



ugr | Universidad
de Granada

TRABAJO DE FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA

RSMaP, reconocimiento y representación del tráfico

Autor

José Manuel Luque Burgos

Tutor

Juan Julián Merelo Guervós



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, 13 de septiembre de 2016



RSMap, reconocimiento y representación del tráfico

José Manuel Luque Burgos

Resumen

Palabras clave: *smart cities, tiempo real, tráfico, sensores, análisis de datos, software libre*

RSMap es un conjunto de tecnologías que trabajan en conjunto con el propósito de ofrecer información en tiempo real del estado del tráfico.

La información es recolectada por sensores situados cerca de cualquier ruta por la que pasen vehículos.

Éstos dispositivos filtran mediante micrófonos el sonido ambiente de la localización en la que se encuentran y remiten la información a los elementos que conforman la parte visual y de almacenamiento de RSMap.

En RSMap los dispositivos recolectores envían las señales de detección de vehículos a una plataforma web. Ésta web representará las señales enviadas sobre un mapa en tiempo real permitiendo a un usuario saber en qué puntos existe tráfico. La información almacenada aquí tiene carácter temporal pues, una vez representada sobre el mapa ya no es útil pero dado que se han invertido una serie de recursos en la recolección y procesado de dicha información, ésta es aprovechada para otros propósitos.

En este punto RSMap se sirve de herramientas orientadas al Big Data, guardando la información obtenida en una base de datos especialmente diseñada para almacenar millones de registros.

Además RSMap incluye otra forma de visualización, que permite consultar los datos almacenados de manera masiva a lo largo del tiempo para su análisis y estudio permitiendo así ofrecer la posibilidad de optimizar el tráfico de ciudades o carreteras, ver que rutas tienen más afluencia o hacer una estimación de el caudal de vehículos que circulará por una vía en un día, usando los datos almacenados en la plataforma.

RSMMap, traffic analysis and recognition

José Manuel Luque Burgos

Extended abstract

Key words: *smart cities, real time, traffic, sensors, data analysis, open source*

The main reason to develop a system like RSMMap is the innovative related technologies needed to cover a solution. Nowadays Big Data and IOT software and techniques are growing fast as well as the preoccupation for making the cities a better place to live improving several aspects like administrative, eco-friendly, and monetary subjects. The problem that RSMMap tries to solve is the perfect scenario to join these topics and technologies in one solution. Traffic is an important element which is related to the aspects mentioned above because with an optimal traffic in a city we may reduce pollution, identify noisy zones or get a better traffic flow.

RSMMap system allows the end-user to know the actual traffic status through several devices which are capturing the noise generated by vehicles and interpreting it as signals which are represented in two different ways. The 'obvious' way is to show it on a map so we can see how many vehicles are running on certain street at a certain time.

The second way, much more interesting, is the possibility of watching the massive stored data along the days to perform Big Data analysis operations for example, to use it to configure predictive models.

The approach of the solution is acquired by 4 main elements.

The first of them are the receptors, They are lightweight computers, (Raspberry Pi in our case). Their purpose is the collector element in RSMMap system, they process and send the signals obtained with a microphone to the storage/visualization points.

As middleware platform between receptors and storage RSMMap uses KAA IOT which is a great open source platform that manages a lot of devices which task is to generate and simplify the way to allow the receptors to send data to the massive storage.

The third element is the massive storage, RSMMap uses Apache Cassandra which is best-in-class database storage system for time series and its conditions of scalability makes it a perfect component to perform the tasks of storage being able to store hundreds of thousands registers per day.

Related to Apache Cassandra, RSMMap makes use of Apache Zeppelin which is a very modern (but powerful) tool to visualize the data stored in

Apache Cassandra in several ways due to the fact that we can get the data as we want through queries.

To conclude, RSMap has a typical end-point for normal users. A Web site where they can observe the traffic in real time, the receptors send signals when they 'think' that a vehicle passed in front of them. This Web site is developed in Django and Django-Rest-Framework. Django is a great web framework developed in Python to build professional web pages and Django-Rest-Framework is a module which allows the receptors to send detected signals to the web page with GET/POST/PATCH/DELETE requests.

At the end, RSMap is a set of technologies and tools assembled to perform traffic detection. However, the possibilities to adapt the system to other purposes are too many thanks to the modular system structure. If we want to catch other kind of environment information we only have to adapt the receptors code and define new models to store the data but the essence of the application would be the same.

Yo, **José Manuel Luque Burgos**, alumno de la titulación **Grado en Ingeniería Informática** de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación** de la **Universidad de Granada**, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado (*RSMap, reconocimiento y representación del tráfico*) en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Toda la información relacionada con el proyecto se encuentra bajo licencia **Creative Commons Attribution-ShareAlike 4.0** (<https://creativecommons.org/licenses/by-sa/4.0/>), por lo que se permite el uso comercial de la obra y de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Fdo: José Manuel Luque Burgos

Granada, a 13 de septiembre de 2016

D. Juan Julián Merelo Guervós, profesor del **Departamento de Arquitectura y Tecnología de Computadores** de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***RSMMap, reconocimiento y representación del tráfico***, ha sido realizado bajo su supervisión por **José Manuel Luque Burgos**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 13 de septiembre de 2016.

El tutor:

Juan Julián Merelo Guervós

Agradecimientos

A mis padres Manuel y Aurora, por haberme apoyado desde el primer al último día de mi paso por la Universidad y a mi hermana Ana María por ayudarme en lo que he necesitado.

