Universidad Zaragoza
1542

Trabajo Fin de Grado

Aplicación móvil para solicitar un servicio de taxi sin intermediarios

Mobile application to request without intermediary a taxi service

Autor

Andoni Salcedo Navarro

Directores

Pedro Álvarez Pérez-Aradros

Titulación

Grado en Ingeniería en Informática

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021/2022

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. 78772400E ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de Estudios de la titulación de

Grado en Ingeniería Informática (Título del Trabajo)

Aplicación móvil para solicitar un servicio de taxi sin intermediarios / Mobile application to request without intermediary a taxi service

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 8 de Junio de 2022

Fdo: Andoni Salcedo Navarro

Aplicación móvil para solicitar un servicio de taxi sin intermediarios

RESUMEN

El avance de la tecnología ha conllevado la normalización del uso de *Smartphones* en las actividades cotidianas. Tareas que en el pasado requerían intermediarios y ahora son realizadas directamente a través de la Web, simplificando generalmente la comunicación y reduciendo el tiempo de espera que es necesario para finalizar la tarea. Por ello, es normal que aparezcan alternativas en sectores que aún siguen manteniendo estos intermediarios.

En el sector del transporte público han surgido diversas empresas que han revolucionado el mercado, ofreciendo facilidades en la utilización de estos servicios en comparación a los métodos convencionales que consisten en contactar mediante una llamada telefónica con la asociación que gestiona la flota de taxis de la ciudad en la que se desea reservar el vehículo.

Este trabajo de fin de grado (TFG) tiene como objetivo presentar una aplicación que gestione el proceso, reserva de un servicio de taxi y asignación de un taxista a un cliente. Para lograrlo, se hace uso de tecnología Web para ofrecer al usuario la mejor experiencia posible y a la vez facilitar al taxista su trabajo.

Se han desarrollado dos aplicaciones para dispositivos móviles con sistemas operativos Android. La primera orientada a la reserva de taxis con la finalidad de que lo use el cliente que requiera el servicio. La segunda para los taxistas que ofrezcan el servicio de transporte.

Adicionalmente y como parte del desarrollo se han llevado a cabo un conjunto de experimentos, que recogen simulaciones del comportamiento del sistema en un entorno real así como pruebas que buscan encontrar los límites del sistema.

Índice

Capítulo 1

Introducción

En este capítulo se expone el contexto del trabajo junto a la motivación y el problema que se aborda. Adicionalmente, se realiza un análisis del mercado con el que se extrae los productos existentes. Por último, se presenta la estructura general de la memoria.

1.1. Contexto y Motivación del trabajo

La mayor parte de asociaciones de taxistas que ofrecen su servicio en las ciudades españolas dependen de alguna centralita que actúa de intermediario en el proceso de reserva de taxi. La finalidad de dichas centralitas es avisar a los taxistas de que un cliente desea utilizar el servicio de taxis e informar cuál es la ubicación del cliente.

En el contexto de Zaragoza, la operativa de reserva de taxi es llevada a cabo por distintas cooperativas de *Radiotaxi*, que basa su funcionamiento en sistemas de gestión de flotas como *SIDUS* o *AUGIA*, que consisten en equipar a los vehículos con un dispositivo de localización que envía su posición a la central, en donde distintos operadores recogen las llamadas, atienden los avisos e informa a los taxistas más cercanos. Estos sistemas permiten a los ciudadanos de Zaragoza hacer uso del servicio de taxis mediante una llamada telefónica. Sin embargo, no existe una aplicación que facilite a un cliente reservar un taxi sin necesidad de intermediarios.

La motivación del proyecto es ofrecer una alternativa al sistema tradicional de reserva de taxi a través de una aplicación que simplifique la comunicación entre el cliente que requiere de un servicio de transporte y el taxista.

1.2. Productos existentes en el mercado

En esta sección se va a analizar las diferentes alternativas que existen en el mercado de reserva de taxis en España.

- *PideTaxi*[1] es un proyecto llevado a cabo por la Asociación de *RadioTaxi* de España, ofrece ventajas al usuario como la elección del recorrido, el tipo de taxi que se necesita (gran capacidad, movilidad reducida), cálculo estimado del recorrido, pago a través de la aplicación y servicio de atención al cliente.

- *TxMad*[2] es un proyecto desarrollado por el Ayuntamiento de Madrid, que permite calcular la ruta del viaje ofreciendo distintas alternativas con tarifas diferenciadas, guarda los viajes anteriores y tramita la reclamación de objetos perdidos.

- *Cabify*[3] y *Uber*[4] son empresas VTC (Vehículo de Transporte con Conductor) que funcionan como punto de contacto entre usuarios y el grupo de conductores autónomos, aunque no sean empresas de taxi, han revolucionado el mercado reintentado el sistema de reserva de vehículos adaptándose a los avances tecnológicos.

- *Uber* y *Cabify* ofrecen ventajas al usuario como el acceso a la localización de los vehículos en tiempo, tarifas estáticas, evaluación del servicio tras haber hecho el viaje y la posibilidad de reducir el coste del trayecto compartiendo vehículo.

En este TFG se plantea desarrollar una aplicación que introduzca las ventajas que ofrecen las VTC al mundo de los taxis y pretende ser competencia de soluciones ya existentes, añadiendo prestaciones como la visualización de rutas en tiempo real o la evaluación del servicio. Además, se prende que la aplicación funcione sobre cualquier ciudad con tan solo configurar algunos parámetros.

1.3. Estructura de la memoria del proyecto

La memoria está dividida en siete capítulos principales y un anexo:

- En el capítulo uno, en el que se ubica este apartado, consiste en la introducción del proyecto, donde se detalla la motivación y el problema que se aborda, además, se realiza un análisis de los productos existentes en el mercado.

[1] `https://pidetaxi.es/`
[2] `https://www.eltaxidemadrid.com/txmad-app-oficial-del-ayuntamiento-de-madrid/`
[3] `https://cabify.com/es`
[4] `https://www.uber.com/es/es-es/`

- En el capítulo dos se cubre la fase en la que se proponen los objetivos y se realiza el análisis previo al diseño, se presentan los requisitos de la aplicación y las tecnologías que se han utilizado en el proyecto.

- En el capítulo tres se detalla la arquitectura de software y los componentes que integran la aplicación final, así como ciertos detalles importantes en la implementación. Además, se analiza las interfaces de la aplicación.

- En el capítulo cuatro cubre la fase de despliegue en la nube, se exponen distintos tipos de despliegue que se han probado para la aplicación.

- En el capítulo cinco se procede a la fase de evaluación en la que se describen múltiples experimentes que se han llevado a cabo para probar el sistema y sus límites.

- En el capítulo seis se explica como se ha gestionado el proyecto y la metodología seguida para su finalización del proyecto, así mismo se aporta vistas temporales sobre su planificación, las horas de dedicación y las dimensiones que alcanza.

- En el capítulo siete se trata las conclusiones extraídas, así como los conocimientos adquiridos y las líneas de trabajo futuras.

- En el primer anexo se expone la documentación de la API asíncrona del servidor Web. Se muestran los eventos que controlan el ciclo de vida de reserva de taxis junto a ejemplos de utilización.

Capítulo 2

Análisis del sistema

En este capítulo se presentan los objetivos de este TFG, así como el análisis de requisitos que debe satisfacer la aplicación y la descripción de la tecnología que se va a utilizar para implementar dichos requisitos.

2.1. Objetivos

El objetivo principal de este trabajo es programar un sistema que facilite a un cliente reservar un taxi sin necesidad de intermediarios (por ejemplo, un centralita de taxis). El sistema consiste de dos aplicaciones móviles, una para el cliente y otra para el taxista, y un servidor accesible vía Web que proporciona la funcionalidad que sustenta la operativa de la solución. Concretamente, los objetivos son cinco:

1. Desarrollar un sistema que simplifique la comunicación entre un cliente y los conductores de taxis conectándolos directamente. Los clientes pueden solicitar un servicio de taxi y los taxistas tienen la opción de aceptar un servicio próximo. Hace uso de algoritmos de geolocalización para avisar de la petición de un nuevo servicio a los conductores más cercanos.

2. Diseño y desarrollo de una aplicación donde un cliente registrado puede solicitar un nuevo servicio de taxi, realizar seguimiento de su solicitud (si está aceptada o cuánto tiempo resta hasta que llegue el taxista que la aceptó), valorar un servicio de taxi y consultar el histórico de sus viajes.

3. Diseño y desarrollo de una aplicación donde un taxista registrado puede ofrecer su servicio de transporte, acceder a información de los clientes disponibles (si se encuentran próximos a la localización del taxista), valorar el comportamiento de un cliente y consultar el histórico de sus viajes.

4. Diseñar y programar un servidor que proporcione funcionalidad para asignar clientes candidatos a taxistas en base a criterios de geolocalización y calidad del servicio, puntuar la fiabilidad de un clientes en la solicitud de servicios y la experiencia ofrecida por los taxistas.

5. Diseñar y desplegar un sistema con una arquitectura basada en microservicios utilizando tecnologías *Cloud* para facilitar la escalabilidad y la adaptabilidad de la aplicación en distintos escenarios.

2.2. Requisitos de la aplicación

Tras analizar los objetivos se puede apreciar que existen tres elementos software que integran la solución; el sistema, el cliente y el taxista. Por ello se van a distinguir los requisitos en tres tablas. El resumen funcional de los objetivos de la aplicación del cliente se traduce en los requisitos recogidos a continuación (véase Tabla 2.1).

Código	**Descripción**
RF-1	El cliente debe registrarse en el sistema utilizando un correo electrónico antes de contratar un servicio.
RF-2	El cliente debe acceder al sistema utilizando un correo electrónico y una contraseña.
RF-3	El cliente puede utilizar la aplicación en Inglés y Castellano.
RF-4	El cliente puede solicitar el servicio de reserva de taxis con tan solo pulsar un botón.
RF-4.1	El cliente puede seleccionar la ubicación de partida y la ubicación de destino.
RF-4.2	El cliente puede visualizar la ruta de viaje antes de efectuar la petición de servicio.
RF-5	El cliente es notificado cuando el sistema le concede el servicio de taxi.
RF-6	El cliente puede acceder a la ubicación del taxista en tiempo real, desde el momento en el que el sistema le asigna un taxi hasta que haya finalizado el trayecto.
RF-7	El cliente debe autenticar al taxista mediante el escaneo de un código QR antes de coger el taxi.
RF-8	El cliente puede cancelar la reserva de un servicio de taxi antes de que el sistema le asigne un taxista.
RF-9	El cliente puede ofrecer una evaluación numérica al taxista entre cero (nota más baja) y cinco (nota más alta) una vez se haya finalizado el trayecto.
RF-10	El cliente puede acceder al historial de viajes realizados y obtener información sobre la ruta realizada, así como sobre el taxista que ha realizado el servicio.

Tabla 2.1: Requisitos funcionales del cliente

El resumen funcional de los objetivos de la aplicación del taxista se traduce en los requisitos recogidos a continuación (véase Tabla 2.2).

Código	Descripción
RF-1	El taxista debe registrarse en el sistema utilizando un correo electrónico antes de contratar un servicio.
RF-2	El taxista debe acceder al sistema utilizando un correo electrónico y una contraseña.
RF-3	El taxista puede utilizar la aplicación en Inglés y Castellano.
RF-4	El taxista puede ofrecer su servicio de reserva de taxi con tan solo pulsar un botón.
RF-4.1	El taxista puede contactar con los clientes más próximos que han solicitado el servicio de reserva de taxi.
RF-4.1	El taxista visualiza a los clientes mediante una representación de colores la puntuación de los clientes en viajes anteriores.
RF-5	El taxista puede seleccionar a un cliente para ofrecerle su servicio de transporte.
RF-5.1	El taxista puede acceder a las estadísticas del cliente, puntuación, número total de viajes y número de viajes cancelados.
RF-5.2	El taxista puede visualizar la ruta que desea realizar el cliente, ubicación de partida y ubicación de destino. Se le ofrece la trayectoria más corta para la realización de la ruta.
RF-6	El taxista puede ofrecer una evaluación numérica al cliente entre cero (nota más baja) y cinco (nota más alta) una vez se haya finalizado el trayecto.
RF-7	El taxista puede acceder al historial de viajes realizados y obtener información sobre la ruta realizada, así como sobre el usuario al que ha ofrecido el servicio.
RF-8	El taxista puede visualizar la ruta del cliente en tiempo real, desde el momento en el que selecciona a un cliente hasta el momento que finalice el trayecto.

Tabla 2.2: Requisitos funcionales del taxista

Queda fuera del alcance del proyecto el sistema de registro de cuentas de taxista mediante la verificación de la licencia de la ciudad donde opere.

El resumen funcional de los objetivos del sistema se traduce en los requisitos recogidos a continuación (véase Tabla 2.3).

Código	Descripción
RF-1	El sistema almacena un registro de todos los viajes realizados por los clientes y los taxistas, desde se guarda la fecha del trayecto, ubicación de origen y destino, datos del taxista, datos del usuario, puntuación del taxista al usuario y puntuación del usuario al taxista.
RF-2	El sistema asigna a los taxistas los usuarios más cercanos, a través de un algoritmo de geolocalización que consiste en dividir por regiones el entorno completo, donde se entiende que si un taxista está en la misma región que un cliente, este está cerca.
RF-3	El sistema analiza la confianza de un usuario mediante la media aritmética de las puntuaciones de los viajes realizados con anterioridad por el usuario.

Tabla 2.3: Requisitos funcionales del taxista

Los requisitos no funcionales son los atributos de calidad, describen la manera en la que debe operar el sistema. Se han dividido en 8 categorías según los distintos aspectos del sistema (véase Tabla 2.4).

Código	Descripción
RNF-1	El sistema se adaptará elásticamente al número de peticiones por segundo enviadas al servidor.
RNF-2	Es necesario disponer de conexión a Internet, ya sea por Wifi o por datos.
RNF-3	El sistema será compatible con dispositivos Android.
RNF-4	El sistema está disponible para todo dispositivo Android con versión igual o superior a Android 5.0 (*Lollipop*) .
RNF-5	La conexión irá cifrada mediante el protocolo HTTPS.
RNF-6	Las contraseñas están cifradas mediante algoritmo de clave hash seguro en la base de datos.
RNF-7	El sistema está disponible las veinticuatro horas del día, siete días a la semana.
RNF-8	El sistema soporta la opción multilenguaje íntegra los idiomas Castellano e Inglés.

Tabla 2.4: Requisitos no funcionales del sistema

2.3. Análisis de tecnologías

En esta sección se recogen las diferentes herramientas y tecnologías utilizadas para el diseño, implementación, experimentación, gestión, despliegue y documentación del proyecto que se han utilizado para desarrollar la aplicación y poder satisfacer todos los requisitos (véase figura 2.1).

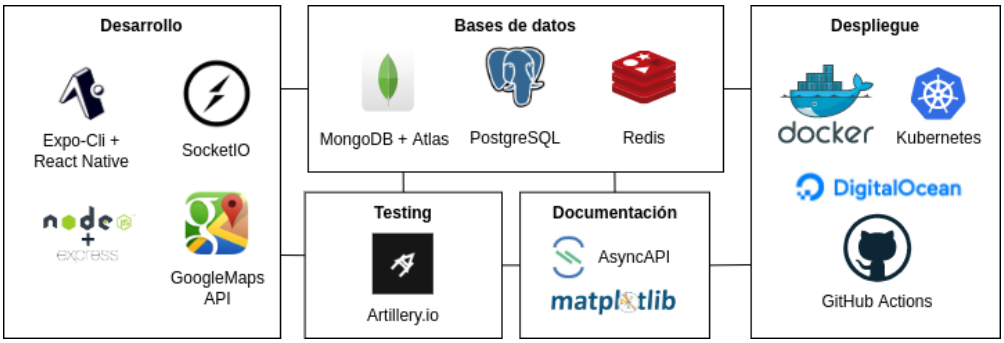


Figura 2.1: Resumen tecnologías y herramientas utilizadas para el proyecto

El papel que juega cada una de las tecnologías en el proyecto. Se dividen las herramientas en cinco cajas agrupadas por tecnologías de desarrollo, bases de datos, herramientas de testing, documentación y despliegue.

Las herramientas que se han utilizado para el desarrollo buscan simplificar el desarrollo de aplicaciones móviles y desplegar un servidor utilizando el mismo lenguaje de programación. En cuanto a las bases de datos, se persigue mezclar los beneficios de emplear bases de datos relacionales con la simplicidad que ofrecen las bases de datos no relacionales. Las herramientas que se aplican en el despliegue ofrecen grandes beneficios de desarrollo. A continuación, se ofrece una descripción de la utilización de cada tecnología, con el fin de situarla en el proceso de desarrollo.

2.3.1. Desarrollo

React Native

React native[1] es un framework de *JavaScript* que permite crear aplicaciones nativas para IOS y Android, se basa en la librería de React para la creación de componentes visuales.

Expo-Cli

Expo[2] es un framework para desarrollar aplicaciones multiplataforma. Este proporciona una capa de abstracción sobre la API de React Native con el que poder desarrollar aplicaciones para IOS y Android sin la necesidad de utilizar lenguajes nativos.

Express.JS

Express[3] es un framework Web, escrito en JavaScript que se ejecuta dentro del entorno de ejecución NodeJS que permite un desarrollo minimalista. Se caracteriza por tener la capacidad de estructurar una aplicación de manera ágil ofreciendo entre otras funcionalidades de enrutamiento y gestión.

