UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso de Profesionales en Actividad

TÍTULO: SAAS PARA MOVILIDAD Y ENVÍOS: DIFICULTADES y SOLUCIONES EMPÍRICAS

AUTOR: **ISLAS, Joaquín Ezequiel**

DIRECTOR ACADÉMICO: DE GIUSTI, Armando

DIRECTOR PROFESIONAL: LOPEZ, Cristian

CARRERA: Licenciatura en Informática

Resumen

El avance tecnológico de los últimos tiempos, ha facilitado el desarrollo y uso masivo de nuevas aplicaciones de software, principalmente móviles. El presente trabajo, tiene como fin recopilar una serie de soluciones a problemas surgidos durante el desarrollo de una aplicación que brinda servicios de movilidad y envíos. La misma fue desarrollada para integrar varias plataformas, entre ellas, webs y móviles, que sirven y comunican a distintos tipos de actores: consumidores finales y empresas, entre otras. Estas soluciones y mejoras del sistema, serán plasmadas en dicha tesis para facilitar desarrollos futuros

Palabras Clave

SAAS, Aplicación de Movilidad, Transporte, Aplicación de Envíos, Aplicación móvil, Mejoras de diseño de software, Algoritmos de Asignación de Choferes

OBJETIVOS

Analizar problemas encontrados en el desarrollo de una aplicación multiplataforma y multiusuario, la cual proporcionará servicios de movilidad y envíos.

Trabajos Realizados

Se realizó un análisis de las condiciones que favorecieron el desarrollo del sistema.
Se definió un caso de estudio de un sistema donde, es posible despachar envíos o traslados a clientes que estén conectados con el sistema mediante aplicaciones móviles.
Se documentó partes de una solución de referencia a problemas surgidos en el desarrollo de una aplicación multiplataforma, que fueron principalmente asuntos de performance y diseño del sistema.

Conclusiones

Como resultado de la presente tesina, se generó una propuesta de solución a problemas surgidos en el desarrollo de una aplicación multiplataforma de movilidad de objetos y/o personas, comparando algunas alternativas implementadas en diferentes versiones de los algoritmos. Se analizaron mejoras en el funcionamiento la misma, tratando diferentes áreas, que incluyen: análisis de requerimientos, diseño de software, optimización del uso de recursos como CPU y/o energía de todos los dispositivos involucrados.

Trabajos Futuros

Cada una de las propuestas de solución puede ser mejorada en cuanto a diseño o tecnologías aplicadas.

Debido a que la industria de software se caracteriza por evolucionar constantemente, los trabajos futuros, pueden abarcar mejoras en tiempos de respuesta, características de los algoritmos y tecnologías utilizadas para su desarrollo.

Fecha de la presentación: Noviembre 2021

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

Tesina de grado:
LICENCIATURA EN INFORMÁTICA

SAAS PARA MOVILIDAD Y ENVÍOS: DIFICULTADES y SOLUCIONES EMPÍRICAS

AUTOR: Joaquín Ezequiel Islas

DIRECTOR ACADÉMICO: Armando De Giusti

DIRECTOR PROFESIONAL: Cristian Lopez

CARRERA: Licenciatura en Informática

Noviembre 2021

RESUMEN

El avance tecnológico de los últimos tiempos ha facilitado el desarrollo y uso masivo de nuevas aplicaciones de software, en particular, se ha visto una proliferación del consumo de aplicaciones móviles en gran parte de la sociedad. El presente trabajo tiene como fin recopilar una serie de soluciones a problemas surgidos durante el desarrollo de una aplicación que brinda servicios de movilidad y envíos. La misma fue desarrollada para integrar varias plataformas como, webs y móviles, que sirven y comunican a distintos tipos de actores de la aplicación (consumidores finales, empresas prestadoras de servicios, empresas clientes, dueños de la aplicación, entre otros). Estas soluciones y mejoras del sistema serán plasmadas en esta tesis para facilitar desarrollos futuros de quienes requieran resolver problemas similares.

PALABRAS CLAVES

SAAS , Aplicación de Movilidad, Transporte, Aplicación de Envíos, Aplicación móvil, Mejoras de diseño de software, Algoritmos de Asignación de Choferes

CONCLUSIÓN

Como resultado de la presente tesina se generó una propuesta de solución a problemas surgidos en el desarrollo de una aplicación multiplataforma de movilidad de objetos y/o personas, comparando algunas alternativas implementadas en diferentes versiones de los algoritmos.

Se analizó acerca de las mejoras en el funcionamiento de una aplicación multiplataforma, tratando diferentes áreas, que incluyen entre otras: análisis de requerimientos, diseño de software, optimización de uso de bases de datos, ahorro en el uso del ancho de banda, memoria, CPU y/o energía en todos los dispositivos involucrados tanto servidores como los sistemas cliente ya sea PCs o teléfonos móviles.

TRABAJOS REALIZADOS

Se realizó un análisis de las condiciones que favorecieron el desarrollo del sistema.

Se definió un caso de estudio de un sistema donde, es posible despachar envíos o traslados a clientes que estén conectados con el sistema mediante aplicaciones móviles.
Se documentó partes de una solución de referencia a problemas surgidos en el desarrollo de una aplicación multiplataforma, que fueron principalmente asuntos de performance y diseño del sistema.

TRABAJOS FUTUROS

Cada una de las propuestas de solución puede ser mejorada en cuanto a diseño o tecnologías aplicadas.

Debido a que la industria de software se caracteriza por evolucionar constantemente, los trabajos futuros pueden abarcar mejoras en tiempos de respuesta, características de los algoritmos y tecnologías utilizadas para su desarrollo.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE EJEMPLOS DE CÓDIGO

CAPÍTULO 1: INTRODUCCIÓN

En el presente capítulo hablaremos sobre la motivación y el objetivo de la tesina de grado.

MOTIVACIÓN

Luego del transcurso de varias iteraciones de desarrollo, de un producto implementado para brindar una aplicación de movilidad de uso empresarial, se fueron encontrando problemas funcionales y no funcionales, entre los cuales podemos mencionar: problemas de mantenimiento y tiempos de respuesta.

Por eso se cree valioso recopilar las dificultades o falencias de desarrollo más importantes, como también la forma en que fueron resueltos o mejorados posteriormente para que sirva de ayuda a futuros desarrolladores que se topen con problemas similares.

Por otro lado, partes interesantes del sistema, se basan en asignar de forma eficiente solicitudes de envíos o traslados para ser atendidos prontamente (en un contexto donde la sociedad se vuelve cada vez más exigente y ansiosa), por tal, fue necesario hacer un análisis exhaustivo de estrategias de asignación, donde las primeras tenían menos complejidad, pero a su vez, mayor tiempo de resolución, luego se fueron ideando mejoras que permitieron brindar un mejor servicio de atención y la posibilidad de ser modificadas mediante parámetros de configuración (***Virues***, 2005).

Dichas mejoras, serán plasmadas en la actual tesis, para posibilitar trabajos futuros de perfeccionamiento en las mismas.

OBJETIVO

Este trabajo de finalización de carrera, tiene como objetivo analizar problemas encontrados en el desarrollo de una aplicación multiplataforma y multiusuario, la cual proporcionará servicios de movilidad y envíos.

Se expondrán diferentes versiones de la solución, que fueron desarrolladas por un equipo integrado por desarrolladores, analistas funcionales y testers, quienes formaron parte de un proyecto encargado a la compañía ***Hexacta S.A***. Se deja en claro que, este sistema se encuentra productivo desde hace un tiempo con expectativas de crecimiento futuro, pero por políticas de privacidad, no podemos revelar de qué aplicación estaremos hablando.

Entre los temas a tratar distinguimos:

- Problemas de sincronización de información como la ubicación, el estado y las tarifas de traslados y envíos, en forma simultánea, sobre distintas aplicaciones; 2 aplicaciones móviles y 3 portales web para servir a distintos tipos de usuarios ya sean: gerentes, empleados, telefonistas de remisería, dueños de la compañía de movilidad y empresas clientes.

- Algoritmos de selección de choferes para cada pedido según distintos criterios: disponibilidad, proximidad, cola de prioridades, ubicación en base de operaciones, ranking, asignación manual, etc.

- Dificultades de performance: escalar la aplicación y no demorar el envío y recepción de notificaciones, para que los clientes tengan el mejor tiempo de atención posible.

- Desacoplar y customizar la lógica principal de la aplicación como es la asignación de choferes.

ORGANIZACIÓN DE LA TESINA

Este trabajo está estructurado de la siguiente manera:

- En el capítulo 2, se realiza un análisis de las condiciones que facilitaron el desarrollo y uso del sistema, como también se brinda un pantallazo del modelo del negocio.

- En el capítulo 3, se describe cómo se compone la arquitectura del sistema desarrollado.

- En el capítulo 4, se introducen detalles de los problemas que surgieron en el desarrollo del software.

- En el capítulo 5, se detalla cómo fueron resueltos los problemas de sincronización de la información, mencionando algunas soluciones implementadas, incluyendo ventajas y desventajas.

- En el capítulo 6, se aborda el diseño de los algoritmos que lograron mejorar los tiempos de resolución de envíos o traslados.

- En el capítulo 7, se trata la resolución de problemas de desacople y configuración, con distintas estrategias o lógicas, para la asignación de choferes.

- En el capítulo 8, se describen estrategias de solución a problemas de performance, analizando las causas.

- En el capítulo 9, se detallan las conclusiones de la tesina.

CAPÍTULO 2: ANÁLISIS DEL ESTADO DEL TEMA

En esta sección revisamos qué cuestiones brindaron la posibilidad de desarrollar el sistema en cuestión, como también se explica el modelo del negocio.

HECHOS QUE POSIBILITARON EL DESARROLLO DEL SISTEMA

Haciendo un análisis tecnológico, los siguientes hechos posibilitaron el desarrollo de la aplicación en estudio. Estos elementos fueron imprescindibles para la factibilidad del desarrollo y ejecución del sistema acorde a las exigencias de los clientes.

Los de mayor importancia podrían ser:

- Servicios de geolocalización
- Teléfonos inteligentes a costos accesibles
- Redes de comunicación de alta velocidad
- ***Frameworks***[1] de desarrollo de aplicaciones móviles para múltiples plataformas
- Servicios de mensajería y notificaciones asincrónicas
- Escasez o Nulidad de aplicaciones de movilidad dedicadas a vincular empresas de transporte tradicionales y preexistentes con el ámbito digital

Se detalla la importancia de algunos de los puntos mencionados.

- **Servicios de geolocalización**

Estos servicios son imprescindibles para poder informar en tiempo real al usuario de la aplicación, el tiempo aproximado de realización y finalización del envío o viaje. Utilizado también para facilitar las rutas óptimas en la realización del viaje, el cual informa a los clientes visualmente en qué punto de un mapa se encuentra su chofer asignado.

Así mismo, este permite desambiguar la identificación de direcciones de destino provistas por el cliente (problema muy frecuente al tratar de identificarlo solamente

[1] ***Framework***: Un entorno de trabajo, o marco de trabajo es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. En informática generalmente soportado por una herramienta de software.

con palabras o números) e incluso a su vez, le quita responsabilidad al transportista de saber la ubicación de tal, de antemano.

- **Teléfonos Inteligentes**

La aplicación en estudio fue realizada teniendo en cuenta la disponibilidad de ***smartphones***, tanto para los clientes como para los choferes de las compañías de transporte. Sin este elemento tecnológico, el desarrollo del sistema no tendría sentido, ya que las principales y más provechosas características, se basan en el hecho de que ambos actores tengan interacción con el sistema en cualquier punto donde se encuentren, mediante algún dispositivo móvil.

Además, hablando exclusivamente de los choferes, es imprescindible que estos dispositivos cuenten con el servicio de ***GPS*** funcionando, para permitir exponer su ubicación tanto a los clientes, como al personal administrativo de la compañía.

Los precios y prestaciones de los teléfonos inteligentes, fueron mejorando en los últimos años, permitiendo que la mayoría de la población (***Statista***.,3 jun. 2021), cuente con algún dispositivo para satisfacer las necesidades de hardware necesarias para ejecutar la aplicación.

