Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de un servicio de aparcamiento en Smart Cities empleando una solución basada en integración de datos.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Benjamín Hernández Benavent

Tutor: Joan Josep Fons Cors

Curso: 2018-2019

Resumen

En este Trabajo de Fin de Grado se ha diseñado una aproximación IoT a la problemática del aparcamiento en las ciudades a la par que se ha implementado ciertas funcionalidades presentes en un entorno como el abordado.

Se describen en el escrito las soluciones tomadas en este campo pertenecientes al ámbito de las "Ciudades Inteligentes", en concreto el denominado "Aparcamiento Inteligente", así como un diseño basado en el procesamiento de mensajes desde distintas fuentes e interconectando los diferentes componentes creando así una red de sensores, vehículos y otros componentes compartan información. Además, se presenta una implementación en Python de la interacción entre sensores, vehículos y la red central que permite a los usuarios reservar plazas de aparcamiento u obtener las plazas más cercanas sin necesidad de aplicaciones móviles, abordando problemas como la integración de datos.

Palabras clave: IoT, "Aparcamiento Inteligente", procesamiento de mensajes, reservar plazas de aparcamiento, integración de datos.

Abstract

In this Bachelor's Thesis an IoT-based solution to the parking space problem in cities is designed, as well as the implementation of some useful functionalities in this kind of scenario.

Actions taken in this field belong to Smart Cities scope, to be concrete, in Smart Parking scope are described in this paper. Furthermore, it is described a message-processed design based on different sources interconnected to build a network where sensors, vehicles and other components share information. It is also presented a Python implementation of the interactions between sensors, vehicles and a central network that enable users to book parking spaces or to get the ones near to their position without the need of a mobile application, approaching problems as data integration.

Implementación de un servicio de aparcamiento en Smart Cities empleando una solución basada en integración de datos.

Keywords: IoT, Smart Parking, message-processing, book parking spaces, data integration.

Tabla de contenidos

Contenido

1. Introducción 9
   1.1. Motivación 10
   1.2. Objetivos 10
   1.3. Estructura 11
   1.4. Planificación y Metodología 12
2. Estado Del Arte 13
   2.1. Origen y Evolución 13
      2.1.1. Sistema de Información y Guía en un Aparcamiento 15
      2.1.2. Sistema Basado en la Información del Tráfico 16
      2.1.3. Sistemas inteligentes de pago 17
      2.1.4. E-Parking 17
      2.1.5. Aparcamiento Automatizado 17
   2.2. Elementos del sistema 18
      2.2.1. Detección del vehículo 18
      2.2.2. Gestión de la ocupación 19
      2.2.3. Comunicaciones 20
   2.3. Propuestas actuales 20
      2.3.1. España 20
      2.3.2. Resto del mundo 21
   2.4. Crítica al estado del arte 21
   2.5. Propuesta 21
3. Análisis del problema 23
   3.1. Especificación de requisitos 23
      3.2. Casos de uso 24
      3.3. Descripción de escenarios 28

3.4. Identificación y análisis de soluciones posibles ... 30
3.5. Solución propuesta ... 30
4. Diseño de la solución ... 31
4.1. Arquitectura del sistema ... 31
4.1.1. Fuentes de datos ... 31
4.1.2. Procesadores de datos ... 32
4.1.3. Almacenamiento de datos ... 33
4.1.4. Comunicaciones ... 33
4.1.5. Esquema general ... 34
4.2. Diseño detallado ... 35
4.3. Tecnología Utilizada ... 38
5. Desarrollo de la solución ... 41
5.1. Estructura general de la solución ... 41
5.2. Proceso de desarrollo ... 41
6. Pruebas ... 51
7. Conclusiones ... 55
7.1. Relación del trabajo desarrollado con los estudios cursados ... 55
8. Referencias ... 57
Anexo ... 59

Tabla de ilustraciones

Ilustración 1. Principales ciudades en el ranking INRIX 14
Ilustración 2. Ejemplo de Sistema Guiado de Aparcamiento. Fuente: Omnitec Group 15
Ilustración 3. Mapa obtenido el 23-06-2019 de la ciudad de Valencia del tráfico en directo 16
Ilustración 4. Ejemplo de parking automatizado. Fuente: http://blog.getmyparking.com/2017/08/30/an-overview-of-automated-parking-system-and-its-types/ 18
Ilustración 5. Sensor magnético de la marca Libellium 19
Ilustración 6. Casos de Uso 24
Ilustración 7. Formatos de mensaje. 33
Ilustración 8. Esquema general de la arquitectura. 34
Ilustración 9. Diagrama de clases de la parte de sensores del aparcamiento. 35
Ilustración 10. Diagrama de clases de la interacción con el vehículo y el usuario. 35
Ilustración 11. Ejemplo de mensaje JSON. 36
Ilustración 12. Ejemplo de mensaje XML 36
Ilustración 13. Ejemplo de mensaje enviado por un vehículo. 37
Ilustración 14. Ejemplo de comunicación para reserva. 37
Ilustración 15. Estructura general de la solución 41
Ilustración 16. Ejemplo de generación de información en XML 42
Ilustración 17. Publicación de mensajes en colas RabbitMQ 43
Ilustración 18. Inserción de elementos en la base de datos a través de la API 44
Ilustración 19. Ejemplo de método GET en la API 44
Ilustración 20. Transformación de mensajes de XML a JSON 45
Ilustración 21. Ejemplo de uso de la librería Schedule 46
Ilustración 22. Función de generado automático de coordenadas pseudoaleatorias ... 47
Ilustración 23. Flujo de ejecución de la lógica del vehículo. 47
Ilustración 24. Fragmento de código donde se calcula la plaza más cercana 48
Ilustración 25. Función de sincronización entre sensores y base de datos 49
Ilustración 26. Envío de información desde SensorSimulator.py. 51
Ilustración 27. Recepción de datos en XML y transformación a JSON. 51
Ilustración 28. Mensajes listos para insertar en la base de datos. 51
Ilustración 29. Diferencia entre cuando se crean los sensores (peticiones POST) a cuando se actualizan (peticiones PUT). 52

Ilustración 30. Vista de la base de datos. 52
Ilustración 31. Reserva de vehículo 52
Ilustración 32. Ejemplo de uso si el usuario tiene reserva. 53
Ilustración 33. Cambio de estado del sensor. 53
Ilustración 34. Ejemplo de uso si el usuario no tiene reserva. 53
Ilustración 35. Reserva realizada a través del método dinámico. 53
Ilustración 36. Ejemplo de uso si el usuario desea otra plaza. 53

1. Introducción

La situación de las ciudades alrededor del mundo evoluciona dentro de las nuevas tecnologías hacia un nuevo modelo de urbe interconectada en la que los diferentes servicios se comunican los unos con los otros buscando una mejora sustancial. Estas ciudades son denominadas “**Ciudades Inteligentes**”. Según Wikipedia, una Ciudad Inteligente puede ser definida como **"inteligente"** o como **"inteligentemente eficiente"**, cuando la inversión social, el capital humano, las comunicaciones, y las infraestructuras, conviven de forma armónica con el desarrollo económico sostenible, apoyándose en el uso y la modernización de nuevas tecnologías (TIC), y dando como resultado una mejor calidad de vida y una gestión prudente de los recursos naturales, a través de la acción participativa y el compromiso de todos los ciudadanos.

Según esta misma fuente, una **"ciudad inteligente"** viene a ser un sistema eco sostenible de gran complejidad (sistema que contiene muchos subsistemas), o sea, un ecosistema global en el que coexisten múltiples procesos íntimamente ligados y que resulta difícil abordar o valorar de forma individualizada. [1]

Asimismo, otra definición diferente posible al concepto de “**Ciudad Inteligente**” podría ser esta:

“Una Ciudad Inteligente es aquella que coloca a las personas en el centro del desarrollo, incorpora Tecnologías de la Información y la Comunicación en la gestión urbana y usa estos elementos como herramientas para estimular la formación de un gobierno eficiente que incluya procesos de planificación colaborativa y participación ciudadana.” (Bouskela, Casseb, Bassi, & De Luca, 2016) [2]

En el contexto de nuevas metrópolis que evolucionan hacía un modelo de sostenibilidad basado en las soluciones del Internet de las Cosas se pueden catalogar diversas subsecciones como podría ser la gestión eficiente de la luz, el agua, el transporte público u otros elementos que pertenecen al ámbito de la vida urbana. Dentro de estas secciones podemos encontrar el Aparcamiento Inteligente.

El “**Smart Parking**” consiste en la gestión del aparcamiento mediante la utilización de sensores, componentes del Internet de las Cosas y aplicaciones multiplataforma de

una forma eficiente con fin de reducir el tiempo empleado en buscar una plaza para estacionar cada vehículo, reducir el tráfico y la contaminación. Este concepto está convirtiéndose en prioridad dentro del ámbito gubernamental en los últimos años con el auge del Internet de las Cosas, las redes 5G y los coches autónomos. Estas cuestiones son interesantes en muchos aspectos como el económico, el medioambiental, la calidad de vida del ciudadano y el nivel de saturación de las vías urbanas.

1.1. Motivación

El coche es uno de los métodos de transporte más habituales en nuestro día a día debido a la distancia respecto al lugar de estudio o puesto de trabajo, la falta de transporte público cerca de nuestra residencia u otros motivos. A causa de esto, miles y miles de personas deciden desplazarse en este medio de transporte, pero su uso en las grandes ciudades puede ser un auténtico caos debido al tráfico, a las obras públicas, los eventos multitudinarios y otros factores que influyen en la vida urbana diaria [3]. A esta problemática se le añade la búsqueda de áreas de estacionamiento, puesto que la cantidad de plazas libres suele ser inferior a la cuantía de automóviles que las necesitan. Por consiguiente, muchas poblaciones han optado por transportes alternativos y más sostenibles como el metro, el bus, la bicicleta o las motos y patinetes eléctricos, facilitando de esa manera el estacionamiento a las afueras de estas. Aunque apoyamos y defendemos estas alternativas que protegen de una forma más eficiente el medioambiente y reducen el tráfico, queremos aportar facilidades tanto a los conductores habituales como a los encargados de controlar el tráfico de estas ciudades y seguir mejorando en la búsqueda de un mundo menos contaminado.

1.2. Objetivos

Los objetivos a cumplir son realistas acordes al tiempo de desarrollo que vamos a emplear.

En primer lugar, diseñar un sistema inteligente de aparcamiento de acuerdo con las necesidades de una gran ciudad siguiendo los estándares IoT y aplicando los principios de las Ciudades Inteligentes.

En segundo lugar, implementar la interacción entre los diferentes elementos del sistema diseñado que cumplan con los casos de uso posibles del sistema.

Finalmente, estudiar los beneficios ofrecidos a las grandes ciudades con este tipo de aparcamiento para demostrar que esta estructura es viable.

1.3. Estructura

En este primer capítulo, los conceptos de “**Smart City**” y “**Smart Parking**” han sido introducidos junto a la relación entre estos. Dentro de este segundo término, serán explicados algunos de sus beneficios y objetivos que queremos lograr con su implantación al mismo tiempo que otros aspectos para tener en cuenta.

En el capítulo 2 se hará un estudio sobre las propuestas que se han realizado hasta ahora en sistemas de “**Smart Parking**”, tanto los implantados y en uso como los teóricos. Se hará un desglose de estas opciones y se expondrá a qué parte del sistema pertenecen (sensores, middleware, aplicación de uso, etc.) y las interpretaremos teniendo en cuenta sus pros y sus contras. También se hará un estudio a lo largo de la historia del porqué de estos sistemas, qué nos ha llevado a ellos y hacia dónde vamos. Al final de este apartado se expondrán las características en las cuales nos centraremos dentro de la amplitud de conceptos que conlleva esta tecnología.

En el capítulo 3 se analizará el problema del aparcamiento en las grandes metrópolis estudiando los diferentes escenarios y casos de uso posibles.

En el capítulo 4 se diseñará la propuesta de solución al problema descrito anteriormente en detalle, tanto arquitectura como las tecnologías a utilizar.

En el capítulo 5 se hará una descripción exhaustiva de todo el proceso de desarrollo de la solución final comentando tanto aspectos relevantes como dificultades a la hora de desarrollar nuestro sistema.

En el capítulo 6 se mostrarán las pruebas realizadas al sistema si han resultado satisfactorias o fallidas.

Finalmente, en el capítulo 7 se revelarán las conclusiones obtenidas del diseño, desarrollo e implementación del producto y la relación que tiene el trabajo realizado con los estudios de Ingeniería Informática.

1.4. Planificación y Metodología

El proceso de investigación, desarrollo y redacción tendrá un tiempo aproximado de unas diez a doce semanas de las cuales se dedicarán de tres a cuatro semanas a la investigación del campo de estudio tratado y del estado tecnológico de este, tres o cuatro semanas dedicados al desarrollo de la implementación y una o dos semanas dedicadas a la redacción del documento. Los tiempos descritos son absolutos y no implica que los diferentes pasos se realicen uno detrás de otro sino intercalándolos. La metodología aplicada se apoya en las denominadas SCRUM o agiles, aunque no de una forma estricta, realizando un desarrollo incremental añadiendo funcionalidades pero sin estructurarlo en ciclos estrictos.

2. Estado Del Arte

En primer lugar, este apartado es adecuado dividirlo en diferentes subsecciones en vista de la complejidad de la materia a exponer. Estas subsecciones siguen criterios determinantes a la hora de clasificar qué partes dentro del concepto de "**Aparcamiento Inteligente**" representan a la vez que se dedicarán otros subapartados a cuestiones como el origen y la evolución además de las soluciones ya implantadas en la actualidad como forma de negocio.

2.1. Origen y Evolución

Para llegar al origen de la necesidad de las ciudades con el fin de buscar solución al problema de la masificación automovilística no es necesario recuperar datos de hace 20 años. Según un estudio realizado por Urban Science y Aniacam, a principios de 2017 en España había una cantidad de 30 millones de vehículos circulando, de los cuales 22 millones correspondían a turismos [4]. A este dato se le puede añadir que en España hay una población entre 18 y 70 años de 32.716.114 ciudadanos obtenido a través del Instituto Nacional de Estadística. Está cerca de llegar a la situación en la cual cada habitante con capacidad de conducir tenga un vehículo privado propio. No es un estudio exhaustivo, pero denota el volumen de tráfico posible que pueden recibir las principales poblaciones españolas.

Un dato más concreto acerca del volumen de tráfico se puede obtener observando el ranking "INRIX Global Traffic Scorecard" donde aparecen las ciudades a nivel mundial en las que más tiempo se pierde en atascos y entre las 100 primeras se encuentran 3 ciudades españolas: Madrid, Barcelona y Valencia.