A mi tutor Juan Julián Merelo Guervós por confiar en mí y permitirme realizar el proyecto bajo su supervisión.

A mis amigos porque siempre están ahí cuando los necesito.

A mis compañeros de Universidad por todos los buenos y no tan buenos momentos que hemos pasado juntos.

Y a todas las comunidades que hacen posible que el Software Libre sea un hecho, en las que se trabaja desinteresadamente por convicciones propias y sin ánimo de lucro.

Índice general

1. Motivación e Introducción	1
1.1. Aplicaciones similares a RSMap	2
1.2. Alcance de la memoria	3
2. Objetivos	5
2.1. Alcance de los objetivos	6
2.2. Interdependencia de los objetivos	7
3. Planificación y presupuesto	9
3.1. Planificación	9
3.1.1. Fases	9
3.1.2. Definición y estimación de tiempo para las tareas de cada fase	10
3.1.3. Diagramas de temporización	12
3.2. Presupuesto	14
4. Análisis	19
4.0.1. Metodología	19
4.1. Análisis de requisitos	20
4.1.1. Actores	20
4.1.2. Requisitos funcionales	20
4.1.3. Requisitos no funcionales	21
4.1.4. Requisitos de almacenamiento	21
4.2. Casos de uso	23
4.2.1. Descripción de actores	23
4.2.2. Casos de uso	23
4.3. Diagramas	29
4.3.1. Diagrama de paquetes	29
4.3.2. Diagrama arquitectónico	30
4.3.3. Diagrama de clases	32
4.3.4. Diagramas de casos de uso	33
4.3.5. Diagramas de secuencia	35
4.3.6. Diagramas de interfaz	38

5. Diseño	41
5.1. Diseño del dispositivo receptor	41
5.2. Diseño del módulo de reconocimiento de vehículos	41
5.2.1. Estructuras de datos en el dispositivo receptor	42
5.2.2. Desarrollos algorítmicos	42
5.3. Almacenamiento de información	45
5.4. Representación de información	45
5.5. Diseño del portal web	46
5.5.1. Diseño de la api REST	47
6. Implementación	49
6.1. Descripción de las tecnologías seleccionadas	49
6.1.1. Plataforma IOT	50
6.1.2. Dispositivo receptor	52
6.1.3. Almacenamiento de información	52
6.1.4. Representación de información	53
6.1.5. Plataforma Web de RSMap	54
6.2. Instalación y configuración de Raspberry Pi	54
6.2.1. Instalación	54
6.2.2. Configuración	55
6.3. Instalación y configuración de Kaa	56
6.3.1. Instalación	56
6.3.2. Configuración	60
6.4. Instalación y configuración de Apache Cassandra.	64
6.4.1. Instalación	64
6.4.2. Configuración	65
6.5. Desarrollo del módulo de detección de vehículos.	74
6.5.1. Módulo de detección	74
6.5.2. Módulo de envío	82
6.6. Desarrollo de la plataforma web.	84
6.6.1. Plataforma web	84
6.6.2. API Rest	90
6.7. Instalación y configuración de Apache Zeppelin.	92
6.7.1. Instalación	92
6.7.2. Configuración	93
6.7.3. Ejemplo de consulta CQL	94
7. Pruebas	97
8. Conclusiones y trabajos futuros	105
8.1. Conclusiones	105
8.2. Trabajos futuros	106
Glosario de términos	107

Anexo. Manual de Usuario 109

Bibliografía 115

Índice de figuras

3.1.	Diagrama fases y tareas	12
3.2.	Diagrama de Pert	13
3.3.	Diagrama de Gantt 1	13
3.4.	Diagrama de Gantt 2	13
3.5.	Raspberry Pi 2B	15
3.6.	Tarjeta de sonido	16
3.7.	Micrófono	16
3.8.	Tarjeta SD	17
3.9.	Memoria USB	17
4.1.	Metodología ágil	19
4.2.	Diagrama de paquetes	29
4.3.	Arquitectura Raspberry Pi	30
4.4.	Arquitectura de RSMap	31
4.5.	Diagrama de clases	32
4.6.	Caso de uso general	33
4.7.	Caso de uso del administrador	33
4.8.	Caso de uso del usuario	34
4.9.	Caso de uso del usuario aportador	34
4.10.	Diagrama de secuencia del Administrador	35
4.11.	Diagrama de secuencia del Usuario	36
4.12.	Diagrama de secuencia del Usuario Aportador	37
4.13.	Boceto de la página principal	38
4.14.	Boceto de la página del mapa	39
5.1.	Proceso de reconocimiento	44
6.1.	Arquitectura de Kaa	51
6.2.	Dashboard de AmazonEC2	57
6.3.	57
6.4.	Selección del tipo de instancia	58
6.5.	Configurar detalles de la instancia	58
6.6.	Selección de almacenamiento	59
6.7.	Configuración de puertos	59

