia, 2008, ISBN: 9788441525924

[10] Wiki OpenStreetMap. `http://wiki.openstreetmap.org/wiki/Main_Page`

[11] Centro de documentación de OpenLayers. `http://trac.osgeo.org/openlayers/wiki/Documentation`

[12] Tutoriales y ejemplos de HTML - w3schools.`http://www.w3schools.com/`

Apéndices

Apéndice A

Sistemas de referencia.

En ocasiones al trabajar con un conjunto de datos podemos abstraernos sin pararnos a pensar en el significado final de los mismo. En algunas partes de este proyecto eso es así, pero a la hora de realizar representaciones de los datos para generar escenarios reales, nos encontramos con la obligación de tener al menos unas nociones básicas para poder saber si los resultados obtenidos son correctos, o para detectar posibles errores.

En este apéndice vamos a explicar de manera resumida algunos conceptos básicos de los sistemas de coordenadas geográficas, sus proyecciones sobre dos dimensiones, y así poder interpretar y comprender cómo un mismo dato físico puede tener diferentes valores dependiendo del entorno en el que estemos trabajando en un momento dado.

A.1. Introducción.

Partimos de la necesidad de tener un sistema de coordenadas que nos permita identificar todos puntos de la superficie terrestre, con la dificultad que entraña al asemejarse la Tierra a una esfera achatada ligeramente por los ejes. Esto lo logramos dividiendo en porciones tanto verticales como horizontales el globo terráqueo a través de líneas imaginarias denominadas:

- **Latitud**: Líneas que se desplazan desde el Ecuador hacia el Norte y hacia el Sur. Nunca se juntan unas con otras, y están situadas de manera casi equidistante (existen pequeñas variaciones, ya que la superficie terrestre no es totalmente homogénea)

- **Longitud**: Líneas que se desplazan hacia el Este y el Oeste. Son equidistantes si tomamos como referencia una franja de latitud, pero entre dos líneas de longitud su distancia es máxima en el Ecuador, y tiende a cero al aproximarse a los polos.

Por tanto, tomando un punto de referencia para la latitud y otro para la longitud y asignándoles el valor cero, podemos indicar con dos datos cualquier posición sobre la superficie de la Tierra. Esto resulta una manera muy sencilla y precisa de trabajar con coordenadas, pero presenta el problema un problema, que normalmente siempre trabajamos sobre una superficie plana, y por ello surge la necesidad del uso de proyecciones.

Una proyección el conjunto de reglas que seguimos para dibujar una superficie esférica como la Tierra, en un superficie plana (pasar de 3D a 2D). Es inevitable que al realizar este paso se produzcan distorsiones de la realidad y haya que tener cuidado a la hora de interpretar los datos. En la figura A.1[1], vemos como al proyectar la superficie terrestre sobre un plano se producen alteraciones tales como que las zonas más al norte tienes un tamaño mucho mayor del real (siempre se suele poner el ejemplo de Groenlandia, la cual parece ser más grande que el continente africano o Sudamérica, cuando en realidad tiene una superficie 14 veces menor). Además, todos los círculos que se muestran en la imagen tiene un radio de 500 km, sin embargo según los alejamos de la línea del Ecuador cada vez son de mayor tamaño.

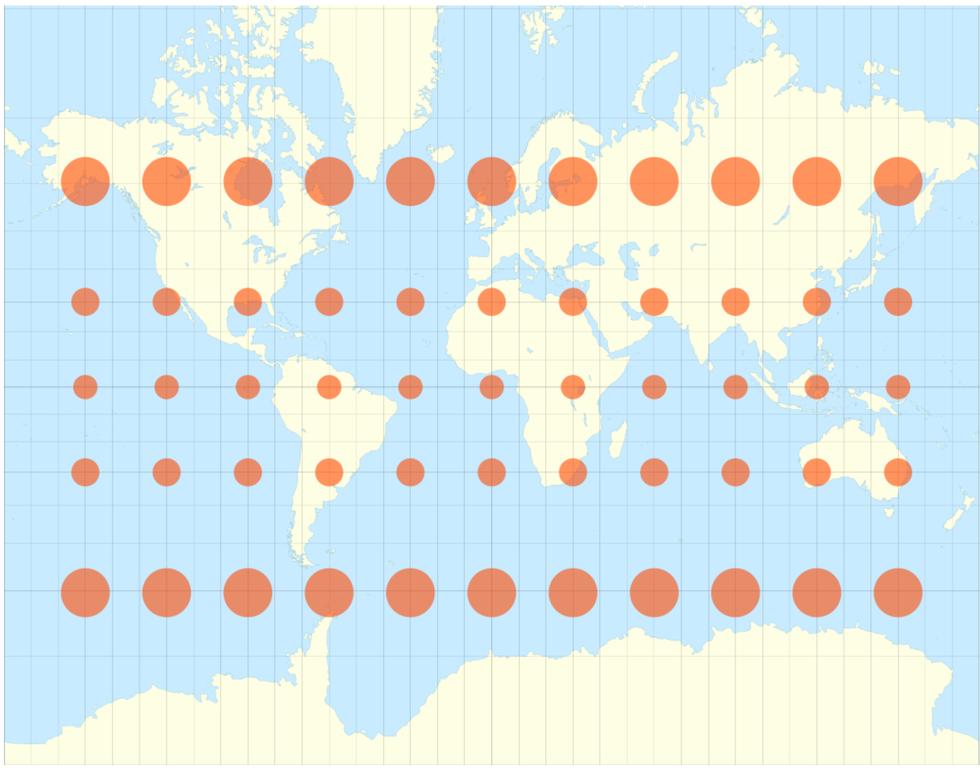


Figura A.1: Ejemplo proyección.

[1]Imagen obtenida de Wikimedia, Eric Gaba `http://commons.wikimedia.org/wiki/User:Sting`

A.2. Sistema de coordenadas WGS84

Todos los sistemas de coordenadas utilizan la misma dinámica, explicada anteriormente, hacer referencia a un punto geográfica a través de su longitud y su latitud.

