UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño de interacciones IoT geolocalizadas en el ámbito de una ciudad inteligente

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Sergi Sanz Carreres

Tutor: Joan Fons Cors

Vicente Pelechano Ferragud

Curso: 2018 - 2019

Resumen

En la actualidad la proliferación de recursos móviles ha originado que cada vez más dispositivos se encuentren conectados a internet, formando así una red conocida como IoT.

El objetivo principal del proyecto consiste en identificar los problemas más reseñables que se producen al interpretar las posiciones GPS proporcionadas por los recursos IoT y proporcionar una solución de diseño e implementación a estos problemas.

Otro objetivo importante se centra en definir unas reglas de interacción geolocalizadas cuyo fin sea el de definir unas reglas para resolver los problemas que se producen cuando varios recursos IoT se encuentran en una localización aproximada y debe producirse una interacción entre ellos.

Para poder alcanzar estos objetivos será necesario diseñar algunas soluciones siendo una estas la 'gestión/control' de localizaciones con el objetivo de comparar ubicaciones entre recursos (en diversas situaciones que pueden ser: cerca/lejos, acercándose/alejándose, dentro/fuera) y solicitar acciones sobre éstos.

La otra solución elegida es la de diseñar una herramienta la cual haciendo uso de una fuente de posiciones GPS, realice un conjunto de cambios con el fin de posibilitar la utilización de los recursos, que componen la fuente de posiciones GPS en otra fuente de posiciones GPS distinta.

Una vez diseñadas estas soluciones se desarrollarán ejemplos concretos utilizando los recursos proporcionados de la plataforma/laboratorio de la *Smart City* del grupo de investigación TaTami del centro PROS.

Para desarrollar estos ejemplos haremos uso del lenguaje de programación Java, también haremos uso de los servicios REST y nos apoyaremos en el uso del bróker Mosquitto, debido a que nos permite implementar un protocolo asíncrono de mensajería MQTT y definir colas de mensajes haciendo uso del patrón publicación/suscripción.

Palabras clave: IoT, Internet de las cosas, TaTami, Owntracks, interacción geolocalizada.

Resum

En l'actualitat la proliferació de recursos mòbils ha originat que cada vegada més dispositius es troben connectats a internet, formant així una xarxa coneguda com Iot.

L'objectiu principal del projecte consisteix en identificar els problemes més ressenyables que es produeixen a l'interpretar les posicions GPS proporcionades pels recursos IOT i proporcionar una solució de disseny e implementació a aquests problemes.

Un altre objectiu important se centra en definir unes regles d'interacció geolocalitzades la fi es el de definir unes regles per resoldre els problemes que es produeixen quan diversos recursos IoT es troben en una localització aproximada i s'ha de produir una interacció entre ells.

Per poder assolir aquests objectius serà necessari dissenyar algunes solucions sent una d'aquestes la 'gestió / control' de localitzacions amb l'objectiu de comparar ubicacions entre recursos (en diverses situacions que poden ser: prop / lluny, apropant-se / allunyant-se, dins / fora) i sol·licitar accions sobre aquests.

L'altra solució escollida és la de dissenyar una eina la qual fent ús d'una font de posicions GPS, faci un conjunt de canvis per tal de possibilitar la utilització dels recursos, que componen la font de posicions GPS en una altra font de posicions GPS diferent.

Una vegada dissenyades aquestes solucions es desenvoluparan exemples concrets utilitzant els recursos proporcionats per la plataforma / laboratori de la Smart City del grup de investigació tatami del centre PROS.

Per desenvolupar aquests exemples farem ús del llenguatge de programació Java, també farem ús dels serveis REST i ens recolzarem en l'ús del broker Mosquitto, pel fet que ens permet implementar un protocol asíncron de missatgeria MQTT i definir cues de missatges fent ús del patró publicació / subscripció.

Palabras clave: IoT, Internet de les coses, TaTami, Owntracks, interacció geolocalizada.

Abstract

Presently, the rapid increase in the number that mobile resources has caused, there are more and more devices that are connected to the Internet, thus forming a network known as IoT.

The main objective of the project is to identify the most notable problems that occur when interpreting the GPS positions provided by IoT resources and provide a design and implementation solution to these problems.

Another important objective is to define geolocalized interaction rules whose purpose is to define rules to solve the problems that occur when several IoT resources are in an approximate location and an interaction between them must occur.

In order to achieve these objectives, it will be necessary to design some solutions, being these the 'management / control' of locations with the objective of comparing locations between resources (in various situations that may be: near / far, approaching / moving away, in / out) and request actions on these.

The other solution chosen is to design a tool which, using a source of GPS positions, makes a set of changes in order to enable the use of resources, which make up the source of GPS positions in another source of GPS positions different.

Once these solutions have been designed, concrete examples will be developed using the resources provided by the Smart City platform / laboratory of the TaTami research group of the PROS center.

In order to develop these examples, we will use the Java programming language, we will also use the REST services and we will rely on the use of the Mosquitto broker, because it allows us to implement an asynchronous MQTT messaging protocol and define message queues using the pattern publication / subscription.

Keywords: IoT, Internet of things, TaTami, Owntracks, geolocation interaction

Tabla de contenidos

Índice de imágenes

Índice de scripts

1. Introducción

En este documento se expone el trabajo de fin de grado correspondiente al Grado en Ingeniería Informática de la Universidad Politécnica de Valencia.

En este capítulo se describirán los motivos para la elaboración de este documento, los objetivos que se esperan lograr, el impacto que se espera conseguir con el cumplimiento de estos objetivos, la metodología empleada para el desarrollo del proyecto, la planificación seguida y la estructura utilizada durante todo el proceso de desarrollo de esta memoria.

1.1 Motivación

Actualmente, vivimos en un mundo cuya tendencia creciente es el estar cada vez más conectado. Por tanto, cada vez existe un número mayor de dispositivos que se encuentran conectados a internet, formando así una red conocida como IoT (*Internet of Things*).

Según se estima, en el año 2020 existirán más de 20.000 millones de dispositivos IoT [1]. El IoT ha supuesto un cambio positivo tanto para la realización de pequeños proyectos personales como para aumentar la capacidad de productividad de las empresas, gracias al aumento de información obtenida. Además de poder utilizarse para facilitar las conexiones en lugares de difícil acceso, siendo una tecnología más barata y ecológica [2].

Desde hace unos años el término *Smart Cities* o Ciudades Inteligentes viene siendo cada vez más importante debido a la obligación de dirigir nuestra vida entorno a la sostenibilidad. Algunas de las ciudades inteligentes más importantes son Tokyo, Londres o Nueva York. Estas ciudades utilizan un gran número de recursos IoT como es la utilización de molinos eólicos en farolas, paneles solares en semáforos o señales, para convertir la ciudad en lo más eficiente y sostenible posible [3].

Figura 1: Servicios esenciales de una Smart City [4]

En la figura 1 podemos observar detalladamente como los distintos elementos dentro de una *Smart City* se encuentran conectados.

Este proyecto viene motivado por el desarrollo de nuevos servicios relacionados con la geolocalización de los recursos ofreciendo una pequeña infraestructura y guía de desarrollo que nos posibilite la oportunidad de desarrollar soluciones en el ámbito de la IoT. De esta forma se añadirá una nueva capacidad de interacción con el laboratorio de la *Smart City* del grupo de investigación TaTami del centro PROS.

1.2 Objetivos

El objetivo principal del proyecto consiste en identificar los problemas más reseñables que se producen al interpretar las posiciones GPS proporcionadas por los recursos IoT y proporcionar una solución de diseño e implementación a estos problemas.

El segundo objetivo se centra en definir unas reglas de interacción geolocalizadas cuyo fin de definir estas reglas es resolver los problemas que se producen cuando varios recursos IoT se encuentran en una localización aproximada y debe producirse una interacción entre ellos.

El tercer objetivo es el de diseñar unas soluciones haciendo uso de las reglas de interacción geolocalizadas definidas anteriormente.

El cuarto objetivo es el de ejemplificar estas soluciones realizando un escenario en el ámbito de las *Smart Cities* haciendo uso de los recursos proporcionados por la plataforma / laboratorio de la *Smart City* del grupo de investigación TaTami del centro PROS.

1.3 Impacto esperado

El impacto deseado con el desarrollo de este proyecto es que una vez se hayan cumplido todos los objetivos propuestos se pueda utilizar esta nueva guía de desarrollo para la creación de nuevas soluciones en el ámbito IoT.

1.4 Metodología

Una vez identificados y definidos los objetivos del proyecto es necesario elegir una correcta metodología para el desarrollo de este. En este caso, se ha decidido elegir una metodología basada en un desarrollo iterativo e incremental.

El desarrollo iterativo e incremental se caracteriza por ser un proceso de desarrollo *software* que agrupa un conjunto de tareas en etapas que se repiten durante el ciclo de vida del proyecto.

Cada vez que finalizamos una etapa obtenemos un producto funcional que se mejora en cada iteración.

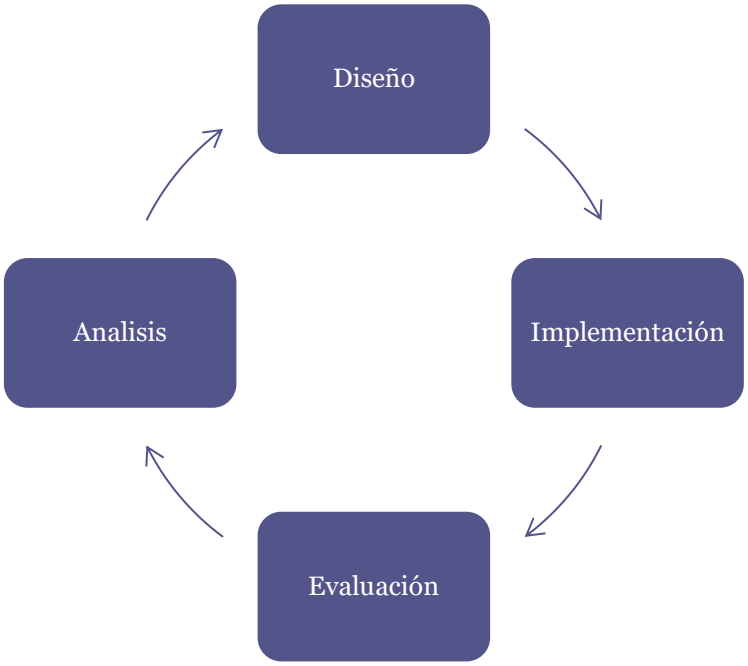

Figura 2: Ciclo de vida iterativo e incremental

Por tanto, podemos apreciar que esta metodología es iterativa por que consiste en un número distinto de tareas que se repiten (proceso iterativo) de forma que aportan o incrementan la calidad o las funcionalidades del proyecto (incremental) [5].

Como se ilustra en la figura 2 podemos observar las distintas fases que componen el ciclo de vida iterativo e incremental.

En la fase de análisis de requisitos se debe analizar las necesidades de los usuarios finales para determinar los objetivos.

En la fase de diseño, el objetivo es describir la estructura interna del software, y las relaciones entre los distintos componentes que lo formen.

En la fase de implementación se implementan las distintas soluciones obtenidas en las etapas anteriores, dando como resultado un programa funcional.

Finalmente, en la fase de evaluación nos encargaremos de comprobar que el programa funcional obtenido en la fase de implementación cumple con los requisitos especificados en la primera fase.

Continuando con el ciclo iterativo e incremental, se vuelve a empezar una nueva iteración donde se evalúa de nuevo el estado del proyecto, identificando nuevos riesgos y requisitos. Siendo posible repetirse tantas veces como sea necesario.

El motivo de la utilización de este ciclo de vida ha sido la división del proyecto en dos partes, aportando una solución para cada uno de los escenarios identificados en los objetivos.

1.5 Planificación

Uno de los aspectos más importantes en el momento de realizar un proyecto es seguir una planificación adecuada que permita satisfacer de forma correcta todas las partes de este.

Siendo la planificación aplicada a este proyecto la siguiente:

- **Preparación de la memoria**: Durante todo el proyecto se ha ido desarrollando la memoria al mismo tiempo que se realizaban las distintas fases de este.
- **Investigación previa**: Tiempo necesario para comprender de forma correcta todas las tecnologías necesarias para el desarrollo del proyecto.
- **Caso de estudio**: Planteamiento del caso de estudio que se va a realizar y los dos escenarios a desarrollar para cumplir las soluciones propuestas.
- **Diseño**: En esta fase se ha abordado cómo realizar el diseño de las soluciones propuestas.
- **Implementación**: El objetivo principal de esta fase es el de desarrollar los dos escenarios cuyo fin es el de demostrar las guías de desarrollo propuestas durante el proyecto.
- **Pruebas**: Esta fase es la de realizar el *testing* de cada módulo que compone el proyecto con el fin de certificar su correcto funcionamiento.
- **Revisión general**: En esta fase se realiza una revisión general de todo el proyecto.

Para la planificación temporal del proyecto se ha optado por la utilización de un diagrama de Gantt que es una herramienta gráfica cuyo objetivo es mostrar la dedicación realizada a las distintas partes del proyecto durante el tiempo total determinado del mismo. Como puede observarse en la siguiente figura:

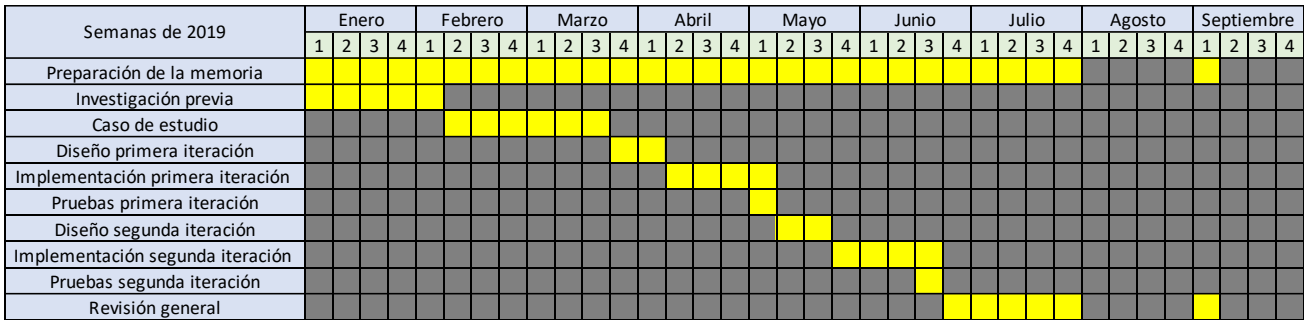

Figura 3: Diagrama de Gantt

1.6 Estructura del documento

Este documento se ha dividido en siete capítulos además de la bibliografía.

El primer capítulo es la introducción donde se describirán los motivos para la elaboración de este documento, los objetivos que se esperan lograr, el impacto que se espera conseguir con el cumplimiento de estos objetivos, la metodología empleada para el desarrollo del proyecto, la planificación seguida y la estructura utilizada durante todo el proceso de desarrollo de esta memoria.

El segundo capítulo se centrará en describir el *software* utilizado para el desarrollo del proyecto, diferenciando entre las distintas tecnologías, herramientas y lenguajes de programación utilizados. Además de destacar posibles alternativas a las utilizadas igual de validas que las mismas.

En el tercer capítulo se identifican los problemas o aspectos que se requieren resolver en este trabajo final de grado.

En el cuarto capítulo se partirá del análisis en más profundidad del problema analizado, en la realización del diseño de la solución con el objetivo de determinar todos los elementos necesarios para realizar la implementación.

El quinto capítulo definirá cómo se implementa el diseño de la solución y se realizara un análisis de los problemas que se propusieron durante el cuarto capítulo más importantes surgidos durante el proyecto, detallando el procedimiento a realizar en cada caso.

En el sexto capítulo se realizarán las pruebas pertinentes para la comprobación del correcto funcionamiento de la solución implementada.

Finalmente, en el séptimo capítulo se efectuará un resumen del trabajo realizado durante el proyecto.

A continuación de los siete capítulos se enunciará la bibliografía utilizada para la realización del proyecto y la memoria.

2. Contexto Tecnológico

Este capítulo se centrará en describir el *software* utilizado para el desarrollo del proyecto, diferenciando entre las distintas tecnologías, herramientas y lenguajes de programación utilizados. Además de destacar posibles alternativas a las utilizadas igual de validas que las mismas.

