Universidad Nacional Autónoma de México
Facultad de Ciencias

Planteamiento de un Sistema de Información para el Transporte Público

---

T E S I S

para obtener el grado de Licenciado en Ciencias de la Computación

Autor:
Alejandro Eduardo Cruz Paz

Directora:
María de Luz Gasca Soto

Ciudad Universitaria, México
FEBRERO 2014

Índice general

Introducción **1**

I Marco Tecnológico **3**

Capítulo 1. Sistemas de información para el transporte público **5**

1.1. Antecedentes 5
1.2. Planteamiento 7
1.3. Sistemas de información geográfica 7
1.4. Sistemas de tiempo real 8
1.5. El proceso de diseño y desarrollo de un sistema 10
1.5.1. Metodologías ágiles 10

Capítulo 2. Desarrollo del modelo **13**

2.1. Representación de datos del transporte público 16
2.1.1. Especificación GTFS 16
2.1.2. Modelo TransModel 20
2.1.3. Especificación mixta de GTFS y Transmodel 22
2.2. Alcances del modelo 23
2.3. Análisis de herramientas 25
2.3.1. Elección de un manejador de bases de datos 28
2.3.2. Elección de un lenguaje de programación y un *framework* web y propuesta de arquitectura para el *backend* 30
2.3.3. Representación de la red 31
2.3.4. Manejo de datos dinámicos 36

2.4. Algoritmos y distribución de datos 48

II Aplicaciones 51

Capítulo 3. Aplicación del modelo propuesto 53

3.1. Manipulación de la topología de la red 53
3.2. Pumabús 59
3.2.1. Solución actual y tipo de datos dinámicos disponibles 59
3.2.2. Retos y oportunidades de mejora 60
3.3. Metrobús 60
3.3.1. Estado actual y solución propuesta 61
3.3.2. Ejemplo de integración para redes de transporte público al modelo . . 61
3.4. Nuevas tecnologías en hardware 67

Conclusiones 69

Bibliografía 71

III Apéndices 75

A. Sistemas formales y sistemas de información 77

A.1. Enfoque formal de sistema 80
A.2. Sistemas de información 83
A.2.1. Algoritmo 83
A.2.2. Información 84
A.2.3. Sistema de información 85

B. Algoritmos de enrutamiento 87

B.1. Recorridos en gráficas 88
B.1.1. DFS (Depth-First Search) 88
B.1.2. BFS (Breadth-First Search) 88
B.2. Rutas más cortas 89

B.2.1. Dijkstra 89
B.2.2. Algoritmo A* 90

C. Mecanismos de persistencia de datos **93**
C.1. Basadas en el estándar SQL 93
C.2. No basadas en el estándar SQL: NoSQL 94
C.2.1. El manejador de BD NoSQL Neo4J 96

D. Scala, un lenguaje funcional orientado a objetos **99**
D.1. Traits 99
D.2. Funciones 100
D.3. Apareamiento de patrones mediante clases 103
D.4. Extractores e inyectores 104
D.5. Soporte para procesos concurrentes 105

E. Aplicaciones web **109**
E.1. Características de una aplicación web 109
E.2. Arquitectura de una aplicación web 111

Introducción

Actualmente, las ciencias de la computación están presentes en casi cualquier contexto de la vida diaria y el área de la transportación no es la excepción.

En nuestra vida diaria es posible encontrar sistemas de cómputo que desempeñan un papel importante cuando nos trasladamos de un lugar a otro (aunque en muchas ocasiones no notemos su presencia); tal es el caso de sistemas embebidos en automóviles de reciente generación, o bien del sistema de control en la red del metro. Cada uno de estos sistemas plantean soluciones a diferentes problemáticas para diferentes tipos de transporte. En este trabajo, se exploran algunas problemáticas presentes en los sistemas de transporte público y se presentan algunas propuestas para solucionarlas.

Particularmente, se analizan las necesidades de información de los usuarios del transporte público relacionadas a la tarea de encontrar la mejor ruta entre dos puntos considerando datos en tiempo real y múltiples modos de transporte, para después presentar una propuesta de sistema que integre los componentes tecnológicos necesarios para dar la respuesta más óptima y confiable al usuario.

La propuesta de sistema que se presenta en este trabajo considera datos estáticos, datos dinámicos del transporte público, así como también datos generados por sus usuarios para su posterior distribución, almacenamiento y procesamiento. Datos geográficos como el trazo de líneas o rutas, la ubicación de paradas o estaciones y datos estadísticos como las frecuencias de paso de vehículos pueden ser considerados estáticos, mientras que datos como la posición geográfica de un vehículo, el estado de una ruta o línea de transporte, un imprevisto o accidente en algún momento determinado serían considerados dinámicos. Finalmente, la retroalimentación del servicio por parte de sus usuarios resulta de utilidad para las agencias de transporte público, así como para el diseño de la arquitectura del sistema propuesto.

El presente trabajo está organizado de la siguiente manera:

En el Capítulo 1 se abordan los conceptos principales relacionados con el sistema que se plantea en este documento, así como algunas metodologías de desarrollo.

En el Capítulo 2 se plantean los alcances del modelo, se analizan tecnologías que podrían funcionar, se refina el modelo para el sistema basándose en estándares internacionales y finalmente se detalla cada aspecto del modelo.

En el Capítulo 3 se presenta un breve análisis del modelo aplicado a dos casos concretos

de sistemas de transporte público; el Metrobús y el Pumabús.

Finalmente, se presentan apéndices que cubren temas especializados en los que se apoya este trabajo.

Parte I

Marco Tecnológico

Capítulo 1. Sistemas de información para el transporte público

En este trabajo se consideran dos tipos de usuarios de los sistemas de información para el transporte público; sus pasajeros y las agencias de transporte que administran la información de la red de transporte público. A partir de esta diferenciación de casos de uso, resulta más sencillo dar una definición más específica de un sistema para cada tipo de uso. Por tanto,

- Un **sistema de información para pasajeros** es un sistema de información que de manera ideal ofrece información en tiempo real sobre los tiempos de arribo y de salida de los vehículos de transporte, notifica a los pasajeros sobre cambios en el itinerario y/o sobre alteraciones en el servicio. Dependiendo de la infraestructura disponible en un medio de transporte, los pasajeros pueden recibir información a través de indicaciones auditivas, pantallas instaladas en las estaciones de transporte y/o mediante aplicaciones web ó móviles.
- Un **sistema de administración de flotas** es un sistema de información que permite a las agencias de transporte público y/o gobierno administrar y monitorear los vehículos en su red de transporte público, así como mecanismos de cobro a los pasajeros usando tarjetas RFID.

1.1. Antecedentes

El tema del transporte público siempre ha sido importante, particularmente en ciudades con alta densidad poblacional, sin embargo, ha sido en años recientes que en las ciudades latinoamericanas se le ha puesto mayor énfasis a la mejora y extensión de la red de transporte público existente. En estas ciudades, es común encontrar transporte público que se apoya en tecnologías muy básicas, la mayoría de ellas de tipo mecánicas y en algunas excepciones digitales. Sin embargo, la transición hacia tecnologías que ayuden a mejorar de manera significativa la experiencia de viaje del pasajero mediante información confiable y actualizada aún están en proceso de implementación.

Este no es el caso de ciudades europeas como Londres, Bruselas, París, Viena o Amster-

dam, que desde hace al menos una década cuentan con sistemas de información para pasajeros que mejoran significativamente la experiencia de viaje de sus usuarios y que constantemente están innovando en la comunicación con el pasajero antes, durante y después del viaje.

Por ejemplo, algunas de las ciudades antes mencionadas, cuentan con pantallas que indican los minutos que faltan para que el próximo tren o autobus llegue a estación, en algunas otras (como es el caso en el metro de Bruselas) hay un panel en el que se puede visualizar la ubicación de los vehículos a lo largo de la línea de transporte. Además, existen infinidad de aplicaciones móviles que permiten a los usuarios planear un recorrido con la información que las agencias de transporte tienen a su disposición o con información de terceros, como por ejemplo CityMapper [cit14] (presente en varias ciudades de Europa y América) y NextStop [nex14] (presente en las principales ciudades de Austria).

Sin embargo, en Helsinki, Finlandia; los vehículos de transporte público cuentan con sistemas de localización automática (AVLs) basados en GPS para determinar su ubicación aproximada, la cual es enviada a un servidor centralizado desde donde son consultadas por una aplicación web que despliega un mapa donde se pueden visualizar dichos vehículos en tiempo real [Hel14]. Éste, fué uno de los primeros sistemas en su tipo en ser implementado, data del 2007Â y aún se encuentra en operación.

Otro ejemplo, que merece la pena mencionar es OpenTripPlanner [Ope11], el cual ofrece una solución *open source* para la generación de rutas óptimas entre dos puntos considerando diferentes medios de transporte. En el último año han estado trabajando en robustecer su algoritmo de generación de rutas óptimas para tomar en cuenta datos en tiempo real.

Google ofrece un servicio de generación de rutas óptimas del transporte público en algunas ciudades, el cual opera con los datos que el proyecto Google Transit [Goo13] obtiene de agencias de transporte público y que permite detallar los aspectos más reelevantes de una red de transporte público. Más adelante se hablará con mayor detalle sobre este proyecto y los formatos de intercambio y representación de datos del transporte público que impulsa Google a través de él.

En el entorno académico, resaltan Carlsson y otros [BCE+10] quienes plantean una solución óptima para la generación de rutas multimodales tomando en cuenta diferentes limitantes como días de tráfico, caminar entre estaciones, consultas entre puntos geográficos en lugar de consultas entre estaciones origen y destino, funciones de costo ajustables, etcétera.

En México, sin embargo, la implementación de las tecnologías necesarias para obtener datos automáticos en tiempo real del transporte público, va un poco más lento; pues aún se sigue dando más prioridad a la infraestructura para autos, que a la infraestructura para el transporte público. Existen proyectos que emergen de ONGs o de ciudadanos interesados en el tema como el proyecto ViaDF [Via11] que cuenta con una base de datos de rutas de todos los medios de transporte público existentes en el DF y genera rutas óptimas a partir de esos datos.

1.2. Planteamiento

En la actualidad, es común encontrar aplicaciones web y móviles conocidos como sistemas de *wayfinding*, que facilitan la orientación del usuario en una ciudad y le ofrecen rutas óptimas para desplazarse en ella. Cada ruta es presentada al usuario a modo de instrucciones detalladas en las que puede desplazarse en una ciudad usando uno o más medios distintos de locomoción combinados (viajes multi-modales), dentro de los cuales se encuentran por lo general: auto, transporte público, bicicleta o caminar. Dependiendo de la sofisticación de cada sistema, éste podría ofrecer al usuario opciones adicionales a considerar en el algoritmo de generación de la ruta óptima de desplazamiento como el tráfico o saturación del metro.

El sistema cuyo diseño se plantea en este trabajo, además de considerar algunas de las funcionalidades existentes en otros sistemas, considera a los datos dinámicos como un aspecto importante durante el proceso de generación de rutas óptimas multi-modales. Los puntos en los que se enfocará este trabajo son los siguientes:

- Representación en memoria de la red de transporte público.
- Herramientas para manipular la representación de la red de transporte público.
- Representación y manipulación de datos dinámicos.
- Generación de rutas óptimas multi-modales segmentadas en viajes usando el transporte público, usando bicicleta y caminando.
- Consumo e incorporación de datos desde y hacia otros sistemas.

En lo que resta de este capítulo se describirán los aspectos más importantes del diseño del sistema que se plantea: sistemas de información geográfica y sistemas de tiempo real. [1]

1.3. Sistemas de información geográfica

Un sistema de información geográfica (SIG) es un sistema computacional que cuenta con las siguientes capacidades para el manejo de datos geo-referenciados:

1. Captura y preparación de datos
2. Administración de datos, incluyendo almacenamiento y mantenimiento
3. Análisis y manipulación de datos
4. Presentación de datos

[1] Para conocer los fundamentos básicos de un sistema de información, ver el Apéndice A.

Un SIG por lo tanto, debe permitir al usuario ingresar datos en alguno de los múltiples formatos geo-espaciales disponibles, proveerle herramientas para analizarlos y un componente de visualización como un mapa o una gráfica de modo que el usuario pueda entender de mejor manera el fenómeno que los datos describen.

A diferencia de otro tipo de sistemas de información, en un SIG la base de datos juega un papel mucho más reelevante, pues los datos geo-espaciales que describen los fenómenos a estudiar constan de atributos los cuales en muchas ocasiones tienen representaciones particulares que solo bases de datos con soporte geo-espacial son capaces de manipular.

Los SIG y SIS[2] son capaces de representar datos espaciales; sin embargo, el primero incorpora funciones que hacen posible realizar operaciones geo-espaciales sobre los objectos representados.

Algunas de las capacidades específicas con las que los SIG suelen contar son:

- **Análisis topológico, geométrico y de conjuntos:** Por la naturaleza de los fenómenos que un GIS modela en muchas ocasiones es necesario poder efectuar operaciones que permitan conocer si un objeto está dentro de otro, la organización jerárquica entre objetos o bien su nivel de conexidad y adyacencia.

- **Atributos, superficies y capas:** Muchas aplicaciones usan atributos espaciales como por ejemplo, la variación de la elevación topológica en una región. Tales atributos pueden ser discretos o continuos, pueden ser simples escalares o estar representados mediante vectores. Algunas operaciones entre campos es la superposición de capas que generalmente contienen campos espaciales de algún tipo, análisis topológico, localización de caminos y aristas, análisis de flujo, entre otros.

- **Análisis de redes:** Una red o una gráfica es una configuración de conexiones entre nodos mediante aristas. Las operaciones más comunes incluyen análisis de conectividad, algoritmos para encontrar las rutas óptimas entre dos nodos, análisis de flujo, trazado de proximidad. Para el sistema que se propone en este trabajo, esta capacidad es una de las más importantes.

1.4. Sistemas de tiempo real

Un sistema se considera de tiempo real si la diferencia de tiempo tomada a partir de que el sistema recibe una entrada y hasta que genera una salida es de suma importancia para la validez de la respuesta de salida. Se puede decir que estos sistemas imponen reestricciones en el tiempo para generar una respuesta de salida; cuando ésta no cumple con dichas reestricciones (aunque sea correcta) será considerada como una respuesta no válida. Con base en lo anterior surgen dos clasificaciones principales en cuanto a tipos de sistemas en tiempo real:

[2]SIS: Sistema de Información Espacial, por sus siglas en inglés

- **Sistemas de tiempo real duros:** Aquellos para los cuales toda salida generada, siempre deberá cumplir con las restricciones de tiempo establecidas.
- **Sistemas de tiempo real suave:** Aquellos en los cuales los tiempos de respuesta tienen cierta importancia, pero para los cuales no hay repercusiones graves en caso de que algunas veces la salida no cumpla las restricciones de tiempo establecidas.

El papel de un sistema de cómputo de este tipo, es funcionar como un componente de procesamiento dentro de un sistema de ingeniería más complejo, dado que, en su mayoría, adquieren datos del mundo físico a través de componentes electrónicos que posteriormente procesan para generar una salida y actuar sobre el mundo físico nuevamente. Un ejemplo de esto, sería un sistema embebido en un automóvil, el cual se encarga de monitorear la actividad en su entorno inmediato y a la vez, tiene la tarea de reportar la actividad capturada a un sistema que monitorea el tráfico en un área de una ciudad.

Un sistema de tiempo real posee algunas características que lo distinguen de otro tipo de sistemas y que imponen un conjunto de requerimientos sobre los lenguajes que son más propicios para construirlos.

Una característica importante de los sistemas de tiempo real es su complejidad; tanto en su arquitectura como en los datos que manejan, esto es porque la mayoría de ellos trabajan con eventos que ocurren en el mundo real en un lugar específico o en múltiples lugares donde diferentes actores intervienen. Aun así, como todo sistema, han de evolucionar a lo largo del tiempo para mantenerse vigentes. Un patrón de desarrollo de software muy utilizado para mitigar este problema dentro del contexto no sólo de los sistemas de tiempo real, sino de los sistemas extensos y complejos es la modularización de componentes. Para algunos de ellos, la precisión de la salida es muy importante, pues en ocasiones está ligada a la entrada; particularmente, si proviene de dispositivos digitales que capturan datos de alta precisión.

Otra característica es la confiabilidad y robustez que deben tener a lo largo de su operación. Aunque no siempre es crucial que no haya retrasos en la generación de la salida como antes ya se vió, sí es importante (como en todo buen sistema) que la salida no sea errónea desde el punto de vista lógico. Esto va muy de la mano con el tamaño y la complejidad de un sistema, aunque existen medidas para mitigar errores surgidos de manera indirecta de las características antes mencionadas, como un buen diseño en la arquitectura del sistema y guiar el desarrollo del sistema mediante pruebas en cada uno de sus componentes.

Un sistema de tiempo real debe poder interactuar de manera simultánea con uno o más componentes externos. Dependiendo del número de componentes involucrados y de su tiempo de respuesta, puede optarse por implementar un sistema con una arquitectura distribuida o bien hacer uso de un lenguaje de programación con altas prestaciones en el manejo eficiente de la memoria y comunicación entre procesos, o bien por integrar ambas soluciones. Algunos de ellos requerirán interactuar con el entorno a través de dispositivos digitales ya sea para efectuar alguna medición o bien para efectuar alguna acción. La interacción de un sistema de tiempo real con alguno de estos dispositivos, puede ser a través de la capa de red; usando un protocolo como UDP o TCP, o bien mediante otro tipo de interfaces de hardware que por

lo general no sufren de problemas inherentes a la comunicación a través de una red, como lo son la latencia, la congestión del canal de transmisión de datos e incluso pérdida de paquetes de datos.

Al seleccionar un lenguaje de programación para construir un sistema de tiempo real, que además sea concurrente, es bueno considerar cómo mitiga los siguientes aspectos:

- Las garantías que ofrece en cuanto a la consistencia de datos en áreas de memoria compartida entre uno o más procesos.
- La calidad de la comunicación entre procesos.
- Las estructuras de control nativas del lenguaje para tratar con problemas como la hambruna (*starvation*) y los abrazos mortales (*dead-locks*) entre procesos.

Actualmente, las grandes compañías de software destinan una gran cantidad de recursos de investigación en nuevos algoritmos, patrones y herramientas (como por ejemplo Map-Reduce[3], concepto desarrollado en Google) que mejoren el rendimiento de las arquitecturas distribuidas, así como en lenguajes de programación cada vez más eficientes. Un ejemplo de esto es un sistema de tiempo real suave como las redes sociales; con cada vez más personas conectadas a internet se han vuelto grandes generadoras de datos que demandan herramientas cada vez más avanzadas en el manejo y extracción de información muchas veces al momento en el que dicha información es generada por el usuario.

En el Apéndice D se presenta el lenguaje de programación Scala, el cual ha ganado popularidad en proyectos que manejan una gran cantidad de datos de manera concurrrente y en tiempo real.

1.5. El proceso de diseño y desarrollo de un sistema

Una metodología de desarrollo de software delinea formas de estructurar, planificar y controlar el proceso de desarrollo de un sistema de información. En esta sección se hablará brevemente de las metodologías ágiles.

1.5.1. Metodologías ágiles

Se denominan Metodologías ágiles a las buenas prácticas en el desarrollo de software basadas en fases incrementales e iterativas que añaden o mejoran funcionalidades a un sistema de información y donde la solución a las necesidades planteadas en los requerimientos evolucionan dentro de equipos donde la colaboración entre miembros con distintas habilidades

[3]Ver *Map-Reduce* en el Glosario.

y roles toma un papel crucial. Las metodologías ágiles tienen un manifiesto en común que incluye cuatro puntos:

1. Personas e interacciones tienen mayor prioridad que procesos y herramientas.
2. Es prioritario tener funcionalidad y satisfacción de necesidades frente a excesiva documentación en un producto de software.
3. Colaboración cercana con el cliente.
4. Adaptabilidad a cambios en lugar de seguir un plan definido.

Las fases de desarrollo que son comunes en casi todas las metodologías son:

- **Requerimientos:** Es el primer paso en el desarrollo de un sistema de software o bien en la modificación de uno existente. Este paso tiene como objeto conocer y tener clara tanto la problemática como las necesidades específicas de los usuarios que el sistema habrá de resolver.
- **Análisis y Diseño:** Consiste en analizar todos los requerimientos planteados y a partir de ello proponer una arquitectura del sistema escalable, basada en el análisis de los requerimientos. Al final de esta etapa suelen definirse las tecnologías complementarias que serán empleadas, como bases de datos y bibliotecas de software.
- **Implementación:** Es la fase en la que se traslada a código todo lo desarrollado en las etapas anteriores.

De todas las metodologías ágiles, Scrum [Scr14] es una de las metodologías de desarrollo de software más utilizadas hoy en día. Esta metodología promueve que el desarrollo de software se realice en pequeños componentes y que cada uno de ellos reutilice componentes previamente existentes. De esta forma, un equipo de desarrollo de software, empieza por los componentes más pequeños y va avanzando hacia los componentes más grandes, lo cual permite a los equipos responder de manera más eficiente a cambios que los clientes requieran, construyendo solo lo mínimo indispensable.

El nombre Scrum viene del *rugby*, en el que un *scrum* se refiere a la forma en la que se reinicia el juego después de una infracción menor.

El objetivo de Scrum es entregar la mayor cantidad posible de software de alta calidad dentro de 3 a 8 series de lapsos cortos conocidos como *sprints* que duran por lo general un mes.

Cada fase del ciclo de desarrollo (Requerimientos, Análisis, Diseño, Evolución y Entrega) se mapea a una serie de *sprints*, según sea necesario. Cada *sprint* opera en un número de elementos de trabajo llamados *backlog*. Como regla, una vez definidos los elementos de trabajo

que compondrán el *backlog* en un *sprint* no se podrán añadir más. Durante cada *sprint* se efectúan reuniones (*scrum meetings*) que le permiten al equipo compartir el conocimiento entre los miembros del equipo, y en ellas se da seguimiento a:

- Los elementos completados desde la última reunión (*Scrum Meeting*).
- Las tareas (*issues*) o bloques de funcionalidad que requieren ser resueltos.
- Tareas relacionadas que tiene sentido resolver hasta la siguiente reunión.

Muchas metodologías incluyen una fase de pruebas que por lo general se sitúan al final del desarrollo del software. A diferencia de ellas, en las metodologías ágiles la fase de pruebas guía el desarrollo del software; previo a escribir una sola línea de código se escriben escenarios de prueba (en código) que el software debe pasar una vez que son ejecutados; con esto se garantiza una mejor calidad en el software, pues se tienen pruebas automatizadas que pueden ejecutarse en cualquier etapa del desarrollo, verificando que las funcionalidades (acompaãdas de un conjunto de pruebas que las respalden) no se rompan al momento de añadir nuevas funcionalidades o modificar las existentes.

Capítulo 2. Desarrollo del modelo

A lo largo de este capítulo se detallarán los aspectos más importantes para la implementación de un sistema que genere rutas multi-modales óptimas usando datos en tiempo real y que tenga las características que se han mencionado en el capítulo anterior.

- Primero, se analizarán algunas especificaciones que definen formatos de intercambio de datos del transporte público utilizadas en otros países por parte de gobiernos y/o empresas.
- Después, se definirá la estructura de los datos que tendrá este sistema, junto con un análisis de los componentes organizados en torno a dos aspectos: aquellos que son dinámicos y aquellos que no lo son.
- Posteriormente, se analizarán algunos algoritmos que sirven para encontrar la ruta más corta en una gráfica y que se ajustan a las características del problema que se plantea resolver en este trabajo.
- Finalmente, se discutirán las opciones disponibles actualmente que facilitan el consumo y transferencia de datos entre aplicaciones.

Antes de abordar esos temas, se presentan en los siguientes *wireframes* [4] el aspecto visual enfocado en la funcionalidad para el usuario que tendrá el sistema. En la Figura 2.1 se muestra la interfaz gráfica para obtener un recorrido multi-modal a partir de que el usuario ingrese un punto origen y un punto destino para el recorrido y seleccione algunos parámetros para éste. En la Figura 2.2, se presenta la interfaz gráfica mediante la cual se consulta el recorrido propuesto por el sistema.

[4] Un *wireframe* es un boceto que representa una guía de los contenidos en una interfaz de usuario. Leer más en el Glosario.

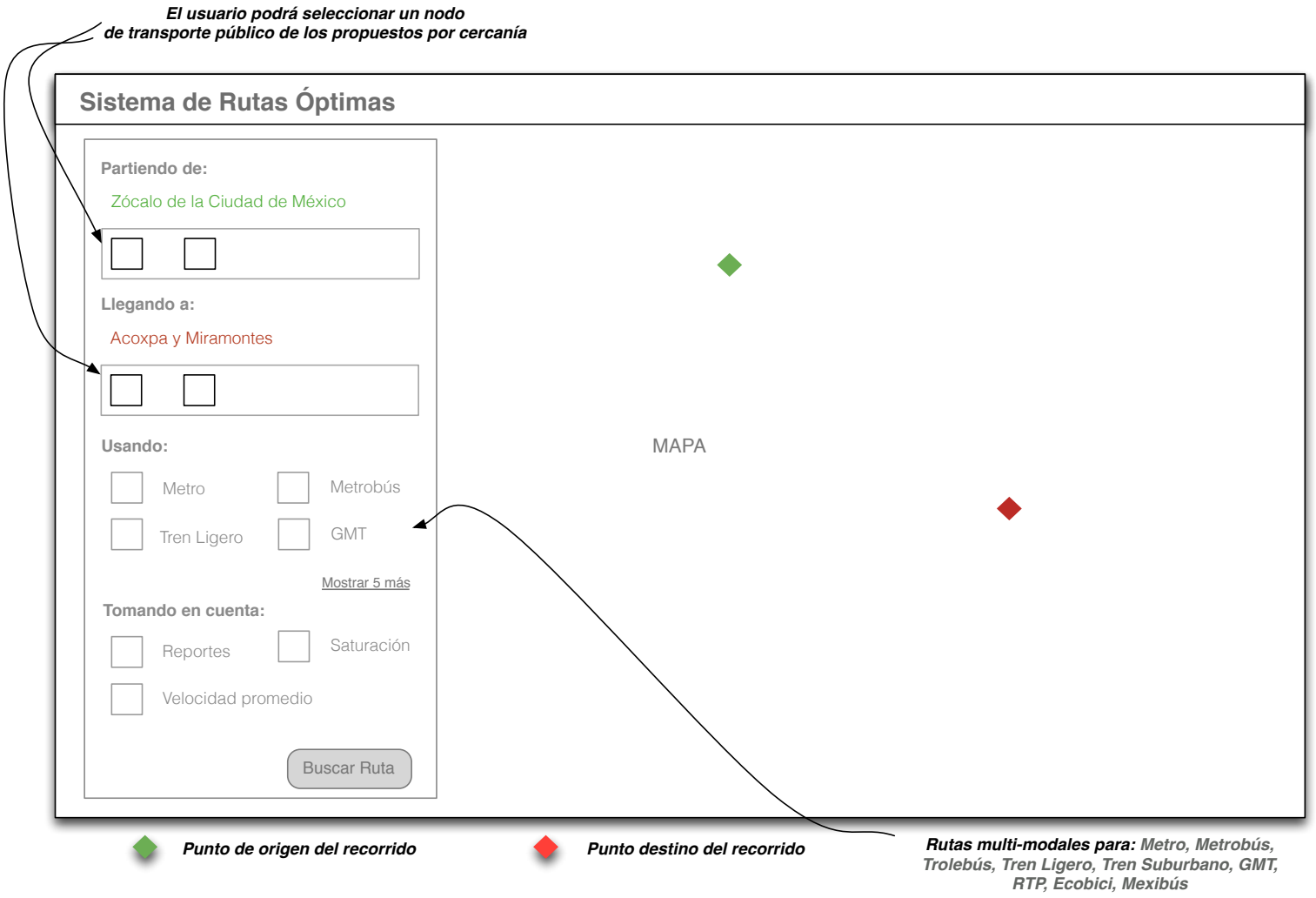

Figura 2.1: Captura de parámetros para generar ruta multi-modal

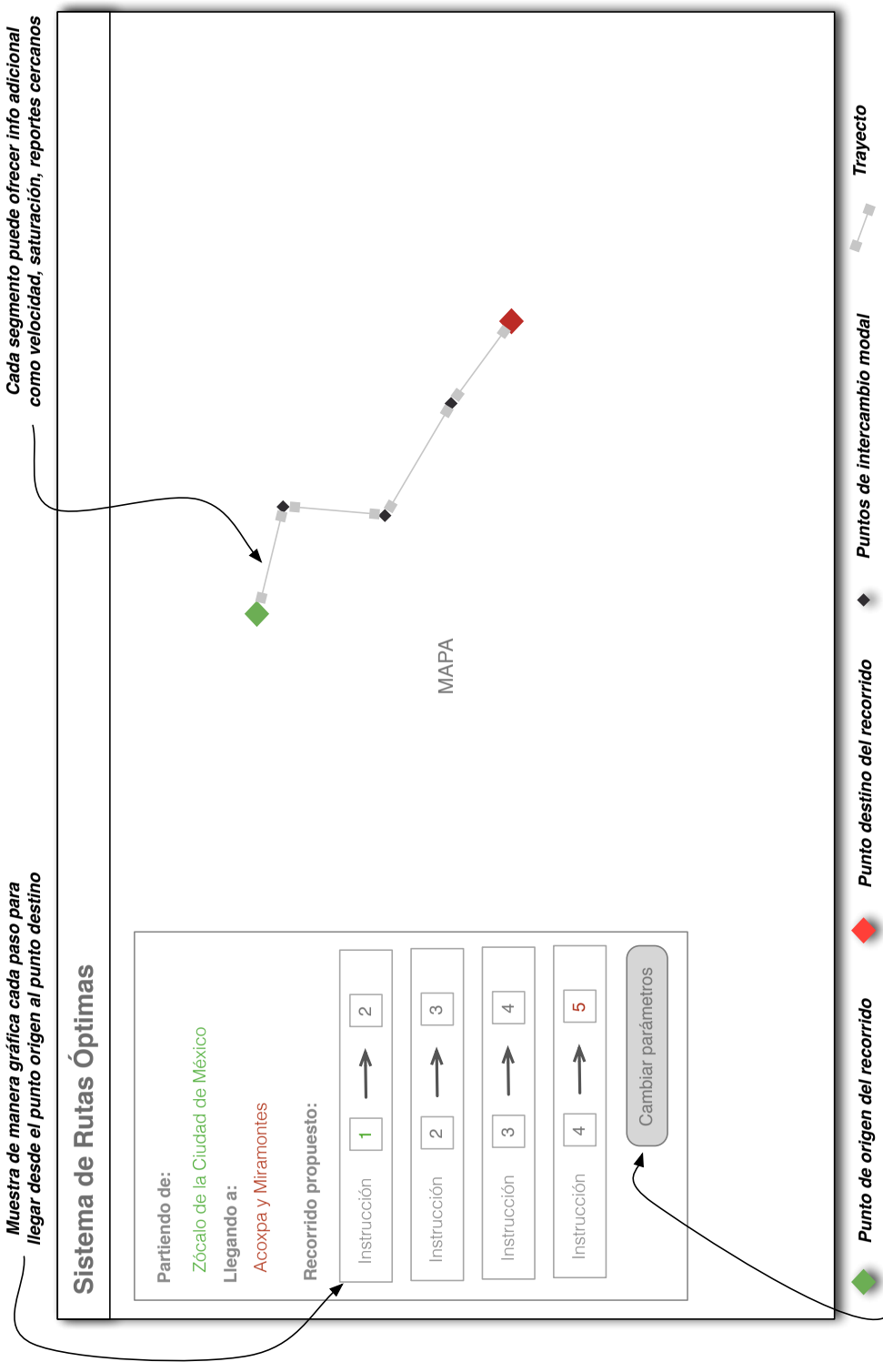

Figura 2.2: Ruta propuesta

2.1. Representación de datos del transporte público

Actualmente, existen dos formatos para representar información del transporte público; uno de ellos desarrollado por Google, mientras que el otro es un estándar europeo que data de 1992. En la región europea han trabajado universidades, empresas y gobiernos europeos en el desarrollo de dicho estándar, cuyo fin es representar e intercambiar información del transporte público. En esta sección se analizarán las ventajas y desventajas de ambos estándares a la luz de las características peculiares del transporte público en la Zona Metropolitana del Valle de México.