Socket.io

Socketio[4] es un librería que permite comunicación de baja latencia, bidireccional y basada en eventos entre un cliente y un servidor. Ofrece una capa de abstracción sobre el protocolo WebSocket. Se ofrecen garantías adicionales como la implementación de distintas colas de tareas que permiten enviar eventos a distintos subconjuntos de clientes.

Google Maps API

La API de Google Maps[5] proporciona a través de la librería *React-Native-Maps* el acceso a un componente mapa que permite visualizar en tiempo real la geoposición del usuario en un entorno real. Además ofrece métodos para calcular y visualizar rutas.

2.3.2. Testing

Artillery.io

Artillery[6] es una herramienta de línea de comandos de código abierto diseñada específicamente para realizar pruebas de carga sobre aplicaciones web.

2.3.3. Bases de datos

MongoDB

Mongo[7] es un sistema para la gestión de datos NoSQL. Se trata de un modelo orientado a documentos que se almacenan en formato BSON que es una representación binaria de JSON. Entre sus principales ventajas se encuentra la flexibilidad de uso cuando se requiere manejar grandes volúmenes de datos, dado que proporciona una solución de almacenamiento de datos de alto rendimiento. Se utiliza **Mongoose** que es un ORM que proporciona una solución sencilla basada en esquemas para modelar datos en formato JSON.

MongoDB Atlas

MongoDB Atlas es un servicio de base de datos que permite desplegar una distribución de MongoDB en la nube que utiliza el despliegue *Multicloud* abstrayendo la comunicación con diversos distribuidores *Cloud*.

Redis

Redis[8][9] es un almacén de estructuras de datos en memoria que se utiliza para implementar distintos paradigmas. Su objetivo es lograr el máximo rendimiento trabajando con los datos en memoria. Entre todas las distintas herramientas que se ofrece Redis se va a hacer uso de dos:

- Base de datos en memoria caché.
- Mecanismo publicador subscriptor.

PostgreSQL

Postgres[10] es un sistema de gestión de base de datos relacional orientado a objetos y código abierto. Se utiliza **Sequelize** que es un ORM para JavaScript que soporta el controlador de base de datos postgres, facilitando la creación de tablas a través del modelado de objetos en JSON. Además ofrece consultas que simplifican la conexión con la base de datos.

2.3.4. Despliegue

Docker

Docker[11] es un proyecto que automatiza el despliegue de aplicaciones dentro de contenedores lo que proporciona una capa de abstracción y automatización de aplicaciones.

DockerHub

DockerHub[12] es un repositorio de imágenes de contenedores Docker que se ha utilizado para almacenar las últimas versiones de los repositorios, facilitando el despliegue continuo.

Kubernetes

Kubernetes[13] es un sistema de despliegue que automatiza, ajusta y escala aplicaciones basadas en contenedores.

Minikube

Minikube[14] permite desplegar un clúster de Kubernetes en la máquina local, haciendo más fácil el desarrollo con Kubernetes. Es donde se han desplegado las primeras versiones del sistema.

Digital Ocean

Digital Ocean[15] es un distribuidor *Cloud* que permite desplegar un clúster de Kubernetes en producción haciendo posible el acceso a la aplicación desde cualquier parte del mundo. Se ha utilizado para la realización de todos los test de la parte de experimentación.

GitHub Actions

GitHub te ofrece la herramienta de GitHub Actions[16] como plataforma de integración y despliegue continuo, permite automatizar pruebas y despliegues a través de flujos de trabajo.

2.3.5. Herramientas de documentación

AsyncApi

AsyncApi[17] proporciona la especificación de un lenguaje de definición para APIs asíncronas que permite un diseño más efectivo para las arquitecturas orientadas a objetos. Proporciona una forma de documentar, generar código y gestionar eventos.

Matplotlib

Matplotlib es un biblioteca completa para crear visualizaciones estáticas, animadas e interactivas en el lenguaje de programación Python. Se ha utilizado para la creación de gráficos con los datos extraídos en la parte de experimentación.

Capítulo 3

Diseño e implementación del sistema

En este capítulo se presenta la arquitectura software del sistema. Se presenta en detalle cómo se ha construido la aplicación partiendo de una visión general del sistema y profundizando en cada uno de los componentes que la integran, así como los puntos destacables de su implementación.

3.1. Entorno del sistema

Una visión simplificada del sistema se podría plasmar utilizando el siguiente diagrama (véase figura 3.1).

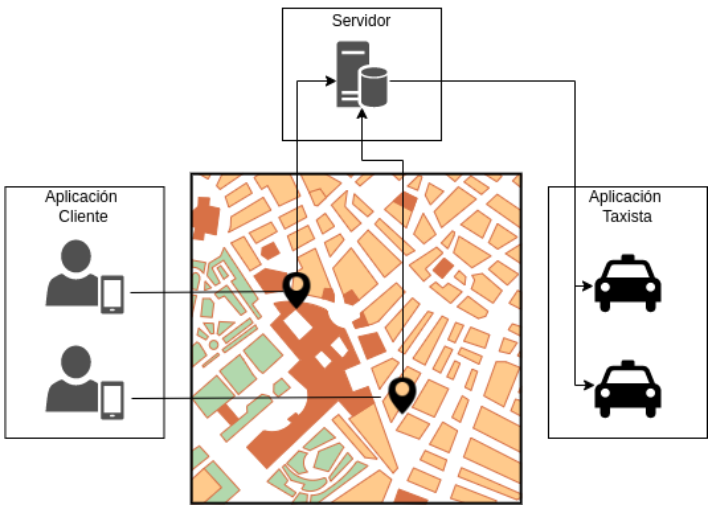


Figura 3.1: Diagrama general de la aplicación

El conjunto de funcionalidades que componen el sistema se divide en tres partes. Es necesario distinguir entre el servidor y las dos aplicaciones que integran la funcionalidad de reservar y ofrecer servicio de taxis.

El servidor es el encargado de la gestión de todo el proceso de reserva. Coordina la petición de reserva del cliente, muestra al taxista los clientes que se encuentran cerca, comunica al cliente con el taxista proporcionando una comunicación en tiempo real entre ambos, administra los datos de los usuarios y maneja la gestión de cuentas de usuario y sus credenciales.

Por otra parte, las aplicaciones móviles otorgan la capacidad tanto al cliente como al taxista de interaccionar con el servidor. La aplicación del cliente ofrece facilidades para utilizar el sistema de reserva de taxi, así como la del taxista permite dar el servicio de reserva de vehículos.

A continuación se ofrece un ejemplo de como sería la interacción entre las tres partes que componen el sistema en el proceso de reserva de un taxi.

Cuando un taxista decide ofrecer su servicio recibe por primera vez una lista de clientes cercanos que están esperando ser atendidos. En el momento que el cliente inicia el proceso de reserva de taxi, este introduce la ubicación de origen y de destino y envía la petición de reserva al sistema. El sistema informa a los taxistas cercanos que hay un cliente que requiere servicio, actualizando la lista de clientes cercanos. Además, se le ofrece al taxista información del cliente como el número de viajes, cancelaciones del cliente, una nota numérica entre cero y cinco que representa la satisfacción de otros taxistas en viajes anteriores del cliente y se le permite visualizar la ruta del cliente y acceder a información

Si a un taxista le conviene atender a un cliente, este informa al servidor de que le quiere ofrecer su servicio, a su vez el servidor informa al cliente de que un taxi se le ha sido asignado. A partir de ese momento empieza una comunicación entre ambos donde el taxista comparte su ubicación en tiempo real hasta llegar a la localización del cliente. Cuando esto ocurre, el cliente tiene que validar a través del QR del taxista la aceptación del servicio.

3.2. Arquitectura del servidor

El objetivo de esta sección es analizar la arquitectura con la que se ha construido el servidor ya que es donde están la mayor parte de los mecanismos que componen el sistema. Una visión conceptual de este se presenta en el siguiente diagrama (véase figura 3.2).

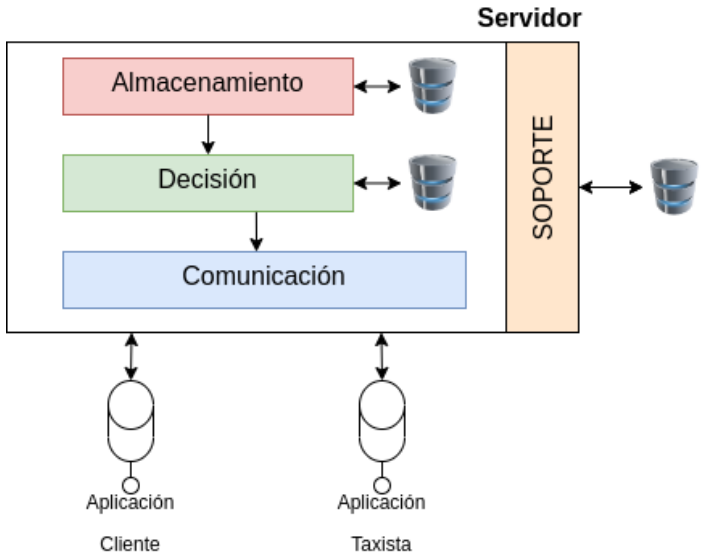


Figura 3.2: Diagrama de conceptual del servidor

En este diagrama se observa que el servidor se divide en cuatro piezas según la funcionalidad que cumplen.

Por un lado, está el **módulo de soporte** (sombreada de naranja), el cual se encarga de gestionar la parte de control de cuentas de usuario y verificación de credenciales.

El **módulo de comunicación** (sombreado de azul) es el que se encarga de controlar la comunicación entre los distintos usuarios del sistema y es el encargado de avisar a los taxis correspondientes de las peticiones de los clientes.

El **módulo de decisión** (sombreado de verde) es el que ayuda al módulo de comunicación a decidir qué clientes están cerca de los taxistas que se encuentran dentro del sistema. Concretamente, es el que decide a qué taxistas avisar cuando un cliente solicita un taxi.

El **módulo de almacenamiento** (sombreado de rojo) es el que guarda la información persistente de los usuarios del sistema.

Los módulos que se han analizado desde una visión conceptual del servidor son mapeados en los componentes reales que lo integran. A continuación se presenta el diagrama de componentes donde se observa con claridad cómo interactúan entre ellos (véase diagrama 3.3).

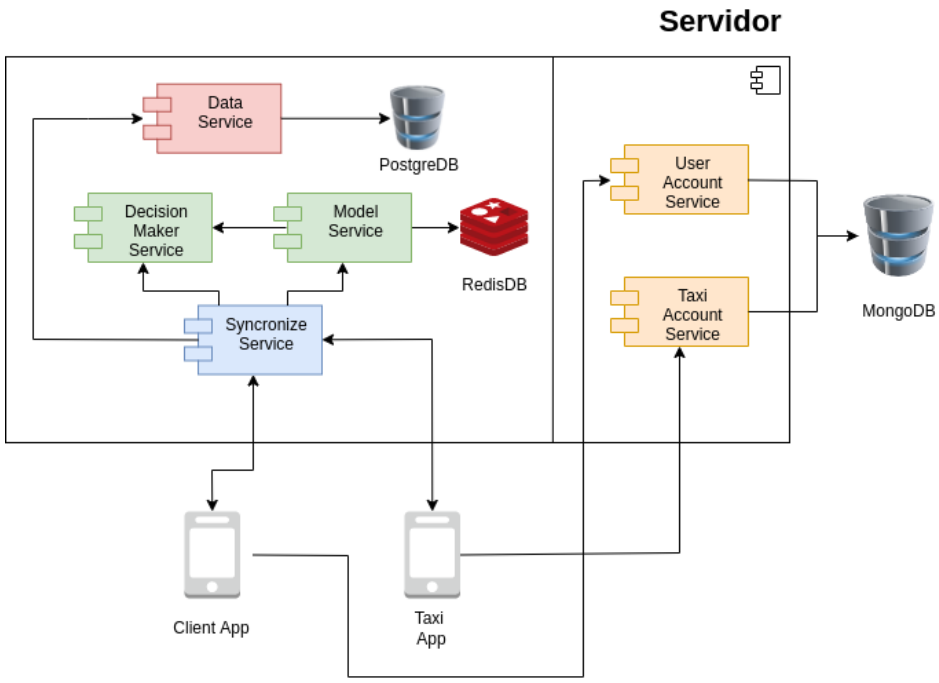


Figura 3.3: Diagrama de componentes de la aplicación

A continuación se ofrece una visión general de la función de cada componente en el servidor. En la sección 3.3.1 es donde se profundiza individualmente en cada uno de los componentes.

- El componente de *Syncronize* es el que gestiona la comunicación entre los clientes y los taxistas y gestiona la de reserva de un servicio. En él se establece una conexión bidireccional entre los clientes y los taxistas para proporcionar una actualización en tiempo real de las peticiones al sistema.
- El componente *Decision Maker* es el sistema de toma de decisiones. Este determina qué usuarios están más cerca de los taxistas según su posición geográfica.
- El componente *Model* es el modelo geográfico sobre el que se toman las decisiones. En él se almacena la información de los usuarios que están a la espera de que se le asigne un servicio.
- El componente *Data* almacena la información histórica y las estadísticas de los usuarios. Es el almacenamiento persistente del sistema donde se lleva un registro de todos los viajes realizados, cancelaciones y evaluaciones realizadas por taxistas y usuarios.
- Los componentes de *UserAccount* y *TaxiAccount* gestionan la sesión de los usuarios y los taxistas de la aplicación. Estos componentes ofrecen las funcionalidades de registro, inicio de sesión, verificación y gestión de cuentas.

El servidor se ha construido basándose en servicios que, interconectados, ofrecen la plena funcionalidad del sistema. Esta arquitectura modular ha tenido una notable importancia para el desarrollo del sistema, dado que una parte significativa de los objetivos están relacionados con el despliegue y la escalabilidad.

Estos requerimientos han sido alcanzados mediante una arquitectura basada en microservicios. Por una parte esta arquitectura, te ofrece escalabilidad dado que una aplicación modular se puede escalar horizontalmente cada parte según sea necesario, aumentando el número de módulos para acelerar el procesamiento. Por otra parte, el reducido tamaño de los servicios permite un desarrollo menos costoso que junto con la utilización de contenedores permite que el despliegue de la aplicación se lleve a cabo eficazmente.

Otra de las propiedades de los microservicios que han sido de utilidad en el proceso de desarrollo del sistema ha sido la reutilización de código, los componentes *UserAccount* y *TaxiAccount* son el mismo microservicio desplegado con distinta configuración para que el primero atienda las peticiones de los clientes y el segundo el de los taxistas.

3.3. Descripción detallada de los componentes

En esta sección se va a profundizar en las partes más técnicas de los componentes. Para cada uno de ellos se detalla la información más relevante de su comportamiento interno.

3.3.1. Componente *Syncronize*

El componente *Syncronize* se basa en un modelo arquitectural por eventos, que se controla con el mecanismo de *publicación/subscripción*[18]. Este patrón permite que una aplicación anuncie eventos de forma asíncrona a varios consumidores interesados sin la necesidad de emparejar a los remitentes con los receptores. Consiste en que una serie de suscriptores (en este caso los taxistas) se inscriben a un evento (en este caso una petición de servicio). El publicador (en este caso el cliente que requiere de un servicio de taxi) es el que notifica de un nuevo evento a todos los interesados, los suscriptores que están inscritos a dicho evento.

En este caso no tiene sentido que un taxista sea notificado de un evento que ocurre en un punto alejado de la ciudad, por lo que para evitar este tipo de situaciones se implementa el mecanismo *publicador/suscriptor*[19] utilizando colas de mensajes. En vez de que un suscriptor se inscriba a un evento, este se suscribe a una cola de mensajes en la que solo los clientes cercanos notifican el evento de petición de servicio, los componentes de toma de decisión (véase componentes 3.3.3 y 3.3.2) son los encargados de determinar cuales son los clientes cercanos. Un ejemplo del funcionamiento de este patrón se observa a continuación (véase figura 3.4).

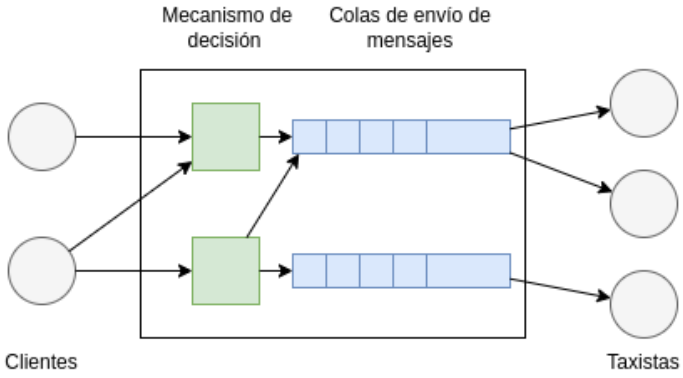


Figura 3.4: mecanismo publicador/subscriptor mediante colas

Para la implementación de las colas de mensajes se ha utilizado la librería *SocketIO* (véase 2.3.1). En ella existe el concepto de *Rooms*[20], que son canales de comunicación arbitrarios que se pueden usar para transmitir eventos a un subconjunto de clientes. Una cola de mensajes se corresponde a una *Room*, este concepto es exclusivo del servidor, por lo que el cliente es abstraído de la existencia de *Rooms*.