- **Redes de comunicación de alta velocidad**

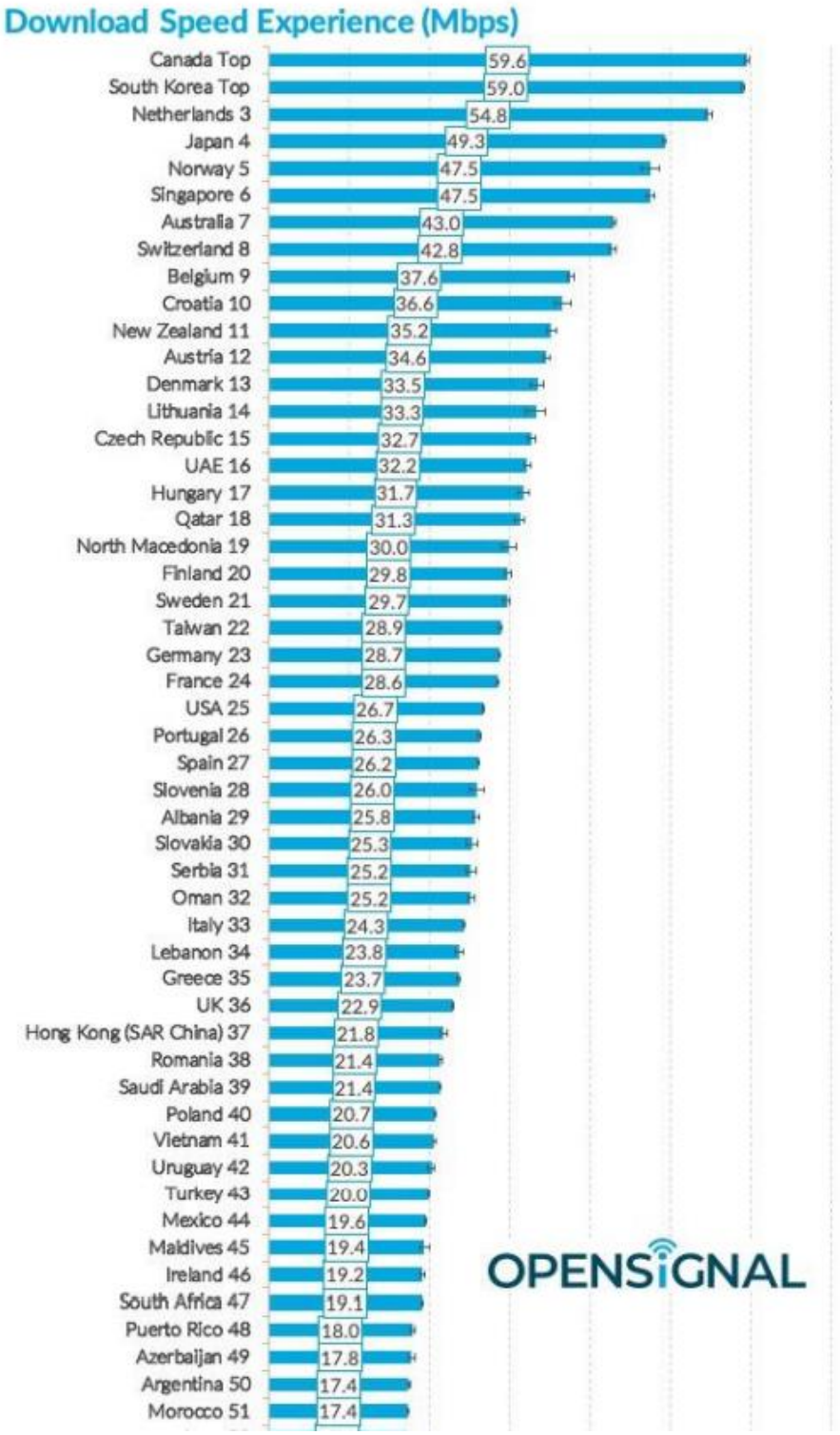
Alineado con el anterior requerimiento, los dispositivos móviles, serían inútiles para los fines de uso de la aplicación si los mismos no estuvieran conectados a internet para comunicarse con el sistema desarrollado.
A su vez por la frecuencia de actualización de datos deseada es necesario que el ancho de banda en esta comunicación sea considerable. Es altamente deseable que como mínimo los dispositivos estén conectados mayormente a una red 3G que brinde una velocidad de descarga de entre 1 a 4 Mbps.

Esto no hubiera sido posible sin el desarrollo de infraestructura que las compañías de comunicación vienen realizando en todas partes del mundo. Según ***OpenSignal***[2], para el 2020 las redes 4G ya estaban al alcance de la mayoría de los usuarios, con desarrollos de redes 5G en algunos sitios.

Como puede verse en los siguientes gráficos, el promedio de descarga ronda los 20 mbps (***Opensignal***, 2020):

[2] ***OpenSignal***: una empresa dedicada a brindar reportes periódicos de avance de internet a nivel global

Tabla 2.1: Experiencia en la velocidad de descarga por país



Hablando puntualmente de la Argentina, lugar donde fue pensado el desarrollo inicial del sistema, el último informe de ***OpenSignal***, muestra que para el 2020 el alcance de la red 4G era de un 86% de disponibilidad, lo que representa un buen

entorno para la puesta en ejecución del sistema en estudio (***OpenSignal***, 2020).

A continuación los detalles reportados para Argentina según ***OpenSignal***:

Tabla 2.2: **Detalles de velocidad de internet de Argentina**

Country	Download Speed Experience (Mbps)	±	Upload Speed Experience (Mbps)	±	4G Availability (% time)	±
Afghanistan	2.9	0.0	0.7	0.0	46.8	1.6
Albania	25.8	0.5	7.3	0.2	79.7	0.6
Algeria	4.0	0.0	2.5	0.0	64.9	0.4
Argentina	17.4	0.1	6.4	0.0	86.0	0.1
Australia	43.0	0.2	9.4	0.0	94.0	0.1
Austria	34.6	0.4	10.2	0.1	91.4	0.2

- **Frameworks de desarrollo de aplicaciones móviles para múltiples plataformas**

Otro punto importante que fue de gran ayuda para el desarrollo del sistema, es la existencia de ***frameworks*** de desarrollo de aplicaciones móviles para múltiples plataformas.

La variedad de sistemas operativos móviles existentes en el mercado implican a su vez que los estándares de desarrollo, los lenguajes de programación y los mercados de distribución de los mismos también sean diversos.

Para maximizar las posibilidades de crecimiento de una aplicación móvil, es deseable poder permitir su instalación en la mayor cantidad de plataformas existentes (principalmente ***Android***, ***IOS***). Con lo cual, son necesarios distintas versiones de la aplicación: 1 por cada tipo de plataforma objetivo (***Luis Corral*** et al., 2012).

Ante esta situación hay varias opciones, a continuación, algunos detalles de las mismas:

- **Desarrollo Nativo por plataforma**

  Para cada tipo de plataforma se genera un desarrollo nativo específico con las características deseadas del sistema. Este enfoque demandaría

conseguir personal capacitado en cada tecnología deseada, tiempo y recursos para desarrollarlos, pero tiene como principal ventaja el aprovechamiento de todos los recursos y características del sistema sobre el que se ejecutan.

Los rendimientos de estas aplicaciones suelen ser óptimos, comparándolos a los que se tendrían en aplicaciones que se desarrollasen utilizando otras técnicas.

- **Utilizando un framework multiplataforma**

Esta técnica, consiste en desarrollar las aplicaciones móviles con un mismo lenguaje de programación, pero luego, genera distintas versiones de la misma aplicación, para ser instalada en más de 1 tipo de plataforma móvil. De esta forma, ya no sería necesario tener que reescribir una solución a un requerimiento en todos los lenguajes de desarrollo correspondientes a las plataformas objetivo. Estas herramientas, utilizan tecnologías que son comunes a diferentes plataformas, como ***HTML***, ***CSS*** y ***JavaScript***, operando la funcionalidad del dispositivo móvil a través de un conjunto de interfaces de programa de aplicación (***API***), que cada fabricante brinda (Luis Corral et al., 2012).

Esta fue la manera utilizada para desarrollar las aplicaciones móviles en el proyecto en cuestión, usando específicamente los ***frameworks Ionic*** y ***Cordova*** para tal fin.

● Servicios de mensajería y notificaciones asincrónicas

La parte más importante del sistema, se basa en la capacidad de notificar a los distintos actores involucrados acerca de distintos eventos que se van sucediendo.

Los principales eventos a notificar son:

- Un cliente solicita un viaje/envío
- Un cliente cancela el viaje/envío
- Un chofer fue asignado a un viaje
- Un chofer acepta realizar un viaje
- Un chofer rechaza un viaje
- Un chofer llegó al punto de partida
- Un viaje finalizó
- Un viaje se canceló
- Un viaje programado debe iniciarse

Como puede observarse, la necesidad de recibir notificaciones es alta, lo que demanda gestionarlas de la forma más óptima posible, garantizando una pronta entrega. Es deseable consumir la menor cantidad de recursos del sistema, como también datos de transferencia,
por esta razón, es importante la existencia de métodos de comunicación livianos, pero que a su vez, tengan una tasa de fiabilidad alta.

Afortunadamente, hay varias opciones al respecto que se integran correctamente con distintos tipos de plataformas, las cuales ahondaremos detalles en los siguientes capítulos.

- **Escasez o Nulidad de aplicaciones de movilidad dedicadas a vincular empresas de transporte tradicionales y preexistentes con el ámbito digital**

El sistema en estudio, está pensado para integrar en el mundo digital, a las ya conocidas remiserías o empresas de traslados. Esto en sí, es algo que no se encuentra en el mercado del software previamente.

Si bien, ya hay aplicaciones móviles de envíos y traslados, (***Uber***, ***Didi*** o ***Cabify***), el sistema del negocio es diferente. En estos tipos de aplicaciones, los proveedores del servicio son usuarios independientes y no una organización en sí misma. La forma de trabajar para la compañía no es un proceso accesible a cualquier remisería preexistente.

Esta es una de las motivaciones del dueño del producto, poder facilitarles a ellas, una forma simple de poner en línea sus servicios de traslado, brindándoles una automatización total a sus clientes, facilidades y beneficios del uso de las aplicaciones móviles.

Para ello, se hace uso de la modalidad conocida como ***SAAS***, en ingles "***Software as a Service***", que significa software como un servicio. Este modelo de entrega del software está basado en la nube, donde el proveedor de la misma, en nuestro caso el dueño del producto, desarrolla y mantiene el software de las aplicaciones, proporciona actualizaciones automáticas y lo pone a disposición de sus clientes a través de Internet con un sistema de pago por uso que, en nuestro caso, es un porcentaje de los viajes manejados por el sistema.

El proveedor de la nube también administra todo el ***hardware***, el ***middleware***[3], el software de aplicaciones y la seguridad. De ese modo, las agencias de transporte pueden reducir drásticamente los costes de implementar, ampliar y actualizar las soluciones empresariales con mayor rapidez de la que posiblemente puedan hacerlo si lo implementara todo por su cuenta, desligándose de tareas como el mantenimiento de sistemas y de software locales (***Oracle***, 2021).

Otro beneficio brindado, es predecir el coste total de implementación con mayor precisión, útil, al evaluar e incluir más servicios disponibles.

ENTENDIENDO EL NEGOCIO

Esta tesina tiene como fin, analizar los problemas encontrados en el desarrollo de una aplicación multiplataforma y multiusuario que proporciona servicios de movilidad y envíos. Provee funcionalidad a distintos tipos de usuarios para tal fin.

Los principales perfiles identificados pueden clasificarse como:

- **Transportistas**: personal jerárquico/dueños y administrativos de las compañías de transporte incorporadas como socias de la aplicación.
- **Clientes Finales:** Personas que solicitan la realización de un envío o un traslado.
- **Clientes Empresa:** Organizaciones que contratan servicios de movilidad con los dueños de la aplicación.
- **Empleados:** personal cuya tarea es realizar el envío o traslado de personas o cosas.
- **Telefonistas:** personal que gestiona la asignación de los viajes a favor de los transportistas.
- **Dueños de la compañía:** acceden para customizar parámetros que afectan las tarifas o para registrar algunas de las entidades antes mencionadas.

Conceptualmente, la aplicación en cuestión, termina siendo un intermediario entre todos los actores anteriores, ayudando a resolver múltiples necesidades en tiempo real.

[3] ***Middleware***: Middleware o lógica de intercambio de información entre aplicaciones, o Agente Intermedio, es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, o paquetes de programas, redes, hardware o sistemas operativos (Wikipedia, 2021).

Podríamos mencionar las siguientes acciones principales para poder resolver los distintos tipos de envíos o viajes:

Clientes Finales/Empresas:

1. Solicitar un viaje
2. Cancelar un viaje
3. Programar un viaje para ejecutarse a posteriori

Empleados:

1. Marcarse como disponibles para trabajar
2. Aceptar un viaje
3. Marcar el viaje como iniciado
4. Marcar el viaje como finalizado
5. Cancelar un viaje
6. Marcarse como fuera del trabajo

Telefonistas:

1. Asignar un viaje manualmente a un chofer
2. Cancelar un viaje
3. Programar un viaje para ejecutarse a posteriori

Los flujos más comunes del manejo de un viaje se representan en el siguiente diagrama de estados:

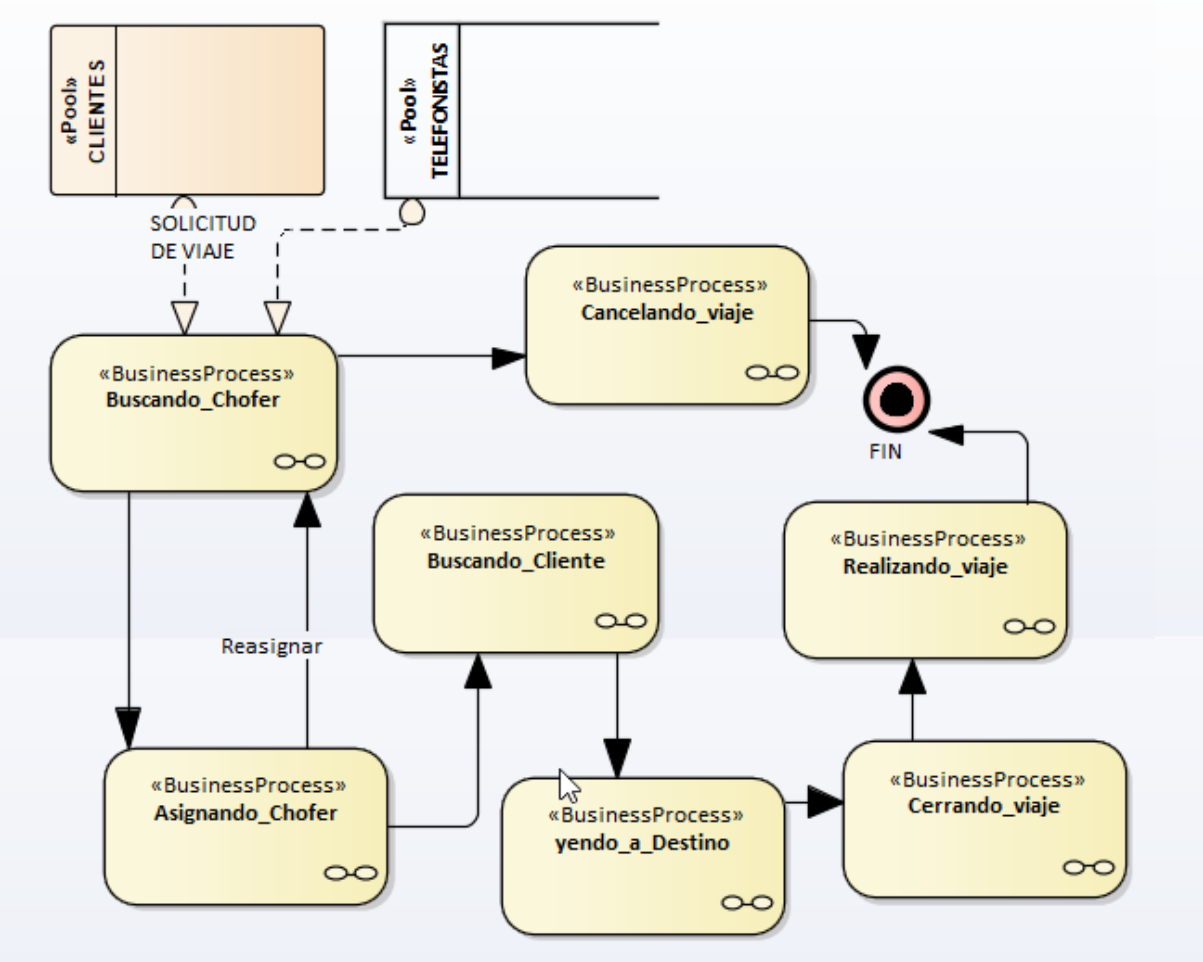


Figura 2.1: Diagramas de estados de un viaje

Como puede verse tanto los clientes o los telefonistas, pueden iniciar el viaje pero luego de que esto ocurra, el flujo depende principalmente de los choferes que deben ir actualizando el estado del viaje para mantener informado al resto de los actores de las acciones.

CAPÍTULO 3: ARQUITECTURA DEL SISTEMA EN ESTUDIO

La arquitectura del sistema está compuesta por varias tecnologías. Algunas son utilizadas para desarrollar la interfaz web, otras para la interfaz de las aplicaciones móviles y otra para la lógica y las operaciones del sistema.

Conceptualmente, esta última parte, se encuentra atendiendo requerimientos ***HTTP***[4], utilizando internet como medio de comunicación y siguiendo políticas ***RESTFUL***[5] a la que llamaremos ***Backend***, quien es el encargado de manejar la lógica del negocio, la búsqueda y persistencia de los datos.

La información es almacenada en una base de datos relacional, que en nuestro caso fue desarrollada por ***ORACLE*** y cuya versión es la 12.

Además, hay partes del sistema que se encargan de manejar la interfaz con los usuarios, para ello se desarrollaron varios tipos de interfaz.

Un tipo de interfaz utilizada es web, diseñada para ejecutarse sobre computadoras de escritorio, en general, utilizada por los empleados de las empresas cliente y de los transportistas o los dueños de la compañía. Mediante ésta, se pueden realizar muchas funcionalidades (más allá de solicitar viajes) como lo es: la rendición de los mismos, el acceso a reportes, la administración de los datos de las entidades, la configuración de los parámetros que afectan el uso de la aplicación, entre otras funciones. Esta parte del sistema se desarrolló utilizando principalmente ***Angular 6***, un sistema que sirve para desarrollar aplicaciones de una sola página (***SPA***, por sus siglas en inglés ***Single Page Application***).

Por otro lado, se desarrollaron aplicaciones móviles para ser ejecutadas sobre teléfonos inteligentes de distintos fabricantes permitiendo su instalación en distintos sistemas operativos o hardware. Éstas, son utilizadas por los clientes y choferes al

---

[4] ***HTTP***: El Protocolo de transferencia de hipertexto es el protocolo de comunicación que permite las transferencias de información a través de archivos en la World Wide Web.

[5] "¿Qué es una ***API*** de ***REST***? - Red Hat." https://www.redhat.com/es/topics/api/what-is-a-rest-api. Se consultó el 6 sept. 2021.

momento de solicitar y atender viajes o envíos. Esta parte del sistema se implementó usando la tecnología ***Ionic*** y ***Cordova,*** las cuales brindan la posibilidad de generar aplicaciones que pueden ser ejecutadas en sistemas ***Android*** e ***IOS*** a partir de una misma fuente de codificación.

En resumen, podemos listar y clasificar las tecnologías utilizadas con la siguiente tabla:

Tabla 3.1: **Detalles de las tecnologías usadas**

MOBILE	WEB	BACKEND
Ionic 3.20 Cordova 8.1.2 TypeScript 2.6.2 Angular 5.2.10 Node 9.8.0 HTML5, CSS3/SCSS	Angular 6 HTML 5, CSS 3 Bootstrap	Hibernate JAVA 8 Spring Boot 2.0 BD Oracle 12

Las anteriores, se pueden integrar con otras tecnologías para brindar soluciones a gran escala y resolver algunos requerimientos no funcionales como, por ejemplo, con los ***API Gateways***[6], los manejadores de carga y replicación de servidores, y así, brindar mayor soporte a los clientes.

[6] ***API Gateway:*** Una puerta de enlace o gateway API es componente de software que se usa para acceder a una interfaz de programación de aplicaciones (API) y actúa como un único punto de entrada para un grupo definido de microservicios. Debido a que una puerta de enlace maneja las traducciones de protocolos, este tipo de programación de front-end es especialmente útil cuando los clientes construidos con microservicios utilizan varias API dispares.

A continuación, un gráfico que muestra cómo se aplica este criterio en los diferentes entornos propuestos del sistema, ya sean: entornos de desarrollo, identificado con el rectángulo nombrado como DEV, entorno de pruebas, nombrado en el gráfico como TEST o entorno de producción, identificado como: VPC PROD:

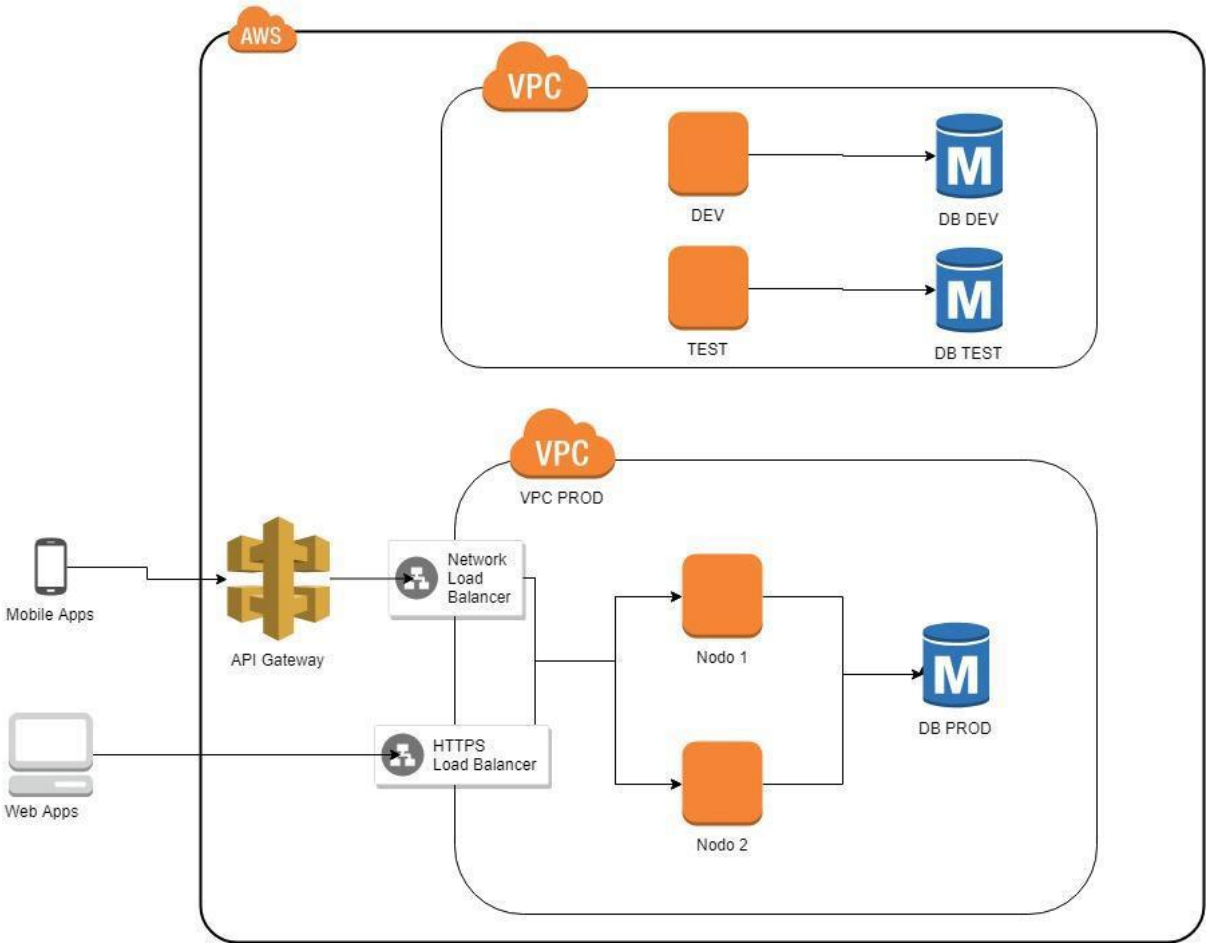


Figura 3.1: Diagrama de la arquitectura para distintos entornos

Como puede verse, el soporte de infraestructura está brindado por el proveedor externo ***Amazon***.

DESCRIPCIÓN DEL HARDWARE SEGÚN TIPO DE AMBIENTE

Detalle del hardware utilizado para los distintos tipos de entornos usados para desarrollar el sistema:

Tabla 3.2: **Detalles del hardware utilizado según ambiente**

Servidor	**producción**	**desarrollo**	**pruebas**	**local**
Equipo	EC2[7] m4.xlarge	RDS[8] db.t2.medium	t2.medium	Lenovo 81AX
CPU	Intel Xeon E5-2676 v3 de 2,4 GHz (4 núcleos)	Intel 3.3GHz (2 núcleos)	Intel 3.3GHz (2 núcleos)	Intel Core i5-8250U 1.6GHz (4 núcleos)
RAM	16 GB	4 GB	4 GB	20 GB
Ancho de Banda dedicado	750 Mbps	---	---	---
Rendimiento de la red	Alto	De bajo a moderado	De bajo a moderado	De bajo a moderado
Base de Datos	RDS db.t2.medium Oracle	RDS db.t2.small Oracle	RDS db.t2.small Oracle	Oracle Express v12
CPU	2 núcleos	1 núcleo	1 núcleo	compartida
RAM	4 GB	2 GB	2 GB	compartida

Como se observa, la mayor cantidad de recursos se destinan a los entornos productivos y los sistemas de desarrollo local. En el primer caso, para poder satisfacer una mayor cantidad requerimientos provocada por el uso de mayor cantidad de usuarios, en el segundo caso, para poder brindar recursos a las herramientas de desarrollo que puedan generar demanda alta del CPU y de la memoria al construir las aplicaciones y ejecutarlas localmente.

[7] ***Amazon Elastic Compute Cloud*** por sus siglas en inglés. https://aws.amazon.com/ec2/. Se consultó el 7 sept. 2021.

[8] ***RDS***: Se sus siglas en inglés: ***Amazon Relational Database Service***, es un servicio de base de datos relacional distribuida de ***Amazon Web Services***. Es un servicio web que se ejecuta "en la nube" diseñado para simplificar su configuración, el funcionamiento y el escalado de una base de datos relacional para su uso en aplicaciones (***Amazon***, 2021).

CAPÍTULO 4: ANÁLISIS DE LOS PROBLEMAS SURGIDOS

En el transcurso de varias iteraciones del sistema desarrollado, fueron surgiendo inconvenientes que entorpecieron la tarea de entregar un producto con niveles altos de calidad y que cumpla con las expectativas del dueño del mismo.
A continuación se brindan detalles de los problemas encontrados.

SINCRONIZACIÓN DE INFORMACIÓN

Como se mencionó anteriormente la aplicación en estudio se diseñó para brindar una interacción entre distintos tipos de usuarios. Por las características del negocio es indispensable poder compartir datos entre ellos en tiempo real.

Los datos más necesarios por la aplicación son los que se relacionan con:

- Información y estado de los viajes
- Información, estado y posición de los choferes
- Características y/o servicios soportados por los vehículos utilizados

Una clasificación según el tipo de criticidad y su influencia por la frecuencia en que ésta cambia, seria:

Tabla 4.1: **Criticidad de sincronización de las entidades del negocio**

Entidad	Frecuencia de Actualización	Criticidad de Actualización
Viaje	Alta	Alta
Chofer	Media	Alta
Vehículo	Baja	Media

Como puede apreciarse en la anterior tabla, es crucial tener actualizaciones de los datos de viajes y de choferes lo antes posible, para mantener el estado de las aplicaciones móviles, el ***Backend*** y la aplicación web en simultáneo.