6.8.	Creación de claves	60
6.9.	Sección Management	61
6.10.	Creación de la nueva aplicación	61
6.11.	Definiendo esquemas de datos	62
6.12.	Esquema de datos definido 1	64
6.13.	Esquema de datos definido 2	64
6.14.	Imagen de Cassandra en EC2	65
6.15.	Security Group para Cassandra	65
6.16.	Accediendo al servidor de Cassandra	66
6.17.	Comprobación de las tablas creadas	70
6.18.	Cassandra con datos almacenados	70
6.19.	Funciones dentro de Cassandra	71
6.20.	Definición del SDK	73
6.21.	Definición del SDK	73
6.22.	Icono para valor -1: conexión de dispositivo	87
6.23.	Icono para valor >55.5: vehículo pesado	87
6.24.	Icono para valor 45.5 <x <55.5: vehículo medio	87
6.25.	Icono para valor x <45.5: vehículo ligero	88
6.26.	Home de la web	88
6.27.	Sección de visualizadores	89
6.28.	Sección de características	89
6.29.	Sección de contacto	90
6.30.	Sección del mapa	90
6.31.	Apache Zeppelin en funcionamiento	92
6.32.	Estructura de un Notebook en Zeppelin	93
6.33.	Cassandra como intérprete para Zeppelin	94
6.34.	Muestra de resultados en Zeppelin	95
7.1.	Carga CPU del servidor web y API	97
7.2.	Carga de entrada de red del servidor web y API	98
7.3.	Carga de salida de red del servidor web y API	98
7.4.	Carga CPU del servidor de Cassandra	99
7.5.	Carga de entrada de red del servidor de Cassandra	99
7.6.	Carga de salida de red del servidor de Cassandra	100
7.7.	Carga CPU del servidor de Zeppelin	100
7.8.	Carga de entrada de red del servidor de Zeppelin	101
7.9.	Carga de salida de red del servidor de Zeppelin	101

Índice de tablas

4.1.	Secuencia de CU-1	24
4.2.	Curso alterno de CU-1.	24
4.3.	Secuencia de CU-2	25
4.4.	Secuencia de CU-3	26
4.5.	Curso alterno de CU-3.	26
4.6.	Secuencia de CU-4	27
4.7.	Curso alterno de CU-4.	27
4.8.	Secuencia de CU-5	28
7.1.	Pruebas realizadas sobre RSMap	103

Índice de fragmentos de código

5.1.	Formato a utilizar	42
6.1.	Copia de Raspbian en tarjeta s y usb	54
6.2.	Modificando el dispositivo de arranque de Raspbian	55
6.3.	Instalación del paquete alsa-utils	55
6.4.	Comprobano que el dispositivo es reconocido	55
6.5.	Identificando los dispositivos de sonido disponibles en el sistema	55
6.6.	Comprobano que el dispositivo es reconocido	56
6.7.	Esquema de datos en JSON	63
6.8.	Keyspace en Cassandra	66
6.9.	Esquema de log appender en JSON	67
6.10.	Mécanica de consultas en CQL según la estructura de tablas .	69
6.11.	Reiniciando Cassandra	72
6.12.	Parámetros usados para la comunicación con la API	76
6.13.	Identificación de dispositivo mediante la API	77
6.14.	Conexión con Java mediante un Socket	78
6.15.	Variables correspondientes al análisis	78
6.16.	Hebra productora	79
6.17.	Hebra consumidora	79
6.18.	Inicialización de las hebras	81
6.19.	Inicialización del Socket	83
6.20.	Creación y envío de datos a Cassandra	83
6.21.	Modelo definidos en Django	84
6.22.	Definición de URL's	85
6.23.	Función checkDevices	86
6.24.	Actualización dinámica del mapa	86
6.25.	Serializador del modelo Signal	91
6.26.	Vistas de la API	91
6.27.	Instalación de Apache-Zeppelin	92
6.28.	Ejemplo de consulta CQL	95
8.1.	Versiones requeridas	111
8.2.	Descarga de repositorio	111

8.3. Obtener los dispositivos de audio conectados	112
8.4. Ejecutar la aplicación de configuración	112
8.5. Compilar DataSender.java	112
8.6. Lanzar DataSender	112
8.7. Instalar depenencias	113
8.8. Ejecución de VehicleDetection.py	113

Capítulo 1

Motivación e Introducción

A día de hoy, las tecnologías han llegado a un punto en el que envuelven completamente a las personas, ciudades y el mundo entero. Cada vez son más los elementos que nos encontramos a nuestro alrededor que poseen conexión a Internet mediante la cual envían información de temperatura, humedad relativa, nuestras pulsaciones o los pasos que damos y que posteriormente podemos consultar en nuestro dispositivo móvil o el ordenador. Debido a esto y a otra gran diversidad de elementos que generan y almacenan cantidades ingentes de información han nacido nuevas ramas en la computación como son el **Big Data** o **IOT**.

Las ciudades sin duda alguna son uno de los mejores candidatos para el desarrollo y aplicación de éste tipo de tecnologías. Todos hemos oído hablar últimamente del término *ciudad inteligente* o *smart city* y es que, con la proliferación de las tecnologías citadas anteriormente así como una imperiosa necesidad de gestionar los recursos energéticos, económicos, medioambientales y administrativos de manera más eficiente han hecho necesaria la cohesión entre la tecnología y las propias ciudades.

Partiendo de esta base existen muchos campos en los que trabajar para hacer una ciudad mejor. Uno de los más relevantes puede ser el tráfico por el impacto que tiene y a los diversos factores que afecta.

Bajo estas premisas se aborda el problema de identificar el tráfico que fluye por una ciudad mediante el ruido que éste genera y poder almacenarlo con las tecnologías adecuadas para su análisis y visualización que permitirán conocer su estado, crear sistemas reguladores del mismo en base a los datos obtenidos así como poder comprobar los niveles de ruido generados en ciertas zonas.