URBAN AREA	2018 IMPACT RANK (2017)	HOURS LOST IN CONGESTION	YEAR OVER YEAR CHANGE	COST OF CONGESTION (PER DRIVER)	INNER CITY TRAVEL TIME (MINUTES)	INNER CITY LAST MILE SPEED (MPH)
Madrid	22 (23)	129 (74)	3%	-	7	8
Barcelona	38 (39)	147 (48)	5%	-	7	9
Valencia	53 (68)	136 (66)	14%	-	6	10
Zaragoza	123 (139)	77 (150)	18%	-	6	10
Bilbao	159 (170)	95 (125)	13%	-	7	9
Cordoba	217 (219)	33 (217)	24%	-	4	14

Ilustración 1. Principales ciudades en el ranking INRIX

Estas circunstancias demuestran la necesidad de aliviar el tránsito hacia las grandes ciudades. Dentro de estas se añaden otros problemas al conflicto. Un estudio publicado por Xerox afirma que los españoles pierden una media de 96 horas al año en buscar aparcamiento, es decir, 4 días de 365 que tiene el año. Al día esto se traduce en 15 minutos diarios. En otras ciudades europeas, como Londres o París, esta espera se puede ver alargada a 20 o 30 minutos. Esta situación empuja cada año a más y más personas a cambiarse a otros medios de movilidad sostenible como podrían ser la bicicleta, el metro o el patinete eléctrico.

En el origen de las “Ciudades Inteligentes” o en nuestro caso del “Aparcamiento Inteligente” tienen especial importancia la aparición de las tecnologías IoT. Las tecnologías IoT (*Internet of Things*) se pueden definir como un paradigma en el cual los objetos se comunican los unos con los otros mediante sensores para servir a un propósito significativo [5]. En otras palabras, las tecnologías IoT sirven para comunicar diferentes objetos, servicios o elementos de una ciudad o una casa por ejemplo y que se produzca un intercambio de información para la realización de una tarea concreta, como podría ser en nuestro caso, una búsqueda eficiente de un espacio para dejar nuestro coche en la ciudad. Gracias a estas tecnologías los diferentes elementos críticos de las urbes son capaces de controlarse de una forma más centralizada y obtener su estado de funcionamiento en tiempo real.

Después de este preámbulo, se procede a describir las diferentes propuestas a lo largo de los años desde los sistemas más básicos a los más complejos. En el estudio de Idris et al. (2009) “Car Park System: A Review of Smart Parking System and its Technology” [6] se nombran y se describen las cinco principales categorías de Aparcamiento Inteligente existentes: Sistema de Información y Guia en un

Aparcamiento (en inglés, Parking Guidance and Information System), sistema basado en la información del tráfico, sistema inteligente de pago, E-parking y parking automatizado. A continuación, se describirán cada uno de ellos y se expondrán los inconvenientes de cada una de ellas:

2.1.1. Sistema de Información y Guía en un Aparcamiento

Estos sistemas se caracterizan por proporcionar al conductor información sobre donde se encuentran los espacios desocupados y guía al usuario a través de las calles o en su defecto, dentro de un parquin privado mediante señales visuales variables o mediante posicionamiento GPS señalizado a través de dispositivos móviles. Se pueden encontrar implementados en una gran cantidad de centros comerciales con espacio para vehículos privado, sin embargo, es más complejo implementar estos sistemas al aire libre.

Ilustración 2. Ejemplo de Sistema Guiado de Aparcamiento. Fuente: Omnitec Group

Un ejemplo al aire libre lo encontramos en la ciudad de Singapur desde 2008, el cual incluye paneles a lo largo de la ciudad, acceso a través de la web e incluso aplicación móvil.

Esta medida es la más básica de todas ya que solo nos proporciona información del número de zonas libres, no nos informa ni del estado de la circulación ni nos permite reservar una plaza ni pagar de forma eficiente.

2.1.2. Sistema Basado en la Información del Tráfico

Este recurso utiliza sensores distribuidos a lo largo de las vías urbanas con el propósito de informar al usuario el estado de estas en cuanto a congestión de vehículos y optimizar la ruta seleccionada para ahorrar la mayor cantidad de tiempo, combustible y con ello contaminar menos. No es realmente una solución para el problema del estacionamiento, pero sí entra en esta categoría porque puede aportar una reducción al tiempo que tarda un usuario en encontrar un sitio para ello. Un ejemplo a gran escala es la función de Google Maps de obtener la información del tráfico. El principal inconveniente es que el flujo de coches es muy complicado de controlar y los atascos se forman de una forma muy rápida.

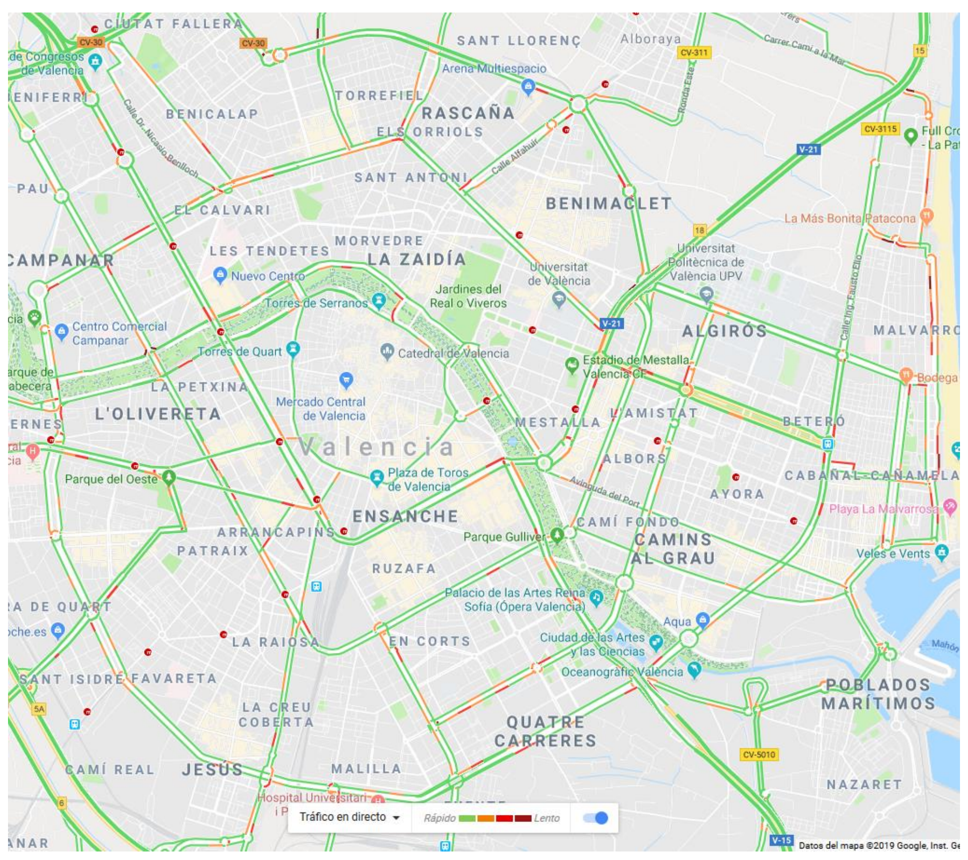


Ilustración 3. Mapa obtenido el 23-06-2019 de la ciudad de Valencia del tráfico en directo

2.1.3. Sistemas inteligentes de pago

Este tipo de sistemas se centran en agilizar y facilitar al usuario métodos de pago que requieran poco tiempo o se hagan automáticamente. Actualmente hay una lista amplia de formas de pago diferentes que requieren poco tiempo como pueden ser las tarjetas de crédito *contactless*, el pago a través del móvil vía NFC o pago previo a uso a través de aplicaciones móviles o PayPal. Un ejemplo claro en la actualidad es 3SCPark, una empresa que ofrece una app móvil para acceder a los usuarios y pagar la estancia.

En el estudio [7] menciona que el pago del aparcamiento se realizará automáticamente a través de la cuenta del usuario. Sin embargo, en [8] el pago de la plaza se realiza a través de una aplicación donde introduces tu tarjeta o tu cartera electrónica. No obstante, la mayoría de los trabajos y los estudios consultados no inciden en este aspecto debido a que su implementación tiene similitudes con otros tipos de pasarelas de pago.

2.1.4. E-Parking

Este mecanismo de gestión es de los más innovadores y estudiados dentro de los distintos tipos. Las funciones que incluye son las de reserva y pago de plazas de aparcamiento, búsqueda de los espacios más cercanos e incluso gestión del tráfico. A diferencia de los anteriores, incluye la función de reservar un sitio para aparcar por adelantado, asegurando al conductor la disponibilidad de esta y reduciendo el tiempo necesario para llegar a su destino. En artículos científicos como [9] [10] [11] se estudian diferentes implementaciones de E-Parking pero el concepto base de todos ellos es el mismo.

Estas implementaciones se adaptan de una forma muy adecuada al contexto actual con soluciones eficientes accesibles a través de dispositivos móviles.

2.1.5. Aparcamiento Automatizado

En este último sistema se confía plenamente el control y cuidado del vehículo a un conjunto de maquinaria que lo transportan desde la entrada hasta un espacio libre. Se describe en el escrito [12] el caso de la empresa holandesa *CVSS Automated Parking Systems*, tanto el funcionamiento del mismo como el tipo de maquinaria que contiene. Este mecanismo de optimización almacena los coches en el menor espacio posible y

despreocupa a los usuarios desde el momento en el que se deja el vehículo. Otros ejemplos de parking automatizados, por ejemplo, en España, son Parking Pizarro 10 en Valencia, Parking Callao en Madrid o Parking Las Ramblas Palau Nou en Barcelona. Cabe destacar también la aplicación de esta tecnología al guardado de bicicletas en Jiyugaoka, Tokio.

Ilustración 4. Ejemplo de parking automatizado. Fuente: http://blog.getmyparking.com/2017/08/30/an-overview-of-automated-parking-system-and-its-types/

2.2. Elementos del sistema

En esta sección se presentan las diferentes tecnologías utilizadas o estudiadas para su utilización en el asunto en cuestión. Cada subsección se centra en partes del sistema con funciones diferenciadas, como son la detección del vehículo, la organización de las plazas libres o las comunicaciones entre las diferentes tareas.

2.2.1. Detección del vehículo

En este campo hay una gran cantidad de propuestas diferentes y por ello mencionaremos las más importantes. Dentro de estos métodos de detección hay dos subclases diferenciadoras, los sensores intrusivos y los sensores no intrusivos. Los sensores intrusivos se colocan de tal forma que se necesita realizar obras para quitarlos, puede ser dentro del pavimento o en otros sitios similares. Son menos sensibles a elementos como la meteorología, el vandalismo u otras causas de deterioro. Los sensores no intrusivos se colocan sin necesidad de realizar labores de

construcción. Su mantenimiento es más sencillo, sin embargo, sufren mayor deterioro. El primer sensor a estudio es el sensor de inducción magnética. Su función es detectar si hay algún automóvil o moto en el espacio que controla (normalmente se sitúan encima o debajo del espacio que quieren controlar) y envían esa información a un dispositivo que pueda comunicarse a largo alcance para que esa información llegue al servidor central o a la nube. Un ejemplo son los sensores Libellium.

Ilustración 5. Sensor magnético de la marca Libellium

Otros sensores son los sensores por infrarrojos que se guían por la cantidad de energía que emiten los vehículos, los sensores de ultrasonidos que funcionan similar a un sonar, es decir, conocen la posición de un vehículo a través del rebote de los ultrasonidos o sensores por RFID que identifican al auto al situarse en su sitio correspondiente.

Una forma más compleja de detección es el análisis de video. Se colocan cámaras en las diferentes plazas de aparcamiento y mediante técnicas de procesamiento de video se analizan las imágenes y se obtiene información como si la plaza está ocupada o la matrícula del coche. Una empresa que ofrece estos servicios es Covert Security.

2.2.2. Gestión de la ocupación

Estos proyectos no se podrían considerar "inteligentes" si no hubiera una lógica detrás de la asignación de los recursos disponibles. Con el propósito de ofrecer este servicio, se implementan algoritmos de optimización los cuales se procede a describir algunos de ellos. En la publicación [13], se estudian dos modelos de emplazamiento de los vehículos, uno basado en la demanda global del sistema y el otro basado en la *filosofía first-book-first-served* y los resultados de aplicar estas dos metodologías. En otro ensayo [14], utilizan un algoritmo de eficiencia basado en colas una para usuarios normales y otras para personas con reserva. Para concluir con este apartado en otros

trabajos como [10] se menciona como un problema matemático estocástico y determinista que debe ser solucionado mediante MILP (*Mixed Integer Linear Programming*).

2.2.3. Comunicaciones

Los elementos del sistema, tanto sensores como el sistema central deben poder comunicarse de una forma efectiva y rápida, puesto que para ofrecer a la población un servicio que controle los recursos en tiempo real las comunicaciones deben ser fluidas. En este aspecto entran en juego los siguientes elementos:

Tipo de red: Al estar los sensores esparcidos por un área amplia de espacio, la mayoría de los autores optan por redes inalámbricas, en concreto redes Wi-Fi para interconectar los diferentes dispositivos. Hay artículos como [15] que también sugieren redes inalámbricas de corto alcance como Zigbee. Dependiendo de la amplitud de la zona a controlar se utilizan elementos como puntos de acceso para facilitar la interconexión de la red. En [8] también se propone una red GSM para las reservas por SMS

Middleware: Hay pocos trabajos que se centren en desarrollar un middleware para estas soluciones IoT. Sin embargo, los que inciden en este tema apuestan por un middleware basado en el procesamiento en la nube debido a su enorme escalabilidad y disponibilidad.[15] propone utilizar un servidor IBM MQTT alojado en la nube y la utilización de estas colas MQTT para la comunicación.

2.3. Propuestas actuales

En este punto se nombrarán y describirán los diferentes negocios o soluciones implementadas en el mundo real que estén en funcionamiento.

2.3.1. España

Dentro del territorio peninsular podemos encontrar, como se ha mencionado anteriormente, Parking Pizarro 10 en Valencia, Parking Callao en Madrid o Parking Las Ramblas Palau Nou en Barcelona, aunque estos se traten de versiones

automatizadas. A nivel de pago eficiente encontramos E-Park, una empresa que trabaja a nivel nacional en distintas ciudades para agilizar el pago de los parquímetros a través del teléfono móvil. Además de esta funcionalidad, la empresa Elparking te ofrece la localización de los párquines más cercanos. Lo más parecido a una implantación en España lo encontramos en los aparcamientos privados de la mano de la empresa 3SCParking. Aunque a nivel de implementación España está por detrás de otros países tiene implementados ejemplos en Santander, Málaga o Madrid. A nivel de tecnología, la empresa de Zaragoza Libellium fabrica una de las tecnologías más punteras del mercado en términos de sensores y ha realizado proyectos como en el ayuntamiento de Montpellier o el aeropuerto internacional de Atenas.

2.3.2. Resto del mundo

A nivel mundial se pueden encontrar implementaciones como la anteriormente mencionada de Montpellier, el aparcamiento del metro de Los Angeles, el Fastprk en Riyadh (Arabía Saudí), la ciudad suiza de Zug y muchos otros más.