2.1 Entorno de desarrollo

Se ha decidido utilizar el *software* Eclipse en su versión 4.8 (Eclipse *Photon*) debido a las facilidades que nos ofrece para la utilización de todos los elementos necesarios para el desarrollo del proyecto.

Originalmente, Eclipse fue desarrollado por IBM, aunque actualmente es una plataforma de código abierto desarrollado por la Fundación Eclipse. Siendo uno de los más utilizados por su facilidad de uso, capacidad para crear entornos de trabajo y capacidad de adaptación. Además, al igual que otros entornos de desarrollo (como NetBeans o IntelliJ IDEA), Eclipse es multiplataforma pudiéndose utilizar en cualquier sistema operativo.

Eclipse es una plataforma de software que ha sido utilizada típicamente para el desarrollo en el lenguaje de programación Java, aunque es compatible con otros lenguajes de programación como C++ o C.

Los principales componentes que forman Eclipse son: OSGi, SWT, JFace i Workbench. Siendo una de las principales ventajas que ofrece la arquitectura modular, donde en el *Marketplace* de Eclipse podemos encontrar la posibilidad de permitir a los usuarios instalar *plugins* o desarrollar sus propios *plugins*. Esto abre la posibilidad de utilizar herramientas externas como puede ser GitHub o utilizar otros lenguajes de programación como puede ser Python o PHP [6].

Algunas posibles herramientas alternativas podrían ser NetBeans o IntelliJ IDEA.

- **NetBeans**: Es una herramienta de código abierto al igual que Eclipse, siendo este su principal competidor. Algunos de los usuarios lo prefieren respecto a Eclipse por que algunos de los *plugins* más importantes vienen ya instalados, de forma que no tienen que configurar el entorno.

- **JetBrains**: Es una empresa privada, por tanto, no es de código abierto como NetBeans o Eclipse. A su vez JetBrains divide cada entorno de programación enfocado en un lenguaje distinto, por ejemplo, para el lenguaje de programación Java se utiliza IntelliJ IDEA, mientras que para Python se utiliza PyCharm.

2.2 Lenguaje de programación

Se ha determinado utilizar el lenguaje de programación Java, para el progreso de este proyecto debido a que como la plataforma/laboratorio de la Smart City está desarrollado en Java, y como este proyecto consiste en añadir un nuevo escenario a la Smart City, es necesario utilizar el lenguaje de programación Java.

Java es un lenguaje de programación concebido por la empresa Sun Microsystems y posteriormente adquirido por la compañía Oracle en 2010. Las principales características que destacan de Java es que es de propósito general, incluye concurrencia, es orientado a objetos, además de ser diseñado específicamente para tener el menor número de dependencias de implementación posibles.

Gran parte de su sintaxis proviene de C y C++, pero presenta menos utilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java se compilan en *bytecode* (clase Java), lo que significa que pueden ser ejecutadas en cualquier máquina virtual Java (JVM) sin importar la arquitectura del ordenador utilizado [7].

Las características que ofrece son [8]:

- **Orientado a objetos**: es un paradigma de programación donde los objetos manipulan los datos de entrada con el fin de obtener datos de salida específicos, de forma que cada objeto proporciona una funcionalidad especial.

- **Independencia de la plataforma**: Esto indica que el *software* desarrollado en Java puede ejecutarse igualmente en cualquier hardware.

- **Concurrente**: Permite la opción de ejecutar nuestros programas en varios hilos de ejecución.

- **Dinámico**: Posibilita la opción de modificar el entorno en tiempo de compilación.

- **Robusto**: Desde un inicio, Java fue diseñado para tener una gran fiabilidad, de esta forma se producen numerosas comprobaciones en tiempo de compilación y ejecución.

- **Distribuido**: Uno de los principales puntos fuertes de Java es la posibilidad de crear aplicaciones distribuidas gracias a la proporcionalidad de clases con el fin de utilizarse en aplicaciones de red.

- **Lenguaje simple**: Este lenguaje se considera que tiene una curva de aprendizaje más rápida que otros lenguajes de programación. Además, en caso de haber aprendido primero otros lenguajes de programación como C o C++ se tendrá mayor facilidad para aprender Java con rapidez, debido a que se han eliminado algunas características destacadas de C y C++ en Java como es el caso de los punteros.

Algunos lenguajes de programación orientados a objetos son:

- **C++**: Fue creado como una mejora respecto a C incorporando la opción de ser orientado a objetos, aunque se suele manifestar que C++ es un lenguaje de programación multiparadigma debido a que se decidió incorporar también un paradigma de programación estructurada además del de programación orientada a objetos.

- **Python**: Se define como un lenguaje de programación interpretado donde se hace énfasis en tener una sintaxis que favorezca la legibilidad del código. Al igual que C++, Python es un lenguaje multiparadigma debido a que proporciona soporte a la programación orientada a objetos, la programación imperativa y la programación funcional.

- **C#**: Es un lenguaje de programación orientado a objetos con características similares a C++, pero en este caso siendo propiedad de Microsoft.

2.3 Comunicaciones

Una parte fundamental cuando se trata de un proyecto IoT es la comunicación entre los distintos componentes que la forman.

2.3.1 MQTT

El MQTT (*Message Queuing Telemetry Transport*) es un protocolo ideado por IBM para la comunicación entre maquinas (*Machine-to-Machine*). Una de la primordial virtud que proporciona es que se requiere muy poco ancho de banda para su uso y destaca por un bajo consumo.

Hace uso de una tipología de estrella donde un bróker central se encarga de gestionar los mensajes mediante el uso de un esquema de comunicación publicación/suscripción.

En el interior de la arquitectura de MQTT, es muy considerable destacar el concepto de *topic* ya que mediante estos *topics* se estructura la comunicación, debido a que emisores y receptores tienen que estar subscritos a un mismo *topic* común para poder producirse la comunicación [9].

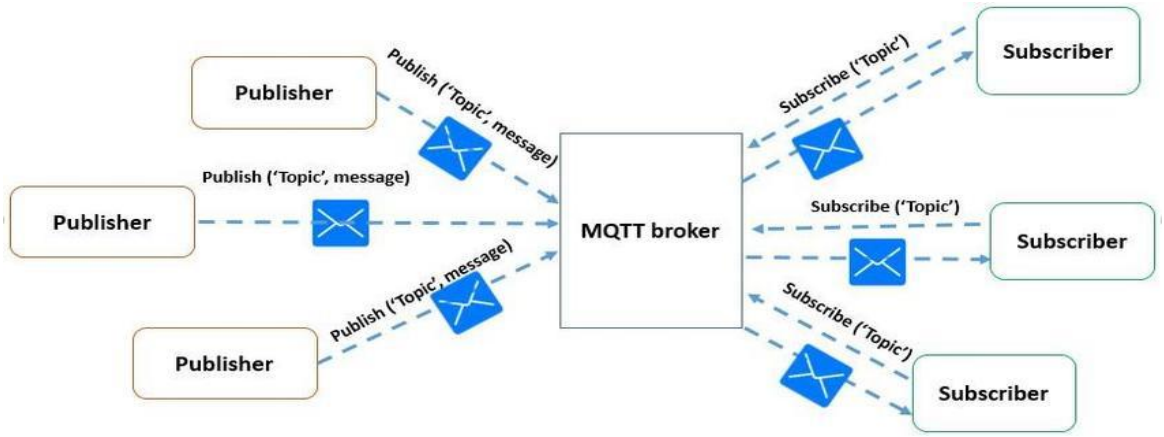

Figura 4: MQTT Esquema

En la figura 4 podemos observar detalladamente como funciona el protocolo MQTT. Algunos protocolos alternativos a MQTT son:

- **AMQP**: Se define como un protocolo binario (*Advanced Message Queuing Protocol*) de código abierto que permite la transferencia de mensajes asíncronos entre clientes y servidores [10].

- **STOMP:** (*Streaming Text Orientated Messaging Protocol*) es un protocolo de tipo textual, sencillo y simple de implementar que admite comunicar clientes STOMP con cualquier *Message Broker* STOMP [11].

El motivo principal por el que se ha decidido utilizar MQTT es por las ventajas que ofrece respecto a otros protocolos alternativos que ayudan a facilitar su uso en IoT. El cual requiere muy poco ancho de banda para su uso y destaca por un bajo consumo.

2.3.2 Eclipse Mosquitto

Eclipse Mosquitto se caracteriza por que es un bróker de mensajería de código abierto creado por Roger Light que implementa el protocolo asíncrono de mensajería MQTT. Una de las principales ventajas que ofrece respecto a otros brókeres de mensajería es su ligereza, ya que está diseñado para poder utilizarse tanto en dispositivos que tengan poca potencia hasta en servidores de grandes prestaciones [12]. Siendo el consumo tan reducido que solo consume 3 Megabytes por cada 1000 clientes conectados, una cantidad que sin duda resulta irrisoria [13].

Mosquitto forma parte de la fundación Eclipse y proviene de una librería en C para posibilitar la opción de implementar clientes MQTT.

Hay que resaltar la versatilidad que ofrece Mosquitto para poder utilizarse en sistemas operativos como Windows, CentOs, Ubuntu, etc

Existen algunas alternativas al bróker Mosquitto, como es el caso del bróker RabbitMQ, este bróker presenta enorme multitud de protocolos como son AMQP, MQTT, STOMP y HTTP. Dispone de una interfaz grafica para la monitorización y manipulación de los elementos de las distintas tecnologías que soporta [14].

Otro elemento interesante de RabbitMQ es que presenta una serie de *plugins* que permite mejorar la interoperabilidad de los distintos protocolos de mensajería y el bróker.

Hay que destacar la capacidad que presenta RabbitMQ para poder utilizarse en sistemas operativos como Windows, CentOs, Ubuntu, etc. Como por la multitud de lenguajes de programación como son Python, Ruby, C#, Java, JavaScript, Haskell, etc.

2.3.3 REST

REST (*Representational State Transfer*) es un estilo arquitectónico para definir estándares entre la comunicación de sistemas.

Una de las particularidades que definen REST es que es *stateless*, de esta forma cada solicitud del cliente al servidor debe incluir toda la aclaración imprescindible para poder entender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor. Entonces, el estado de sesión se conserva enteramente en el cliente, el cuál es responsable de almacenar y manejar toda la información relacionada con el estado de la aplicación en el lado del cliente.

Para referirnos a términos de información en REST se emplea la palabra recursos. Estos recursos se organizan mediante una URI (*Universal Resource identifier*). De esta forma para interactuar con los servicios REST, se accederá a la URI del recurso y se utilizará la operación pertinente (utilizándose las mismas que en HTTP, GET, PUT, POST y DELETE) [15].

Una alternativa a REST es el protocolo SOAP, el cual permite realizar servicios web sin ningún estado, a través de TCP y utilizando un formato XML. Además, SOAP se encuentra ampliamente estandarizado, lo que significa que existen reglas concretas para la creación del mensaje, el contrato entre cliente/servidor o el formato de datos a enviar.

El principal problema que ofrece SOAP es que, al estar ampliamente estandarizado, es poco flexible y se pueden producir errores durante el proceso de desarrollo. Debido a este motivo se ha decidido optar por REST para la realización de este proyecto.

2.3.4 JSON

JSON es un formato de texto sencillo para el intercambio de datos. Originalmente formaba parte de JavaScript, pero con el paso del tiempo debido a su amplia aceptación como una opción a XML, se constituyo como un formato independiente [16].

Uno de sus principales puntos a favor respecto a XML es la simpleza de los datos, ya que mientras que en XML se necesita un formato concreto, en JSON una cadena de texto se puede utilizar como un objeto. Por este motivo es por el que se ha optado por la utilización de JSON para el proyecto.

JSON utiliza un lenguaje marcado mientras que XML se presenta como un metalenguaje que permite definir lenguaje de marcas. Al igual que HTML, XML posibilita la opción de definir la gramática de lenguajes específicos para estructurar documentos grandes, además, proporciona soporte a base de datos resultando de utilidad cuando algunas aplicaciones quieren comunicarse entre si con el fin de integrar información [17].

XML

```
<empinfo>
   <employees>
      <employee>
         <name>James Kirk</name>
         <age>40></age>
      </employee>
      <employee>
         <name>Jean-Luc Picard</name>
         <age>45</age>
      </employee>
      <employee>
         <name>Wesley Crusher</name>
         <age>27</age>
      </employee>
   </employees>
</empinfo>
```

JSON

```
{  "empinfo" :
      {
            "employees" :  [
            {
                  "name" : "James Kirk",
                  "age" : 40,
            },
            {
                  "name" : "Jean-Luc Picard",
                  "age" : 45,
            },
            {
                  "name" : "Wesley Crusher",
                  "age" : 27,
            }
                              ]
      }
}
```

Figura 5: Ejemplo JSON y Lenguaje de marcas [18]

En la figura 5 podemos apreciar la diferencia entre el lenguaje de marcado y el metalenguaje de una forma más clara, con el fin de esclarecer lo expuesto anteriormente.

2.3.5 Owntracks

Owntracks es una plataforma de código abierto que nos permite realizar un seguimiento de nuestra ubicación. Además, permite crear un diario propio donde tengamos un registro de esta o compartirlo con nuestros amigos o familiares.

Uno de los aspectos que más han sido significativos para la utilización de Owntracks, es el hecho de que sea de código abierto, también nos proporciona la posibilidad de hacer que estas ubicaciones sean enviadas a un servidor propio.

Otra característica fundamental es la posibilidad de definir regiones o zonas en la que se detecte cuando se entra y se sale mediante nuestro movimiento, gracias a que Owntracks nos lo notificara.

Además, hay que resaltar otra característica reseñable, los iBeacons que son un sistema de posicionamiento en interiores propiedad de Apple.inc que funciona mediante la conectividad Bluetooth que se incorpora en el software Owntracks.

Owntracks está disponible tanto para iOS como para Android [19].

Figura 6: Ejemplo ventana principal owntracks

Su funcionamiento es sencillo, en la figura 6 se aprecia un ejemplo de la ventana principal de la aplicación, donde encontramos las localizaciones, tanto nuestras como de nuestros amigos o familiares. En la parte inferior también tenemos disponible las opciones de “Friends” o “Regions” como se puede observar en la figura 7.

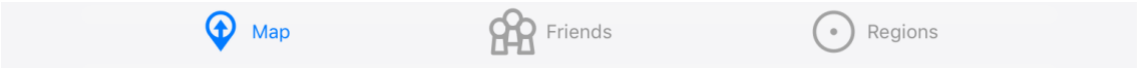

Figura 7: Ejemplo barra inferior owntracks

Si pulsamos en la barra superior en la “i” de información (como se aprecia en la figura 8), accederemos al panel “Status Info” donde podremos configurar los parámetros de conexión a nuestro servidor, añadiendo la información en los “Settings”.

Figura 8: Ejemplo barra superior owntracks

En la siguiente imagen podemos ver un ejemplo de como se ha llevado a cabo la configuración de Owntracks. Se define el Host, el puerto utilizado, además del usuario y contraseña para poder empezar a compartir la información.

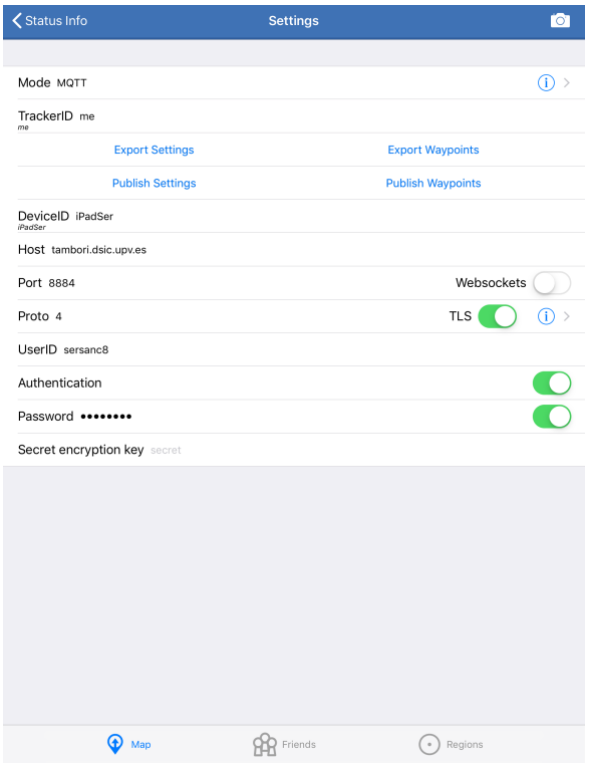

Figura 9: Settings

Si volvemos al menú principal y nos fijamos en “Regions” podemos observar las regiones que hemos creado con las coordenadas de estas.