2.1.1. Especificación GTFS

La especificación GTFS[5] [GTF11], define un formato para la representación de horarios del transporte público e información geográfica asociada, que las agencias de transporte público publican con el fin de que cualquier programador pueda escribir aplicaciones usando tal información.

Los modelos presentes en la especificación, están agrupados en archivos de texto simple, independientes, con formato CSV[6]. Dicho formato representar valores de forma tabular y su ventaja principal es que puede ser leído y modificado fácilmente por casi cualquier persona usando un editor de texto simple; además, al ser todavía de uso popular, tiene amplio soporte en diferentes lenguajes de programación. Cada modelo tiene diferentes atributos que pueden ser de carácter opcional o requerido. Aquellos atributos requeridos por la especificación en cada modelo se presentarán en **negritas** y cada subtabla se referirá a un modelo. La descripción detallada de cada atributo puede encontrarse en el sitio de GTFS.

Agencies.txt					
agency_id	**agency_name**	**agency_url**	**agency_timezone**	agency_lang	agency_phone
Contiene registros de agencias públicas o privadas que tienen a su cargo una o más líneas de transporte público.					

Cuadro 2.1: Tabla de agencias de transporte

[5] General Transit Feed Specification
[6] *CSV*: Valores separados por comas (*Comma separated Values* en inglés)

STOPS.txt						
stop_id	stop_code	**stop_name**	stop_desc	**stop_lat**	**stop_lon**	**stop_lon**
zone_id	stop_url	location_type	parent_station			
Representa puntos definidos de manera única donde los pasajeros abordan o descienden de algún vehículo						

Cuadro 2.2: Tabla de paradas

ROUTES.txt					
route_id	agency_id	**route_short_name**	**route_long_name**	route_desc	**route_type**
route_url	route_color	route_text_color			
Representa una ruta que tiene dos estaciones terminales y dentro de la cual puede haber uno o más recorridos.					

Cuadro 2.3: Tabla de rutas

TRIPS.txt					
route_id	**service_id**	**trip_id**	trip_headsign	trip_short_name	direction_id
block_id	shape_id				
Representa el recorrido que puede elegir un pasajero dentro de cada ruta que ofrece una agencia de transporte público. Por ejemplo, en el caso del Metrobús para la Línea 1 existe un recorrido entre las estaciones El Caminero y Buenavista y otro entre las estaciones Insurgentes y Doctor Gálvez.					

Cuadro 2.4: Tabla de recorridos

StopTimes.txt					
trip_id	**arrival_time**	**departure_time**	**stop_id**	**stop_sequence**	stop_headsign
pickup_type	drop_off_type	shape_dist_traveled			
Lista los horarios de llegada y salida de los vehículos para cada estación o parada definida en Stops.txt.					

Cuadro 2.5: Tabla de tiempos de llegada y salida

Calendar.txt					
service_id	**monday**	**tuesday**	**wednesday**	**thursday**	**friday**
saturday	**sunday**	**start_date**	**end_date**		
Define los días de la semana y horas en las que un servicio está disponible a los usuarios.					

Cuadro 2.6: Tabla de horarios de servicio

CalendarDates.txt		
service_id	**date**	**exception_type**
Define excepciones para los servicios listados en Calendar.txt		

Cuadro 2.7: Tabla de excepciones a la tabla de horarios de servicio

FareAttributes.txt					
fare_id	**price**	**currency_type**	**payment_method**	**transfers**	transfer_duration
Contiene información sobre las tarifas de los servicios que ofrece una agencia de transporte.					

Cuadro 2.8: Tabla de tarifas del servicio

FareRules.txt				
fare_id	route_id	origin_id	destination_id	contains_id
Define un conjunto de reglas para aplicar la información contenida en FareAttributes.txt a las rutas de una agencia de transporte.				

Cuadro 2.9: Tabla de reglas en la aplicación de las tarifas del servicio

Shapes.txt				
shape_id	**shape_pt_lat**	**shape_pt_lon**	**shape_pt_sequence**	shape_dist_traveled
Define reglas para dibujar líneas en el mapa que representan las rutas de una agencia de transporte.				

Cuadro 2.10: Tabla de trazos de rutas

Frequencies.txt			
trip_id	**start_time**	**end_time**	**headway_secs**
Representa horarios que no tienen una lista fija de horas de parada, a diferencia de los contenidos en StopTimes.txt. En esta tabla se especifica el horario de inicio y fin de servicio para cada recorrido listado en Trips.txt y la frecuencia de paso de vehículos			

Cuadro 2.11: Tabla de frecuencias del servicio

Transfers.txt			
from_stop_id	**to_stop_id**	**transfer_type**	min_transfer_time
En esta tabla se definen los puntos de transbordo entre dos paradas de rutas distintas.			

Cuadro 2.12: Tabla de transbordos

El proyecto Google Transit de Google tiene como objetivo mantener y actualizar la especificación GTFS para que se mantenga como un formato simple e interoperable que permita

a agencias de transporte público y programadores trabajar cada uno en su área de conocimiento con información relacionada al transporte público. De esta forma, las agencias de transporte público solo deben preocuparse por describir como opera su servicio, mientras que los programadores tienen acceso a los datos de casi cualquier agencia de transporte público para así, construir aplicaciones que informen a los pasajeros acerca del estado de la red de transporte público.

CSV, el formato que esta especificación utiliza para representar la información tiene algunas ventajas:

- Es muy sencillo de manipular por cualquier persona, ya sea mediante un editor de texto o mediante cualquier programa de hoja de cálculo.
- Es muy ligero, pues la estructura de la información es muy simple y no requiere de especificar más que una línea de metadatos.

Pero también tiene algunas desventajas:

- A diferencia de XML, es un formato poco confiable en asegurar la integridad de los datos.
- No incluye metadatos que indiquen al programa que genera un CSV o al que lo lee el tipo de codificación del texto, lo cual puede generar problemas en idiomas que usan caracteres especiales.

2.1.2. Modelo TransModel

Es actualmente el estándar predeterminado para la representación y comunicación de información relacionada con las redes de transporte público en la mayoría de los países europeos. Al igual que GTFS, Transmodel es un modelo abstracto para la representación de información relacionada al transporte público; sin embargo, y a diferencia del primero, Transmodel considera conjuntos más amplios de información que puede ser abstraída y representada de manera consistente. Esta especificación data ya de algunos años (su primera versión es de 1993), pero ha sido actualizada regularmente para adaptarse a los cambios en los sistemas de transporte público del continente europeo. Desarrollada al interior de un amplio rango de proyectos europeos de diversos programas (Drive I, Drive II, TAP) con apoyo de la Comisión Europea y la participación de instituciones públicas y privadas de varios países, en particular la Dirección de Transportes Terrestres de Francia.

La especificación GTFS fue desarrollada [GTF14] con el objetivo de tener un estándar para representar información útil a los usuarios del transporte público; Transmodel además de cumplir con esa tarea, también es capaz de representar información que puede considerarse de uso interno para las agencias de transporte, además de haber sido conceptualizado desde

el inicio para que sistemas que sean implementados bajo ese estándar no tengan problema en comunicarse entre sí en el intercambio de datos, ni tampoco al momento de añadir funcionalidades adicionales al sistema. La clave de los beneficios anteriores es el buen diseño de la arquitectura de datos, la cual está formada por componentes modularizados que exponen servicios mediante interfaces bien definidas. Sin embargo, Transmodel a diferencia de GTFS no promueve la apertura de datos y el libre acceso a ellos a través de plataformas innovadoras y tecnologías como Google Transit lo hace. En el cuadro 2.13 se presenta una tabla comparativa de ambas especificaciones.

Aspecto	**Transmodel**	**GTFS**
Operando	Desde 1993	Desde 2005
Alcance	Solo en la región europea	773 agencias de transporte
Interfaces	Señalización electrónica en estaciones y Web	Web y Móvil
Abierto	Especificación si, pero no los datos	Especificación y datos son abiertos
Complejidad	Alta	Baja
Enfoque	Diseñado pensando en las agencias	Diseñado pensando en los pasajeros
Documentación	Muy poca y antigua	Mucha y con muchos ejemplos
Usado por	Gobiernos, agencias y universidades	Google, Agencias y programadores

Cuadro 2.13: Tabla comparativa entre Transmodel y GTFS

La especificación define como elementos base: 1.- la descripción de la red, 2.- el manejo de versiones y 3.- validaciones y capas sobre los cuales se construyen otros más complejos que igualmente caen dentro del modelo de referencia de datos. La descripción topológica de la red se construye con tramos y puntos que representan distintos elementos relacionados al fenómeno a modelar. Así mismo, una red de rutas es la infraestructura fundamental sobre la cual se modelan los servicios que una o varias agencias ofrecen. Adicionalmente, existen vistas funcionales de la red, descritas como capas, siendo un ejemplo de esto último la posibilidad de conjuntar capas de servicios cercanos a estaciones y capas de la red de uno o más medios de transporte. Finalmente, este estándar propone un formato para describir datos geográficos asociados a la red: Geographical Data Files (GDF)[GDF14]. Uno de los propósitos de este formato es facilitar el intercambio de información geográfica entre agencias de transporte y otros prestadores de servicios relacionados. Se ha definido entonces una interface en el modelo de referencia de datos entre el formato GDF y el módulo encargado de representar la topología de la red. El manejo de versiones de los elementos geográficos es un componente de igual importancia en la especificación, pues hace posible dar seguimiento a cambios en los servicios que se ofrecen de manera regular. Es posible, por otro lado, representar relaciones entre diferentes niveles de jerarquías de grupos de datos mediante validación de condiciones impuestas en éstas.

La última versión del documento que contiene el modelo de referencia de datos para Transmodel[Tra14] toma en cuenta, además de la operación multi-modal del transporte público y los casos en los que están presentes múltiples operadores, a seis grandes grupos los cuales

contienen aspectos del transporte público sujetos a ser modelados y que llama "necesidades informativas":

- Planeación y administración de los servicios.
- Disponibilidad de personal (conductores).
- Administración de operaciones y control.
- Información a pasajeros.
- Recolección de tarifas.
- Administración de la información.

Este trabajo, como se mencionó anteriormente, se enfocará en el aspecto de "información a pasajeros". A continuación se presentará el conjunto de modelos considerados para la propuesta de este sistema.

2.1.3. Especificación mixta de GTFS y Transmodel

Si bien Transmodel y GTFS abordan el problema de la representación de los datos del transporte público desde diferentes enfoques, resulta interesante contrastar los beneficios que cada uno ofrece y sobre todo la experiencia que ambas especificaciones han acumulado durante los años. Por ejemplo, Transmodel tiene mucha más experiencia modelando los diversos tipos de transporte público que existen en el continente europeo, el cual es uno de los más modernos y eficientes en el mundo. Sin embargo, GTFS es una especificación que promueve bancos de datos abiertos para que cualquier persona pueda construir aplicaciones para el transporte público. El primero se enfoca a resolver las necesidades tecnológicas de las agencias para que estas puedan ofrecer un servicio óptimo y adicionalmente proveer de información a los pasajeros a través de canales que la misma agencia desarrolla y mantiene; mientras que el último está pensado para que las agencias de transporte público no tengan que preocuparse por desarrollar aplicaciones web y móviles para sus pasajeros y en cambio esta tarea la hagan empresas o personas interesadas en el tema.

Para este trabajo, se partirá de un modelo híbrido propuesto por Kizoom[Kiz08] que unifica GTFS y Transmodel en el conjunto de información para pasajeros de este último. La ventaja principal es que se obtiene compatibilidad con aplicaciones basadas en GTFS como por agencias que hayan adoptado el estándar Transmodel, teniendo además la posibilidad de incorporar modelos ya definidos en Transmodel o nuevos modelos que hagan uso de ellos.

2.2. Alcances del modelo

En la figura 2.3 se muestran los componentes y subcomponentes más importantes considerados en el sistema que propone este trabajo. En él aparecen, dentro de un bloque amarillo, los componentes que conforman la aplicación web (*Backend*) que centraliza los datos del sistema y se proponen dos *APIs*[7], uno público y otro privado, los cuales distribuyen datos a otros sistemas o bien los integran desde fuentes externas como sensores e incluso otros sistemas. Los componentes principales del sistema que se discutirán más adelante son:

- **APIs públicos y privados**, y las tecnologías actuales para la representación y consulta de información a través de servicios web. También se discutirá sobre el papel que juega el buen diseño de un *API* en un sistema.

- **Base de datos para los modelos de la especificación**, si bien se hará uso de la especificación GTFS compatible con Transmodel propuesta por Kizoom[Kiz08], sólo serán modeladas las características topológicas de la red de transporte público junto con los modelos que las definen y aquellos de los que dependen, según dicha especificación. Aquí se seleccionará alguna de las tecnologías disponibles para persistencia de datos, que de mejor manera represente los datos que se pretenden almacenar.

- **Almacén de datos dinámicos**, con una arquitectura genérica de almacenamiento de datos dinámicos que permita acceder a ellos de la forma más eficiente posible, para su posterior integración con otros datos del sistema; principalmente en la fase de generación de rutas óptimas multi-modales.

- **Generador de rutas óptimas**, mediante el uso de algoritmos de enrutamiento[8] como Dijkstra o A*. Si se decide representar la topología de la red mediante una base de datos no tradicional[9] como AllegroGraph [All11] o Neo4J [Neo11a], entonces se tendrá a disposición un conjunto amplio de algoritmos de teoría de gráficas que operarán al nivel de la base de datos. Otra solución deberá ofrecer igualmente un conjunto de algoritmos de enrutamiento que puedan operar eficientemente sobre los datos almacenados.

- **Interfaz de mantenimiento**, con algunas especificaciones que deberá cumplir un sistema de administración que facilite la modificación de los datos asociados a la topología de la red de transporte público, así como hacer posible su captura desde fuentes como GTFS o XML.

Este diagrama será de uso recurrente para el análisis de cada uno de los componentes que se proponen para este sistema.

[7] Ver *Application Programming Interface* en el Glosario.
[8] Algoritmos Descritos en el Apéndice B
[9] Algoritmos Descritos en el Apéndice C

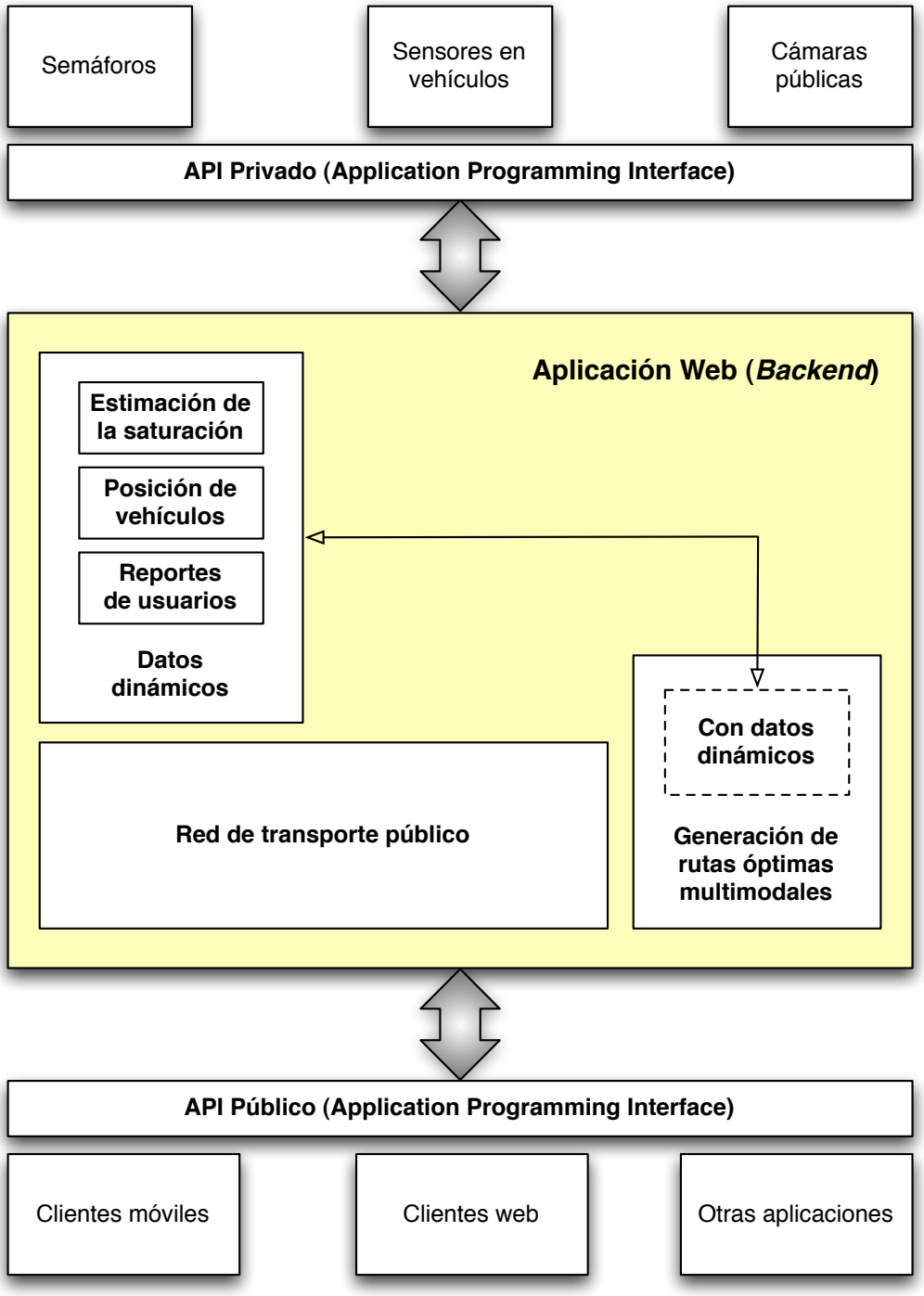

Figura 2.3: Componentes propuestos del sistema

La figura 2.4 presenta un diagrama con la especificación de Kizoom, el cual es el punto de partida para construir la propuesta del sistema del que habla este documento. En este diagrama cada modelo presenta dos nombres, el que está en negro es el nombre dado por Transmodel, mientras que el más claro corresponde a su equivalente en GTFS.

El diagrama de la figura 2.5 presenta una especificación ligeramente diferente a la de Kizoom. Las diferencias entre ambas radican principalmente en la simplificación de los modelos y las relaciones entre éstos, se adopta un nombre para cada modelo y cambian los tipos enumerados a tipos que mejor describen la forma de operar de los sistemas de transporte público de nuestro país. Con base en este último diagrama se plantea el desarrollo de este trabajo.

El modelo más simple es el de Agencies y representa a una agencia de transporte como Metrobús o STC Metro. Una agencia por lo general tiene al menos una línea de transporte con un nombre y detalles adicionales como el color de línea, el tipo de vehículo que emplea para trasladar pasajeros, entre otros. Una línea solo pertenece a una agencia de transporte y todos sus detalles se describen en el modelo Lines. El modelo para VehicleJourneys representa un recorrido con una dirección fija a lo largo de una línea con paradas previamente definidas, generalmente fijas. Este modelo está relacionado con una o más estaciones donde se da el ascenso y descenso de pasajeros. Se plantean dos modelos para representar estaciones o paradas. PhysicalStations representa estaciones físicas por donde pasan uno o más recorridos (VehicleJourneys) y que ofrecen servicios adicionales a los usuarios (definidos en el modelo StationServices). Un recorrido tiene una o más estaciones lógicas LogicalStations que definen un orden de estaciones a lo largo del recorrido. Una estación lógica puede tener tiempos de salida y llegada a lo largo de un día.

El modelo ConnectionLinks representa correspondencias entre recorridos a través de estaciones lógicas. Finalmente, VehicleJourneyFrequencies representa la frecuencia con la que vehículos de transporte público inician recorridos y el tiempo de separación entre ellos considerando diferentes días especiales.

Los modelos hasta ahora descritos representan los aspectos estáticos de una red de transporte público, mientras que los aspectos dinámicos son Vehicles, que representa a un vehículo de pasajeros cualquiera asignado a una línea, asociado a muchos instantes, tantos como la frecuencia de actualización de la posición del vehículo. El modelo Instants contiene las coordenadas y la velocidad de un vehículo.

2.3. Análisis de herramientas

Una vez identificados los componentes principales del sistema planteado, es tiempo de hacer un análisis de las herramientas a utilizar y de por qué una u otra es la más adecuada para cada componente.

- Un manejador de base de datos moderno que esté optimizado para trabajar con datos

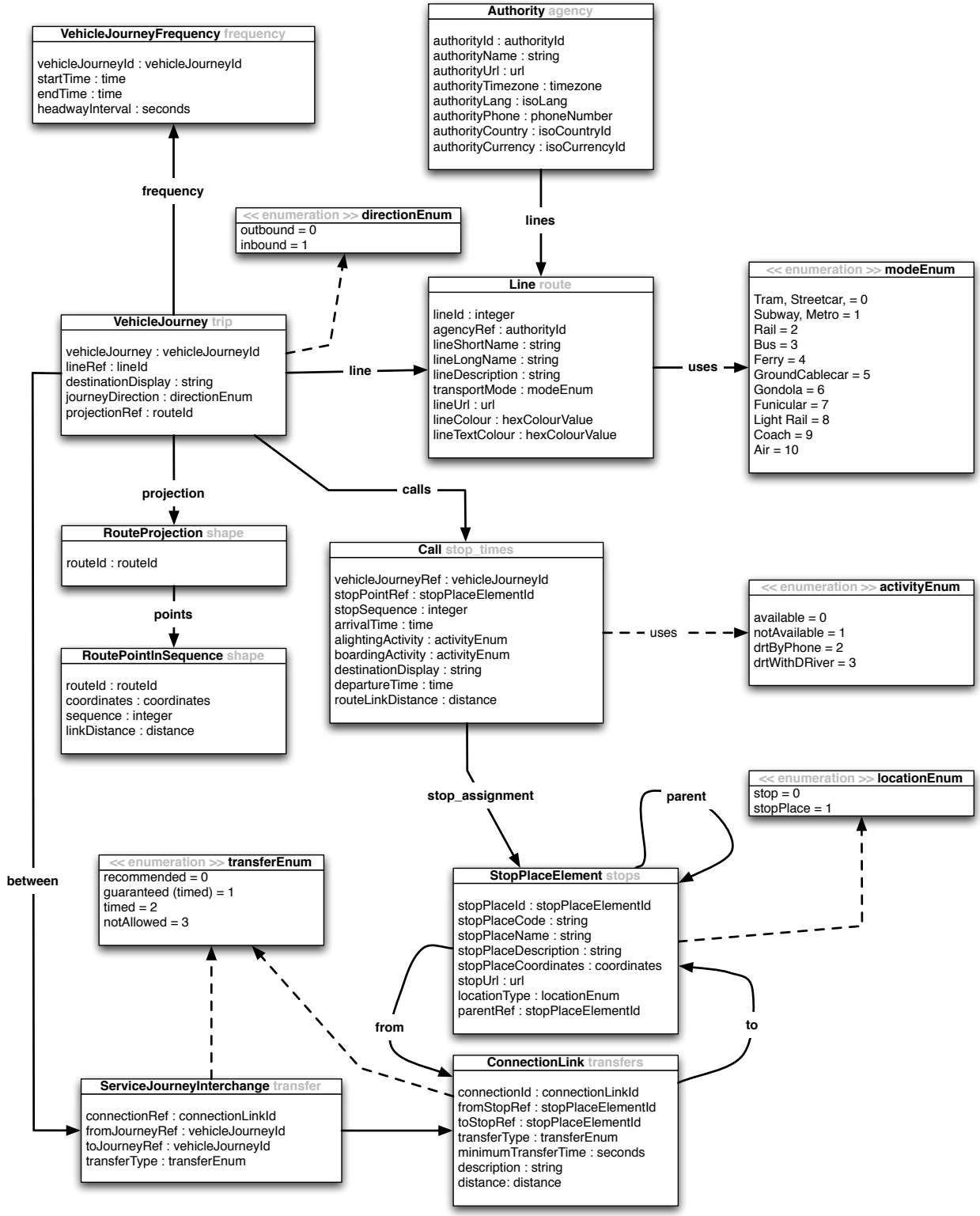

Figura 2.4: Diagrama de clases con la especificación mixta de GTFS y Transmodel

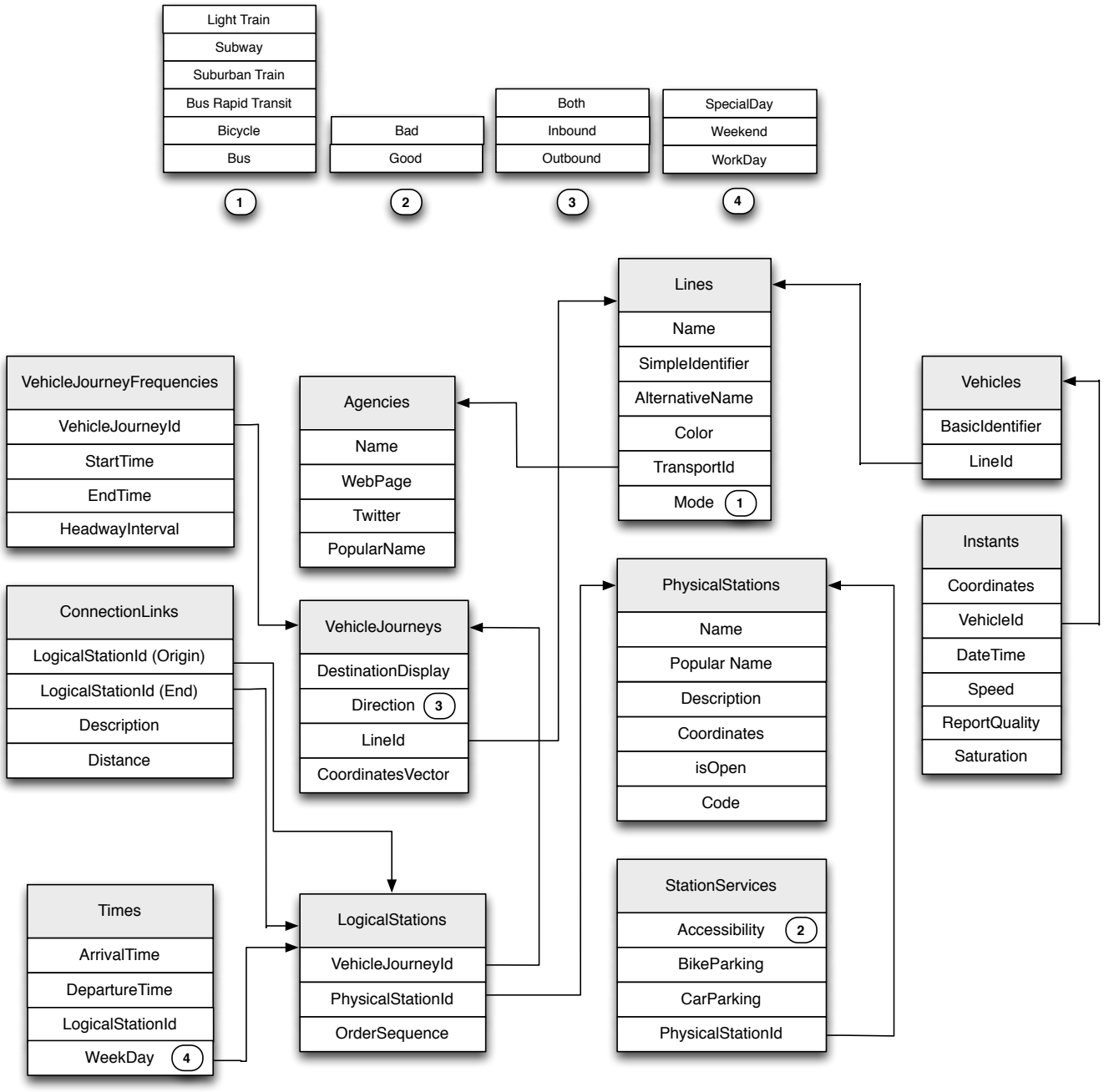

Figura 2.5: Diagrama de clases de los componentes seleccionados de la especificación mixta de GTFS y Transmodel

geoespaciales.