2.4. Crítica al estado del arte

Tras un estudio intensivo de los diferentes tipos de sistemas, las diferentes tecnologías y sus implementaciones, cabe hacer una valoración de lo descrito. En primer lugar, aunque haya una gran cantidad de artículos estudiando la optimización de espacios para aparcar, pocos de ellos lo estudian en espacios abiertos y no controlados. Se ha de hacer un inciso también en la falta de concienciación por parte de los autores en la necesidad de escalar estos sistemas y fácilmente accesibles con servicios como la nube. Tampoco se preocupan por el tratamiento de los datos y su adaptación entre dispositivos de diferentes marcas. Para finalizar, aún siendo claramente beneficioso a nivel ecológico pocas ciudades han apostado por este tipo de propuestas inteligentes.

2.5. Propuesta

Nuestro trabajo va a consistir en focalizar nuestro esfuerzo en realizar una propuesta escalable, aunque esté realizada en un entorno local y remarcar el concepto del tratamiento de los datos sin abandonar la idea principal que es implementar una

solución de “Smart Parking”. Visto que es un campo estudiado en profundidad por muchos autores, cogeremos ideas de sus escritos, como por ejemplo el tipo de sensor o la estructura del sistema y aplicaremos nuestros conocimientos en materia de desarrollo de aplicaciones para que sea una solución original.

3. Análisis del problema

El punto inicial previo al desarrollo consiste en el estudio en detalle del problema planteado y posibles enfoques alternativos al propuesto por nosotros. Para ello utilizaremos distintas técnicas de modelado conceptual y especificaremos también que requisitos son necesarios para nuestro sistema.

3.1. Especificación de requisitos

Nuestro proyecto necesita cumplir una serie de requisitos para conseguir que se adapte a las necesidades de una ciudad interconectada que, mediante la comunicación de los diferentes elementos involucrados, consiga gestionar el servicio que pretendemos dar. Para ello, es necesario, en primer lugar, una gran cantidad de sensores repartidos a lo largo del terreno que envíen información.

Estos sensores pueden estar situados en sitios tan dispares como en la propia calle, en semáforos, en obras públicas incluso en grandes avenidas calculando el tráfico que está circulando en un instante concreto. Es posible también utilizar cámaras de tráfico como fuente de datos mediante técnicas de procesado de imagen.

Esta gran cantidad de mensajes deben ser procesados de alguna forma para ser consumidos por otras herramientas y que sean transformados en servicios para la población o para los entes públicos o privados. Debido a la naturaleza heterogénea que puede encontrarse entre diferentes fuentes es necesario fomentar la integración entre ellas utilizando transformadores de diferentes formatos a otros. Una buena práctica dentro del campo de la integración de aplicaciones consiste en crear un formato estándar (conocido como formato canónico) que sirva para facilitar el entendimiento.

Además de transformadores, debido a la gran cantidad de datos, se necesita un sistema que almacene temporalmente los mensajes para ser utilizados a posteriori. Esta función es posible con la implementación de un gestor de colas de mensajería. Hay una gran cantidad de tecnologías de gestión de colas, pero una de las más

utilizadas es RabbitMQ (*open-source*). Empresas como Amazon, Microsoft o IBM tienen sus propios gestores.

Para finalizar, se necesitará una estructura lógica que opere como "cerebro" y proporcione la inteligencia ante una ingente cantidad de información. Para que la gente u otros servicios se beneficien de la función realizada, es conveniente el despliegue de APIs, Servicios Web u otras soluciones que faciliten la conectividad.

3.2. Casos de uso

Se procede a describir los diferentes casos de uso que hemos podido identificar dentro de la problemática que pretendemos abordar.

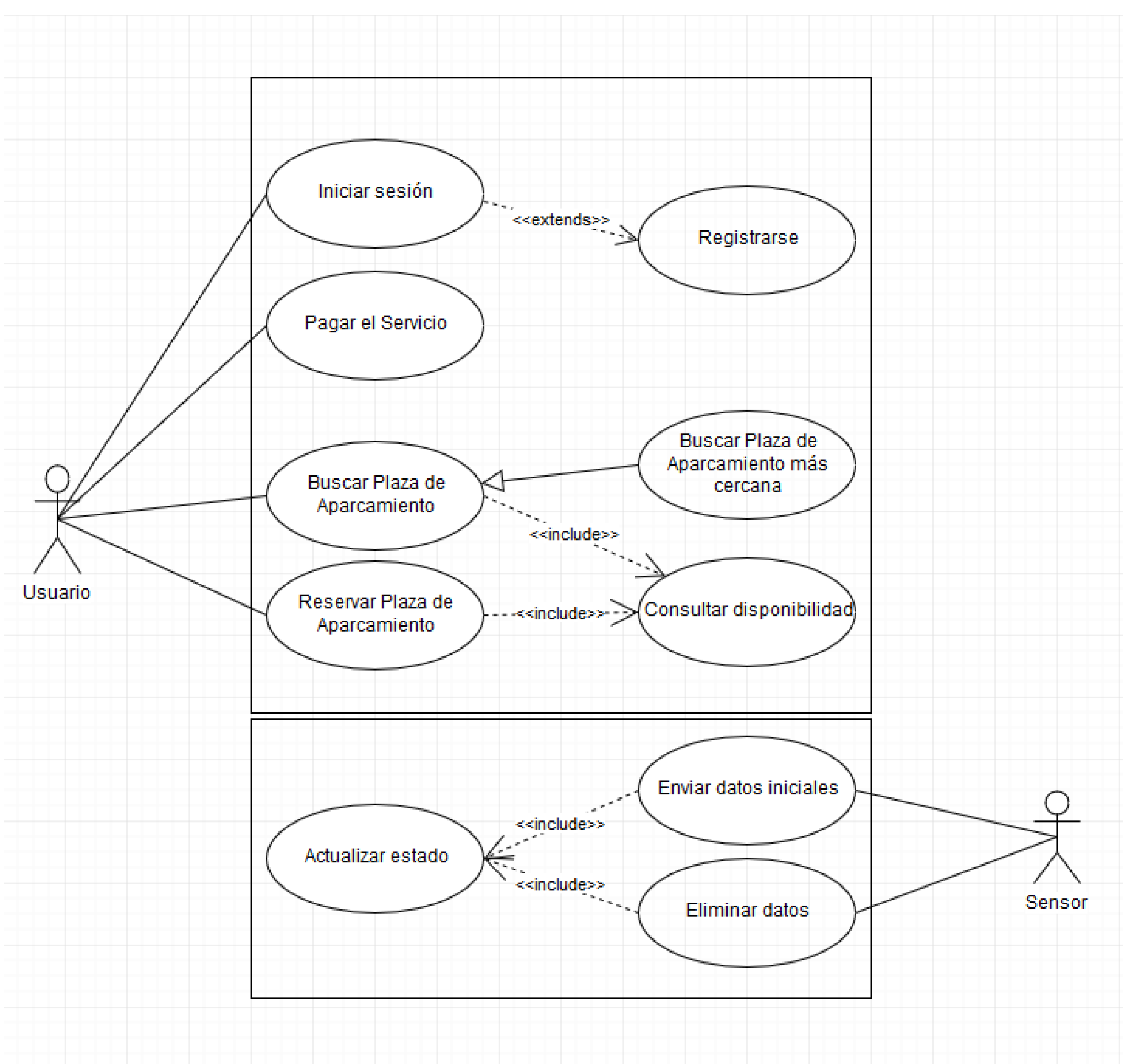


Ilustración 6. Casos de Uso

En este diagrama se describen los actores participes de los usos posibles del sistema y que acciones incluyen otras. Se procede a describir estos casos de uso con más detalle en las siguientes tablas:

Caso de uso	Iniciar sesión
Actores	Usuario (iniciador)
Propósito	Autenticarse en el sistema con sus credenciales
Resumen	El usuario abre la aplicación e introduce sus credenciales para poder acceder al servicio.
Precondiciones	El usuario debe haberse registrado en el sistema.
Postcondiciones	La sesión del usuario queda almacenada en el sistema.
Incluye	-
Extiende	Registrar el usuario
Hereda de	-

Tabla 1. Caso de Uso Iniciar Sesión

Caso de uso	Buscar Plaza de Aparcamiento
Actores	Usuario (iniciador)
Propósito	Buscar una plaza de aparcamiento a deseo del usuario.
Resumen	El usuario elige una plaza de aparcamiento dentro de las opciones.
Precondiciones	El usuario debe iniciado sesión

Postcondiciones	El sistema le debe mostrar la disponibilidad de la plaza.
Incluye	Consultar disponibilidad
Extiende	-
Hereda de	-

Tabla 2.Caso de Uso Buscar Plaza de Aparcamiento

Caso de uso	Buscar Plaza de Aparcamiento más cercana.
Actores	Usuario (iniciador)
Propósito	Buscar una plaza de aparcamiento más cercana a la posición del usuario.
Resumen	El usuario introduce su localización al sistema
Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	El sistema le debe mostrar la disponibilidad de la plaza más cercana.
Incluye	Buscar Plaza de Aparcamiento
Extiende	-
Hereda de	-

Tabla 3. Caso de Uso Buscar Plaza de Aparcamiento más Cercano

Caso de uso	Reservar plaza de aparcamiento.
Actores	Usuario (iniciador)
Propósito	Reservar una plaza de aparcamiento a elección del usuario.
Resumen	El usuario reserva una plaza en una franja horaria.
Precondiciones	El usuario debe haber elegido una plaza disponible.
Postcondiciones	El sistema debe reservar la plaza deseada.
Incluye	Buscar Plaza de Aparcamiento
Extiende	-
Hereda de	-

Tabla 4. Caso de Uso Reservar Plaza de Aparcamiento

Caso de uso	Enviar datos iniciales
Actores	Sensor (iniciador)
Propósito	Registrar el sensor en la base de datos
Resumen	El sensor se conectará al sistema y enviará sus datos iniciales.
Precondiciones	-
Postcondiciones	El sistema debe registrar los datos del sensor en su base de datos.
Incluye	Actualizar datos
Extiende	-
Hereda de	-

Tabla 5. Caso de Uso Insertar Datos Iniciales

Caso de uso	Borrar datos de sensor
Actores	Sensor (iniciador)
Propósito	Borrar el sensor en la base de datos
Resumen	El sensor activará el borrado del sensor en la base de datos si va a apagarse.
Precondiciones	El sensor está registrado en la base de datos.
Postcondiciones	El sistema debe hacer un borrado en su base de datos.
Incluye	Actualizar datos
Extiende	-
Hereda de	-

Tabla 6. Caso de Uso Borrar Datos de Sensor

3.3. Descripción de escenarios

Previamente, hemos descrito los diferentes casos de uso que se podrían originar si se intentara hacer una resolución del problema. Ahora nuestro objetivo es detallar los posibles escenarios que pueden surgir con la utilización del servicio.

Nombre	Reservar una plaza de aparcamiento anticipadamente
Descripción	Permite al usuario reservar una plaza para un periodo de tiempo deseado previo al desplazamiento.
Actor	Cliente
Pre-condición	Estar registrado en el sistema y haber iniciado sesión
Flujo Normal	El usuario introduce los datos del vehículo, la localización de la plaza y la franja horaria deseada en la aplicación. El Sistema reserva la plaza, introduce los datos en la base de datos y envía un justificante.
Flujo Alternativo	2.A. Si la plaza no concuerda con la descripción del vehículo o la franja horaria está ocupada, el sistema devuelve plazas cercanas que puedan ser reservadas.
Post-condición	-

Tabla 7. Escenario reserva anticipada.

Nombre	Reservar una plaza de aparcamiento durante el desplazamiento
Descripción	Permite al usuario reservar una plaza para un periodo de tiempo deseado durante el trayecto al destino deseado.
Actor	Cliente
Pre-	Registrar el vehículo y que estén almacenadas las credenciales en

condición	este.
Flujo Normal	El usuario con su vehículo llega a la ciudad y se envían los datos del vehículo de forma automática. El Sistema ofrece la plaza de aparcamiento más cercana al vehículo que concuerde con sus características. El usuario procede a reservar la plaza. El Sistema reserva la plaza, introduce los datos en la base de datos y envía un justificante y las coordenadas para ser procesadas por el GPS del coche u otros.
Flujo Alternativo	3.A. El usuario puede no guardar la plaza por motivos varios (distancia, precio, sombra…). 3.B El Sistema preguntara en un espacio de tiempo por otra localización más adecuada.
Post-condición	-

Tabla 8. Escenario reserva durante el desplazamiento.

Nombre	Consultar reservas en desplazamiento
Descripción	Permite al usuario consultar su reserva.
Actor	Cliente
Pre-condición	Estar registrado en el sistema y haber iniciado sesión.
Flujo Normal	El usuario consulta las reservas asociadas a su vehículo. El Sistema devuelve los detalles de la reserva y las coordenadas para ser utilizadas por el GPS.
Flujo Alternativo	2.A. El usuario puede no haber realizado reservas en ese caso se procedería a la búsqueda de un espacio disponible.

Post-condición	-

Tabla 9. Escenario consultar reservas en desplazamiento

3.4. Identificación y análisis de soluciones posibles

Aparte de nuestro enfoque IoT al problema, se puede enfocar de la forma tradicional, es decir, un sistema cliente-servidor en el cual las reservas se hicieran siempre previamente al trayecto. Esto sería una solución viable en un entorno donde no todos los coches posean software integrado como si lo hacen los vehículos modernos.

3.5. Solución propuesta

Con estos casos de uso y estos escenarios, nuestra propuesta es enfocar el diseño en crear un entorno dinámico que interconecte los diferentes actores y proporcione escalabilidad y adaptabilidad a variaciones en los escenarios con un tiempo de respuesta adecuado. Esto es posible gracias a las soluciones basadas en el Internet de las Cosas.

4. Diseño de la solución

Tras el análisis del problema, nuestro objetivo es focalizar nuestros esfuerzos en el esquema de nuestro proyecto, como van a interactuar los diferentes componentes entre ellos y justificar el porqué de la elección de un tipo de tecnología y no otra.

Primero, haremos un diseño general de nuestro concepto de aparcamiento inteligente, que piezas del entramado urbano entran en juego y como se complementarán las unas con las otras.

A continuación, se mostrarán diferentes diagramas de diseño software para clarificar como se va a realizar la implementación y en qué sector de la arquitectura general vamos a operar.

Finalmente, se realizará una descripción de las tecnologías que se utilizarán en el proyecto y las funciones que llevan a cabo.

4.1. Arquitectura del sistema

Nuestro enfoque a la hora de buscar un diseño óptimo es aquel en el que el servicio sea escalable (no se vea afectado en gran medida por el tamaño de la ciudad), sea adaptable (es decir, aunque haya cambios en alguno de los componentes, sea fácil adaptar esos cambios para que trabajen adecuadamente) y flexible (acepta nuevas tecnologías y estructuras diversas sin variar el servicio final) entre muchas otras características. Con este objetivo se procede a describir aquellas cosas que deben estar presentes.