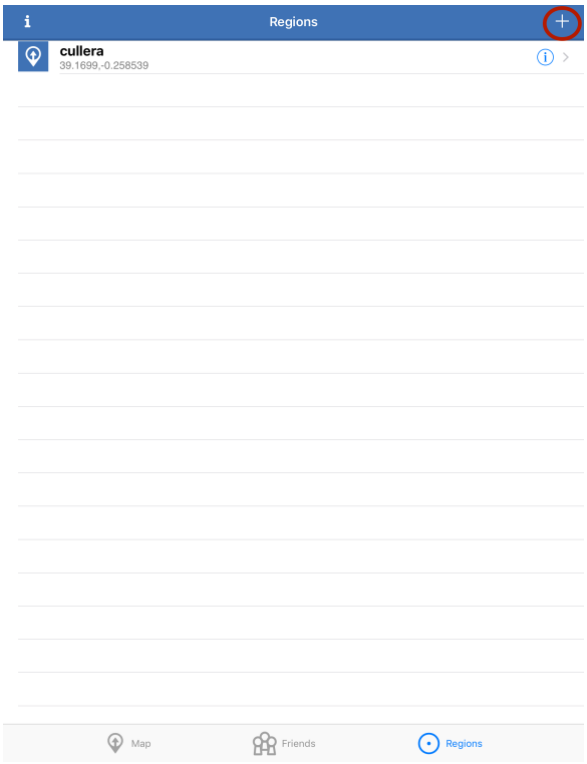

Figura 10: Menu regiones owntracks

En la esquina superior derecha tenemos la posibilidad de modificar algunos parámetros como es el nombre, el radio de la región, la latitud o la longitud (como se aprecia en la figura 10).

No olvidemos también la posibilidad de incluir iBeacons que son un sistema de posicionamiento en interiores desarrollado por Apple Inc. Como se observa en la figura 11 tenemos incluidos los valores predefinidos de nuestra posición y si lo deseamos los podemos modificar.

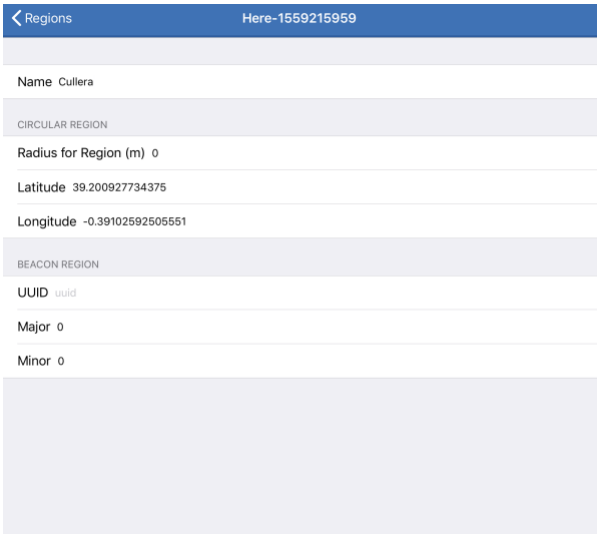

Figura 11: Ejemplo regiones owntracks

2.4 Control de versiones y gestión del proyecto

Uno de los aspectos más relevantes a lo largo del proyecto es tener un control de las distintas versiones que se van produciendo, asimismo de las herramientas que pueden servir de utilidad para la gestión de este.

2.4.1 GitLab

GitLab se define como un sistema de alojamiento de repositorios basado en Git. Promulgado bajo una licencia de código abierto, el servicio permite también el almacenamiento de wikis y seguimiento de errores.

Una de las principales características que ofrece GitLab es una distinción de versiones en función de si se necesita para usuarios (GitLab *Comunity Edition*) o para empresas (GitLab SaaS). Además, usan una aplicación única desarrollada desde cero con el fin de dar soporte al ciclo entero de desarrollo, en vez de integrar múltiples herramientas distintas [20].

También permite crear repositorios privados de forma gratuita, lo cual resulta una gran ventaja frente a otras herramientas que no ofrecen este servicio de forma gratuita.

Existen algunas herramientas alternativas a GitLab también basadas en Git, como son el caso de GitHub y Bitbucket.

- **GitHub**: Al igual que GitLab se trata de un sistema de alojamiento de repositorios, pero difieren en algunos conceptos. Github, es propiedad de Microsoft por tanto no se encuentra publicado bajo una licencia de código abierto. Mientras que para utilizar un repositorio privado se ha de utilizar la versión de pago.

- **Bitbucket**: Al igual que GitLab y GitHub se trata de un sistema de alojamiento de repositorios. Una de las principales diferencias con GitHub y GitLab es que mientras que GitLab nos permite crear repositorios privados con un número de colaboradores ilimitado de forma gratuita, GitHub nos ofrece los repositorios privados en su versión de pago y Bitbucket nos ofrece los repositorios privados limitado a máximo 5 colaboradores [21].

En la figura 12 podemos apreciar las tres principales herramientas que conforman git.

Figura 12: GitHub,GitLab, Bitbucket [22]

2.4.2 Gradle

Gradle, es un instrumento cuyo fin es la automatización de compilación de código abierto, centrándose en la flexibilidad y el rendimiento. Los scripts de compilación de Gradle se escriben utilizando Groovy o Kotlin DSL (*Domain Specific Language*).

Uno de los principales puntos fuertes de Gradle es la gran flexibilidad que nos proporciona dejándonos hacer usos de otros lenguajes y no solo de Java, además cuenta con un sistema de gestión de dependencias muy estable.

Por si fuera poco, Gradle destaca por ser altamente personalizable además de rápido ya que completa las tareas de forma rápida y precisa gracias a la reutilización de las salidas de las ejecuciones anteriores, solamente es necesario procesar las entradas que presentan cambios en paralelo.

Otra de las características de Gradle, es que es el sistema de compilación oficial para Android y proporciona soporte para diversas tecnologías y lenguajes [23].

Por todos estos motivos se ha determinado utilizar Gradle en el proyecto.

```
plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'java-library'
}
apply plugin: 'application'
mainClassName = 'Main'

dependencies {
    // This dependency is exported to consumers, that is to say found on their compile classpath.
    api 'org.apache.commons:commons-math3:3.6.1'

    // This dependency is used internally, and not exposed to consumers on their own compile classpath.
    implementation 'com.google.guava:guava:23.0'
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.1'
    // Use JUnit test framework
    testImplementation 'junit:junit:4.12'
}

// In this section you declare where to find the dependencies of your project
repositories {
    // Use jcenter for resolving your dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}
```

Figura 13: Ejemplo funcionamiento de gradle

En la figura 13 podemos observar un ejemplo en el que se ilustra como funciona gradle, donde todo se gestionara a través del archivo build.gradle que es el encargado de gestionar el build. Como se aprecia, se trata de un DSL (*Domain Specified Language*) bastante compacto.

3. Análisis del problema

3.1 Introducción

En este capítulo se describen los problemas o aspectos que se requieren resolver en este trabajo final de grado. Además, de los requisitos necesarios para poder diseñar una solución que permita trabajar con localizaciones en el ámbito IoT, donde, empleando distintas estrategias de comunicación, se pretende desarrollar interacciones entre dispositivos y recursos a través de las localizaciones de estos.

En base a esto, se deberán diseñar unas reglas de interacción con el fin de expresar acciones / interacciones entre recursos haciendo uso de su localización.

3.2 Problemática presente

Uno de los problemas principales que se pretende resolver a lo largo de este proyecto, es la necesidad de interpretar correctamente las posiciones GPS obtenidas mediante la consulta de algún tipo de API (conjunto de funciones y procedimientos que cumplen una o muchas funciones con el fin de ser utilizados por otro *software*) que dispongan los recursos IoT, pese a que hagan uso de distintos modelos geoespaciales o de diferentes formatos de datos.

Otro problema importante que se pretende resolver es el que se produce cuando varios recursos IoT se encuentran en una localización aproximada, siendo necesario que interaccionen entre ellos de forma adecuada.

3.3 Requisitos iniciales

Uno de los requisitos más importantes que se identifican en el ámbito de las interacciones geolocalizadas, es la necesidad imperiosa de que los recursos IoT se encuentren conectados en todo momento y sean accesibles, permitiendo así que puedan ser utilizados por diferentes usuarios sea cual sea su objetivo.

Además, para ello será necesario que se dispongan de algún tipo de API para poder consultar la posición GPS de los recursos IoT.

También, uno de los requisitos principales producidos al realizar la comunicación entre diversos componentes *software*, surgen en el modelo de comunicación indirecta o asíncrona, ya que si los componentes no se encuentran correctamente configurados pueden producirse pérdidas de mensajes.

3.4 Interacciones geolocalizadas posibles

En este apartado se estudiarán las posibles acciones / situaciones que pueden producirse al tratar con geolocalizaciones.

En el caso de tener que tratar con un solo recurso IoT, sería recomendable conocer la posición GPS (aunque el formato de datos que presente no sea el adecuado), además de conocer si se encuentra en movimiento, de forma que puede ser interesante conocer si se acerca o si se aleja de una posición GPS. También podría resultar interesante saber si el recurso IoT se encuentra cerca o lejos de una posición GPS concreta para que, si ese fuera el caso, realizar las acciones pertinentes.

Otra posible situación que se puede producir es cuando se debe tratar con dos recursos IoT, en este caso cada uno de los recursos IoT puede tener un rol diferente, como es el caso de que uno se encuentre en movimiento y el otro no. Al igual que en el ejemplo sería recomendable conocer las posiciones GPS, y si fuera el caso de que este en movimiento, saber si se esta acercando o alejando un recurso IoT respecto del otro o si se encuentran cerca o lejos el uno del otro.

Los problemas que presentan estas acciones / situaciones son que nos encontramos con recursos IoT de varios tipos como son móviles e inmóviles, lo cual supone una dificultad para acceder a su posición GPS, además de que será importante gestionar las interacciones geolocalizadas entre varios recursos IoT.

3.5 Reglas de interacción geolocalizadas

En el apartado anterior se han expuesto las posibles acciones / situaciones que pueden producirse al tratar con geolocalizaciones.

En este apartado se procederá a realizar uno de los objetivos de este proyecto, que es el de definir unas reglas de interacción geolocalizadas para resolver los problemas que se producen cuando varios recursos IoT se encuentran en una localización aproximada y debe producirse una interacción entre ellos. Las cuales pueden ser:

- **Cerca**: Un semáforo esta cerca de otro semáforo.
- **Lejos**: Un vehículo inmóvil se encuentra lejos de un comercio local.
- **Acercándose**: Un vehículo en movimiento se acerca a un semáforo.
- **Alejándose**: Un vehículo en movimiento se aleja de una estación de servicio.
- **Entrar**: Un vehículo entra en un túnel de lavado.
- **Salir**: Un vehículo sale de un garaje.

4. Diseño de la solución

Este capítulo parte de un análisis profundo del problema, siendo su objetivo la realización del diseño de la solución para posteriormente determinar todos los elementos necesarios para realizar la implementación de la solución.

4.1 Arquitectura software

Uno de los aspectos más importantes para llevar a cabo una implementación en todo proyecto *software*, es el de utilizar una arquitectura *software* adecuada para desarrollar el proyecto pertinente.

La arquitectura de *software* es la organización fundamental de un sistema encarnado en sus componentes, las relaciones entre ellos y el ambiente, además de los principios que orientan su diseño y evolución.

Existen un gran número de arquitecturas *software*, siendo las más comunes las siguientes [24]:

- **Arquitectura cliente-servidor**: Este tipo de arquitectura se centra en repartir el computo en dos partes independientes, pero sin existir un reparto claro de las funciones.

- **Arquitectura pipeline:** Este tipo de arquitectura se centra en que los sistemas / servicios se encadenan para ofrecer una funcionalidad donde cada parte tiene una responsabilidad limitada y clara. De esta forma no se producen excesivas dependencias entre los sistemas.

- **Arquitectura en capas:** Este tipo de arquitectura se centra en un modelo de desarrollo de *software* donde el objetivo primordial es la separación por partes que componen un sistema *software*.

- **Arquitectura modelo-vista-controlador:** Este tipo de arquitectura se centra en separar los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Para la utilización de este proyecto se ha decidido utilizar la arquitectura pipeline debido a que es el tipo de arquitectura que más encaja en el sistema que se desea implementar.

4.2 Estrategias de comunicación

Este apartado se centra en tratar las distintas formas de establecer la comunicación entre los distintos recursos IoT que se proporcionan en la *Smart City*.

4.2.1 Comunicación síncrona o directa

Este esquema de comunicación se caracteriza por que siempre existe un receptor que se encuentra esperando una conexión (o una petición de conexión) para que la comunicación pueda establecerse entre emisor y receptor.

De forma que obliga a definir el orden de aparición de los distintos elementos que forman parte de la comunicación, teniendo que existir siempre un receptor en el momento en el que aparezca un emisor.

Un ejemplo podría ser la comunicación que se establecen entre navegadores web y servidores web. Otro ejemplo de comunicación síncrona, muy relacionada con la solución que se va a definir a lo largo del proyecto es la arquitectura REST.

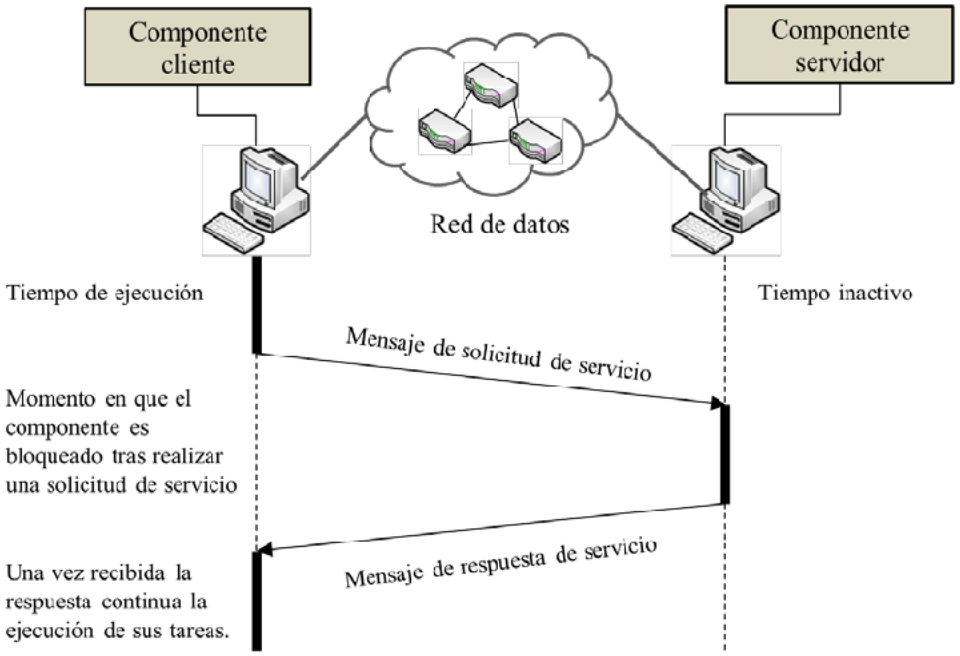

Figura 14: Uso de comunicación síncrona entre dos dispositivos [25]

En la figura 14 podemos apreciar un ejemplo de la explicación expuesta anteriormente.

4.2.2 Comunicación asíncrona o indirecta

Este esquema de comunicación difiere del esquema anterior, en que en este caso no es necesario que existan receptores para que un mensaje pueda enviar la información, por tanto, no es necesario definir el orden de los elementos que van a intervenir en la comunicación.

Los problemas que presenta es que en diversas situaciones pueden producirse momentos de incomunicación, como puede ser el caso de que en un sistema existan muchos emisores, pero ningún receptor disponible. Aunque el funcionamiento de los receptores no se vería comprometido, simplemente la información que están tratando de transmitir no sería transmitida, por tanto, la no disponibilidad de receptores o emisores no afecta al funcionamiento del sistema.