RSMap es capaz de detectar el tráfico que fluye por uno (o varios) puntos en los que se encuentran dispositivos **Raspberry Pi** analizando el ruido ambiente. Una vez identificados se envían mediante una **API REST** construida con **Django Rest Framework** y se muestran en un mapa web. La web basada en **Django** hace uso de **Ajax** para una cómoda navegación sin necesidad de refrescar la página manualmente.

El mapa es proporcionado por la API de **Google Maps**.

Por otra parte tenemos **Kaa** que es una plataforma para administrar un gran número de dispositivos. Ésta plataforma genera un **SDK** del cual se sirven los receptores para enviar de manera masiva los datos recabados al SGBD **Apache Cassandra** los cuales pueden ser consultados y descargados mediante un Notebook que proporciona **Apache Zeppelin**.

1.1. Aplicaciones similares a RSMap

- **Libelium** (<http://www.libelium.com>)

Provee una red de sensores Bluetooth sobre una red ZigBee para el análisis del tráfico. Su mayor virtud e inconveniente a la vez es que es capaz de detectar el tráfico con una gran acierto pero por otra parte es necesario que los vehículos tengan Bluetooth lo cual no siempre sucede.

- **Houston Radar** (<http://houston-radar.com>)

Posee un detector multidireccional para identificar el tráfico pero necesita los sensores tienen una autonomía de unas dos semanas por lo que implica un mantenimiento por cada dispositivo detector.

- **Diamond Traffic** (<http://diamondtraffic.com/>)

Tiene una arquitectura parecida a la de un radar, ésto es un gran inconveniente pues no se pueden situar dispositivos a lo largo de una calle sin una gran alteración del entorno.

- **Counting Cars** (<http://www.countingcars.com>)

Posee varios tipos de detectores basados en audio y video por contrapartida el precio de cada dispositivo es desorbitado.

Un factor común que tienen todas estas plataformas es que es software propietario por lo que se desconoce el uso que se le pueden dar a los datos recabados además de tener unos costes nada asumibles si se pretende monitorizar el tráfico con muchos dispositivos.

RSMap compite ofreciendo una plataforma **Open Source**, totalmente personalizable y con costes más equilibrados.

Mediante un análisis *DAFO* podemos concluir que las **debilidades** de RSMap es que a diferencia de sus competidores, usa equipos más baratos para la recolección de datos lo cual puede suponer una pérdida de acierto a la hora de la detección del tráfico no obstante existe un margen de mejora para optimizar el algoritmo de detección de vehículos.

La **amenaza** principal es sin duda que éstas plataformas cuenta con años de trabajo lo que se traduce en una gran amplitud de clientes.

Las partes que hacen atractivo a RSMap son que es totalmente libre y personalizable y que la infraestructura tiene un coste menor por tanto hablamos de claras **fortalezas** frente a sus competidores.

Por último las **oportunidades** que se proyectan son prometedoras debido al auge de las *Smart Cities* luego, con un buen prototipo y técnicas de marketing se podría dar a conocer e implementar en lugares modestos e ir depurando y mejorando su funcionamiento hasta que sea capaz de funcionar óptimamente en grandes urbes.

1.2. Alcance de la memoria

El proceso técnico se encuentra detallado en el *capítulo 5 (Diseño)*, que contiene el modelo que se pretende implementar y en el *capítulo 6 (Implementación)* que contiene los elementos software desarrollados.

Antes de ello se expondrán en el *capítulo 2* los (**Objetivos**) que se consideran necesarios para obtener una solución válida al problema; el *capítulo 3 (Planificación)* consta de las distintas fases a superar y en el *capítulo 4 (Análisis)* de los requisitos necesarios a alcanzar.

Por último en el *capítulo 7 (Pruebas)*, se remiten las pruebas realizadas que corroboran el correcto funcionamiento y se concluye con el el *capítulo 8 (Conclusiones y trabajos futuros)* en el cual se analiza el camino recorrido a lo largo de todas las fases anteriores así como posibles funcionalidades incorporables al sistema.

Capítulo 2

Objetivos

El propósito de RSMap es proporcionar un sistema compuesto por varios modulos cuyo cometido final sea que cualquiera pueda conocer el estado del tráfico actual en una localización concreta.

Los objetivos se enumeran con más detalle a continuación:

- **O1:** Desarrollar un método que permita reconocer cuando un vehículo ha pasado por la localización en la que el dispositivo recolector se encuentra.
- **O2:** La arquitectura de la aplicación cliente debe permitir a los usuarios instalar un dispositivo y aportar información del tráfico en el punto que ellos deseen.
- **O3:** Hacer uso de una base de datos especialmente diseñada para trabajar bajo el paradigma del Big Data.
- **O4:** Proveer un método fácil de consulta de los datos que el conjunto de dispositivos recolectores obtiene, para que cualquier usuario pueda acceder a ellos.
- **O5:** Dotar al sistema de una aplicación web en la que mediante un mapa, se muestren los puntos en los que están pasando vehículos.

Los aspectos formativos más utilizados son:

- **Infraestructuras virtuales**, ya que los servicios usados son en su totalidad virtualizados así como el scripting necesario para la automatización de ciertas tareas.
- **Sistemas concurrentes**, debido a que al trabajar con dispositivos de un perfil medio-bajo para la recolección se hace necesario el paralelizar ciertas operaciones.
- **Bases de datos**, un elemento indispensable para tratar el problema que se plantea de almacenamiento de grandes cantidades de datos.
- **Ingeniería de servidores**, para la configuración de los distintos VPS usados.
- **Procesamiento digital de señales**, para un correcto entendimiento del análisis del sonido.
- **Desarrollo de aplicaciones para internet**, para el desarrollo de la plataforma web.