Varios actores del sistema pueden intervenir en la modificación de la información, incluyendo estado de un mismo viaje, lo que le agrega complejidad al asunto.

En el siguiente capítulo abordaremos las estrategias de solución y sus limitaciones.

ALGORITMOS DE SELECCIÓN DE CHOFERES

Un problema importante a abordar, es la forma en que se asignan los viajes a los diferentes choferes de las remiserías, esta función es vital para poder resolver los requerimientos de los clientes finales.
El problema tiene varias cuestiones a considerar:

1) Minimizar el tiempo de atención del cliente
2) Distribuir en forma equitativa el trabajo
3) Cumplir con características particulares solicitadas
4) Favorecer a empleados que tengan mejores calificaciones

Para resolver este problema, se fueron ideando distintas formas de solución las cuales serán plasmadas en el capítulo 6.

DESACOPLAR Y CUSTOMIZAR DISTINTAS SOLUCIONES EN LA LÓGICA DE ASIGNACIÓN DE CHOFERES

Otro problema surgido en el desarrollo de la solución, fue la tarea de incorporar distintas formas de asignar a los choferes los viajes, sin tener que hacer grandes cambios en el resto del sistema.

Dado que no se conocía una solución óptima de antemano, se creyó conveniente permitir tener varias opciones a las que se pudiera acudir sin tener que hacer cambios de “código” para ello.

Por lo antes dicho, se utilizó un patrón de programación del tipo comportamiento llamado originalmente ***Strategy*** (Gamma et al, 1994).

Se explicarán detalles de la solución en el capítulo 7.

PROBLEMAS DE PERFORMANCE

Como la mayoría de los sistemas, luego de desarrollar las funcionalidades necesarias, se suelen identificar problemas de rendimiento y en general se detectan cuando empiezan a escalar la cantidad o tamaño de datos manejados o cuando la cantidad de usuarios utilizando el sistema crece considerablemente.

Teniendo en cuenta que una buena parte del sistema está integrado por aplicaciones móviles, es importante analizar además del tiempo de ejecución, el consumo de memoria, el uso o el consumo de batería de estos dispositivos, como también el consumo de datos de internet para no perjudicar a los usuarios en mayores costos o molestias, esto favorece la cantidad de usuario satisfechos y futuras descargas de la aplicación (***Luis Corral*** et al., 2012).

En el capítulo 8 revisaremos qué acciones se realizaron para mejorar estas cuestiones.

CAPÍTULO 5: ESTRATEGIAS DE SOLUCIÓN A PROBLEMAS DE SINCRONIZACIÓN

Como la mayoría de los sistemas, la información de nuestra aplicación es compartida por varios usuarios. La misma, está manejada y distribuida por una parte puntual de la aplicación la que denominamos ***Backend***.

Backend, es una parte de la aplicación que principalmente se conecta con la base de datos, donde se almacena la información que se va generando con el uso de la misma. Además, en esta parte del sistema, se realizan la mayoría de las validaciones de acceso y reglas del negocio para mantener una consistencia en el mismo, como también garantizar el acceso y la seguridad de los datos.

Uno de los principales problemas que presenta la aplicación es la forma de mantener actualizados a todos los usuarios en simultáneo y a su vez realizarlo de la forma más óptima posible para economizar recursos de hardware.

Es por ello, que se aplicaron varias técnicas para tal fin, las más relevantes de ellas son:

- Uso de notificaciones asíncronas
- Manejo de memoria caché en los dispositivos móviles
- Lógica de aplicación duplicada en varias partes del sistema
- Manejo “retardado” para la generación de reportes (de choferes/rendiciones/ganancias)

A continuación, brindaremos algunos detalles de implementación cada una de estas técnicas.

USO DE NOTIFICACIONES ASÍNCRONAS

Dado que la aplicación tiene que tener actualizados varios clientes webs y dispositivos móviles en simultáneo, se buscó un mecanismo apropiado para realizarlo sin malgastar recursos de los distintos artefactos involucrados.

La primera opción y menos compleja es generar frecuentemente consultas al sistema ***Backend*** por la información que cada dispositivo tenga interés. Esto conlleva

a demandar al ***Backend*** una mayor cantidad de trabajo a pesar de que no haya habido actualizaciones en la información que se esté manejando. Esta técnica también desperdicia recursos como la batería de los teléfonos móviles, o electricidad de las computadoras de escritorio generando un gasto innecesario en el uso de internet al realizar la comunicaciones con el ***Backend*** con mucha frecuencia.

La mejor solución encontrada para resolver estos requerimientos, fue la utilización de un mecanísmo que se conoce con el nombre de "***sistema de notificaciones push***" (**PNS** de su siglas en inglés).

Los sistemas de notificación ***push,*** tienen un diseño similar a los modelos cliente-servidor. La estructura básica de un sistema de notificaciones ***push,*** se divide en tres partes: por un lado una aplicación que recibirá datos o instrucciones (en nuestro caso los dispositivos móviles y web), por otra parte el servidor, lo que identificamos como ***Backend***, que proporcionará las instrucciones o información para la aplicación anterior, y por último un servicio (por ejemplo, ***Google Cloud Servers***, ***Apple Push Notification Service***, etc.) que se encarga del intercambio de información entre ambas partes.

El siguiente gráfico resume la comunicación entre estas partes:

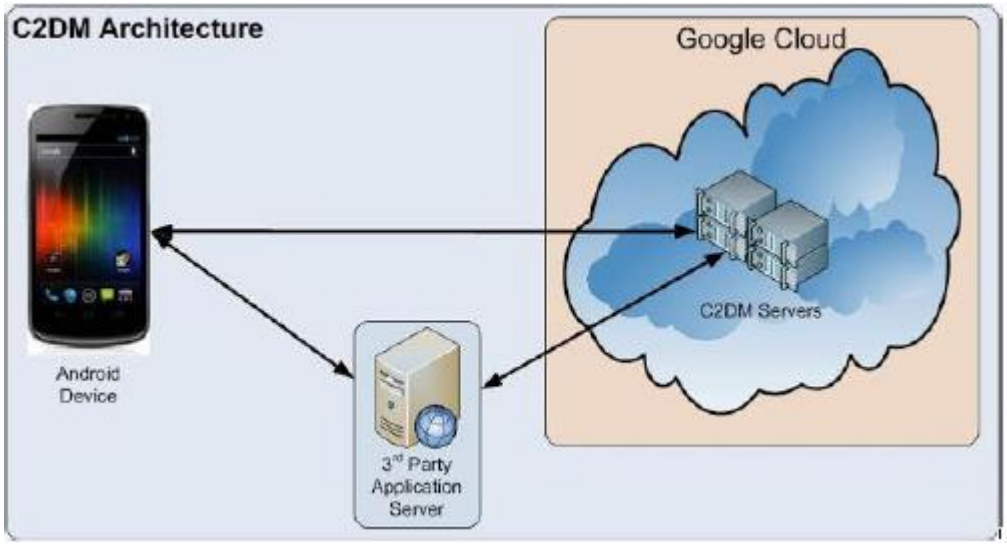


Figura 5.1: Vista del alto nivel de arquitectura para sistemas PNS (Li, N., et al. ,2013).

El servidor que envía la información a la aplicación, usa el servidor de terceros que actúa como intermediario para comunicarse y controlar el ritmo del intercambio de

datos. Esto permite," ***push"*** (empujar/ mover), la información a un dispositivo sin tener que sobre exigir el dispositivo, de lo que sería mantener varias aplicaciones en ejecución. Las aplicaciones pueden permanecer apagadas o inactivas mientras el usuario recibe una notificación de que hay nueva información o actualizaciones disponibles para una aplicación específica.

Esta técnica, aumenta en gran medida la eficiencia del tiempo de ejecución de las aplicaciones clientes, instaladas en los dispositivos, permitiendo al usuario tener el control de cuáles se están ejecutando en primer plano, mientras también sigue recibiendo información de los servidores de terceros (Li, N., et al. ,2013).

En nuestro caso, se utilizó la tecnología provista por Google, llamada ***Firebase Cloud Messaging*** (que se abrevia como ***FCM***, versión 6.11), tiene soporte multiplataforma para ***android*** o ***Ios***, y soportes de seguridad. Es gratuita, tolerante a fallos de red, y tiene soporte de caché.

Podemos definir 5 pasos que van ocurriendo secuencialmente para notificar a un dispositivo de cierto tipo de evento:

1. Nuestro sistema ***Backend*** envía un mensaje a los servidores de ***FCM*** de ***Google*** con la información que desea transmitir a los clientes (web o móviles).
2. Los servidores de ***Google***, ponen en cola y almacenan el mensaje en su base de datos hasta que pueda ser entregado al o los dispositivos indicados.
3. Una vez que él o los dispositivos del destinatario se conectan, los servidores de ***Google*** transmiten el mensaje.
4. A continuación, el sistema del dispositivo cliente (***Android***/***Ios***/***Web***), difunde el mensaje enviado desde los servidores, a la aplicación a través de ***Intent broadcast***, un mecanismo interno para despertar las aplicaciones que correspondan, comprobando los permisos adecuados para que sólo la aplicación de destino prevista reciba el mensaje. Este paso "despierta" a la aplicación, lo que significa que ésta no necesita estar ejecutándose para recibir un mensaje.
5. Por último, la aplicación procesa el mensaje. En nuestro caso, mostrar una notificación o recargar la pantalla actual con información relevante, por ejemplo: "Viaje asignado", "Viaje cancelado", etc.

Para que la anterior secuencia sea posible, existe una etapa de "registro" que debe ejecutarse desde los dispositivos clientes, tanto en los servicios de ***Google***, como en el sistema ***Backend***. En esta etapa se indica a los servidores ***FCM*** cierta información

para poder recibir los mensajes, generando un ***token*** de identificación por cada dispositivo.

El token de registro puede cambiar en las siguientes situaciones:

- La aplicación se restablece en un dispositivo nuevo.
- El usuario desinstala y vuelve a instalar la aplicación.
- El usuario borra los datos de la aplicación.

Es por eso, que cada cierta frecuencia, se vuelve a consultar el ***token*** más reciente. Se realiza desde las aplicaciones clientes, consultando a los servidores FCM y luego actualizando este dato en el ***Backend,*** para evitar problemas al generar las notificaciones.

Utilizando estos ***token***, que podríamos traducir como la dirección de envío del mensaje, se generan distintos tipos de notificaciones según se vaya actualizando la información importante del sistema.

Luego de que los mensajes son transferidos a los servicios de ***FCM*** de ***Google***, el ***Backend*** se desentiende de la tarea, continuando con otros flujos de ejecución que sean necesarios.

La siguientes líneas de código grafican la simplicidad del desarrollo del lado del ***Backend*** del envío de las notificaciones:

Ejemplo 5.1: **Implementación envío de Notificaciones Push**

```
public void  sendWebPushNotification(String clientToken, String
dataContent) {
  if(clientToken!=null){
      try {
         Message message = Message.builder().putData("body",
dataContent).setToken(clientToken).build();
         FirebaseMessaging.getInstance().sendAsync(message).get();
      } catch (InterruptedException e) {
         logNotificationError(Platform.Web, clientToken);
       logger.error(e.getMessage() + "Error al enviar notificacion al token: "
+ clientToken);
      } catch (ExecutionException e) {
         logNotificationError(Platform.Web, clientToken);
         e.getMessage();
```

```
        logger.error(e.getMessage() + "Error al enviar notificacion al token: "
+ clientToken);
      }
    }
}
```

La variable ***dataContent,*** contiene la información que es utilizada en las aplicaciones clientes, para mejorar la experiencia del usuario y evitar consultar datos nuevamente al ***Backend***. La variable ***clientToken,*** contiene el ***token*** que es reconocido por el sistema de ***Firebase***, para enviar la notificación al dispositivo del cliente.

La clase ***FirebaseMessaging,*** es un utilitario provisto por la ***JDK*** [9] brindado por Google, a la que se tiene acceso luego de importar la librería correspondiente.

Como puede verse, el hecho de usar este mecanismo, genera beneficios en ahorro de recursos de los dispositivos móviles. No se necesita estar activamente consultando por la actualización de los datos que estén utilizando. Estos son principalmente: la información de los viajes, de los clientes y de los choferes.

Este mecanismo, brinda la posibilidad de generar notificaciones, aun cuando la aplicación no está en primer plano de los dispositivos móviles. También simplifica el código a desarrollar por el ***Backend***, aprovechando un servicio provisto por un tercero que posee altos niveles de disponibilidad del servicio, (más del 95% según las fuentes oficiales), lo cual es un gran beneficio.

MANEJO DE MEMORIA CACHÉ EN LOS DISPOSITIVOS MÓVILES

En algunos momentos de la ejecución de la aplicación, se puede asumir que, ciertos datos, no van a cambiar al menos hasta cerrada o deslogueada la misma, es por eso, que se intenta aprovechar el uso de una base de datos local y temporal almacenada en las aplicaciones clientes, que se denomina ***local storage***, y así, evitar volver a consultar los mismos datos en un período de tiempo corto al Backend.