Algunos ejemplos de sistemas que hacen uso de comunicación asíncrona serían, el correo electrónico, donde entre el momento de la emisión de la información y el procesamiento por parte del receptor puede transcurrir un lapso que varia entre el procesamiento casi inmediato o hasta días después de la recepción. Otro ejemplo de comunicación asíncrona muy relacionado con la solución que se va a definir a lo largo del proyecto es la comunicación MQTT.

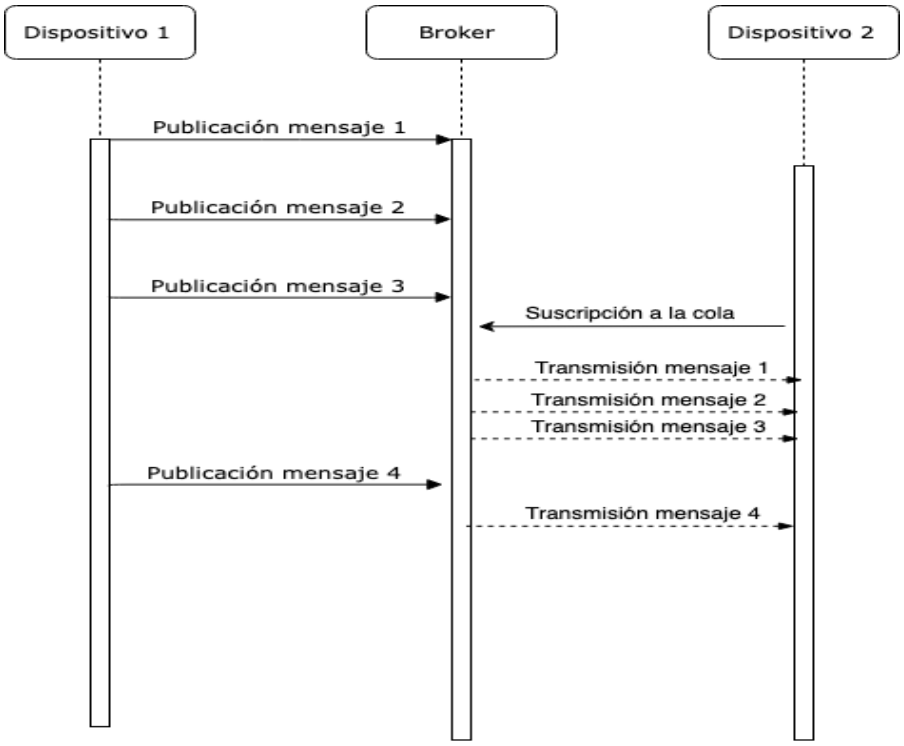

Figura 15: Ejemplo mensajería asíncrono o indirecto

En la figura 15 podemos apreciar un esquema de la explicación expuesta anteriormente.

4.3 Diseño infraestructura

Partiendo de las reglas propuestas anteriormente y de los problemas presentes en el ámbito de las interacciones geolocalizadas se han propuesto unas soluciones para solventar estos problemas, cumpliendo con el objetivo de definir y diseñar una infraestructura que ayude a poner en funcionamiento las soluciones.

4.3.1 Solución controlador de acciones de interacción geolocalizadas

Como se ha expuesto anteriormente, un problema al tratar con geolocalizaciones que se pretende resolver es el que se produce cuando varios recursos IoT se encuentran en una posición GPS aproximada y tiene que producirse la interacción entre ellos de forma adecuada. Con el fin de solventar este problema se han propuesto distintas reglas de interacción.

En el caso de las interacciones geolocalizadas en el ámbito de la *Smart City*, un ejemplo que se podría aplicar siguiendo las reglas de interacción propuestas, sería la utilización de las posiciones GPS de los recursos IoT disponibles donde se pretende diseñar un controlador de acciones de interacción geolocalizadas cuyo propósito sea el de implementar un servicio de tipo “gestión / control” de localizaciones realizando una comparación entre las ubicaciones de un recurso móvil y un recurso inmóvil (en diversas situaciones como son: cerca / lejos, acercándose / alejándose, dentro / fuera, etc) realizando las acciones pertinentes sobre los mismos.

Para ello será necesario que los recursos IoT dispongan de una API accesible para poder conocer la posición GPS de los recursos IoT en movimiento o recursos IoT inmóviles (aunque en este caso esta solución sería menos optima).

En el caso del recurso inmóvil es recomendable que publique su ubicación y posteriormente hacer uso del “Adaptador fuente de posiciones GPS” para la adaptación de la fuente de posiciones GPS al servicio de “gestión/control”.

En la figura 16 se observa con detalle el esquema seguido para diseñar la solución propuesta.

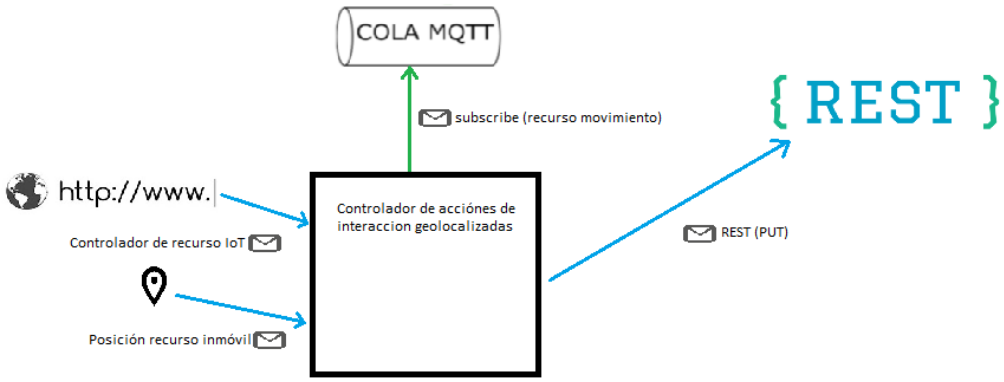

Figura 16: Esquema "Controlador de acciones de interacción geolocalizadas"

4.3.2 Solución adaptador de fuente de posiciones GPS

Como se ha expuesto anteriormente un problema en el momento de tratar con geolocalizaciones es que las posiciones GPS pueden hacer uso de distintos modelos geoespaciales o de diferentes formatos de datos.

Para ello la solución propuesta a este problema es la de diseñar una solución que sirva como herramienta donde partiendo de una fuente de posiciones GPS se pueden realizar un conjunto de cambios, con el objetivo de posibilitar la utilización de los recursos que componen la fuente de posiciones GPS en otra fuente de posiciones GPS distinta.

Por tanto, será necesario un controlador de recursos IoT que disponga de una API accesible (permitiendo así que los distintos usuarios puedan hacer uso de un conjunto de funciones y procedimientos, con el fin de ser utilizados por otro *software*) para la obtención de esta fuente de posiciones GPS, posteriormente se realizaran los cambios pertinentes para realizar la adaptación de los datos obtenidos y se publicarán a una nueva fuente de posiciones GPS.

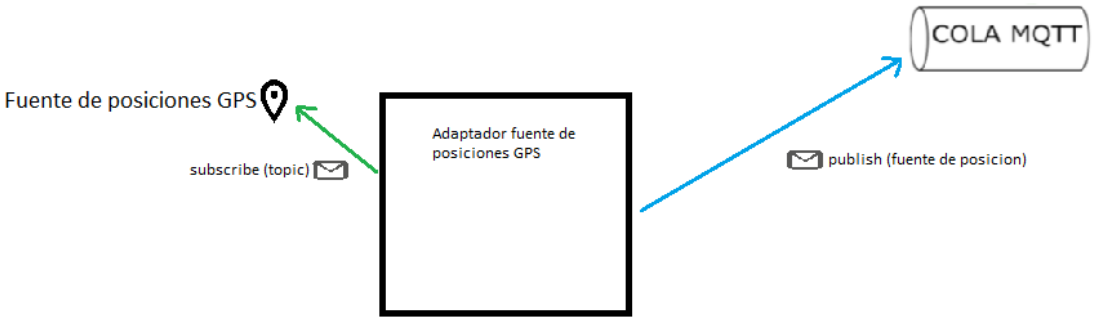

Figura 17: Esquema "Adaptador fuente de posiciones GPS"

En la figura 17 se observa con detalle el esquema seguido para diseñar la solución propuesta.

4.3.3 Solución conjunta

Estas soluciones pueden trabajar tanto individualmente como conjuntamente, donde en el caso de trabajar de forma conjunta, primero el “Adaptador fuente de posiciones GPS” adaptará la fuente de posiciones GPS proporcionadas por el recurso móvil al formato utilizado por el “Controlador de acciones de interacción geolocalizadas”. De esta forma el “Controlador de acciones de interacción geolocalizadas” utilizará la fuente de posiciones GPS proporcionada por el “Adaptador fuente de posiciones GPS”.

Además, será necesario disponer una API accesible que nos permita conocer la posición GPS del recurso móvil en movimiento.

También será necesario conocer la posición GPS del recurso inmóvil para ello será recomendable que el recurso implemente una API accesible para posibilitar su consulta.

Finalmente, se realizará el cálculo de la distancia entre el recurso móvil y el recurso inmóvil realizando las acciones pertinentes sobre los mismos.

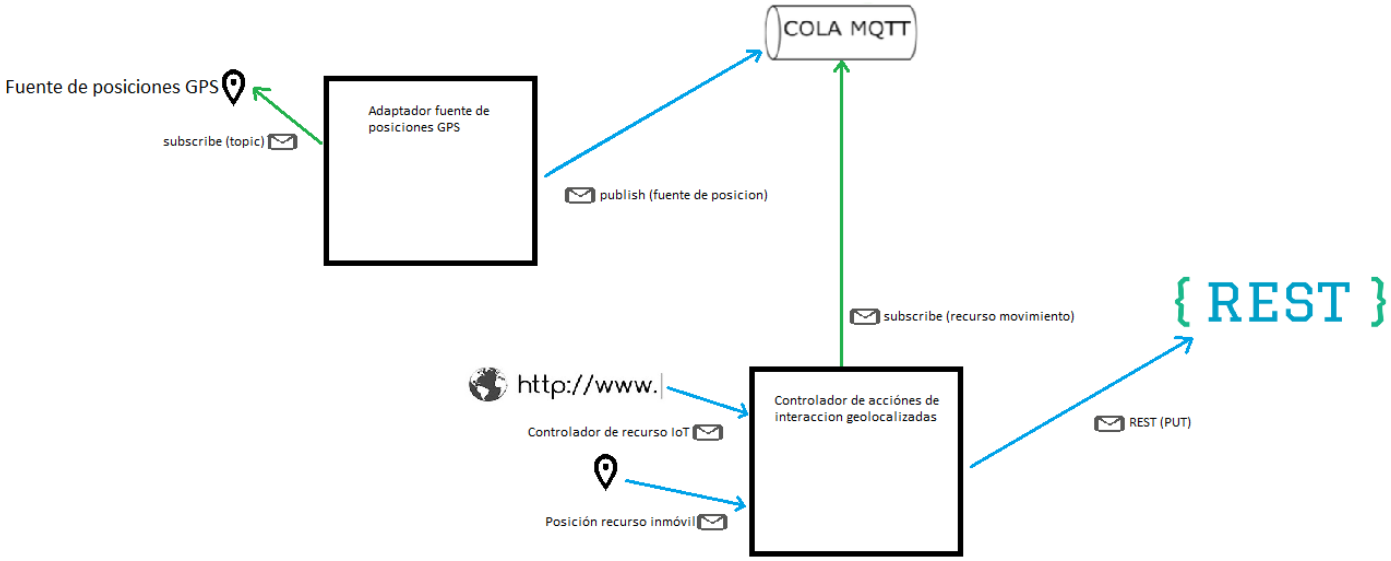

Figura 18: Funcionamiento de las dos soluciones al mismo tiempo

En la figura 18 se observa con detalle el esquema de las dos soluciones en conjunto.

4.4 Diseñando interacciones IoT geolocalizadas

Uno de los objetivos a desarrollar en este proyecto es el de definir y diseñar una infraestructura aplicándola en un escenario en el ámbito de la *Smart City.* Como ejemplo ilustrativo se ha decidido diseñar un servicio de atención a emergencias desplegado en vehículos de tipo ambulancia. De manera que cuando circulan atendiendo una emergencia, a través de la localización real de la ambulancia, solicita la regulación automática del tráfico.

4.4.1 Escenario controlador

Para la solución "Controlador de acciones de interacción geolocalizadas" se ha decidido desarrollar una aplicación "Controlador" que tiene como objetivo implementar un servicio de "gestión/control" de localizaciones entre un recurso en movimiento como es el caso de un vehículo de emergencias (ambulancia, coche de policía o bomberos) con un recurso inmóvil como sería el caso de un semáforo.

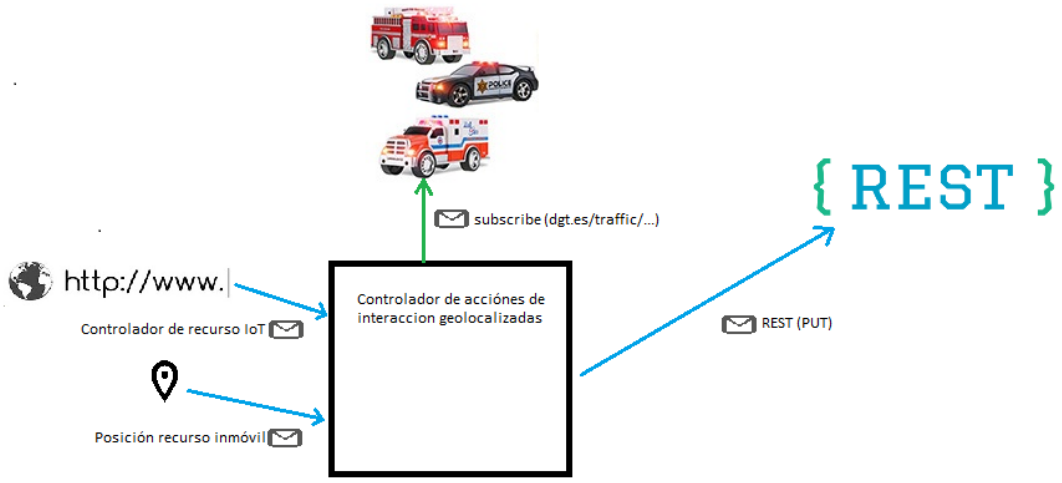

Figura 19: Esquema escenario controlador

En la figura 19 se puede observar una representación de la estructura que se ha llevado a cabo para la realización del escenario.

De esta forma, para iniciar el "Controlador" será necesario proporcionar el controlador de recursos IoT seleccionado que disponga tanto de una API accesible (que puede ser de tipo REST o de tipo MQTT), además de proporcionar la ubicación del recurso inmóvil en este caso el semáforo.

Posteriormente será necesario suscribirse a una cola de tipo MQTT donde se proporcionará la posición GPS del recurso móvil siendo en este caso un vehículo de emergencias (una ambulancia).

Una vez obtenida la información sobre las posiciones proporcionadas por el controlador semafórico y el vehículo de emergencias, será necesario realizar el cálculo de la distancia.

Finalmente, en función de la distancia obtenida, se solicitará al controlador de recursos IoT un cambio de estado de este para facilitar la interacción con el recurso en movimiento.

De forma que en caso de que el vehículo en estado de emergencias se encuentre a una distancia menor de 300m se solicitará al controlador semafórico un cambio de estado para otorgarle prioridad al vehículo en estado de emergencia para realizar el cruce, mientras que si se encuentra a una distancia mayor de 300m el controlador semafórico se mantendrá en el estado actual.

Por tanto, con la realización de este escenario se habrá cumplido el objetivo de definir y diseñar una infraestructura donde poner en funcionamiento las reglas de interacción definidas, haciendo uso de los recursos proporcionados por la *Smart City*.

4.4.2 Escenario adaptador owntracks

Para la solución “Adaptador fuente de posiciones GPS” se ha decidido desarrollar una aplicación “Adaptador Owntracks” que tiene como objetivo servir de herramienta para que partiendo de una fuente de posiciones GPS realizar un conjunto de cambios para posibilitar la utilización de los atributos que componen la fuente de posiciones GPS en otra fuente de posiciones GPS distinta.