La variación existente entre unos y otros es el sistema de referencia que utilizan. Para ello aparece el llamado Datum, el cual en cada se sistema se halla con los datos del Geoide (superficie teórica de la tierra que une todos los puntos que tienen igual gravead) y el Elipsoide (modelo matemático que representa la superficie de la tierra). Al variar los modelos matemáticos que se aplican, también varía el Datum, y por tanto la referencia de la que parte el sistema de coordenadas.

El Datum más utilizado es el WGS84, debido a que es el que utiliza el sistema GPS.

A.3. Proyección de Mercator

Es una de las llamadas proyecciones **conformes**, ya que persigue mantener la forma de las superficies u objetos que aparecen en el mapa. Los paralelos y los meridianos se cruzan formando ángulo recto, pero a cambio distorsionan el tamaña de las superficies, lo que provoca que al escala no sea la misma en todas las zonas del mapa.

Dentro de las proyecciones conformes, la de Mercator es **cilíndrica**, es decir, es el resulta de rodear la superficie esférica de la Tierra con un cilindro, proyectar las formas y abrir el cilindro (se trata de la proyección que hemos visto antes en la figura A.1).

El sistema más extendido y utilizado que está basado en esta proyección es UTN (Sistema de Coordenadas Universal Transversal de Mercator), el cual enrolla el cilindro de la proyección sobre cada uno de los meridianos, asignando diferentes *"husos"* dependiendo de la zona donde nos encontremos. La gran ventaja que presenta, es que las referencias utilizando UTN ya no se miden en grados, sino en metros, por lo que resulta muy cómoda su interpretación en un mapa.

A.4. Correspondencia

Para realizar la conversión de coordenadas geográficas a coordenadas en la proyección se utilizan formulas matemáticas que varían según la zona que queramos convertir. Esta es una tarea compleja, que en nuestro caso asumirán las librerías que ya están implementadas en MATSim. Lo que si debemos de tener claro es que lo que ocurre es una adaptación de los datos en 3D a 2D (figura A.2).

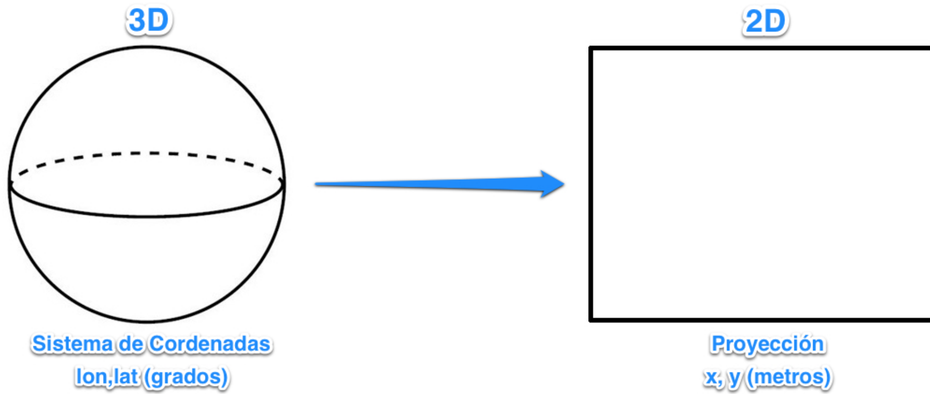


Figura A.2: Paso de sistema de coordenadas a proyección.

Siempre que realicemos una conversión debemos de saber qué Datum se utilizaba en el sistema de coordenadas geográficas, y a qué proyección, y con que huso, han sido convertido los datos, para luego en caso de ser necesario poder realizar la conversión a la inversa.

Además, cabe destacar, que tras realizar estas conversiones se suele producir un redondeo de los decimales, por lo que se produce perdida de información. Esto no tiene mucha transcendencia, pero es aconsejable almacenar los datos tanto en su sistema de coordenadas geográficas como de proyección, para evitar al tratar los datos, hacer conversiones sobre conversiones anteriores, lo que nos llevaría a ir sumando los errores acumulados.

Apéndice B

Manual de usuario

B.1. Introducción.

Este es el manual de usuario de la aplicación web correspondiente al Trabajo Fin de Grado .*Estudio y desarrollo de una aplicación web para la generación y tratamiento de datos del simulador MATSim”*.

Vamos a detallar e ilustrar los pasos a seguir para poder movernos por la aplicación y realizar las tareas para las que ha sido implementada.

Aunque es compatible con cualquier navegador web moderno, ha sido desarrollada y probada utilizando Firefox, por lo que es posible que con otros navegadores, la representación visual de algunas partes varíe (sobre todo si utilizamos Internet Explorer).

La aplicación cuenta con un menú horizontal con cuatro pestañas, a través de las cuales podemos acceder a las diferentes secciones en las que se divide.

B.2. Exportar y convertir.

Pulsando en la pestaña *“Exportación”* accedemos a la ventana de la figura B.1. Por defecto nos sitúa en la opción de exportar un nuevo mapa desde OSM. En la parte de la izquierda tenemos el menú lateral para poder elegir entre diferentes opciones.

- Nuevo - Desde OSM.
- Nuevo - Desde archivo.

- Pendientes.
- Convertidos.