- Un lenguaje de programación adecuado para trabajar con datos concurrentes y en tiempo real, que pueda interoperar con la biblioteca de bases de datos que se seleccione y que además ofrezca un *framework* de desarrollo web moderno.

- El *framework*[10] de desarrollo web que mejor se adecué a la arquitectura del *backend* y que facilite la implementación de APIs.

2.3.1. Elección de un manejador de bases de datos

En el cuadro 2.14 se aprecian las ventajas y desventajas de dos manejadores de bases de datos que se pueden utilizar para representar la topología de la red de transporte. El primero está basado en el estándar SQL, mientras que el segundo es un manejador de bases de datos NoSQL. En esta tabla se consideran los siguientes aspectos:

- Si tiene soporte o no para hacer *Dumps* (volcados) de los datos de forma que se pueda reconstruir la base de datos desde el *dump*. Es importante que el volcado esté en un formato estándar que permita cambiar de manejador de base de datos en caso de ser necesario.

- Si tiene soporte o no para hacer *Restores* (recuperaciones) del estado previo de una base de datos a partir de un archivo de copia de seguridad o *dump*.

- Si cuenta con *Bindings* (bibliotecas para conectarse de forma eficiente a un lenguaje de programación) para el lenguaje de programación seleccionado .

- Si cuenta con soporte geoespacial.

- Si incluye funciones para calcular rutas óptimas entre dos puntos, integradas directamente en el manejador de forma nativa.

[10] *Framework*: Es un conjunto de bibliotecas de software que le brinda al programador un punto de inicio sólido en la construcción de software especializado. En los apéndices se explica qué es un framework de desarrollo web.

Característica	PostgreSQL	Neo4j
Soporte para *dumps*	Solamente a SQL.	Solamente a GraphML, CSV y Geoff.
Soporte para *restores*	Solamente desde SQL.	Desde GraphML, CSV y Geoff.
Soporte para *bindings*	Java, Scala, C, C++, Python, Ruby, Go, Haskell y Erlang, entre otros.	Con lenguajes compatibles con la máquina virtual de Java o mediante HTTP.
Soporte geoespacial	Mediante su extensión PostGis	Integrado en el mismo manejador
Algoritmos de enrutamiento	Mediante pgRouting: Dijkstra, A*, Dijkstra con múltiples destinos	Solamente Dijkstra, aunque los datos en este manejador están ordenados a modo de gráfica.

Cuadro 2.14: Tabla de rutas

De los aspectos listados en la tabla anterior, tanto el soporte geoespacial como la posibilidad de contar con una estructura que represente de manera eficiente la red de transporte público son aspectos de gran importancia. Sin embargo, ninguna de las dos herramientas ofrece una solución que cubra tales aspectos actualmente. Por un lado, con Neo4J se tiene solucionado de manera inmediata el problema de la representación de la red usando una gráfica y con ello la generación de rutas óptimas entre dos puntos, pero un problema de esta tecnología es que la representación de sus datos no está apegada a un estándar equivalente a SQL, presente en las bases de datos relacionales. Por otro lado, si se necesita realizar búsquedas con condiciones, tiene que adaptarse un algoritmo de búsqueda en gráficas (BFS o DFS) que vaya realizando podas al conjunto de resultados a lo largo de su ejecución, aunque es posible indexar nodos y aristas en representaciones tabulares de los datos en una base de datos de Neo4j para un acceso optimizado.

Un aspecto adicional a tomar en cuenta, es que Neo4j es una tecnología aún por madurar y no existen servicios reconocidos en la web que la estén usando actualmente, como sí sucede, por ejemplo, con MongoDB, otro manejador de bases de datos NoSQL.

Con lo anterior en mene, se considera que usar el manejador de bases de datos PostgreSQL con su extensión PostGIS es una opción mucho más robusta por lo siguiente:

1. Es un proyecto con una tecnología madura y probada a lo largo de 20 años desde su creación
2. Tiene *bindings* con java o cualquier lenguaje que pueda interoperar con la máquina virtual de java.
3. Se tienen todas las ventajas existentes en un RDBMS y es posible usar pgRouting o plantear una solución personalizada en la capa del servidor de la aplicación.

Para complementar este análisis, en el Apéndice C se presentan algunos conceptos que tienen que ver con las garantías que ofrecen los manejadores de bases de datos en cuanto a la integridad de los datos que almacenan y la eficiencia tanto en la escritura como en la lectura

de estos. Además se habla sobre manejadores de bases de datos NoSQL que prometen ser más eficientes que aquellos basados en el estándar SQL y que son una alternativa confiable para algunos tipos de sistemas.

En la siguiente sección se detalla la arquitectura que tendrá el sistema propuesto utilizando el manejador PostgreSQL y considerando los puntos mencionados anteriormente.

2.3.2. Elección de un lenguaje de programación y un *framework* web y propuesta de arquitectura para el *backend*

Una vez decidido el mecanismo de persistencia de datos del sistema y la arquitectura de sus datos, lo siguiente es tener claro el funcionamiento de las demás capas que lo componen. En el caso de la propuesta de sistema que se presenta en este trabajo, muchas de las capas que lo componen resultan comunes a las de muchos otros sistemas que están basados en la web. Particularmente, el componente web de este sistema requerirá:

- Comunicarse con el manejador de bases de datos y traduce tuplas en la base de datos a objetos en el lenguaje de programación que usa la aplicación web.
- Interoperar con un servidor de aplicaciones moderno como Apache, Nginx o Tomcat.
- Acceder a bibliotecas modernas que permitan:
  - Construir APIs
  - Construir y desplegar HTML dinámico

Con el fin de entender y aprender como funciona el lenguaje de programación Scala, se ha optado por implementar el componente web del sistema que se propone en este documento en tal lenguaje de programación. Las razones principales por las cuales se ha optado por Scala son:

- Es interoperable con la máquina virtual de java
- Utilizado por los dos servicios que atienden el mayor número de operaciones concurrentes y/o geoespaciales: Twitter [Twi14] y Foursquare [Fou14].
- Es un lenguaje orientado a objetos con un enfoque funcional.

En el Apéndice D, se presenta una breve introducción al lenguaje con algunos ejemplos de como su sintáxis y semántica aprovechan su carácter funcional. También se habla de Akka, su biblioteca basada en el modelo de actores para trabajar con procesos concurrentes, la cual es uno de los puntos más fuertes que tiene el lenguaje.

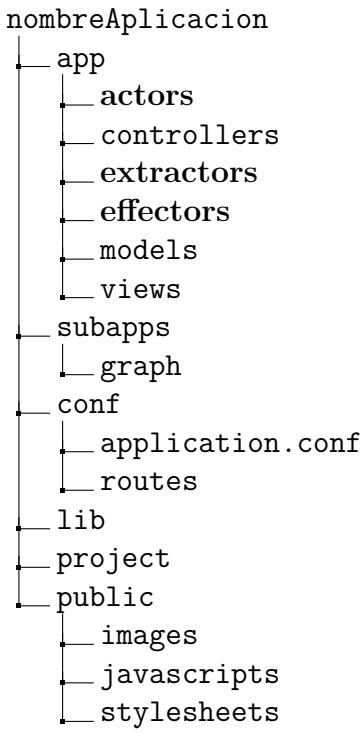

Figura 2.6: Directorios generados por Play

La propuesta que se presenta a continuación considera una disposición de directorios similar a la generada por el *framework* de desarrollo web Play basado en Scala que se muestra en la Figura 2.6, salvo quizás por los directorios marcados en negritas, los cuales guardan funciones muy particulares al sistema que se está presentando y que más adelante se detallan. En el Apéndice E se presentan detalles adicionales sobre aplicaciones web.

En las siguientes secciones se retomarán los modelos presentados en la figura 2.5 para dar mayores detalles relacionados a su implementación, abarcando más allá del mecanismo de persistencia que ya fué discutido.

2.3.3. Representación de la red

Se decidió en secciones anteriores que la red de transporte público se almacenaría en una base de datos SQL (usando el manejador de bases de datos PostgreSQL) con soporte geoespacial. Sin embargo no se ha explicado aún cómo aplicar sobre una estructura basada en tablas, tuplas y relaciones entre tuplas, algoritmos para encontrar caminos más cortos entre dos puntos. Existen tres posibilidades: usar PL/PgSQL[11], pgRouting o usar una biblioteca de Java que permita representar una gráfica en memoria y que tenga algoritmos de gráficas que solucionen el problema de encontrar caminos más cortos entre dos puntos. A continuación se

[11] Ver PL/PgSQL en el Glosario.

presentan las desventajas para cada una de las opciones planteadas:

1. **Usar PL/PgSQL:**
   - Difícil de mantener.
   - Realizar operaciones que cambian el estado es costoso, pero si dichas operaciones son recurrentes, entonces pueden tener un impacto negativo en el rendimiento del sistema, esto empeora si hay resultados que el sistema debe generar a partir de dichas operaciones de manera casi instantánea.
   - Mayor dependencia con el manejador de bases de datos PostgreSQL.
   - Implementación de algoritmos específicos según se requiera.
2. **Usar pgRouting:**
   - Requiere crear una topología de la gráfica a partir de las tablas, lo cual no es inmediato.
   - Complejidad de uso.
3. **Usar una biblioteca de gráficas:**
   - Puede ser costoso mantener en memoria una representación de la red de transporte público.
   - Creación de tablas para cada meta-dato asociado a las aristas y nodos de la gráfica.

Considerando las desventajas de las tres opciones propuestas, se puede concluye que una biblioteca de gráficas es una mejor opción por lo siguiente:

- La representación en memoría es mucho más eficiente y versátil.
- Resulta mucho más fácil de implementar y mantener.
- Dependiendo de la biblioteca que se use, se puede disponer de un buen número de algoritmos para operar sobre la gráfica.
- Una vez construida la gráfica, ésta puede ser serializada usando GraphML, un formato estándar para representar gráficas.
- La implementación puede escribirse en Java o en Scala, ya que ambos compilan código a *bytecode*.
- Se puede integrar fácilmente con otros módulos de la aplicación.

Se propone usar la biblioteca JUNG [JUN12] para construir la representación de la red como una gráfica en memoria donde los nodos y aristas estén asociados a uno o varios modelos en la base de datos según su tipo. Por ejemplo, tuplas en LogicalStations estarían modeladas por un tipo de nodo, mientras que cada tupla en ConnectionLinks estaría modelada por un tipo de arista. La implementación deberá asegurar que cada elemento de la gráfica corresponda a una tupla de la tabla correspondiente, según el tipo del elemento.

En la figura 2.7 se muestra un esquema en el que se describen los campos que contiene una arista y un nodo. Por un lado, una arista contiene un diccionario o tabla *hash* con capacidad para almacenar una lista de pares de llaves y valores que representan atributos de la arísta. También incluye un arreglo que puede almacenar palabras clave y un objeto de referencia. Un nodo, por su parte almacena hasta tres campos: su tipo asociado a una tupla en una tabla de la base de datos, su identificador y adicionalmente, un objeto de referencia. Es importante hacer notar, que JUNG permite a cualquier POJO[12] ser un nodo o una arista.

Con lo anterior definido, se contaría con una gráfica sobre la cual ejecutar algoritmos. JUNG soporta muchos tipos de gráficas: dirigidas, no dirigidas, árboles, bosques, multi-gráficas e hiper-gráficas. En cuanto a algoritmos específicos para encontrar rutas más cortas soporta Dijkstra y Prim (para construir árboles y bosques de peso mínimo). La biblioteca ofrece dos interfaces: "Distance", que define métodos para algoritmos que etiquetan distancias en los nodos, mientras que "ShortestPath"define métodos para algoritmos que devuelven la lista de nodos que conforman la ruta más corta. Todo el código relacionado con lo que se presentó en estos párrafos quedaría en **subapps/graph**.

La siguiente definición establece una formalización básica para asociar aspectos generales de una red de transporte público con conceptos de teoría de gráficas, la cual será de utilidad de ahora en adelante.

Sea $G = (V, E)$ la digráfica asociada a una red de transporte público R, tal que las aristas $e \in E$, tienen un costo asociado a la distancia entre los nodos que unen. La dirección de una arista representa la dirección en la que los pasajeros pueden moverse entre estaciones asociadas a cualesquiera dos nodos en R.

Transporte individual: bicicletas compartidas

En años recientes, los sistemas de préstamo de bicicletas públicas alrededor del mundo se han vuelto populares. Dichos sistemas funcionan mediante estaciones distribuidas en puntos cercanos al transporte público en las ciudades en donde los usuarios inician (o terminan) un recorrido usando una bicicleta. De manera intuitiva, este tipo de transporte puede verse como una subred dentro de la red del transporte público de la ciudad.

Definición Se define como $G_i \subset G$ a la subgráfica inducida por aristas asociada a la sub-red de transporte formada por estaciones de préstamo de bicicletas (bici-estaciones) dentro de

[12] Ver POJO en el Glosario.

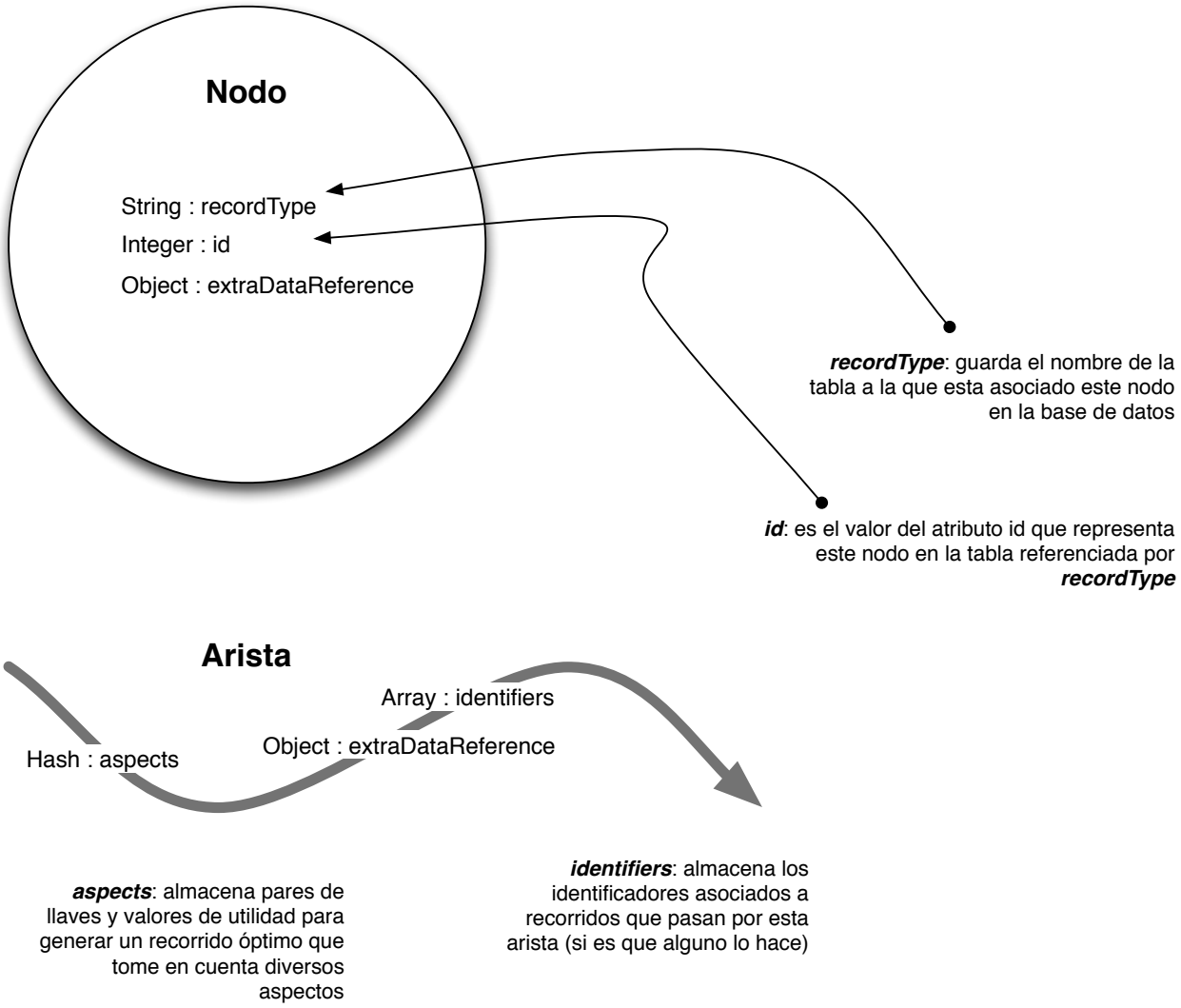

Figura 2.7: Atributos de Aristas y Nodos

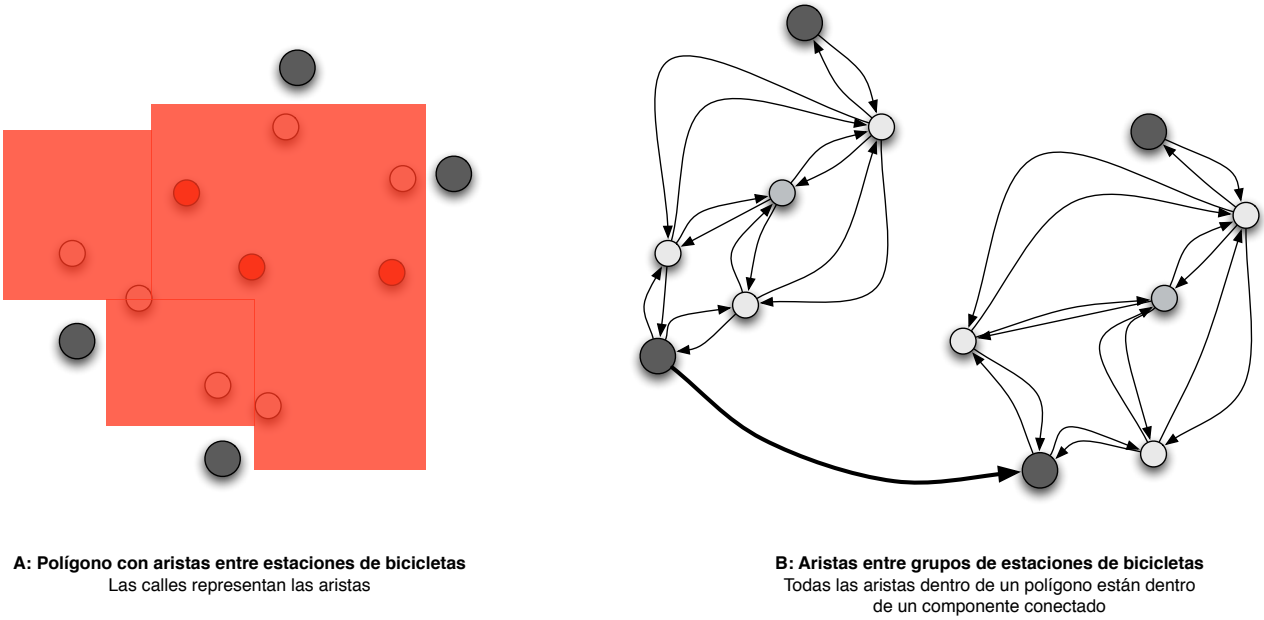

Figura 2.8: Sistemas de transporte individual: Préstamo de bicicletas

un polígono geográfico i, teniendo que:

$$\forall(v, e) \in G_i \subseteq G$$

donde cada e es una arista que representa una forma de ir desde un nodo v a otro asociando cualesquiera dos bici-estaciones. Si $G_i = G$ entonces la red de transporte público está formada únicamente por bici-estaciones dentro del polígono i.

En la figura 2.8 se presentan dos diagramas (**A** y **B**) mostrando distintas formas de integrar esa subred a la representación de la red de transporte público ya planteada.

En ambos diagramas los nodos más grandes representan estaciones de transporte público únicamente, mientras que los pequeños representan bici-estaciones. En el diagrama **A** las aristas de la gráfica están representadas por la red de calles dentro del polígono marcado, mientras que en el diagrama **B** representan la posibilidad de ir de una bici-estación a otra por su pertenencia a un mismo polígono y por la posible cercanía entre ellas no tomando en cuenta las calles entre ellas.

De ambas propuestas, la más fidedigna es la del diagrama **A**, pues involucraría integrar la red de calles que conectan cicloestaciones a estaciones de transporte público, lo que permitiría representar aspectos asociados a las calles por donde circulan los usuarios en bicicleta; sin embargo, no siempre es posible acceder a los datos de las calles en cualquier parte de la ciudad, sin considerar el trabajo adicional que implica transformar la representación geográfica de

las calles a una representación compatible con el modelo de gráfica que se plantea en este documento.

2.3.4. Manejo de datos dinámicos

Un aspecto importante para el sistema que se está proponiendo son los datos dinámicos. Los componentes básicos para el manejo de este tipo de datos que se proponen son los extractores (*extractors*) y los efectores (*effectors*); los primeros se encargan de integrar datos dinámicos al sistema desde diferentes orígenes, mientras que los últimos notifican a otros sistemas de cambios en los datos internos del sistema. En el directorio *extractors* se localizan los extractores, mientras que en el directorio *effectors* los efectores. Cada uno de ellos es un actor con funciones a ejecutar a modo de procesos independientes del proceso principal del sistema (la aplicación web). Para tener un mejor rendimiento de los procesos que se ejecutan en la máquina virtual se propone el uso de un almacén de hilos tal y como se muestra en el diagrama de la figura 2.9.

La mayoría de esos procesos están vivos la mayor parte del tiempo, mientras que las notificaciones a otros sistemas se suelen mantener a cargo de procesos ligeros; Scala y Akka usan una técnica conocida como Comet [13], que suele ser de utilidad en sistemas que envían mensajes como respuesta a un evento y cuyo destinatario es un cliente web. En el caso de las aplicaciones móviles existen servicios que permiten *empujar* notificaciones desde un servidor remoto, a modo de una alerta vibratoria y/o auditiva.

A continación se presentan cuatro bloques de datos dinámicos, tomándo en consideración su utilidad para los usuarios y proponiendo para cada uno un API simple.

Estimación de la saturación de estaciones

Existen implementaciones de sistemas y bibliotecas de software [Ope12] que usan algoritmos de visión computacional[TP07] para determinar el número de personas en un lugar, a partir de los cuales resulta relativamente sencillo incorporar datos sobre la saturación de una estación al sistema que se propone en este trabajo. Un mecanismo adicional, quizás más confiable, es mediante sensores en las puertas de los vehículos, los cuales llevan una cuenta de los pasajeros que entran y salen de ellos. Conocer la saturación de un vehículo o de una estación es de utilidad tanto para pasajeros como para personal operativo de los sistemas de transporte por las siguientes razones:

- **Pasajeros:** Al disponer de datos sobre la saturación de la red, es posible ajusta el perso de las aristas y así encontrar una ruta de peso mínimo, que para el pasajero representará un fragmento de información útil para llegar más rápido a su destino.

[13] Ver *Comet* en el Glosario.

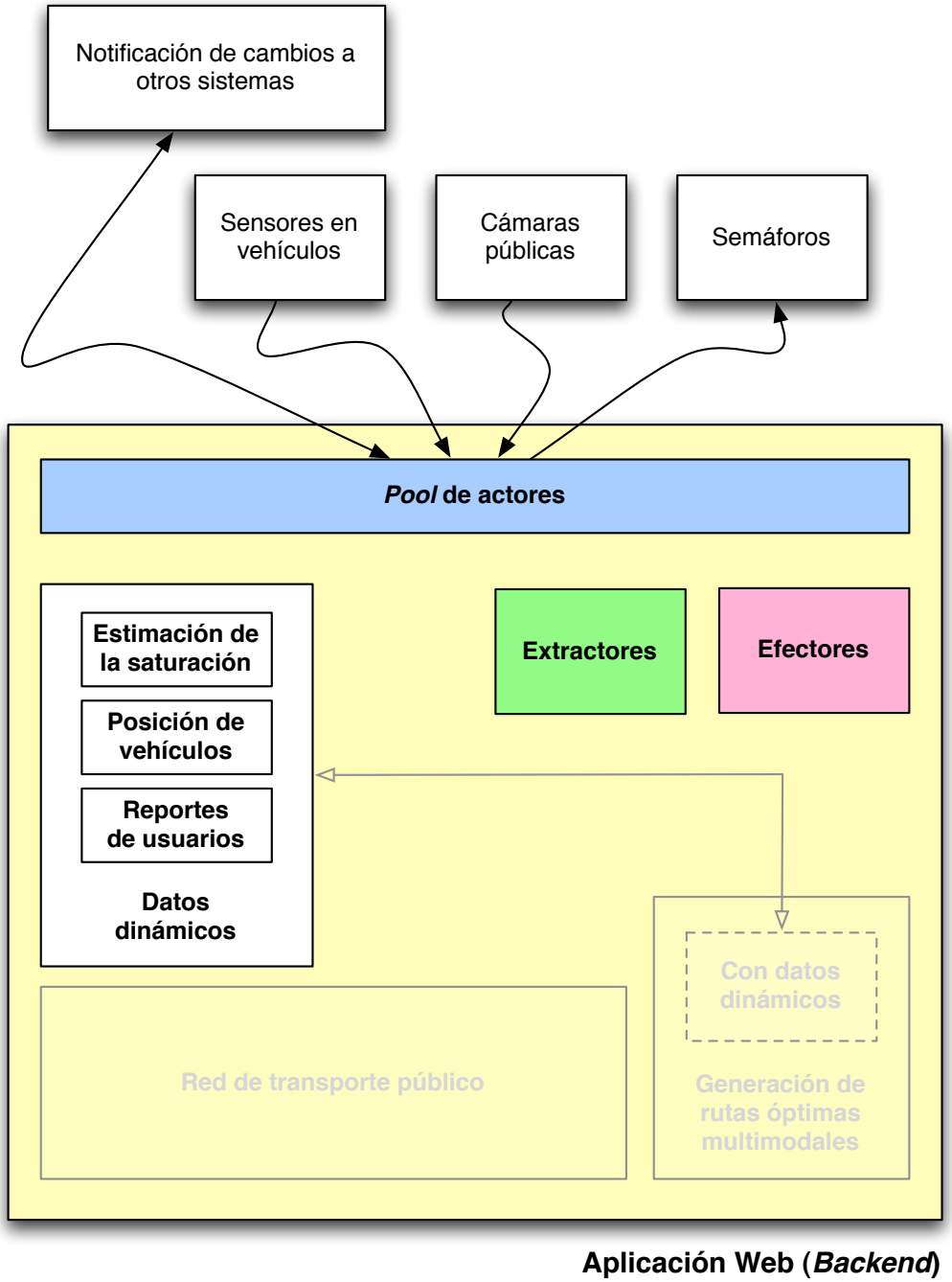

Figura 2.9: Detalles de los componentes dinámicos

- **Personal Operativo:** Visualizar el estado de saturación de vehículos y estaciones para tomar decisiones al respecto, es decir, enviar el número de vehículos necesario para cubrir la demanda de manera eficiente.