4.1.1. Fuentes de datos

La base de nuestro sistema es el procesamiento de datos de diferentes fuentes. Como fuente principal debemos considerar la información obtenida de las propias plazas de aparcamiento, aunque debemos considerar también fuentes de gran valor como

pueden ser semáforos, volumen de tráfico, obras en la vía pública o la existencia de eventos multitudinarios. Todos estos focos de información deben transformarse en datos, pero para ello deben de ser registrados de alguna forma. Hay diferentes métodos para ello, pero nosotros hemos elegido los más adecuados acorde a nuestros criterios.

Para empezar, debemos resolver como obtenemos la información necesaria de las plazas de aparcamiento. Se han barajado diferentes opciones (procesamiento de imágenes, sensores de inducción magnética o de ultrasonidos…) pero finalmente asumimos que la utilización de sensores RFID es la más adecuada para ello ya que nos permite identificar al vehículo portador del identificador y enviar los datos desde el sensor hasta el sistema central y no es tan complejo el procesamiento de la información como el procesamiento de imágenes aunque se pueden utilizar las dos tecnologías como apoyo para obtener mayor fiabilidad en los datos.

La segunda cuestión a tratar de mayor complejidad sería el volumen de tráfico, aunque con los avances actuales, las estimaciones que se consiguen son bastante precisas gracias a las cámaras de tráfico y sensores *in-road.* No tenemos intención de entrar en este tema en mayor profundidad.

Por último, se han de mencionar el resto de las fuentes, que pueden enviar la información directamente a través de la red si están conectados los sensores directamente a internet o a través de transmisores que estén conectados con ellos vía radiofrecuencias de corto o medio alcance. Esto puede ser de gran utilidad para los sensores RFID descritos anteriormente para hacer llegar la información a los diferentes procesadores.

4.1.2. Procesadores de datos

En consecuencia, obtenemos un volumen de datos suficiente para que nuestro servicio comience a funcionar. Sin embargo, hay que tener en cuenta una problemática adicional, no todos los sensores “hablan el mismo idioma”. Significa que no todos los sensores serán del mismo fabricante, ni enviarán los mensajes en el mismo formato, ni aun enviándolos en el mismo formato (por ejemplo, JSON) los campos se encontrarán en el mismo lugar o tendrán los mismos campos.

Los mensajes son la base del servicio, por ello los procesaremos y transformaremos con componentes que cambien del formato propio del sensor a un formato canónico que entiendan el resto de los elementos del sistema y facilite las comunicaciones.

En nuestro caso tenemos sensores que envían información en diferentes formatos como JSON, XML y CSV y mediante una serie de procesadores todo se transforma a un único tipo de formato, JSON. Hemos elegido este porque es el formato más extendido en el ámbito del IoT y es cómodo a la hora de trabajar con él.

```
Nombre,Apellido1,Apellido2
Benjamín,Hernández,Benavent
```

```
<persona>
  <nombre>Benjamín</nombre>
  <apellido1>Hernández</apellido1>
  <apellido2>Benavent</apellido2>
</persona>
```

```
{
  "nombre":"Benjamín"
  "apellido1":"Hernández"
  "apellido2":"Benavent"
}
```

Ilustración 7. Formatos de mensaje.

4.1.3. Almacenamiento de datos

Una característica importante es la persistencia de los datos y para ello, es conveniente la utilización de bases de datos para conseguir persistencia y accesibilidad a los datos del sistema. Hay dos grupos dentro de las bases de datos: relacionales y no relacionales.

Consideramos que, en un entorno real, la cantidad de datos que puede manejar una ciudad inteligente o, en nuestro caso, un aparcamiento inteligente es lo suficientemente cuantiosa como para considerarlo un problema de Big Data. Y en este campo, es recomendable utilizar bases de datos no relacionales para acceder a los datos en el menor tiempo posible y son fácilmente escalables. Ya que nuestra base de datos va a manejar datos complejos, nuestra opción sería MongoDB.

4.1.4. Comunicaciones

El último apartado a tratar dentro del diseño de la arquitectura sería el de las comunicaciones. Es un aspecto crucial dado que depende gran parte del funcionamiento. Los mensajes que se envían necesitan una estructura auxiliar para que persistan dentro del sistema y no desaparezcan. Para ello utilizaremos un software de gestión de colas para almacenar los mensajes temporalmente para que los diferentes componentes puedan consumirlos adecuadamente. En nuestro caso apostamos por RabbitMQ ya que es el más conocido y su funcionamiento cumple con nuestros requisitos.

Otra propiedad es la de facilitar el consumo de estos servicios porque tratándose de un entorno IoT los elementos estarán conectados a Internet. Por lo tanto, habilitaremos

API REST y/o Servicios Web que faciliten la obtención de información sin necesitar software específico para ello.

4.1.5. Esquema general

En este apartado juntamos los subapartados anteriores para dibujar como sería la solución dentro de una ciudad real puesto que es más aclaratorio hacerlo en forma de imagen para ver el escenario completo y no solo parte por parte.

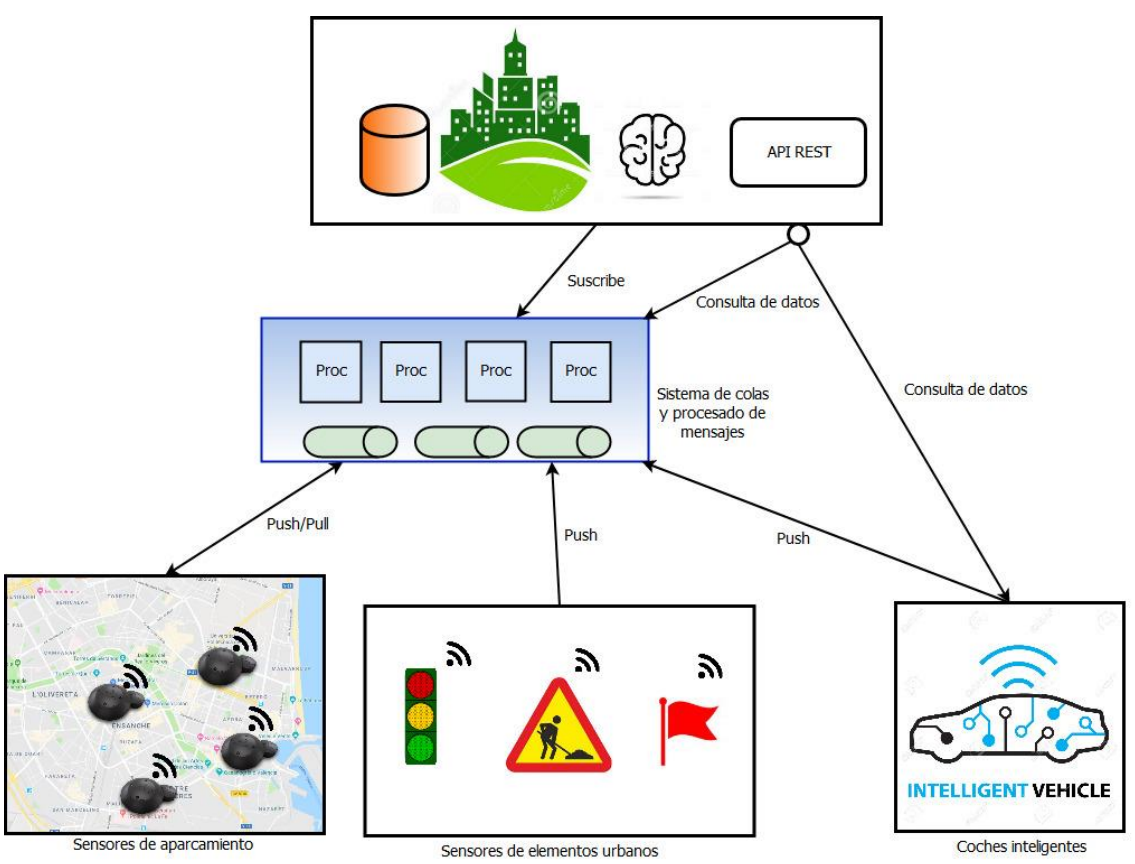


Ilustración 8. Esquema general de la arquitectura.

4.2. Diseño detallado

En el proceso de diseño de nuestro proyecto hemos traducido todos los materiales previos a un esquema de módulos y componentes que nos permitirán conseguir nuestro objetivo. Como no disponemos de sensores reales, incluiremos también el diseño de la simulación de estos. Utilizaremos varios diagramas de clases que nos permitirán ver como interactúan las diferentes partes a la hora del funcionamiento.

El primero es de la parte de los sensores de aparcamiento y el segundo de la interacción con el vehículo.

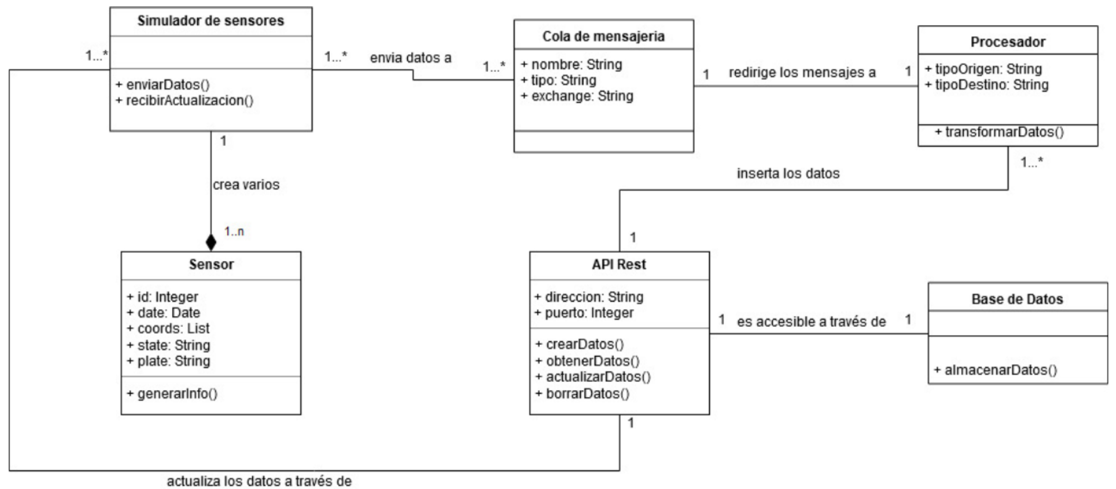


Ilustración 9. Diagrama de clases de la parte de sensores del aparcamiento.

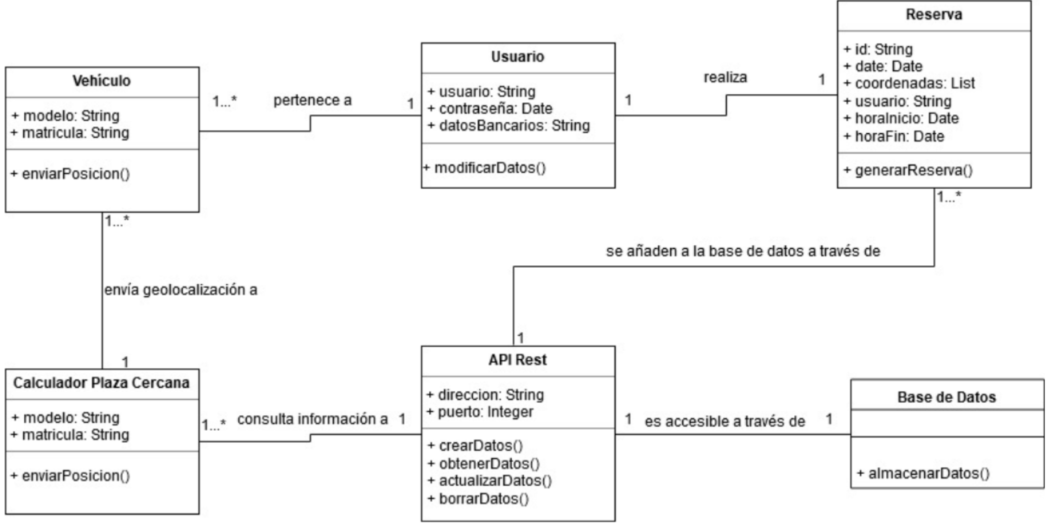


Ilustración 10. Diagrama de clases de la interacción con el vehículo y el usuario.

Después de haber mostrado las clases de nuestra solución entramos en detalle en un apartado decisivo para el correcto desarrollo: los mensajes. Éstos son la fuente de información que interconecta y nutre a los diferentes componentes. Aquellos datos que proporcione un sensor, un automóvil u otros deben ser suficientes para que la parte lógica los utilice y provea a los usuarios del servicio demandado. Anteriormente, hemos mencionado los diferentes tipos de mensaje que podían ser enviados. El propósito de nuestro trabajo es mostrar que información deben dar los mensajes y su estructura.

En primer lugar, hablaremos de los mensajes con origen en los sensores de aparcamiento. Es importante que cada uno de ellos esté identificado, la fecha en la que se ha producido la última actualización, clasificar para que tipo de vehículos está dirigida tanto tamaño, tipo de vehículo y condiciones especiales de la plaza, por ejemplo, que esté habilitada para personas con diversidad funcional. Pero sobre todo saber el estado de la plaza (libre, ocupada o reservada).

```
{
  "sensorID":"458"
  "coords":"39.469771,-0.376791"
  "date":"10-08-2019 19:51"
  "category":"car"
  "description":"" #Por ejemplo, si es para personas con diversidad funcional
  "size":"normal" #Small/normal/big
  "state":"free" #Free/booked/occupied
  "plate":"" #Si hubiera un vehículo estaría la matricula
}
```

Ilustración 11. Ejemplo de mensaje JSON.

Hemos decidido añadir sensores que envíen mensajes en XML para trabajar en la integración de diferentes tipos de fuentes. Este sería el resultado:

```
<sensor>
  <sensorID>"458"</sensorID>
  <coords>"39.469771,-0.376791"</coords>
  <date>"10-08-2019 19:51"</date>
  <category>"car"</category>
  <description>""</description> #Por ejemplo, si es para personas con diversidad funcional
  <size>"normal"</size> #Small/normal/big
  <state>"free"</state> #Free/booked/occupied
  <plate>""</plate> #Si hubiera un vehículo estaría la matricula
</sensor>
```

Ilustración 12. Ejemplo de mensaje XML

Por otro lado, encontramos las comunicaciones con los diferentes medios de transporte. La problemática de la diversidad dentro de los sensores también se repite en estos, puesto que, dependiendo del software instalado en cada uno, pueden enviar la información en diferentes formatos. Nosotros asumiremos que todos los vehículos envían los mensajes en el mismo formato. Estos deben contener tanto la matrícula del vehículo para ser identificado por el sistema, las coordenadas dónde se encuentra, el tipo y el tamaño y tiene adjuntos los datos de pago. En una posible mejora, se podrían añadir los datos directamente si la transferencia fuera segura.