El componente *Syncronize* se encarga en realizar la comunicación integra entre el taxista y el cliente utilizando la librería *SocketIO*. Para ello se establece un protocolo que se divide en tres partes. A continuación se detalla cada una de las fases y los paradigmas internos que que se utilizan.

La primera fase consiste en establecer la conexión con el servidor. Para ello se utilizan *websockets* con los que se crea un canal bidireccional entre el servidor y el usuario. Una vez la conexión se realiza con éxito se inicia la segunda fase (véase diagrama 3.5). En el caso de los taxistas, el servidor los suscribe a la *Room* que les corresponde. En donde esperan recibir eventos de clientes que emitan en la misma *Room*. Mientras que en el caso del cliente este emite a la *Room* que le corresponde un evento de petición de servicio que será recibida por los taxistas que están conectados a dicha *Room*.

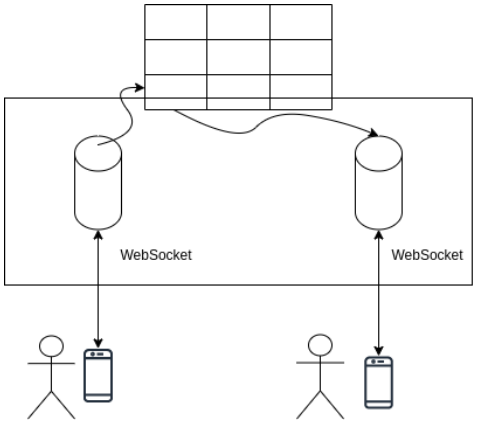


Figura 3.5: Segunda fase del protocolo de comunicación

SocketIO también implementa un mecanismo que permite el envío de mensajes a un usuario específico. En esto se basa la tercera fase de la comunicación entre el cliente y el taxista (véase diagrama 3.6). Una vez un cliente es asignado a un taxista, la comunicación pasa a ser privada entre ambos usuarios. Esta comunicación se basa en el envío de las coordenadas en tiempo real de ambos actores hasta el momento en que el taxi llega a las coordenadas de salida del cliente.

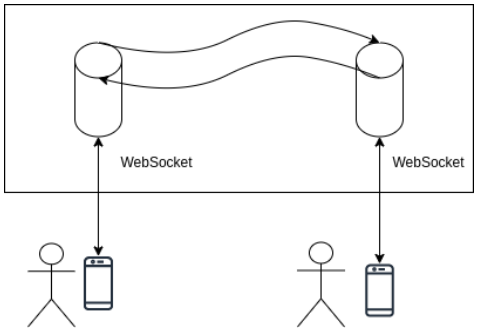


Figura 3.6: Tercera fase del protocolo de comunicación

3.3.2. Componente *Decision Maker*

El componente *Decision Maker* se encarga de la toma de decisiones relacionadas sobre la cercanía entre taxistas y clientes. Para ello se divide el espacio geográfico de la zona donde está disponible el servicio de taxi en regiones. Dentro del mismo cuadrante se considera que la distancia entre taxista y cliente es significativamente cercana como para ofrecerle servicio.

El componente requiere de los límites geográficos de la ciudad en la que se despliega (*Bounding Box*). Se utiliza la esquina superior derecha y la esquina inferior izquierda para trazar una cuadrícula con un número de cuadrantes establecido.

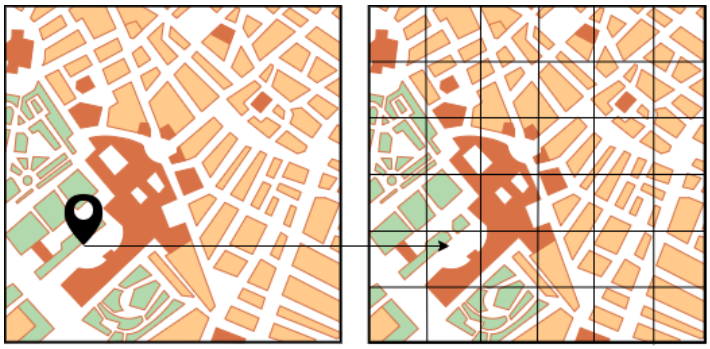


Figura 3.7: Mapeo de coordenada a cuadrante

El componente discretiza las coordenadas geográficas del usuario en un número fijo de cuadrantes (véase figura 3.7). Por cada cuadrante existe una *Room*. Dada la localización de un usuario se le asigna el cuadrante correspondiente. En donde en el caso del cliente se publica la petición de servicio en la *Room*, mientras que en el caso del taxista se suscribe a dicha *Room*.

El número total de cuadrantes establecidos en una ciudad puede ser modificado con el fin de mejorar las prestaciones de servicio. Cuanto mayor sea el número de cuadrantes en una ciudad menor será el número medio de taxistas por cuadrante, lo que produce que el evento de petición de servicio de un cliente sea recibida por menos taxistas, evitando sobrecargar el sistema. Por otro lado, existe la posibilidad de haya zonas en la ciudad donde no opere ningún taxista en ese momento, por lo que un evento de petición de servicio no tendría respuesta alguna.

En el caso en que ningún taxista se encuentre en el cuadrante del cliente el evento se envía a los cuadrantes colindantes hasta que la petición de servicio sea recibida por algún taxista.

3.3.3. Componente *Model*

El componente *Model* es el modelo geográfico sobre el que se toman las decisiones. Funciona como una memoria caché[8] que almacena un vector donde el índice del vector corresponde a un cuadrante, cada índice almacena mediante un *Sorted Set* la información y las estadísticas de los usuarios que están a la espera de que un taxista les ofrezca servicio en esa región.

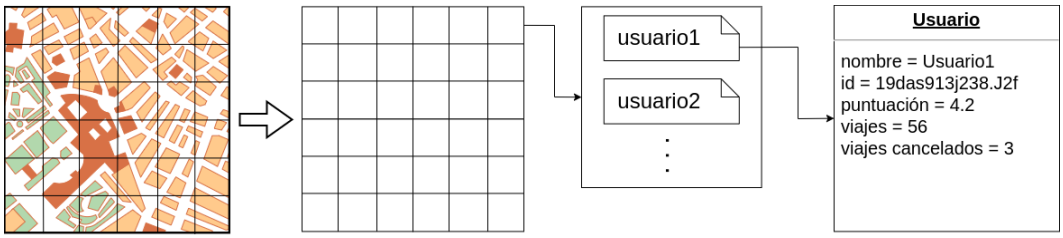


Figura 3.8: Diagrama de componentes de la aplicación

Existe una relación directa entre las *Rooms* y el índice donde se almacenan los usuarios. Los clientes que están en la misma región geográfica se introducen en el mismo índice, así pues, cuando un taxista se suscribe a una *Room* recibe la información de todos los clientes que están a la espera de recibir servicio.

Este componente también funciona como mecanismo de exclusión mutua, cuando dos taxistas quieren atender a la vez la petición de un mismo cliente, se actuará para que solamente un taxista sea el asignado para ofrecerle su servicio, evitando condiciones de carrera.

3.3.4. Componente *Data*

El componente *Data* se encarga de gestionar el almacenamiento del registro de viajes de los usuarios de manera persistente, así como sus estadísticas. Se modela utilizando una base de datos relacional con dos entidades principales que son el cliente y el taxista relacionados mediante una tercera entidad, la cual almacena una instancia de un viaje.

Este componente resuelve tres necesidades del sistema. La primera es guardar los datos personales de los usuarios (nombre, email, teléfono...). También se requiere tener el registro de los viajes realizados por cada usuario con la finalidad de poder consultar viajes realizados en el pasado. La tercera función cumple con el objetivo de implantar un sistema de valoración de usuarios, el cual se basa en puntuaciones. La puntuación de un usuario será la media aritmética de las puntuaciones otorgadas por otros usuarios en los viajes efectuados.

Este componente ofrece su funcionalidad a los competentes que toman decisiones (3.3.3 y 3.3.2) que son los que se encargan de procesar los datos.

El esquema relacional que se ha implementado (véase diagrama 3.9) muestra las relaciones que existen entre los distintos elementos que conforman el almacenamiento persistente.

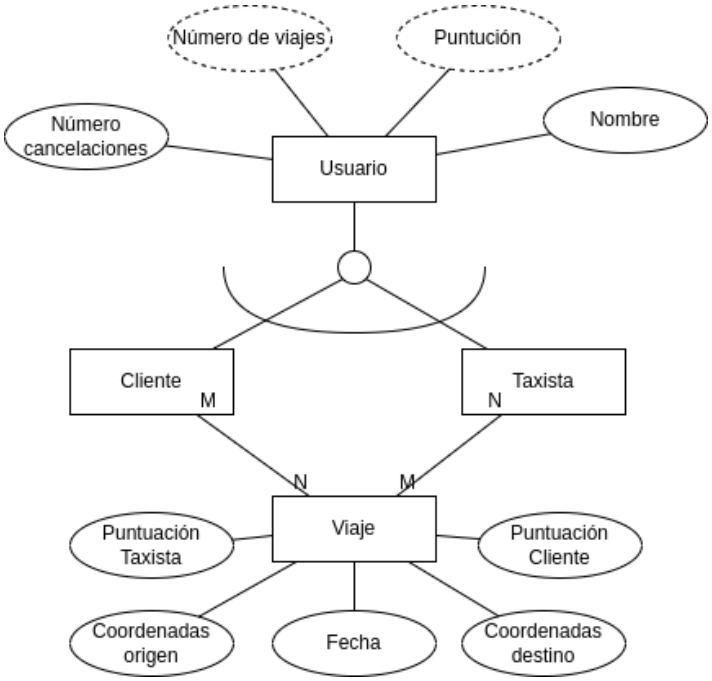


Figura 3.9: Entidad relación

3.3.5. Componentes *Account*

Los componentes *UserAccount* y *TaxiAccount* están basados en el mismo componente, pero se han configurado de manera que el *UserAccount* solo atienda las peticiones de los clientes y el *TaxiAccount* solo las peticiones de los taxistas.

Este componente es el encargado de gestionar las credenciales de los usuarios. El proceso de autorización se realiza utilizando tokens de sesión, los cuales proporcionan confidencialidad e integridad entre los usuarios y el servidor. Las claves se almacenan cifradas en la base de datos externa *MongoDB Atlas*, que se encarga de gestionar el registro de usuarios.

La autenticación del *UserAccount* se efectúa mediante el método *UserPassword*. Se toma como credenciales de usuario el correo electrónico y una contraseña introducida por el cliente.

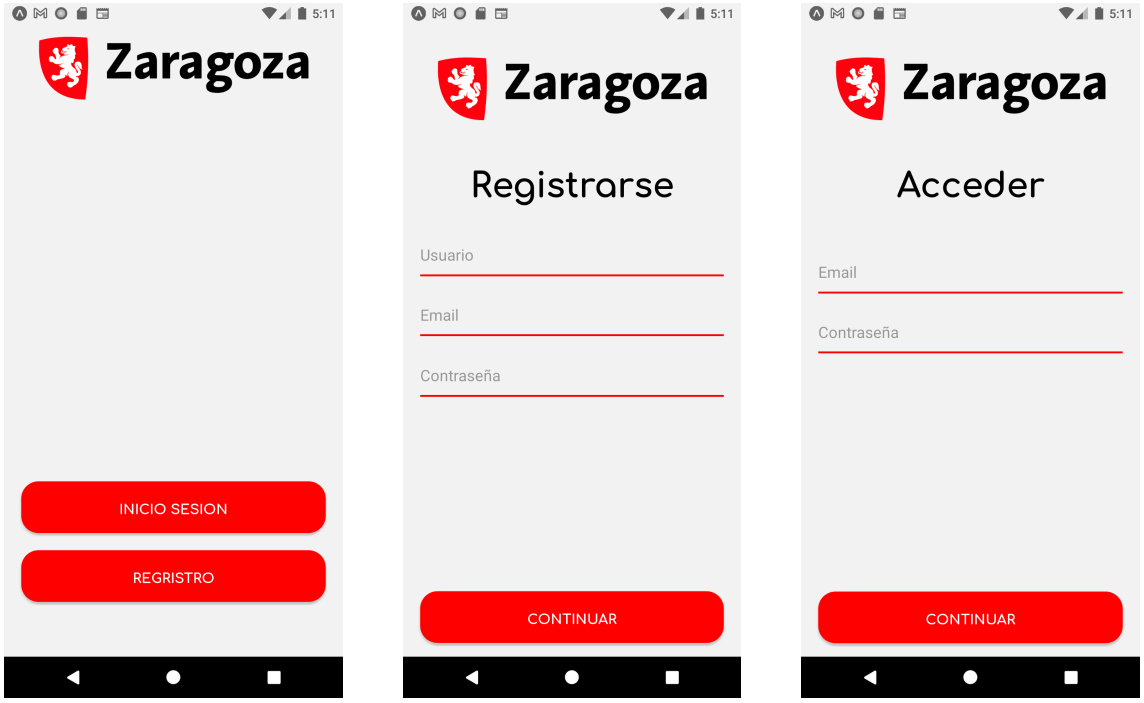
La autenticación del *TaxiAccount* se efectúa mediante el método de distribución de claves. Es necesario obtener una clave para poder darse de alta en el sistema. Esto cumple la necesidad de limitar los usuarios que pueden darse de alta como taxistas en el sistema, ya que solo los usuarios con licencia de taxi pueden funcionar como taxistas.

3.4. Aplicación móvil

En esta sección se va a analizar las interfaces gráficas tanto de la aplicación del cliente y la del taxista. La interfaz gráfica se encarga de proporcionar al usuario una ventana a través de la cual interactuar con el sistema para acceder a las características de la aplicación de una manera sencilla.

3.4.1. La aplicación para los clientes

A continuación se realiza una explicación acerca de las diferentes pantallas que se muestran cuando se pone en funcionamiento de la aplicación del cliente.



(a) Pantalla de bienvenida (b) Pantalla de registro (c) Pantalla de inicio de sesión

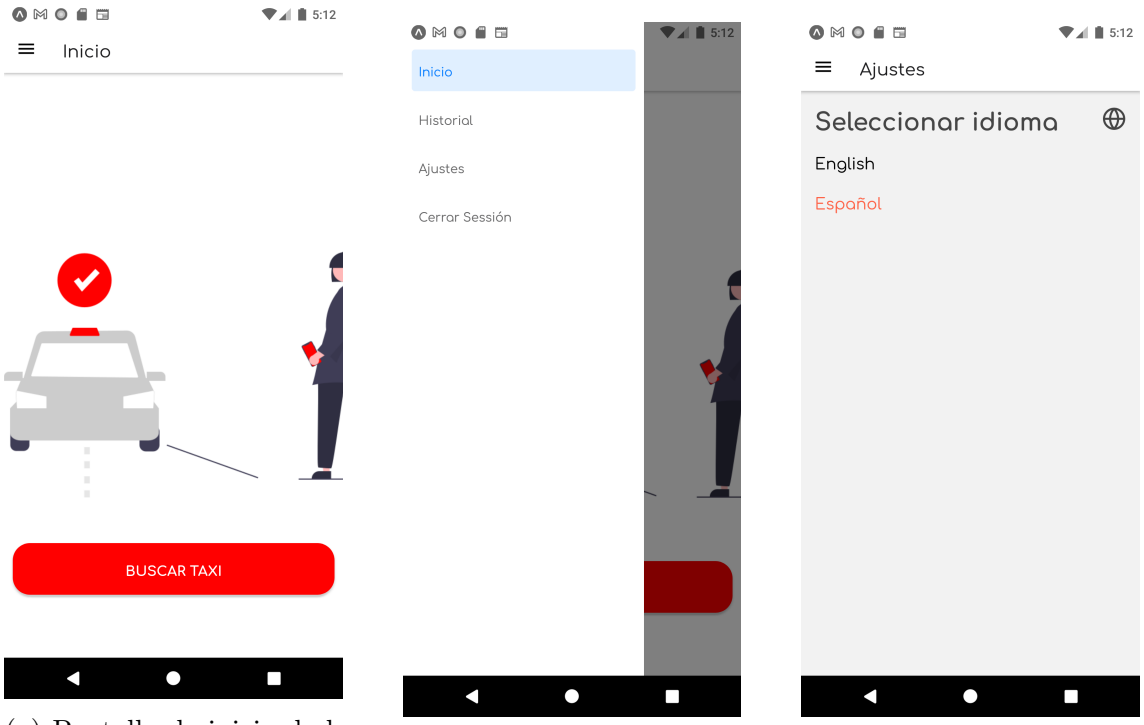
Figura 3.10

Cuando abrimos la aplicación la primera pantalla que se muestra es la de bienvenida (véase pantalla 3.10a), en esta aparece en la zona inferior dos botones rojos, el primero de ellos sirve para iniciar sesión, mientras que el segundo se utilizará en el caso de querer realizarse como nuevo registro de usuario.

Cuando se pulsa el botón de registro se abre una nueva pantalla(véase pantalla 3.10b). Para registrarse es necesario rellenar tres campos con los datos del usuario. El primero que debe completar es el nombre con el que quiere referirse el usuario, el segundo debe ser un email válido y que no haya sido usado anteriormente. Por último, se requiere establecer una contraseña para poder acceder posteriormente al sistema.

En la zona inferior de la pantalla hay un botón rojo que al pulsarlo te registra en el sistema y te redirige a la pantalla de bienvenida.