Los datos sobre la saturación de un vehículo pueden incluirse junto con los datos de su localización geográfica, por lo tanto, el API para consultar y actualizar la saturación de vehículos se presentará más adelante.

A continuación se presenta el API para consultar y crear datos relacionados a la saturación de una estación de transporte público.

Cuadro 2.15: API para saturación de estaciones

POST	**/dynamic/stations/saturation**	
int	**logical_station_id**	Identificador de la estación lógica asociada con un reporte de saturación
date	**date_time**	Fecha y hora del reporte
float	**saturation**	Saturación aproximada de la estación
GET	**/dynamic/stations/**: id/(: ms)	
int	**id**	Identificador de la estación cuyo reporte de saturación se está solicitando
int	**ms**	Fecha, en milisegundos desde 1970, para la cual se pretende obtener el reporte
GET	**/dynamic/stations/**(: ms)	
int	**ms**	Fecha, en milisegundos desde 1970, para la cual se pretende obtener el reporte

De acuerdo a la figura 2.7, se presenta la entrada *station_saturation* en el diccionario *aspects*, de cada arista que conecta cualesquiera dos estaciones de transporte público. Esta entrada representa únicamente la saturación de una estación de transporte público y se presentan las siguientes definiciones:

Sea s_i un valor para la saturación de la estación cuya arista entrante es e_i, para el cual se establece la entrada *station_saturation* al valor de s_i en su diccionario *aspects*. La arista e_i representa un tramo entre dos estaciones de transporte público.

Definición Se define a W_i como el costo de recorrer una arista e_i. Inicialmente W_i tiene un valor establecido en 0.

Incorporando el valor del atributo *station_saturation* (s_i) a W_i durante un lapso $[t_k, t_{k+j}], j \in \mathbb{N}$ se tiene que: $W_i = W_i + s_i$

En la sección anterior se habló del servicio de préstamo de bicicletas a través de bici-estaciones ubicadas en distintos puntos, pero no se mencionó que cada una de ellas almacena

el número de bicicletas disponibles y el número de lugares para estacionarlas. Ambos números pueden integrarse a la representación de la gráfica del sistema de una manera simple. Para esto se presenta la entrada *traversable* en el diccionario *aspects* que opera de manera similar a la entrada *station_saturation*.

Definición Se define como tr_i el número que representa la viabilidad de recorrer la arista e_i y que puede tomar dos valores: cero o infinito.

De la definición anterior se derivan dos escenarios:

- Si p_i es el número de espacios disponibles para estacionar una bicicleta en una bici-estación representada por el nodo v_i, se tiene entonces que para toda arista e_i entrante en v_i, si $p_i = 0$ entonces $tr_i = \infty$; en cualquier otro caso $tr_i = 0$.
- Si b_i es el número de bicicletas disponibles en una bici-estación representada por el nodo v_i, entonces para toda arista e_i saliente de v_i, se tiene que si $b_i = 0$ entonces $tr_i = \infty$, en cualquier otro caso $tr_i = 0$.

Incorporando a W_i el valor del aspecto *traversable* (tr_i) para cada arista e_i durante un periodo de tiempo $[t_k, t_{k+j}], j \in \mathbb{N}$ se tiene que: $W_i = W_i + tr_i$

En la figura 2.10 se ilustra un ejemplo con esos dos tipos de aristas marcados en colores rojo y amarillo dependiendo su estado.

Ubicación de vehículos

Mediante Sistemas de Posicionamiento Global (*GPS*) y sensores adicionales que pueden integrarse a un vehículo es factible obtener datos como su localización geográfica, velocidad, el número de pasajeros a bordo, entre otros. Sistemas de este tipo, como los construídos por [Cel14], están configurados para enviar datos a una dirección IP cada que se cumplan condiciones preestablecidas; en el caso de un GPS, cada x metros recorridos o bien cada x minutos.

Con la ubicación de cada vehículo de transporte público es posible determinar entre qué estaciones se encuentra y así calcular valores como su velocidad y saturación para poder asignarlos a la arista asociada en la gráfica. Con estos datos, se pueden obtener rutas multi-modales óptimas que tomen en cuenta factores como la disponibilidad de vehículo cubriendo alguna ruta, el menor tiempo para que un pasajero aborde un vehículo que lo acerque a su destino o bien que lo lleve a un punto de transferencia a otro tipo de transporte, así como también la posibilidad de poder elegir abordar vehículos con asientos disponibles.

Sea e_i una arista para el cual se definen los valores $sa_i[t_k, t_{k+j}]$ y $sp_i[t_k, t_{k+j}]$ que representan la saturación y velocidad promedio, respectivamente, en un lapso entre t_k y t_{k+j} con $j \in \mathbb{N}$ dados por:

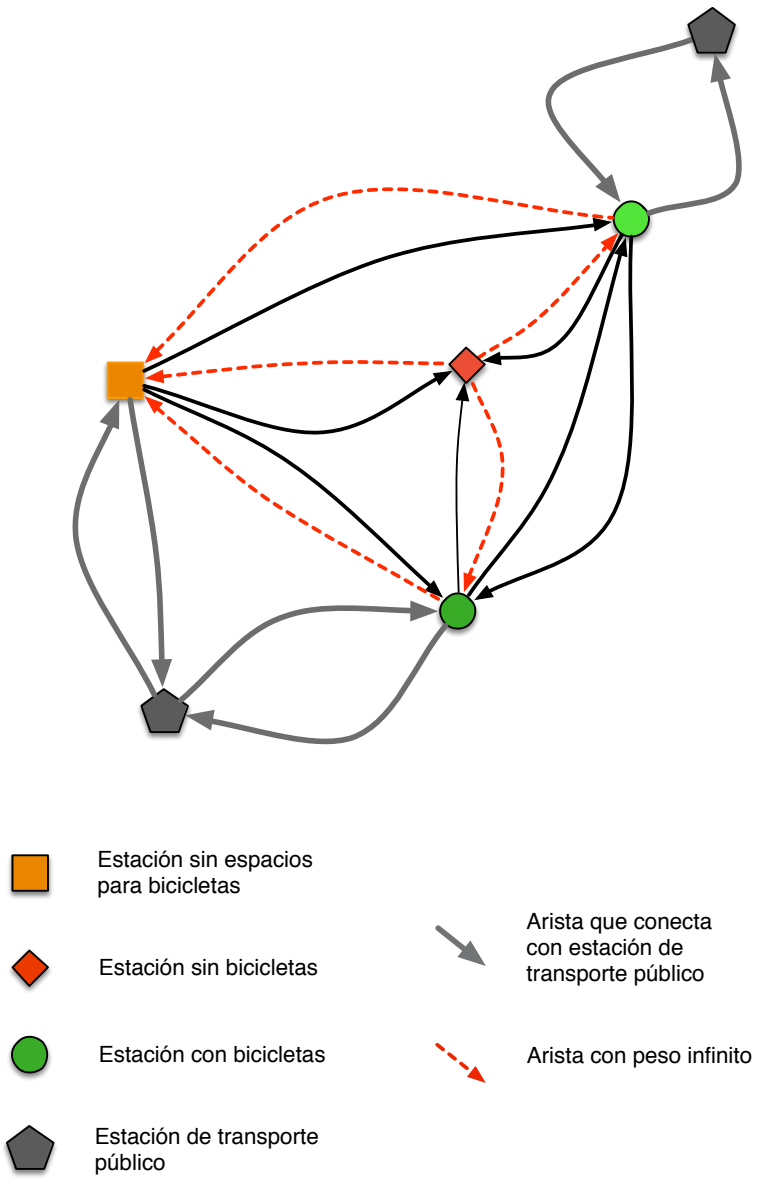

Figura 2.10: Bicicletas disponibles en bici-estaciones

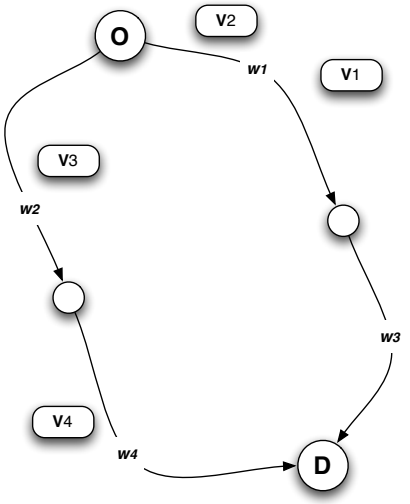

Figura 2.11: Esquema de vehículos en diferentes segmentos de caminos

$$sa_i[t_k, t_{k+j}] = \frac{1}{n} \sum_{m=0}^{n} sa_n[t_k, t_{k+j}]$$

$$sp_i[t_k, t_{k+j}] = \frac{1}{n} \sum_{m=0}^{n} sp_n[t_k, t_{k+j}]$$

donde cada $sa_n[t_k, t_{k+j}]$ representa un valor de saturación para un vehículo en el periodo de tiempo $[t_k, t_{k+j}]$ respecto a la arista e_i. De manera análoga $sp_n[t_k, t_{k+j}]$ representa un valor de velocidad para el mismo vehículo bajo las mismas condiciones.

Definición Se definen para cualquier arista de la gráfica los atributos *vehicle_saturation* y *vehicle_speed* con los valores sa_n y sp_n, respectivamente.

El postulado y definición anteriores se ilustran en el diagrama de la figura 2.11, en la que se presentan los puntos origen(O) y destino (D) con nodos intermedios. Entre esos puntos se presentan algunos vehículos V_1, V_2, V_3, V_4, cada uno de los cuales tiene un valor entero que describe su saturación $sa_1 = V_1(sa), ..., sa_i = V_i(sa)$, así como un valor decimal que describe su velocidad $sp_1 = V_1(sp), ..., sp_i = V_i(sp)$.

Para estos dos últimos aspectos, el valor del número W_i se reajusta al tomar en cuenta el valor de cada uno de ellos, almacenados como *vehicle_speed* y *vehicle_saturation* durante un periodo de $[t_k, t_{k+j}], j \in \mathbb{N}$. Tenemos para saturación: $W_i = W_i + sa_i$ y se observa para velocidad: $W_i = W_i + sp_i$.

Además de la integración de los datos a la gráfica, es deseable acceder a los atributos asociados a cada tupla de la tabla INSTANTS. En el cuadro 2.16 se presenta un API con algunas funciones.

De igual forma en la que un grupo de tuplas en la tabla INSTANTS de un vehículo facilitarían conocer su velocidad promedio, es deseable conocer la tendencia de saturación de un vehículo a lo largo del tiempo para que otros sistemas puedan consultarla. Pensando un poco más en torno a la flexibilidad que ofrece el conjunto de tecnologías actuales, resulta interesante, al menos tecnologicamente hablando, pensar en nuevas formas en que pasajeros, supervisores y conductores del transporte público interactúen dentro de este contexto.

Cuadro 2.16: API para la ubicación de vehículos y su saturación

POST	**/dynamic/vehicles/instant**	
int	**vehicle_id**	Identificador del vehículo asociado con un nuevo instante por registrar
date	**date_time**	Fecha y hora del reporte
float	**passengers**	Número actual de pasajeros en el vehículo
int	**speed**	Velocidad del vehículo
float	**latitude**	Latitud
float	**longitude**	Longitud
int	**report_quality**	Calidad del reporte recibido
GET	**/dynamic/vehicles/**: *id*/**instants/**(: *ms*)	
int	**id**	Identificador del vehículo para el cual se están solicitando sus instantes registrados
int	**ms**	Fecha en milisegundos, desde 1970, para la cual se pretende obtener el reporte

Reportes de usuarios

El uso de dispositivos de cómputo móvil es cada vez mayor, por lo que se propone que el sistema cuente con un API para poder integrar datos desde otros sistemas que faciliten a los usuarios reportar eventos como accidentes, bloqueos o retrasos en el servicio. La idea es que desde una aplicación móvil o web, cualquier usuario pueda reportar un incidente en algún tramo o estación de la red de transporte público; para ello se considera necesario establecer un conjunto de requerimientos que permitan saber si un reporte (o conjunto de reportes) es válido. Se proponen los siguientes requerimientos que habrían de ser implementados en tales sistemas:

- Detectar el origen geográfico (coordenadas) de los reportes.
- Detección de tendencias que involucren palabras clave relacionadas dentro de un área geográfica determinada en un periodo de tiempo

De manera adicional a la verificación que pudieran implementar los sistemas que faciliten la participación de los usuarios, con carácter opcional los reportes recibidos podrían colocarse en una bandeja de notificaciones que estarían siendo recibidas por personas encargadas de la supervisión del correcto funcionamiento de los sistemas de transporte público.

Otra alternativa, son las redes sociales como Twitter, Facebook y Waze, que por su alcance y número de usuarios facilitarían la recopilación de reportes en tiempo real; considerando además, que actualmente hay personas encargadas de administrar las cuentas de los sistemas de transporte público principales, las cuales al detectar una tendencia de reportes relacionados a un mismo incidente podrían verificar su veracidad y finalmente levantar un reporte que, sin requerir verificación, se integre al sistema.

Los datos que se integran al sistema desde cada reporte contribuyen a reajustar los pesos de las aristas de la gráfica. Para casos en los que estaciones lógicas (LogicalStation) sean reportadas, la arista cuyo peso será reajustado es aquella que incide en el nodo que representa a la estación en la gráfica.

Se plantea el concepto de Reporte propagable (Pr), que se recibe desde cualquiera de las fuentes anteriores, el cual es una tupla en dos presentaciones:

$$Pr_1(coordinate, radius, traversable\_value, opts)$$

En esta presentación el sistema obtiene la lista de aristas que pertenecen al radio, dado en metros en la entrada *radius*, respecto a la coordenada, dada en la entrada *coordinate*. Para cada arista el sistema ajusta el valor del aspecto *traversable* dado en la entrada *traversable_value*:

$$Pr_2(arc\_list\_ids, traversable\_value, opts)$$

En esta presentación el sistema solamente ajustará el valor del aspecto *traversable* dado en la entrada *traversable_value* para cada una de las aristas presentes en la lista con la entrada *arc_list_ids*.

Adicionalmente, en la entrada *opts* puede haber un diccionario con dos valores: *resource_type* y *resource_id*. En el cuadro 2.17 se presenta un API para crear reportes que consideran un radio de afectación asociado a una coordenada geográfica.

Cuadro 2.17: API para integrar reportes de tipo Pr_1

POST	**/dynamic/reports/first**	
int	**model_type** (*opcional*)	El tipo de modelo en el que ocurrió el incidente. Los tipos posibles son: (1) LogicalStations; (2) Vehicles; (3) ConnectionLinks
int	**model_id** (*opcional*)	El identificador en la base de datos del modelo en el que ocurrió el incidente
date	**date_time**	Fecha y hora de reporte del incidente
float	**latitude**	Latitud
float	**longitude**	Longitud
float	**radius**	Radio del reporte
int	**traversable_value**	Detalla el nivel de afectación que representa este evento reportado como incidente. Se proponen dos escalas con valores numéricos: (1) *Bajo*, (2) *Alto*

En el cuadro 2.18 se presenta el API para crear reportes que consideran una lista de aristas afectadas por este reporte.

Cuadro 2.18: API para integrar reportes de tipo Pr_2

POST	**/dynamic/reports/second**	
int	**model_type** (*opcional*)	El tipo de modelo en el que ocurrió el incidente: (1) LogicalStations; (2) Vehicles; (3) ConnectionLinks
int	**model_id** (*opcional*)	El identificador en la base de datos del modelo en el que ocurrió el incidente
date	**date_time**	Fecha y hora de reporte del incidente
list	**edges**	La lista de identificadores de las aristas marcadas como afectadas por este reporte
int	**traversable_value**	Detalla el nivel de afectación que representa este evento reportado como incidente. Se proponen dos escalas con valores numéricos: (1) *Bajo*, (2) *Alto*

El tiempo de cada trayecto

Antes de cerrar esta sección se considera también el aspecto del tiempo (*time*) en cada uno de las aristas de la gráfica, definido como el tiempo que toma transitar esa arista a determinada hora, basándose en datos dinámicos o bien en datos estadísticos en caso de que los primeros no estén disponibles.

Suponiendo que para cada vehículo en la red de transporte público sea posible obtener la hora precisa en la que deja una estación y la hora precisa en la que llega a la siguiente, entonces sería posible obtener el tiempo que le tomó transitar entre ambas estaciones.

Definición Se define ti_i como el tiempo que le tomó al último vehículo moverse a lo largo del trayecto representado por una arista e_i que conecta a dos estaciones de transporte representadas en la gráfica de la red de transporte público como v_i y v_{i+1} durante un periodo de tiempo definido por $[t_k, t_{k+j}], j \in \mathbb{N}$.

Una definición alternativa a la anterior que considera a otros vehículos desplazándose en el trayecto ti_i se presenta a continuación.

Definición Se define ti_i como el promedio de los reportes de los vehículos que han transitado una arista e_i que conecta a dos estaciones de transporte representadas en la gráfica de la red de transporte público como v_i y v_{i+1} durante un lapso determinado $[t_k, t_{k+j}], j \in \mathbb{N}$.

Para este conjunto de datos, resultaría conveniente reutilizar el API presentado en la figura **??** para vehículos. Mediante la tecnología de *geofencing* [14] sería relativamente sencillo conocer el tiempo en el que un vehículo circuló dentro del área de un polígono previamente delimitado y asociado a una estación de transporte público: de esta forma se puede calcular el tiempo de desplazamiento entre dicha estación y la siguiente según la dirección del vehículo.

El valor de ti para cualquiera de las dos definiciones será asignado a una entrada en el diccionario de aspectos bajo la llave *time* para poder ser integrada en el valor W_i de cada arista.

Una alternativa a la ausencia de datos dinámicos que permitan conocer el tiempo de traslado entre estaciones, se propone integrar datos estadísticos con los tiempos promedio para cada tramo entre cualesquiera dos estaciones en la red. Según la Encuesta Origen-Destino del INEGI[INE07] existen tres bloques de tiempo de máxima demanda junto con cuatro bloques de tiempo de baja demanda distribuidos a lo largo del día. La gráfica con estos datos se presenta en la figura 2.13. Para tener mayor precisión en el valor que toma la llave *time* a lo largo del día en cada arista de la gráfica se propone adjuntar una tabla con el tiempo que toma recorrer cada arista en algunos de los bloques de tiempo predominantes identificados por la Encuesta Origen-Destino.

En la tabla 2.12 se presenta la estructura de la tabla en la base de datos que habría de almacenar los datos estadísticos esbozados anteriormente. Esta tabla es una extensión a la tabla 2.5.

[14] *Geofencing* descrito en el Glosario.

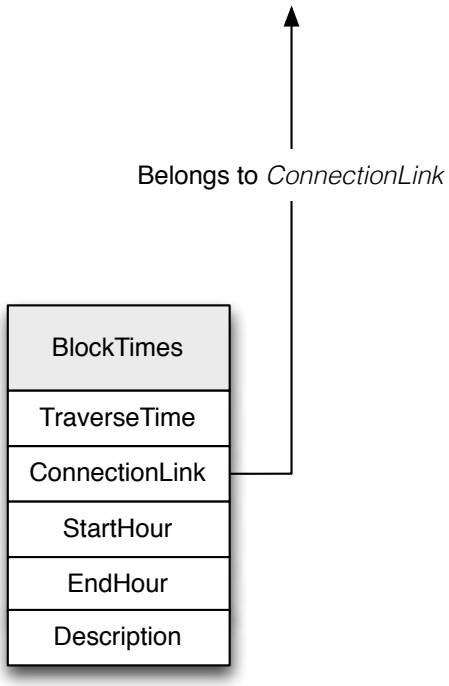

Figura 2.12: Tabla para almacenar tiempo de recorrido para cada bloque de tiempo y cada arista

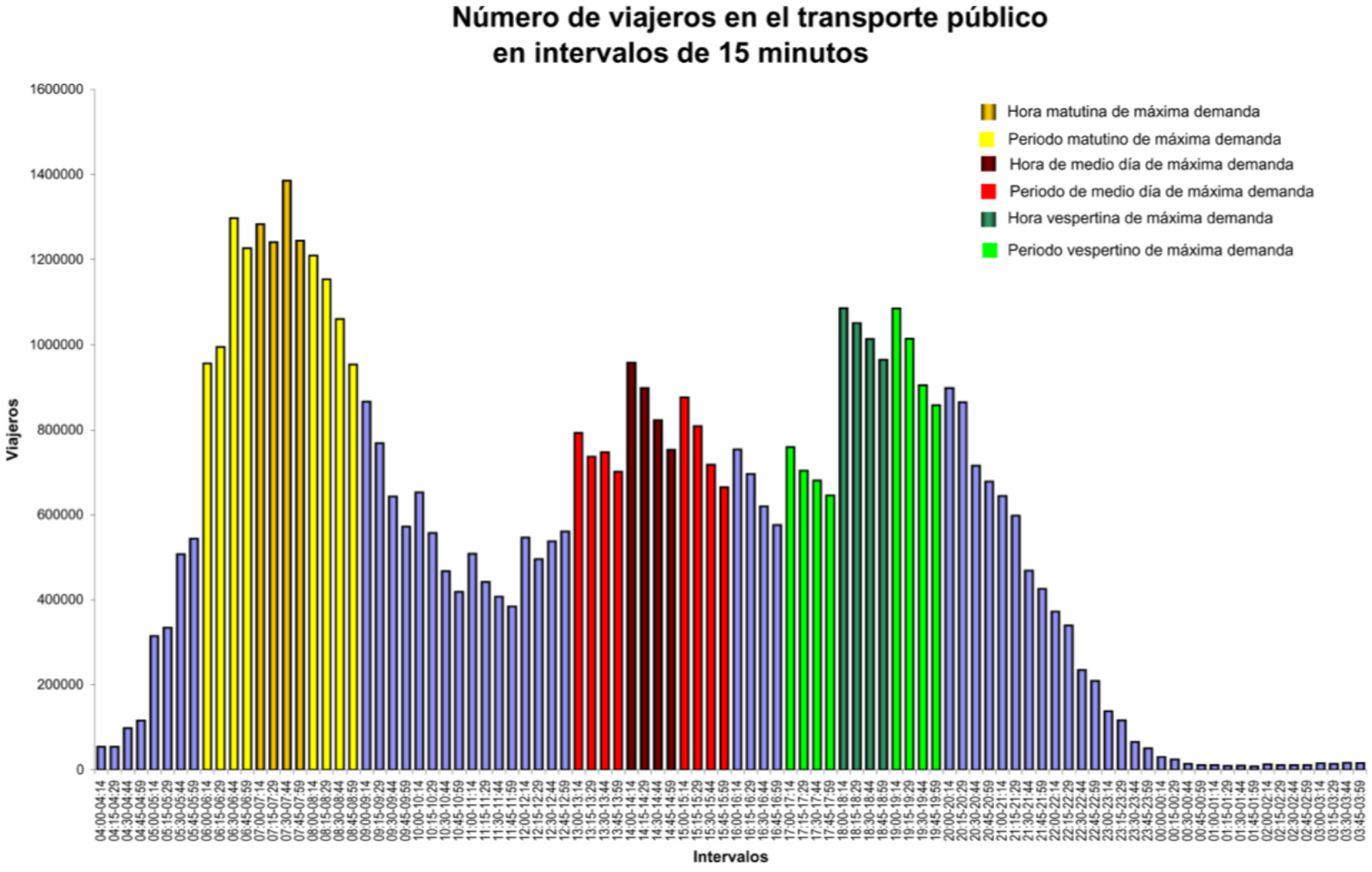

Figura 2.13: Afluencia de usuarios a lo largo del día (Fuente INEGI)

2.4. Algoritmos y distribución de datos

En lo que va del capítulo se describieron los aspectos más relevantes de la red de transporte público sujetos a ser representados en el sistema usando tecnologías disponibles hoy en día, junto con algunos detalles técnicos para su implementación. Los aspectos dinámicos modelados para cada arista en la gráfica son; *traversable*, *vehicle_saturation*, *vehicle_speed*, *station_saturation* y *time*. Ahora resta definir el mecanismo mediante el cual el sistema habría de interactuar con otros sistemas web y con aplicaciones móviles para responder preguntas relacionadas a la generación de rutas multi-modales óptimas.

Un API para que cualquier aplicación pueda solicitar rutas óptimas entre cualesquiera dos nodos de la gráfica asociada a la red de transporte público se presenta en el cuadro 2.19. De manera interna, estarían operando el algoritmo de Dijkstra o bien la heurística A*, los cuales se detallan en el Apéndice B.

Simplificando, para cada petición a la primera ruta del API para generación de rutas óptimas, el valor W_i para cada arista en la gráfica será de 1, la distancia entre un nodo y otro en la gráfica; mientras que para cada petición a la segunda ruta del mismo API, el valor de W_i para cada arista será el resultado de la suma de los parámetros habilitados. Con un peso asignado a cada arista de la gráfica, el algoritmo de Dijkstra o el de A* puede obtener la ruta más corta entre la estación origen (*origin_station*) y la estación destino (*destination_station*).

Cuadro 2.19: API para generación de rutas óptimas

GET	**/routing/simple**	
int	**origin_station**	Identificador de la estación desde donde se inicia el recorrido.
int	**destination_station**	Identificador en la estación donde termina el recorrido.
GET	**/routing/advanced**	
int	**origin_station**	Identificador de la estación desde donde se inicia el recorrido.
int	**destination_station**	Identificador en la estación donde termina el recorrido.
list	**using_transports**	Usa los medios de transporte en la lista de acuerdo a su identificador entero (Según los presentados en el punto 1 de la figura 2.5).
boolean	**considers_speed**	Considerar o no la evaluación de la velocidad promedio en las aristas. Valor predeterminado *true*.
boolean	**considers_saturation**	Considerar o no la saturación de las aristas. Valor predeterminado *true*.
boolean	**only_accessible_stations**	El recorrido está restringido únicamente a estaciones con infraestructura de accesibilidad para personas con alguna discapacidad motriz. Valor predeterminado *false*.
boolean	**considers_time**	Considerar o no los reportes recientes sobre el estado operativo del servicio. Valor predeterminado *false*.
int	**algorithm_type**	El tipo de algoritmo/heurística a usar: DIJKSTRA o A*. Valor predeterminado DIJKSTRA (1).
boolean	**ignores_traversable**	Ignora el estado de las aristas marcados como no transitables debido a algún reporte. Valor predeterminado *false*.

Parte II

Aplicaciones

Capítulo 3. Aplicación del Modelo propuesto

En este último capítulo se analiza la aplicación del modelo desarrollado durante este trabajo en dos escenarios distintos. El primer escenario involucra el servicio de transporte interno universitario de la UNAM, PUMABÚS; mientras que el segundo involucra al METROBÚS de la Ciudad de México.

3.1. Manipulación de la topología de la red

Con los principales componentes del sistema detallados, lo que sigue es plantear una solución que facilite la captura y adquisición de datos tanto estáticos como dinámicos. En esta sección se plantea un sub-sistema que facilita la captura de la topología de la red de transporte público y de los datos geográficos asociados.

A continuación se proponen una serie de *wireframes*[15] para los modelos de datos asociados a la topología antes mencionada, de forma que la topología de la red pueda ser modificada con facilidad.

El modelo VEHICLEJOURNEYS

Es importante considerar las relaciones que tiene este modelo con otros modelos en la base de datos al momento de plantear un componente del sistema para administrar la topología de la red. Este modelo, pertenece a una línea de transporte público, la cual a su vez, pertenece a una agencia de transporte. En las figuras 3.1 y 3.2 se muestran *wireframes* con la lista de agencias y sus líneas asociadas, así como la forma para añadir una nueva línea a una agencia existente.

Adicionalmente, hay que notar que en las formas de registro de modelos que integran una llave foránea de un modelo padre, se omite la presentación de un campo de captura visible correspondiente a esa llave foránea y, en cambio, se utiliza un campo oculto que asocia a un

[15] *Wireframes* descrito en el Glosario.