Un ejemplo de mensaje sería así:

```
{
  "vehicle_type":"car"
  "plate":"brr4558"
  "coords":"39.469771,-0.376791"
  "description":"" #Por ejemplo, si es para personas con diversidad funcional
  "size":"normal" #Small/normal/big
  "payment_data":"yes"
}
```

Ilustración 13. Ejemplo de mensaje enviado por un vehículo.

Estos mensajes se pueden utilizar para asignar una plaza adecuada a las necesidades del usuario.

Para concluir con el tema de las comunicaciones, es interesante mostrar también como los usuarios realizan reservas al sistema, las cuales utilizan mucha información acerca del vehículo pero también se debe añadir el código de identificación del sensor reservado, la localización y la hora de inicio y final de la reserva.

```
{
  "plate": "brr4558"
  "date": "10-08-2019 19:51"
  "sensorID":"458"
  "coords": "39.469771,-0.376791"
  "startTime": "17:51"
  "finishTime": "18:51"
  "payment": "yes" #Si se ha pagado la reserva
}
```

Ilustración 14. Ejemplo de comunicación para reserva.

Por otra parte, debemos describir cómo va a ser nuestra base de datos. Al llevar a cabo nuestro proyecto en un entorno local y controlado hemos preferido utilizar una base de datos relacional como es MySQL y ajustarla a nuestras necesidades. Para

ello se ha pensado utilizar una base de datos con dos tablas, una para los sensores y otra para las reservas y que estas se ajusten al formato de mensajes descritos anteriormente para cada una de estas. Por el contrario, en un entorno real se utilizaría una base de datos no relacional ya que estaríamos hablando de un volumen de mensajes descontrolado y una relacional aumentaría el tiempo de respuesta del sistema por la lentitud al acceso de los datos en comparación a las bases de datos no relacionales.

4.3. Tecnología Utilizada

Al principio, se comenzó con la idea de utilizar Java para la simulación de los sensores y Anypoint Studio para la integración de los componentes, pero finalmente, después de un estudio de las posibilidades de los diferentes lenguajes y las funcionalidades que queríamos dar se optó por utilizar Python como tecnología principal del proyecto. Los motivos por los cuales se ha elegido son los siguientes:

- Es un lenguaje de programación con una curva de aprendizaje muy baja.
- Es útil tanto en entornos de programación orientada a objetos como de programación funcional.
- Tiene una gran cantidad de módulos creados por la comunidad. Muchos de ellos convierten a Python en un lenguaje muy potente en el ámbito de la integración de aplicaciones por la facilidad en las tareas de transformación de datos. Describiremos más adelante cuales hemos utilizado nosotros.
- El módulo GeoPy de Python nos facilita la transformación de coordenadas a puntos en el mapa y la distancia entre dos puntos.

Para la Base de Datos hemos utilizado el programa Xampp basado en Apache y MySQL para que sean accesibles las tablas y su contenido a través de la web. A continuación, haremos un breve resumen de la función de cada módulo utilizado.

Json: Este módulo es de gran utilidad para transformar mensajes en formato JSON a objetos de Python y viceversa. Muy fácil de utilizar y permite ser versátil a la hora de crear los mensajes.

Xml.etree.cElementTree: Este módulo hace la misma función que el anterior pero en formato XML, es un poco más complejo de usar ya que la conversión XML a objetos de Python no es tan sencilla. Para ello utilizamos otro modulo llamado xmltodict.

Pika: Se utiliza para la publicación y consumo en colas de mensajeria. Se combina con el uso de RabbitMQ para el envío y recepción de mensajes.

Requests: Se utiliza para hacer peticiones HTTP, en nuestro caso la utilizaremos para hacer peticiones a la API REST del sistema.

Flask: Es una librería de Python que se utiliza para crear aplicaciones web sencillas. En nuestro caso será utilizada para la creación de la API que interactuará con la base de datos.

Schedule: Sirve para realizar tareas periódicamente como consultar información o enviarla.

GeoPy: Es un módulo que utiliza APIs de diferentes sistemas de geolocalización para transformar coordenadas en puntos reales, es decir, nombre de la calle, población, ciudad, etc. También emplea varios métodos para calcular la distancia entre dos puntos.

PyMySQL: Como su nombre indica, sirve para hacer peticiones SQL a nuestra base de datos.

Fuera ya de Python emplearemos RabbitMQ como gestor de colas de mensajería lo que nos permite una mayor escalabilidad en la difusión de la información.

5. Desarrollo de la solución

5.1. Estructura general de la solución

Procedemos a mostrar la estructura general de la solución de forma gráfica para facilitar la comprensión del proceso de desarrollo.

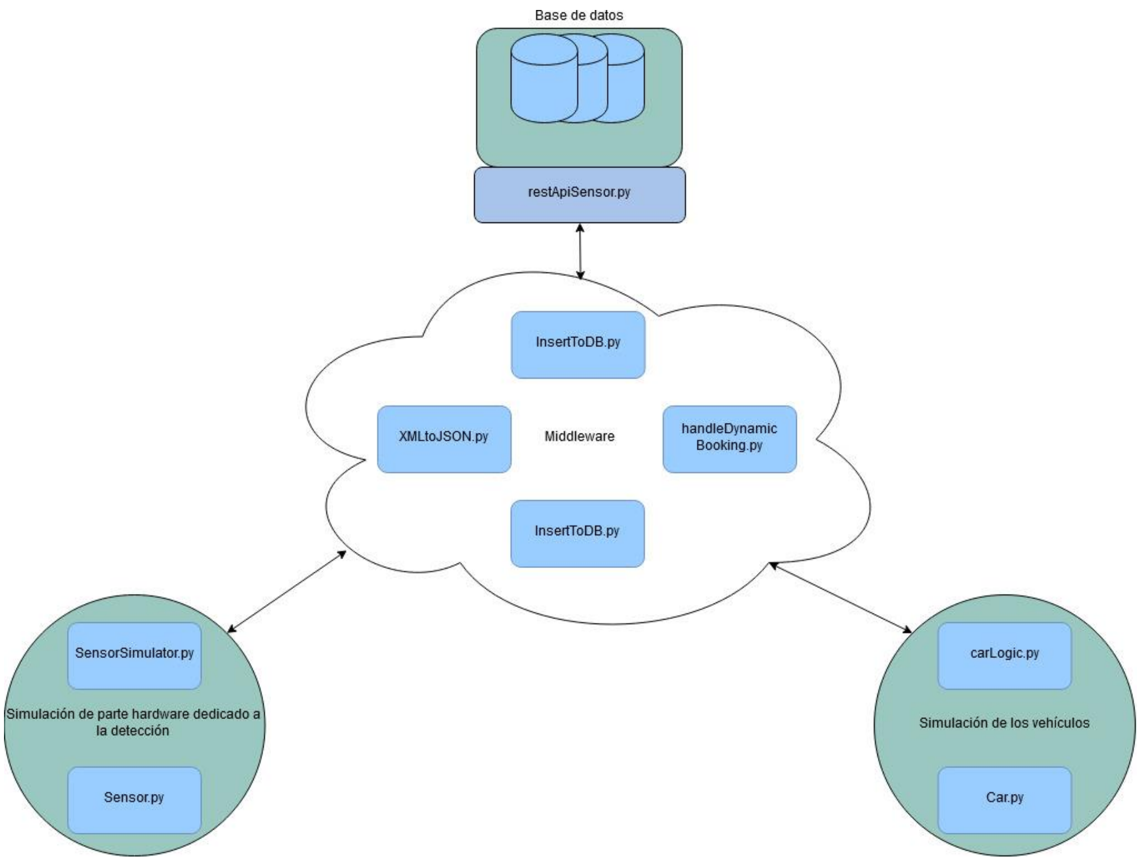


Ilustración 15. Estructura general de la solución

5.2. Proceso de desarrollo

Al iniciar el desarrollo, lo primero que pensamos fue en que parte debería de ser la base de nuestro proyecto y después de analizar detenidamente la situación decidimos comenzar por la parte de la infraestructura de sensores de aparcamiento. Teníamos claro que no iba a haber un único tipo de sensor así que pensamos en dos tipos, uno que enviara la información en JSON y otro en XML. Para facilitarlo creamos dos clases

que derivaran de la primera, sensorJSON y sensorXML. Cada uno de estos incluye un método que genera la información en el tipo de datos deseado.

```
#Clase que deriva de la inicial y genera mensaje en XML
class sensorXML(Sensor):
    pass

    def generateXML(self):

        root = ET.Element("sensor")
        ET.SubElement(root, "sensorID").text = self.sensorID
        ET.SubElement(root, "coords").text = self.coords
        ET.SubElement(root, "date").text = self.date
        ET.SubElement(root, "category").text = self.category
        ET.SubElement(root, "description").text = self.description
        ET.SubElement(root, "size").text = self.size
        ET.SubElement(root, "state").text = self.state
        ET.SubElement(root, "plate").text = self.plate

        return ET.tostring(root,encoding='unicode')
```

Ilustración 16. Ejemplo de generación de información en XML

Estos sensores no incluían la información acerca del tipo de vehículo, tamaño y descripción, se incluyó más tarde. El código identificador de los sensores en principio se generaba a través del módulo uuid de Python para generar uno totalmente aleatorio. Sin embargo, al final optamos por un valor aleatorio entre 0 y 1024 (al ser un prototipo a pequeña escala era más fácil de manejar que el obtenido por el módulo mencionado anteriormente).

Una vez creada la estructura de los objetos Sensor, es el momento de simular el funcionamiento de un sensor real. Para ello creamos el módulo SensorSimulator. Decidimos dividir nuestro esfuerzo en dos partes: el envío de información a la base de datos y la actualización y reenvío de esta. En ese instante instalamos el servidor de RabbitMQ para gestionar las conexiones entre este componente y la base de datos. A continuación, configuramos nuestro sistema mediante la librería Pika para que se conectara a este servidor. En un inicio, pusimos credenciales a este servidor, pero finalmente no las vimos útiles en nuestro proyecto. Si se hiciera una versión final en producción, sería adecuado añadir seguridad a estas conexiones. En una primera versión, se declararon dos colas de mensajería llamadas XML y JSON pero por motivos de diseño de aplicaciones basadas en soluciones de integración vimos que no era lo correcto y decidimos crear un *exchange* llamado "smartparking/place/<sensor-

ID>/status" del cual se crean dos colas de tipo *topic* llamadas json.message y xml.message. Esto se debe a que, dependiendo del tipo de mensaje, deben ser procesados de una forma u otra.

```
#Envia cada uno a un sitio. Tengo que automatizar esta tarea.
def sendInfo(s):
    if isinstance(s,sensorJSON):
        channel.basic_publish(exchange='smartparking/place/<sensor-ID>/status',routing_key='json.*',body=s.generateJSON())
        print("Enviado sensor JSON con id "+s.sensorID+"\n")
    if isinstance(s,sensorXML):
        channel.basic_publish(exchange='smartparking/place/<sensor-ID>/status',routing_key='xml.*',body=s.generateXML())
        print("Enviado sensor XML con id "+s.sensorID+"\n")
```

Ilustración 17. Publicación de mensajes en colas RabbitMQ

Desechamos la idea de utilizar Anypoint Studio como software de integración porque no nos proporcionaba la libertad que nos daba desarrollar toda la solución en Python, aunque nuestra opción fuera más compleja y costara un mayor tiempo.

Dentro de la simulación debemos crear instancias de las clases declaradas previamente. Nuestra opción es crear tres de cada tipo situados en las siguientes localizaciones situadas en la ciudad de Valencia: Plaza del Ayuntamiento, Paseo de la Alameda (a la altura del Oceanográfico), al lado del edificio del reloj del Puerto de Valencia, en el Bioparc (avenida Pío Baroja), cerca del centro comercial Nuevo Centro y al lado de la Universidad Politécnica de Valencia. De estas, la situada en el paseo de la Alameda está destinada a personas con algún tipo de discapacidad y la de Nuevo centro está destinada a vehículos grandes. En nuestra muestra de prueba todas las plazas están destinadas a coches y comienzan como libres. Estos sensores estaban organizados en listas diferentes dependiendo del tipo, pero finalmente se optó por una sola lista y dependiendo de la clase a la que pertenecen se envían a la cola correcta.

En relación a los mensajes JSON, la cola utilizada es consumida por un módulo llamado insertToDB.py. Este tiene la función de recibir los mensajes ya procesados para ser consumidos por el núcleo central e insertar o actualizar los datos dependiendo de su existencia previa. Se consulta si el sensor existe en la base de datos y si es positivo, se actualizan los datos del sensor, en caso contrario, se crea un nuevo sensor. Estas interacciones con la base de datos se realizan a través de una API REST creada por nosotros en Flask. La primera idea fue gastar la API solo para consultar información, pero decidimos dar mayor funcionalidad a esta para facilitar la accesibilidad y la integración.

```python
#Inserta los mensajes en la base de datos
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    message = dict(json.loads(body))
    msg = json.dumps(message)
    sensorID = message.get("sensorID")
    url="/".join(["http://localhost:5000/sensor",sensorID])

    if requests.get(url).json() != None:
        requests.put(url,json=msg)
    else:
        requests.post(url,json=msg)
```

Ilustración 18. Inserción de elementos en la base de datos a través de la API

La API cumple con las funciones necesarias del proyecto, aunque es fácilmente ampliable. Esta alojada en el puerto 5000 de la máquina donde se ejecute el proyecto. Respecto a los sensores, haciendo una petición GET a la ruta /sensor obtienes los datos de todos los sensores almacenados. Para actualizar, crear o consultar sensores en concreto, se deben realizar peticiones PUT, POST o GET a la ruta /sensor/<sensorID> sustituyendo <sensorID> por el identificador del sensor en concreto. Dentro de la lógica de estas peticiones se encuentran sentencias SQL que realizan estas acciones en la base de datos. Esta se encuentra en el puerto 3306. Más adelante se comentarán su cometido en las reservas.

```python
@app.route('/sensor/<sID>',methods=['GET'])
def getSensor(sID):
    try:
        conbd = pymysql.connect(host='localhost',user='root', password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.cursors.DictCursor)
        with conbd.cursor() as cursor:
                sql = "SELECT * FROM `sensors` WHERE `sensorID` = %s"
                cursor.execute(sql % sID)
                conbd.commit()
                response = jsonify(cursor.fetchone())
                cursor.close()
                conbd.close()
                return response
    except ConnectionError:
        print("Error de conexión")
```

Ilustración 19. Ejemplo de método GET en la API

Sin embargo, si se trata de un mensaje XML necesita un procesado previo para ser transformado al formato canónico de la solución. Con ese propósito, utilizamos el módulo XMLtoJSON.py. Dentro de nuestro diseño, esta clase se consideraría un procesador. Su objetivo es transformar todos los mensajes que le llegan en XML a una versión equivalente en JSON, igual a la de los mensajes enviados a la base de datos directamente. Finalmente, enviaría los mensajes a la misma cola que los anteriores.

```python
#Función que se ejecuta con cada mensaje recibido, procesa y reenvía
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    tempdict = dict(xmltodict.parse(body))

    sensorID = tempdict.get("sensor").get("sensorID")
    coords = tempdict.get("sensor").get("coords")
    date = tempdict.get("sensor").get("date")
    category = tempdict.get("sensor").get("category")
    description = tempdict.get("sensor").get("description")
    if description == None:
        description=""
    size = tempdict.get("sensor").get("size")
    state = tempdict.get("sensor").get("state")
    plate = tempdict.get("sensor").get("plate")
    if plate == None:
        plate=""
    tempdict.clear()

    tempdict["sensorID"] = sensorID
    tempdict["coords"] = coords
    tempdict["date"] = date
    tempdict["category"] = category
    tempdict["description"] = description
    tempdict["size"] = size
    tempdict["state"] = state
    tempdict["plate"] = plate

    message = json.dumps(tempdict)
```

Ilustración 20. Transformación de mensajes de XML a JSON

Con esta estructura hemos conseguido una vía de transportar los mensajes desde el origen hasta el destino. Pero nos surge dos preguntas que debemos resolver: ¿qué sucede si un vehículo ocupa una plaza? ¿Qué sucede si se reserva una plaza y el sensor no tiene constancia de ello?