Si, por el contrario, se pulsa el botón de inicio de sesión, la nueva pantalla que aparece (véase pantalla 3.10c). Esta consta de dos campos a rellenar con los que poder acceder a la aplicación. En ellos se deberá introducir el nombre de usuario y la contraseña que se designó en el registro previo.



(a) Pantalla de inicio de buscar taxi

(b) Menú

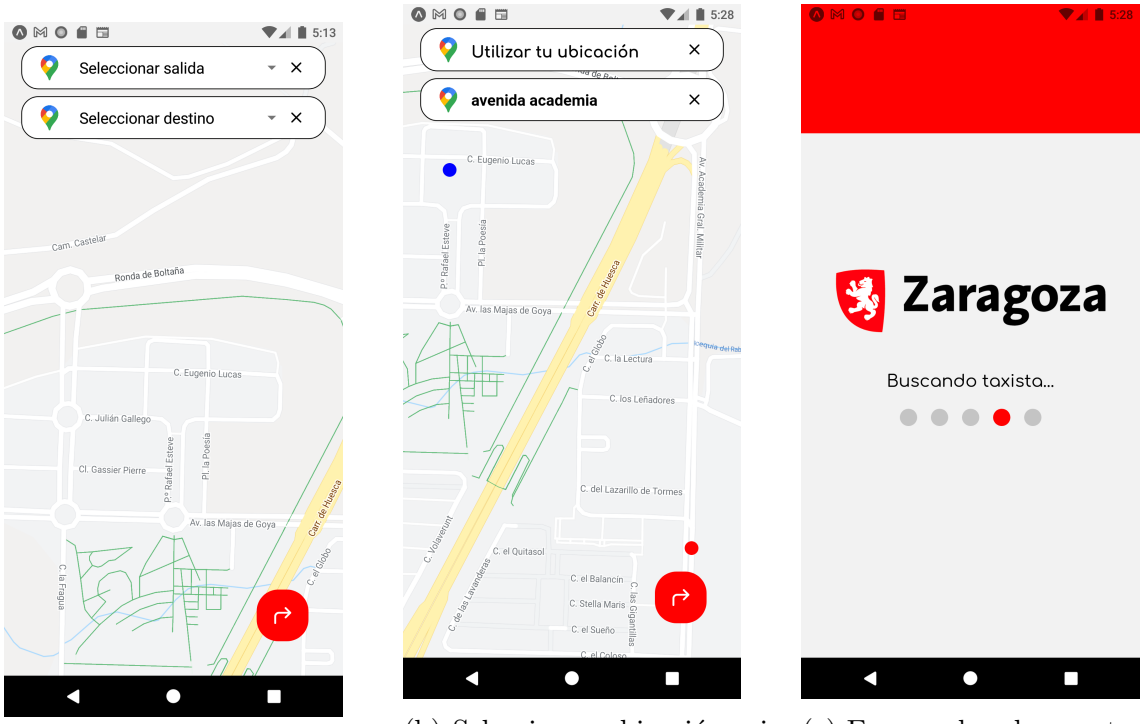
(c) Selección de idioma

Figura 3.11

Una vez se ha iniciado sesión, la aplicación te muestra una nueva pantalla (véase pantalla 3.11a). Esta es la página principal donde directamente se puede acceder al servicio de reserva de taxis. Además, se puede acceder al menú de la aplicación pulsando el botón que se encuentra en la esquina superior izquierda.

Al apretar el menú contextual aparecen las opciones del usuario (véase pantalla 3.11b). En él se permite acceder a recursos adicionales de la aplicación aparte del de reserva de taxis. Estos son el historial de viajes, los ajustes de idioma y el cierre de sesión.

En caso de querer configurar el idioma en el que se presenta la aplicación, se accede a la opción de configuración donde se abre una nueva pantalla (véase pantalla 3.11c) en la que se permite escoger el idioma de la aplicación entre Inglés y Español. Cuando se inicia la aplicación por primera vez se detecta el idioma del dispositivo móvil que ejecuta la aplicación y se utiliza ese por defecto.



(a) Buscar taxista

(b) Seleccionar ubicación origen y destino

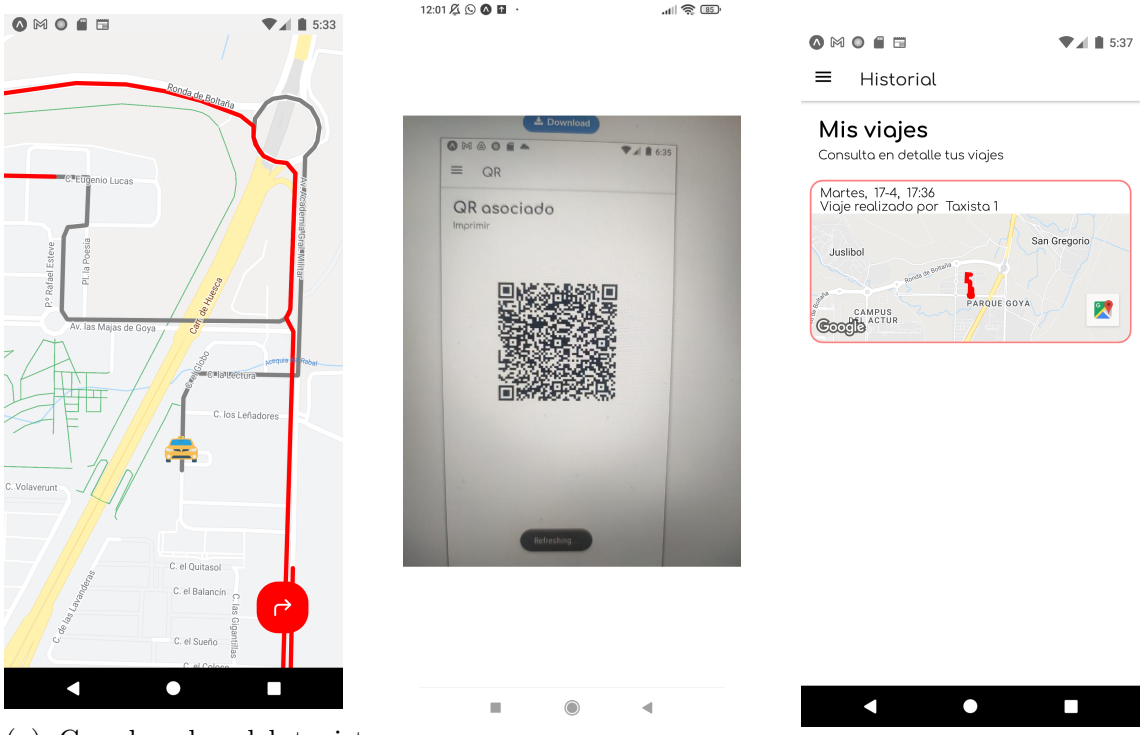
(c) Esperando a buscar taxista

Figura 3.12

Cuando se inicia el proceso de reserva de taxi. La aplicación redirige al usuario a una nueva pantalla (véase pantalla 3.12a). En esta se muestra un mapa de la ciudad de Zaragoza donde se puede seleccionar tanto las coordenadas de salida como las de destino a través de dos selectores que se encuentran en la parte superior de la pantalla, cuando pulsas en ellos aparece una pestaña en la que se ofrecen dos opciones al usuario. La primera de ellas es para escoger la ubicación actual del usuario como coordenadas y la segunda permite al usuario introducir por teclado en lenguaje natural la ubicación.

Una vez se han seleccionado ambas ubicaciones aparece la ruta del trayecto por pantalla (véase pantalla 3.12b). El punto azul muestra la ubicación de salida, mientras que el punto rojo la de destino. Una vez el usuario ha comprobado que ambas posiciones son correctas para iniciar la búsqueda de taxi, debe pulsar el botón situado en la zona inferior derecha de la pantalla.

Este redirige al usuario a una nueva pantalla (véase 3.12c) en la cual debe esperar a que un taxi atienda su petición de servicio. En ella aparece un mensaje que hace referencia a que se está buscando un taxista junto a un indicador de carga.



(a) Coordenadas del taxista en tiempo real

(b) Introducir QR de confirmación

(c) Historial de viajes

Figura 3.13

Una vez se ha encontrado un taxista interesado en realizar el trayecto propuesto, se abre una nueva ventana (véase pantalla 3.13a). En ella se muestra en tiempo real la posición del taxista junto al trayecto que el taxista ha de efectuar hasta llegar a las coordenadas del usuario (marcado en gris) y la ruta más corta desde la ubicación de salida hasta al destino (marcada en rojo).

Cuando el taxista ha llegado a la posición del usuario, este debe validar la llegada del taxista. Para ello se pulsa el botón de la zona inferior derecha de la ventana (véase pantalla 3.13b) donde se abre un escáner de códigos QR con el que el usuario debe leer el QR asociado al taxista para validar el viaje.

Una vez se ha concluido el viaje, este aparecerá registrado en el historial (véase pantalla 3.13c) donde se puede acceder los datos de todos los viajes anteriores efectuados por el cliente.

A continuación se muestra el mapa de navegación de la aplicación del cliente donde se muestra la interacción entre las diferentes pantallas que forman la aplicación (véase figura 3.14).

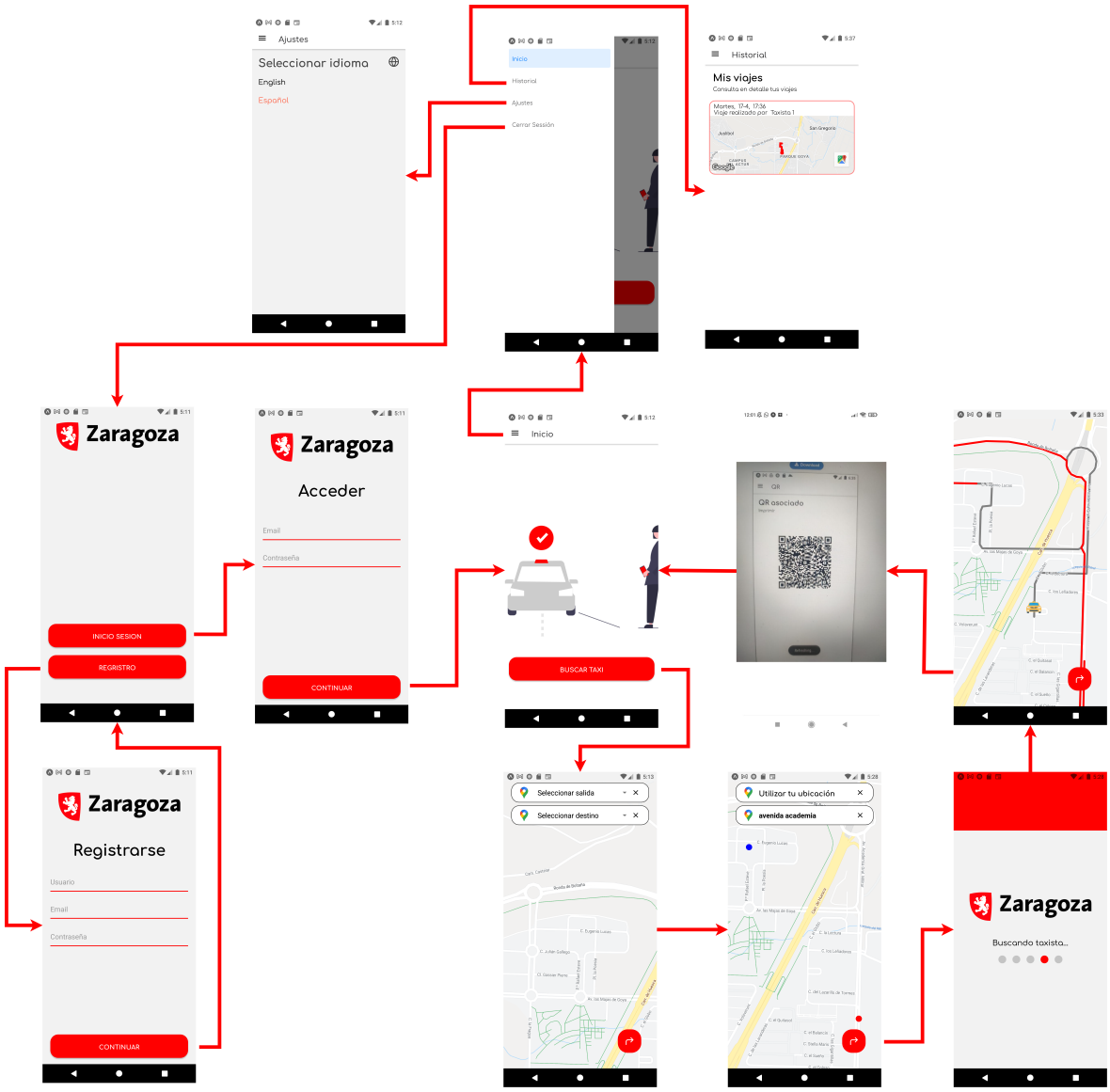
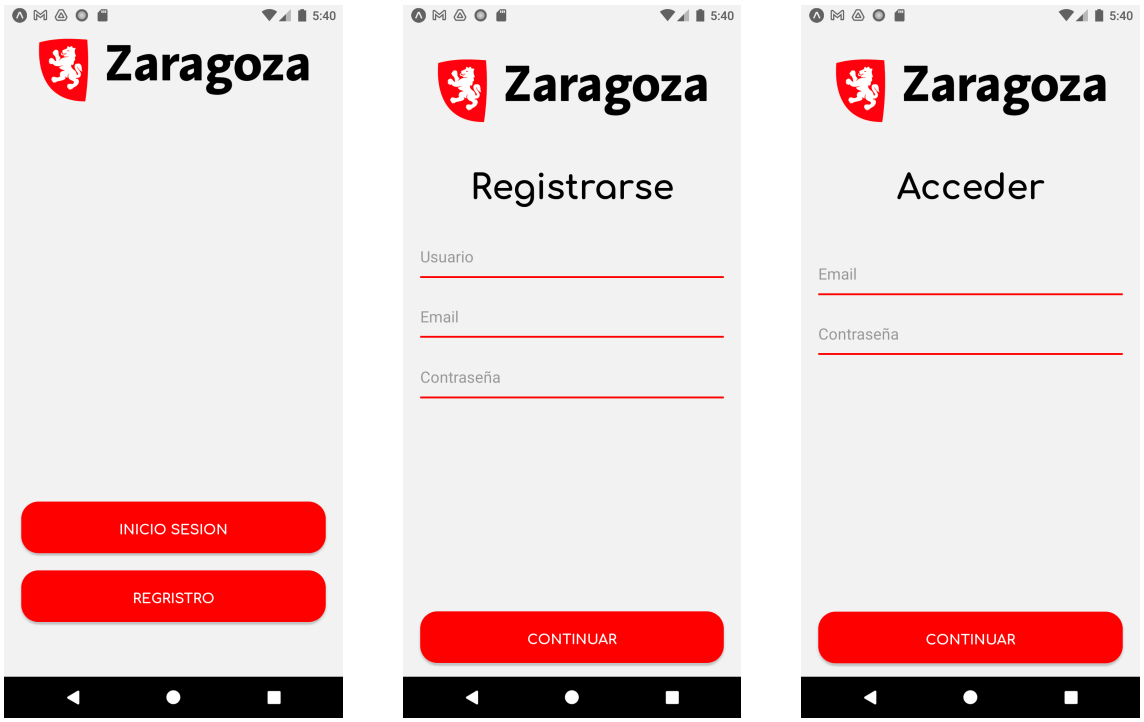


Figura 3.14: Mapa navegación cliente

3.4.2. Interfaz del taxista

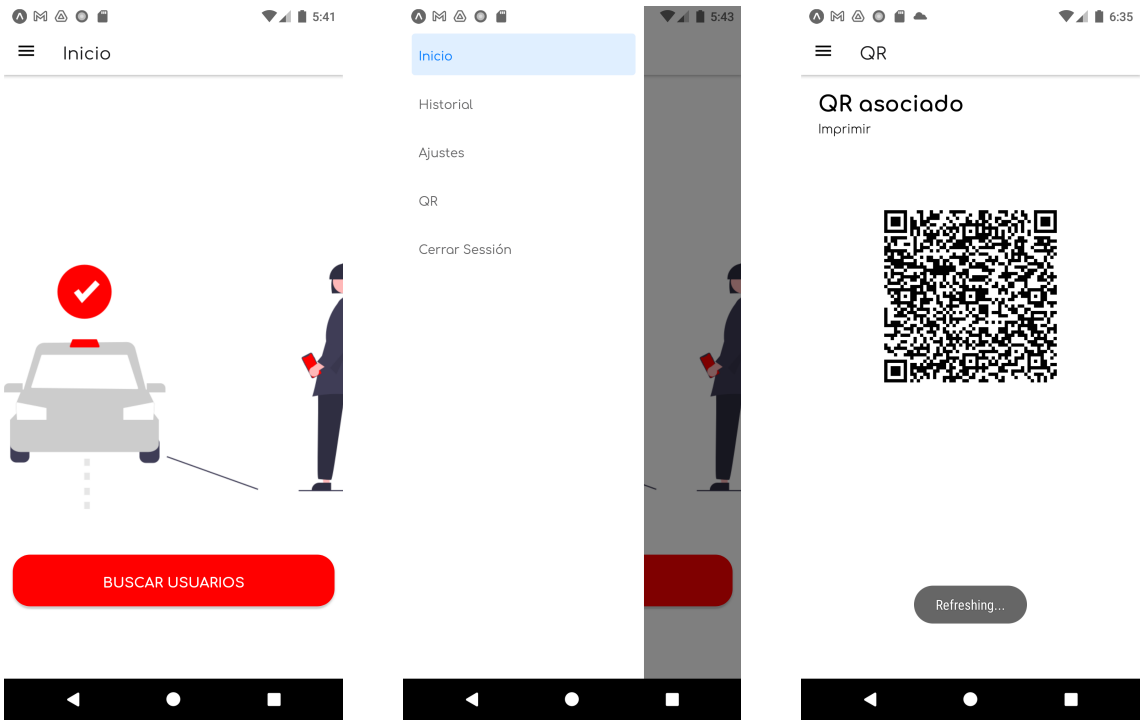
A continuación se realiza una explicación acerca de las diferentes pantallas que se muestran cuando se pone en funcionamiento de la aplicación del taxista.