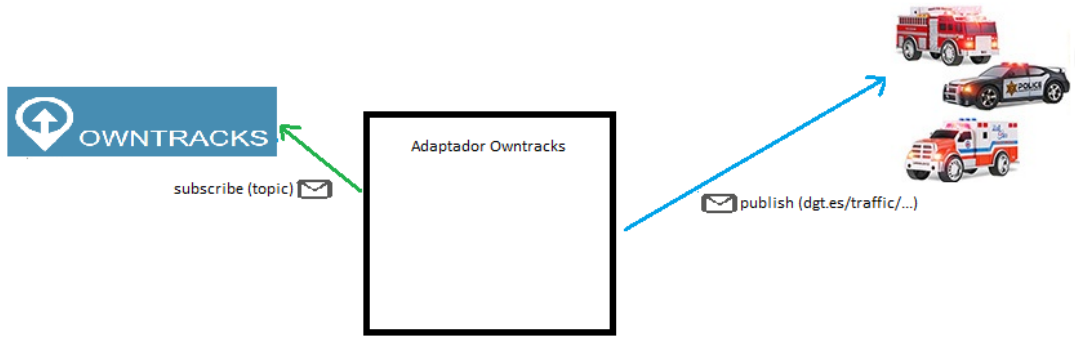

Figura 20: Esquema adaptador owntracks

En la figura 20 se puede observar una representación de la estructura que se ha llevado a cabo para la realización del escenario.

Para iniciar el “Adaptador Owntracks” será necesario proporcionarle una fuente de posiciones GPS, que se puede obtener mediante la utilización de una API accesible (siendo de tipo REST o de tipo MQTT).

Posteriormente será necesario adaptar la fuente de posiciones GPS obtenida a una más concreta, de forma que concuerde con el formato de datos de las posiciones GPS utilizado en la aplicación “Controlador”.

Finalmente, una vez realizados los cambios correspondientes a la fuente de posiciones GPS, se publicará al *topic* propio de la aplicación “Controlador”.

Por tanto, con la realización de este escenario se habrá resuelto la problemática que se produce al utilizar fuentes de posiciones GPS que hacen uso de formatos de datos distintos.

4.5 Diagrama de clases

Una vez expuestos los diferentes componentes que forman parte del sistema, se presenta un diagrama de clases que representa la estructura interna del mismo.

Además de las relaciones que se producen entre los distintos componentes del sistema, se exponen los atributos y funciones de los que dispone, para facilitar el entendimiento de la composición y funcionamiento del proyecto desarrollado.

En la siguiente imagen se observa detenidamente la estructura expuesta a lo largo del capítulo. Se pueden diferenciar fácilmente los dos escenarios que se han expuesto a lo largo de este capítulo.

Siguiendo con el esquema comentado, en el diagrama de clases se observan distintas relaciones de dependencia entre componentes siendo su función la de definir que un componente esta utilizando a otro, de forma que, si este último se altera, el anterior se verá afectado.

Por último, es importante comentar las relaciones de asociación. Como se puede observar en la siguiente imagen la existencia de una relación estructural entre clases.

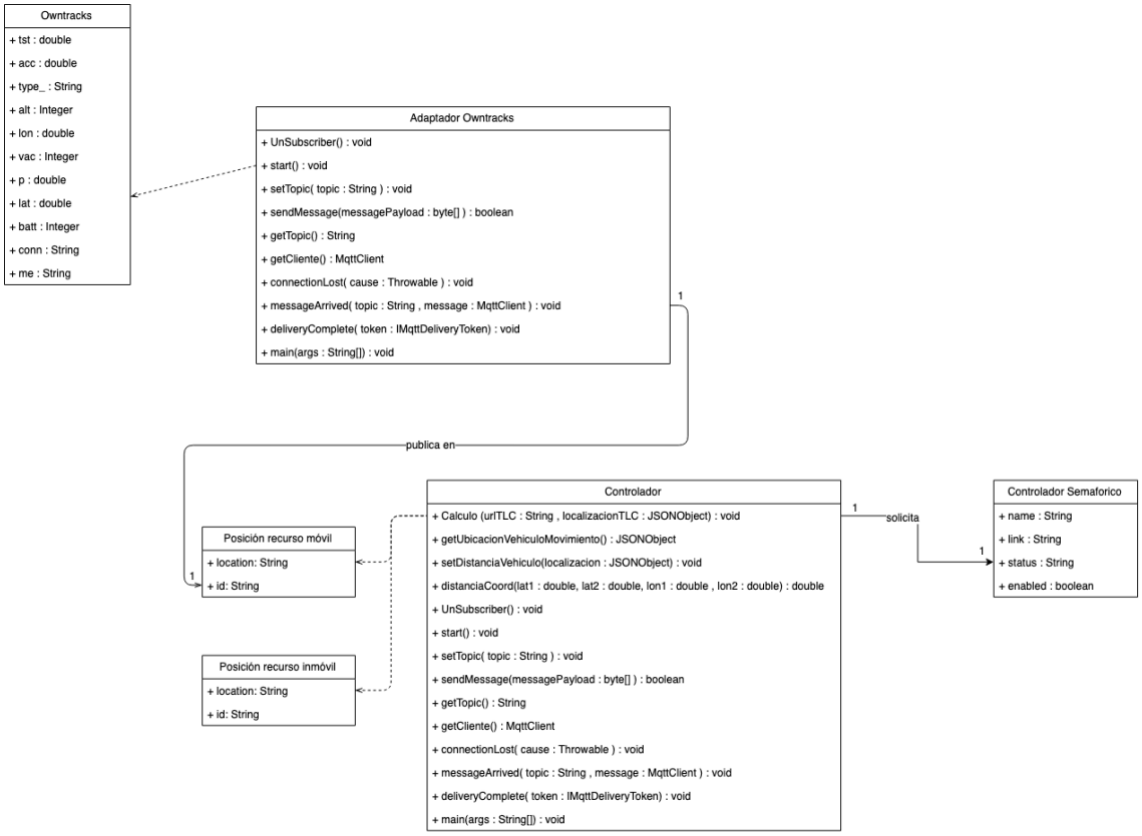

Figura 21: Diagrama de clases del proyecto

4.6 Repercusión de las reglas de interacción geolocalizadas

En el capítulo anterior se expusieron las reglas de interacción geolocalizadas cuyo fin es el de servir de guía para el desarrollo de proyectos IoT. Además, se hizo uso de distintos ejemplos para demostrar su aplicación práctica.

En este capítulo se ha seguido la guía propuesta en el capítulo anterior para la realización del diseño de las interacciones IoT con el fin de cumplir con el objetivo de definir y diseñar una infraestructura donde poner en funcionamiento las reglas definidas.

Para ello se han diseñado dos escenarios que hacen uso de las siguientes reglas de interacción geolocalizadas:

- **Cerca**: Un recurso móvil que se encuentra parado se encuentra a una distancia cercana al recurso IoT del controlador semafórico. De forma que el recurso IoT se mantendrá funcionando sin cambio alguno.
- **Lejos**: Un recurso móvil que se encuentra parado se encuentra a una distancia lejana al recurso IoT del controlador semafórico. De forma que el recurso IoT se mantendrá funcionando sin cambio alguno.
- **Acercándose**: Un recurso móvil que se encuentra en movimiento se encuentra acercándose al recurso IoT del controlador semafórico. De forma que mientras el recurso móvil este fuera del rango de distancia establecido del controlador semafórico, este se mantendrá funcionando sin cambio alguno. Mientras que en el momento en que el recurso móvil este dentro de la distancia establecida del controlador semafórico se procederá a realizar un cambio de estado para otorgar prioridad al vehículo móvil.
- **Alejándose**: Un recurso móvil que se encuentra en movimiento se encuentra alejándose del recurso IoT del controlador semafórico. De forma que mientras el recurso móvil este dentro del rango de distancia establecido del controlador semafórico, este se mantendrá otorgando prioridad al vehículo móvil. Mientras que en el momento en que el recurso móvil este fuera de la distancia establecida del controlador semafórico procederá a realizar un cambio de estado para volver a su estado anterior.
- **Entrar**: Un recurso móvil que se encuentra en movimiento entra en la zona establecida del recurso IoT del controlador semafórico. Por tanto, se procederá un cambio de estado en el controlador semafórico otorgándole prioridad al recurso móvil.
- **Salir**: Un recurso móvil que se encuentra en movimiento sale de la zona establecida del recurso IoT del controlador semafórico. Por tanto, se procederá un cambio de estado en el controlador semafórico volviendo a su estado anterior.

5. Implementación

Este capítulo se centra en detallar los pasos seguidos para la implementación de la arquitectura *software* del proyecto, que se ha descrito a lo largo de los capítulos anteriores.

5.1 Introducción

Para poner en contexto la importancia de este apartado hay que resaltar la curva de aprendizaje requerida para la implementación del proyecto.

El proceso de implementación de los escenarios ha sido posible gracias a la utilización de la librería Eclipse Paho, haciendo uso de esta librería se podrá realizar una conexión al bróker Mosquitto para la realización de los escenarios definidos en los capítulos anteriores.

Posteriormente se hará un análisis detallado de los recursos disponibles en la plataforma/laboratorio de la *Smart City* del grupo de investigación TaTami del centro PROS para elegir con minuciosidad los recursos más adecuados para el desarrollo de los escenarios.

Seguidamente se detallará cronológicamente la implementación de cada uno de los escenarios exponiendo la funcionalidad con detalle, así como sus características.

5.2 Librería Eclipse Paho

Eclipse Paho es un proyecto de código abierto cuyo objetivo es el de proporcionar bibliotecas de clientes MQTT y MQTT-SN para un gran número de lenguajes de programación, como son Java, C++, C#, .NET, Python [26].

Algunas de las características de esta librería son:

- Adecuado para dispositivos de baja capacidad, como sensores instalados en equipos conectados.
- Proporciona calidad de servicios para manejar la conectividad y otros errores, que pueden ser bastante comunes en el caso de automóviles, debido a que se encuentran en movimiento, y la conectividad puede ser un problema.
- Una de las principales ventajas es que se pueden personalizar los formatos de mensajes, lo que permite tanto a fabricantes como usuarios personalizar e innovar soluciones.
- También se encuentra integrado de forma satisfactoria con algunos de los servicios de mensajería más importantes como son WebSphere MQ y ActiveMQ.

5.3 Plataforma/Laboratorio de la Smart City

Este laboratorio está construido teniendo en cuenta los principios que propone la IoT-A (arquitectura de referencia para la internet de las cosas, que intenta establecer un paradigma conceptual y una arquitectura de referencia que posibilite la integración en múltiples y variadas soluciones en el ámbito IoT), a través de un conjunto de maquetas físicas de dispositivos, con los que poder interactuar.

Dado que estas maquetas físicas no siempre están disponibles, el grupo de investigación TaTami ha desarrollado un espacio de computación donde ubicar versiones virtuales de estos dispositivos, con el objetivo de permitir la interactuación de la misma manera que sus homólogas físicas. Siendo la única diferencia entre los dispositivos virtuales y los físicos el identificador utilizado para la realización de la conexión.

Estas maquetas ofrecen varias APIs de interacción que son usadas para desarrollar diferentes servicios.

Este espacio estará siempre disponible y accesible en el servidor http://tambori.dsic.upv.es. Donde se ha desplegado una ciudad inteligente (*Smart City*), que contiene diferentes objetos inteligentes y servicios, que se comunican entre sí para ofrecer algunas funcionalidades.

A continuación, se describen los diferentes dispositivos y servicios que proporciona la *Smart City*, indicando las diferentes APIs que ofrecen, siendo posible utilizar comunicación directa a través de REST, como la utilización de la comunicación indirecta mediante el uso de colas de mensajería MQTT.

Los dispositivos que están disponibles en la *Smart City* son:

- **Vehículos Conectados**: Flota de vehículos inteligentes que son capaces de interactuar / comunicarse con las carreteras, señales de trafico y las incidencias.

- **Edificios Inteligentes**: Se dispone de varias maquetas que implementan, tanto usando tecnología domótica KNX como Arduino, el concepto de edificio inteligente. A diferencia del resto de servicios no ofrece comunicación indirecta de tipo MQTT.

- **Semáforos y Controladores**: Un controlador semafórico (TLC) gestiona el trafico en un cruce entre dos carreteras o calles.

- **Drones**: Presentan múltiples usos debido a su capacidad de movimiento, de disponer de cámaras que facilitan la visualización de manera remota siendo geoposicionados por GPS.

- **Vehículos virtuales de la colonia:** Es un servicio que se ha creado para poder controlar y utilizar vehículos de diferente tipo en el ámbito de la ciudad, pudiendo contar actualmente con cinco vehículos terrestres y tres drones.

5.4 Implementación escenario controlador

Como ya se ha expuesto en el capítulo 4, el propósito de desarrollar la aplicación "Controlador" es el de ejemplificar la solución de un controlador de acciones de interacción geolocalizadas donde se implementará un servicio de "gestión / control" entre un vehículo de emergencia en movimiento y un semáforo con el fin de realizar una comparación de las ubicaciones (acercándose / alejándose, dentro / fuera, cerca / lejos) y producirse acciones en función de estas.

Para ello se hará uso del lenguaje de programación Java por todas las ventajas que ofrece y el poder ejecutar la aplicación en cualquier sistema operativo gracias a la JVM (*Java Virtual Machine*). También se hará uso de la librería Eclipse Paho, para establecer la conexión al bróker Mosquitto y la librería REST para que en función de la posición GPS realizar las acciones pertinentes.

5.4.1 Proceso Main

La clase Main es el ejecutable principal de nuestro proyecto cuyo principal objetivo es el de crear un objeto de tipo Subscriber y posteriormente inicializar la interacción entre las distintas clases que conforman el escenario.

Además de esto, en esta clase se encuentra implementada la funcionalidad para realizar el cálculo de distancia entre dos posiciones GPS.

También se encuentra implementada la funcionalidad para que, en base a la distancia entre el vehículo de emergencias en movimiento y el semáforo, comprobar si se cumple una condición de encontrarse a una distancia de 300m y en ese caso realizar una acción PUT sobre el semáforo para solicitar un cambio de estado a estado de emergencia. En caso de no cumplirse, el semáforo se mantendrá en el funcionamiento tradicional.

Inicialmente necesitaremos proporcionar el controlador de recursos IoT, siendo en este caso un semáforo, para ello utilizaremos el identificador/URL http://tambori.dsic.upv.es:10050/tlc donde si realizamos una acción de tipo GET (aunque se posibilita la opción de utilizar MQTT) obtenemos un JSON como la figura 22:


```
{"name":"tlc3126","link":"/tlc","enabled":true,"status":"CROSS_ROAD2_YELLOW"}
```


Figura 22: Ejemplo petición GET controlador de recurso IoT

Que tiene los atributos siguientes:

- *Name*: Nombre correspondiente al controlador semafórico utilizado.
- *Link*: Nomenclatura del controlador
- *Enabled*: Estado de disponibilidad, pudiendo estar habilitado o deshabilitado
- *Status*: Estado en el que se encuentra el controlador semafórico variando entre los colores verde, amarillo y rojo.

5.4.2 Proceso Subscriber

Esta clase proporciona la posibilidad de suscribirnos a una cola MQTT donde se producirá un evento para recibir información de esta. Siendo esta funcionalidad de gran ayuda para la realización del escenario propuesto.

En el siguiente script podemos apreciar el procedimiento necesario donde primero se declaran los atributos necesarios para llevar a cabo la conexión y el constructor donde se realiza la conexión al bróker.

```
public class Subscriber {

    public static final String BROKER_URL = "tcp://tambori.dsic.upv.es:8885";
    private MqttClient client;
    final String username = "sersanc8";
    final String password = "sersanc8";
    private int qos = 0;

    private String topic = "dgt.es/trafic/vehicle/+/info";
    private String clientId;
    /**
     * Metodo constructor donde se inicializa la conexión con el broker
     */
    Subscriber() {
        MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
        mqttConnectOptions.setAutomaticReconnect(true);
        mqttConnectOptions.setCleanSession(false);
        mqttConnectOptions.setUserName(username);
        mqttConnectOptions.setPassword(password.toCharArray());
        try {
            clientId = UUID.randomUUID().toString();
            client = new MqttClient(BROKER_URL, clientId);
            client.connect(mqttConnectOptions);

        } catch (MqttException ex) {
            Logger.getLogger(Subscriber.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Script 1: Ejemplo conexión al bróker escenario controlador

Una vez realizada la conexión a la dirección del bróker se realizará la suscripción al *topic* correspondiente para recibir la ubicación del recurso móvil siendo en este caso la ambulancia (como se aprecia en el script 2).

Para obtener la información relativa a la ubicación GPS del recurso en movimiento se utilizará el protocolo de comunicación MQTT con el *topic* correspondiente al vehículo de emergencia en movimiento que es: “dgt.es/trafic/vehicle/+/info”.