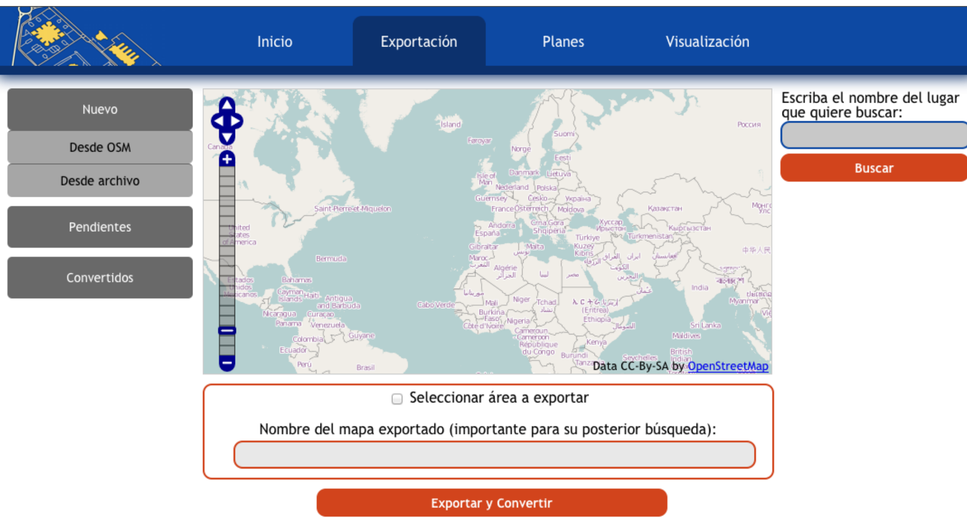
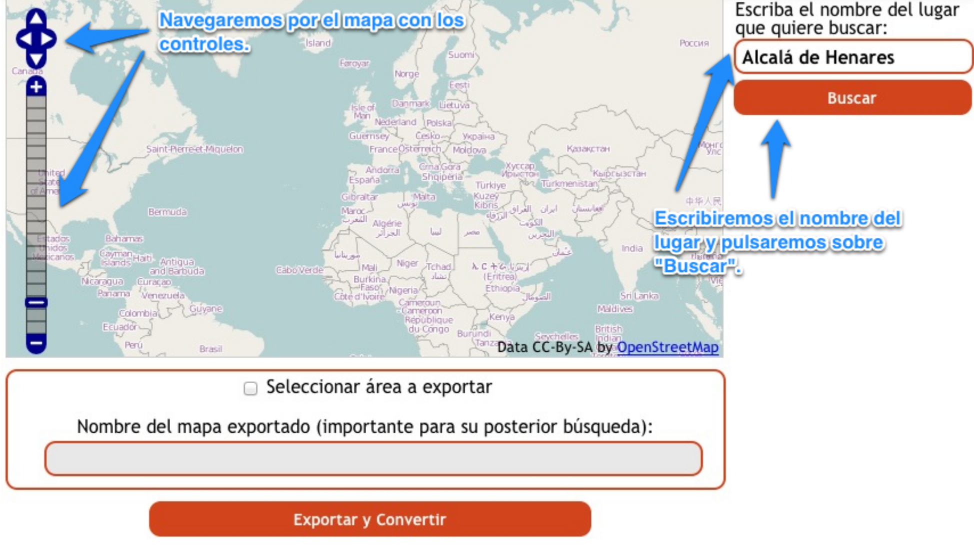


Figura B.1: Contenido pestaña *Exportación*.

B.2.1. Nuevo desde OSM

Nos carga la pantalla que vemos en la figura B.1. A partir de ese momento, podemos utilizar los controles que aparecen a la izquierda del mapa para movernos y aumentar el zoom hasta situarnos en la zona que queramos exportar o bien utilizar la herramienta que aparece a la derecha de la pantalla para situar la vista del mapa según el lugar buscado.



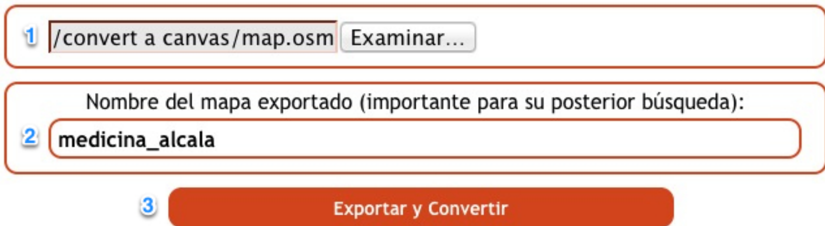
Si elegimos la opción de *Buscar*, se desplegara una ventana centrada en la pantalla con los resultado de la búsqueda. Elegiremos de entre las opciones, la que más se adapte a lo que habíamos pedido y al hacer click sobre ella se cargará esa ubicación en el mapa. En caso de que no encuentre ningún resultado, nos aparecerá un mensaje de error con el texto *"No se encontraron resultados"*.

Para iniciar la selección de un área, marcaremos la opción *"Seleccionar área a exportar"* y dibujaremos un rectángulo sobre el mapa manteniendo presionado el click izquierdo del ratón. Al soltar, quedara marcada esa zona en color amarillo, y a la derecha se habrán cargado las coordenadas del área elegida.

Por último, elegimos un nombre para nuestro mapa (será por el nombre que luego lo busquemos), y pulsamos sobre el botón *"Exportar y Convertir"*.

B.2.2. Nuevo desde archivo

Si queremos subir nuestro propio archivo fuente OSM, pulsaremos en el menú lateral sobre la opción Nuevo, y después sobre Desde archivo, y nos aparecerá el formulario.



Pulsaremos sobre *Examinar* para elegir el archivo a subir, y escribiremos un nombre que le vamos a asignar al mapa, y al igual que en el apartado anterior pulsamos sobre *Exportar y Convertir*. Debemos esperar hasta que aparezca el mensaje *"El archivo se ha subido"* que nos indicara que el proceso de subida del archivo se ha completado correctamente. En caso de aparecer algún mensaje de error, le especificará el motivo por el cual no se ha podido subir.

B.2.3. Pendientes

Una vez hagamos la petición de conversión de los archivos, se irán acumulando todas las peticiones hasta ser procesadas por completo.

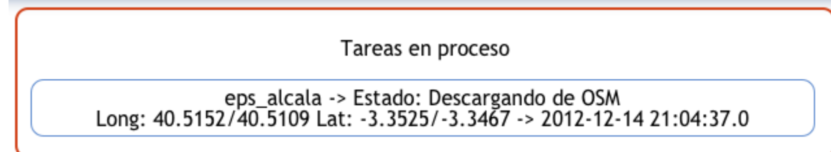
Si entramos en el menú lateral, en la opción *Pendientes*, nos aparecerá el listado de todos los mapas que hemos solicitado pero que todavía no han finalizado todo el proceso.

Si todas nuestras tareas se han completado veremos el siguiente mensaje:

Sin tareas pendientes.
Búsquelas en la pestaña "Convertidos".