[9] ***JDK*** significa ***Java development kit*** *por sus siglas en inglés, que representa las herramientas o utilitarios para aprovechar las funcionalidades provistas por terceros.*

En particular, se accede a esta funcionalidad utilizando la interfaz ***window.localStorage,*** que es definida por la W3C[10].
Una aplicación puede utilizarse para guardar los datos persistentes usando pares de clave y valor en los dispositivos móviles, y utilizando la implementación provista por Apache Cordova, cuya interfaz funciona en forma similar al comportamiento definido para las páginas web, no obstante, en los dispositivos móviles todos los datos se borran cada vez que la aplicación se cierra (***Apache Cordova***, 2021).

Un ejemplo de código implementado, que representa la simplicidad de uso de la información almacenada en el ***local storage***, es:

Ejemplo 5.2: Uso de datos almacenados en Local Storage de Apache Cordova

```
/**
* Elimina los datos de sesión en local storage
*/
clearSession(): void {
   localStorage.removeItem(StorageKeys.tokenUserLogged);
   localStorage.removeItem(StorageKeys.userBaseInfo);
   localStorage.removeItem(StorageKeys.userInfo);
   localStorage.removeItem(StorageKeys.currentProfile);
}
```

LÓGICA DE LA APLICACIÓN DUPLICADA

En ciertos momentos, cuando no se dispone de conectividad de internet, o para economizar recursos principalmente del o los servidores de ***Backend***, se opta por duplicar cierta funcionalidad, tanto en el sistema cliente o como el servidor. Por ejemplo, se aplica esto al momento de mostrar el presupuesto de los costos de un viaje, aclarando que el precio final puede cambiar según el recorrido realizado o los tiempos de espera del mismo. Está estrategia reduce los momentos de consulta, transferencia y procesamiento de la información.

[10] ***W3C*** es el consorcio internacional de internet que se encarga de generar recomendaciones y estándares que aseguran el crecimiento de la ***World Wide Web*** a largo plazo.

MANEJO “RETARDADO” PARA LA GENERACIÓN DE REPORTES

Como se ha comentado, ésta aplicación, es utilizada no sólo por usuarios finales sino también por personal administrativo y por aquellos que poseen cargos gerenciales, tanto de las empresas clientes, como las empresas de transportes o los dueños del producto. Por lo tanto, éstos perfiles necesitan contar con funcionalidades que resuman cierta información de interés, por ejemplo cantidad de viajes realizados por un chofer en particular, volumen de ganancias obtenidas en cierto periodo de tiempo, cálculo de comisiones recibidas, consumos acumulados hasta la fecha.

Los reportes mencionados, implican realizar la contabilización y agregación de muchos registros de la base de datos.

Para mejorar la performance del sistema, se ideó una forma de tener los reportes generados de antemano, y además, que los mismos solo sean generados en los horarios no laborales según la zona horaria de las empresas en cuestión.
Como consecuencia, la información que reflejan estos reportes contempla registros que se hayan generado hasta el día anterior del momento que se esté visualizando.
Con esta estrategia, es más ameno el trabajo que debe delegarse a la base de datos, y además, se contempla realizar las operaciones de agregación en los momentos que el sistema tiene la menor cantidad de operaciones en ejecución.

Con el resultado de aplicar las anteriores prácticas, se logró economizar la cantidad de veces que se consulta y procesa la información, logrando así, ahorro de recursos como el: uso del CPU, uso de la memoria (principalmente de los servidores) y uso del ancho de banda en las comunicaciones, de todos los dispositivos involucrados, para lograr un mejor rendimiento general del sistema.

CAPÍTULO 6: DISEÑO DE ALGORITMOS DE ASIGNACIÓN DE CHOFERES

La función principal del sistema, es coordinar el traslado de pasajeros o cosas, por tal, es de crucial importancia la forma en que los viajes o envíos son despachados, es decir, cómo son los mismos asignados a los conductores disponibles para realizarlos.

Cuestiones que se consideraron para esto:

- Que la política sea lo más justa posible
- Que la carga de trabajo sea lo más balanceada posible
- Que se optimice el tiempo de solución del envío
- Que se consideren características especiales del servicio
- Que se valorice el buen servicio que brindan los choferes

Se revisarán 3 estrategias de solución, que se fueron implementando, realizando un análisis de sus ventajas y desventajas.

ESTRATEGIA INICIAL

La estrategia inicial, sólo contemplaba resolver los envíos de la forma más rápida posible, es por ello, que solamente se consideraba para la asignación del viaje la distancia entre la ubicación de los choferes y el origen del envío o traslado.
Al momento de realizar la búsqueda, eran considerados solamente los choferes que presentaban disponibilidad y que se hubieran conectado al sistema recientemente. Fue así, ya que era la única forma de asegurar que la ubicación registrada por el sistema, fuese lo más real posible.

Para esto el sistema del ***Backend*** provee el siguiente ***endpoint*** [11]para registrar la actividad de los choferes:

Ejemplo 6.1: Diseño del endpoint para actualizar estado del chofer

URI /drivers/{driverUserId}/refreshStatus
Method: Post

Body:

```
{
  "driverState": "OFFLINE",
  "driverSubState": "IN_BASE",
  "firebaseToken": "string",
  "latitud": "string",
  "longitud": "string",
  "platform": "Web"
}
```

El mismo utiliza como parámetros los puntos en el mapa obtenido por medio del sistema ***GPS***,(el cual los choferes deben tener habilitados), al que se accede mediante el ***plugin*** de ***Apache Cordova*** configurado con las siguientes versiones de dependencias:

Ejemplo 6.2: Configuración de dependencia de geolocalización de apache cordova

```
"cordova-background-geolocation": {
 "GOOGLE_API_VERSION": "16.+",
 "APPCOMPAT_VERSION": "28.+",
 "OKHTTP_VERSION": "3.12.+",
 "EVENTBUS_VERSION": "3.0.0"
}
```

Ya obtenidos los 2 puntos de interés que representan las ubicaciones del chofer y punto de origen, (obtenido desde la aplicación cliente o la programada por los telefonistas desde la interfaz web), no queda más que calcular las distancias entre

[11] **Endpoint**: en el diseño de interfaces, lo que podría considerarse como un extremo de un canal de comunicación. Cuando una API interactúa con otro sistema, los puntos de contacto de esta comunicación se consideran puntos finales o endpoints.

ambos. Para ellos utilizamos el servicio de ***Google Maps*** integrable al entorno de ***Backend,*** mediante la inclusión de la siguiente dependencia ***Maven***[12]:

Ejemplo 6.3: Configuración de dependencia de geolocalización del Backend java

```
<dependency>
  <groupId>com.Google.Maps</groupId>
  <artifactId>Google-Maps-services</artifactId>
  <version>0.1.9</version>
</dependency>
```

El siguiente código, realiza la solicitud al servicio para el cálculo de la distancia:

Ejemplo 6.4: Algoritmo de cálculo de la distancia versión 1

```
private DistanceMatrix getDistanceMatrixApi(Place from,Place to, Boolean
avoidTolls, Boolean avoidHighways){

   GeoApiContext context = new GeoApiContext().setApiKey(apiKey);
DistanceMatrixApiRequest request =
DistanceMatrixApi.newRequest(context).mode(TravelMode.DRIVING)
  .origins(new LatLng(Double.valueOf(from.getLatitude()),
Double.valueOf(from.getLongitude())))
  .destinations(new LatLng(Double.valueOf(to.getLatitude()),
Double.valueOf(to.getLongitude())))
  .units(Unit.METRIC);

if (avoidTolls != null && avoidTolls) {
  request = request.avoid(DirectionsApi.RouteRestriction.TOLLS);
}
if (avoidHighways != null && avoidHighways) {
 request = request.avoid(DirectionsApi.RouteRestriction.HIGHWAYS);
}

DistanceMatrix distance = request.awaitIgnoreError();

return distance; }
```

[12] ***Maven*** es el manejador de dependencias que se utilizó para desarrollar el proyecto

Como puede verse, se aplican algunos parámetros posibles cómo: evitar peajes o autopistas si el chofer así lo indica.
Cabe destacar que, la distancia obtenida, es igual al mejor recorrido posible calculado por la ***Api*** de ***Google Maps***, utilizando las rutas registradas en este servicio.

El resultado, que es de tipo "***DistanceMatrix***", es un arreglo de las posibles rutas para resolver el recorrido, cada una de ellas posee el tiempo estimado de recorrido y la distancia del mismo.

Mostramos cómo se integra el resultado obtenido de ***Google Maps Api,*** con la lógica de asignación y así poder, calcular la distancia o el tiempo de viaje entre 2 puntos:

Ejemplo 6.5: **Integración con la API de Google Maps**

```
public BigDecimal getDistanceInKms(Place from,Place to) throws
NotRouteFoundException {
  DistanceMatrix distance = this.getDistanceMatrixApi(from,to, Boolean.FALSE,
Boolean.FALSE);
  if (distance.rows[0].elements[0].distance != null) {
    BigDecimal distanceInMeters =
BigDecimal.valueOf(distance.rows[0].elements[0].distance.inMeters) ;
    //Return distance in KM
    return distanceInMeters.divide(new BigDecimal("1000"), 2,
RoundingMode.HALF_EVEN);
  } else {
    throw new NotRouteFoundException();
  }
}


public Long getDurationInSeconds(Place from, Place to) {
  DistanceMatrix distance = this.getDistanceMatrixApi(from,to, Boolean.FALSE,
Boolean.FALSE);
  return distance.rows[0].elements[0].duration.inSeconds;
}
```

Al elegir los choferes que serán candidatos para ser asignados a un viaje, se calculan los tiempos de arribo y distancia al punto de partida, teniendo en cuenta solo los choferes disponibles, los mismos, son seleccionados solo si la distancia al punto de partida es menor a un límite configurado de antemano. Luego de esto, los mismos son ordenados de menor a mayor según el tiempo de llegada, también, se incorpora un posible segundo criterio de ordenamiento en caso de que haya "empate", de 2 o más candidatos, el llamado "rating", que es un número promedio de todas las puntuaciones obtenidas de los pasajeros a los que se les brindó dicho servicio.

El código que termina de generar la lista de choferes a los cuales se les asignará el viaje es:

Ejemplo 6.6: **Consolidación de lista de candidatos**

```
public List<TravelCandidates> orderCandidates(List<TravelCandidates>
travelCandidates, CarrierAccount carrier){
  final Parameters paramRating = parameterService.getParameter(carrier,
ParametersType.PRIORITYRATING);
  final Boolean priorityRating;

  if (paramRating.getValue() != null) {
     priorityRating = Boolean.valueOf(paramRating.getValue());
  }else {
     priorityRating = false;
  }

   if (priorityRating) {

travelCandidates.sort(Comparator.comparingLong(TravelCandidates::getArriveEst
imate)
              .thenComparing(Comparator.comparing(TravelCandidates::getDriver,
                   Comparator.comparing(DriverAccount::getGeneralRating,
                        Comparator.nullsFirst(Comparator.naturalOrder())).reversed())));
   }else {

travelCandidates.sort(Comparator.comparingLong(TravelCandidates::getArriveEst
imate));
   }

   return travelCandidates; }
```

Una vez que la lista anterior está ordenada, se intentará asignar de a uno a la vez el viaje en cuestión. Si el primer chofer en la lista, luego de ser notificado, no acepta el viaje en una cantidad de tiempo determinada por parámetro, se lanzará la asignación del segundo conductor en la lista, y así sucesivamente hasta que alguno haya aceptado el viaje.

Como puede verse, esta estrategia se basa principalmente en reducir el tiempo de resolución del viaje, sin importar prácticamente otro criterio (como puede ser cantidad de viajes asignados a los choferes y un reparto equitativo), a su vez, depende principalmente de la respuesta de los servicios de ***Google Maps*** para el ordenamiento de los mismos, hecho que puede generar una demora extra en la resolución de la asignación.

A continuación, se explicará otra forma de resolver el problema que intenta mejorar algunas de las desventajas de la estrategia inicial.

ESTRATEGIA DE CDL

La segunda versión utilizada para el algoritmo de asignación de choferes, trató de mejorar principalmente la dependencia con el servicio de ***Google Maps***, ya que de por sí, es un servicio que al utilizarse en forma masiva genera costos para la empresa que lo contrata y a su vez introduce una sobrecarga en el tiempo de ejecución del sistema de asignación.

Por tal motivo, fue que se aplicó otra estrategia para no depender tan asiduamente de este sistema de ***Google***, la estrategia fue llamada ***CDL,*** siglas de "***cálculo de distancia lineal"***.

Como las palabras lo indican, la estrategia se basa en, calcular cuál es la distancia en línea recta entre 2 puntos del globo.

Se utiliza la base de un algoritmo conocido como "***Haversine",*** el cual tiene como objetivo calcular la distancia ortodrómica entre dos puntos, es decir, la distancia más corta sobre la superficie terrestre, lo que da una distancia "al vuelo" entre los puntos. Éste, se utilizó en sus orígenes para la navegación.

El nombre del algoritmo deriva del hecho que suele expresarse en términos de la función llamada ***semiverseno*** (***haversine*** en inglés), dada por:

$$\mathbf{haversin(\theta) = \sin^2(\theta/2)}$$

El mismo presenta bajos márgenes de error, salvo cuando se calcula distancias en puntos antipodales, o sea puntos opuestos del planeta, pero no es un caso posible para el uso actual de la aplicación.