(a) Pantalla de bienvenida (b) Pantalla de registro (c) Pantalla de inicio de sesión

Figura 3.15

En la aplicación del taxista el procedimiento de autenticación es el mismo que la aplicación del cliente (véase pantallas 3.15a, 3.15b, 3.15c) con la excepción de que en la pantalla de registro se requiere un campo adicional. Este hace referencia a una clave que es distribuida solamente a los taxistas con la que poder verificar en un futuro que solo existan cuentas de taxista con licencia.



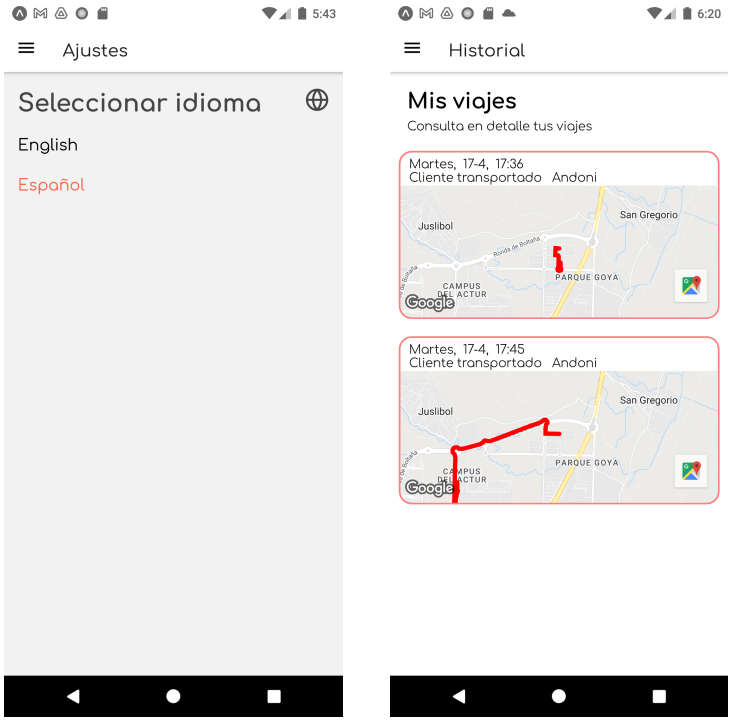
(a) Pantalla principal (b) Menú de la aplicación (c) Qr asociado

Figura 3.16

Una vez se ha iniciado sesión, la aplicación muestra la pantalla principal (véase pantalla 3.16a). En esta el taxista puede acceder al sistema para ofrecer su servicio. Además, se puede acceder al menú de la aplicación pulsando el botón que se encuentra en la esquina superior izquierda.

Al apretar el menú contextual aparecen las opciones del taxista (véase pantalla 3.16b). En él se permite acceder a recursos adicionales de la aplicación aparte del de reserva de taxis. Estos son el historial de viajes, QR identificador del taxista, ajustes de idioma y el cierre de sesión.

El taxista puede acceder a su QR asociado (véase pantalla 3.16c) mediante el menú, el cual codifica un token con el que el sistema identifica al taxista, para ser usado para cuando el taxista llegue a la posición de un cliente y este tenga que validar que ha recibido servicio.

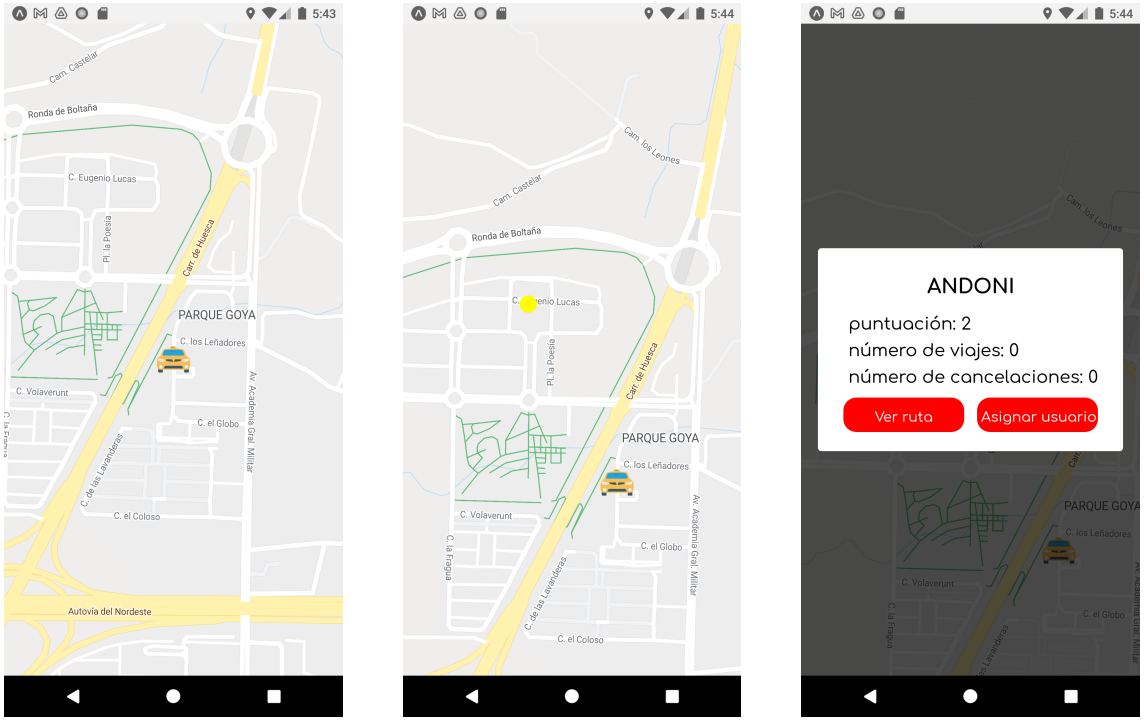


(a) Ajustes de idioma (b) Historial de viajes

Figura 3.17

En caso de querer configurar el idioma en el que se presenta la aplicación, se accede a la opción de configuración donde se abre una nueva pantalla (véase pantalla 3.17a) en la que se permite escoger el idioma de la aplicación entre Inglés y Español. Cuando se inicia la aplicación por primera vez se detecta el idioma del dispositivo móvil que ejecuta la aplicación y se utiliza ese por defecto.

Si el taxista quiere acceder al histórico de viajes realizados podrá acceder al historial (véase pantalla 3.17b) donde se puede acceder los datos de todos los viajes sus anteriores.



(a) Taxista buscando clientes (b) Aparece un cliente (c) Información del cliente

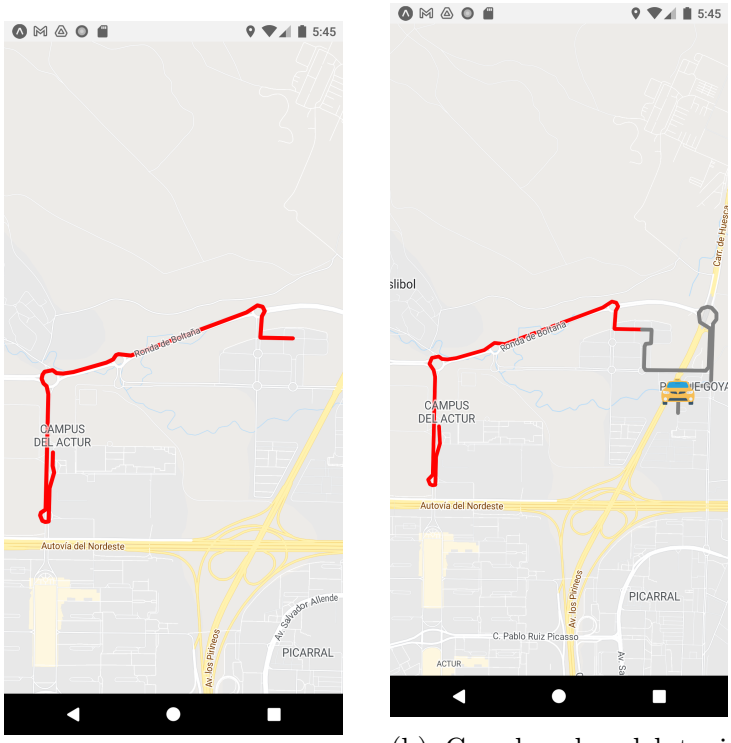
Figura 3.18

Cuando el taxista quiera ofrecer sus servicios la aplicación le redirige a una nueva pantalla (véase pantalla 3.18a) corresponde con el mapa de la ciudad de Zaragoza, en este caso sobre el mapa encontramos el símbolo de un taxi que señala la ubicación en tiempo real del taxista que está usando la aplicación.

Cuando aparece por primera vez un cliente interesado en coger un taxi, la aplicación emite un sonido y aparece en la pantalla (véase pantalla 3.18b) un punto que señala la ubicación de salida del usuario que solicita un taxi. En caso de que todos usuarios hayan sido atendidos por otros taxistas, se emite un sonido diferente, el cual avisa que ya no hay clientes que requieren servicio.

El color del punto puede ser de 5 colores diferentes. Dependiendo de qué valoración media posee dicho cliente. Cada color corresponde con una puntuación entre el uno y el cinco, siendo el uno la peor puntuación y el cinco la mejor. Si el punto es de color verde oscuro significa que cliente tiene una valoración media de cinco puntos, en el caso de ser verde claro la puntuación media es de cuatro puntos, si es amarillo de tres, naranja de dos puntos y rojo si la puntuación media del usuario es de un punto.

Cuando el taxista presiona sobre el punto aparece una pestaña (véase 3.18c) donde el taxista accede a datos del cliente como el nombre, la puntuación, el total de viajes realizados y los viajes cancelados, además se le ofrece la opción de visualizar la ruta entera del cliente y de asignar su servicio.



(a) Ruta del cliente

(b) Coordenadas del taxista en tiempo real

Figura 3.19

Si el taxista desea comprobar la ruta, se le redirige a una pantalla (véase pantalla 3.19a) donde se le muestra la ruta más corta desde la posición de origen del cliente hasta la posición final.

Si se acepta el viaje de dicho cliente aparece una nueva pantalla (véase pantalla 3.19b) con la ruta a realizar desde el punto dónde se encuentra el taxista hasta el punto de salida del cliente (en gris) y ruta hasta el punto de destino del cliente (en rojo). El taxista transmite sus coordenadas en tiempo real al cliente hasta que llegue a la posición de salida en donde el cliente deberá escanear el código QR del taxista para terminar el proceso de reserva.

A continuación se muestra el mapa de navegación de la aplicación del taxista donde se muestra la interacción entre las diferentes pantallas que forman la aplicación (véase figura 3.20).

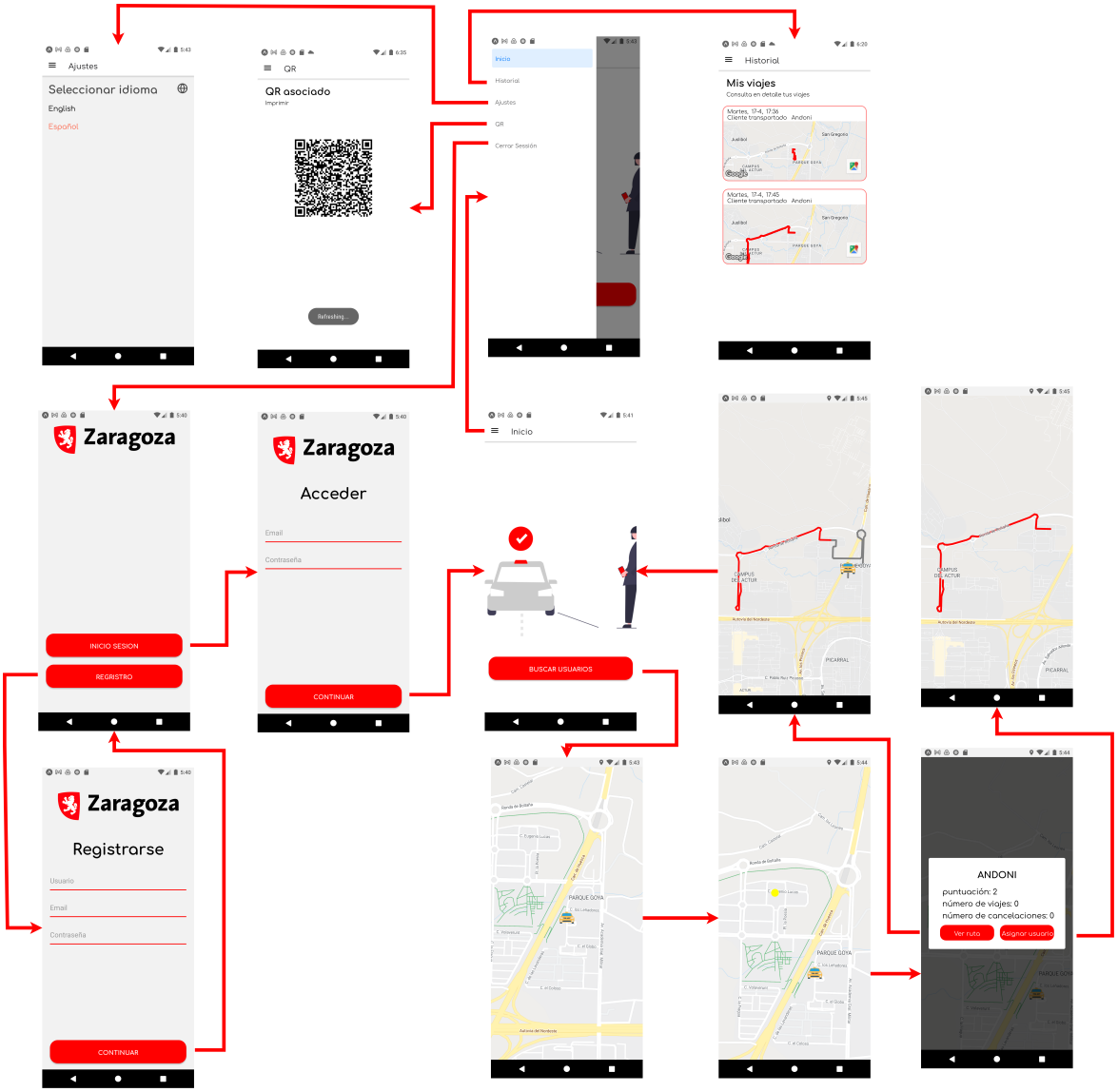


Figura 3.20: Mapa de navegación de la aplicación

Capítulo 4

Despliegue de la aplicación

En este capítulo se expone el proceso que se ha seguido para desplegar la aplicación en un entorno de producción. Se explica las primeras fases de desarrollo y cómo se ha ido adaptando el sistema para darle capacidad de escalado y para que al final se haya construido un sistema realista que funciona en la nube.

4.1. Desarrollo continuo

Desde las primeras fases del desarrollo se ha llevado a cabo una estrategia de despliegue continuo que consiste en publicar automáticamente los cambios en el entorno de producción. Para ello se ha utilizado la herramienta *GitHub Actions* como mecanismo de automatización para verificar, publicar y desplegar las actualizaciones de software. Se han creado cuatro flujos de trabajo; verificación de código, contenerización, subida de imágenes al repositorio de Docker y actualización del clúster de Kubernetes.

El flujo de revisión de código consiste en la reproducción de pruebas unitarias que aíslan cada parte del sistema. Estas son generadas con la finalidad de asegurarse que no se han producido errores en la integración en las nuevas funcionalidades del sistema. Se ejecutan una serie de pruebas desarrolladas en iteraciones anteriores del desarrollo. En el caso de que no se produzca ningún error, se podrá asegurar el correcto funcionamiento del sistema y se procederá a integrar los nuevos cambios.

El segundo flujo consiste en la creación de las nuevas imágenes Docker de cada microservicio del sistema. Se etiqueta cada contenedor como versión más reciente para que el tercer flujo lo publique en el repositorio de imágenes de DockerHub donde se podrá acceder a los recursos desde cualquier lugar.

El último flujo consiste en actualizar la versión desplegada en Kubernetes con los nuevos cambios. En primer lugar, se fuerza la finalización de los recursos que no están al día en la última versión para posteriormente volver a iniciar el sistema.

4.2. Despliegue en Kubernetes

Una de las principales razones de la implementación de una arquitectura basada en microservicios es la facilidad con la que se puede realizar un escalado horizontal. Por ello, una de las claves del proyecto es el despliegue de la aplicación sobre un clúster de Kubernetes. En este apartado se va a explicar los tipos de escalado que se han llevado a cabo y las partes más destacadas de cada sistema.