Si algún proceso está todavía en curso, veremos un listado donde aparecerán con su nombre, coordenadas, hora de solicitud, y estado. Los estados posible son:

- Descargando de OSM
- Convirtiendo
- Error Conversión



B.2.4. Convertidos

Al finalizar el proceso, podremos descargar el resultado de nuestro proceso pulsando sobre la pestaña *"Convertidos"*, donde aparecerán un listado de todos los mapas que hubiéramos solicitado y hayan finalizado el proceso.

Para asegurarnos de que sea el mapa que queremos, si pulsamos sobre *"Ver"* nos cargará las coordenadas del mapa (solo en los mapas exportados directamente de OSM).

Podemos descargar tanto el archivo fuente, como el convertido:

- **Descargar OSM**: Archivo con la información fuente en formato de OpenStreetMap.
- **Descargar XML**: Archivo en formato MATSim.

B.3. Generación de planes.

Pulsando en la pestaña *“Planes”* accedemos a la ventana de la figura B.3. Por defecto nos sitúa en la opción de crear un nuevo plan. En la parte de la izquierda tenemos el menú lateral para poder elegir entre diferentes opciones.

- Nuevo Plan/Zona.
- Ver/Modificar Plan.
- Pendientes.
- Finalizados.

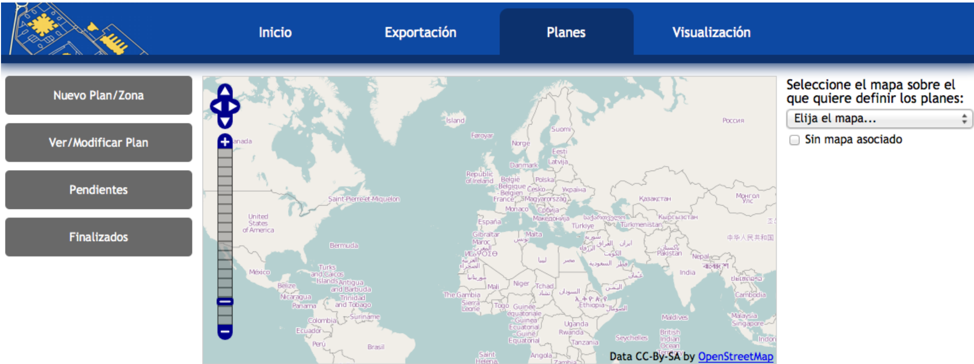


Figura B.2: Contenido pestaña *Exportación*.

B.4. Generación de planes.

Pulsando en la pestaña *“Planes”* accedemos a la ventana de la figura B.3. Por defecto nos sitúa en la opción de crear un nuevo plan. En la parte de la izquierda tenemos el menú lateral para poder elegir entre diferentes opciones.

- Nuevo Plan/Zona.
- Ver/Modificar Plan.
- Pendientes.
- Finalizados.

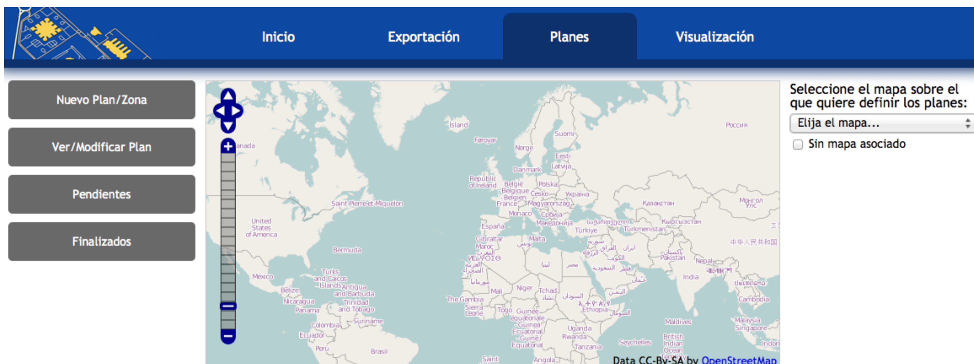
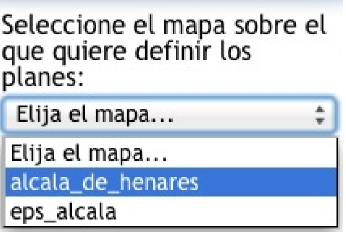


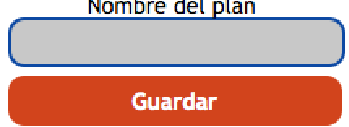
Figura B.3: Contenido pestaña *Exportación*.

B.4.1. Nuevo Plan/Zona

Tras cargarnos la página tendremos un área de trabajo como el que aparece en la figura anterior. En la parte de la derecha tenemos una lista desplegable donde aparecen los mapas que hayamos convertido anteriormente, en caso de no tener ninguno aparecerá un único registro con la denominación *"Sin Mapas"*. La creación de los planes va asociada a los mapas para facilitar el trabajo, pero también puede ser creado en una categoría genérica.



Debemos elegir en la lista de la derecha el nombre del mapa al que queremos asociar nuestro nuevo plan. En el caso de no querer asociar nuestro plan con un mapa, debemos marcar la casilla *"Sin mapa asociado"*, que se encuentra justo debajo de la lista desplegable. Al elegir cualquiera de las dos opciones, nos aparecerá una nueva lista desplegable con todos los planes asociados a ese mapa. En caso de querer añadir un nuevo plan, marcaremos la casilla *"Nuevo Plan"*, y debajo se nos mostrará el siguiente cuadro de texto para que indiquemos el nombre el nuevo plan:

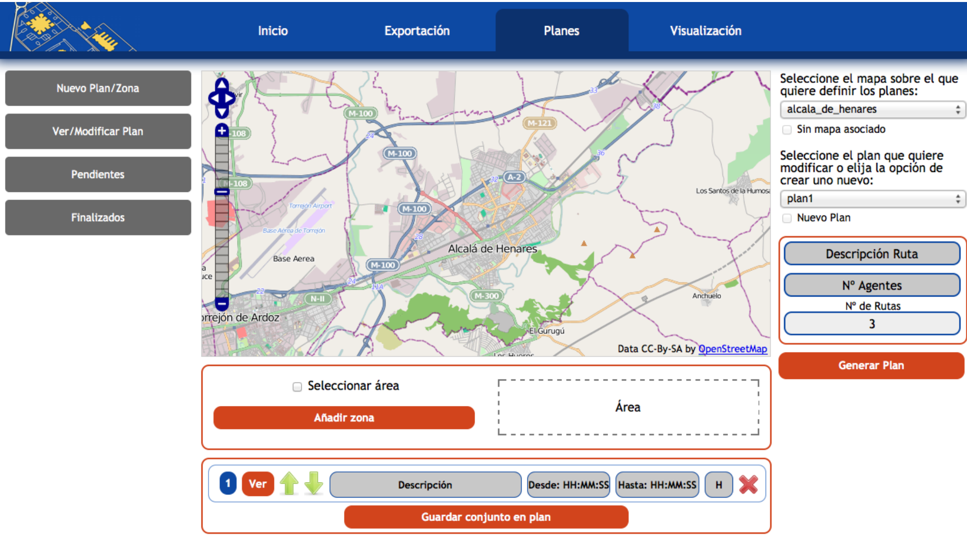


Al elegir cualquiera de las dos opciones, aparecerán el resto de las herramientas para continuar con el proceso.

Tras pulsar el botón de "Guardar", se verificará que el nombre el nuevo plan

no exista ya (nos avisará en caso de error), y será almacenado en memoria.

Lo opción que elijamos debe de permanecer inalterada durante todo el proceso de creación del plan, ya que si variamos el mapa o el plan perderemos el trabajo que llevemos hecho hasta ese momento (por ello una vez elegido un plan, se deshabilita el desplegable de mapas, para evitar posibles errores).



Cada plan está compuesto por las rutas de agentes que definamos. Al seleccionar un nombre de plan, o crear uno nuevo, en el cuadro resumen de la derecha podremos ver el número total de rutas que componen ese plan (cero en el caso de ser un nuevo plan). La metodología para añadir nuevas rutas al plan es la siguiente:

- Seleccionar y añadimos tantas zonas del mapa, como puntos en la ruta queremos tener. Se deben introducir un mínimo de dos, que actuarán como origen y destino (en caso de no ser así, la aplicación nos mostrará un mensaje de error al tratar de guardar la ruta).

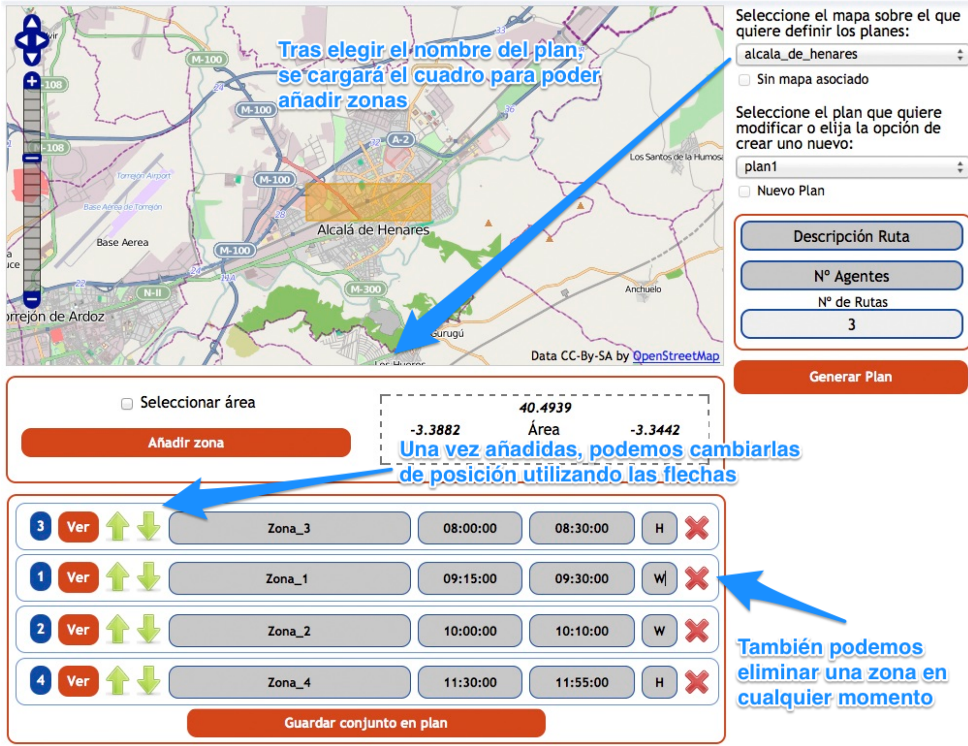
- A cada zona, le asignamos una breve descripción que nos ayude identificarla, el horario de partida de los agentes, y el tipo de actividad.

- Podemos añadir, eliminar, o variar el orden de las zonas de la ruta, utilizando los controles habilitados para ello hasta el momento de guardar el conjunto.

- En el cuadro resumen de la derecha debemos añadir una descripción para todo la ruta y el número de agentes que van a intervenir.

- Por último, pulsaremos sobre el botón “Guardar conjunto en plan”, y si hemos rellenado todos los campos, la nueva ruta quedará almacena y veremos que se ha incrementado el número de planes que se muestra en una unidad.

Para cada zona, aparecen los siguientes datos:



- Nº de la zona. Se corresponde con la posición en la que fue insertada, y esta se mantiene aunque se eliminen, o cambien de lugar, para mantener una cierta coherencia.
- Botón "Ver", con el cual podemos dibujar de nuevo la zona en el mapa (por si no recordamos cual era).
- Flechas de posición, que nos permiten subir a bajar una zona ya insertada dentro del grupo.
- Cuadro de texto para añadir una breve descripción de la zona.
- Franja horario, desde/hasta, cuando inician su trayecto cada agente desde esa zona.
- Tipo de actividad. Por defecto aparece el valor "H", pero se puede variar si se desea dar más información (que puede ser tratado posteriormente desde MATSim).
- Aspa para eliminar una zona.