La fórmula de obtención puede resumirse como:

Ejemplo 6.7: **Algoritmo Haversine para el cálculo de la distancia entre 2 puntos**

$$\Delta\varphi = \varphi 1 - \varphi 2$$
$$\Delta\lambda = \lambda 2 - \lambda 1$$

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$
$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R \cdot c$$

Donde “φ” es la latitud, “λ” es la longitud, “R” es el radio de la tierra (en nuestro sistema representado con 6.378,137 kilómetros), y “d” el resultado buscado. (Díaz, A. I. (2012))

A continuación, el código del ***Backend*** utilizado para resolver la anterior fórmula, cuyo resultado está expresado en metros, tomando como entrada las coordenadas de 2 puntos:

Ejemplo 6.8: **Implementación del algoritmo Haversine en java**

```
static final int EARTH_RADIUS = 6378137;

/**
 * Return the distance in Meters between from and to latLong points
 * @param from latitude and longitude point
 * @param to latitude and longitude point
 * @return distance in Meters between from and to latLong points
 */
public static Double computeDistanceBetween(LatLng from, LatLng to) {
  return computeDistanceBetweenHelper(from, to) * EARTH_RADIUS;
}

public static Double toRadians(Double angleDegrees) {
  return (angleDegrees * Math.PI) / 180.0; }
```

```
private static Double computeDistanceBetweenHelper(LatLng from, LatLng to) {
  Double radFromLat = toRadians(from.lat);
  Double radFromLng = toRadians(from.lng);
  Double radToLat = toRadians(to.lat);
  Double radToLng = toRadians(to.lng);
  return (2 * Math.asin(Math.sqrt(Math.pow(Math.sin((radFromLat - radToLat) / 2),
2)
      + Math.cos(radFromLat) * Math.cos(radToLat) *
Math.pow(Math.sin((radFromLng - radToLng) / 2), 2))));
}
```

El anterior algoritmo, reemplazó la utilización del servicio de ***Google*** al momento de generar la lista de candidatos para resolver un viaje. Igual que la estrategia inicial, se da prioridad a los choferes que estén más próximos al punto de origen del viaje, pero a su vez, entra en cuestión una nueva característica disponible para los choferes, el poder registrarse en una cola virtual llamada "***estar en base***".

Conceptualmente, cuando los choferes se estacionan en ciertos lugares configurados de antemano por los administradores del sistema, tienen la posibilidad de accionar un botón en sus aplicaciones que indica que ellos están esperando pedidos desde ***la base***, (puede ser el espacio físico de las oficinas y estacionamiento de las empresas o sitios elegidos estratégicamente por los dueños de las remiserías).

"Estar en base", les da cierta prioridad a los choferes al momento de ser asignados a los viajes.

Para comprender, a continuación, el algoritmo que genera la lista de posibles candidatos del viaje para esta estrategia:

Ejemplo 6.9: Consolidación de choferes candidatos versión 2

```
private Stream<Pair<Double, DriverAccount>>
getCombinedList(Supplier<Stream<Pair<Double, DriverAccount>>>supplierOnline,
Supplier<Stream<Pair<Double, DriverAccount>>>supplierInTravel,
Supplier<Stream<Pair<Double, DriverAccount>>>supplierInBase, Travel travel){

  long MAX_CANDIDATES = MAX_ONLINE + MAX_IN_TRAVEL;
  long countInTravel = supplierInTravel.get().count();
  long countOnline = supplierOnline.get().count();
```

```
  long REAL_ONLINE_SIZE =  (countInTravel < MAX_IN_TRAVEL) ?
MAX_CANDIDATES  - countInTravel : MAX_ONLINE;
  long IN_TRAVEL_REAL_SIZE = (countOnline < MAX_ONLINE) ? MAX_CANDIDATES
- countOnline : MAX_IN_TRAVEL;

  Stream<Pair<Double, DriverAccount>> topOnline = supplierOnline.get().filter(pair
-> isVehicleType(pair.getValue(), travel)).limit(REAL_ONLINE_SIZE);
  Stream<Pair<Double, DriverAccount>> topInTravel =
supplierInTravel.get().filter(pair -> isVehicleType(pair.getValue(),
travel)).limit(IN_TRAVEL_REAL_SIZE);

  Stream<Pair<Double, DriverAccount>> combinedList =
Stream.concat(supplierInBase.get(), Stream.concat(topOnline, topInTravel));

  return combinedList;
}
```

En las anteriores líneas de código puede verse que: en los parámetros de entrada de la función, hay distintas listas que representan choferes disponibles, en viaje o en la base. El resultado es una lista combinada que restringe la cantidad de choferes a contemplar, de los que no se encuentra en base, por lo tanto, algunos serán eliminados de ella. A su vez, los que sí estén en base, van a ser los primeros en ser evaluados para recibir un viaje.

En resumen: está estrategia generó independencia del servicio de ***Google Maps,*** al poder realizar cálculos de la distancia en forma independiente y a su vez incorporó un nuevo elemento de importancia, que es la funcionalidad de registrar empleados ubicados en distintas bases, quienes tendrán prioridad al momento de ser elegidos para los viajes.

ESTRATEGIA DE ANILLOS

Si bien, con la segunda versión del algoritmo se lograron mejoras al momento de asignar un viaje, se notó que el beneficio que representaba para los choferes el estar en base, atentaba contra el tiempo de solución del pedido, ya que no se tenía en consideración la distancia con respecto al origen del viaje. Además, se crearon turnos para confirmar la aceptación del pedido, asignados individualmente según la cantidad de candidatos de la lista, lo que podría llegar a demorar el viaje cuando

muchos no acepten el mismo, (hay una cierta cantidad de tiempo de espera por cada empleado que no acepte el viaje).

El siguiente diagrama de actividad resume un escenario genérico relacionado:

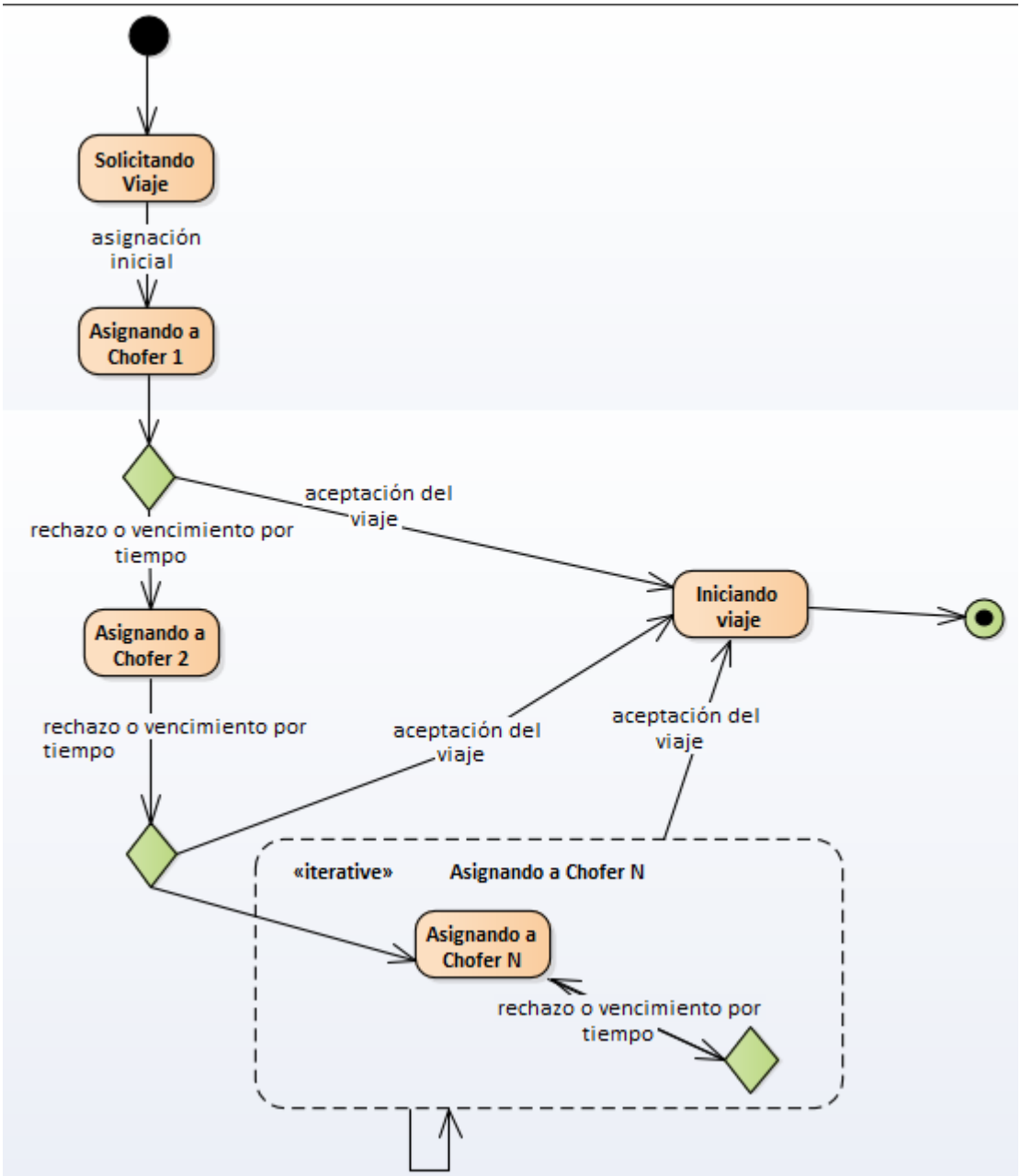


Figura 6.1: **Diagrama de actividad del algoritmo de asignación versión 2**

Para resolver esta situación, se generó una última versión del algoritmo que pretende resolver la demora innecesaria para el inicio del viaje. A este algoritmo lo llamaremos ***estrategia de anillos***.

La estrategia de anillos, pretende por un lado, acotar la cantidad de choferes que tienen privilegios al momento de recibir asignaciones a un valor establecido por parámetros de configuración de la aplicación (esto antes era ilimitado), por otro lado, luego de que esa cantidad es alcanzada, ya no se reservan tiempos individuales de espera para confirmar el pedido, sino que, se clasifican a los candidatos en distintos grupos según su ubicación actual para ser notificados y asignados al viaje requerido en forma conjunta.

Como consecuencia, la cantidad de tiempo de espera del cliente será acotada a una cantidad de turnos determinada. Por ejemplo, suponiendo que la cantidad de choferes privilegiados sea de 4, luego se generen grupos de 3 anillos, (teniendo en cuenta distancias al punto de origen), en total, habrá 7 rondas de notificaciones (4 de los choferes privilegiados y 3 de los grupos de anillos) antes que se dé por perdido el viaje. Por tal, se notificará al cliente que no hay choferes disponibles en ese momento.

A continuación, el algoritmo que genera la separación en grupos según sus ubicaciones con respecto a los anillos configurados:

Ejemplo 6.10: **Generación de lista de candidatos en algoritmo de anillos**

```
private List<List<TravelCandidates>>
generateCandidateGroups(List<TravelCandidates> candidates) {

  long i = 0;

  List<List<TravelCandidates>> result = new ArrayList<>();

  List<TravelCandidates> candidatesInRange =
        candidates.stream()
              .filter(s -> s.getDistanceToPPUP() <= distanceInKLMLimit * 1000)
              .collect(Collectors.toList());


  List<TravelCandidates> group0Aux =
        candidates.stream().filter(s -> candidatesInRange.indexOf(s) <
max_privilaged).collect(Collectors.toList());

  if (baseIsPrioritized) {
     List<TravelCandidates> candidatesInBase =
```

```
            group0Aux.stream().filter(s ->
s.getDriver().isInBase()).collect(Collectors.toList());

      List<TravelCandidates> candidatesOutOfBase =
            group0Aux.stream().filter(s ->
!s.getDriver().isInBase()).collect(Collectors.toList());


      for (TravelCandidates currentCandidate : candidatesInBase) {

         result = addCandidateGroup(result, currentCandidate);
      }

      for (TravelCandidates currentCandidate : candidatesOutOfBase) {

         result = addCandidateGroup(result, currentCandidate);
      }
   } else {

      for (TravelCandidates currentCandidate : group0Aux) {

         result = addCandidateGroup(result, currentCandidate);

      }
   }

   List<TravelCandidates> candidatesInRangeRemaining =
         candidatesInRange.stream().filter(s ->
!group0Aux.contains(s)).collect(Collectors.toList());

   List<TravelCandidates> group1Aux =
         candidatesInRangeRemaining.stream()
               .filter(s -> s.getDistanceToPPUP() <= distanceInKLMGroup1 * 1000)
               .collect(Collectors.toList());


   result.add(group1Aux);

   List<TravelCandidates> group2Aux =
         candidatesInRangeRemaining.stream()
               .filter(s ->  s.getDistanceToPPUP() > distanceInKLMGroup1 * 1000 &&
s.getDistanceToPPUP() <= distanceInKLMGroup2 * 1000)
```

```
                .collect(Collectors.toList());

    result.add(group2Aux);

    List<TravelCandidates> groupFinalAux =
            candidatesInRangeRemaining.stream()
                .filter(s -> s.getDistanceToPPUP() > distanceInKLMGroup2 * 1000 &&
s.getDistanceToPPUP() <= distanceInKLMLimit * 1000)
                .collect(Collectors.toList());
    groupFinalAux.addAll(getAffiliateCandidates());

    result.add(groupFinalAux);

    return result;
}
```

Como puede deducirse de las líneas de código anteriores, ***distanceInKLMLimit,*** es el límite máximo en la que un candidato puede estar posicionado con respecto al punto de origen del viaje teniéndolo en cuenta al momento de asignarlo.
Max_privilaged, es la cantidad máxima de “choferes privilegiados” para ser notificados y asignarles un viaje en forma individual. ***BaseIsPrioritized,*** es un parámetro adicional que genera un privilegio extra a los choferes que se encuentren en base, y es utilizado para ordenar a los mismos dentro del grupo de privilegiados almacenados en la variable ***group0Aux***.