Kubernetes es una plataforma pensada para arquitecturas de microservicios que ofrece una plataforma para administrar contenedores y facilitar la automatización.

El despliegue inicialmente se ha efectuado de forma local utilizando la herramienta *Minikube* que permite desplegar un clúster de Kubernetes localmente. Este configura automáticamente una máquina virtual donde despliega un clúster de una distribución minimalista de Kubernetes. A continuación se ofrece una leyenda de los distintos recursos de Kubernetes que se ha utilizado para el despliegue (véase figura 4.1).

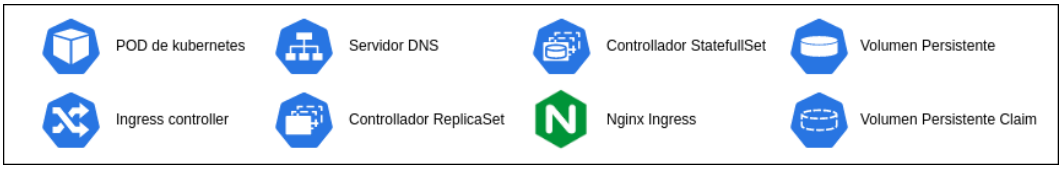


Figura 4.1: Leyenda

4.2.1. Sistema monolítico en Kubernetes

Una primera aproximación consiste en asignar cada componente de la arquitectura en un recurso de Kubernetes. De esta forma se tendrían seis nodos que son los correspondientes a los componentes de *Syncronize*, *User-Account*, *Taxi-Account*, *Data*, *Decisión Maker* y *Model*.

Cada nodo del sistema está compuesto por un *Pod*. Esta es la unidad más pequeña que se puede desplegar en Kubernetes que es donde se despliega el contenedor del microservicio. Además, un servicio de *DNS* que sirve para que los pods puedan comunicarse entre ellos a través del nombre del recurso.

A excepción de la mayoría, los nodos *Model* y *Data* no solo se componen del contenedor asociado a ese microservicio, sino que también se despliega dentro del mismo recurso la base de datos que necesitan para su funcionamiento.

Adicionalmente, se ha utilizado el recurso *Ingress* que se usa para exponer rutas de dentro del clúster al exterior. Los *Ingress* necesitan un controlador que es el que se asegura de que el tráfico entrante se administra de modo que se haya especificado en la configuración del *Ingress*. No solo es necesario que se pueda acceder desde el exterior al componente de comunicación del sistema, sino que también se requiere por ejemplo

acceder a los datos del sistema o a las cuentas del usuario. Por ello en la configuración se permite acceder a estos recursos desde el exterior. De esta forma se restringen las rutas que solo se pueden acceder desde el interior del clúster evitando fallos de seguridad.

A continuación se muestra un diagrama que esquematiza el despliegue monolítico en el clúster de Kubernetes (véase diagrama 4.2). Al igual que en el resto de diagramas, se ha seguido un patrón de colores para identificar los nodos asociados a los componentes del diagrama de componentes de la aplicación.

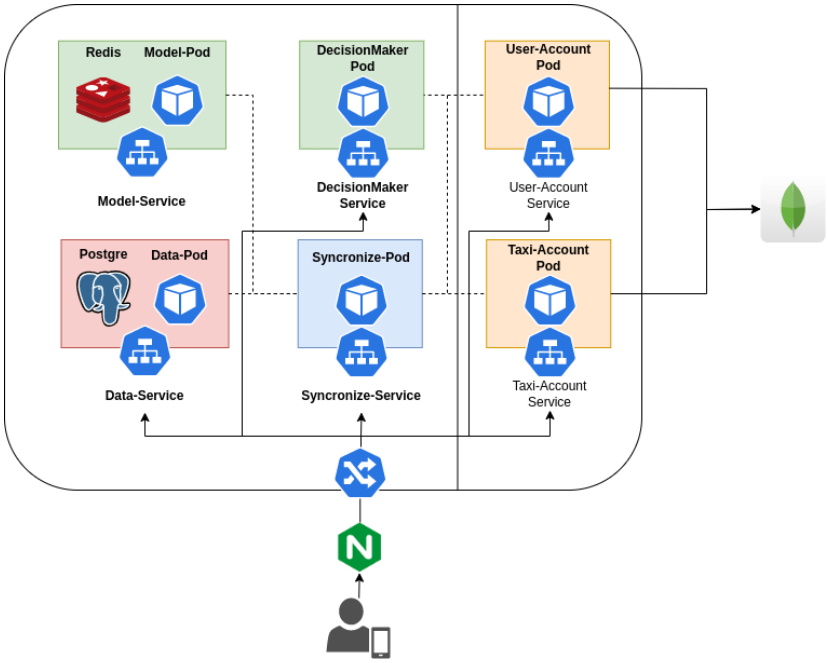


Figura 4.2: Sistema monolítico

Sin embargo este tipo de despliegue cuenta con varios problemas sustanciales que hace se que planteen despliegues más complejos:

- Tolerancia a fallos por caída. El recurso *Pod*, no es tolerante a fallos por caída por lo que la caída de un recurso conlleva una interrupción en el servicio del sistema.

- Persistencia de los datos. Las bases de datos desplegadas en Kubernetes no se mantienen en el tiempo. Por lo que en cada actualización del sistema se vacían y se perdieron los datos que había en el sistema.

- Ventajas de Microservicios. El motivo inicial de implementar una arquitectura basada en microservicios es poder escalar la aplicación para mejorar el rendimiento y adaptarse a la carga de trabajo que requiera la aplicación. Con este despliegue el propósito inicial no se cumple y se desperdician las ventajas de usar microservicios.

– Recursos de Kubernetes. El propósito general de Kubernetes es que ofrece facilidades para realizar un escalado horizontal en una arquitectura basada en microservicios. Por ello carece de sentido utilizarlo si no se van a aplicar las características de escalado que ofrece Kubernetes.

Como consecuencia se ve necesario mejorar el tipo de despliegue solucionando los problemas que aparecen con el despliegue monolítico.

4.2.2. Sistema con varios nodos en Kubernetes

Los problemas que surgían con la aproximación anterior del despliegue buscan ser resueltos utilizando los recursos que ofrece Kubernetes para realizar un despliegue con escalado horizontal.

La aproximación final del sistema desplegado y distribuido en varios nodos se mostraría a continuación (véase figura 4.3).

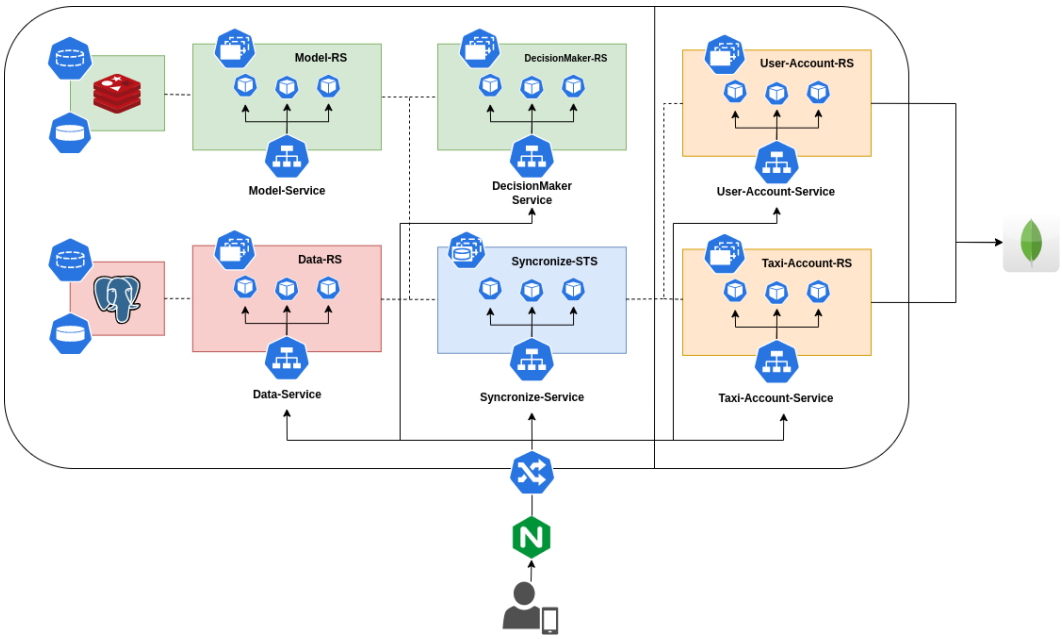


Figura 4.3: Sistema escalado

Como se puede observar, cada nodo de Kubernetes ya no está compuesto solo por un *Pod*, sino que ahora se le ha añadido un controlador[21] para mantener un conjunto estable de réplicas de *Pods* ejecutándose en todo momento. Además, se han separado las bases de datos de los *Pods*, así cuando hay actualización en el software se evita perder los datos persistentes del sistema.

Se han realizado tres tipos distintos de escalados según las necesidades que requiera cada servicio. Los microservicios *Stateless* se han replicado utilizando un controlador *ReplicaSet*[22] y se ha configurado sus respectivos servicios como balanceadores de carga. El único microservicio que mantiene el estado es el *Syncronize* donde se ha empleado un controlador *StatefullSet*[23] para mantener el estado aun produciéndose fallos de caída. Sin embargo, al tratarse de un componente en el cual sus réplicas tienen que estar comunicadas, su escalado requiere más complejidad. Por ello se va a analizar en profundidad el escalado del componente *Syncronize* en la siguiente sección (véase apartado 4.2.2). El último tipo de escalado es el que se ha aplicado en las bases de datos dentro del clúster en el que también se ha definido un controlador *StatefullSet*. En el que adicionalmente se ha configurado un volumen persistente[24] con el que se gestiona el almacenamiento.

Syncrozice

Como se ha comentado en el apartado anterior, el componente *Syncronize* requiere un escalado especial. El problema que surge porque las replicas son independientes, cuando dos usuarios se conectan mediante *WebSockets* al servidor. El balanceador de carga coloca a ambos usuarios en distintas replicas, por lo que al ser réplicas aisladas ambos usuarios no se pueden comunicar entre ellos. A continuación, se muestra un ejemplo de este problema, los clientes A, B y D están suscritos en la misma *Room*. Sin embargo, si B emite un mensaje a toda la *Room* D no lo recibe.

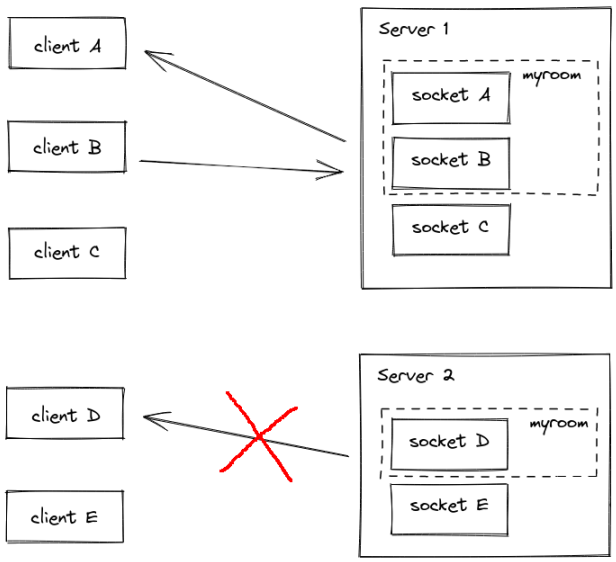


Figura 4.4: Ejemplo problema replicación

Para solucionarlo se utiliza un componente que reemplaza al adaptador de memoria predeterminado por otra implementación. De este modo, los eventos del lado del servidor se transmiten correctamente a todos los usuarios (véase diagrama 4.5).

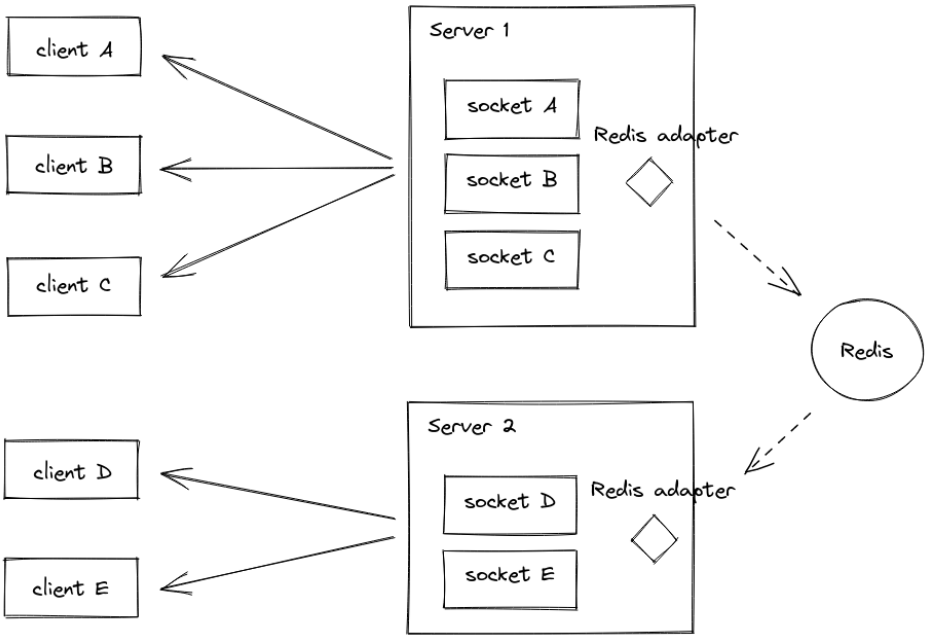


Figura 4.5: Diagrama escalado *Syncronize*

Se ha decidido utilizar la implementación del adaptador *Redis*[9] que se basa en el mecanismo de publicación suscripción.

4.3. Despliegue en *Digital Ocean*

Una vez se ha tenido el sistema completamente funcional en el clúster local, se ha llevado a modo de producción, para ello se ha confiado en el proveedor de servicios *Cloud Digital Ocean*. Este te ofrece una prueba gratuita de dos meses, los cuales han sido suficiente para terminar de testear y poner a punto el sistema. Durante este periodo se ha tenido el sistema funcionando continuamente y se ha visto expuesto a pruebas con la finalidad de extraer conclusiones sobre las capacidades del sistema. Todos los experimentos han sido realizados en la misma máquina para que los resultados sean comparables. La máquina es desplegada en la nube y se sitúa en Francia y cuanta con 6 CPU, una memoria RAM de 6 GB y una memoria en disco de 180 GB.

Los test del sistema (véase sección 5) han sido realizados todos sobre esta maquina. De este modo, los tiempos de respuesta que se obtienen de los test describen perfectamente un entorno realista. Por lo que se puede afirmar que los resultados son fiables respecto al límite de conexiones simultáneas y peticiones al servidor y serían similares a los que se lograrían en un sistema en producción.

El despliegue ha sido sencillo, una vez creada la máquina que alberga el clúster simplemente se ha tenido que autenticar desde la terminal con la herramienta *Docktl*. Esta configura *kubectl* para que pueda ejecutar instrucciones en el clúster remoto.

Capítulo 5

Experimentación y testing

En este capítulo se describen los experimentos realizados sobre el sistema. Uno de los objetivos principales que persigue la aplicación es dar soporte a la máxima cantidad de usuarios posibles simultáneamente y además ofreciendo una calidad de servicio satisfactoria para el usuario. En estos test se pretende también probar la fiabilidad del sistema y su tolerancia a fallos.

El primer subconjunto de experimentos se compone principalmente de test de carga que buscan colapsar el sistema con la finalidad de obtener de manera experimental el límite de usuarios que soporta la aplicación en un entorno real. Para ello se van a alternar los tipos de despliegue del sistema para sacar conclusiones sobre cuál es el más adecuado.

El segundo conjunto de pruebas persiguen simular el comportamiento real del sistema en donde se busca ajustar el número de peticiones de usuario y el número de taxistas que ofrecen servicio para llevarlo a un entorno práctico en vez de llevar el sistema al colapso.

5.1. Experimentos de volumen

En esta sección se van a realizar el primer conjunto de pruebas que están relacionadas con el volumen de datos que soporta el sistema.

5.1.1. Pruebas con un sistema monolítico

El objetivo de este primer test es llevar al sistema al colapso. La estrategia que se va a seguir es ir aumentando progresivamente el número de usuarios simultáneos en el sistema, dándoles servicio, simulando tres latencias distintas en el sistema, sin latencia, latencia de un minuto y latencia de dos minutos. El sistema recoge el tiempo total desde que un usuario ha solicitado un taxi hasta que se le ha asignado un taxista.

Como esta prueba experimental no busca simular un comportamiento realista del sistema, este cuenta con infinitos taxistas que atienden las peticiones. Este test cuenta con tres actores principales.

El servidor, que es el que va a soportar la carga y el encargado de reexpedir los mensajes de los usuarios.

Un conjunto de servicios desplegados en varias máquinas en la nube, que cada uno lanza una serie de usuarios concurrentemente y distribuidos uniformemente a lo largo de un minuto donde cada uno inicia el proceso de reserva de taxi.

Un servicio también desplegado en la nube, que atiende todas las peticiones de los usuarios del sistema, simulando un tiempo de espera que depende de la latencia con la que se ha configurado el test. La latencia se simula con una distribución *Poisson* con λ el tiempo de llegada. Se ha decidido usar una distribución de estas características, dado que su uso es muy frecuente en teoría de colas y tiempos de espera.