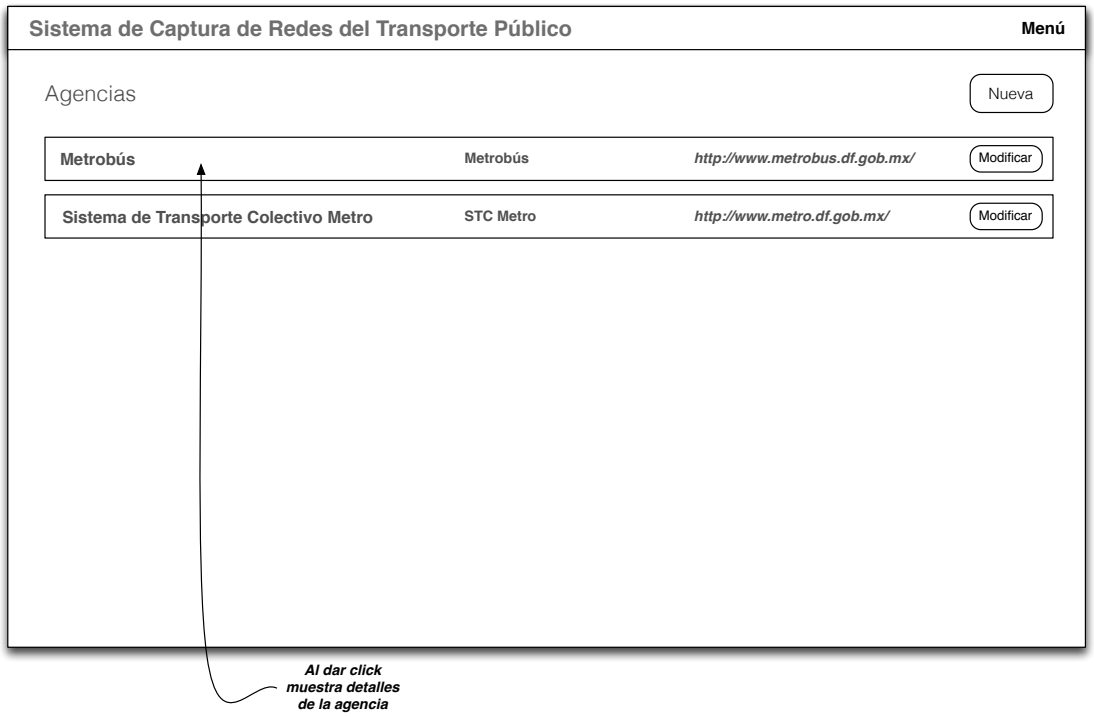

Figura 3.1: Listado de Agencias de Transporte

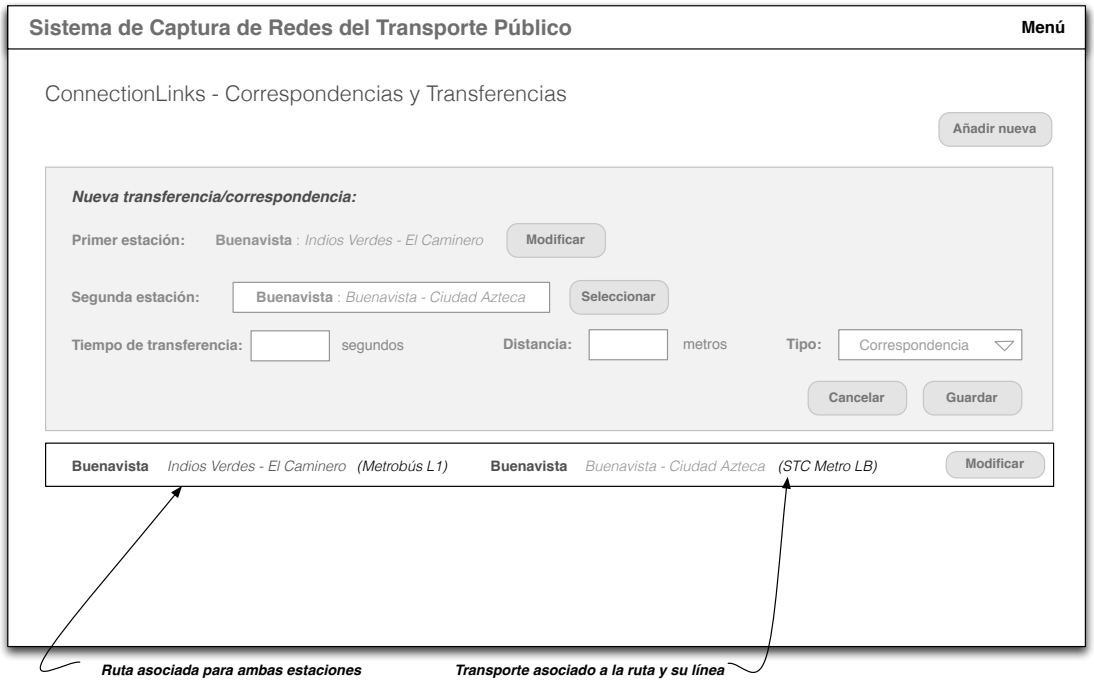

Figura 3.2: Agregando una nueva línea

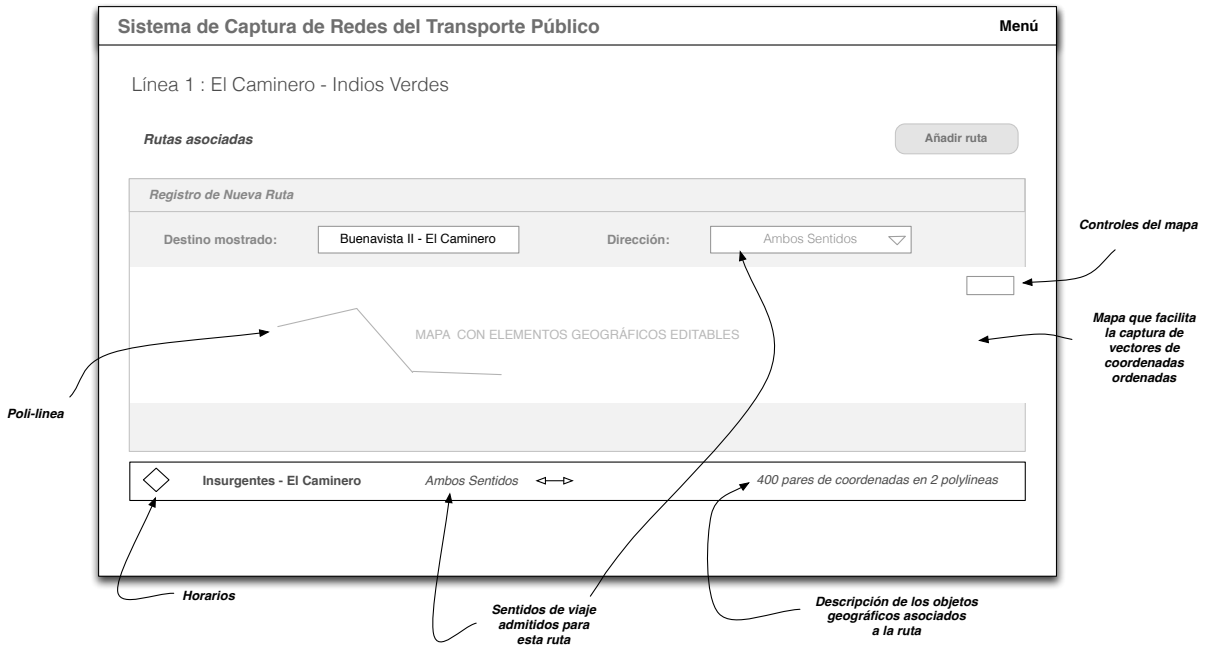

Figura 3.3: Agregando una ruta asociada a una línea

modelo padre existente a partir de su selección en una pantalla previa. Un ejemplo de esto se puede apreciar en el formulario de registro de una línea de transporte público. De acuerdo con la Figura 3.2, al añadir una línea de transporte público al METROBÚS se da por hecho que la llave foránea que representa la agencia de transporte a la que pertenecerá la línea a registrar apuntará a la tupla existente en la tabla AGENCIES; METROBÚS.

En la figura 3.3 aparece un *wireframe* que muestra la interfaz para registrar una ruta de transporte público (VEHICLEJOURNEYS), así como el listado de rutas existentes. Dado que uno de los campos de este modelo de datos representa datos geográficos asociados a vectores de coordenadas, se plantea que este sistema de captura permita al usuario registrar y posteriormente modificar líneas poligonales (*polyline*) [16] usando un mapa. Desde esta pantalla, el usuario debería también poder registrar los horarios en los que esta ruta opera y el promedio de las frecuencias de paso entre vehículos que la recorren (no mostrado).

Modelo PHYSICALSTATIONS

Para los datos de este modelo se proponen los *wireframes* que se muestran en las figuras 3.4 y 3.5. En ellos se detalla tanto el registro de nuevas estaciones y modificaciones a las existentes, como componentes que auxilien al usuario que se encarga de administrar los datos de la red de transporte público. Por ejemplo, se provee de un componente de búsqueda tomando en cuenta que el número de estaciones en la red es un número extenso. Se plantea

[16] Ver *polyline* en el Glosario.

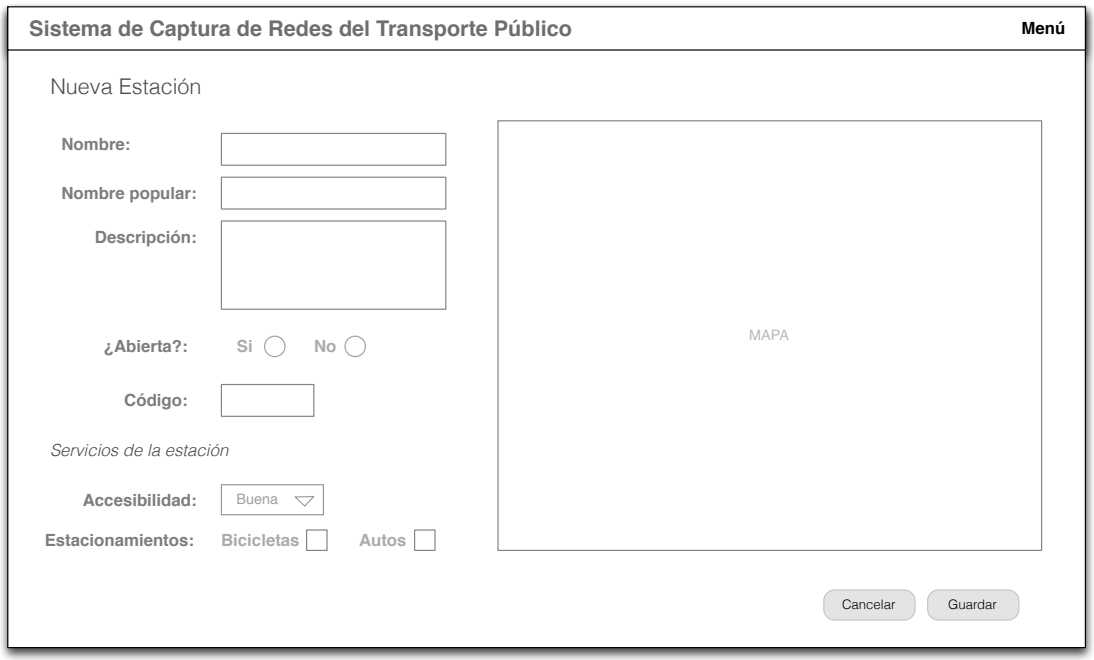

Figura 3.4: Forma para registrar estaciones

que la funcionalidad mínima de dicho componente sea la búsqueda de estaciones por nombre. Adicionalmente, se permite capturar/modificar información adicional a una estación de transporte público como la representada en la tabla StationServices.

Los modelos LogicalStations y ConnectionLinks

En la figura 3.6 se muestra una manera de asociar estaciones físicas (previamente registradas) a una ruta de transporte público y al mismo tiempo asignarles un orden único en el recorrido de la ruta, lo cual es requerido por la representación de la gráfica. Por el tipo de interfaz es posible cambiar el orden de cada estación con sólo arrastrar el elemento y colocarlo antes o después de otros elementos de la lista. Por otro lado, en la figura 3.7 se muestra el *wireframe* que describe la interfaz para añadir y modificar correspondencias y transferencias entre rutas y líneas de transporte público.

Finalmente, será deseable que este sistema ofrezca la funcionalidad de serializar la topología de la gráfica a uno o varios formatos de representación de datos. Un formato común para ello es *GraphML*, el cual está basado en XML y soporta gráficas dirigidas y no dirigidas, entre otras. Adicionalmente permite referenciar datos externos que no hayan sido incluídos conjuntamente con la serialización de la gráfica.

Se sugiere como funcionalidad adicional añadir una vista que permita al usuario visualizar la topología de la red capturada en el sistema, así como los metadatos más importantes, pues resulta mucho más fácil examinar un conjunto de datos que se entienden mejor en una

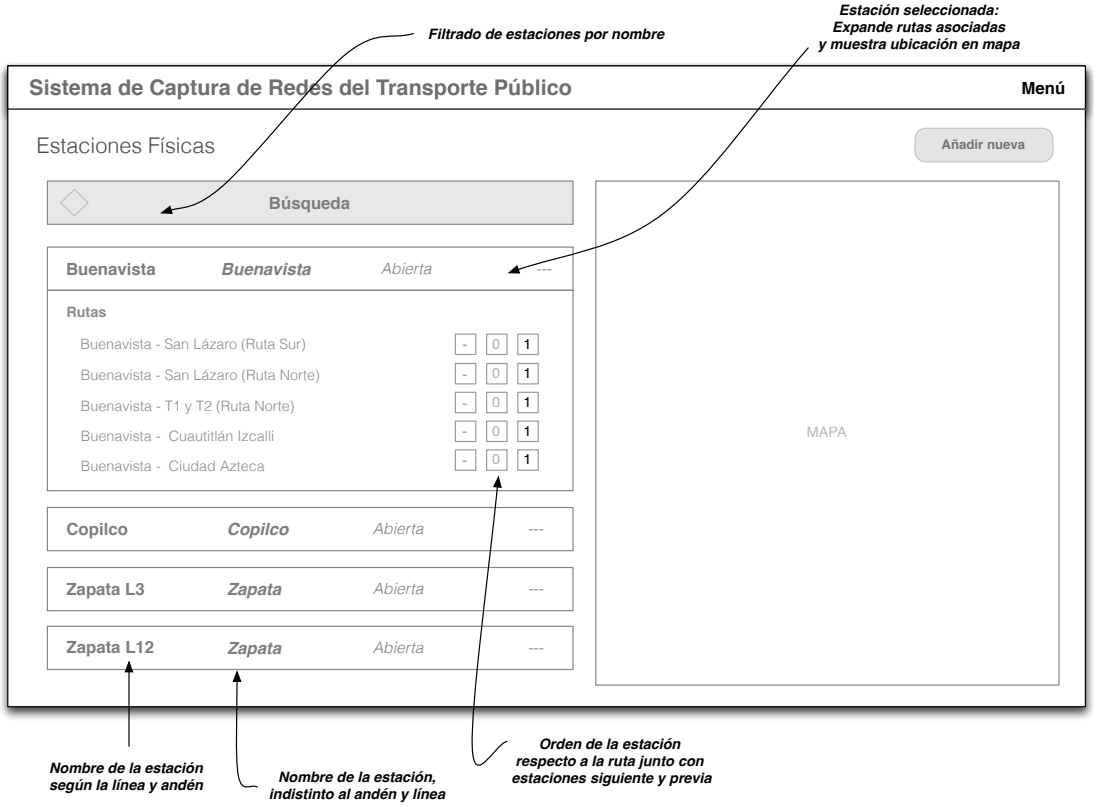

Figura 3.5: Listado de estaciones

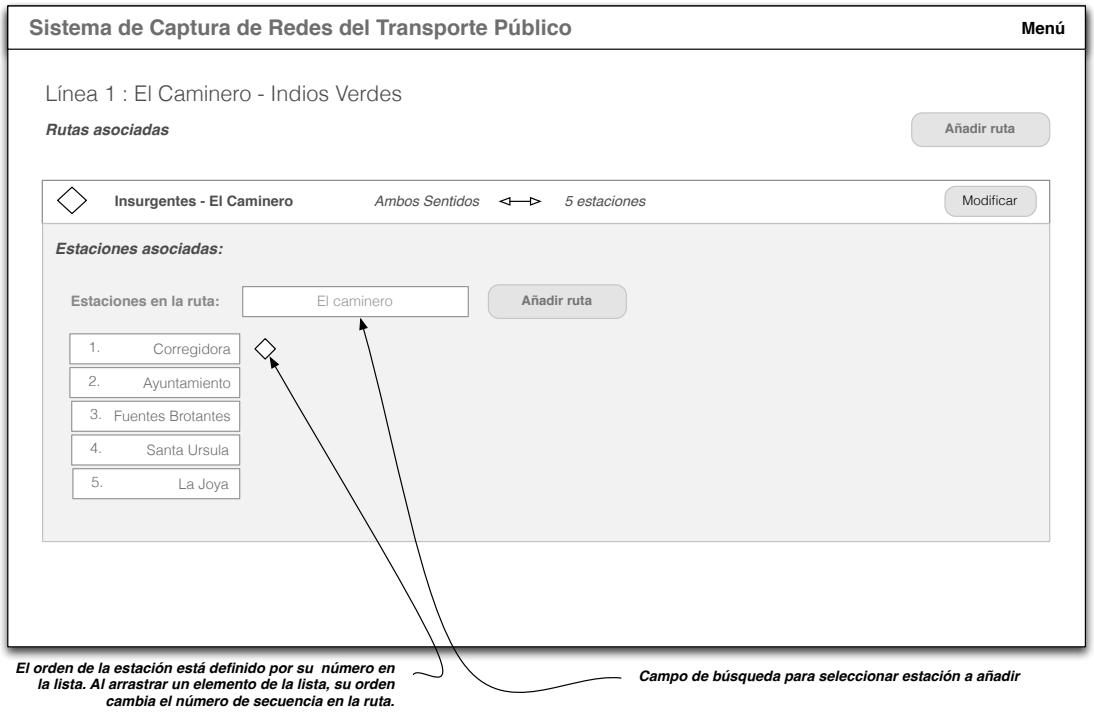

Figura 3.6: Forma para asociar estaciones físicas a una ruta

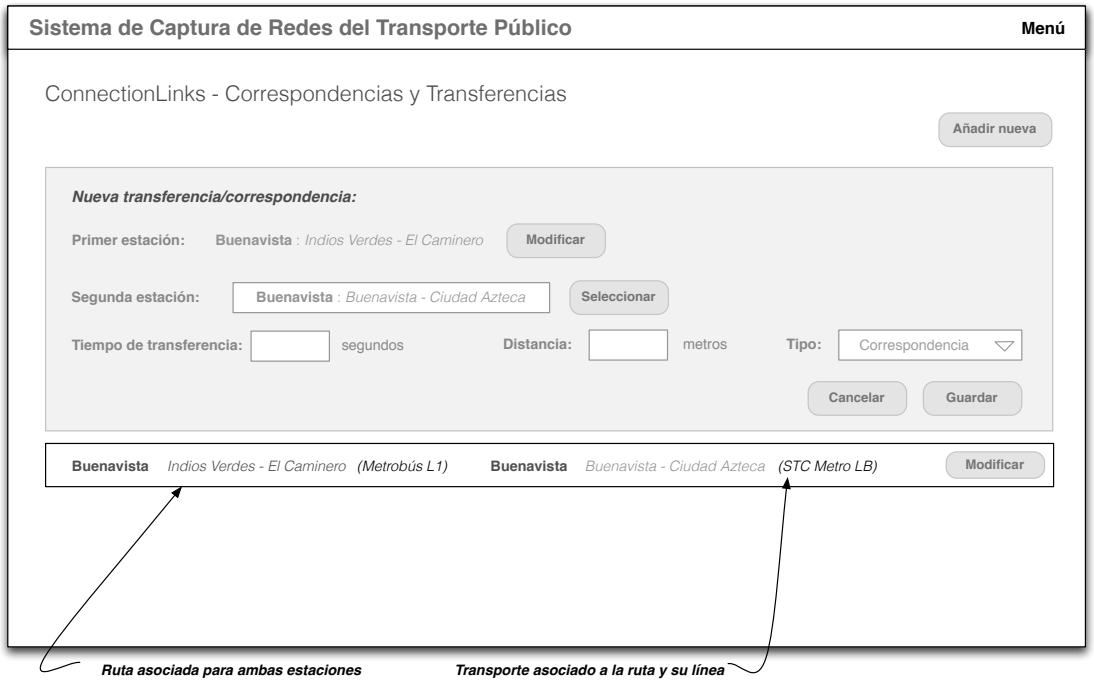

Figura 3.7: Listado y formato para registrar correspondencias/transferencias entre estaciones

Figura 3.8: Captura de pantalla de un sistema que muestra la topología de una red de transporte público

representación de gráfica que como un conjunto de listas aisladas. La Figura 3.8 presenta un ejemplo de esto usando la biblioteca Springy.js[Spr13] para dibujar la gráfica en el browser usando el elemento canvas de HTML 5.

3.2. Pumabús

Ciudad Universitaria cuenta desde el 2008 con un plan de movilidad al interior del campus en el cual es de gran importancia el sistema de transporte interno que da servicio diariamente a alrededor de 135,000 personas. Recientemente se implementó un sistema de localización vehicular en tiempo real para los autobuses del sistema de transporte interno el cual está basado en sistemas GPS/GPRS[17] y controladores de hardware instalados en cada unidad. Además de reportar la ubicación del autobús, este sistema permite detectar y controlar acciones como el encendido del motor, botón de pánico y la apertura o cierre de puertas.

3.2.1. Solución actual y tipo de datos dinámicos disponibles

Originalmente los datos de los reportes de ubicación de cada vehículo eran usados únicamente por el Centro de Monitoreo de la vialidad interna de Ciudad Universitaria para monitorear el estado del servicio y detectar anomalías y/o alteraciones en este.

En el año 2012 se comenzó con el desarrollo del proyecto Pumabús Móvil a través del

[17] Ver *GPS/GPRS* en el Glosario.

cual se construyó un *back-end*[18] para organizar los reportes recibidos a lo largo del día para cada autobús, con lo que es posible hacer, entre otras funcionalidades, consultas específicas tales como 1) encontrar los reportes recibidos en los últimos x minutos, 2) los vehículos que recorren una ruta en particular, y 3) el último reporte para una unidad en específico.

Además del *back-end* existente se cuenta con un API que permite a otros sistemas consultar datos. Haciendo uso de dichos datos se implementaron tres *front-ends*[19], los cuales permiten a los usuarios consultar la ubicación en tiempo real de los autobuses dentro del campus de Ciudad Universitaria. Dos de estos front-ends funcionan de manera nativa[20] en los dos sistemas operativos móviles más populares, mientras que el tercero funciona únicamente en la web.

3.2.2. Retos y oportunidades de mejora

Si bien el número de rutas de este sistema de transporte público son pocas y cada una tiene un número pequeño de estaciones, es conveniente contar con un sistema para capturarlas y modificarlas. Una vez capturadas la construcción de la gráfica sobre la cual puedan obtenerse rutas multi-modales óptimas es inmediata. Esta representación permitiría incluir el sistema de préstamo de bicicletas BiciPuma. El mecanismo de integración de los datos en tiempo real disponibles en el proyecto Pumabús operaría tal y como se ha planteado en este trabajo.

Una oportunidad de mejora adicional es permitir que la misma aplicación móvil pueda generar la ruta más eficiente para llegar de un punto a otro sin depender de una conexión activa a internet, aunque para ello los algoritmos habrían únicamente de considerar datos estáticos de la red de transporte; contar con esta funcionalidad sería de gran utilidad para el usuario. Existen dos posibilidades para representar la gráfica en la memoria del dispositivo de cómputo móvil: la primera usando una matriz de adyacencias para representar los nodos y las aristas entre ellos (incluyendo sus pesos, si fuera necesario) y la segunda reconstruyendo la gráfica usando una biblioteca específica similar a *Jung*. Probablemente, la primera opción sea la mejor, pues requiere de menor espacio en memoria y es simple de entender y de representar tomando en cuenta que su propósito sería solamente el de representar la topología de la gráfica.

3.3. Metrobús

En el año 2005 inicia operaciones la primera línea del Metrobús en la Ciudad de México. El Metrobús es un sistema de autobuses artículados que implementa el modelo *BRT*[21]. Actualmente cuenta con cuatro líneas y transporta alrededor de 760 mil pasajeros diariamente.

[18] Ver *back-end* en el Glosario.
[19] Ver *front-ends* en el Glosario.
[20] Ver aplicaciones móviles nativas en el Glosario.
[21] Ver *Bus rapid transit* en el Glosario.

Este medio de transporte público ha tenido muy buena aceptación por parte de los usuarios en sus cuatro líneas, tanta, que la Línea 1 sufre saturaciones en horas de alta demanda del servicio.

3.3.1. Estado actual y solución propuesta

A pesar de que existen muchas soluciones basadas en sistemas de telecomunicaciones y de software que ayudan a mejorar la logística de un servicio con las características del Metrobús, a la fecha en la que se escribe este documento, las herramientas logísticas que se utilizan en este sistema de transporte público son el papel y la comunicación entre operadores mediante dispositivos de radiofrecuencia para supervisar el flujo de las unidades. Evidentemente esto impacta la calidad del servicio porque depende de la sincronización entre humanos, quienes tienden a cometer errores y sobre todo porque los datos no están centralizados, lo cual impide ver el panorama completo en cuanto al desempeño del servicio en determinado momento.

Actualmente, el Metrobús no cuenta con ninguna fuente de datos en tiempo real que pueda ser de utilidad para informar a sus usuarios sobre el estado del servicio. La solución ideal habría de integrar datos dinámicos como la saturación y la ubicación de cada autobús para que el usuario pueda planear de mejor forma sus recorridos. Sin embargo, por ahora sólo es posible contar con una solución medianamente útil que considera datos estáticos y posiblemente datos estadísticos.

En la siguiente sección se detalla el funcionamiento del sistema, desde la captura de los datos para dos rutas del Pumabús y dos rutas del Metrobús hasta su representación en la gráfica.

3.3.2. Ejemplo de integración para redes de transporte público al modelo

Para concluir con este modelo se ejemplificará el proceso de integración de datos para cuatro rutas: dos del Pumabús y dos del Metrobús. Se tomará un subconjunto de las paradas totales de cada ruta procurando que haya paradas compartidas entre rutas, o bien, lo suficientemente cercanas entre sí para ser consideradas como correspondencias.

La tabla principal del sistema, de la cual depende la tabla que contiene las líneas de todos los transportes (Lines) es la tabla de agencias (Agencies). En la figura 3.9 se presentan los datos básicos de las dos agencias de transporte público que se expondrán en este ejemplo, mientras que en la figura 3.10 se listan dos líneas de transporte asociadas a esos transportes.

En la figura 3.11 se listan las estaciones del Pumabús que fueron seleccionadas para este ejercicio, mientras que en la figura 3.12 las del Metrobús. Para ambos grupos se seleccionaron estaciones o paradas aleatorias, respetando su orden en la línea o ruta. Las

Agencies				
Id	**Name**	**WebPage**	**Twitter**	**PopularName**
1	Metrobús	www.metrobus.df.gob.mx	@Metrobus_GDF	Metrobús
2	Pumabús	www.pumabus.unam.mx	@Pumabusunam	Pumabús

Figura 3.9: Tabla con datos de dos agencias de transporte público

paradas del Pumabús serán identificadas mediante números, mientras que las estaciones del Metrobús usando letras. Es conveniente hacer notar que la elección de un número mayor de estaciones para la Línea 2 del Metrobús respecto a las demás líneas y rutas se debe a que se desea mostrar claramente la representación de estaciones especiales, las cuales cuentan con andenes separados y por lo general sirven a una única dirección en el caso del Metrobús.

Habiendo seleccionado las estaciones, en la figura 3.13 se presenta la tabla con los detalles de cada estación física. En ella se muestran solo algunas para ejemplificar sus datos. Cabe señalar que la columna de *Code* está formada por el identificador de cada estación (letra o número) y por el par de símbolos "PM"para Pumabús y "MB"para Metrobús. De esta forma será más fácil trabajar con ellas al momento de registrar estaciones lógicas que son necesarias para formar recorridos y correspondencias/transferencias.