El algoritmo va generando invocaciones a un método nombrado ***addCandidateGroup,*** que representa quiénes serán notificados al mismo tiempo para recibir una asignación del viaje.
Hay configuradas 2 distancias más, ***distanceInKLMGroup1*** y ***distanceInKLMGroup2***, quienes separan los choferes según sus ubicaciones en 3 grupos restantes que serán notificados entre sí al mismo tiempo.

Con esta estrategia, se sigue manteniendo una forma de asegurar que la distribución de los viajes sea lo más equitativa posible. Si un chofer se ubica en base, tendrá más chances de ser asignado a un viaje. Se habilita el parámetro de configuración correspondiente y a su vez exige a los mismos a estar atentos para aceptar el viaje lo más pronto posible antes que sus compañeros sean notificados del mismo. Con esta misma, también se logró mejorar principalmente el tiempo de demora que el solicitante debía esperar para ser trasladado o enterarse que no habría choferes disponibles para resolver su pedido.

El siguiente gráfico ejemplifica un escenario posible, suponiendo que el pedido es realizado desde un punto cercano al centro geográfico de la ciudad de La Plata, La Plaza Moreno:

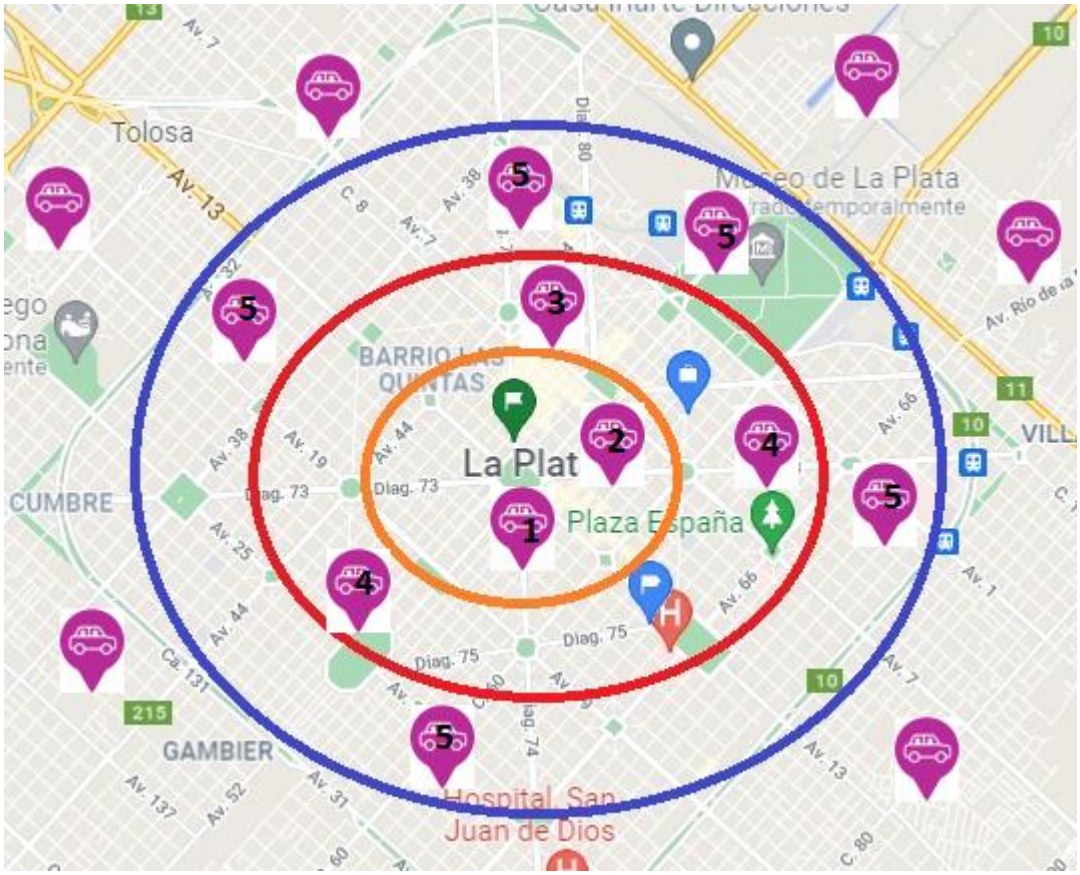


Figura 6.2: Simulación del orden de prioridad en algoritmo de asignación de Anillos.

Cada marcador **color violeta**, representa a un chofer a la espera de recibir asignación de un viaje.
El anillo **color naranja,** representa al primero, el **rojo** al segundo y el **azul** es el anillo de exclusión.
Los números sobre los marcadores, indican el turno que les fue asignado a cada chofer (suponiendo que la cantidad de vehículos “privilegiados” es de 3), luego de que los 3 primeros choferes reciben un turno individual para aceptar el viaje, se empieza a agrupar a éstos según su ubicación. Los que estén en el anillo más cercano serán notificados en primer lugar, luego los del segundo lugar y tercer anillo si lo hubiese. Como se puede ver en el ejemplo, algunos choferes no se encuentran en el área de cobertura para ese viaje y, por lo tanto, no son notificados.

El siguiente, es un cuadro comparativo a modo de resumen de algunos conceptos mencionados para las 3 estrategias planteadas:

Tabla 6.1: **Comparativa de las estrategias de asignación de choferes**

Concepto Observado	Estrategia Inicial	Estrategia CDL	Estrategia de Anillos
Usa una política justa	NO, solo se prioriza la rapidez de atención	SI, incorporando lista prioritaria “en base”	NO, puede priorizar a los choferes en base pero solo si están próximos al origen del viaje
Hace balance de trabajo	NO, solo se prioriza rapidez de atención	SI, usa una lista con política FIFO	NO, solo se prioriza rapidez de atención
Optimiza el tiempo de atención	SI, pero con demoras por uso de recursos externos	SI, con mejora al evitar uso de recursos externos	Si, limita choferes prioritarios
Considera características especiales del pedido	Si	SI	SI
Valoriza el servicio que presenta el chofer	Si, cuando 2 o más choferes que compiten están a igual distancia	Si, cuando 2 o más choferes que compiten están a igual distancia	No, se prioriza tiempos de respuesta

CAPÍTULO 7: DESACOPLAR Y CUSTOMIZAR DISTINTAS ESTRATEGIAS DE SOLUCIÓN

Como fue descripto anteriormente, en el desarrollo de la aplicación, se presentaron muchas mejoras en la forma en que se atendían los pedidos de clientes.

Es por ello que, fue necesario utilizar efectivas prácticas de programación, para facilitar el mantenimiento del sistema en cuanto a la adaptabilidad, a incorporar cambios, en especial en el módulo de asignación de choferes, siendo una funcionalidad que posiblemente se modifique con el uso de la aplicación cuando se identifiquen posibles mejoras. Por ello, se realizaron tareas de refactorización del código preexistente, incorporando el uso de algunos patrones de diseño en la codificación del ***Backend***.

El principal patrón aplicado para este fin, es llamado "***Strategy***". El mismo, se aplicó para abstraer la lógica de obtención de los candidatos a ser asignados para los viajes.

Se creó la interfaz llamada "***TravelCandidateStrategy***", cuyo cuerpo es el siguiente:

Ejemplo 7.1: Definición de inteface TravelCandidatesStrategy

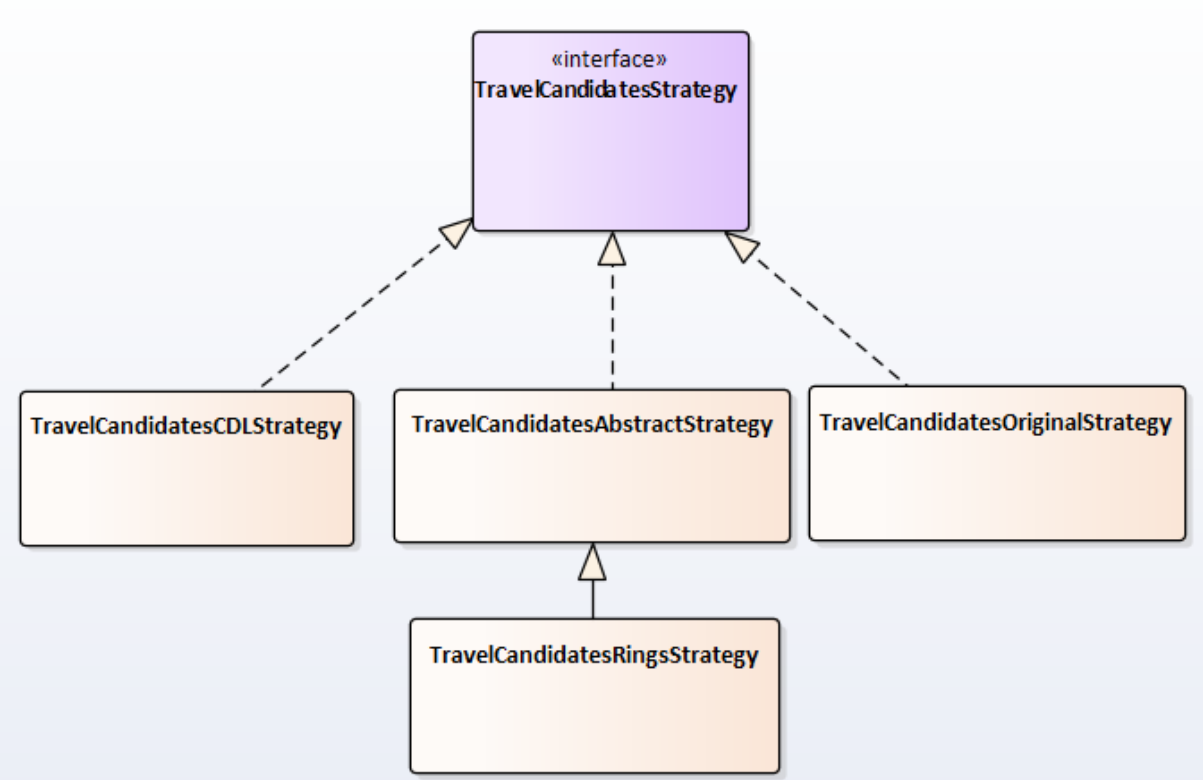
```
public interface TravelCandidatesStrategy {

  public List<TravelCandidates> getTravelCandidates(final Travel travel,
TravelCandidatesParametersToStrategy travelCandidatesParametersToStrategy)
throws InvalidCarrierStateException, DriversNotAvailableException ;

}
```

El parámetro ***travel*** del método ***getTravelCandiates,*** se utiliza para compartir los datos del viaje a asignar, y ***travelCandidatesParametersToStrategy,*** se utiliza para parametrizar servicios que puedan ser utilizados por las diferentes estrategias para resolver la generación de los candidatos.

Las clases que se fueron implementando para resolver las estrategias se pueden representar con el siguiente diagrama de clases:



Como puede deducirse, el uso de este patrón facilitó la posibilidad de cambiar fácilmente la forma en que se asignan los pedidos solicitados, sin necesidad de realizar grandes cambios de códigos preexistentes, e incluso posibilitó parametrizar qué estrategia usar en cualquier momento, brindando así mayor flexibilidad al sistema.

CAPÍTULO 8: ESTRATEGIAS DE SOLUCIÓN A PROBLEMAS DE PERFORMANCE

Para resolver o mitigar problemas de performance se fueron identificando una serie de acciones a implementar, destacaremos algunas de ellas en el presente capítulo.

ELIMINACIÓN DE LOS LOGS DE LA APLICACIÓN

Cierta parte del sistema registra la ubicación de los choferes con bastante frecuencia, está información es útil para realizar varios reportes que los empleados gerenciales necesitan verificar. Con el correr del tiempo, este log, se vuelve muy grande, además, ciertos registros contenidos en él, pierde utilidad. Por ello, se implementó un proceso recurrente que elimina información innecesaria, cuando la misma tiene cierto grado de antigüedad.

La siguiente porción de código fuente es un ejemplo de uno de los procesos desarrollados:

Ejemplo 8.1: Definición de proceso de truncado de información

```
package com.xxx.xxx.jobs;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.joda.time.DateTime;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Scheduled;

import java.util.Date;

import com.xxx.xxx.services.LogService;

@Component
public class TruncateTempTablesCron {
 Logger logger = LoggerFactory.getLogger(TruncateTempTablesCron.class);
```

```
@Autowired
LogService logService;

@Scheduled(cron = "0 0 3 * * ?", zone = "GMT-3:00")
public void run() {
    logger.info("Iniciando Truncando Tablas Temporales:" + new Date());

    DateTime now = new DateTime();

    int deleted = this.logService.deleteByDateBefore(now.plusMonths(-
2).toDate());

    logger.info("Finalización del proceso de Truncando Tablas Temporales: " +
deleted + " log driver location borrados");
  }
}
```

El anterior proceso se ejecuta diariamente a las 3 AM,(hora de Argentina) y elimina datos cuya antigüedad sea mayor a 2 meses.

Con este tipo de componente, se pudo reducir el tiempo de uso de la base de datos, al realizar búsquedas de información en las tablas relacionadas.