Resultados

Se ha realizado el experimento para un rango de usuario entre cincuenta y siete mil con un progresivo aumento de quinientos usuarios entre cada intervalo. Los resultados obtenidos se muestran en la siguiente gráfica (véase gráfica 5.1).

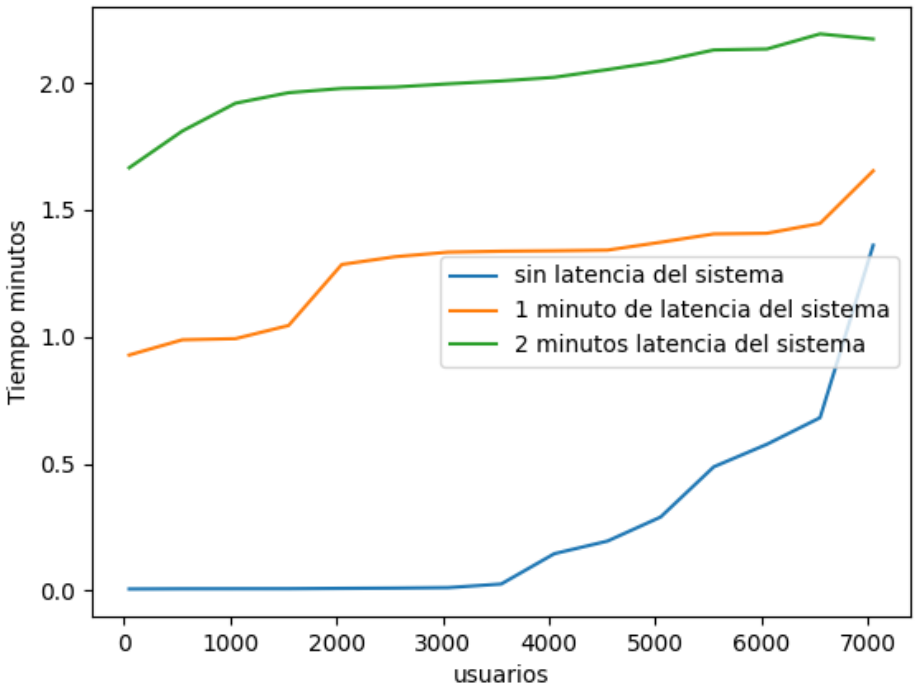


Figura 5.1: Tiempo de respuesta del sistema emulando latencia

Conclusiones

Lo primero que se puede observar en la gráfica es que cuando el tiempo de latencia del sistema es nulo, a medida que aumenta el número de usuarios, el tiempo de respuesta del sistema también lo hace. El sistema empieza a colapsar a partir de aproximadamente tres mil quinientos usuarios, que es cuando la latencia del flujo de reserva supera treinta segundos. A partir de seis mil quinientos usuarios el sistema empieza a perder paquetes, lo que vuelve al sistema inestable.

Cuando se simula una latencia de un minuto en el tiempo de respuesta del taxista, se observa que ante un aumento del número de usuarios la alteración del tiempo de respuesta del sistema es significativamente inferior al del experimento realizado sin latencia simulada. Además, la pérdida de paquetes del sistema se reduce.

Los problemas de alteración de tiempo de respuesta y pérdida de paquetes se atenúan cuando se simula una latencia de dos minutos. Aunque existan pérdidas de paquetes, estos son causados por paquetes de confirmación de recepción de mensaje que son necesarios al tener el protocolo de tiempo real sobre una conexión TCP. Ya que para mantener la conexión se mandan paquetes que mantienen la conexión abierta. Si tres paquetes lanzados no son respondidos, la conexión del cliente se cierra, lo que provoca que el cliente no sea atendido.

5.1.2. Pruebas con sistemas escalados

El objetivo de este segundo test es centrarse en ofrecer al usuario la mínima latencia posible. El anterior test ha sido ejecutado sobre un sistema monolítico en donde cada módulo del sistema se corresponde a un servicio, por lo que no existe explicación alguna ni balanceadores de carga para múltiples nodos. Lo que sugiere que escalando el sistema los resultados anteriores se pueden mejorar sustancialmente.

La estrategia que se ha seguido también es ir aumentando progresivamente el número de usuarios simultáneos en el sistema ofreciendo una respuesta inmediata. El sistema recoge el tiempo total desde que un usuario ha solicitado un taxi hasta que se le ha asignado un taxista. Cuanto mayor sea el número de usuarios en el sistema mayor será el número de peticiones que tendrá que gestionar el sistema. Por ello llega un punto donde el número de peticiones es tan alto que aunque no exista latencia simulada en el sistema, el tiempo de respuesta aumenta.

Resultados

Las pruebas se han realizado con escalado de dos nodos y tres nodos para un rango de usuario entre cincuenta y catorce mil, con un progresivo aumento de quinientos usuarios entre cada intervalo. Los resultados obtenidos se muestran en la siguiente gráfica (véase gráfica 5.2).

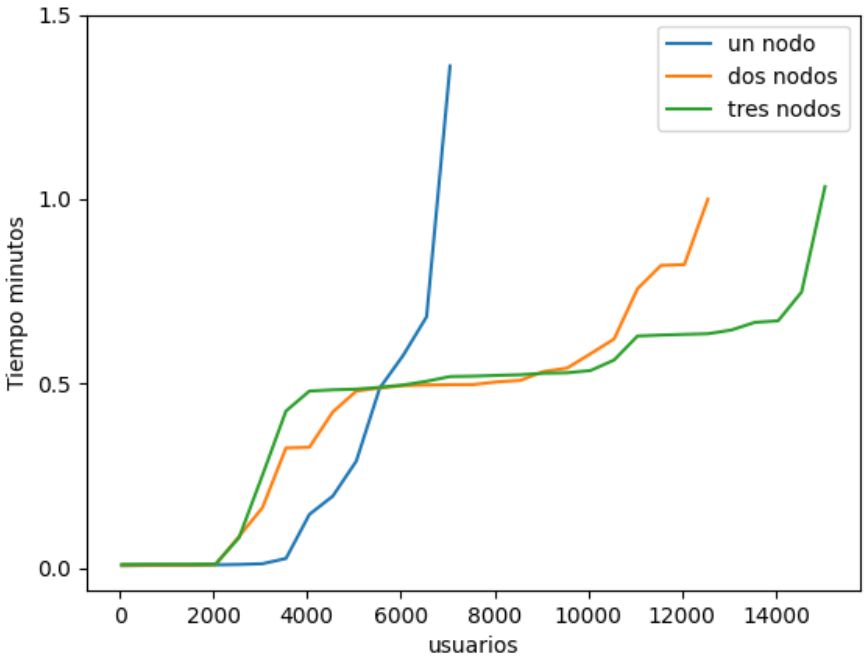
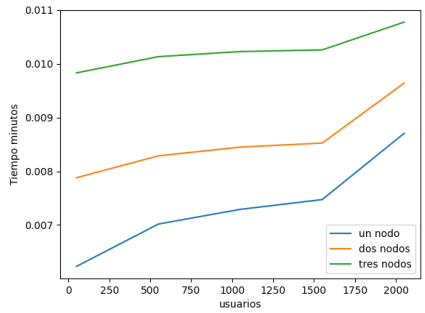


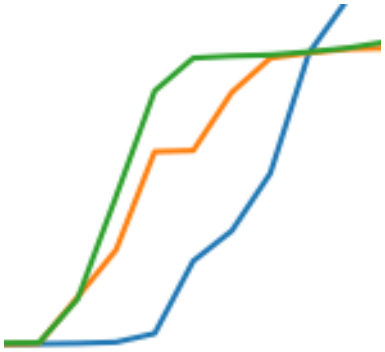
Figura 5.2: Tiempo de respuesta sistema con distintos nodos

Conclusiones

Desde un punto de vista general se observa una clara mejoría en el número de usuarios que puede soportar el sistema con tan solo añadirle un segundo nodo. Con tres nodos el número de usuarios que tolera el sistema sigue aumentando, aunque la mejora no es tan significativa como cuando se le añade un segundo nodo. No obstante, si se examina la gráfica con profundidad, existen dos temas a tener en cuenta. Para verlo con más claridad se ha aumentado la escala de la gráfica extrayendo dos sub gráficas (véase gráficas 5.3a y 5.3b).



(a) gráfica 5.2 con sólo los primeros 2000 usuarios



(b) gráfica 5.2 entre los 2000 y los 4000 usuarios

En el gráfico 5.3a se observa el tiempo medio de respuesta de los sistemas con un solo nodo, dos nodos y tres nodos en los primeros dos mil clientes. Aunque la diferencia sea insignificante, se muestra con claridad un patrón, al aumentar el número de nodos que componen el sistema mayor es el tiempo medio de respuesta. Esto es debido a cómo se ha escalado la comunicación bidireccional, mientras que cuando el sistema se integra con un solo nodo, todas las conexiones se realizan sobre ese nodo. Cuando se manda un evento a un grupo de taxistas, todos están conectados al mismo nodo, por lo que la emisión del mensaje no pasa por ningún intermediario, evitando un sobre coste en la latencia del sistema.

Sin embargo, cuando aparecen más nodos. Las conexiones de los clientes y los taxistas son distribuidos entre los distintos nodos a través de un balanceador de carga, por lo que cuando se emite un evento a un grupo de taxistas no necesariamente todos están conectados al mismo nodo. Por lo que la emisión del mensaje si tiene que pasar por un intermediario (en este caso un adaptador Redis) que se asegure que todos los taxistas reciban ese evento. Lo que produce un pequeño sobre coste en el tiempo de respuesta de las peticiones.

Este hecho cuando el número de peticiones es soportable por el sistema realmente es insignificante. No obstante, cuando el número de conexiones se dispara, sí que este coste extra toma relevancia. En el gráfico 5.3b registra este hecho, el tiempo de respuesta de los sistemas escalados con múltiples nodos aumenta antes que en un sistema monolítico. Esto es debido a que todas las emisiones de mensajes pasan por un adaptador.

5.2. Experimentos realistas

En esta sección se va a realizar el segundo conjunto de pruebas, que se centran en simular un comportamiento realista del sistema.

5.2.1. Simulación realista de un sistema

El objetivo de este test es centrarse en simular el comportamiento de un sistema real en vez de sobrecargar el sistema para saber el número de usuarios que soporta simultáneamente. Se va a limitar el número de usuarios a los que utilizarían el sistema en un entorno normal. En vez de simular taxis infinitos para poder atender a todos los usuarios, se va a limitar el número de taxistas, por lo que no va a haber taxistas disponibles para ofrecer servicio a los clientes. Se va a configurar un tiempo de respuesta del taxista de un minuto y la duración del viaje se va a simular mediante una distribución *Poisson* de λ diez minutos y σ cinco minutos, lo que significa que la duración de cada viaje es de entre cinco y quince minutos.

Resultados

La gráfica que se muestra a continuación recoge el tiempo máximo, medio y mínimo que a un taxista le costaría atender las peticiones en un entorno donde se limita el número de taxistas a 200 (véase gráfica 5.4).

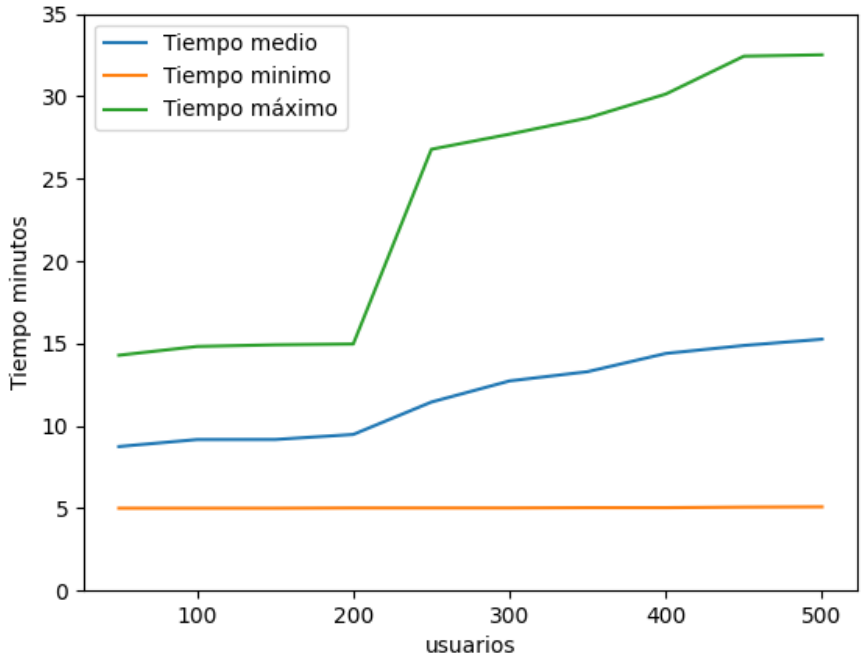


Figura 5.4: Simulación realista del sistema suponiendo 200 taxis

Conclusiones

Cuando el sistema cuenta con menos de doscientos usuarios el número de taxis es suficiente para cubrirlos a todos por lo que los tiempos máximos de espera se sitúan en el caso peor que es cuando la duración de los viajes ha sido de quince minutos. A medida que se aumenta el número de clientes los taxistas no son suficientes para atender a todos los clientes por lo que el tiempo medio de asignación de servicio aumenta. El mayor tiempo que un cliente espera para llegar a su destino es de aproximadamente treinta y cinco minutos. Se deduce que el taxista que atiende a dicho cliente ya ha realizado dos viajes previos y por ello el tiempo total es tan elevado. Si solo se tiene en cuenta el tiempo medio, realmente la diferencia entre el caso peor y el caso mejor es de cinco minutos, que teniendo en cuenta las duraciones de los viajes es una cifra asumible.

Capítulo 6

Gestión del proyecto

En este capítulo se profundiza en la gestión del proyecto y cómo se ha enfocado el desarrollo para una correcta puesta en marcha del trabajo, se presenta la planificación de tareas, el proyecto y la metodología utilizada para ello, y se realiza un análisis de las dimensiones finales de la aplicación y del esfuerzo total dedicado para la finalización el trabajo.

6.1. Planificación del proyecto

Para el desarrollo del proyecto se ha seguido un modelo de desarrollo incremental (véase figura 6.1). Se caracteriza por desgranar la aplicación final en pequeños proyectos en los que se divide el ciclo de desarrollo en distintas etapas (análisis, desarrollo y testing). Lo que se busca es que en cada iteración evolucione la aplicación final dependiendo de las funciones ya integradas en iteraciones anteriores, agregando más requisitos al sistema y logrando una mejoría en el software.

La razón de la elección de esta metodología se centra en conseguir un prototipo inicial temprano. Además, se adapta perfectamente a la arquitectura inicial planteada que se basa en microservicios cada uno de estos va a representar una iteración en el desarrollo del proyecto.

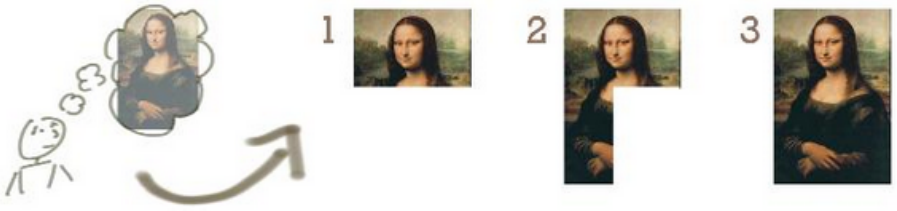


Figura 6.1: Modelo de desarrollo incremental

Se puede observar a continuación el diagrama de Gantt que ilustra la planificación final del proyecto (véase figura 6.2).

NÚMERO EDT	TÍTULO DE LA TAREA	FECHA INICIO	FECHA FIN	OCTUBRE-NOVIEMBRE				DICIEMBRE-ENERO				FEBRERO				MARZO				ABRIL				MAYO			
				1-2	3-4	5-6	7-8	1-2	3-4	5-6	7-8	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	**Iteración 1: Desarrollo del sistema de reserva de Taxi**																										
1.1	Caputra de requisitos	3/10/2021	14/11/2021																								
1.2	Análisis del problema	12/11/2021	10/12/2021																								
1.3	Diseño de la solución	21/11/2021	12/01/2022																								
1.4	Implementación de la solución	18/12/2021	18/03/2022																								
1.4.1	Componentes Account	3/01/2022	9/02/2022																								
1.4.2	Componente Sycnronize	1/02/2022	19/03/2022																								
1.4.3	Componente Matrix	16/02/2022	27/02/2022																								
1.4.4	Componente Parser	1/03/2022	6/03/2022																								
1.4.5	Componente Data	8/03/2022	13/03/2022																								
1.5	Testing (pruebas de desarrollo)	8/03/2022	28/03/2022																								
2	**Iteración 2: Interfaz grafica e integración del sistema**																										
2.1	Caputra de requisitos	15/03/2022	17/03/2022																								
2.2	Análisis del problema	21/03/2022	30/03/2022																								
2.3	Diseño de la solución	2/04/2022	5/04/2022																								
2.4	Implementación de la solución	8/04/2022	30/04/2022																								
2.4.1	Aplicación cliente	8/04/2022	17/04/2022																								
2.4.2	Aplicación taxista	18/04/2022	30/04/2022																								
2.5	Testing (pruebas de desarrollo)	24/04/2022	30/04/2022																								
3	**Iteración 3: Despliegue en Kubernetesy mejoras de la la aplicación**																										
3.1	Caputra de requisitos	24/04/2022	30/04/2022																								
3.2	Análisis del problema	1/05/2022	1/05/2022																								
3.3	Diseño de la solución	3/05/2022	5/05/2022																								
3.4	Implementación de la solución	5/05/2022	12/05/2022																								
3.4.1	Mejoras de rendimiento	3/05/2022	5/05/2022																								
3.4.2	Mejoras de usabilidad	7/05/2022	9/05/2022																								
3.5	Testing (pruebas de desarrollo)	9/05/2022	14/05/2022																								
3.6	Despliegue en Digital Ocean	15/05/2022	24/05/2022																								
3.6.1	Pruebas de volumen	16/05/2022	30/05/2022																								
4	**Documentación**																										
4.1	Redacción de la memoria del proyecto	8/05/2022	30/05/2022																								
5	**Reuniones**																										
5.1	Reuniones presenciales	3/10/2021	7/06/2022																								

Figura 6.2: Diagrama de Gantt

6.2. Dedicación y dimensión del proyecto

El número de horas totales dedicadas al proyecto alcanza la suma de 312 horas distribuidas de la siguiente manera (véase figura 6.2).