Podemos mover o eliminar zonas libremente, sin preocuparnos de alterar los datos ya introducidos en otras zonas, ya que estos quedan almacenamos de forma temporal hasta que se proceda a guardar todo el conjunto.

Se pueden introducir tantas zonas dentro de una misma ruta como se requieran (la primera se considerará como punto origen, y la última como punto destino). De igual forma, se pueden incluir todas las rutas que se deseen a un mismo plan.

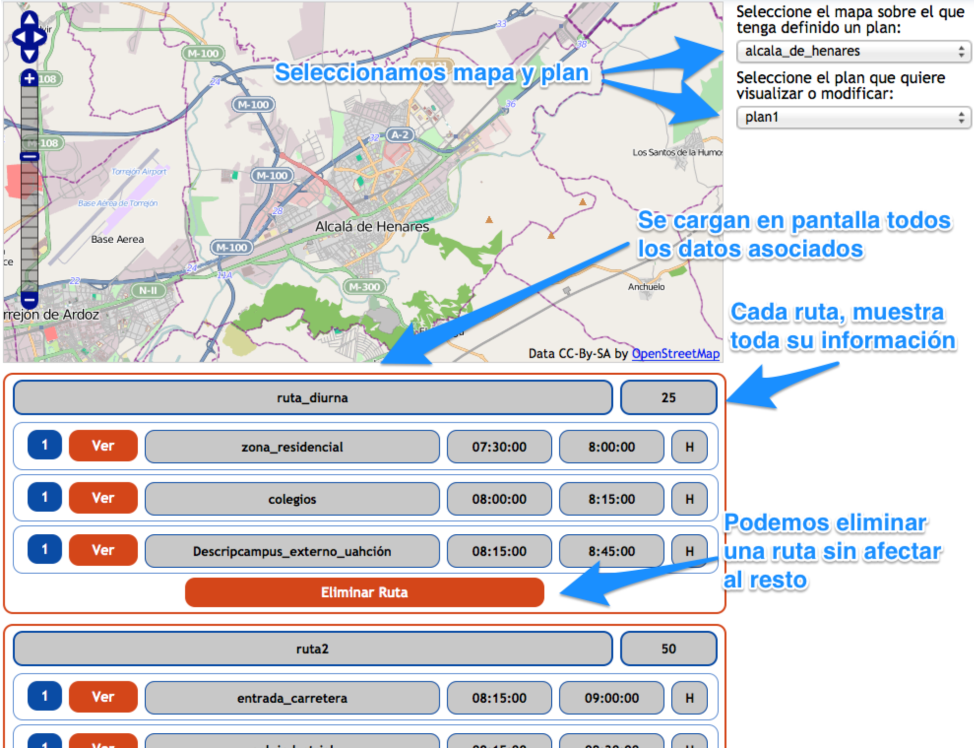
Una vez almacenadas todas las rutas que van a componer nuestro plan, pulsaremos sobre el botón *"Generar Plan"*. La página se recargará cuando haya completado la petición.

Nota: Es importante que las descripciones que se indiquen tanto para las zonas de cada ruta, como para la propia ruta, sean claras e indicativas, ya que aparecerán en los comentarios del archivo de planes final, y nos pueden ayudar a localizar con posterioridad estos datos.

B.4.2. Ver/Modificar Plan

Puede ocurrir que queramos revisar en un determinado momento todas las rutas que componen un determinado plan, o que tras haber almacenado una ruta en el apartado anterior queramos eliminarla sin necesidad de tener que volver a añadir todos los datos del plan.

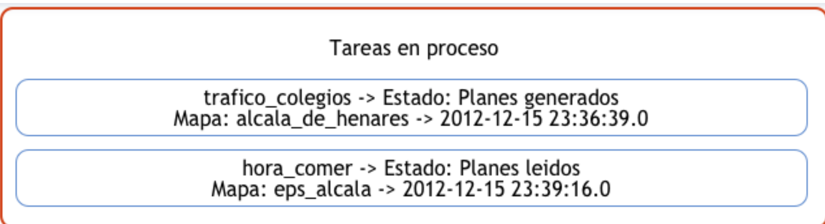
Para ello, pulsaremos sobre el menú lateral "Ver/Modificar Plan", y elegiremos en la lista desplegable de la derecha, el nombre del mapa que contiene nuestro plan, y el nombre del propio plan. En pantalla se cargarán los datos almacenados de ese plan como podemos ver en la siguiente imagen:



Cada ruta muestra toda su información asociada. En caso de querer eliminar una de las rutas, tras localizarla pulsaremos sobre el botón "Eliminar Ruta" y después de procesar la petición, la página se recarga sin esa ruta.

B.4.3. Pendientes

El proceso de creación de planes, es mucho más rápido que las conversiones del apartado anterior, incluso así, pulsando sobre el menú lateral izquierdo sobre *Pendientes*, se nos mostrará un listado de las peticiones para generar un plan que hemos solicitado y todavía no han finalizado.



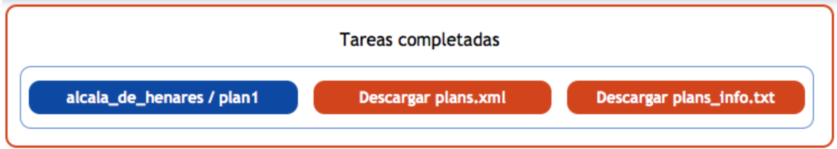
Los estados posibles son:

- Petición recibida, para indicar la que petición se ha guardo en el sistema.
- Planes leídos, se han procesado los datos.
- Planes generados, se ha generado los planes.
- FINALIZADO, se ha almacenado en disco y puede ser descargado.

En caso de no haber ninguna tarea pendiente, se nos avisará de igual modo.

B.4.4. Finalizados

Por último para poder descargar el archivo de planes generado, pulsaremos en el menú lateral izquierdo sobre el enlace *Finalizados*.



Aparecerá un listado con todas las tareas hayan finalizado el proceso de creación de planes. Se muestran por el orden de creación, y nos indica el mapa al qué están asociadas y el nombre que reciben.