OPTIMIZACIÓN DE CONSULTAS AL ORM

Otra forma de optimizar los tiempos de respuesta y evitar cierto uso de memoria, fue limitar el uso exhaustivo de objetos "***del modelo***", ya que son hidratados puramente por el motor de ***ORM*** [13], (una tecnología usada en la mayoría de los lenguajes de programación orientados a objetos), y en lugar de ellos, realizar consultas más específicas mediante el uso de nuevas clases intermedias.
Estas clases respetan el patrón de diseño conocido como ***DTO,*** siglas de "***data transfer object***". Se crean objetos de las mismas solamente para transferir cierta información de las entidades del modelo del sistema.

Como **desventaja**, de usar este patrón se genera una mayor cantidad de clases, lo que podría atentar contra la legibilidad del código.

[13] ***ORM***: de sus siglas en inglés ***Object-Relational mapping***, es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.

Como **ventaja**, se evita una sobrecarga de trabajo, la que sería generada en forma automática por el ***ORM***, en nuestro caso es ***Hibernate*** en su versión 5.4.1 integrado con ***Spring JPA***[14] 1.7. Lo que se omite es, analizar e hidratar los objetos de las clases originales y sus relaciones, hecho de ocurrir podría también requerir la realización de varias consultas a la base de datos.

La forma de solucionar esto, fue instanciar estos objetos DTO en base al resultado de las consultas a datos de las entidades del negocio, como se puede en el siguiente fragmento:

Ejemplo 8.2: **Ejemplo de uso de DTO en consultas JPA**

```
@Query("SELECT new com.xxx.xxx.entities.driver.DriverAccountDTO(d.id, d.state,
d.operability) FROM DriverAccount d WHERE d.id = ?1")
DriverAccountDTO getBasicDriverStatus(Long id);
```

Se obtiene datos de la tabla relacionada a la entidad ***DriverAccount*** instanciando una clase ***DTO DriverAccountDTO***.

De esta forma no se trabaja con la clase ***DriverAccount,*** que tiene una gran cantidad de datos almacenados en la base como también y muchas relaciones a otros objetos del modelo.

Las diferencias en los tiempos de ejecución pueden ser considerables cuando se opera con una gran cantidad de objetos, por ejemplo, al realizar reportes que contengan muchos registros para procesar.

DEFINICIÓN DE NUEVOS ÍNDICES EN EL ESQUEMA DE BASES DE DATOS

La mayoría de los sistemas proveen funcionalidades de búsqueda de información. En muchos casos, éstos datos, necesitan ser buscados usando propiedades que no representan la clave primaria de las tablas que lo contienen, por ejemplo, en una tabla llamada persona, se intenta buscar por el nombre en lugar del identificador de ésta.

[14] **JPA:** siglas del inglés Java Persistence Api, es la API de persistencia desarrollada para la plataforma Java Enterprise Edition. Es un que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard y Enterprise.

Las claves primarias poseen, en general y de forma automática, un índice generado por el motor de base datos que facilita la búsqueda de los registros dentro de las tablas. Éstos ayudan al motor a encontrar los registros relacionados en forma rápida, evitando leerlos a todos los hasta encontrarla. Podríamos considerar a los índices como una tabla simplificada y organizada que brinda mejoras en los tiempos de consultas de la tabla que están indexando.

Cuando se realizan búsquedas por propiedades que no tienen relación con algún índice, el tiempo de consulta puede prolongarse si la tabla contiene muchos registros para evaluar. Una forma de mejorar esta tarea es definir nuevos índices, justificándose aún más cuando las búsquedas a realizar se realizan con mucha frecuencia.

En el desarrollo del sistema, fue necesario incluir índices adicionales para poder, de forma más eficiente, implementar cierta funcionalidad, por ejemplo aquella que generan reportes.

A continuación, un ejemplo de implementación de un índice utilizando una anotación provista por la ***Java Persistence Api*** (***JPA***):

Ejemplo 8.3 Definición de índices mediante anotaciones de la API JPA

```
@Entity
@Table(name = "User", indexes = {
    @Index(name = "first_name_idx", columnList = "firstName", unique =
false),
    @Index(name = "last_name_idx", columnList = "lastName", unique =
false)
    }
  )
```

UTILIZACIÓN DE SERVICIOS DE TERCEROS, PARA ALMACENAR Y GESTIONAR TRANSFERENCIAS DE ARCHIVOS PESADOS

Otra estrategia abordada en el desarrollo del sistema, fue la integración con servicios externos para almacenar archivos, como imágenes de las identificaciones de los choferes y reportes pdf generados. En particular se utilizó un servicio provisto por ***Amazon*** llamado ***S3***.

La idea es aliviar el tráfico de datos hacia el servidor del ***Backend,*** y en lugar de ello, redirigir el tráfico a los servidores de terceros en la medida posible, en general solamente para archivos que pueden ser accedidos públicamente.

La forma de integrar con este servicio se realizó utilizando una ***JDK*** ofrecida por el proveedor. La misma está disponible para varios lenguajes de programación. En particular, para java, resultó de fácil utilización, como puede verse en el siguiente fragmento de código que muestra la función para persistir un archivo en el servicio:

Ejemplo 8.4 **Uso de la API Amazon S3 para carga de archivos**

```
public void upload(File toUpload, String key) {
  Region region = Region.US_EAST_2;
  S3Client s3 = S3Client.builder().region(region).build();


s3.putObject(PutObjectRequest.builder().bucket(getBucketName()).key(key).build(
),
      RequestBody.fromFile(toUpload));
  System.out.println("Uploaded file with key: " + key);
  s3.close();
}
```

El archivo de la variable ***toUpload,*** será persistido en el repositorio externo con un identificador contenido en la variable: ***key***.

USO DE REQUEST, PAGINADOS Y FILTRADOS

Otra forma de reducir los tiempos de respuesta, es utilizar lo que se denomina paginación y filtrado, (limita la cantidad de datos que se les proveen a los usuarios en un momento determinado), ésto sirve para reducir el tiempo de transferencia y procesamiento de la información.

El siguiente es un ejemplo de cómo se codificó la solución desde un ***endpoint*** de la aplicación:

Ejemplo 8.5: **Implementación de paginación en endpoints**

```
@GetMapping("{carrierAccountId}/branches/paginated")
public ResponseEntity<Page<CarrierBranchResponseDTO>>
getCarrierBranchesPaginated(
    @PathVariable(value = "carrierAccountId") Long carrierAccountId,
    @RequestParam Optional<Integer> page,
    @RequestParam Optional<Integer> size,
    @RequestParam Optional<String> column,
    @RequestParam Optional<Sort.Direction> sort,
    @RequestParam Optional<String> find) {
  PageRequest pageRequest;
    pageRequest = PageRequest.of(page.orElse(1) - 1, size.orElse(10),
sort.orElse(Sort.Direction.ASC),
        column.orElse("id"));
  Optional<CarrierAccount> carrierAccount =
carrierAccountService.findById(carrierAccountId);
  if (carrierAccount.isPresent()) {
    return new ResponseEntity<>(carrierBranchService
        .getCarrierBranchesByCarrierAccountPaginated(carrierAccount.get(),
pageRequest, find.orElse(""))
        .map(branch ->
CarrierBranchEntityToResponseMapper.toCarrierBranch(branch)), HttpStatus.OK);
  } else {
    return
ErrorsResponseBuilder.getNotFoundError(ErrorsResponseBuilder.SystemError.CA
RRIER_NOT_FOUND,
        "Carrier user not found");
  }
}
```

Básicamente, se cuenta con un tamaño de página," la página actual", y así se consulta por un determinado orden o el provisto por el cliente los registros que correspondan a la página solicitada.

Como alternativa o en conjunción con la paginación, se pueden realizar filtros obligatorios de ser seleccionados por el usuario, muy útil cuando los resultados posibles son realmente demasiados y que no caben siquiera en la página donde deberían mostrarse, en nuestro ejemplo anterior provisto por las propiedades ***find*** y ***column***. ***Find,*** sería un texto a buscar y ***column*** cuál es la propiedad del modelo que se está utilizando como filtro de búsqueda.

CAPÍTULO 9: CONCLUSIONES

Como resultado de la presente tesina, se generó una propuesta de solución a problemas surgidos en el desarrollo de una aplicación multiplataforma de movilidad de objetos y/o personas, comparando algunas alternativas implementadas en diferentes versiones de los algoritmos.

Se realizó un análisis acerca de las mejoras en el funcionamiento de una aplicación multiplataforma, explicando que tiene varias áreas a tratar, que incluyen entre otras: análisis de requerimientos, diseño de software, optimización de uso de bases de datos, economizar el uso de ancho de banda, memoria, CPU y/o energía en todos los dispositivos involucrados, tanto de los servidores de los sistemas como las PCs o los teléfonos móviles de los clientes. Contemplando estas cuestiones se aplicaron varias técnicas que utilizan estos recursos de la manera más provechosa posible.

Por otro lado, se explicaron algunas condiciones que favorecieron el desarrollo de la aplicación.

Se definió un caso de estudio de un sistema donde: se trató la temática de despachar envíos o traslados a clientes y que los mismos estén conectados con el sistema mediante aplicaciones móviles, definiendo cuestiones funcionales y no funcionales que afectaron la toma de decisiones para desarrollar la aplicación.

Se hizo hincapié también, en cuestiones de mantenimiento del sistema, mencionando y ejemplificando el uso de varios patrones de diseño que facilitaron la tarea de modificar la aplicación por cambios de requerimientos o mejoras que debían aplicarse de forma continua.

Se espera que esta tesis sea un ejemplo útil de referencia al iniciar un nuevo proyecto.

TRABAJOS FUTUROS

Cada una de las propuestas de solución puede ser mejorada en cuanto al diseño o las tecnologías aplicadas. En particular, el algoritmo de asignación de choferes podría mejorar los aspectos de equidad en la carga de trabajo, pero sin atentar con los tiempos de resolución de los viajes, el mismo tiempo a su vez, puede crecer en cuanto a complejidad de la lógica de selección, para evitar por ejemplo perder viajes por no encontrar choferes que brinden exactamente los mismos servicios que el usuario solicite, pudiendo asumir reducción de ganancias en algunos casos.

Debido a que la industria de software se caracteriza por evolucionar constantemente, los trabajos futuros también pueden abarcar mejoras en tiempos de respuesta, características de las soluciones y de las tecnologías utilizadas, por ejemplo, se podrían actualizar los ***frameworks*** utilizados para desarrollar las aplicaciones móviles, webs o de ***Backend,*** y obtener así posiblemente mejores prestaciones o brindar otro tipo de característica a los clientes del sistema.

BIBLIOGRAFÍA

Delia, L., Galdamez, N., Thomas, P., Corbalan, L., & Pesado, P. (2015, May). Multi-platform mobile application development analysis. In 2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS) (pp. 181-186). IEEE.

Bazán P. et al. (2017). Arquitecturas, Servicios y Procesos Distribuidos. Una Visión desde la construcción del software. Libro de Cátedra. Editado por EDULP (Editorial de la UNLP). ISBN 978-950-34-1520-7
http://sedici.unlp.edu.ar/bitstream/handle/10915/62354/Documento_completo.pdfPDFA.pdf?sequence=1

Arcidiacono, J. (2020). DEHIA: una plataforma liviana para definir y ejecutar actividades con intervención humana basadas en workflows (Tesis Doctoral, Universidad Nacional de La Plata).

Corral, L., Sillitti, A., & Succi, G. (2012). Mobile multiplatform development: An experiment for performance analysis. Procedia Computer Science, 10, 736-743.

Boyland, P. (2019). The state of mobile network experience.

Virues, R. A. (2005, 25 de mayo). Estudio sobre ansiedad. Revista PsicologiaCientifica.com, 7(8) http://www.psicologiacientifica.com/ansiedad-estudio

Statista. (3 jun. 2021), Smartphones: número de usuarios mundiales 2016-2021, https://es.statista.com/estadisticas/636569/usuarios-de-telefonos-inteligentes-a-nivel-mundial/. Se consultó el 5 sept. 2021.

BBVA (2019), Tyk y Kong: analizamos estos dos API Gateways | BBVA." 11 abr. 2019, https://www.bbva.com/es/tyk-kong-analizamos-estos-dos-api-gateways/. Se consultó el 7 sept. 2021.

Paradigmadigital (2021), API Management: ¿qué es y para qué sirve? - Paradigma Digital." https://www.paradigmadigital.com/dev/api-management-que-es-y-para-que-sirve/. Se consultó el 7 sept. 2021

Amazon (2021), "Amazon EC2." https://aws.Amazon.com/ec2/. Se consultó el 7 sept. 2021.

(Google, 2021) «Firebase Cloud Messaging». Google Developers (en inglés). Consultado el 28 de mayo de 2016.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). Elements of reusable object-oriented software (Vol. 99). Reading, Massachusetts: Addison-Wesley.

Li, N., Du, Y., & Chen, G. (2013, December). Survey of cloud messaging push notification service. In 2013 International Conference on Information Science and Cloud Computing Companion (pp. 273-279). IEEE.

"Apache Cordova." https://cordova.apache.org/docs/es/latest/cordova/storage/localstorage/localstorage.html. Se consultó el 22 sept. 2021.

Diaz, A. I. (2012). Introducción a los cálculos geoespaciales en C. In XV Concurso de Trabajos Estudiantiles (EST 2012)(XLI JAIIO, La Plata, 27 al 31 de agosto de 2012).

Open Signal (2020). "The State of Mobile Network Experience 2020: One year into the 5G ...." https://www.opensignal.com/reports/2020/05/global-state-of-the-mobile-network. Se consultó el 25 oct. 2021.

Oracle (2021), "What is SaaS (Software as a Service)? - Oracle." https://www.oracle.com/applications/what-is-saas/. Se consultó el 18 oct. 2021.