2.1. Alcance de los objetivos

El proyecto ofrecerá dos secciones en las que se podrá visualizar el estado del tráfico mediante un mapa y consultar los valores obtenidos de los dispositivos que se encuentran retransmitiendo información.

Tras el desarrollo, RSMap será capaz de identificar cuando pasa un vehículo cerca de la ubicación del dispositivo recolector de datos que a su vez, se encargará de transmitirlo a distintas plataformas para su almacenamiento, consulta y representación. Cabe destacar que el proyecto estará constituido únicamente con software libre lo que ofrece la posibilidad de añadir nuevas funcionalidades por parte de terceros gracias al carácter modular de la aplicación, además de ser una obligación moral debido a que todas las tecnologías usadas que se detallarán más adelante son completamente abiertas.

Por otra parte se pretende ofrecer los datos de manera libre al público o entidades que deseen acceder a ellos.

En este caso, la aplicación hace uso del módulo desarrollado de detección de tráfico, sin embargo, el ámbito de la aplicación no se cierra exclusivamente a éste propósito ya que debido a su modularidad será posible el desarrollo y acoplamiento de distintos módulos de detección como podrían ser la temperatura, la humedad relativa o la luminosidad ó el análisis de los datos obtenidos mediante herramientas como Apache Spark por ejemplo,

quedando completamente abierto a cualquier modelo que un desarrollador quiera representar, por tanto hablamos de una herramienta completamente personalizable según las necesidades requeridas.

2.2. Interdependencia de los objetivos

Los objetivos podríamos alinearlos mentalmente a modo de una pirámide inversa, el principal y sobre el que gira este proyecto es **O1??** el cual va a generar la información, sin la cual el resto no tendrían sentido. Por otra parte existe una fuerte relación con **O2** ya que el mismo módulo encargado de generar la información deberá ser replicable y configurable tantas veces como se desee, de esta manera contribuimos a como se ha mencionado anteriormente, cualquier usuario pueda aportar datos al sistema.

En el siguiente escalón de la pirámide tendríamos **O3** y **O4** los cuales también están fuertemente ligados entre sí, ya que de nada nos serviría almacenar toda la información si no proveemos un método fácil para consultarla.

Por último tenemos **O5**, que al igual que los anteriores necesita consumir información, por tanto su dependencia con **O1**, es total.

Capítulo 3

Planificación y presupuesto

3.1. Planificación

3.1.1. Fases

Tomando en consideración que la parte del proyecto referente al análisis del sonido tiene un fuerte componente teórico, se ha dedicado una sección inicial a recabar todo tipo de información relacionada con la materia para obtener una visión global y definir así un enfoque claro antes de empezar a analizar tanto las especificaciones como los requisitos. El resto de las fases son en su mayoría, comunes a cualquier proceso de elaboración de un proyecto de software.

- **Fase 1:** Lecturas relacionadas con la materia del proyecto.
- **Fase 2:** Especificaciones y planificación.
- **Fase 3:** Análisis y diseño.
- **Fase 4:** Implementación.
- **Fase 5:** Pruebas.
- **Fase 6:** Documentación.

3.1.2. Definición y estimación de tiempo para las tareas de cada fase

Las fases detalladas anteriormente tienen varias tareas como sub-elementos, en la siguiente lista se detalla dichos sub-elementos y la estimación aproximada para llevarla a cabo. En la última línea de cada tarea se indica el total en horas aproximado dedicado al subconjunto de tareas.

- **Lecturas relacionadas con la materia del proyecto:**

- Como se comportan las señales.
- Muestreo de señales.
- Digitalización de señales.
- Identificación de patrones en señales.
- Duración estimada: 10 días, 30 horas.
- Duración requerida: 10 días, 40 horas.

- **Especificaciones y planificación:**

- Objetivos.
- Fases.
- Tareas y temporización.
- Presupuesto.
- Duración estimada: 4 días, 30 horas.
- Duración requerida: 4 días, 24 horas.

- **Análisis y diseño:**

- Análisis de requisitos.
- Diagramas de la aplicación.
- Duración estimada: 20 días, 86 horas.
- Duración requerida: 20 días, 80 horas.

- **Implementación:**

- Selección de tecnologías.
- Configuración Raspberry Pi
- Detección de sonido
- Identificación de vehículos
- Configuración de la plataforma IOT
- Configuración de base de datos de almacenamiento masivo.

- Configuración de visualizador de los datos recolectados.
- Desarrollo de sistema que comunique cuando un vehículo es detectado a la plataforma web.
- Desarrollo de la plataforma web
- Duración estimada: 60 días, 300 horas.
- Duración requerida: 60 días, 216 horas.

Cabe destacar que al ser un proyecto en el que existen fuertes dependencias entre los objetivos, la fase de pruebas ha sido completada al mismo tiempo que la de desarrollo pues se necesitan resultados concluyentes en algunos aspectos para poder avanzar en objetivos posteriores.

▪ **Pruebas:**

- Pruebas de detección de vehículos.
- Pruebas de escritura en base de datos.
- Pruebas de lectura de la base de datos.
- Pruebas de comunicación entre dispositivos y plataforma web.
- Duración estimada: 60 días, 120 horas.
- Duración requerida: 60 días, 168 horas.

La documentación al igual que las pruebas, ha ido desarrollándose al mismo tiempo que la implementación, como se puede observar en la *Figura 3.1*.