Este ejercicio supone que existe un sistema para capturar datos similar al que se presentó en la sección 3.1. Lo siguiente es seleccionar un recorrido para añadir a él estaciones. Es importante explicar algunos detalles relacionados a como se planean representar los recorridos. Un recorrido, de forma muy genérica, es una lista de estaciones o paradas con un orden específico en una dirección. Por tanto, en el caso del Pumabús cualquier ruta es a la vez un recorrido y en su estado actual, para cada ruta existe un único recorrido, pues el servicio en cada una de ellas tiene una sola dirección. Esto es diferente al Metrobús, puesto que el servicio en todas las líneas se ofrece en ambos sentidos, pero además existen recorridos identificados por letras entre estaciones establecidas. La Línea 1 tiene seis recorridos que prestan servicio en ambas direcciones; por ejemplo, en el recorrido $Indios\,Verdes \longleftrightarrow Insurgentes$ hay autobuses que llevan pasajeros entre ambas estaciones en ambas direcciones, lo mismo ocurre para los demás recorridos. Finalmente, es bueno hacer notar que para esta línea existen dos recorridos que inician en una estación dentro de ella, pero que terminan en una estación de otra línea. La representación de recorridos como el anterior no supone trabajo adicional respecto a lo que ya se ha desarrollado, pues este tipo de recorridos se conocen como transferencias.

En la figura 3.14 se presenta la tabla de recorridos con un recorrido por cada línea o ruta de los transportes seleccionados para este ejemplo. Entre los campos de esta tabla se representa el campo de dirección de recorrido que puede tomar tres valores; si su valor es "1", la dirección del servicio va de la estación a la izquierda a la estación de la derecha, si el valor es "2"lo inverso y si es "3", entonces el servicio se presta en ambas direcciones en estaciones asociadas al recorrido. Por ejemplo, en la Línea 2 del Metrobús, aunque su recorrido $Tepalcates \longleftrightarrow Tacubaya$ opere en ambas direcciones, no sería posible establecer

Lines						
Id	**Name**	**SimpleIdentifier**	**AlternativeName**	**Color**	**TransportId**	**Mode**
1	Ruta 7: Circuito Interior - Estadio Olímpico	R7PB	Ruta 7	#A37707	2	3
2	Ruta 8: Circuito Exterior - Estadio Olímpico	R8PB	Ruta 8	#3A53CE	2	3
3	Línea 1: Insurgentes - El Caminero	L1MB	Línea 1	#A37707	1	3
4	Línea 2: Tacubaya - Tepalcates	L2MB	Línea 2	#3A53CE	1	3

Figura 3.10: Tabla con las líneas de transporte seleccionadas

Ruta 7	
1	Ingeniería
2	Arquitectura
3	E8
4	E2
5	Derecho
6	Medicina
7	Química

Ruta 8	
8	Ingeniería
9	Frontones
10	IIMAS
11	Camino Verde
12	Metrobús CU
13	E3
14	MUCA

Figura 3.11: Dos rutas del Pumabús con algunas paradas

Línea 1	
A	El Caminero
B	CU
C	Polifórum
D	Nuevo León
E	Euzkaro
F	Indios Verdes

Línea 2	
G	Tacubaya
H	Antonio Maceo
I	De la Salle
J	Parque Lira
K	Nuevo León
L	Las Américas
M	Río Mayo
N	Del Moral
O	Río Frío
P	Leyes de Reforma
Q	Tepalcates

Figura 3.12: Dos líneas del Metrobús con algunas paradas

PhysicalStations				
Id	**Name**	**Coordinates**	**IsOpen**	**Code**
1	Ingeniería	(lat, lon) válidas	true	1PB
2	Frontones	(lat, lon) válidas	true	2PB
3	IIMAS	(lat, lon) válidas	true	3PB
4	Camino Verde	(lat, lon) válidas	true	4PB
5	Metrobús CU	(lat, lon) válidas	true	12PB
.	.	.	.	.
15	El Caminero	(lat, lon) válidas	true	AMB
16	CU	(lat, lon) válidas	true	BMB
17	Polifórum	(lat, lon) válidas	true	CMB
18	Nuevo León	(lat, lon) válidas	true	DMB
19	Euzkaro	(lat, lon) válidas	true	EMB
20	Indios Verdes	(lat, lon) válidas	true	FMB
21	Tacubaya	(lat, lon) válidas	true	GMB
.	.	.	.	.

Figura 3.13: Tabla con las estaciones seleccionadas para ambos medios de transporte

el valor de dirección a "3", puesto que las estaciones físicas en un recorrido no son las mismas que en el recorrido inverso. Asociada a esta tabla se presenta, en la figura 3.15, la tabla de estaciones lógicas, cuyos elementos hacen referencia a una estación física y son vitales para el proceso de construcción de la gráfica. Con los datos de esta tabla se construyen las relaciones de correspondencia entre estaciones de líneas de cualquier tipo de transporte. En la tabla de la figura 3.16 aparecen, finalmente todas las correspondencias de la red de transporte público, las cuales permiten a los pasajeros, mediante un cambio de andén, abordar vehículos de diferentes recorridos y medios de transporte. Para efectos de este ejemplo, se presenta solamente una tupla en esta última tabla, la cual asocia a dos recorridos presentados en esta sección. En esta tupla se asocia al recorrido $Indios\,Verdes \longleftrightarrow El\,Caminero$ de la Línea 1 del Metrobús con el recorrido $Circuito\,Interior \longrightarrow Estadio\,Olimpico$ del Pumabús mediante dos estaciones físicas pertenencientes a cada recorrido involucrado. En esta tabla existe un campo que permite establecer la distancia al caminar entre ambas estaciones, cuyo valor puede incorporarse a la representación de la gráfica como un componente más del costo calculado para la arista que las uniría. Con esta última tabla se tendrían listos los datos con los que la gráfica habría de ser construida.

Para terminar este ejemplo, en la figura 3.17 se presenta la estructura que tendría la representación de la gráfica en memoria con los datos presentados en este ejemplo. En ella se ejemplifican los datos para dos aristas de distinto tipo y para un vértice tal y como se definió su estructura en la figura 2.7.

VehicleJourneys				
Id	**DestinationDisplay**	**Direction**	**LineId**	**CoordinatesVector**
1	El Caminero - Indios Verdes	3	1	$[(lat, lon), (lat, lon)...]$
2	Tepalcates - Tacubaya	3	1	$[(lat, lon), (lat, lon)...]$
3	Circuito Interior - Estadio Olímpico	1	1	$[(lat, lon), (lat, lon)...]$
4	Circuito Exterior - Estadio Olímpico	1	2	$[(lat, lon), (lat, lon)...]$
5	Col. del Valle - Tepalcates (Línea 2)	3	2	$[(lat, lon), (lat, lon)...]$

Figura 3.14: Tabla de recorridos

LogicalStations			
Id	**VehicleJourneyId**	**PhysicalStationId**	**OrderSequence**
1	3	12	0
.	.	.	.
3	1	15	6
4	1	16	7
5	1	20	11
.	.	.	.

Figura 3.15: Tabla de recorridos

ConnectionLinks			
Id	**LogicalStationId(Origin)**	**LogicalStationId(End)**	**Distance**
1	1	4	10

Figura 3.16: Tabla de recorridos

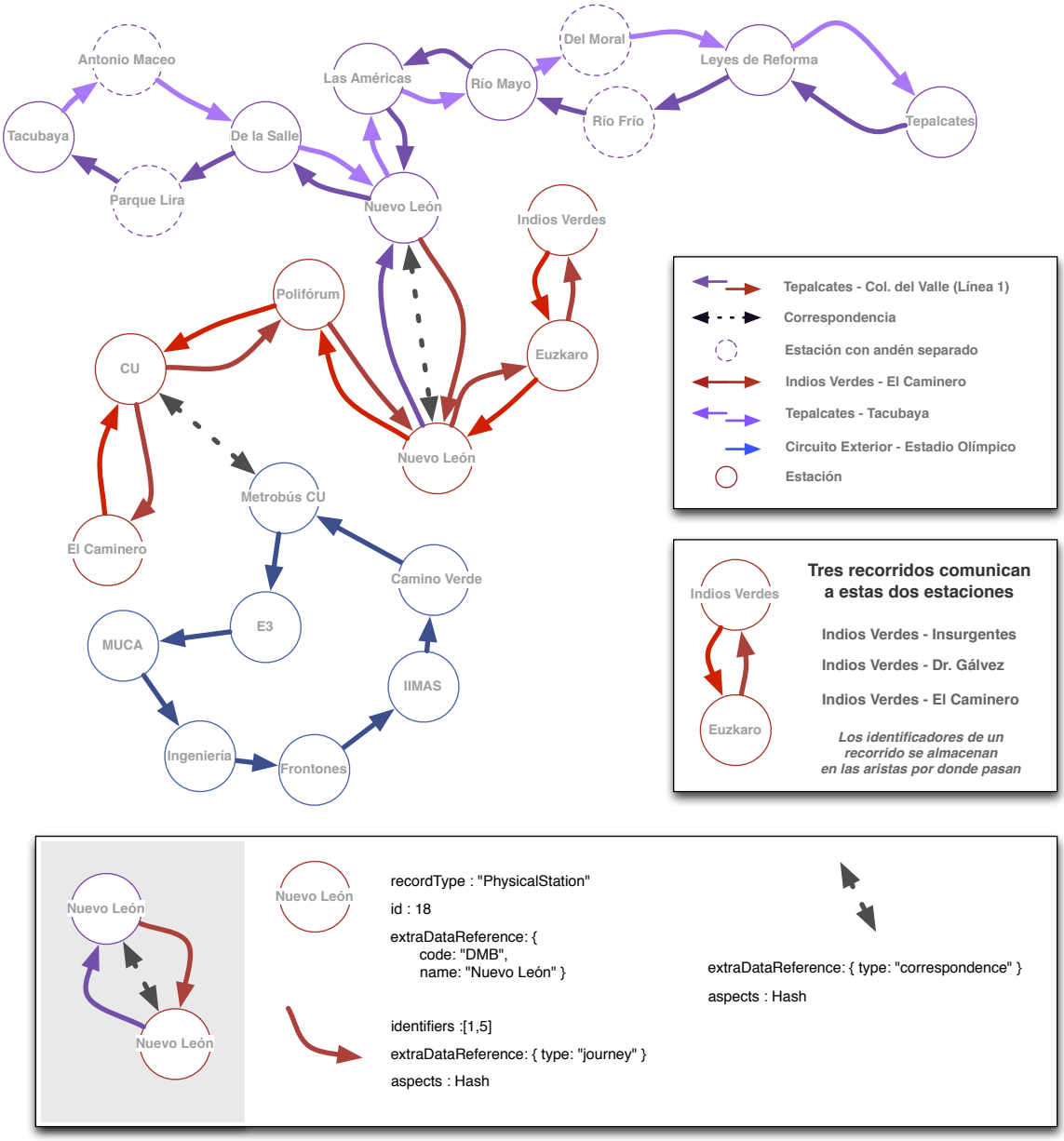

Figura 3.17: Estructura de la Gráfica para el conjunto de datos presentados

3.4. Nuevas tecnologías en hardware

En fechas recientes la compañia Apple presentó una tecnología muy innovadora basada en el estándar de Bluetooth 4.0 a la que le dió el nombre de **iBeacon**. Fabricantes alrededor del mundo han construido sus propias implementaciones de la tecnología que promete ser el bloque constructor del *Internet of things* [22]. Esta tecnología permite a quienes desarrollamos aplicaciones de cómputo móvil, construir un diseño de la interacción basado en eventos definidos por la cercanía de un usuario a un dispositivo con estas características.

Algunas de las principales ventajas que tienen estos dispositivos es que su consumo energético es muy bajo (la batería de cada **iBeacon** puede durar hasta 2 años). No están diseñados para transferir grandes volúmenes de datos entre las partes involucradas, sino, más bien, para funcionar a modo de disparadores de eventos en las aplicaciones que usamos en nuestros teléfonos de cómputo móvil. Adicionalmente, modelos recientes de dispositivos de cómputo móvil que usan sistemas operativos iOS y Android incorporan el estándar Bluetooth 4.0, a diferencia de la tecnología **NFC** [23] que no ha tenido una amplia aceptación en el mercado. Sin duda, esta tecnología abrirá una nueva gama en el mundo de las aplicaciones móviles; para concluir este trabajo, queremos plantear algunas implicaciones que tienen los **iBeacons** para resolver el problema que hemos venido tratando.

Dado que estos dispositivos funcionan a modo de disparadores de eventos en aplicaciones móviles, es muy factible darles uso en el proceso de recolección de datos del transporte público en tiempo real. Simplemente, bastaría con que cada usuario de un dispositivo de cómputo móvil estuviera dispuesto a compartir su ubicación geográfica cada vez que una o varias de sus aplicaciones (previamente registradas a un conjunto de **iBeacons**) recibiera un evento de un **iBeacon** en un radio cercano. De esta forma se tendría un mecanismo para la generación de datos del transporte público casi automática y que estaría realmente fundada en sus propios usuarios como elementos generadores y consumidores de datos.

Con una correcta implementación y difusión de los términos del servicio, las agencias de transporte público que cuentan con un servicio de GPS para el rastreo de sus unidades podrían incrementar la confiabilidad de su información, mientras que las que aún no cuentan con esta tecnología, podrían mejorar la confiabilidad que tienen sus usuarios en el servicio de transporte público que ofrecen.

[22] Ver *El internet de las cosas* en el Glosario
[23] Ver *NFC* en el Glosario

Conclusiones

A lo largo de este trabajo se desarrolló una propuesta de sistema para mejorar la experiencia de los pasajeros del transporte público mediante el acceso a información en tiempo real con la que los pasajeros podrían tomar mejores decisiones antes y durante sus recorridos por la ciudad.

Esta propuesta considera el estado en el que se encuentran las tecnologías computacionales en el transporte público de ciudades latinoamericanas, respecto al de otras ciudades en países europeos principalmente. Se analizaron los sistemas y tecnologías que han funcionado mejor en aquellas ciudades y se hizo un análisis de los dos estándares más robustos que definen la estructuración de datos del transporte público.

Finalmente, se planteo una arquitectura simple de un sistema cuyo funcionalidad principal es generar rutas óptimas entre dos puntos usando datos en tiempo real que pueden provenir de dos fuentes principales; sensores y los mismos pasajeros. Dicha arquitectura se esbozó de forma que pueda representar la red de transporte público de cualquier ciudad sin mayores cambios.

Sin duda, un sistema con estas características sería de gran beneficio para los habitantes de ciudades grandes, donde el transporte público es en muchas ocasiones es impredecible.

Un aspecto reelevante en este planteamiento, es el tema del *crowdsourcing*, que permitiría a los usuarios, además de consultar información, actuar como sensores humanos. Cada reporte hecho por los usuarios del sistema, contendría datos útiles que el sistema podría usar la próxima vez que un pasajero le preguntará por la ruta óptima entre dos ubicaciones.

Otro aspecto interesante, es la posibilidad de que a través de un API, otros sistemas puedan ser fuentes de datos para este. En esto están implícitos los retos de diseñar una intefaz de usuario simple para todo cliente (web o móvil) que quiera actuar como un puente entre el usuario y el sistema, tanto para generar reportes como para consultar rutas o el estado de la red de transporte público.

En este trabajo se procuraron buenas prácticas, tanto en el proceso de diseño como en el de desarrollo, siendo las principales tener claridad respecto a los problemas que la propuesta busca solucionar y los retos implícitos en la construcción de un modelo que los solucione. Generar una propuesta medianamente robusta para la solución de este problema requirió experiencia y conocimiento en muchas de las áreas de las Ciencias de la Computación,

así como de las herramientas disponibles hoy en día para resolver la clase de problemas que este tipo de sistema plantea.

Finalmente, se tuvo la posibilidad de experimentar de primera mano con datos en tiempo real del transporte interno universitario Pumabús a partir de los cuales fue posible construir una plataforma similar a la descrita en este trabajo, mediante la cual los estudiantes pueden conocer la última ubicación de un autobús en cualquiera de las rutas de este sistema de transporte. La implementación de dicho sistema hizo uso del lenguaje de programación y base de datos propuestos en este trabajo. Sin embargo, la arquitectura presentada en este trabajo no fué implementada a totalidad en dicho caso, pues no se requirió (al menos en la primera etapa) generar rutas óptimas de la red de transporte interno.

Bibliografía

[All11] Allegrograph rdfstore. Franz Inc. Web 3.0 database, Julio 2011. [online] http://www.franz.com/agraph/allegrograph/.

[BCE+10] Bast, Carlsson, Eigenwillig, Geisberger, Harrelson, Raychev, and Viger. Fast routing in very large public transportation networks using transfer patterns. *Algorithms - ESA 2010, 18th Annual European Symposium*, pages 290–301, 2010.

[BW89] Burns and Wellings. *Real-time Systems and their programming languages*. Addison-Wesley Publishing Company, 1st edition edition, 1989.

[Cas12] Datamodel - Cassandra Wiki, Junio 2012. [online] http://wiki.apache.org/cassandra/DataModel.

[Cel14] Cellocator, Octubre 2014.

[Chu41] Alonzo Church. The calculi of lambda-conversion. *Princeton: Princeton University Press*, 1941.

[Chu79] West Churchman. *The Design of Inquiring Systems: Basic Concepts of Systems and Organizations*. Basic Books, New York, 1979.

[cit14] Citymapper, Septiembre 2014. [online] http://citymapper.com.

[CL08] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition edition, 2008.

[Cou01] Transport Research Board / National Research Council. *Contracting for Bus and Demand-Responsive Transit Services*. National Academy Press, 2001. Washington, D.C.

[DBP12] Consistency Models in Non-Relational Databases, Junio 2012. [online] http://dbpedias.com/wiki/NoSQL:Consistency_Models_in_Non-Relational_Databases.

[Fou14] Foursquare statistics, Octubre 2014. [online] https://es.foursquare.com/about.

[GDF14] Gdf especification, Septiembre 2014. [online] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=54610.

[Goo13] Google transit. Google Transit, Julio 2013. [online] https://developers.google.com/transit/.

[GTF11] Google transit feed specification (GTFS). Google Transit Feed Specification, Junio 2011. [online] https://developers.google.com/transit/gtfs/.

[GTF14] Gtfs history, Septiembre 2014. [online] http://sf.streetsblog.org/2010/01/05/how-google-and-portlands-trimet-set-the-standard-for-open-transit-data/.

[Hel14] Mapa de helsinki con ubicación de transportes en tiempo real, Septiembre 2014. [online] http://transport.wspgroup.fi/hklkartta/.

[Hil02] David Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society, vol. 8, no. 10*, pages 437–479, 1902.

[INE07] INEGI. Encuesta origen-destino. 2007.

[jsP13] jsplumb, Febrero 2013. [online] http://jsplumbtoolkit.com.

[JUN12] JUNG, Junio 2012. [online] http://jung.sourceforge.net/.

[Ken85] Boulding Kenneth. *The World as a Total System*. Sage Publications, London, 1985.

[Kiz08] Kizoom. A transmodel based xml schema for the google transit feed specification with a gtfs / transmodel comparison. *Kizoom*, 2008.

[Kle36] Stephen Cole Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal (2)*, pages 340–353, 1936.

[Mat93] Yuri Matiyasevich. Hilbert's tenth problem. *MIT Press, Cambridge, Massachusetts*, 1993.

[Neo11a] Neo4j Graph DB. Neo4J Graph Database, Julio 2011. [online] http://neo4j.org/.

[Neo11b] Neo4j Graph Algorithm implementation, Julio 2011. [online] https://github.com/neo4j/community/tree/master/graph-algo.

[nex14] Nextstop, Septiembre 2014. [online] http://citymapper.com.

[NM08] Martin Nicholas Milkovits. Simulating Service Reliability of a High Frequency Bus Route Using Automatically Collected Data. Master's thesis, Massachusetts Institute of Technology, June 2008.

[Ode08] Martin Odersky. *Programming in Scala*. Artima Press, 1st edition edition, 2008.

[Ope11] Sitio del proyecto OpenTripPlanner. Open Source multi-modal trip planner, Febrero 2011. [online] http://opentripplanner.org/.

[Ope12] Opencv, Julio 2012. [online] http://opencv.willowgarage.com/wiki/.

[Ope14] Openstreetmap, Septiembre 2014. [online] http://www.openstreetmap.org.

[Scr14] Scrum guide, Septiembre 2014. [online] http://www.scrumguides.org/scrum-guide.html.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1st edition edition, 1997.

[SK09] Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design, An Introduction*. Morgan Kaufmann, 1st edition edition, 2009.

[Sky05] Lars Skyttner. *General Systems Theory: Problems, Perspectives, Practice*. World Scientific Co. Pte. Ltd., 2nd edition edition, 2005.

[Spr13] Springy.js. A force directed graph layout algorithm in JavaScript, Febrero 2013. [online] https://github.com/dhotson/springy.

[Sta11] Amit's A* pages, Septiembre 2011. [online] http://theory.stanford.edu/~amitp/GameProgramming/.

[TP07] Oncel Tuzel and Meer Porikli. Human detection via classification on riemannian manifolds. *Conference on Computer Vision and Pattern Recognition*, 2007.

[Tra14] Transmodel, Septiembre 2014. [online] http://www.transmodel.org/en/cadre1.html.

[Tur37] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, pages 230–265, 1937.

[Twi14] Twitter daily statistics, Octubre 2014. [online] https://about.twitter.com/company.

[Via11] Generación de rutas óptimas multi-modales en el df, Febrero 2011. [online] http://www.viadf.com.mx/.

[W3C12] World Wide Web Consortium (W3C), Junio 2012. [online] http://www.w3.org/.

[Wei71] Paul Weiss. *Hierarchically Organized Systems in Theory and Practice*. Hafner, 1971.

[Wil] Nigel Wilson. The Role of Information Technology in Improving Transit Systems. Santiago Keynote Lecture.

[WZR09] Nigel H.M Wilson, Zhao, and Rahbee. The Potential Impact of Automated Data Collection Systems on Urban Public Transport Planning. In *Schedule-Based Modeling of Transportation Networks*, volume 46. Operations Research/Computer Science Interfaces Series, 2009.

Parte III

Apéndices

Apéndice A

Sistemas formales y sistemas de información

La palabra sistema, con raíces griegas, denota un todo conexo o regular. El concepto que tenemos actualmente de sistema hubiera sido una manera de referirse a un todo o a una unión de cosas en tiempos de Platón, Euclides y Aristóteles. Personajes de distintas áreas del conocimiento han dado a lo largo de la historia definiciones de sistema. Por ejemplo el biólogo Weiss considera que un *sistema es cualquier cosa lo suficientemente unitaria como para merecer un nombre* [Wei71]; el economista Boulding considera que un *sistema es cualquier cosa que no es caótica* [Ken85]; mientras que para el filósofo Churchman un *sistema es una estructura que tiene componentes organizados* [Chu79].

El diccionario de la Real Academia Española define sistema como un conjunto de cosas que relacionadas entre sí ordenadamente contribuyen a determinado objetivo. El *Standard Dictionary of Electrical and Electronic Terms* de IEEE define sistema como la combinación de componentes que actúan de manera conjunta para realizar una función que ninguno de esos componentes podría realizar de manera independiente. Por otro lado, una definición propuesta por la teoría general de sistemas define un sistema como *un todo organizado en el cual hay partes relacionadas entre sí, lo cual genera propiedades emergentes que le otorgan a ese todo un propósito.*

Un sistema en sí mismo es siempre una abstracción seleccionada con base en aspectos estructurales o funcionales. Tal abstracción puede estar asociada, pero no identificada, con un ente físico; y a las relaciones entre sus elementos se les debe dar tanta importancia como a los elementos mismos.

La teoría general de sistemas supone que muchos sistemas existen dentro de un entorno, el cual además puede ser considerado como un supersistema. Este entorno posee otros sistemas contenidos en él y puede afectar de manera indirecta o directa el funcionamiento de sus subsistemas, pero no puede ser controlado por ellos. Es entonces cuando se puede empezar a hablar de una jerarquía de sistemas y de una descomposición recursiva de sistemas como

una manera de analizarlos desde diferentes puntos de vista. Esta organización jerárquica por ejemplo, puede encontrarse en la naturaleza inorgánica, en la vida orgánica y social así como en el cosmos mismo.

La teoría general de sistemas pone a consideración las interacciones e interrelaciones entre diferentes partes de los problemas asociados a sistemas y emplea una metodología compuesta de pasos racionales y bien organizados, tomando en cuenta alternativas y perspectivas variadas.

La ingeniería de un sistema, considerada como una metodología formal desde el punto de vista de la teoría general de los sistemas, propone cuatro etapas para su efectivo desarrollo e implantación.

- Análisis de Sistemas
  - Reconocimiento y formulación del problema.
  - Organización del equipo de trabajo del proyecto.
  - Definición del sistema.
  - Definición de como encaja el sistema dentro de su supersistema.
  - Definición de objetivos del supersistema.
  - Definición de objetivos del sistema.
  - Definición del criterio económico.
  - Recolección de datos e información.
- Diseño de Sistemas
  - Planeación a mediano y largo plazo.
  - Construcción de un modelo y simulación.
  - Optimización.
  - Evaluación de la solución.
- Implementación
  - Documentación.
  - Construcción.
- Operación
  - Operación inicial y seguimiento de operaciones.
  - Evaluación y mejora en retrospectiva del sistema.

Las etapas anteriores y otros principios propuestos por la teoría general de sistemas fueron incorporados desde la década de los 80 en el proceso de desarrollo de sistemas de software.

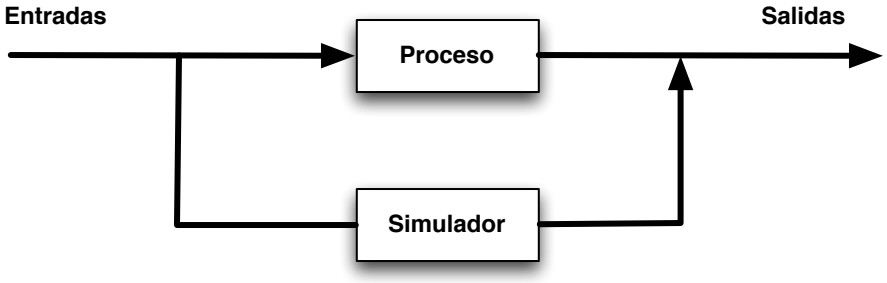

Figura A.1: Sistema de ciclo cerrado

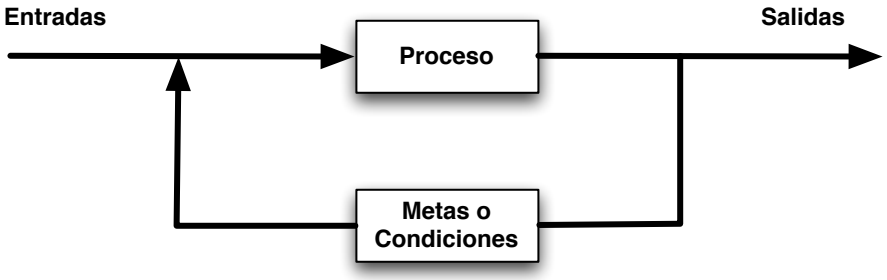

Figura A.2: Sistema de ciclo cerrado por retroalimentación

La cibernética, definida por el matemático Wiener, es un área de estudio dentro de la teoría general de sistemas cuyo objetivo principal es el analizar, entender y definir los procesos y funciones de los sistemas que participan en cadenas causales circulares.

Desde el marco de referencia de la cibernética, un sistema está compuesto de *componentes estructurales* (partes estáticas), *componentes operativos*, que llevan a cargo el procesamiento, y *componentes de flujo*, que pueden ser matería, energía o información siendo procesada, siendo estas últimas la matería prima de las *funciones* o procesos primordiales de un sistema: entrada, transformación y salida. Dichas funciones tomadas en conjunto son generalmente denominadas *throughput* , refiriéndose al volumen de matería, energía o información que fluye a través de un sistema.

Finalmente, dentro de la cibernética se considera el factor de retroalimentación que puede estar presente en muchos sistemas, lo cual define su grado de auto-regulación. Los procesos

Figura A.3: Sistema de ciclo abierto

de un sistema pueden estar o no auto-regulados por el propio sistema. En el primer caso (sistemas de arco cerrado *closed-loop system*), la salida del sistema está ligada a su entrada (Figuras A.1 y A.2), mientras que para el segundo caso (sistemas de arco abierto *open-loop system*), la salida no está conectada directamente a la entrada (Figura A.3).

A.1. Enfoque formal de sistema

Como científicos generalmente se busca una descripción un tanto más formal de qué es un sistema. Para ello se propone definir un *modelo* de un sistema genérico que permita describir su comportamiento.

Como parte del proceso se definirá un conjunto de *variables medibles* asociadas al sistema, las cuales se pueden medir durante periodos de tiempo $[t_0, t_k]$ con el fin de recolectar datos. Suponiendo que es posible hacer que cada variable cambie a lo largo del tiempo, se determina un primer subconjunto de funciones de tiempo al que se llamará *variables de entrada*:

$$\{u_1(t), \ldots, u_p(t)\}, \quad t_0 \leq t \leq t_f$$

A partir de aplicar variaciones a $u_i(t)$ se puede definir un conjunto de variables medibles de manera directa. A dicho conjunto se le denomina *variables de salida* :

$$\{s_1(t), \ldots, s_m(t)\}, \quad t_0 \leq t \leq t_f$$

Los conjuntos de variables de entrada y de salida pueden pensarse como conjuntos de estímulo y respuesta, pudiéndose dar el caso en el que no haya variables suprimidas, las cuales no están relacionadas ni con la entrada ni con la salida.