Pulsando sobre el botón *"Descargar plans.xml"*, se iniciará la descargar del archivo que contiene el listado de los planes que hemos definido, y pulsando sobre *"Descargar plans_info.txt"* el archivo con la información auxiliar asociada al plan.

B.5. Visualización de los resultados de simulación

Pulsando en la pestaña *“Visualización”* del menú principal se cargará la vista que aparecen la figura B.4. Por defecto, nos sitúa en al opción de crear una nueva vista. En la menú de la izquierda podemos elegir entre:

- Nuevo
- Pendientes
- Finalizados

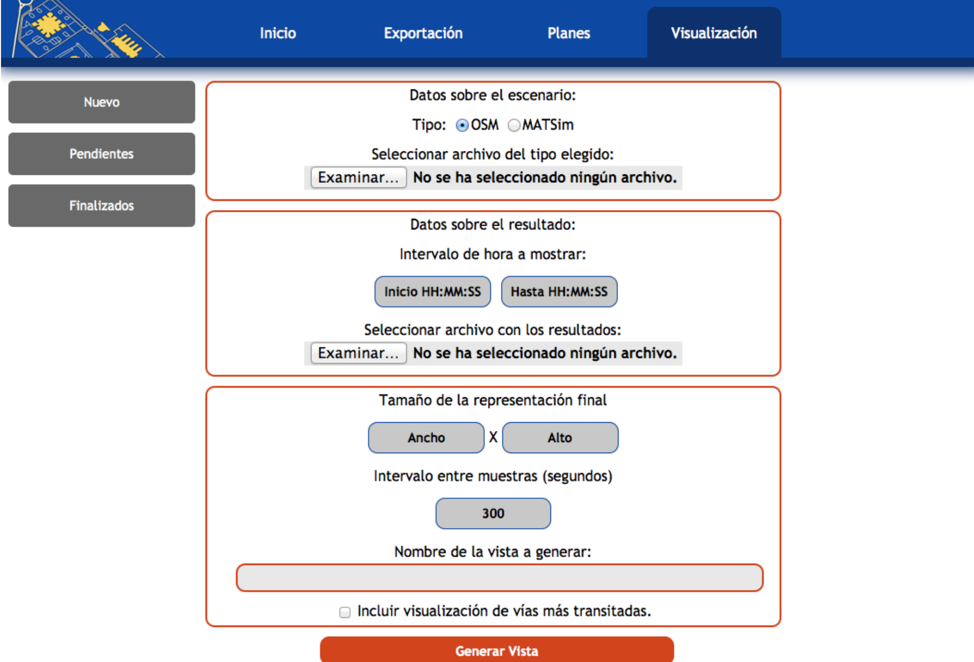


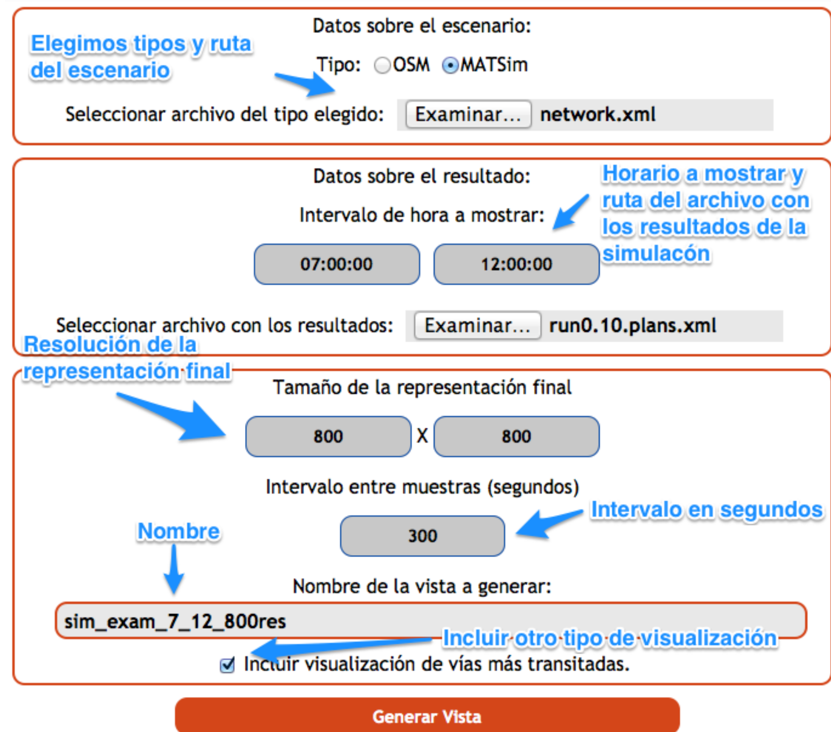
Figura B.4: Contenido pestaña *Vista general pestaña Visualización.*

B.5.1. Nuevo

Tras cargar la página tenemos la interfaz que nos facilita la generación de las vista.

Debemos seguir los siguientes pasos:

1. Seleccionar el archivo que contiene el escenario de la simulación y elegir su formato origen. La aplicación permite dos opciones en este apartado:
    - OSM: Normalmente archivo del tipo *map.osm* a partir del cual se obtuvo el escenario de simulación. Contiene más detallas y la visión es más



realista, pero a cambio, debido a que utiliza sistemas de coordenadas diferentes al de los resultado, la posición que presentará los agentes de la población puede no coincidir de manera tan exacta como la siguiente opción, ya que es inevitable que se produzcan redondeos de las operaciones de conversión.

- MATSim: Archivo *network.xml*, que hemos usado a la entrada del simulador (Opción recomendada, ya que los resultados son más rápidos de generar y los agentes coincides de manera precisa con las vías por las que transitan).

2. Definir el intervalo de tiempo que vamos a simular.

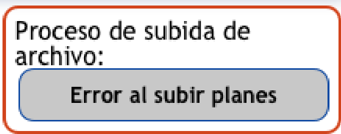
3. Elegir el archivo que contiene los resultados de la simulación.