Por este motivo, decidimos incluir un mecanismo que ejecutara de forma periódica cierta parte del código. Lo conseguimos gracias a la librería *Schedule*. Cada dos minutos pregunta el estado de los sensores y sí es diferente al actual lo actualiza. Independientemente de si varía el estado o no, actualiza la fecha de la información del sensor para notificar que es reciente. Al no utilizar sensores reales esta tarea se complicaba debido a que un sensor programado no se encuentra en el mundo real y no sufre cambios de estado. En un escenario real se debería de comprobar los

estados tanto del servidor como del sensor y ajustar el estado final dependiendo de prioridades (una plaza ocupada no puede ser reservada).

```
#Bucle donde se periodiza la función
schedule.every(2).minutes.do(periodicCalls)
while True:
    try:
        schedule.run_pending()
        time.sleep(1)
    except KeyboardInterrupt:
        print("Apagando sistema")
        break
```

Ilustración 21. Ejemplo de uso de la librería Schedule

Inmediatamente después de haber finalizado la primera parte del proyecto, tuvimos serias dudas de como encaminarlo. Después de un breve intento por crear una aplicación que permitiera gestionar a los usuarios sus reservas, decidimos apostar por una solución más enfocada a un entorno inteligente. Se decidió desarrollar un programa que simulara la entrada de un vehículo en la ciudad y como este realizaba una interacción con el ecosistema construido.

Inicialmente, lo fundamental es hablar de que propiedades tendrá un vehículo y en que formato las comunicará. Los vehículos pueden sufrir y en mayor medida el problema de la heterogeneidad: mayor cantidad de fabricantes pueden diferir en el software que utilicen sus creaciones y provocar una mayor diferencia a la hora de enviar mensajes. A pesar de ello, hemos asumido un tipo genérico de transporte que envía sus características en formato JSON como hemos descrito anteriormente. Estas características son el tipo de vehículo, la matricula, la posición del vehículo, descripción de necesidades especiales, tamaño y datos de pago. Hay una pequeña peculiaridad en la matrícula y es que al no poder empezar las cadenas de texto por números en Python hemos tenido que colocar las letras al principio seguidas de los dígitos de la matrícula.

En nuestro caso particular, las coordenadas se generan pseudoaleatoriamente. Es decir, hemos cogido el rango de coordenadas en el cual se sitúa la ciudad de Valencia (latitud 39.4 y longitud -0.3) y el resto de las cifras las hemos generado con la librería *random*.

```
@staticmethod
def generateCoords():
    x = 39.4 + (random.random() * 0.1)
    y = -0.3 - (random.random() * 0.1)
    return ",".join([str(x),str(y),])
```

Ilustración 22. Función de generado automático de coordenadas pseudoaleatorias

De igual manera que en el caso de los sensores, por un lado, hemos creado el objeto Vehículo y por otro hemos creado la lógica que debe ejecutar, de ésta se ello se encarga la clase carLogic.py, la cual sigue una secuencia de decisiones que llevarán a un usuario a un desenlace u otro. Se describe a continuación:

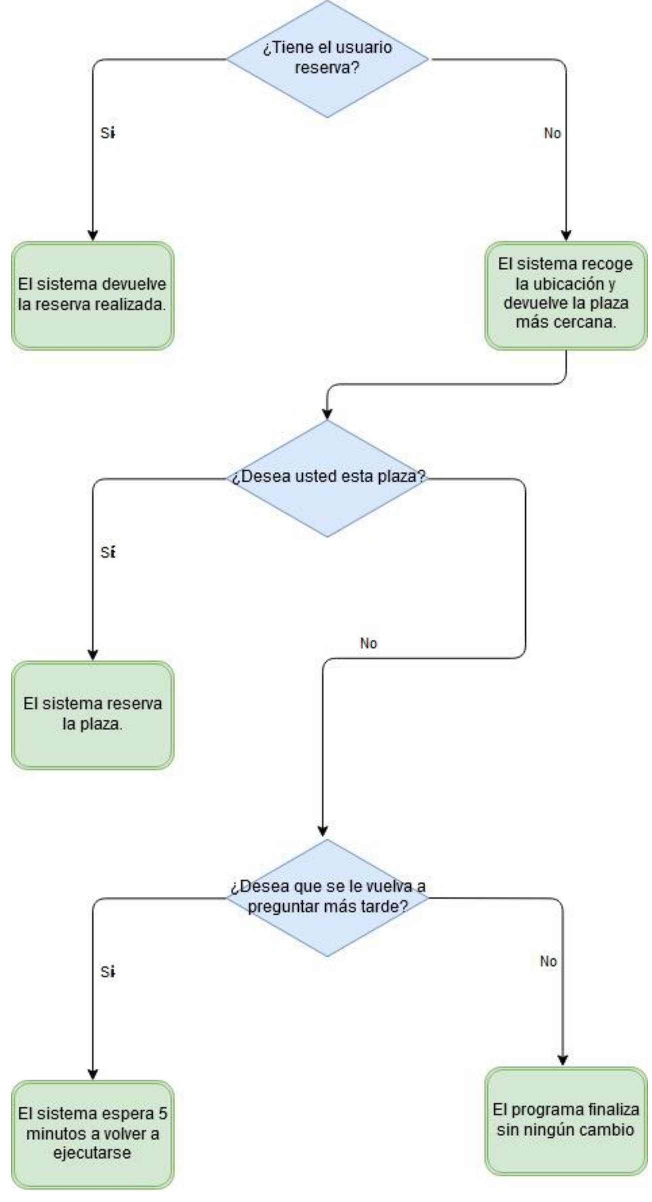


Ilustración 23. Flujo de ejecución de la lógica del vehículo.

En la primera decisión, se le pregunta al usuario si tiene una reserva hecha o no. En el caso de que sí, se hace una llamada a la API mediante el método GET a la ruta /reservas/{matrícula} sustituyendo por la del vehículo. El servidor le responde con los datos de la reserva y los muestra por pantalla. Antes de mostrarlo se traduce las coordenadas de la reserva por el nombre del lugar. Esto lo conseguimos a través de la librería GeoPy, concretamente haciendo una llamada a la API *open-source* Nominatim. Probamos a intentar utilizar la API proporcionada por Google, pero, a nuestro pesar, no la pudimos utilizar porque las peticiones no se efectuaban de forma correcta sin motivo aparente.

Si el usuario no posee reserva, envía su posición a un módulo auxiliar llamado handleDynamicBooking.py. Este se encarga de la búsqueda de la plaza más cercana al conductor. Con ese propósito, la librería GeoPy también nos ofrece utilidades para el cálculo. Dados dos puntos, cada uno con su par de coordenadas nos devuelve la distancia entre ellos. Aunque había varios métodos para ello basados en fórmulas diferentes, optamos por el que nos ofrecía la distancia en línea recta buscando la distancia mínima entre todos los sensores, obtenemos la plaza adecuada. En un caso real, se acotaría una distancia máxima de búsqueda inferior, en nuestro caso hemos utilizado 500 kilómetros por buscar un valor lo suficientemente alto para que no fallase.

```
spaces = requests.get("http://localhost:5000/sensor").json()
nearSens = ""
dist = 500.0
for s in spaces:
    if s.get("size") != carDetails.get("size"):
        continue
    if s.get("category") != carDetails.get("vehicle_type"):
        continue
    if s.get("description") != carDetails.get("description"):
        continue
    if s.get("state") != "free":
        continue
    x,y = str(s.get("coords")).rsplit(",")
    x = float(x)
    y = float(y)
    p2 = (x,y)
    dtmp = distance.distance(p1,p2).kilometers
    print(dtmp)
    if dtmp < dist:
        dist = dtmp
        nearSens = s.get("sensorID")
print(nearSens)
print(dist)
info = json.dumps(requests.get("/".join(["http://localhost:5000/sensor",nearSens])).json())
```

Ilustración 24. Fragmento de código donde se calcula la plaza más cercana

Al no realizar conexiones asíncronas que sería lo óptimo, hemos optado por una espera de 1 minuto para que el usuario espere las coordenadas de la plaza asignada. Al devolverla, el sistema le muestra la localización de esta en formato legible como hemos descrito anteriormente.

En caso de que el usuario desee esa plaza el sistema reserva esa plaza haciendo una llamada a la API con el método POST pasando los datos necesarios del vehículo y de la reserva. A estas reservas no se les pone fecha de fin ya que se calcularía después del uso para cobro del servicio. En caso contrario, se le pregunta si desea volver a ser preguntado y si no el programa finaliza. Si lo desea, la espera marcada es de cinco minutos.

Para concluir, la última pieza en nuestro proyecto es un pequeño script que sincroniza cada minuto la base de datos de sensores y reservas para que sean coherentes.

```python
#Se sincroniza las reservas con los sensores
def syncTask():
    reservas = requests.get(url)
    if reservas != None:
        reservas = reservas.json()
        for r in reservas:
            sID = r.get("sensorID")
            url2 = "/".join(["http://localhost:5000/sensor",sID,])
            sens = requests.get(url2).json()
            if sens.get("state") == "free":
                sens["state"] = "booked"
                sens["plate"] = r.get("plate")
                sens["date"] = r.get("date")
                requests.put(url2,json=json.dumps(sens))

schedule.every(1).minute.do(syncTask)
```

Ilustración 25. Función de sincronización entre sensores y base de datos

6. Pruebas

Para comprobar si nuestro sistema funciona, ejecutaremos el código en diferentes consolas y veremos cómo interactúan los diferentes componentes. Es importante que primero se enciendan los módulos relacionados con la simulación y manejo de los sensores. Esto es debido a que, para el correcto funcionamiento de los vehículos, debe haber información de los sensores en la base de datos.

El sistema parte con una base de datos vacía e iremos introduciendo datos en ella. En una primera fase encenderemos los siguientes módulos: SensorSimulator.py, XMLtoJSON.py, insertToBD.py y la API REST restApiSensors.py.

```
Enviado sensor JSON con id 740

Enviado sensor JSON con id 672

Enviado sensor JSON con id 608

Enviado sensor XML con id 849

Enviado sensor XML con id 255

Enviado sensor XML con id 582

Pulsa Ctrl+C para parar el sistema
```

Ilustración 26. Envío de información desde SensorSimulator.py.

```
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received b'<sensor><sensorID>849</sensorID><coords>39.479908,-0.407300</coords><date>2019-08-17 10:51:52.908187</date><category>car</category><description /><size>normal</size><state>free</state><plate /></sensor>'
{'sensorID': '849', 'coords': '39.479908,-0.407300', 'date': '2019-08-17 10:51:52.908187', 'category': 'car', 'description': '', 'size': 'normal', 'state': 'free', 'plate': ''}
 [x] Received b'<sensor><sensorID>255</sensorID><coords>39.480676,-0.392694</coords><date>2019-08-17 10:51:52.908187</date><category>car</category><description /><size>big</size><state>free</state><plate /></sensor>'
{'sensorID': '255', 'coords': '39.480676,-0.392694', 'date': '2019-08-17 10:51:52.908187', 'category': 'car', 'description': '', 'size': 'big', 'state': 'free', 'plate': ''}
 [x] Received b'<sensor><sensorID>582</sensorID><coords>39.481452,-0.350148</coords><date>2019-08-17 10:51:52.908187</date><category>car</category><description /><size>normal</size><state>free</state><plate /></sensor>'
{'sensorID': '582', 'coords': '39.481452,-0.350148', 'date': '2019-08-17 10:51:52.908187', 'category': 'car', 'description': '', 'size': 'normal', 'state': 'free', 'plate': ''}
```

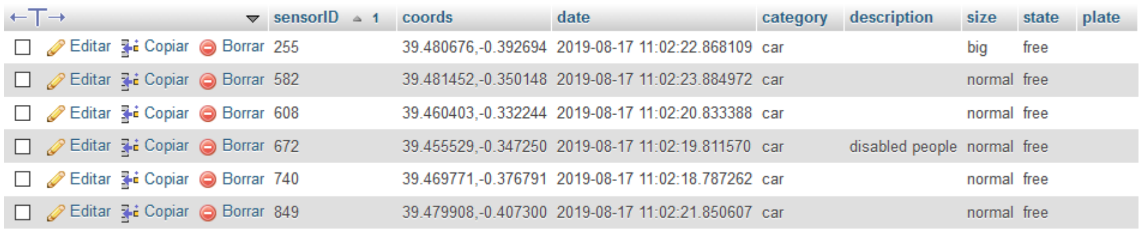
Ilustración 27. Recepción de datos en XML y transformación a JSON.

```
 [x] Received b'{"sensorID": "740", "coords": "39.469771,-0.376791", "date": "2019-08-17 10:51:52.908187", "category": "car", "description": "", "size": "normal", "state": "free", "plate": ""}'
 [x] Received b'{"sensorID": "672", "coords": "39.455529,-0.347250", "date": "2019-08-17 10:51:52.908187", "category": "car", "description": "disabled people", "size": "normal", "state": "free", "plate": ""}'
 [x] Received b'{"sensorID": "608", "coords": "39.460403,-0.332244", "date": "2019-08-17 10:51:52.908187", "category": "car", "description": "", "size": "normal", "state": "free", "plate": ""}'
 [x] Received b'{"sensorID": "849", "coords": "39.479908,-0.407300", "date": "2019-08-17 10:51:52.908187", "category": "car", "description": "", "size": "normal", "state": "free", "plate": ""}'
 [x] Received b'{"sensorID": "255", "coords": "39.480676,-0.392694", "date": "2019-08-17 10:51:52.908187", "category": "car", "description": "", "size": "big", "state": "free", "plate": ""}'
 [x] Received b'{"sensorID": "582", "coords": "39.481452,-0.350148", "date": "2019-08-17 10:51:52.908187", "category": "car", "description": "", "size": "normal", "state": "free", "plate": ""}'
```

Ilustración 28. Mensajes listos para insertar en la base de datos.

```
127.0.0.1 - - [17/Aug/2019 10:51:53] "GET /sensor/740 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:51:54] "POST /sensor/740 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:51:55] "GET /sensor/672 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:51:57] "POST /sensor/672 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:51:58] "GET /sensor/608 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:51:59] "POST /sensor/608 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:52:00] "GET /sensor/849 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:52:01] "POST /sensor/849 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:52:02] "GET /sensor/255 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:52:03] "POST /sensor/255 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:52:04] "GET /sensor/582 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:52:05] "POST /sensor/582 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:53] "GET /sensor/740 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:55] "GET /sensor/740 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:55] "GET /sensor/672 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:56] "GET /sensor/608 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:56] "PUT /sensor/740 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:57] "GET /sensor/849 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:57] "GET /sensor/672 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:58] "GET /sensor/255 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:58] "PUT /sensor/672 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:59] "GET /sensor/608 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:53:59] "GET /sensor/582 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:00] "PUT /sensor/608 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:01] "GET /sensor/849 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:02] "PUT /sensor/849 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:03] "GET /sensor/255 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:04] "PUT /sensor/255 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:05] "GET /sensor/582 HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2019 10:54:06] "PUT /sensor/582 HTTP/1.1" 200 -
```

Ilustración 29. Diferencia entre cuando se crean los sensores (peticiones POST) a cuando se actualizan (peticiones PUT).