```java
/**
 * Metodo para realizar la suscripción al topic
 */
public void start() {
    try {
        client.setCallback(new SubscribeCallback());
        client.subscribe("dgt.es/trafic/vehicle/+/info");
    } catch (MqttException e) {
        System.exit(1);
    }
}
```

Script 2: Ejemplo suscripción al topic escenario controlador

5.4.3 Proceso SubscriberCallback

Esta clase se invoca desde la clase Subscriber, posibilitándonos la opción de:

- Recibir los mensajes a un *topic* suscrito, siendo en este caso los mensajes proporcionados a través del topic “dgt.es/trafic/vehicle/+/info” que encontrarán en formato JSON.
- Notificar, que hemos mandado un mensaje (al igual que en el caso de recibir mensajes, en formato JSON utilizando la estructura propuesta por Owntracks)
- Notificar, que la conexión se ha perdido.

La ubicación del recurso móvil la recibiremos en un JSON como el siguiente:

```
{"location":{"lng":-0.3374,"lat":39.481106},"id":"ambulancia2"}
```

Figura 23: Información recurso móvil

Se creará un objeto de tipo Main, en el que se realizará una llamada al método setDistanciaVehiculo donde mandará la posición para realizar la comparación entre las distintas ubicaciones y realizar el cálculo de la distancia. En esta situación se presentan dos posibilidades, por un lado, si el cálculo de la distancia es igual o menor que 300m se solicitara a la API de tipo REST la realización de una petición de tipo PUT al controlador de recursos IoT para la realización de un cambio de estado en este, con el fin de facilitar la interacción con el recurso en movimiento.

De esta forma si volvemos a realizar una petición de tipo GET podemos comprobar que el estado del controlador de recurso IoT ha cambiado para facilitar la interacción con el recurso en movimiento que en este caso es el vehículo de tipo ambulancia. Obteniendo un resultado como el siguiente:

```
{"name":"tlc1049","link":"/tlc","enabled":true,"status":"CROSS_EMERGENCY"}
```

Figura 24: Resultado petición REST de tipo PUT

Por otro lado, si el cálculo de la distancia entre el semáforo y el vehículo de tipo ambulancia obtiene un resultado mayor que 300m, se mantendrá el estado del semáforo (pudiendo estar en verde, amarillo o rojo, dependiendo del instante), mientras que si el vehículo en movimiento ha pasado anteriormente por el semáforo haciendo que cambiara su estado, se volverá a realizar una petición de tipo PUT para solicitar al controlador de recurso IoT que el controlador semafórico vuelva a funcionar con normalidad. Obteniendo un resultado como el siguiente:

{"name":"tlc1049","link":"/tlc","enabled":true,"status":"CROSS_INIT"}

Figura 25: Resultado petición REST de tipo PUT en caso de condición errónea

5.5 Implementación escenario adaptador owntracks

5.5.1 Proceso AdaptadorOwntracks

La clase AdaptadorOwntracks es el ejecutable principal de nuestro proyecto cuyo principal objetivo es el de crear un objeto de tipo Subscriber y posteriormente iniciar la interacción entre las distintas clases que conforman el escenario.

5.5.2 Proceso Subscriber

Esta clase presenta dos posibilidades, por una parte, nos ofrece la opción de suscribirnos a una cola MQTT donde se producirá un evento para recibir información de esta, además de que, por otra parte, nos posibilita la opción de publicar información en una cola MQTT. De forma que, tanto una como otra funcionalidad resulta de gran ayuda para la realización del escenario propuesto.

Inicialmente, necesitaremos utilizar una fuente de posiciones GPS proporcionada en este caso por el *software* owntracks, mediante el uso de la librería Eclipse Paho. En la siguiente imagen podemos apreciar el procedimiento imprescindible donde primero se declaran los atributos necesarios para llevar a cabo la conexión y el constructor donde se realiza la conexión al bróker. Este procedimiento similar al de la implementación anterior, pero variando algunos detalles clave como es el *topic* al que se realiza la suscripción.

```
public class Subscriber {

    public static final String BROKER_URL = "tcp://tambori.dsic.upv.es:8885";
    private MqttClient client;
    final String username = "sersanc8";
    final String password = "sersanc8";
    private int qos = 0;
    private String topic = "dgt.es/trafic/vehicle/ambulancia2/info";
    private String subscribe = "owntracks/#";
    String clientId;

    public void UnSubscriber() {...8 lines }
    /**
     * Constructor para inicializar la conexión al broker MQTT
     */
    Subscriber() {
        MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
        mqttConnectOptions.setAutomaticReconnect(true);
        mqttConnectOptions.setCleanSession(false);
        mqttConnectOptions.setUserName(username);
        mqttConnectOptions.setPassword(password.toCharArray());
        try {
            clientId = UUID.randomUUID().toString();
            client = new MqttClient(BROKER_URL, clientId);
            client.connect(mqttConnectOptions);

        } catch (MqttException ex) {
            Logger.getLogger(Subscriber.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Script 3: Ejemplo conexión al bróker escenario adaptador owntracks

Una vez se realizada la conexión con el bróker se realizará la suscripción a una cola MQTT comunicada mediante MQTT con el *topic* correspondiente a owntracks que es: “owntracks/<device>/<name>” siendo en el caso concreto del ejemplo la utilización del *topic* “owntracks/sersanc8/iPhoneSergi”. Como se puede apreciar en el siguiente script:

```
/**
 * Metodo donde se inicia la conexión. Primero se conecta al BROKER_URL y
 * posteriormente se realiza la conexión con la configuración establecida.
 * Para finalmente subscribirse al topic deseado, en este caso owntracks/#
 */
public void start() {

    try {

        client.setCallback(new SubscribeCallback());
        client.subscribe(subscribe);

    } catch (MqttException e) {
        System.exit(1);
    }

}
```

Script 4: Ejemplo suscripción al topic escenario adaptador owntracks

Una vez se haya suscrito con éxito, se inicializará el evento donde se recibirá el mensaje, para ello se accederá a la clase SubscribeCallback.

5.5.3 Proceso SubscriberCallback

Esta clase se invoca desde la clase Subscriber, posibilitándonos la opción de:

- Recibir los mensajes a un *topic* suscrito siendo en este caso los mensajes proporcionados a través del topic “owntracks/sersanc8/iPhoneSergi”. Siempre en formato JSON, y utilizando la estructura definida por Owntracks.

- Notificar, que hemos mandado un mensaje (al igual que en el caso de recibir mensajes en formato JSON)

- Notificar, que la conexión se ha perdido.

En el caso particular correspondiente al escenario propuesto, recibiremos un mensaje proporcionado por Owtracks con la siguiente estructura:

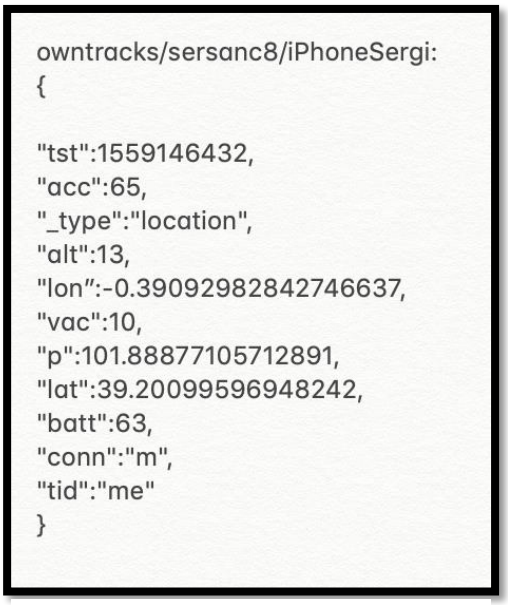

Figura 26: Ejemplo formato proporcionado por owntracks

Donde se recibirán estos atributos:

- tst: Marca de tiempo
- acc: Exactitud de la ubicación reportada en metros sin unidad.
- _type: Describe la ubicación del dispositivo.
- alt: Altitud medida sobre el nivel del mar.
- lon: Siglas correspondientes a la longitud
- vac: Exactitud vertical.
- p: Ping emitido
- lat: Siglas correspondientes a la latitud
- batt: Nivel de batería del dispositivo.
- conn: Estado de conectividad a internet.
- tid: ID de rastreador utilizado para mostrar las iniciales de un usuario.

El problema que se presenta es que el JSON recibido por la fuente de posiciones GPS no coincide con el formato utilizado por la aplicación donde se pretende utilizar.En ese mismo instante se aplicará la transformación al mensaje para que se ajuste a una nueva fuente de datos.

Para ello, se utilizará la librería de JSON para la modificación y creación de objetos de tipo JSON, donde se extraerán los atributos necesarios para la nueva fuente de posiciones de la fuente de posiciones GPS como se aprecia en el siguiente script:

```
20      /**
21       * Es llamada cuando se recibe un mensaje, al estar suscrito a un topic del
22       * broker MQTT
23       * @param topic
24       * @param message
25       */
26      @Override
        public void messageArrived(String topic, MqttMessage message) {
28          cliente = new Subscriber();
29
30          JSONObject json = new JSONObject();
31          JSONObject mensaje = new JSONObject(message.toString());
32          JSONObject localizacion = new JSONObject();
33
34          if (mensaje.get("tid").equals("SE")) {
35              Object lat = mensaje.get("lat");
36              Object lng = mensaje.get("lon");
37
38              localizacion.put("lat", lat);
39              localizacion.put("lng", lng);
40
41              json.put("id", topic);
42              json.put("location", localizacion);
43
44              cliente.sendMessage(json.toString().getBytes());
45          }
46      }
```

Script 5: Fragmento codigo para adaptar formato JSON

Dando como resultado una estructura como la siguiente:

```
{
    id: owntracks/sersanc8/iPhoneSergi,
    location: {
        lat: 39.200995969482,
        lng:-0.39092982842746637
    }
}
```

Figura 27: Ejemplo mensaje transformado

Donde se incorporarán estos atributos:

- id: Es un elemento identificador, cuya función es la de recuperar datos de una entidad en concreto. Se utiliza como nombre a la hora de recuperar los datos.

- location: Posición física detectada mediante GPS. Es representado mediante longitud y latitud:

- lat: Siglas correspondientes a la latitud
- lng: Siglas correspondientes a la longitud

Posteriormente será necesario crear un objeto de tipo Subscribe y mandar el mensaje transformado, siendo en este caso la fuente de datos correspondiente al vehículo en movimiento ubicado en el *topic* “dgt.es/trafic/vehicles/ambulance2/info”.

Para realizar la publicación se volverá a hacer uso de la librería Eclipse Paho, mediante el método sendMessage que se ha desarrollado anteriormente, cuyo fin es el de mandar un mensaje a un *topic* deseado siendo en este caso “dgt.es/trafic/vehicle/ambulancia2/info” (como se aprecia en el script 6).

```
/**
 * Metodo para el envio de mensajes
 * @param messagePayload
 * @return boolean
 */
public boolean sendMessage(byte[] messagePayload) {
    System.out.println("llego a sendMessage");
    boolean res = false;
    if (client != null) {
        try {
            MqttMessage message = new MqttMessage(messagePayload);
            message.setQos(qos);
            message.setRetained(false);
            client.publish(topic, message);
            res = true;
        } catch (MqttException e) {
            System.err.print("Failed to send outbound message to topic:  "+
                topic + "- unexpected issue: "+ new String(e.toString()));
        }
    } else {
        System.out.println("estoy vacio");
    }
    return res;
}
```

Script 6: Ejemplo código para mandar mensajes

5.6 Problemas encontrados

5.6.1 Escenario Adaptador Owntracks

El problema surge en la manera de tratar los mensajes proporcionados por la cola de mensajería para realizar la transformación al nuevo formato utilizado por la fuente de datos y su posterior publicación.

Inicialmente la forma de tratar el mensaje proporcionado por la cola de mensajería MQTT, fue utilizando métodos de la clase String, procesar el mensaje, y posteriormente publicar la cadena en la nueva fuente de datos.

Uno de los principales problemas que presentaba esta forma de tratar los mensajes era su incompatibilidad para procesar los distintos tamaños de las cadenas de String.

Se intento solucionar este problema mediante la utilización de la clase JSONObject que es una clase proporcionada por la API de java para la correcta creación y manipulación de objetos de tipo JSON.

Otro problema era que no se publicaba de forma correcta el nuevo mensaje procesado, de forma que no era recibido por la aplicación “Controlador” debido a que se creaba un nuevo objeto de tipo Subscriber al recibir un mensaje, lo que provocaba un bucle infinito.

Mientras que, en la problemática referente a la publicación de mensajes en la cola MQTT se declaro un objeto de tipo Subscribe que se inicializaba cuando se recibía el mensaje y no se volvía a establecer la conexión como anteriormente.

El código mostrado a continuación ejemplifica el tratamiento utilizado para tratar los mensajes recibidos por la fuente de datos y transformarlos a un nuevo formato.

```
20      /**
21       * Es llamada cuando se recibe un mensaje, al estar suscrito a un topic del
22       * broker MQTT
23       * @param topic
24       * @param message
25       */
26      @Override
        public void messageArrived(String topic, MqttMessage message) {
28          cliente = new Subscriber();
29
30          JSONObject json = new JSONObject();
31          JSONObject mensaje = new JSONObject(message.toString());
32          JSONObject localizacion = new JSONObject();
33
34          if (mensaje.get("tid").equals("SE")) {
35              Object lat = mensaje.get("lat");
36              Object lng = mensaje.get("lon");
37
38              localizacion.put("lat", lat);
39              localizacion.put("lng", lng);
40
41              json.put("id", topic);
42              json.put("location", localizacion);
43
44              cliente.sendMessage(json.toString().getBytes());
45          }
46      }
```

Script 7: Recepción de mensajes del escenario adaptador owntracks

Por tanto, una vez solucionado el problema correspondiente a la adaptación del formato de datos, se ha podido realizar la publicación a la cola MQTT de forma satisfactoria.

5.6.2 Escenario Controlador

En este escenario se produce un problema al recibir mensajes de distíntas fuentes de datos, debido a que para probar su correcto funcionamiento se utilizo la estructura estática siguiente:

```
{id: ambulancia2, location: {lat: 39.481106, lng:-0.33740} }
```

Figura 28: Ejemplo mensaje estático propuesto

De forma que se seleccionaban los elementos requeridos de ese mensaje mediante el uso de métodos de la clase String.

El problema surge cuando se manda un objeto de tipo JSON con esa estructura, debido a que por defecto JSON ordena los mensajes de forma que lo que se recibía era un mensaje como este:

```
{"location":{"lng":-0.3374,"lat":39.481106},"id":"ambulancia2"}
```

Figura 29: Ejemplo mensaje JSON

Lo cual suponía un problema por que se procesaban de forma incorrecta los atributos lng (longitud) y lat (latitud) dando errores significativos en el cálculo de la distancia al realizar la comparación entre el vehículo en movimiento y el semáforo.

Se intentó solucionar este problema mediante la utilización de la clase JSONObject para manipular los mensajes recibidos. JSONObject es una clase proporcionada por la API de java para la correcta creación y manipulación de objetos de tipo JSON.

El código mostrado a continuación ejemplifica el tratamiento utilizado para tratar los mensajes recibidos por la fuente de datos mediante la utilización de la clase JSONObject.

```
23      /**
24       * Es llamada cuando se recibe un mensaje, al estar suscrit a un topic del broker MQTT
25       * @param topic
26       * @param message
27       */
28      @Override
        public void messageArrived(String topic, MqttMessage message) {
30          messageComparador = topic.substring(0, 14);
31          if (messageComparador.equals("dgt.es/trafic/")) {
32              JSONObject mensaje = new JSONObject(message.toString());
33              JSONObject localizacion = new JSONObject(mensaje.get("location").toString());
34
35              Main m = new Main();
36              m.setDistanciaVehiculo(localizacion);
37          }
38      }
```

Script 8: Lógica para la recepción de mensajes del escenario controlador

Por tanto, el problema se ha solucionado de forma satisfactoria al cambiar la forma en tratar el mensaje recibido en la clase SubscriberCallback, de esta forma ahora todos los mensajes se procesarán de la misma forma y el cálculo de la ubicación entre las dos posiciones se realiza satisfactoriamente.