Para terminar de definir el modelo es conveniente declarar que existe una relación matemática entre ambos conjuntos (entrada y salida). Expresando entonces el conjunto de entrada como $\mathbf{u_t}$ y el de salida como $\mathbf{s_t}$ se tiene que:

$$\mathbf{u}(t) = [u_1(t), \ldots, u_p(t)] \qquad \mathbf{s}(t) = [s_1(t), \ldots, s_m(t)]$$

Suponiendo como válidas las funciones siguientes

$$s_1(t) = g_1(u_1(t), \ldots, u_p(t)), \ldots, s_m(t) = g_m(u_1(t), \ldots, u_p(t))$$

se obtiene finalmente un modelo matemático de sistema

$$\mathbf{s} = \mathbf{g}(\mathbf{u}) = [g_1(u_1(t), \ldots, u_p(t)), \ldots, g_m(u_1(t), \ldots, u_p(t))]$$

Es importante hacer notar que para cualquier sistema es posible obtener un modelo; sin embargo, construir un sistema a partir de un modelo no es siempre realizable.

Definición. El *estado* de un sistema en el tiempo, denotado por t_0 es la información requerida en t_0 tal que la salida $\mathbf{s}(t)$ para todo $t \geq t_0$, se determina de manera única a partir de esta información y de $\mathbf{u}(t), t \geq t_0$.

Dicho de otra manera, el estado de un sistema en un tiempo dado es la información requerida para determinar de manera única la salida del sistema en un instante de tiempo posterior.

Como la entrada y la salida, el estado de un sistema puede ser representado a través de un vector $\mathbf{x}(t)$ cuyos componentes son *variables de estado.*

Con la introducción del concepto de estado resulta aparente que el proceso de modelado de un sistema consiste en determinar las relaciones matemáticas entre la entrada $\mathbf{u}(t)$, salida $\mathbf{s}(t)$ y el estado $\mathbf{x}(t)$, las cuales generalmente son conceptualizadas como la *dinámica* de un sistema.

Definición. El *espacio de estados* de un sistema, denotado como X, es el conjunto de todos los posibles valores que un estado puede tomar.

A continuación se presentan diferentes tipos de sistemas y sus características principales.

Sistemas dinámicos y estáticos

Un sistema se considera *estático* si la salida s_t es independiente de valores anteriores de la entrada $u(t'), t' < t$, para todo t, mientras que es *dinámico* si la salida **sí** depende de valores anteriores de la entrada. Para el caso particular de los sistemas dinámicos entonces es necesario tener un mecanismo que actúe como memoria de almacenamiento de la entrada.

Sistemas dinámicos y el tiempo

En el modelo definido antes aparece la función $\mathbf{g}$ que actúa de manera independiente del tiempo (sistema *invariante respecto al tiempo*), pero si la relación entre la entrada, salida y el tiempo se replantea en el modelo como

$$\mathbf{s} = \mathbf{g}(\mathbf{u}, t)$$

haciendo que $\mathbf{g}$ dependa del tiempo t, se tendría como resultado un sistema *variante respecto al tiempo.* Un sistema se dice ser invariante respecto al tiempo si la función de entrada aplicada al sistema t' unidades de tiempo más tarde que t arroja una función de salida idéntica a la obtenida en el tiempo t.

Sistemas de estados contínuos y de estados discretos

Los sistemas pueden ser clasificados según la naturaleza del espacio de estados asociado al modelo. En sistemas de *estados continuos* el espacio de estados X consiste de todos los vectores n-dimensionales de números reales o a veces complejos. Generalmente, son de dimensión finita, aunque sí hay casos donde X es dimensionalmente infinita. En los sistemas de *estados discretos* el espacio de estados es un conjunto discreto y los cambios en sus variables de estado son igualmente discretos.

Es interesante mencionar que existen sistemas que pueden considerarse como *híbridos* en el sentido que algunas variables de estado pueden ser discretas y otras pueden ser contínuas.

Sistemas deterministas y estocásticos

Un sistema es *estocástico* si al menos una de sus variables de salida es una variable aleatoria; en caso contrario el sistema es *determinista*. Generalmente, el estado de un sistema estocástico dinámico resulta de un proceso aleatorio, cuyo comportamiento puede ser solamente descrito mediante métodos probabilísticos.

Por tanto, mientras que en un sistema estocástico su estado en t es un vector aleatorio, pudiendo ser evaluada únicamente su función probabilística; en un sistema determinista con entrada $\mathbf{u}(t)$ dada para toda $t \geq t_0$, el estado x_t puede ser evaluado directamente.

Existen sistemas en los cuales el espacio de estados es un conjunto discreto y las transiciones entre estados son puntos discretos en el tiempo. Esas transiciones de estado son generalmente consideradas eventos. Este tipo de sistemas son conocidos como sistemas de eventos discretos. Un evento es generalmente el resultado de un conjunto de condiciones que se cumplieron en algún momento determinado. Se denotará un evento cualquiera por la letra e y será asociado a un conjunto de eventos E en los casos de sistemas que pueden ser afectados por diferentes tipos de eventos.

Sistemas dirigidos por eventos y por el tiempo

Tomando en cuenta todo lo anterior se puede hacer una última clasificación en los tipos de sistemas, esta vez considerando los factores que inducen en ellos cambios en sus variables de estado. Los sistemas *dirigidos por el tiempo* están cambiando de manera constante a lo largo del tiempo, pues sus variables de estado continuas también lo hacen. De manera contraria; en un sistema *dirigido por eventos* el estado cambia únicamente en ciertos puntos del tiempo y dicho cambio es producido por transiciones instantáneas que responden a un conjunto de condiciones que se cumplen.

Así, es posible resaltar que en los sistemas dirigidos por el tiempo las transiciones están *sincronizadas* con el reloj y éste es el único responsable de cualquier transición de estado posible, mientras que en los sistemas dirigidos por eventos cada evento $e \in E$ define una serie

de procesos distintos que determinan el instante de tiempo t en el que e ocurre, pudiendo haber eventos que ocurren de manera *concurrente* y *asíncrona*.
Es evidente que los sistemas dirigidos por eventos son mucho más difíciles de analizar y modelar; el tiempo exacto en el que un evento ocurre en este tipo de sistemas depende de una serie de factores que en muchos casos puede ser difícil, si no imposible de calcular.

A.2. Sistemas de información

Al inicio de este apéndice se habló sobre la materia prima de la que está compuesto un sistema de información; esto es la información misma con la que opera y los algoritmos que trabajan sobre ella. El concepto de algoritmo, dicho sea de paso, es fundamental en las ciencias de la computación.

A.2.1. Algoritmo

De manera informal, un *algoritmo* es una colección de instrucciones concretas para llevar a cabo una tarea. Un algoritmo toma un conjunto de valores como entrada y produce un conjunto de valores como salida. Por tanto, un algoritmo es una secuencia de pasos computacionales que transforman la entrada en la salida. Muchas veces se compara un algoritmo con una receta de cocina, pues ambos describen de manera concisa pasos ordenados requeridos para lograr un objetivo definido.

Un algoritmo puede ser visto también como una herramienta para resolver un problema computacional bien especificado. Mientras que el problema computacional describe en términos generales las características requeridas tanto en el conjunto de valores de entrada como en el de salida y la relación entre ambos, el algoritmo describe un procedimiento computacional específico para dicho problema computacional particular.

Un requisito importante para cualquier algoritmo es que termine cuando ha computado un resultado que satisface el problema computacional, en cuyo caso se dice que es un algoritmo **correcto** y **eficaz**, sin embargo, en otras áreas como análisis numérico, se obliga a que los procesos tengan condiciones de terminación; ya que teóricamente alcanzan la solución óptima, pero computacionalmente no siempre es posible.

En el año 1900, el matemático alemán Hilbert propuso veintitrés problemas matemáticos como retos para el siglo siguiente, donde el décimo tenía que ver con algoritmos [Hil02]. El problema tenía que ver con encontrar un algoritmo que probara si un polinomio tiene una raíz entera. Hilbert supuso de manera intuitiva que un algoritmo para solucionar ese problema existía; sin embargo, hoy sabemos que no hay un algoritmo para dicho problema, pues carece de una solución algorítmica.

Al no existir en esos años una definición clara de algoritmo, los matemáticos de ese tiempo usaron su definición intuitiva de algoritmo no pudiendo llegar a una conclusión puntual sobre

la existencia o no de un algoritmo que resolviera dicho problema computacional.

A mediados del siglo pasado los matemáticos Church, Turing y Kleene llegaron a definiciones formales de algoritmo; Church creó un método para definir funciones llamado cálculo lambda [Chu41], Turing un modelo teórico de una máquina actualmente conocida como *máquina de Turing* que podría realizar cálculos a partir de entradas [Tur37], y Kleene junto con Church y el lógico Rosser crearon una definición formal de una clase de funciones cuyos valores pueden ser calculados mediante recursion [Kle36].

La conexión entre la noción informal de algoritmo y la definición precisa se conoce como la **tesis de Church-Turing**. En dicha tesis se afirma que si algún algoritmo existe para llevar a cabo un cálculo, entonces ese mismo cálculo puede ser realizado por una máquina de Turing, o bien por una función recursivamente definible o bien por una función lambda. Esta definición fue crucial para que Matijasevich en 1970 esbozara la solución [Mat93] al problema décimo propuesto por Hilbert, aunque no existe un algoritmo computacionalmente.

A partir de esa definición se han obtenido más resultados que son herramientas para probar si éste y otros problemas computacionales tienen una solución algorítmica.

Aunque profundizar en este tema no está dentro del alcance de este trabajo, es importante incluir algunos de sus conceptos y resultados más importantes por varias razones. Saber que un problema carece de una solución algorítmica nos lleva a replantear o restringir el problema de manera que pueda tener una solución algorítmica. Es posible apreciar de manera más clara los límites de la computación al igual que sus capacidades. Así mismo, se cuenta con mayor formalidad en el proceso de diseño de sistemas que usan algoritmos que sí tienen una solución algorítmica. Finalmente, tales conceptos y resultados resultan útiles para explicar de una manera sencilla por qué las computadoras no pueden realizar cualquier tarea, como mucha gente piensa.

A.2.2. Información

El concepto de información tiene que ver con cómo describimos un objeto y mide qué tan precisa o ambigua es la caracterización que damos del objeto para después poder recrearlo a partir de ella. La cantidad de información de un objeto corresponde al tamaño de la descripción *más* breve y completa del objeto.

En 1965 el matemático Kolmogorov contribuyó en las áreas de complejidad computacional y teoría de la información con su definición de complejidad descriptiva (o algorítmica) de un objeto. Definió la complejidad algorítmica de un objeto como la longitud más pequeña de un programa computacional que lo describa.

Complementando lo anterior, la descripción de un objeto (información) es una secuencia ordenada de símbolos que almacena o transmite un mensaje. El concepto de **entropía**, muy relacionado al de información y al de complejidad, se utiliza para medir la cantidad de bits necesarios para almacenar o transmitir la representación de un objeto. Un objeto se dice que tiene baja entropía si requiere de un número pequeño de bits para ser representado.

A.2.3. Sistema de información

Un sistema de información tiene como fin principal asistir a personas en el manejo de la información; ya sea dentro de un ente corporativo, gubernamental o civil, el cual puede manejar información meramente privada; es decir, que es accesible solamanente por los miembros que pertenecen a él o bien maneja información que es de acceso público a diferentes escalas. Lo que nunca cambia en ninguno de ellos es el significado de manejo de información, el cual puede englobarse en cuatro aspectos: entrada de datos, almacenamiento de datos, procesamiento de datos y transmisión de datos.

Antes de avanzar más, conviene hacer explícita la diferencia entre datos e información. En el contexto de la sección anterior podríamos decir que datos e información son prácticamente lo mismo, sin embargo, los datos no siempre tienen un significado claro para un observador humano, pero siempre lo tienen para un sistema de información, los datos se convierten en información una vez que han sido procesados y desplegados a un observador humano que sea capaz de comprenderlos e interpretarlos.

Las primeras computadoras estaban limitadas a únicamente recibir un conjunto de datos de entrada y devolver como respuesta un conjunto de datos de salida. Esto era llevado a cabo mediante procedimientos largos y complejos de transformación de datos usando algún lenguaje como APL o Fortran. No había posibilidades de almacenar ninguno de los dos conjuntos de datos, mucho menos de analizar y modificar esos datos de manera interna haciendo uso de algún soporte de almacenamiento.

Actualmente, los sistemas de información pueden clasificarse en tres grandes grupos según el tipo de plataforma en la que operen y su contexto de uso; sistemas de escritorio (*desktop*), sistemas web y sistemas embebidos[1]. Este trabajo se enfoca más a sistemas web y a sistemas embebidos, dentro de los cuales existe una clasificación que engloba a los sistemas de cómputo móvil que se pueden encontrar en diferentes dispositivos como por ejemplo teléfonos que tienen características adicionales al recibo de llamadas y envío de mensajes de texto.

Los sistemas web actuales están compuestos por uno o más medios de almacenamiento de datos que pueden estar distribuidos en diferentes computadoras; componentes que se encargan de manipular los datos asi como también de presentarlos al usuario (interfaces de usuario) y componentes que exponen funciones como servicios a otros sistemas con el fin de intercambiar información. Un sistema web require para funcionar de un componente de software denominado servidor que se encarga de mantener al sistema web en funcionamiento, asignándole un puerto disponible en el sistema, de modo que cualquier petición dirigida desde un sistema remoto o local a la dirección IP de la computadora en la que éste se está ejecutando pueda ser atendida y resuelta.

En los últimos años hemos visto cada vez más como los sistemas web han ido ganando terreno consolidándose como un paradigma de acceso masivo a servicios que van desde sistemas simples como bitácoras personales (*blogs*) hasta sistemas más complejos como redes

[1]Se usará sistemas embebidos como traducción de *embedded systems*

sociales en tiempo real y que hacen uso de los avances más recientes en lo que concierne a las ciencias de la computación y áreas relacionadas.

Apéndice B

Algoritmos de enrutamiento

Se detallarán algunos de los algoritmos de enrutamiento que se han mencionado en este trabajo y que representan la pieza clave para resolver el problema de encontrar la ruta más corta entre dos nodos de manera eficiente. Durante gran parte de este trabajo se desarrolló una representación de la red de transporte público en una gráfica dirigida, pues es la estructura de datos ideal sobre la cual operan estos algoritmos, los que consideran una función de peso para cada arista en la gráfica, la cual, en su versión más simple puede tomar el valor de 1, la distancia entre un nodo y otro. Dicho valor por lo general es un número real.

Una gráfica puede tener uno o más caminos, que son listas de vértices y aristas ordenadas. Cada camino C en una gráfica tiene un peso dado por:

$$w(C) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$$

La ruta más corta entre dos nodos u y v en una gráfica está dada por el camino C entre u y v para el cual su peso mínimo está dado por $\delta(u, v) = \text{mín}\{w(C)\}$. Puede darse el caso de que existan aristas con valores negativos en una gráfica y que el valor $\delta(u, v)$ sea un valor negativo. Un escenario interesante resulta si en esa gráfica existe un ciclo que al ser recorrido cada vez decrementa el peso acumulado de la ruta, en tal caso, el peso de la ruta más corta para esa gráfica en particular estaría definida como $\delta(u, v) = -\infty$. Naturalmente, puede existir alguna gráfica para la cual no exista una ruta más corta entre dos de sus nodos debido a que no existan aristas que los conecten, en cuyo caso el peso de la ruta más corta estaría definida como $\delta(u, v) = \infty$. En el caso del algoritmo de Dijkstra y otros algoritmos de enrutamiento una de sus precondiciones es que no existan aristas con pesos negativos.

B.1. Recorridos en gráficas

Los dos algoritmos siguientes constituyen la base sobre la cual operan los algoritmos de enrutamiento que se presentarán más adelante, pues explican las dos formas principales para recorrer los nodos de una gráfica. Estos dos algoritmos, tal y como se presentan, suelen ser de ayuda para resolver problemas de búsqueda, verificación y enrutamiento en una gráfica, tales como encontrar un vértice en la gráfica, encontrar un árbol generador, verificar si la gráfica es conexa o si tiene ciclos.

B.1.1. DFS (Depth-First Search)

El algoritmo de búsqueda a profundidad, DFS, recorre todos los nodos alcanzables en una gráfica desde un nodo inicial, uno a la vez. La salida de este algoritmo es una lista ordenada de nodos que representa el orden en el que fueron visitados los nodos de la gráfica. Se caracteriza por que el proceso de descubrir nodos se realiza a profundidad, por lo cual generalmente se utiliza una estructura FIFO como una pila durante la ejecución del algoritmo, aunque otra implementación usual de este algoritmo es recursiva y usa la pila de ejecución.

El costo de recorrer una gráfica usando DFS es $O(n+a)$, donde n es el número de nodos en la gráfica y a el número de aristas. El pseudo-código del algoritmo se presenta a continuación.

Algoritmo 1 DFS

Entrada: Una gráfica G y un nodo inicial o
Salida: Una lista ordenada con los nodos visitados

```
1: S ← [0]
2: push(S, o)
3: marcar o como visitado
4: mientras S no esté vacía hacer
5:    extraer el nodo v en el tope de la pila
6:    para todo nodo w adyacente a v hacer
7:       marcar w, si no ha sido marcado como visitado aún
8:       push(S, w)
9:    fin para
10: fin mientras
```

B.1.2. BFS (Breadth-First Search)

El algoritmo de búsqueda por amplitud, BFS, al igual que DFS, parte de un nodo inicial recorriendo nodos alcanzables desde éste, pero el proceso de marcar nodos como visitados se realiza en amplitud, esto es, añadiendo grupos de nodos adyacentes a distancia i empezando en $i = 1$. Este algoritmo es uno de los más importantes para encontrar soluciones óptimas a

un conjunto de problemas que pueden ser representados por una gráfica y cuya solución se reduce a resolver el problema de las rutas más cortas. A continuación se presenta el pseudo-código de este algoritmo.

Algoritmo 2 BFS

Entrada: Una gráfica G y un nodo inicial o
Salida: Una lista ordenada con los nodos visitados

```
1: Q ← [0]
2: push(Q, o)
3: marcar o como visitado
4: mientras Q no esté vacía hacer
5:    seleccionar algún nodo v del frente de Q
6:    para todo vecino w del nodo v que no haya sido marcado aún hacer
7:       marcar w como visitado
8:       push(Q, w)
9:    fin para
10: fin mientras
```

Como se mencionó, la diferencia entre ambos algoritmos es el mecanismo en el que recorren los nodos de una gráfica, pero cabe señalar que en la implementación de cada uno de ellos, lo que los diferencia es la estructura de almacenamiento temporal para los vértices que van siendo marcados; mientras que para BFS la estructura que se usa es de tipo FIFO, para DFS es de tipo LIFO.

B.2. Rutas más cortas

En esta sección revisaremos los algoritmos de Dijkstra y A* para resolver el problema de la ruta más corta, el último siendo una extensión del primero que logra alcanzar un mejor rendimiento respecto al tiempo al usar heurísticas.

B.2.1. Dijkstra

El algoritmo de Dijkstra encuentra la ruta más corta en una gráfica desde un nodo origen hacia los demás nodos. Solamente funciona sobre gráficas cuyas aristas están etiquetadas con valores positivos. El valor de cada arista es el elemento clave para que algoritmo pueda determinar cuál es el valor mínimo acumulado hasta el nodo actual, en cada iteración y entregar una ruta de peso mínimo. En la siguiente página se presenta el pseudo-código del algoritmo de Dijkstra.

Algoritmo 3 Dijkstra

Entrada: Una gráfica G y un nodo inicial o

Salida: Una lista con las distancias de cada vertice respecto al nodo origen

```
1: Inicializar una cola Q de prioridades
2: Asignar ∞ al valor de distancia de todos los vértices de G
3: Q ← {o}
4: mientras Q tenga nodos sin visitar hacer
5:    seleccionar algún nodo v del frente de Q
6:    marcar v como visitado en Q
7:    para todo nodo u adyacente a v hacer
8:       w ← weight(u, v)
9:       Si distance(v) + w < distance(v)
10:      entonces el nuevo valor de distancia para u es distance(v) + w
11:      añadir u a Q
12:   fin para
13: fin mientras
```

B.2.2. Algoritmo A*

El algoritmo A* es un algoritmo de búsqueda en gráficas que mediante una heurística determina el nodo más cercano al nodo destino en cada una de sus iteraciones. Para cada nodo se asigna un valor estimado, el cual está dado por una función $f(n)$ que corresponde a la suma de dos funciones $g(n)$ y $h(n)$, donde $g(n)$ representa el costo para llegar al nodo actual, mientras que $h(n)$ representa el costo estimado para llegar al nodo destino desde el nodo actual. Por tanto, en cada iteración A* cuenta con dos componentes de información que lo dirigen: el nodo cuya distancia al nodo origen es menor y el nodo siguiente cuya distancia al nodo destino es menor. Este algoritmo integra el primer componente presente en el algoritmo de Dijkstra y el segundo componente basado en la heurística, que usa el algoritmo *Greddy Best-first search*. Este último algoritmo estima, mediante una heurística que tan cercano del nodo destino está el último nodo de una ruta; aquellos más cercanos son los primeros en ser explorados.

La heurística seleccionada impacta el comportamiento del algoritmo A*, por lo que es importante seleccionar aquella que se ajuste mejor a las características específicas de la gráfica con la que se está trabajando y al problema que ésta representa. Por otro lado, si se opta por no usar ninguna heurística, este algoritmo sería igual al de Dijkstra. Para este algoritmo no se presenta una descripción, dado que la diferencia respecto a Dijkstra recae en la heurística seleccionada. El siguiente análisis puede resultar interesante para comprender mejor el papel que juega la heurística en este algoritmo:

- Si $h(n) = 0$, entonces $g(n)$ es quien dirige el algoritmo, lo cual equivale a usar Dijkstra.
- Si $h(n)$ es siempre menor o igual al costo de ir desde cualquier nodo al nodo destino, entonces A* encontrará una ruta más corta.

- Entre más pequeño es el valor de $h(n)$, A* se expande más, lo que impacta en su eficiencia haciéndolo más lento.
- Si $h(n)$ es igual al costo de ir desde cualquier nodo hasta el nodo destino, entonces A* seguirá solamente por el mejor camino sin expandir otros, lo cual lo hace muy eficiente.
- Si $h(n)$ es mayor al costo de ir desde cualquier nodo hasta el nodo destino, entonces A* puede no encontrar la ruta más corta.

Apéndice C

Mecanismos de persistencia de datos

La persistencia de datos en un sistema es equivalente a poner la primera piedra en una construcción; pues es este componente, que, aunado a una buena arquitectura del sistema, define los módulos de abstracción para la representación y acceso a los datos. Hoy en día se cuenta con diversas tecnologías para el almacenamiento de datos, las cuales pueden agruparse en aquellas que se adhieren al estándar SQL y aquellas que no (NoSQL).

C.1. Basadas en el estándar SQL

El lenguaje SQL (*Structured Query Language*) fue diseñado para facilitar la administración y consulta de datos almacenados en sistemas de bases de datos relacionales. SQL está basado en la lógica de predicados y en la teoría de conjuntos. SQL ha definido un estándar probado a lo largo de más de veinte años para el manejo confiable de datos. Las abstracciones de las que depende una base de datos relacional para almacenar y dar sentido a los datos son:

- Un conjunto de objetos que define un dominio o un tipo de datos. Generalmente conocido como tabla.
- Tuplas o renglones que son elementos de una tabla, los cuales contienen pares ordenados, cada uno representando a un dominio con su respectivo valor.
- Relaciones entre tablas, que no son más que conjuntos de tuplas.
- Reglas que se aplican a un dominio, a las tuplas que definen y a sus relaciones.

Además de la lista anterior hay otro aspecto importante que suele diferenciar a los manejadores de bases de datos basados en el estándar SQL respecto a los que no. En las ciencias de la computación se utiliza el acrónimo *ACID*[1] para referirse a un conjunto de propiedades

[1] ACID es el acrónimo de *Atomicity*, *Consistency*, *Isolation* y *Durability*

que garantizan que las transacciones en una base de datos se efectúan de manera confiable. Una transacción en una base de datos ocurre alrededor de una operación lógica en los datos sin importar que dicha operación los altere o no.

A continuación se detalla cada una de las propiedades a las que se refiere este acrónimo.

- **Atomicidad:** Cualquier transacción se realiza de manera atómica teniendo dos posibles resultados: completada correctamente o no. En el último caso, el estado de los datos posterior a la transacción debe ser igual al estado de los datos previo a la transacción.
- **Consistencia:** Cualquier transacción realizada debe dejar al conjunto de datos en un estado válido. Solo datos válidos son escritos en la base de datos.
- **Aislamiento**[2]**:** Cualquier transacción ocurre de manera independiente a otras, esto es; no afectándolas.
- **Durabilidad:** Cualquier transacción que haya modificado el estado de los datos es permanente.

C.2. No basadas en el estándar SQL: NoSQL

NoSQL es una forma de llamar a un conjunto de sistemas manejadores de bases de datos que no se adhieren al estándar SQL. Este tipo de tecnologías surgen (durante los últimos años) en respuesta al cambio de los patrones (tipo, frecuencia, origen) de los datos transmitidos vía internet, generado en gran parte por la adopción masiva que ha habido en los últimos años de tecnologías de cómputo y acceso a redes de alta velocidad. Este cambio ha impactado la manera en la que servicios populares en internet almacenan y distribuyen datos a sus usuarios.

En los primeros años del nuevo siglo muchos de ellos empezaron a llegar al "límite"de los sistemas manejadores de bases de datos basados en SQL. Google, Facebook y Twitter, entre otros, desarrollaron manejadores de bases de datos que les permitieran adaptarse a la demanda de usuarios actual y futura. En el caso de Twitter, la demanda no estaba limitada únicamente al aspecto de la persistencia de datos, sino que se añadía un factor adicional: el manejo eficiente de una cantidad de datos inmensa, generados en tiempo real.

A continuación se presentan algunas características comunes a los manejadores de bases de datos tipo NoSQL:

- No usan SQL como lenguaje para realizar consultas.
- La mayoría de ellos no dan garantía de todas las propiedades presentes en el acrónimo ACID.

[2]Se usará Aislamiento como traducción literal de *Isolation*

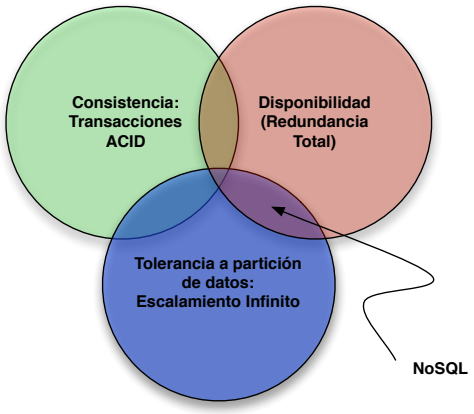

Figura C.1: Diagrama de garantías del Teorema CAP

- Muchos funcionan de manera distribuida y tienen una arquitectura tolerante a fallas.
- El bloque base de abstracción en muchos de ellos está dado por llaves y valores ordenados.

Estos manejadores de bases de datos no dan garantías respecto a lo que enarbola el acrónimo ACID. Por la naturaleza distribuida de la mayoría de estos manejadores de bases de datos, el teorema CAP (también conocido como el teorema de Brewer) les es aplicable. Dicho teorema dice que un sistema de cómputo distribuido solo puede ofrecer dos de las siguientes tres garantías de manera simultánea.

- **Consistencia** (O atomicidad): Todos los nodos ven los mismos datos al mismo tiempo.
- **Disponibilidad:** Una falla en algún nodo no afecta la operatividad del sistema.
- **Tolerancia a partición:** El sistema continúa operando sin importar que haya pérdida de mensajes o un nodo deje de funcionar.

La mayoría de los sistemas manejadores de bases de datos tipo NoSQL cumplen las garantías de disponibilidad y tolerancia a partición como se muestra en la figura C.1. Existe un concepto conocido como *Consistencia Alcanzable*, el cual establece que una base de datos puede tener varios estados de inconsistencia en algún punto del tiempo, pero una vez que todas las modificaciones a sus datos cesen, llegará a un punto en donde será consistente. *BASE (Basically Available Soft-state Eventually)* es un acrónimo usado para comparar las garantías que un sistema de este tipo puede ofrecer respecto a las que *ACID* ofrece.