▪ **Documentación:**

- Documentación del código.
- Documentación del proyecto.
- Duración estimada: 60 días, 80 horas.
- Duración requerida: 96 horas.

Los resultados nos muestran que en el conjunto de Implementación, Pruebas y Documentación se ha invertido un 76 % del tiempo total dentro del cual se ha distribuido de la siguiente forma:

- **Implementación:** 45 %.
- **Pruebas:** 20 %.
- **Documentación:** 35 %.

Mientras que la estimación inicial suponia un:

- **Implementación:** 50 %.
- **Pruebas:** 30 %.
- **Documentación:** 20 %.

Las tareas cuyo trabajo han requerido más tiempo del calculado han sido la Documentación, Pruebas y las Lecturas relacionadas con la materia del proyecto. Por el contrario en las tareas de Especificaciones y Planificación, Análisis y Diseño e Implementación los tiempos se han visto reducidos.

3.1.3. Diagramas de temporización

La información acerca de la temporización queda expuesta en las siguientes figuras mediante un diagrama de Fases y Tareas (*Figura 3.1*), un diagrama de PERT (*Figura 3.2*) y otro de GANTT (*Figura 3.3*) y (*Figura 3.4*).

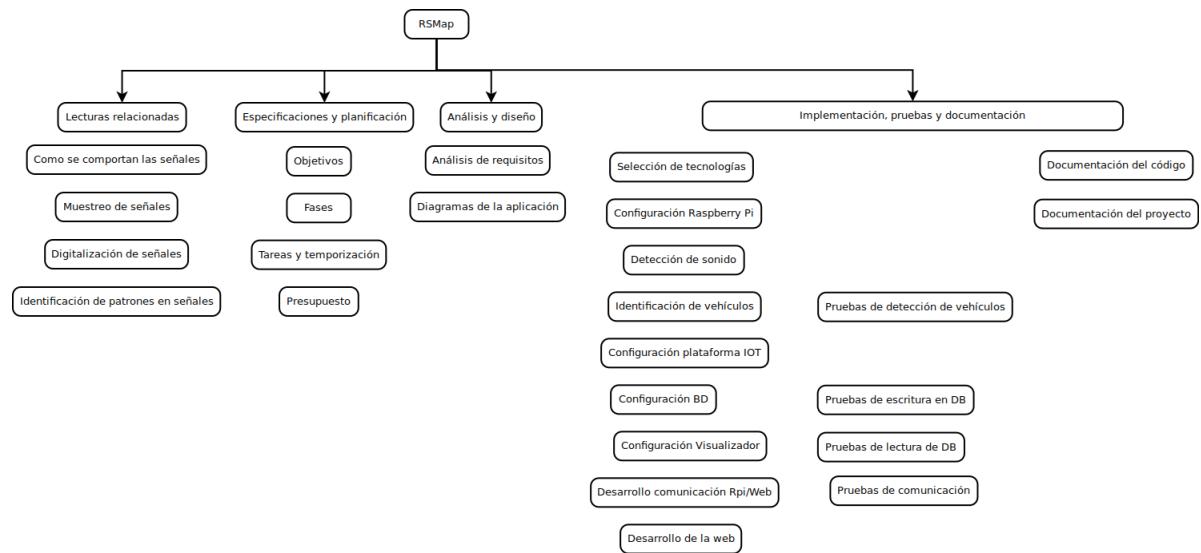


Figura 3.1: Diagrama fases y tareas

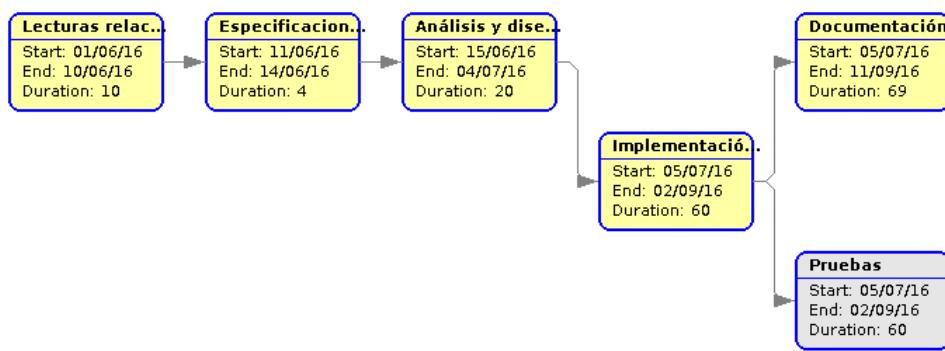


Figura 3.2: Diagrama de Pert

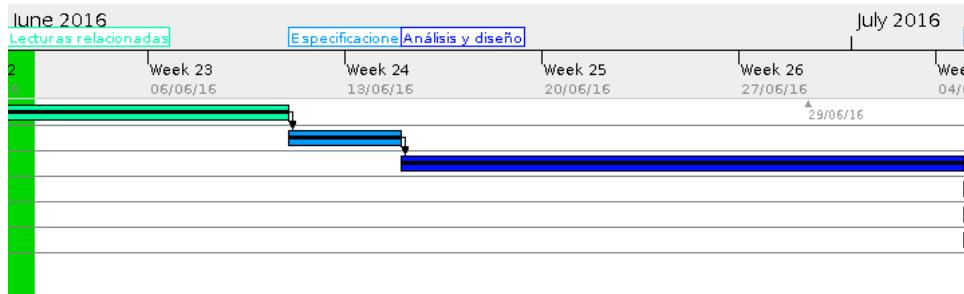


Figura 3.3: Diagrama de Gantt 1

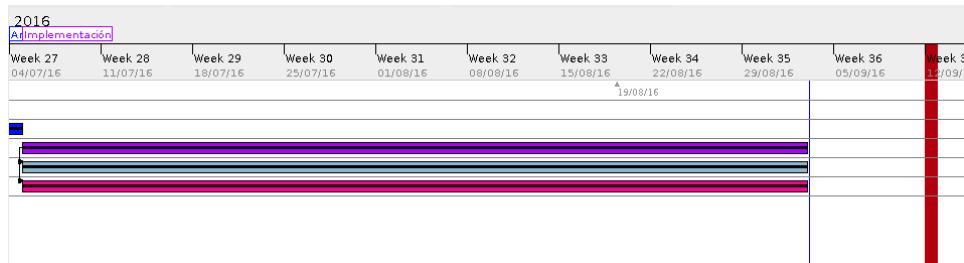


Figura 3.4: Diagrama de Gantt 2

3.2. Presupuesto

En el presupuesto podemos distinguir entre el software y el hardware. Por una parte el aspecto software queda totalmente cubierto de manera gratuita ya que todas las tecnologías usadas son de código abierto por tanto esto nos ahorra muchos - *por no decir todos* - los costes de este tipo.

El proyecto consta de los siguientes elementos software y hardware:

Hardware:

- Raspberry Pi 2B,
- Tarjeta de sonido USB.
- Micrófono
- Adaptador wifi USB o cable de red RJ45.
- Adaptador de alimentación para la Raspberry Pi.
- Tarjeta SD y memoria USB.

Software:

- Sistema de control de versiones para almacenar tanto la información relacionada proyecto como el código.
- Módulo de detección y envío de información.
- Plataforma IOT, que gestiona los dispositivos conectados a el sistema.
- Base de datos NoSQL, almacenara la información recolectada.
- Notebook que permite analizar cantidades masivas de datos almacenadas en la base de datos.
- API REST, la cual recibe las señales de detección de vehículos.
- Web, representa en un mapa las señales recibidas por la API.
- Servidores, almacenan y ofrecen la información recopilada así como la interfaz para ello.

Los servidores utilizados tienen las siguientes características:

■ **VPS Amazon EC2**

- Descripción: Conjunto de servidores que conforman el proyecto, de tipo **t2.micro**