←T→	sensorID	coords	date	category	description	size	state	plate
Editar Copiar Borrar	255	39.480676,-0.392694	2019-08-17 11:02:22.868109	car		big	free	
Editar Copiar Borrar	582	39.481452,-0.350148	2019-08-17 11:02:23.884972	car		normal	free	
Editar Copiar Borrar	608	39.460403,-0.332244	2019-08-17 11:02:20.833388	car		normal	free	
Editar Copiar Borrar	672	39.455529,-0.347250	2019-08-17 11:02:19.811570	car	disabled people	normal	free	
Editar Copiar Borrar	740	39.469771,-0.376791	2019-08-17 11:02:18.787262	car		normal	free	
Editar Copiar Borrar	849	39.479908,-0.407300	2019-08-17 11:02:21.850607	car		normal	free	

Ilustración 30. Vista de la base de datos.

Se han utilizado capturas de pantalla para demostrar que el funcionamiento del proyecto es correcto y ofrece aquello descrito a lo largo del escrito. Por otro lado, para probar el funcionamiento del software que gastaría el usuario, procedemos a recrear los diferentes hilos de ejecución que se describen en la ilustración 15. Encenderemos, además de los componentes ya en funcionamiento, los módulos carLogic.py, handleDynamicBooking.py y syncSens.py. Para el caso de que el conductor ya posea de una reserva, crearemos una reserva temporal sobre la plaza 582 por un coche con matrícula 3212bhr (debido a que las cadenas de texto en Python no pueden empezar por números en la base de datos aparecerá como bhr3212).

←T→	plate	date	sensorID	coords	startTime	finishTime	payment
Editar Copiar Borrar	bhr3212	2019-08-17 11:26:25.777626	582	39.481452,-0.350148	11:26:25.777626	12:26:25.777626	Yes

Ilustración 31. Reserva de vehículo

```
¿Has realizado una reserva?
Si
Detalles de la reserva:
 Vehículo: bhr3212
Localización: Universitat Politècnica, Avinguda dels Tarongers, Ciutat Jardí, Algirós, València, Comarca de València, València / Valencia, Comunitat Valenciana, 46020, España
 Hora inicio: 11:26:25.777626
 Hora final: 12:26:25.777626
```

Ilustración 32. Ejemplo de uso si el usuario tiene reserva.

sensorID ▲ 1	coords	date	category	description	size	state	plate
255	39.480676,-0.392694	2019-08-17 11:40:13.995358	car		big	free	
582	39.481452,-0.350148	2019-08-17 11:26:25.777626	car		normal	booked	bhr3212

Ilustración 33. Cambio de estado del sensor.

Ahora pasamos al segundo caso que es si el usuario no tiene reserva. Para ello borramos la reserva hecha y reestablecemos el estado de la base de datos.

```
¿Has realizado una reserva?
No
¿Desea usted esta plaza?
Localización: Plaça de l'Ajuntament, Carrer de Santa Irene, Ciutat Vella, València, Comarca de València, València / Valencia, Comunitat Valenciana, 46002, España
Si
Plaza reservada
```

Ilustración 34. Ejemplo de uso si el usuario no tiene reserva.

plate	date	sensorID	coords	startTime	finishTime	payment
bhr3212	2019-08-17 11:49:01.530112	740	39.469771,-0.376791	11:49:01.530112		No

Ilustración 35. Reserva realizada a través del método dinámico.

La última prueba por realizar para corroborar el correcto funcionamiento es comprobar que, si el usuario no desea la primera plaza, a los cinco minutos se genere otra diferente.

```
¿Has realizado una reserva?
No
¿Desea usted esta plaza?
Localización: Universitat Politècnica, Avinguda dels Tarongers, Ciutat Jardí, Algirós, València, Comarca de València, València / Valencia, Comunitat Valenciana, 46020, España
No
¿Quiere que le volvamos a preguntar?
Si
Le preguntaremos en 5 minutos
¿Desea usted esta plaza?
Localización: Avinguda de Pio Baroja, Sant Pau, Campanar, València, Comarca de València, València / Valencia, Comunitat Valenciana, 46920, España
Si
Plaza reservada
```

Ilustración 36. Ejemplo de uso si el usuario desea otra plaza.

7. Conclusiones

En el capítulo uno se han propuesto tres objetivos que debían ser cumplidos al final del proceso. En el caso de diseñar una solución de aparcamiento inteligente, ha sido abordado en el capítulo cuatro de una satisfactoria justificando todas las decisiones tomadas en nuestro diseño. Por ejemplo, en temas de escalabilidad y adaptabilidad si decidiéramos añadir un formato más de envío de datos desde los sensores, solo deberíamos abordar la integración de ese tipo de sensores en el sistema y no modificar el resto de los componentes. Lo mismo ocurriría con otros tipos de vehículos u otras fuentes de información.

El segundo objetivo ha sido abordado en los capítulos cinco y seis. Hemos cumplido el objetivo de forma parcial debido a que una solución IoT en una ciudad es un trabajo de meses ya que hay muchos agentes que tener en cuenta y no se disponía del tiempo y recursos suficientes. Sin embargo, nuestro trabajo sirve como base a otros trabajos posteriores que enfoquen su esfuerzo en los elementos hardware o en la gestión del sistema a mejorarlo y completar las múltiples vías abiertas.

El tercer objetivo no ha sido parte de nuestro trabajo como tal, pero en el capítulo dos al estudiar los diferentes sistemas habidos hasta el momento se ha hablado de los beneficios para las ciudades tales como menor volumen de vehículos, menores emisiones de CO2, menor tiempo para la búsqueda de aparcamiento por parte del usuario y otros beneficios.

7.1. Relación del trabajo desarrollado con los estudios cursados

Como estudiantes de informática, nuestro proceso de aprendizaje no se centra solo en un ámbito concreto, como podría ser la programación, sino que también se obtienen nociones de redes, hardware, software y otros campos. Nuestro trabajo parte de los conocimientos obtenidos en la asignatura de Integración de Aplicaciones y su enfoque a la hora de resolver escenarios con fuentes u aplicaciones que tratan los datos de

formas distintas. Otra de las materias que ha ayudado al desarrollo ha sido Tecnologías y Servicios en Red en cuestiones como la gestión de colas de mensajería. Por último, los cursos de programación y de bases de datos recibidos han ayudado a la hora de tener un pensamiento lógico.

Respecto a las competencias transversales, hemos desarrollado las siguientes:

- Aplicación y pensamiento práctico
- Análisis y resolución de problemas
- Diseño y proyecto
- Responsabilidad ética, medioambiental y profesional
- Pensamiento crítico
- Conocimiento de problemas contemporáneos

8. Referencias

[1] Wikipedia. (8 de Abril de 2019). *Wikipedia*. Obtenido de Wikipedia, La Enciclopedia Libre: https://es.wikipedia.org/wiki/Ciudad_inteligente

[2] Bouskela, M., Casseb, M., Bassi, S., & De Luca, C. (2016). *La ruta hacia las smart cities: Migrando de una gestión tradicional a la ciudad inteligente.* Inter-American Development Bank.

[3] Nuñez, G. (15 de Abril de 2014). *Nielsen.* Obtenido de Nielsen website: https://www.nielsen.com/es/es/insights/news/2014/el-41-de-los-espanoles-que-va-en-coche-al-trabajo-tarda-mas-de-una-hora-segun-nielsen.html

[4] Dalmau, J. (2017). En España hay 30 millones de vehículos en circulación | Noticias Coches.net. Retrieved June 22, 2019, from https://www.coches.net/noticias/parque-de-vehiculos-en-espana

[5] Sethi, P., & Sarangi, S. R. (2017). Internet Of Things: Architecture,Issues and Applications. *International Journal of Engineering Research and Applications*, *07*(06), 85–88. https://doi.org/10.9790/9622-0706048588

[6] M.Y.I.Idris, Y.Y.Leng, E.M. Tamil, N. M. N. and Z. R. (2009). Car Park System: A Review of Smart Parking System and its Technology. Information Technology Journal, 8(2), 101–113.

[7] Hainalkar, G. N., & Vanjale, M. S. (2018). Smart parking system with pre & post reservation, billing and traffic app. *Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems, ICICCS 2017*, *2018-Janua*, 500–505. https://doi.org/10.1109/ICCONS.2017.8250772

[8] Khanna, A., & Anand, R. (2016). IoT based smart parking system. *2016 International Conference on Internet of Things and Applications (IOTA)*, 266–270. https://doi.org/10.1109/IOTA.2016.7562735

[9] Hodel, T. B., & Cong, S. (2016). Parking Space Optimization Services , a uniformed Web Application Architecture. *University of Zurich, Department of Information Technology, Database Technology Research Group Winterthurerstr*, *190*(May), 1–12.

[10] Kotb, A. O., Shen, Y. C., & Huang, Y. (2017). Smart Parking Guidance, Monitoring and Reservations: A Review. *IEEE Intelligent Transportation Systems Magazine*, *9*(2), 6–16. https://doi.org/10.1109/MITS.2017.2666586

[11] Šilar, J., Růžička, J., Bělinová, Z., Langr, M., & Hlubučková, K. (2018). Smart parking in the smart city application. *2018 Smart Cities Symposium Prague, SCSP 2018*, 1–5. https://doi.org/10.1109/SCSP.2018.8402667

[12] Mathijssen, A., & Pretorius, H. (2005). *Specification , Analysis and Verification of an Automated Parking Garage*. (June), 1–12.

[13] Shao, C., Yang, H., Zhang, Y., & Ke, J. (2016). A simple reservation and allocation model of shared parking lots. *Transportation Research Part C: Emerging Technologies*, *71*, 303–312. https://doi.org/10.1016/j.trc.2016.08.010

[14] Geng, Y., & Cassandras, C. G. (2012). A new "Smart Parking" System Infrastructure and Implementation. *Procedia - Social and Behavioral Sciences*, *54*, 1278–1287. https://doi.org/10.1016/j.sbspro.2012.09.842

[15] Ji, Z., Ganchev, I., O'Droma, M., Zhao, L., & Zhang, X. (2014). A cloud-based car parking middleware for IoT-based smart cities: Design and implementation. *Sensors (Switzerland)*, *14*(12), 22372–22393. https://doi.org/10.3390/s141222372

Anexo

Este anexo sirve para completar el apartado 5 adjuntando el código fuente del sistema organizado por módulos facilitando al lector la comprensión del proyecto.

Sensor.py

```
import random,json,datetime
import xml.etree.ElementTree as ET

#Clase base Sensor
class Sensor:
    def
__init__(self,sensorID,coords,date,category,description,size,state,plate)
:
        self.sensorID=sensorID
        self.coords=coords
        self.date=date
        self.category=category
        self.description=description
        self.size=size
        self.state=state
        self.plate=plate




#Clase que deriva de la inicial y genera mensaje en JSON
class sensorJSON(Sensor):
    pass
    def generateJSON(self):
        msg =
{"sensorID":self.sensorID,"coords":self.coords,"date":self.date,"category
":self.category,"description":self.description,"size":self.size,"state":s
elf.state,"plate":self.plate,}

        return json.dumps(msg)

#Clase que deriva de la inicial y genera mensaje en XML
class sensorXML(Sensor):
    pass

    def generateXML(self):

        root = ET.Element("sensor")
```

```
        ET.SubElement(root, "sensorID").text = self.sensorID
        ET.SubElement(root, "coords").text = self.coords
        ET.SubElement(root, "date").text = self.date
        ET.SubElement(root, "category").text = self.category
        ET.SubElement(root, "description").text = self.description
        ET.SubElement(root, "size").text = self.size
        ET.SubElement(root, "state").text = self.state
        ET.SubElement(root, "plate").text = self.plate

        return ET.tostring(root,encoding='unicode')

#Genera ID aleatorio
def generateID():
        return str(random.randint(0,1024))

#Genera Fecha
def generateDate():
    return str(datetime.datetime.today())
```

SensorSimulator.py

```
from Sensor import *
import pika,json,requests
import xml.etree.cElementTree as ET
import schedule,time

#Para coger en caso de necesidad información de la api
def getInfo():
    url="http://localhost:5000/sensor"
    response = requests.api.get(url)
    info = json.loads(response.json)
    return info



#Creación de la conexión

connection =
pika.BlockingConnection(pika.ConnectionParameters("localhost"))
channel = connection.channel()

channel.exchange_declare("smartparking/place/<sensor-
ID>/status",exchange_type="topic")
channel.queue_declare(queue='xml.message')
channel.queue_bind("xml.message","smartparking/place/<sensor-
ID>/status",routing_key="xml.*")
channel.queue_declare(queue='json.message')
```

```
channel.queue_bind("json.message","smartparking/place/<sensor-
ID>/status",routing_key="json.*")

#Creación de los sensores
sensor1 = sensorJSON(generateID(),"39.469771,-
0.376791",generateDate(),"car","","normal","free","")
sensor2 = sensorJSON(generateID(),"39.455529,-
0.347250",generateDate(),"car","disabled people","normal","free","")
sensor3 = sensorJSON(generateID(),"39.460403,-
0.332244",generateDate(),"car","","normal","free","")
sensor4 = sensorXML(generateID(),"39.479908,-
0.407300",generateDate(),"car","","normal","free","")
sensor5 = sensorXML(generateID(),"39.480676,-
0.392694",generateDate(),"car","","big","free","")
sensor6 = sensorXML(generateID(),"39.481452,-
0.350148",generateDate(),"car","","normal","free","")

#Agrupacion de sensores

sensores = [sensor1,sensor2,sensor3,sensor4,sensor5,sensor6]

#Envia cada uno a un sitio. Tengo que automatizar esta tarea.
def sendInfo(s):
    if isinstance(s,sensorJSON):
        channel.basic_publish(exchange='smartparking/place/<sensor-
ID>/status',routing_key='json.*',body=s.generateJSON())
        print("Enviado sensor JSON con id "+s.sensorID+"\n")
    if isinstance(s,sensorXML):
        channel.basic_publish(exchange='smartparking/place/<sensor-
ID>/status',routing_key='xml.*',body=s.generateXML())
        print("Enviado sensor XML con id "+s.sensorID+"\n")

#Función para hacer tareas periodicas

def periodicCalls():
    for s in sensores:
        response =
requests.get('/'.join(['http://localhost:5000/sensor',s.sensorID,])).json
()
        if response != None:

            tmpo = response
            if tmpo.get("state") != s.state:
                d1=datetime.datetime.strptime(s.date, '%Y-%m-%d
%H:%M:%S.%f')
                d2=datetime.datetime.strptime(tmpo.get("date"), '%Y-%m-%d
%H:%M:%S.%f')
                if d1 < d2:
                    s.date = tmpo.get("date")
```

```
                    s.state = tmpo.get("state")
                    s.plate = tmpo.get("plate")
                    sendInfo(s)
                else:
                    s.date = generateDate()
                    sendInfo(s)
            else:
                s.date = generateDate()
                sendInfo(s)

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

#Info inicial
for s in sensores:
    sendInfo(s)
print("Pulsa Ctrl+C para parar el sistema")


#Bucle donde se periodiza la función
schedule.every(2).minutes.do(periodicCalls)
while True:
    try:
        schedule.run_pending()
        time.sleep(1)
    except KeyboardInterrupt:
        print("Apagando sistema")
        break
```