Muchos de los manejadores de bases de datos que se consideran tipo NoSQL implementan la persistencia de sus datos usando abstracciones y algoritmos novedosos para representar y acceder a los datos, aunque muchas de ellas existen inclusive antes de la aparición del

estándar SQL. A continuación se presentan los sistemas manejadores de bases de datos más representativos:

- **MongoDB:** Almacenamiento orientado a documentos estilo JSON (diccionarios).
- **Apache CouchDB:** Almacenamiento orientado a documentos JSON.
- **Redis:** Almacenamiento basado en llaves y valores. Se le conoce también como un servidor de estructuras de datos.
- **Apache Cassandra:** Implementación abierta de Google BigTable. Almacenamiento basado en tripletas con campos para nombre, valor y fecha. Presenta familias de columnas y super columnas.
- **Neo4J:** Almacenamiento en nodos y aristas, orientado a gráficas.

Dados los alcances de este trabajo, no se ahondará en describirlos, solamente se explicará un poco sobre Neo4J y se contrastará con PostgreSQL y Postgis [3]. De todos los manejadores de bases de datos NoSQL es el que mejor se adapta al problema que plantea este trabajo.

C.2.1. El manejador de BD NoSQL Neo4J

El manejador de base de datos Neo4J hace uso de nodos y aristas para almacenar datos y formar relaciones entre ellos a diferencia de tablas y relaciones en una base de datos relacional. Cada uno de los objetos representados puede tener propiedades, pero igualmente las relaciones entre ellos, simulando el concepto de gráficas con peso que contienen un valor asociado a la relación entre dos nodos.

Una gráfica es una estructura de datos muy versátil, ya que dependiendo de las relaciones entre sus nodos y aristas puede tomar la topología de una lista, de un árbol o de una malla; tal y como se muestra en las Figuras C.3, C.2 y C.4, respectivamente. La una unión de varios de los componentes anteriores facilitan asimismo la representación de conjuntos de datos interrelacionados.

Un aspecto interesante de la implementación del modelo de gráfica de Neo4J es la existencia de índices, los cuales operan sobre propiedades asignadas a nodos y aristas de la gráfica.

Las propiedades están dadas por tuplas compuestas por una llave y un valor que pueden estar asociadas a un nodo o bien a una arista. La llave de una propiedad es de tipo cadena, mientras que su valor puede ser un tipo primitivo o un arreglo de tipos primitivos. Un nulo no es considerado un valor válido para una llave. En otros modelos de bases de datos, que se

[3]Biblioteca con extensiones para soportar datos geoespaciales en el manejador de bases de datos relacional PostgreSQL.

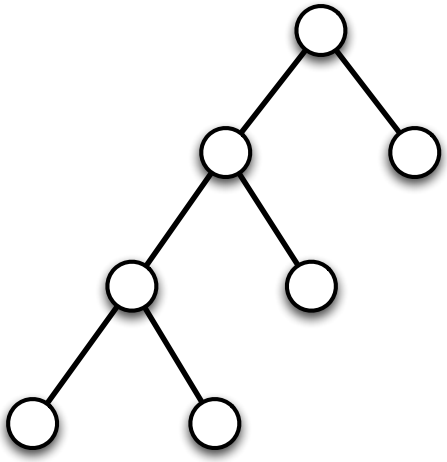

Figura C.2: Un árbol

Figura C.3: Una lista

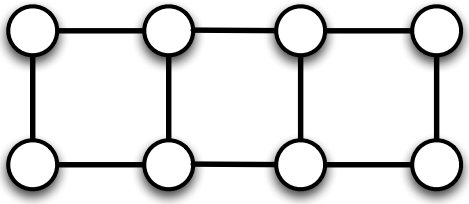

Figura C.4: Una malla

apegan al paradigma NoSQL, este nivel de encapsulamiento de datos es el más elemental. Tal es el caso de MongoDB que trae de vuelta a las bases de datos basadas en documentos, donde cada documento puede contener tuplas de valores y llaves o bien otros documentos que a su vez las contienen. Uno de los beneficios que ofrece Neo4J al momento de trabajar con grupos de datos altamente relacionados es que se puede contar con muchas de las ventajas que tienen las bases de datos basadas en documentos, más la posibilidad de formar relaciones complejas entre ellos. Por su versatilidad, es posible también hacer uso de cada nodo para almacenar familias de renglones y relacionarlas mediante aristas mostrando una relación entre los datos mucho más explícita.

En esta implementación, todas las gráficas tienen aristas dirigidas y según la perspectiva del nodo en el que se esté parado, una arista puede ser saliente o entrante. Sin embargo, la tarea de ir de un nodo a otro puede ser completada sin que la dirección sea un inconveniente (violando parte de la teoría matemática para gráficas dirigidas) al modelado de los datos. Neo4J permite además asignar tipos a las relaciones entre nodos. Finalmente, un nodo puede tener una arista cuyo extremo opuesto sea él mismo, es decir puede estar relacionado consigo mismo.

Ya se habló de los componentes fundamentales y de sus propiedades, los cuales forman parte de la implementación de Neo4J. Falta describir cómo es que Neo4J opera sobre todos esos grupos de nodos y sus relaciones representadas por aristas.

Neo4J cuenta con un proceso que recorre nodos y aristas y que regresa rutas, nodos o bien relaciones. Dicho proceso requiere una descripción algorítmica que considera un componente que se encarga de expandir el alcance del recorrido, considerando la dirección de éste y el tipo de relaciones que van siendo encontradas; uno más que toma en cuenta el orden en el que se visitan los nodos (DFS y BFS)[4]; otro componente que verifica la unicidad en nodos, relaciones y caminos entre nodos; y uno adicional que se encarga de realizar podas en los resultados encontrados evaluando que incorporar y qué descartar. Neo4J implementa diferentes procesos que operan tomando en cuenta dichos componentes, los cuales se ajustan a la definición formal de varios algoritmos de búsqueda en gráficas.

Además de DFS y de BFS, Neo4J cuenta con la implementación del algoritmo de recorridos más cortos propuesto por Dijkstra y del algoritmo A*. Cualquiera de estos dos algoritmos servirá para el propósito central de este trabajo: encontrar rutas óptimas en redes de transporte público.

[4] *Depth First Search* o Búsqueda a profundidad y *Breadth First Search* o Búsqueda a amplitud son dos algoritmos de búsqueda en gráficas descritos en el Apéndice B.

Apéndice D

Scala, un lenguaje funcional orientado a objetos

Scala recibe su nombre de *Scalable Language*. Es un lenguaje que se ejecuta sobre Java y, por lo cual, puede hacer uso de sus bibliotecas. Scala incorpora programación orientada a objetos y muchos aspectos de programación funcional. Ambos aspectos reunidos hacen de Scala un lenguaje que facilita la modularización de código y en el cual el código suele ser muy conciso. Adicionalmente, es un lenguaje que resulta una muy buena opción para aplicaciones concurrentes debido a que incorpora un modelo de concurrencia basado en actores y mensajes.

A continuación se discuten algunos de sus conceptos más útiles y novedosos.

D.1. Traits

En Scala, un `trait` es un mecanismo que facilita la reutilización de código. A diferencia del mecanismo de herencia usando en Java por ejemplo (no herencia múltiple como en C++), una clase puede mezclar cualquier número de traits. A continuación se muestra cómo se declara un `trait`.

```
trait Pavimento {
        def circulan() {
                println("Vehiculos")
        }
}
```

```
trait Banqueta {
        def transitan() {
                println("Peatones")
        }
```

```
}
```

Se puede decir de manera genérica que una calle tiene (o debiera tener) dos grupos de usuarios: peatones y vehículos (autos, buses, bicicletas, otros). La clase que se define a continuación corresponde a una calle donde transitan peatones y circulan vehículos. Dicha clase ha mezclado los dos traits definidos previamente.

```
class Calle with Banqueta with Pavimento {

}
```

Al mezclar los `traits` anteriores con la clase calle, cualquier objeto de esta clase podrá llamar a cualquiera de los métodos que hayan sido definidos en ellos. Resulta conveniente para efectos de este ejemplo presentar la palabra clave `override`. Esta palabra clave sirve para redefinir métodos en clases que heredan de otras clases o que mezclan traits. En este ejemplo, si se deseara poder construir calles donde sólo circulen bicicletas, usando `override` se tiene el siguiente código.

```
class Calle with Banqueta with Pavimento {
       override def circulan() {
              println("Bicicletas")
       }
}
```

D.2. Funciones

Scala implementa muchos aspectos de la programación funcional, siendo uno de los principales el soporte para *funciones de primera clase*; y es que en Scala una función no es más que un objeto. Una función en Scala generalmente se escribe de manera literal, siendo en tiempo de ejecución, una vez que se ha compilado a una clase, cuando sus objetos reciben el nombre de *valores de funciones*.

La sintaxis para declarar una función en Scala es la siguiente:

```
(x: Int ) => x + 1
```

La parte de la izquierda declara los parámetros de la función, mientras que la parte de la derecha la operación que ésta realiza. Dado que una función es también un objeto, es posible asignarlas a una variable para más adelante llamarlas o pasarlas como un objeto a otra función o método.

```
var aumento = (x: Int ) => x + 1
```

```
scala > aumento(1)
res0: Int = 11

def operacionFinDeMes(args, aumentoFunc) {
        var x = ....
        aumentoFunc(x)
}

operacionFinDeMes(args, aumentoFunc)
```

Considerar la siguiente línea de código que corresponde a una lista a la cual se le aplica la función que es pasada como parámetro. En este caso, tal función es la de aumento.

```
listaDeNumeros.map((x : Int) => x + 1)
```

El lenguaje ofrece mecanismos que permiten escribir literales de funciones de manera más concisa. La primera de ellas es eliminar la presencia de los tipos en los parámetros de las funciones. El código anterior quedaría como:

```
listaDeNumeros.map((x) => x + 1)
```

El compilador de Scala utiliza un mecanismo conocido como *tipado del objetivo* que permite (en ciertas ocasiones) identificar el tipo de una expresión, en este caso el tipo del argumento a la función (x). Hay ocasiones, sin embargo, en las cuales el compilador requerirá que se provea un tipo para la expresión.

Adicionalmente, pueden eliminarse los paréntesis de la expresión dejándola como sigue:

```
listaDeNumeros.map(x => x + 1)
```

Aún es posible simplificar la declaración de la función, pues puede usarse un guión bajo para reemplazar uno o más parámetros, siempre y cuando cada parámetro aparezca una sola vez en el cuerpo de la función:

```
listaDeNumeros.map(_ + 1)
```

Un caso particular del uso del guión bajo en una función, es cuando aquel representa una lista de parámetros, en cuyo caso se dice que es una *función parcialmente aplicada*. Recibe ese nombre dado que al invocar la función, ésta es aplicada a los argumentos que le son pasados, no necesitando que estos sean exactamente los que la función requiere según su firma. Esta flexibilidad permite hacer lo siguiente para un método suma como el que se presenta a continuación:

```
scala> def suma(a: Int, b: Int, c: Int) = a + b + c
```

```
sum: (Int, Int, Int) Int
```

Es posible aplicar esta función a los argumentos 1, 2 y 3 como sigue:

```
scala> suma(1,2,3)
res1: Int = 6
```

La función `sum` puede ser asignada también a una variable cualquiera y después puede ser aplicada a un conjunto de parámetros, como se muestra a continuación:

```
scala> val a = suma _
a: (Int, Int, Int) => Int = <function>
scala> a(1,2,3)
res2: Int = 6
scala> a.apply(1,2,3)
res3: Int = 6
```

Ahora, un ejemplo en donde no se esté aplicando la función a todos y cada uno de los argumentos, sino sólo a algunos sería:

```
scala> val b = suma(1, _: Int, 3)
b: (Int) => Int = <function>
```

Resulta interesante ver lo que responde el compilador al asignar la declaración anterior. Parece ser que está generando una nueva función que recibe un solo parámetro. Entonces al llamar a la función suma que ha sido asignada a la variable `b` se tiene:

```
scala> b(2)
res4: Int 6
```

La implementación del paradigma funcional en este lenguaje resulta interesante, pues adopta muchas de sus características de manera muy rigurosa en comparación con otros lenguajes funcionales populares como Ruby. Por la naturaleza de este trabajo quedarán pendientes temas como los *closures* y el manejo de recursión de cola que son casi inmediatos en este lenguaje.

D.3. Apareamiento de patrones mediante clases

El lenguaje Scala provee un mecanismo bastante interesante para aparear ejemplares de una clase de acuerdo a su estructura. Para hacer que una clase pueda ser apareable por patrones solo hay que añadir la palabra clave `case`.

```
case abstract class Vehiculo
case class Bicicleta(frente : Llanta, tras : Llanta) extends Vehiculo
case class Tren(llantas : List[Llanta]) extends Vehiculo
case class Autobus(llantasFrontales : List[Llanta],
              llantasTraseras : List[Llanta]) extends Vehiculo
```

Los ejemplares de este tipo de clases pueden ser construidos sin necesidad de usar la palabra clave `new`, a modo de función. Además, para cada una de ellas el compilador de Scala genera un método `equals` que implementa una comparación estructural y un método `toString`.

Un apareamiento de patrones en Scala consiste de una secuencia de opciones iniciando con la palabra clave `case`; cada una incluye un patrón y una o más expresiones a evaluar en caso de que el patrón coincida. Una expresión `match` es evaluada probando cada uno de los patrones en el orden en el que fueron escritos.

Una expresión de ejemplo para las clases anteriores que ilustra el funcionamiento de esta característica en el lenguaje, se muestra a continuación.

```
expr match {
  case Bicicleta(frente, tras) => println("Las bicis tienen normalmente dos llantas")
  case Tren(llantas) => println("Los trenes tienen muchas llantas")
  case Autobus(llantasFrontales, llantasTraseras) =>
              println("Los autobuses tienen por lo general cuatro llantas")
  case _ =>
}
```

El último patrón devuelve un valor predeterminado en caso de que ninguno de los tres casos anteriores hayan coincidido con el valor presentado. Si `expr` no fuera ejemplar de alguna de las tres primeras clases, coincidiría entonces con el último patrón y no pasaría nada, pues no hay código a ejecutar definido para él.

Esta característica del lenguaje permite hacer apareamientos que toman en cuenta el tipo de la expresión que le es pasada al método `match`. Adicionalmente a esto, dentro de cualquier expresión que define un patrón pueden haber guardias que especifiquen condiciones adicionales que lo complementen y que de manera directa aumentan la precisión del apareamiento.

```
def seleccionarVehiculo(x : Any) = x match {
       expr match {
         case b : Bicicleta => println("Nosotros, en bicicleta somos veloces")
```

```
        case t : Tren => println("Los trenes son muy muy veloces")
        case m : Autobus if m.isBRT =>
              println("Los BRT son casi inmunes a los embotellamientos")
        case _ => println("Los otros se atoran en los embotellamientos")
      }
}
```

Otro lugar donde se puede hacer uso de un patrón es en una expresión `for`. Suponiendo que la variable `lines` es un `Map` se puede hacer uso de una tupla para probar la coincidencia de los pares de llaves y valores que tenga.

```
for ((linea, estacion) <- lineas)
      println("La estacion "+ estacion +" pertenece a la linea "+ linea)
```

D.4. Extractores e inyectores

El concepto de extractores está muy relacionado con el concepto de apareamiento de patrones en el lenguaje Scala, pues ambos conceptos son útiles para efectuar descomposiciones sobre valores y, en el caso espacial de los extractores, no importando de qué clase sean.

Un extractor en Scala es un objeto que implementa el método `unapply`. El propósito de ese método es verificar la coincidencia de un valor con un patrón y después descomponerlo. Es una buena práctica escribir la implementación del método `apply`, el cual debe devolver una construcción de un valor a partir de sus componentes más atómicos. Por convención, para todo valor que coincida con `unapply`, su composición con `apply` da como resultado el valor original antes de ser procesado por `unapply`. Por el papel que desempeña `apply` se dice que dicho objeto es un inyector y tal método es por tanto un inyector.

En el siguiente ejemplo se muestra un objeto que actúa como inyector y también como extractor[Ode08].

```
object EMail {
    // The injection method (optional)
    def apply(user: String, domain: String) = user +"@"+ domain
    // The extraction method (mandatory)
    def unapply(str: String): Option[(String, String)] = {
        val parts = str split "@"
        if (parts.length == 2) Some(parts(0), parts(1)) else None
    }
}
```

Este objeto construye una dirección de correo a partir de dos cadenas y descompone una cadena la cual debe contener al caracter @ para que el objeto descompuesto sea diferente de

`None`. El método `apply` permite construir un objeto de tipo Email a partir de dos cadenas; `Email("usuario", "dominio")` de manera similar a la que se puede inicializar un objeto con un constructor definido.

Cada vez que el apareamiento de patrones encuentre un patrón relacionado con un objeto tipo extractor, su método `apply` será aplicado en la expresión.

```
expression match { case Email(name, domain) => ... }
```

```
Email.unapply(expression)
```

Una de las ventajas de los extractores sobre las clases `case` es que para ellos no es importante la relación que un patrón pueda tener con una clase en particular, en cambio siempre que una expresión es apareada por una clase `case` se sabe inmediatamente que tal expresión era un ejemplar de la clase. Esta característica se conoce como *independencia en la representación*. Una ventaja adicional es que los extractores son un mecanismo mucho más poderoso al momento de definir patrones complejos.

Las clases `case` tienen así mismo algunas ventajas sobre los extractores. La primera de ellas es la sencillez con la que puede definirse un patrón; también el hecho de que en la mayoría de los casos son mucho más eficientes que los extractores pues el compilador puede realizar optimizaciones sobre ellas.

D.5. Soporte para procesos concurrentes

Scala propone un cambio de paradigma al momento de lidiar con un número considerable de procesos dentro de una aplicación de tamaño considerable. Mientras la abstracción lógica-sintáctica basada en el bloqueo de objetos y de bloques de código que ofrece Java funciona para aplicaciones simples o pequeñas, para otras simplemente no. Scala propone el modelo de paso de mensajes entre procesos, en el cual un proceso opera solamente con objetos inmutables, reforzando la no compartición de datos en memoria. De esta manera se evitan los abrazos mortales y la hambruna entre procesos.

Un actor es una entidad similar a un hilo con un espacio para recibir mensajes. Para implementar un actor, basta con escribir una clase cuya superclase sea `scala.actors.Actor` e implementando el método `act`.

```
import scala.actors._

class ActorFlojo extends Actor {
       def act() {
              while(true) {
                     println("Espero con flojera")
                     Thread.sleep(100)
```

```
                }
        }
}
```

Al invocar el método `start` el actor empieza a ejecutarse. En este caso, el actor lo único que hace es imprimir un mensaje e irse a dormir.

Una manera alternativa y más concisa para construir un actor es mediante el método `actor` en el objeto `scala.actors.Actor`. Dicho método recibe una función parcial. El actor anterior puede reescribirse como sigue.

```
import scala.actors.Actor._

val actorFlojo = actor {
        while(true) {
                println("Espero con flojera")
                Thread.sleep(100)
        }
}
```

El actor anterior puede interactuar con otros actores sólo si puede recibir mensajes. Para poder recibir mensajes un actor tiene que declarar una función parcial para el método `receive` que especifique qué hacer con ellos. En el siguiente fragmento de código se presenta a un actor capaz de recibir mensajes.

```
import scala.actors.Actor._

val actorImitador = actor {
        println("Calladito me veo mas bonito. Esperare mensajes que repetir'")

        while(true) {
                receive {
                        case msg =>
                                println("todo lo que me llega lo repito: " + msg)
                }
        }
}
```

Cabe señalar que al enviar un mensaje un actor no se bloquea, tampoco es interrumpido al recibirlo. Cada mensaje recibido por un actor espera en su espacio de mensajes hasta que éste llama al método `receive`. Para enviar un mensaje a un actor se usa el operador **!**.

```
scala> actorImitador ! "soy tonto"
scala> todo lo que me llega lo repito: soy tonto
```

Vale la pena hacer notar la presencia de *cases* dentro del bloque `receive` de un actor. En el caso del último actor, para todo tipo de mensaje que reciba, siempre lo imprimirá, pero cuando sea necesario que haga algo especial dependiendo del tipo de mensaje, resulta conveniente usar *case classes*. Incorporando esto a los actores anteriores se tendría algo como lo siguiente:

```
import scala.actors.Actor._

val actorImitador = actor {
        ...

        while(true) {
                receive {
                        case msg : String =>
                                println("todo lo que me llega lo repito: " + msg)
                        case entero : Int =>
                                println("Mira, un entero: " + entero)
                        case vehiculo : Bicicleta =>
                                println("Ya ve moy, en bici sin miedo: " + vehiculo)
                }
        }
}
```

Ya se han visto los conceptos básicos respecto al manejo de procesos concurrentes en Scala. Sin embargo hay todavía muchos otros aspectos interesantes en torno al buen uso del modelo de actores que no han sido cubiertos y que por la naturaleza de este trabajo no es posible cubrir aquí. En el Capítulo 30 del libro *Programming in Scala* [Ode08] se profundiza un poco más en torno a este tema.

Apéndice E

Aplicaciones web

E.1. Características de una aplicación web

Actualmente, la mayoría de los sistemas que se desarrollan están basados en la web; usan el procolo HTTP para intercambiar datos entre el servidor remoto (la aplicación web) y un cliente (un explorador web o una aplicación especializada). A continuación se listan algunas de las razones por las cuales han proliferado los sistemas basados en la web:

- Promueven la centralización de los datos.
- Son potencialmente más seguro.
- Mantienen funcionalidades del sistema independientemente del sistema operativo (algunos exploradores que no se adhieren al estándar del W3C[1] pueden presentar problemas).
- Facilitan la interoperabilidad con sistemas que soportan el protocolo HTTP para transmitir información.
- No requieren instalación del lado del cliente.
- Las actualizaciones del sistema son transparentes para el usuario final.
- En la mayoría de las ocasiones, es posible hacer uso del sistema a través de un dispositivo de cómputo con acceso a internet.
- Son de fácil acceso desde un explorador, al visitar la URL del sistema.

[1] W3C: World Wide Web Consortium . Es una comunidad internacional donde miembros de organizaciones, un equipo de tiempo completo y el público en general trabajan en desarrollar estándares para la web.

Como el desarrollo de aplicaciones web se ha vuelto una tarea por demás común, existe una gran cantidad de bibliotecas modulares, mejor conocidas como *frameworks* y que encapsulan funcionalidades utilizadas comúnmente en sistemas de este tipo. Muchos de ellos están basados en patrones de arquitectura de software probados a lo largo de las últimas décadas.

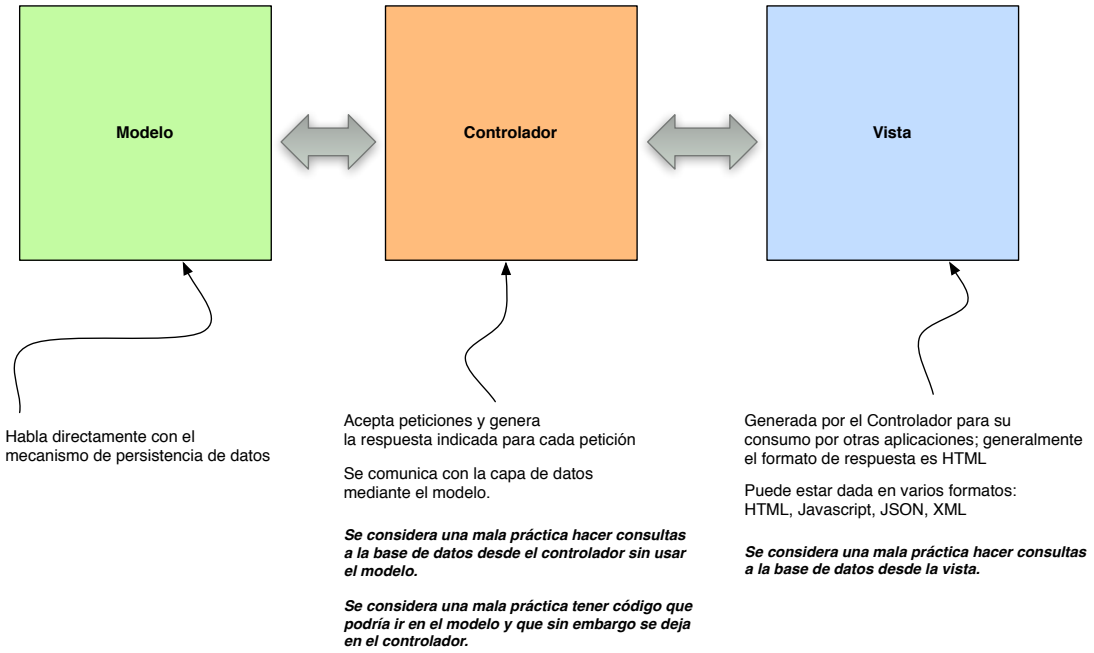

Figura E.1: Patrón Modelo-Vista-Controlador

El patrón más popular hoy en día es el MVC (*Modelo-Vista-Controlador*), aunque existen muchos otros como PAC (*Presentation-Abstraction-Control*), VF (*View-First*), CO (*Component-Oriented*).

Algunos *frameworks* de desarrollo web que usan el patrón MVC son:

- *Ruby on Rails* (*Ruby*)
- *Play* (*Java y Scala*)
- *Struts* (*Java*)
- *Django* (*Python*)
- *Yii* (*PHP*)

E.2. Arquitectura de una aplicación web

La arquitectura de una aplicación web, y por tanto, del sistema que se está planteando no es muy distinta al diagrama presentado en la figura E.1. De hecho, es una vista muy general de la arquitectura que pasa por alto componentes que es importante detallar para efectos de este trabajo, particularmente las implementaciones específicas del *framework* de desarrollo web que se propone utilizar: Play.

Play está escrito en Java y tiene soporte para Scala. Se propone usar su versión para Scala, porque el código tiende a ser más conciso y porque ayuda a reutilizar código. Ahora bien, como componentes comunes en un *framework* web moderno se encuentran:

- ORM (Object-Relational-Mapper).
- Servicios Web.
- Generador de HTML dinámico (*template engines*).
- Soporte para ruebas Unitarias y de Integración.
- Internacionalización.

En este trabajo sólo consideraremos los dos primeros componentes: ORM y Servicios web.

Un ORM facilita la interacción con sistemas manejadores de bases de datos, principalmente con aquellos basados en SQL, aunque también existen aquellos que interactuan de manera transparente con los que no lo están. El funcionamiento básico detrás de ellos es transformar tuplas de una tabla en objetos pertenecientes a una tabla, aunque también manejan sesiones y transacciones con la base de datos, construyen consultas, entre otras tareas. En el caso de Play se ofrece por omisión un envoltorio (o *wrapper*) alrededor de JDBC[2] que usa las características funcionales de Scala para interactuar con el sistema manejador de base de datos. Su única desventaja quizás es que se requiere escribir SQL específico al manejador de base de datos que se esté utilizando, lo cual significaría tener que reescribir las consultas en caso de cambiar el manejador de base de datos.

El componente de servicios web incluído en Play y en muchos otros *frameworks* web soporta la generación de la respuesta en formatos XML y JSON. De estos dos, el más importante es JSON por la facilidad que representa generarlo y consumirlo con otras aplicaciones. Dentro de este componente ha venido adquiriendo relevancia el concepto de servicios web tipo REST (*Representational State Transfer*). REST es una arquitectura de software para sistemas distribuidos como la web, que en años recientes se ha venido usando como un modelo de diseño para servicios web, desplazando a modelos arcaicos como SOAP y WSDL. REST se basa en muchos de los conceptos naturales al protocolo HTTP, siendo los más reelevantes

[2]JDBC: *Java Database Connectivity* es un API que facilita la comunicación con sistemas manejadores de bases de datos desde Java

las URL y los verbos HTTP. Las primeras definen la ubicación única de recursos, mientras que los verbos definen operaciones para interactuar con esos recursos. Los verbos HTTP comunmente usados en REST son: *GET*, *POST*, *PUT*, *DELETE*.

Además de los componentes anteriores, PLAY integró la biblioteca *Akka* que añade soporte para procesos concurrentes en la aplicación web mediante el modelo de actores.

Finalmente, REST utiliza *mime-types* [3] que son de utilidad para que el cliente que realizó una petición conozca el tipo de los contenidos que el servidor ha generado y enviado dentro de la respuesta. Así mismo, utiliza códigos de respuesta definidos en el protocolo HTTP para que el servidor incluya en la respuesta al cliente el resultado de la petición. Generalmente, en muchos *frameworks* MVC de desarrollo web, REST relaciona recursos en una URL con acciones (métodos) definidas en los controladores de la aplicación, y, dependiendo del formato especificado por el cliente que consumirá la respuesta, el controlador generará una representación del recurso en algún *mime-type*.

[3] *Internet media type* es un mecanismo para la identificación de formatos de archivos para internet