 - 1 instancia para la plataforma IOT.
 - 1 instancia para la base de datos de almacenamiento masivo.
 - 1 instancia para el visualizador de datos.
 - 1 instancia para el servidor web.
- Especificaciones: <https://aws.amazon.com/es/ec2/instance-types/>
- Precio: Este tipo de servidores son gratuitos en amazon EC2 durante un año, por lo que no existe coste computable a priori.

El precio aproximado del hardware requerido se detalla a continuación:

■ **Raspberry Pi**

- Descripción: Dispositivo usado para la detección de datos.
- Especificaciones: (<https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>)
- Precio: 31.98 Euros



Figura 3.5: Raspberry Pi 2B

■ Tarjeta de audio USB

- Descripción: Interfaz por el cual capturamos el sonido.
- Especificaciones: <http://www.logilink.org/showproduct/UA0078.htm>
- Precio: 9.37 Euros



Figura 3.6: Tarjeta de sonido

■ Micrófono

- Descripción: Entrada a la interfaz de audio.
- Especificaciones: <https://www.amazon.co.uk/Flexible-3-5mm-Microphone-Notebook/dp/B00CRASVFC>
- Precio: 5 Euros



Figura 3.7: Micrófono

- **Tarjeta SD (4GB)**

- Descripción: Tarjeta SD que permite la carga del SO en la Raspberry Pi.
- Especificaciones: <https://www.amazon.com/SanDisk-Class-Memory-SDSDB-004G-B35-Changdp/B000WQK0QM>
- Precio: 12 Euros



Figura 3.8: Tarjeta SD

- **Memoria USB (16GB)**

- Descripción: Memoria que contiene el SO de la Raspberry Pi.
- Especificaciones: <http://tiendas.mediamarkt.es/p/pendrive-de-16gb-sandisk-cruzer-b0-ultracompacto-en-color-negro-y-rojo-1117065>
- Precio: 5 Euros



Figura 3.9: Memoria USB

Por tanto, por un total de unos 65 Euros aproximadamente podemos disponer de todo lo necesario para construir el proyecto.

Capítulo 4

Análisis

4.0.1. Metodología

Desde el primer momento se tiene claro que para el desarrollo del proyecto se usará el método de Desarrollo Ágil basado en iteraciones, de éste modo se van obteniendo pequeñas funcionalidades del sistema, descartando las que se consideren prescindibles.

Dentro de cada iteración tenemos unas tareas que cumplir para darla por terminada como podemos observar en la *Figura 4.1*, éste método incremental permite que el desarrollo se haga sobre una base sólida evitando modificaciones que afecten al funcionamiento global de la aplicación una vez éste se encuentre en un punto avanzado del desarrollo.



Figura 4.1: Metodología ágil

Por otra parte, es la manera que más natural resulta a la hora de desarrollar software pues se van completando hitos. Cada uno de ellos define una funcionalidad implementada por lo que si se desea abrir un nuevo camino (*también conocidos como Fork o Roadmap*) en la aplicación tenemos varios puntos de partida evitando tener que empezar desde cero en caso de llegar a un *punto muerto* en el desarrollo.