4. Seleccionar la dimensión en pixeles de las representaciones solicitadas (para mantener la relación de aspecto del escenario, se tomará el valor más restrictivo entre el alto y ancho).

5. Marcar la casilla "Incluir visualización de vías más transitadas", si queremos que además de la vista normal, se genero otra con una representación con escala de colores sobre las vías, indicando desde un verde claro para las menos transitadas, hasta el rojo de las más utilizadas (esta operación conlleva un tiempo de procesamiento bastante elevado).

6. Definir un nombre para la vista y pulsar sobre "Generar Vista".

La aplicación comprobará la correcta subida de los archivos al servidor, y en caso de producirse algún tipo de error se nos mostrará en la parte derecha de la pantalla el mensaje:

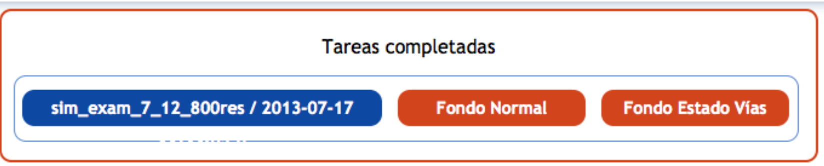


B.5.2. Pendientes

Tiene la misma funcionalidad que la de los apartados anteriores. Nos mostrará por pantalla el listado de las vistas que hayamos solicitado y que todavía no estén finalizadas.

B.5.3. Finalizadas

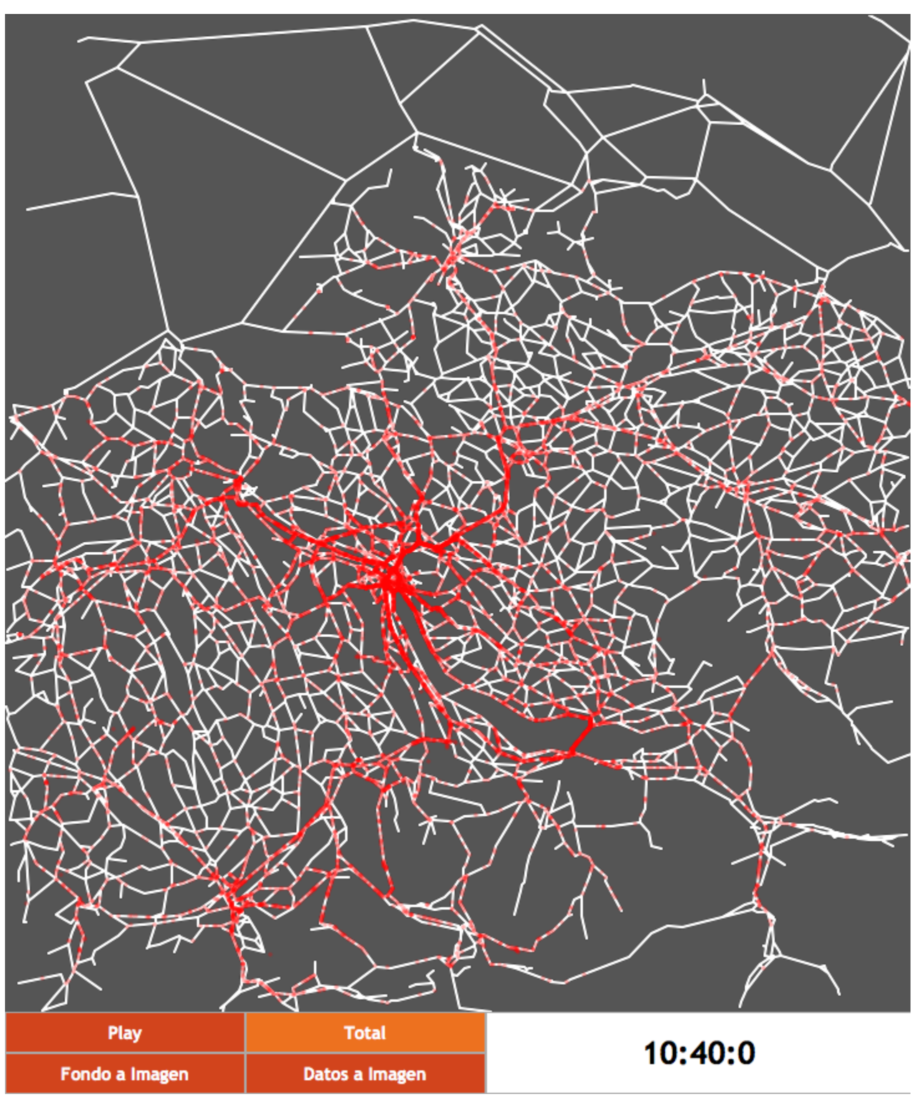
Veremos un listado de todas las vistas solicitadas y finalizadas.



Nos aparecerá el enlace para abrir el reproductor tanto de la vista normal, como en caso de haberlo solicitado de la vista con información adicional. Las vistas se abren en nuevas ventanas del navegador para poder trabajar con varias al mismo tiempo.

En la nueva ventana, tenemos los botones que nos permiten realizar diferente operaciones.

- **Play/Stop**. Inicia o para la reproducción del desplazamiento de los agentes sobre el escenario en color azul.
- **Total/Stop**. Inicia o para la reproducción de las posiciones por donde van transitando los agentes en color rojo. Es una muestra acumulativa, por lo que en el momento en que la detengamos, tendremos sobre el escenario es rastro por donde han transitado todos los agentes hasta ese instante.
- **Fondo a imagen**. Lanza la descarga del CANVAS de fondo convertido a imagen de tipo PNG.
- **Datos a imagen**. Lanza la descarga del CANVAS que contiene los desplazamientos de los agentes, convertido a imagen de tipo PNG



Además, contamos con un cuadro de texto que actúa como indicador del instante temporal que se está reproduciendo en cada momento.

Por último, comentar que las imágenes que se obtiene son del mismo tamaño, y la que contiene la información sobre los agentes no tiene fondo, por lo que pueden ser superpuestas posteriormente de manera muy sencilla.