5.6.3 Comunicación entre escenarios

La comunicación (es decir, que los dos escenarios propuestos trabajen de forma conjunta) es uno de los objetivos principales del proyecto.

De forma que el “Controlador de acciones de interacción geolocalizadas” utilizará los recursos proporcionados por el “Adaptador fuente de posiciones GPS” y con el controlador de recursos IoT, a su vez que la ubicación del recurso inmóvil se llevará a cabo el proceso de comparación de las ubicaciones y solicitar acciones sobre el recurso inmóvil.

El problema se produce al ejecutar ambos escenarios, debido a que se producía un error y se terminaba de manera abrupta la ejecución. Especificando más concretamente, el problema se produce en el momento de realizar la inicialización del bróker MQTT.

Se intento solucionar mediante la utilización del método randomUUID de la clase UUID en el proceso de inicializar el objeto de tipo ClientMQTT. Este método se utiliza para recuperar un UUID de tipo 4 lo que significa que el UUID se produce de forma aleatoria, esta solución evitara que se reproduzca el mismo valor en el clientID (ya que esta secuencia se utiliza para que el servidor almacene datos relacionados con el cliente).

El código mostrado a continuación ejemplifica la solución propuesta:

```
36      /**
37       * Constructor para inicializar la conexión al broker MQTT
38       */
39      Subscriber() {
40          MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
41          mqttConnectOptions.setAutomaticReconnect(true);
42          mqttConnectOptions.setCleanSession(false);
43          mqttConnectOptions.setUserName(username);
44          mqttConnectOptions.setPassword(password.toCharArray());
45          try {
46              clientId = UUID.randomUUID().toString();
47              client = new MqttClient(BROKER_URL, clientId);
48              client.connect(mqttConnectOptions);
49
50          } catch (MqttException ex) {
51              Logger.getLogger(Subscriber.class.getName()).log(Level.SEVERE, null, ex);
52          }
53      }
```

Script 9: Funcionalidad metodo subscriber

Por tanto, el problema se ha solucionado permitiendo la ejecución tanto del escenario “Controlador” como del escenario “Adaptador Owntracks”

5.7 Impacto de las reglas de interacción geolocalizadas

En los capítulos anteriores se definieron las reglas de interacción geolocalizadas y se expuso la relevancia que presentaba en el diseño utilizado en los escenarios propuestos a lo largo de este documento.

Es importante destacar el impacto que produce en el ámbito de la implementación las reglas de interacción geolocalizadas propuestas para hacer posible que se apliquen en los escenarios desarrollados.

Para ello, ha sido necesario definir un método que se encarga de calcular la distancia entre el vehículo de emergencias en movimiento y el semáforo. De forma que comprueba que se cumple de forma satisfactoria las reglas de interacción geolocalizadas.

Además, también será necesario realizar un método que se encargue de realizar las acciones pertinentes en el recurso IoT en función de la distancia obtenida. Como puede observarse en la siguiente imagen:

```
/**
 * Metodo para calcular la distancia entre dos coordenadas
 *
 * @param lat1
 * @param lat2
 * @param lng1
 * @param lng2
 * @return double
 */
public static double distanciaCoord(double lat1, double lng1, double lat2, double lng2) {

    double radioTierra = 6371;//en kilómetros
    double dLat = Math.toRadians(lat2 - lat1);
    double dLng = Math.toRadians(lng2 - lng1);
    double sindLat = Math.sin(dLat / 2);
    double sindLng = Math.sin(dLng / 2);
    double va1 = Math.pow(sindLat, 2) + Math.pow(sindLng, 2)
            * Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2));
    double va2 = 2 * Math.atan2(Math.sqrt(va1), Math.sqrt(1 - va1));
    double distancia = radioTierra * va2;

    return distancia * 1000;
}
```

Script 10: Método para calcular la distancia entre dos coordenadas

En la imagen anterior se observa como este método utiliza el radio del planeta, la longitud y latitud de dos puntos distintos para realizar el cálculo de la distancia de manera precisa.

Sera necesario hacer uso de la librería JSONObject para la modificación y creación de objetos de tipo JSON, se extraerán los atributos necesarios del semáforo y del vehículo de emergencias en movimiento para obtener su posición GPS. Como puede observarse en la siguiente imagen:

```
/**
 * Metodo que se utiliza para realizar las acciones pertinentes en base
 * a la distancia entre el recurso móvil y el recurso IoT
 *
 * @param urlTLC
 * @param localizacionTLC
 */
public void Calculo(String urlTLC, JSONObject localizacionTLC) {
    double latitudVehiculo = Double.parseDouble(ubicacionVehiculoMovimiento.get("lat").toString());
    double longitudVehiculo = Double.parseDouble(ubicacionVehiculoMovimiento.get("lng").toString());
    double latitudTLC = Double.parseDouble(localizacionTLC.get("lat").toString());
    double longitudTLC = Double.parseDouble(localizacionTLC.get("lng").toString());

    System.out.println("\nLatitudVehiculo: " + latitudVehiculo);
    System.out.println("LongitudVehiculo: " + longitudVehiculo);
    System.out.println("LatitudTLC: " + latitudTLC);
    System.out.println("LongitudTLC: " + longitudTLC);

    double distancia = distanciaCoord(latitudVehiculo, longitudVehiculo, latitudTLC, longitudTLC);
```

Script 11: Uso de la librería JSONObject

En base a la posición GPS obtenida, se procederá a hacer uso del método definido para calcular la distancia, el cual nos devolverá la distancia entre los dos objetos.