XMLtoJSON.py

```
import pika,json
import xmltodict

#Declaración de la conexión y la cola
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='xml.message')


#Función que se ejecuta con cada mensaje recibido, procesa y reenvía
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    tempdict = dict(xmltodict.parse(body))

    sensorID = tempdict.get("sensor").get("sensorID")
    coords = tempdict.get("sensor").get("coords")
    date = tempdict.get("sensor").get("date")
```

```
    category = tempdict.get("sensor").get("category")
    description = tempdict.get("sensor").get("description")
    if description == None:
        description=""
    size = tempdict.get("sensor").get("size")
    state = tempdict.get("sensor").get("state")
    plate = tempdict.get("sensor").get("plate")
    if plate == None:
        plate=""
    tempdict.clear()

    tempdict["sensorID"] = sensorID
    tempdict["coords"] = coords
    tempdict["date"] = date
    tempdict["category"] = category
    tempdict["description"] = description
    tempdict["size"] = size
    tempdict["state"] = state
    tempdict["plate"] = plate

    message = json.dumps(tempdict)
    print(tempdict)
    channel.basic_publish(exchange="smartparking/place/<sensor-
ID>/status",routing_key="json.*",body=message)

channel.basic_consume(queue='xml.message', on_message_callback=callback,
auto_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
try:
    channel.start_consuming()
except KeyboardInterrupt:
    print("Apagando sistema")
    channel.close()
    connection.close()
```

syncSens.py

```
import requests,json,time,schedule

url = "http://localhost:5000/reservas"

#Se sincroniza las reservas con los sensores
def syncTask():
    reservas = requests.get(url)
    if reservas != None:
        reservas = reservas.json()
        for r in reservas:
```

```
            sID = r.get("sensorID")
            url2 = "/".join(["http://localhost:5000/sensor",sID,])
            sens = requests.get(url2).json()
            if sens.get("state") == "free":
                sens["state"] = "booked"
                sens["plate"] = r.get("plate")
                sens["date"] = r.get("date")
                requests.put(url2,json=json.dumps(sens))

schedule.every(1).minute.do(syncTask)
while True:
    try:
        schedule.run_pending()
        time.sleep(1)
    except KeyboardInterrupt:
        print("Apagando sistema")
```

insertToDB.py

```
import pika,json,pymysql,requests



connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='json.message')

#Inserta los mensajes en la base de datos
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    message = dict(json.loads(body))
    msg = json.dumps(message)
    sensorID = message.get("sensorID")
    url="/".join(["http://localhost:5000/sensor",sensorID])

    if requests.get(url).json() != None:
        requests.put(url,json=msg)
    else:
        requests.post(url,json=msg)


channel.basic_consume(queue='json.message', on_message_callback=callback,
auto_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
try:
    channel.start_consuming()
except KeyboardInterrupt:
```

```
    print("Apagando sistema")
    channel.close()
    connection.close()
```

handleDynamicBooking.py

```
import requests,pika,json,time,geopy,schedule
from geopy import distance

connection =
pika.BlockingConnection(pika.ConnectionParameters("localhost"))
channel = connection.channel()
channel.exchange_declare("smartparking/booking/<plate>")
queue1 = channel.queue_declare("dynamicBooking")
channel.queue_declare("handleCar")
channel.queue_bind("handleCar","smartparking/booking/<plate>",routing_key
="handleCar")

#Cada 30 segundos se consume un mensaje de la cola y se obtiene la plaza
más cercana al vehículo recibido
def cronJob():
    method,properties,msg =
channel.basic_get("dynamicBooking",auto_ack=True)

    if msg != None:
        carDetails = json.loads(msg)
        x,y=carDetails.get("coords").rsplit(",",2)
        x = float(x)
        y = float(y)
        p1 = (x,y)

        spaces = requests.get("http://localhost:5000/sensor").json()
        nearSens = ""
        dist = 500.0
        for s in spaces:
            if s.get("size") != carDetails.get("size"):
                continue
            if s.get("category") != carDetails.get("vehicle_type"):
                continue
            if s.get("description") != carDetails.get("description"):
                continue
            if s.get("state") != "free":
                continue
            x,y = str(s.get("coords")).rsplit(",")
            x = float(x)
            y = float(y)
            p2 = (x,y)
            dtmp = distance.distance(p1,p2).kilometers
```

```
            print(dtmp)
            if dtmp < dist:
                dist = dtmp
                nearSens = s.get("sensorID")
        print(nearSens)
        print(dist)
        info =
json.dumps(requests.get("/".join(["http://localhost:5000/sensor",nearSens
])).json())


channel.basic_publish("smartparking/booking/<plate>",routing_key="handleC
ar",body=info)

schedule.every(30).seconds.do(cronJob)
while True:
    try:
        schedule.run_pending()
        time.sleep(1)
    except KeyboardInterrupt:
        print("Apagando sistema")
        break
```

Car.py

```
import random,json

class Vehicle:
    def
__init__(self,vehicle_type,plate,coords,size,description,payment_data):
        self.vehicle_type = vehicle_type
        self.plate = plate
        self.coords = coords
        self.description = description
        self.size = size
        self.payment_data = payment_data

    def generateJSON(self):
        msg =
{"vehicle_type":self.vehicle_type,"plate":self.plate,"coords":self.coords
,"description":self.description,"size":self.size,"payment_data":self.paym
ent_data}

        return json.dumps(msg)
    @staticmethod
    def generateCoords():
        x = 39.4 + (random.random() * 0.1)
        y = -0.3 - (random.random() * 0.1)
```

```
        return ",".join([str(x),str(y),])
```

carLogic.py

```
from Car import Vehicle
import requests,geopy,pika,time,datetime,json


nominatim = geopy.geocoders.Nominatim()
myVehicle =
Vehicle('car','bhr3212',Vehicle.generateCoords(),'normal','','No')
url="/".join(["http://localhost:5000/reservas",myVehicle.plate])

dec1 = input("¿Has realizado una reserva?\n")
if dec1 == "Sí" or dec1 == "Si" or dec1 == "si" or dec1 == "sí":
    #Chequeamos lista de reservas y recibimos la reserva deseada
    reserva = requests.get(url).json()
    if reserva == None:
        print("No se ha encontrado ninguna reserva con ese vehículo")
    else:
        #Obtenemos las coordenadas de la reserva y lo pasamos a un punto
        x,y = str(reserva.get("coords")).rsplit(",")
        x = float(x)
        y = float(y)
        p = geopy.Point(x,y)
        location = nominatim.reverse(p)
        print("Detalles de la reserva:\n Vehículo: {}\nLocalización: {}\n
Hora inicio: {}\n Hora final:
{}".format(reserva.get("plate"),location,reserva.get("startTime"),reserva
.get("finishTime")))
elif dec1 == "No" or dec1 =="no":
#Buscamos la ubicación más cercana y preguntamos si esa es la deseada
    while True:
        #Creamos la conexión y las colas
        connection =
pika.BlockingConnection(pika.ConnectionParameters("localhost"))
        channel = connection.channel()
        channel.exchange_declare("smartparking/booking/<plate>")
        channel.queue_declare("dynamicBooking")
        channel.queue_declare("handleCar")

channel.queue_bind("dynamicBooking","smartparking/booking/<plate>",routin
g_key="dynamicBooking")
        #Publicamos la información del vehiculo
```

```
channel.basic_publish("smartparking/booking/<plate>",routing_key="dynamic
Booking",body=myVehicle.generateJSON())
        #Esperamos respuesta
        time.sleep(60)
        #Obtenemos la respuesta
        method,params,plzcerca =
channel.basic_get("handleCar",auto_ack=True)
        plzcerca = json.loads(plzcerca)
        x,y = str(plzcerca.get("coords")).rsplit(",")
        x = float(x)
        y = float(y)
        p = geopy.Point(x,y)
        location = nominatim.reverse(p)
        dec2 = input("¿Desea usted esta plaza?\nLocalización:
{}\n".format(location))
        if dec2 == "Sí" or dec2 == "Si" or dec2 == "si" or dec2 == "sí":
            #Se adjudica la plaza
            data = {"plate": myVehicle.plate,
"date":str(datetime.datetime.now()),"sensorID":plzcerca.get("sensorID"),
"coords": plzcerca.get("coords"), "startTime":
str(datetime.datetime.now().time()), "finishTime": "", "payment":
myVehicle.payment_data}
            requests.post(url,json=json.dumps(data))
            #Añadiria las coordenadas al mapa
            print("Plaza reservada")
            channel.close()
            connection.close()
            break
        elif dec2 == "No" or dec2 =="no":
            #Se espera un tiempo y se repite el proceso
            dec3=input("¿Quiere que le volvamos a preguntar?\n")
            if dec3 == "Sí" or dec3 == "Si" or dec3 == "si" or dec3 ==
"sí":
                channel.close()
                connection.close()
                myVehicle.coords = Vehicle.generateCoords()
                print("Le preguntaremos en 5 minutos")
                time.sleep(300)
            elif dec3 == "No" or dec3 =="no":
                print("Hasta pronto")
                channel.close()
                connection.close()
                break
    else:
        print("Respuesta no contemplada, por favor conteste Sí o No")
else:
    print("Respuesta no contemplada, por favor conteste Sí o No")
```

restApiSensors.py

```
import pymysql,json
from flask import Flask,request,Response,jsonify

app = Flask(__name__)

#Métodos de la API REST
@app.route('/sensor', methods=['GET'])
def getSensors():
    try:
        conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
        with conbd.cursor() as cursor:
                sql = "SELECT * FROM `sensors`"
                cursor.execute(sql)
                conbd.commit()
                response = jsonify(cursor.fetchall())
                cursor.close()
                conbd.close()
                return response
    except ConnectionError:
        print("Error de conexión")

@app.route('/sensor/<sID>',methods=['GET'])
def getSensor(sID):
    try:
        conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
        with conbd.cursor() as cursor:
                sql = "SELECT * FROM `sensors` WHERE `sensorID` = %s"
                cursor.execute(sql % sID)
                conbd.commit()
                response = jsonify(cursor.fetchone())
                cursor.close()
                conbd.close()
                return response
    except ConnectionError:
        print("Error de conexión")
```

```
@app.route('/sensor/<sID>',methods=['PUT'])
def updateSensor(sID):
    try:
        conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
        with conbd.cursor() as cursor:
                sql = "UPDATE sensors SET date=%s, state=%s, plate=%s
WHERE sensorID = %s"
                queryparams = json.loads(request.json)
                cursor.execute(sql,
(queryparams.get("date"),queryparams.get("state"),queryparams.get("plate"
),sID))
                conbd.commit()
                cursor.close()
                conbd.close()
                return "Elemento actualizado"

    except ConnectionError:
        print("Error de conexión")

@app.route('/sensor/<sID>',methods=['POST'])
def createSensor(sID):
    try:
        conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
        with conbd.cursor() as cursor:
                sql = "INSERT INTO `sensors` (`sensorID`, `coords`,
`date`,`category`,`description`,`size`, `state`, `plate`) VALUES (%s, %s,
%s, %s, %s, %s, %s, %s)"
                queryparams = json.loads(request.json)
                cursor.execute(sql,
(queryparams.get("sensorID"),queryparams.get("coords"),queryparams.get("d
ate"),
queryparams.get("category"),queryparams.get("description"),queryparams.ge
t("size"),queryparams.get("state"),queryparams.get("plate")))
                conbd.commit()
                cursor.close()
                conbd.close()
                return "Elemento creado"
    except ConnectionError:
        print("Error de conexión")

@app.route('/reservas/<platenumb>', methods=['GET'])
def getReserva(platenumb):
        try:
```

```
                conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
                with conbd.cursor() as cursor:
                        sql = "SELECT * FROM `reservas` WHERE
`reservas`.`plate` = %s"
                        cursor.execute(sql, (platenumb,))
                        conbd.commit()
                        response = jsonify(cursor.fetchone())
                        cursor.close()
                        conbd.close()
                        return response
        except ConnectionError:
                print("Error de conexión")

@app.route('/reservas/<plate>', methods=['POST'])
def createReservas(plate):
        try:
                conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
                with conbd.cursor() as cursor:
                        sql = "INSERT INTO `reservas` (`plate`, `date`,
`sensorID`,`coords`,`startTime`,`finishTime`, `payment`) VALUES (%s, %s,
%s, %s, %s, %s, %s)"
                        queryparams = json.loads(request.json)
                        cursor.execute(sql,
(queryparams.get("plate"),queryparams.get("date"),queryparams.get("sensor
ID"),
queryparams.get("coords"),queryparams.get("startTime"),queryparams.get("f
inishTime"),queryparams.get("payment")))
                        conbd.commit()
                        cursor.close()
                        conbd.close()
                        return "Elemento creado"
        except ConnectionError:
                print("Error de conexión")

@app.route('/reservas', methods=['GET'])
def getReservas():
    try:
        conbd = pymysql.connect(host='localhost',user='root',
password='',db='smart_parking',charset='utf8mb4',cursorclass=pymysql.curs
ors.DictCursor)
        with conbd.cursor() as cursor:
                sql = "SELECT * FROM `reservas`"
                cursor.execute(sql)
                conbd.commit()
                response = jsonify(cursor.fetchall())
```

```
                cursor.close()
                conbd.close()
                return response
    except ConnectionError:
        print("Error de conexión")

if __name__ == '__main__':
    app.run()
```