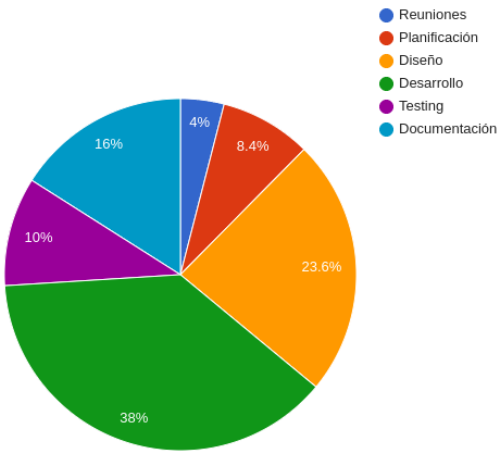


Figura 6.2: División horas de trabajo

El total de líneas de código ascienda a 37.883 y se han distribuido tal y como se muestra continuación (véase figura 6.3). Para contarlas se ha utilizado una herramienta de Git en la que no se tiene en cuenta ni los ficheros de configuración, ni librerías adicionales, ni conjuntos de datos.

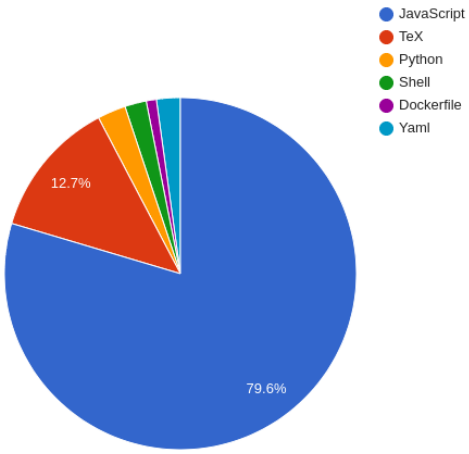


Figura 6.3: División de líneas de código

Capítulo 7

Conclusiones

En este capítulo se presentan las conclusiones extraídas y los conocimientos adquiridos durante y tras la realización del proyecto. Finalmente, se comentan líneas de trabajo futuras para mejorar y expandir la aplicación.

7.1. Conclusiones

Se han cumplido exitosamente los cinco objetivos propuestos en el capítulo 1 del documento.

Se ha desarrollado un sistema que comunica al cliente con los conductores de taxis sin ningún intermediario. Para ello se ha utilizado un algoritmo de geolocalización que conecta a los taxistas con los usuarios más cercanos. Se ha realizado una aplicación para el cliente que funciona de interfaz del sistema en la que se permite encontrar un taxi y como acceder al historial de viajes. También se ha desarrollado una aplicación homóloga para el taxista en la que le permite ofrecer su servicio de taxi. Además, se ofrece un mecanismo en el que el cliente puede dar una valoración al trayecto efectuado por el taxista y viceversa. Con esto, la aplicación proporciona información extra para los taxistas a la hora de tomar decisiones sobre qué cliente escoger. Por último, el despliegue del sistema en la nube ha sido satisfactorio, se ha usado el proveedor *Digital Ocean* y se ha diseñado un sistema escalable y que es capaz de adaptarse a los distintos escenarios que puedan suceder.

7.2. Conocimientos adquiridos

A continuación se enumeran los conocimientos técnicos y personales adquiridos a raíz de la realización del proyecto

7.2.1. Conocimientos Técnicos

En cuanto a conocimientos técnicos adquiridos, se resalta que ya se había utilizado antes el *Stack* de tecnología utilizado para la aplicación. Por lo que se tenía experiencia previa en desarrollo móvil con *React Native* y se habían desarrollado aplicaciones backend previamente con *Express* y *SocketIO*.

Esto ha facilitado en gran medida el desarrollo de la aplicación, ya que se han evitado errores iniciales en los que se desperdicia mucho tiempo. Por el contrario, sí que se han desarrollado nuevos conocimientos tras aprender tecnologías que no se usan por lo general en proyectos personales. Además de haber aprendido nuevos estándares web que se emplean hoy en día y cada vez más. A continuación un resumen de todo lo que se ha aprendido.

1. Diseño, gestión y desarrollo de arquitecturas basadas en microservicios.

2. Documentación de API de arquitecturas impulsadas por eventos utilizando la herramienta *AsyncAPI*.

3. Realización de experimentos y pruebas de carga sobre servicios construidos con *SocketIO* utilizando el kit de herramientas *ArtilleryIO*.

4. Estructuración y redacción de documentación para proyectos de tamaño mediano y grande.

5. Despliegues a nivel de producción en proveedores *Cloud* (*Digital Ocean*).

6. Utilización de la API de geolocalización de *Google Maps* e integración en un dispositivo móvil.

7.2.2. Conocimientos Personales

En cuanto a los conocimientos personales adquiridos, son referidos a la experiencia ganada tras desarrollar por primera vez un proyecto al completo en solitario, desde el diseño y análisis, pasando por la implementación funcional del prototipo, hasta despliegue en la nube.

Se ha recorrido todo el proceso de desarrollo software y se han pasado por todas las etapas cumpliendo exitosamente los requisitos planteados al inicio del proyecto. Por lo que es de gran satisfacción haber sido capaz de completar el desarrollo del sistema al completo.

7.3. Líneas de trabajo futuras

Aunque se han cumplido exitosamente todos los objetivos planteados al principio del proyecto, existen muchas vías de mejora posible para la aplicación desarrollada.

En primer lugar, se debería refinar el diseño de la interfaz, teniendo en cuenta desde un principio variables como la usabilidad y la accesibilidad del usuario a la aplicación. Se deben tener en cuenta sesgos de edad, así como el usuario final a la que va destinada la aplicación. En dicho caso se propone realizar evaluaciones de la aplicación a usuarios, aplicando por ejemplo test de usabilidad y cuestionarios de calidad a grupos dirigidos.

Por otra parte, se podría mejorar el algoritmo de toma de decisiones según la geolocalización del usuario. Actualmente, consiste en conectar a los clientes y taxistas que se encuentren en la misma región de la ciudad. Se sugiere por ejemplo, utilizar algoritmos de *Clusterización* para formar grupos de usuarios que requieran servicio y de esa forma conectar los taxistas al conjunto más cercano.

Por último, la aplicación actualmente solo permite la búsqueda de taxi en líneas futuras, se podría implementar la función de reserva de taxi para una hora determinada.

Bibliografía

[1] *Documentación React Native*. https://reactnative.dev/docs/getting-started. accessed June 8, 2022.

[2] *Documentación ExpoCli*. https://docs.expo.dev/workflow/expo-cli/. accessed June 8, 2022.

[3] *Documentación Express*. https://expressjs.com/es/guide/routing.html. accessed June 8, 2022.

[4] *Documentación SocketIO*. https://socket.io/docs/v4/. accessed June 8, 2022.

[5] *Google Maps Geocoding API*. https://developers.google.com/maps/documentation/geocoding?hl=es-419. accessed June 8, 2022.

[6] *Documentación Artillery.io*. https://www.artillery.io/docs. accessed June 8, 2022.

[7] *Documentación MongoDB Atlas*. https://www.mongodb.com/docs/atlas/. accessed June 8, 2022.

[8] *Redis caché*. https://redis.io/docs/manual/cli/. accessed June 8, 2022.

[9] *Redis Pub/Sub*. https://redis.io/docs/manual/pubsub/. accessed June 8, 2022.

[10] *Postgre ORM*. https://sequelize.org/docs/v6/getting-started/. accessed June 8, 2022.

[11] *Documentación Docker*. https://docs.docker.com/get-started/. accessed June 8, 2022.

[12] *Documentación DockerHub*. https://hub.docker.com/. accessed June 8, 2022.

[13] *Documentación Kubernetes*. https://kubernetes.io/es/docs/home/. accessed June 8, 2022.

[14] *Documentación Minikube*. https://minikube.sigs.k8s.io/docs/start//. accessed June 14, 2022.

[15] *Documentación Digital Ocean*. https://docs.digitalocean.com/products/getting-started/. accessed June 8, 2022.

[16] *Documentación github actions*. https://docs.github.com/es/actions. accessed June 8, 2022.

[17] *Documentación AsyncAPI*. https://www.asyncapi.com/docs/getting-started. accessed June 8, 2022.

[18] *Patrón publicador suscriptor*. https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern. accessed June 8, 2022.

[19] *Broker de Mensajes*. https://en.wikipedia.org/wiki/Message_broker. accessed June 8, 2022.

[20] *SocketIO Rooms*. https://socket.io/docs/v4/rooms/. accessed June 8, 2022.

[21] *Kubernetes Controllers*. https://kubernetes.io/es/docs/concepts/workloads/controllers/. accessed June 8, 2022.

[22] *Kubernetes ReplicaSet*. https://kubernetes.io/es/docs/concepts/workloads/controllers/replicaset/. accessed June 8, 2022.

[23] *Kubernetes StatefulSet*. https://kubernetes.io/es/docs/concepts/workloads/controllers/statefulset/. accessed June 8, 2022.

[24] *Kubernetes Volumen Persistente*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. accessed June 8, 2022.

Anexos

Anexos A

Documentación API

En este anexo se muestra la documentación de la API del servidor que se ha construido con la herramienta *AsyncAPI*, la cual permite documentar las arquitecturas desarrolladas con un paradigma de comunicación asíncrona. La especificación de la API se divide en dos canales de comunicación, en uso se admiten los eventos destinados al taxista y en el otro los destinados al cliente. En la documentación se expone los tipos de mensaje que existen así como ejemplos del contenido de los mensajes.

Taxi booking service 1.0.0 documentation

The main service which control the taxi's booking lifecycle

Table of Contents

Servers

`production` Server

- URL: `syncronize.com`
- Protocol: `wss`

Operations

PUB `/cabbie` Operation

`http` Channel specific information

Name	Type	Description	Value	Constraints	Notes
query	object	-	-	-	**additional properties are allowed**
query.token	string	-	-	-	**required**

Accepts **one of** the following messages:

Message `change`

when cabbie changes quadrant, notify the server in order to update the users who are subscribed to that quadrant.

Payload

Name	Type	Description	Value	Constraints	Notes
(root)	object	-	-	-	**additional properties are allowed**
room	integer	-	-	-	-

> Examples of payload

Entering in room 1

```
{
  "room": 1
}
```

Message `assign`

if the user ($id) is still waiting for a taxi, assign the taxi to that user and notify the user that a taxi is on the way.

Payload

Name	Type	Description	Value	Constraints	Notes

Name	Type	Description	Value	Constraints	Notes
(root)	object	-	-	-	**additional properties are allowed**
id	string	-	-	-	-
coords	object	-	-	-	**additional properties are allowed**
coords.latitude	number	-	-	-	-
coords.longitude	number	-	-	-	-

> Examples of payload

Assigning a user

```
{
  "id": "HNR9B5MTTLgY_NJzAAPV",
  "coords": {
    "latitude": 41.58910558784315,
    "longitude": -1.090098864658791
  }
}
```

SUB `/cabbie` Operation

`http` Channel specific information

Name	Type	Description	Value	Constraints	Notes
query	object	-	-	-	**additional properties are allowed**
query.token	string	-	-	-	**required**

Accepts **one of** the following messages:

Message `getusers`

update the clients who are subscribed in a quadrant.

Payload

Name	Type	Description	Value	Constraints	Notes
(root)	array	-	-	-	-
userID	number	-	-	-	-
socketID	number	-	-	-	-
coords	object	-	-	-	**additional properties are allowed**
coords.latitude	number	-	-	-	-

Name	Type	Description	Value	Constraints	Notes
coords.longitude	number	-	-	-	-
user	object	-	-	-	additional properties are allowed
user.name	string	-	-	-	-
user.score	number	-	-	-	-
user.totalRide	integer	-	-	-	-
user.cancelRide	integer	-	-	-	-

> Examples of payload

Getting two users data who are subscribed on the system

```
[
  {
    "socketID": "HNR9B5MTTLgY_NJzAAPV",
    "userID": "FrMMP0MeBkRDCoBUAAAH",
    "coords": {
      "latitude": 41.58910558784315,
      "longitude": -1.090098864658791
    },
    "user": {
      "name": "Javier Fernandez",
      "score": 2.67,
      "totalRide": 253,
      "cancelRide": 12
    }
  },
  {
    "socketID": "HNR9B0Dv1fLgY_NJzABBAc",
    "userID": "FrMMP0MeBkRDCoHHHAAcc",
    "coords": {
      "latitude": 43.5891044413665,
      "longitude": -3.00988643441111
    },
    "user": {
      "name": "Juan Diez",
      "score": 3.67,
      "totalRide": 17,
      "cancelRide": 0
    }
  }
]
```

Message `response`

send back to the cabbie if user assign has been completed

Payload

Name	Type	Description	Value	Constraints	Notes
(root)	boolean	-	-	-	-

> Examples of payload

User correctly assigned

```
{
  "response": true
}
```

User not assigned

```
{
  "response": false
}
```

Message `response`

notify the user that a taxi is on the way.

Payload

Name	Type	Description	Value	Constraints	Notes
(root)	object	-	-	-	**additional properties are allowed**
SocketID	string	-	-	-	-
CabbieID	string	-	-	-	-
coords	object	-	-	-	**additional properties are allowed**
coords.latitude	number	-	-	-	-
coords.longitude	number	-	-	-	-

> Examples of payload

Cabbie is on the way

```
{
  "SocketID": "HNR9B5MTTLgY_NJzAAPV",
  "CabbieID": "FrMMP0MeBkRDCoBUAAAH",
  "coords": {
    "latitude": 41.58910558784315,
    "longitude": -1.090098864658791
  }
}
```

PUB `/user` Operation

`http` Channel specific information

Name	Type	Description	Value	Constraints	Notes
query	object	-	-	-	**additional properties are allowed**
query.token	string	-	-	-	**required**

Message `end`

When a taxi arrives at user's position, this informs the server.

Payload

Name	Type	Description	Value	Constraints	Notes

Name	Type	Description	Value	Constraints	Notes
(root)	object	-	-	-	**additional properties are allowed**
cabbieSocketID	string	-	-	-	-
cabbieID	string	-	-	-	-

> Examples of payload

User end

```
{
  "SocketID": "HNR9B5MTTLgY_NJzAAPV",
  "CabbieID": "FrMMP0MeBkRDCoBUAAAH"
}
```

SUB `/user` Operation

`http` Channel specific information

Name	Type	Description	Value	Constraints	Notes
query	object	-	-	-	**additional properties are allowed**
query.token	string	-	-	-	**required**

Accepts **one of** the following messages:

Message `getusers`

update the clients who are subscribed in a quadrant.

Payload

Name	Type	Description	Value	Constraints	Notes
(root)	array	-	-	-	-
userID	number	-	-	-	-
socketID	number	-	-	-	-
coords	object	-	-	-	**additional properties are allowed**
coords.latitude	number	-	-	-	-
coords.longitude	number	-	-	-	-
user	object	-	-	-	**additional properties are allowed**
user.name	string	-	-	-	-
user.score	number	-	-	-	-

Name	Type	Description	Value	Constraints	Notes
user.totalRide	integer	-	-	-	-
user.cancelRide	integer	-	-	-	-

> Examples of payload

Getting two users data who are subscribed on the system


```
[
  {
    "socketID": "HNR9B5MTTLgY_NJzAAPV",
    "userID": "FrMMP0MeBkRDCoBUAAAH",
    "coords": {
      "latitude": 41.58910558784315,
      "longitude": -1.090098864658791
    },
    "user": {
      "name": "Javier Fernandez",
      "score": 2.67,
      "totalRide": 253,
      "cancelRide": 12
    }
  },
  {
    "socketID": "HNR9B0Dv1fLgY_NJzABBAc",
    "userID": "FrMMP0MeBkRDCoHHHAAcc",
    "coords": {
      "latitude": 43.5891044413665,
      "longitude": -3.00988643441111
    },
    "user": {
      "name": "Juan Diez",
      "score": 3.67,
      "totalRide": 17,
      "cancelRide": 0
    }
  }
]
```


Message `end`

Once a taxi arrives at user position, server send back to the cabbie that travel is already completed.