De esta forma, en función de la distancia obtenida, se realizará la acción pertinente sobre el controlador semafórico; en caso de que el vehículo en estado de emergencias se encuentre a una distancia menor de 300m, se cambiará el estado del semáforo para otorgarle prioridad o se mantendrá en su estado actual sin necesidad de realizar ninguna acción sobre el controlador.

```
if (distancia <= 300) {
    try {
        String payload = "{action:emergency}";
        ClientResource resource = new ClientResource("http://tambori.dsic.upv.es:10050/tlc");
        System.out.println("Prueba : "+new JSONObject(resource.get().getText()));
        resource.put(payload);
        String response;
        try {
            response = resource.get().getText();
            if (response != null) {
                System.out.println(response);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (IOException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
} else {
    String payload = "{action:init}";
    ClientResource resource = new ClientResource("http://tambori.dsic.upv.es:10050/tlc");
    resource.put(payload);
    String response;
    try {
        response = resource.get().getText();
        if (response != null) {
            System.out.println(response);
        }
    } catch (IOException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Script 12: Acción en base a la distancia

6. Pruebas

Este capítulo se centrará en la realización de las pruebas pertinentes para la comprobación del correcto funcionamiento de la solución implementada.

6.1 Introducción

Las pruebas de *software* pueden definirse como la verificación dinámica del comportamiento de un programa contra el comportamiento esperado, usando un conjunto finito de casos de prueba, seleccionados de manera adecuada. Lo cual resulta un factor critico para determinar la calidad del *software*.

La importancia de la realización de pruebas en el desarrollo de *software* reside en que en cualquier etapa del ciclo de vida es posible que aparezcan errores, los cuales pueden permanecer sin ser descubiertos.

Los objetivos principales para la realización de las pruebas son:

- Detectar un error especifico.
- Descubrir errores no descubiertos anteriormente.
- Determinar un caso de prueba adecuado.

Un caso de prueba se compone de un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular [27].

Existen diversos tipos de pruebas, pero se pueden clasificar según la forma en que se ejecutan en:

- **Pruebas manuales**: Son realizadas por el usuario paso a paso.
- **Pruebas automáticas**: Consiste en la utilización de un *software* alternativo al usado para la comprobación de los resultados obtenidos y los esperados. La ventaja que presenta respecto a las pruebas manuales es que permite realizar pruebas que podrían ser tediosas o de una mayor dificultad.

Una prueba funcional es un tipo prueba cuyo objetivo es la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el *software*. Hay distintos tipos como:

- **Pruebas Unitarias**: Este tipo de pruebas tienen como objetivo asegurar que cada unidad funcione de forma correcta y eficiente por separado. Además de que el código se comporta respecto a las expectativas que se esperan de el, verificando que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, además si el estado inicial es válido, entonces el estado final es válido.

- **Pruebas de Integración**: Consiste en la realización de pruebas específicas, concretas y exhaustivas con el fin probar y validar que el comportamiento del *software* es el que se ha especificado.

- **Pruebas del Regresión**: Son un tipo de pruebas de *software* que tienen como objetivo descubrir *bugs* (errores), carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del *software*. Esto se produce debido a la realización de un cambio en el programa. Donde se evalúa el correcto funcionamiento del *software* desarrollado respecto a evoluciones o cambios funcionales.

- **Prueba de Componentes**: Tienen por objetivo localizar defectos y comprobar el funcionamiento de módulos *software*, programas, objetos, clases, etc. Que puedan probarse por separado; es decir, se pueden realizar de manera independiente al resto del sistema en función del contexto.

En este proyecto se ha optado por la utilización de pruebas unitarias por las ventajas que ofrece al posibilitar el análisis de cada módulo del código de forma individual, ayudando a detectar algunos errores concretos.

6.2 Pruebas unitarias

Una prueba unitaria tiene como objetivo comprobar el correcto funcionamiento de una unidad de código.

Además, como se ha expuesto anteriormente se verifica que el código cumple con la funcionalidad esperada, se realiza la verificación para comprobar que sea correcto el nombre, los tipos de los parámetros, el tipo que se devuelve, si el estado inicial es valido y si el estado final también lo es.

Un ejemplo de la utilización de este tipo de pruebas sería que en el diseño estructurado o funcional se realizará la prueba a una función o un procedimiento, mientras que si fuera necesario hacer una prueba a un diseño orientado a objetos se realizaría una prueba a una clase [28].

6.2.1 JUnit

JUnit es un conjunto de bibliotecas desarrollado por Erich Gamma y Kent Beck cuyo objetivo en el ámbito de la programación es la realización de pruebas unitarias en aplicaciones desarrolladas haciendo uso del lenguaje de programación Java.

JUnit es un *framework* (conjunto de clases) que posibilita la opción de realizar la ejecución de clases Java de una manera controlada con el fin de poder evaluar si el funcionamiento realizado por cada uno de los métodos de la clase se comporta como se espera. En función del valor de entrada proporcionado, se evalúa el valor de retorno esperado, si la clase cumple satisfactoriamente con la especificación.

JUnit devolverá un *true* indicando que se ha pasado de forma exitosa la prueba. En caso contrario el valor devuelto por JUnit será un *false* indicando un error en el método correspondiente [29].

6.2.2 Pruebas Escenario Adaptador Owntracks

Una vez expuesto los distintos tipos, especificando más concretamente como funcionan las pruebas unitarias y que herramienta es apropiada utilizar, se ha optado por la utilización de estas en el escenario del adaptador owntracks para certificar su correcto funcionamiento.

Para ello se ha realizado un caso de prueba a cada método de cada clase.

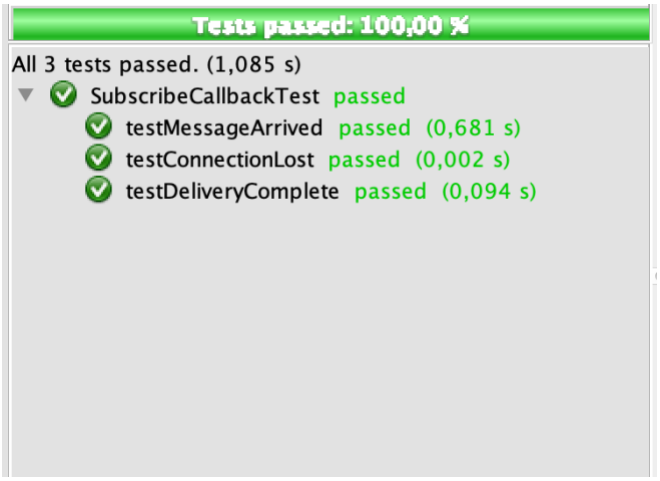

Figura 30: Demostración test satisfactorio clase SubscriberCallback

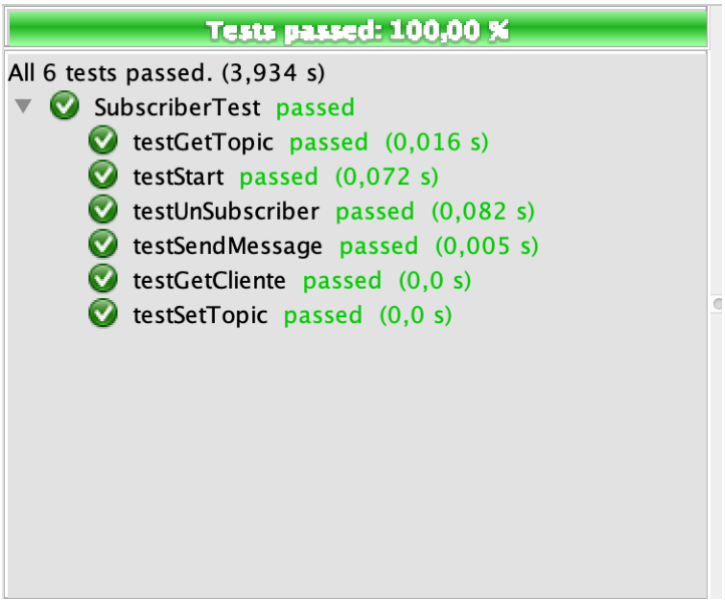

Figura 31: Demostración test satisfactorio clase Subscriber

Figura 32: Demostración test satisfactorio clase AdaptadorOwntracks

6.2.3 Pruebas Escenario Controlador

Al igual que en el escenario anterior se ha realizado un caso de prueba a cada método de cada clase.

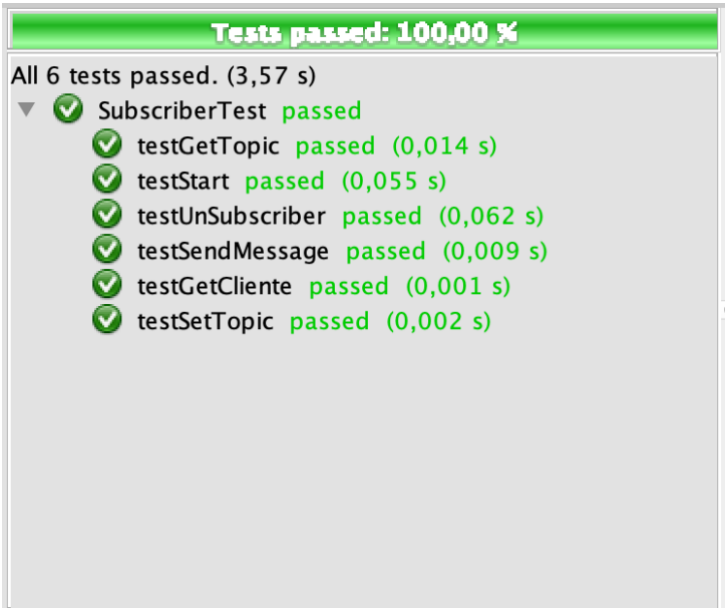

Figura 33: Demostración test satisfactorio clase Subscriber

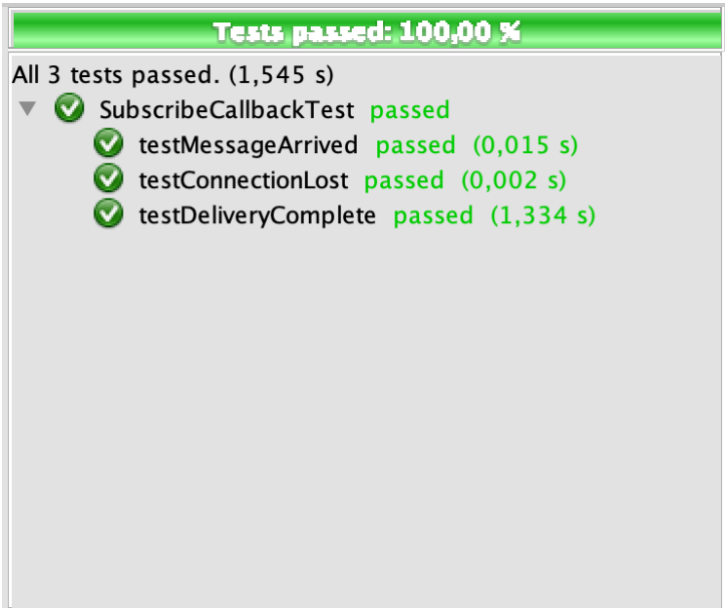

Figura 34: Demostración test satisfactorio clase SubscribeCallback

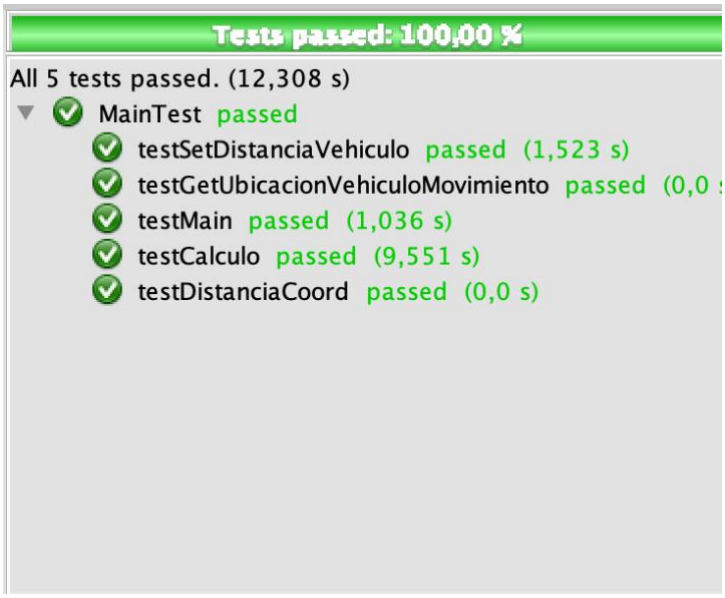

Figura 35: Demostración test satisfactorio clase Main

6.3 Conclusión pruebas

Uno de los aspectos más importantes en el momento de desarrollar un proyecto *software* es la fase de pruebas, ya que como se ha comentado en la introducción de este capítulo, en cualquier etapa del ciclo de vida pueden producirse errores, los cuales no siempre son detectados.

La importancia de las pruebas reside en que son una medida de la calidad del software desarrollado. Por tanto, como se ha podido comprobar se han realizado las pruebas pertinentes en un entorno critico con el fin de solucionar los errores presentes en el proyecto para poder utilizarlo de forma satisfactoria.

7. Conclusión

En este capítulo se expresan las conclusiones del desarrollo del proyecto realizado. Además de tratar un resumen del trabajo realizado a lo largo del proyecto, también se propone una visión de futuro presentando posibles proyectos que pueden derivar del actual. Finalmente se presenta también la valoración personal sobre el proyecto.

7.1 Resumen del trabajo realizado

El aumento de la tendencia cada vez más creciente por estar conectados, hace que se posibilite un aumento del número de dispositivos conectados a internet formando una red conocida como IoT (*Internet of Things*).

Por tanto, siguiendo con los objetivos propuestos en la introducción de este documento, se han identificado los problemas más reseñables que se producen al interpretar las posiciones GPS proporcionadas por los recursos IoT y se ha proporcionado una solución de diseño e implementación a estos problemas.

Posteriormente, se han definido unas reglas de interacción geolocalizadas cuya finalidad es la de definir unas reglas para resolver los problemas que se producen cuando varios recursos IoT se encuentran en una localización aproximada y debe producirse una interacción entre ellos.

A continuación, se han diseñado unas soluciones haciendo uso de las reglas de interacción geolocalizadas definidas anteriormente.

Finalmente, se han ejemplificado estas soluciones realizando dos escenarios en el ámbito de las *Smart Cities* haciendo uso de los recursos proporcionados por la plataforma / laboratorio de la *Smart City* del grupo de investigación TaTami del centro PROS.

Por otra parte, se ha utilizado el entorno de desarrollo Eclipse en su versión Pothon, haciendo uso del lenguaje de programación Java para la realización de los escenarios propuestos.

Para ello se ha elegido diseñar un controlador cuyo propósito sea el de ejemplificar una interacción geolocalizada mediante el control de un semáforo en función de la proximidad de un vehículo de emergencia como son las ambulancias, policías y bomberos. Siendo las funcionalidades presentes que al iniciarse el controlador se le proporcionará la información correspondiente al controlador de recurso IoT del semáforo haciendo uso de la API accesible de tipo REST, mientras que la ubicación del recurso inmóvil se proporcionará también a través de una API de tipo REST. El controlador se suscribirá a una cola MQTT donde recibirá las posiciones del recurso móvil. En base a estas informaciones se calculará la distancia entre el recurso móvil y el recurso inmóvil. Cuando la distancia sea menor a 300 metros, se hará uso de una sentencia de tipo REST, en concreto PUT en el controlador del recurso IoT proporcionada al iniciarse el controlador con el fin de cambiar el estado en el que se encuentra el recurso inmóvil.

Para la solución adaptador fuente de posiciones GPS se ha decidido desarrollar una herramienta que tiene como objetivo implementar una adaptación de la fuente de datos GPS proporcionada haciendo uso del *software* Owntracks al servicio de “control / gestión” de la aplicación controlador. Para ello será necesario un controlador de recursos IoT que disponga de una API accesible (siendo de tipo REST o de tipo MQTT) donde se le

proporcionará la fuente de posiciones GPS (en este caso se suscribirá a una cola MQTT donde Owntracks proporciona la posición GPS) que deberá ser transformada a un nuevo formato fuente y publicarse en la nueva cola MQTT.

Como resultado, se han obtenido dos escenarios que consiguen demostrar las soluciones que se han definido y diseñado.

7.2 Trabajo futuro

Existen algunas mejoras que podrían producirse en trabajos futuros como es la mejora de las reglas de interacción geolocalizadas con la definición de otras posibles interacciones geolocalizadas o la implementación de nuevos escenarios.

Por ejemplo, podría resultar interesante la implementación de un nuevo escenario para cerciorar una de las múltiples soluciones que ofrece la solución propuesta. Se podría implementar un escenario donde un controlador de recurso IoT (el cual nos proporcionará una API de tipo REST o MQTT para su consulta) que podría ser perfectamente un vehículo averiado se encuentre esperando atención, en ese instante un recurso móvil (que se recomienda que proporcione su ubicación mediante una API de tipo REST o que publique sus ubicaciones en una cola MQTT) que vaya a atender la emergencia, el cual podría ser una ambulancia, notificara al vehículo averiado la distancia a la que se encuentra mediante una acción de tipo REST.

Otro posible ejemplo, podría ser la implementación de un nuevo escenario donde un recurso en movimiento, que podría ser un vehículo en movimiento, ceda el paso a otro recurso móvil como son los vehículos especiales en estado de emergencia. Para ello en el momento en que se cumpla la condición de distancia entre los dos vehículos se notificara al vehículo en movimiento que debe ceder el paso.

7.3 Valoración personal

Este trabajo ha supuesto para mi un cambio en la manera de pensar e interpretar el desarrollo de aplicaciones *software*. Desde el inicio cuando elegí el proyecto me sorprendió enormemente el concepto de las *Smart Cities* y todo lo relacionado con el internet de las cosas, algo de lo que no tenia ningún tipo de conocimiento hasta que me adentré en el desarrollo de este proyecto.

A lo largo del proyecto he ido descubriendo la importancia que presenta el internet de las cosas en la sociedad, siendo presente cada vez más en nuestro día a día con cosas cuotidianas como es la domótica en los hogares, la gestión ambiental en las ciudades o la gestión de trafico.

Este proyecto esta ciertamente relacionado con la asignatura de integración de aplicaciones la cual al no ser de la especialidad que he cursado ha dificultado enormemente la realización del proyecto debido a que carecía de los conocimientos básicos que se imparten en esa asignatura. Siendo uno de los aspectos de más dificultad desde el principio el entender como funcionan y utilizar correctamente todas las tecnologías necesarias para diseñar las soluciones y plantear los distintos escenarios que se han desarrollado.

De esta forma, he dedicado todo mi esfuerzo y dedicación para intentar realizar de la mejor manera este trabajo intentando plasmar algunos de los aspectos más importantes que he aprendido a lo largo de la carrera que no son otros que dedicación, constancia y trabajo.

Como resultado, se ha conseguido desarrollar dos escenarios que consiguen escenificar las reglas de interacción geolocalizadas definidas que se han ido proponiendo a lo largo del proyecto.

A nivel personal, considero que el proyecto ha sido beneficioso para mi, ya que gracias a este he podido aprender a interpretar el desarrollo de aplicaciones de una forma que hasta ahora no había creído posible, además de la gran cantidad de tecnologías que no había utilizado hasta la fecha y que gracias a este trabajo he podido utilizar como es el protocolo de mensajería MQTT, el estilo arquitectónico *software* REST, la aplicación owntracks que ha sido de gran utilidad para el desarrollo del escenario adaptador owntracks entre muchos otros.

Obviamente, todo esto no hubiera sido posible sin la ayuda por parte de mis tutores por el apoyo y paciencia para explicarme que debía hacer en cada momento y orientarme de la mejor manera posible.

Bibliografía

[1] IoT: ¿qué esperar en 2019?, 2019. Disponible en: https://www.computerworld.es/tendencias/iot-que-esperar-en-2019

[2] NADAL, M., 2018, Siete formas en que el internet de las cosas cambia nuestro trabajo. *EL PAÍS RETINA* [online]. 2018. Disponible en: https://retina.elpais.com/retina/2018/01/22/talento/1516639374_582456.html

[3] ¿Qué es una Smart City? Top 5 ciudades inteligentes | Sostenibilidad para todos. *Sostenibilidad.com* [en línea]. Disponible en : https://www.sostenibilidad.com/construccion-y-urbanismo/que-es-una-smart-city-top-5-ciudades-inteligentes/

[4] Smart City - Building Tomorrow's Cities. (2018). [Imagen]. Disponible en: https://i1.wp.com/energiahoy.com/wp-content/uploads/2018/09/SMART-CITY-e1537548425551.png?fit=760%2C432&ssl=1

[5] JOSKOWICZ, José. *Reglas y prácticas en eXtreme Programming*. *Universidad de Vigo*, 2008, vol. 22. Disponible en: http://iie.fing.edu.uy/~josej/docs/XP%20-%20Jose%20Joskowicz.pdf

[6] Help - Eclipse Platform. Disponible de: https://help.eclipse.org/2019-06/index.jsp

[7] ARNOLD, Ken; GOSLING, James; HOLMES, David. *The Java programming language*. Addison Wesley Professional, 2005. Disponible en : http://etf.beastweb.org/index.php/site/download/Java_Programming.pdf

[8] GROUSSARD, Thierry. *JAVA 7: Los fundamentos del lenguaje de programación Java*. Cornellà de Llobregat (Barcelona): Ediciones ENI, 2012. Disponible en: https://books.google.es/books?hl=es&lr=&id=JaPTzKZxbN4C&oi=fnd&pg=PA9&dq=java++caracteristicas&ots=pV5CqhymVi&sig=xNe3nLXota0uCFn-4_huBkG6j84#v=onepage&q=java%20%20caracteristicas&f=false.

[9] SONI, Dipa; MAKWANA, Ashwin. A survey on MQTT: a protocol of internet of things (IoT). En *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*. 2017. Disponible en : https://www.researchgate.net/profile/Dipa_Soni/publication/316018571_A_SURVEY_ON_MQTT_A_PROTOCOL_OF_INTERNET_OF_THINGSIOT/links/58edafd4aca2724f0a26e0bf/A-SURVEY-ON-MQTT-A-PROTOCOL-OF-INTERNET-OF-THINGSIOT.pdf

[10] AL-FUQAHA, Ala ; MOHAMMADI, M. ; ALEDHARI, M ; AYYASH, M. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 2015, vol. 17, no 4, p. 2347-2376. Disponible en : https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7123563

[11] MAGNONI, Luca. Modern messaging for distributed sytems. *En Journal of Physics: Conference Series*. IOP Publishing, 2015. p. 012038. Disponible en : https://iopscience.iop.org/article/10.1088/1742-6596/608/1/012038/pdf

[12] Mosquitto | The Eclipse Foundation. Disponible en : https://www.eclipse.org/proposals/technology.mosquitto/

[13] Eclipse Mosquitto. Disponible en : https://mosquitto.org/

[14] BOSCHI, Sigismondo; SANTOMAGGIO, Gabriele. *RabbitMQ cookbook*. Packt Publishing Ltd, 2013. Disponible en: http://www.spooch.dk/Ebooks/Programming/RabbitMQ%20Essentials%20%5BeBook%5D.pdf

[15] MASSE, Mark. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011. Disponible en: https://s3.eu-central-1.amazonaws.com/barbarabonte/eed5d8774844f7e9e7a31e48b6135064.pdf

[16] CROCKFORD, Douglas. *The application/json media type for javascript object notation (json)*. 2006. Disponible en: https://www.rfc-editor.org/rfc/pdfrfc/rfc4627.txt.pdf

[17] *Extensible Markup Language (XML)*. 2011. Disponible en : https://www.w3.org/XML/

[18] *XML VS JSON*. (2018). [Imagen]. Disponible en : https://2.bp.blogspot.com/-iUfbUA2Mm5g/W4paxu6cJ-I/AAAAAAAAAPc/O7mrGCpknywlCaajQmy-yzGzsu3bUxd1ACLcBGAs/s1600/json-vs-xml-which-format-to-use-for-your-api.png

[19] OwnTracks Booklet. Disponible en : https://owntracks.org/booklet/

[20] HETHEY, Jonathan M. *GitLab Repository Management*. Packt Publishing Ltd, 2013. Disponible en : http://www.spooch.dk/Ebooks/Programming/GitLab%20Repository%20Management%20%5BeBook%5D.pdf

[21] *GitHub vs. GitLab vs. Bitbucket: ¿Que repositorio elegir?*. Disponible en: http://www.clubdetecnologia.net/blog/2017/github-vs-gitlab-vs-bitbucket-que-repositorio-elegir/

[22] *Cómo aprender a usar GitHub y GitLab sin morir en el intento*. (2018). [Imagen]. Disponible en: https://instatecno.com/wp-content/uploads/2018/06/Git-GitHub-GitLab.jpeg

[23] MUSCHKO, Benjamin. *Gradle in action*. Manning, 2014. Disponible en: https://gradle.org/books/manning-publications-gradle-in-action.pdf

[24] REYNOSO, Carlos Billy. Introducción a la Arquitectura de Software. *Universidad de Buenos Aires*, 2004, vol. 33.Disponible en: http://carlosreynoso.com.ar/archivos/arquitectura/Arquitectura-software.pdf

[25] Andrés Pérez García, O. (2016). Mecanismo de comunicación síncrono. [Imagen]. Disponible en: https://www.researchgate.net/profile/Osvaldo_Andres_Perez_Garcia/publication/296639063/figure/fig3/AS:337356096851974@1457443369499/Mecanismo-de-comunicacion-sincrono.png

[26] DHALL, Rohit; SOLANKI, Vijender. An IoT Based Predictive Connected Car Maintenance. *International Journal of Interactive Multimedia & Artificial Intelligence*, 2017, vol. 4, no 3. Disponible en: https://pdfs.semanticscholar.org/2742/de1b098e0de5bf08af081f2bfde3dc16c044.pdf

[27] MERA PAZ, Julián Andrés, et al. Análisis del proceso de pruebas de calidad de software. 2016. Disponible en: http://backdoortechnology.net/bitstream/ucc/962/1/Pruebas.pdf

[28] CADAVID, Andrés Navarro; MARTÍNEZ, Juan Daniel Fernández; VÉLEZ, Jonathan Morales. Revisión de metodologías ágiles para el desarrollo de software. *Prospectiva*, 2013, vol. 11, no 2, p. 30-39. Disponible en: https://www.redalyc.org/pdf/4962/496250736004.pdf

[29] GAMMA, Erich; BECK, Kent. JUnit. 2006. Disponible en: https://wiki.hsr.ch/APF/files/06_Junit_Dissection.pdf