Las iteraciones se llevan acabo sobre las tareas de Implementación, pruebas.

4.1. Análisis de requisitos

A continuación se detallan los actores implicados en el sistema, requisitos funcionales y no funcionales, requisitos de almacenamiento, casos de uso y por último los diagramas generados para el desarrollo incluyendo en éste punto los bocetos ilustrativos de interfaz de la web.

4.1.1. Actores

Los actores que hacen uso del sistema son tres, **administrador**, el **aportador de información** y **usuario visitante**.

El **administrador** es el encargado de mantener el correcto funcionamiento de la aplicación y sus servicios.

El **aportador de información** puede ser cualquier usuario que decida configurar un dispositivo para añadir información, en éste caso creando un punto que recolecte datos y los remita al sistema.

El **usuario visitante** puede ser cualquier persona que consulte información en la aplicación, ya sea a través de la web o através de la plataforma para consultar todos los datos almacenados (en éste último caso el usuario debe tener un mínimo de conocimientos acerca de las consultas necesarias para obtener la información).

4.1.2. Requisitos funcionales

En ésta sección se describen las funcionalidades básicas que el sistema debe cubrir para dar el problema planteado como resuelto.

- **RF-1.** Mapa web:
 - **RF-1.1.** Desplegar la aplicación web.
 - **RF-1.2.** Leer datos recibidos desde los clientes.
 - **RF-1.2.** Mostrar los datos sin necesidad de refrescar la página.
- **RF-2.** Acceso libre a la información almacenada:
 - **RF-2.1.** Proveer un sistema que sea capaz de mostrar los datos almacenados y filtrarlos según la conveniencia.
 - **RF-2.2.** Proveer un sistema que sea capaz de extraer la información consultada.

- **RF-3.** Añadir información:

- **RF-3.1.** Un usuario podrá aportar información al sistema de manera independiente y autónoma.

- **RF-3.** Identificación de vehículos:

- **RF-3.1.** Filtrar la información entrante en forma de sonido e identificar si se trata de tráfico.
 - **RF-3.2.** Hacer una distinción según el tipo de vehículo.
 - **RF-3.3.** Si el dispositivo recolector se queda sin conexión, debe ser capaz de seguir almacenando los datos obtenidos y enviarlos cuando le sea posible.

4.1.3. Requisitos no funcionales

Aquí se enumeran las propiedades ligadas al desarrollo e implementación que el sistema debe cumplir.

- **RNF-0.** El software de terceros usado debe ser Open Source, así como los SO usados tanto para los servidores como para la Raspberry Pi.
- **RNF-1.** El módulo encargado de la detección del sonido estará desarrollado en Python por su simplicidad, las dependencias necesarias se detallarán en la sección de implementación(6). Además éste módulo debe aprovechar el procesamiento en paralelo para que su funcionamiento sea óptimo.
- **RNF-2.** La retransmisión de datos debe ser establecida en un intervalo de tiempo mínimo para garantizar que la aplicación actúa en tiempo real.
- **RNF-3.** El mapa web debe mostrar la información durante un breve periodo de tiempo para no acumular señales anteriores.
- **RNF-4.** Siempre que sea posible se automatizarán las interacciones con cualquier elemento del sistema mediante scripts.
- **RNF-5.** El código de la aplicación debe estar disponible en algún sistema de control de versiones y de manera pública.

4.1.4. Requisitos de almacenamiento

La información que se almacenará en el sistema se puede dividir en dos apartados, la información almacenada que sirve para representar los datos en la web y por otra parte la información que se almacena en una base de datos diseñada para consultas de datos masivas, que aunque tienen elementos similares difieren en algunos aspectos.

- **RA-1.** Almacenamiento para la web.
 - Timestamp de la señal recibida.
 - Coordenadas de la señal recibida.
 - Valor (numérico) de la señal recibida.
- **RA-2.** Almacenamiento de datos masivo.
 - Localización.
 - Timestamp.
 - Id del dispositivo.
 - Valor de la señal.

Obsérvese que no se almacena ningún dato personal luego a efectos prácticos, los artículos 7 y 8 de la LOPD en los que se detalla el tipo de datos de carácter sensible que hay que preservar con especial atención no tienen aplicación sobre RSMap.

4.2. Casos de uso

Debido a que RSMap es un servicio para proveer y aportar información, los casos de uso quedan simplificados de manera que la mayor parte de ellos hacen referencia a la aportación/consulta de datos.

4.2.1. Descripción de actores

- **A-1.** Administrador.

- Descripción: Es el responsable del correcto funcionamiento de la aplicación.
- Tareas: Administración, mantenimiento y actualización de todos los elementos que conforman el sistema.

- **A-2.** Usuario aportador de información.

- Descripción: Persona que se dispone a configurar un dispositivo para añadir información al sistema.
- Comentarios: Debe tener un conocimiento mínimo en sistemas operativos, concretamente en Linux (Raspberry Pi) para configurar el dispositivo y proceder a remitir información.

- **A-2.** Usuario visitante.

- Descripción: Persona que se dispone a consultar los datos que almacena el sistema.
- Comentarios: Para la consulta de datos mediante la web no necesita ningún conocimiento, para la consulta en el almacenamiento masivo tal vez sea necesario tener conocimientos básicos en consultas. En cualquier caso se proveerán consultas predefinidas cuyos parámetros sean totalmente configurables.

4.2.2. Casos de uso

- **CU-1.** Desplegar servicio Web RSMap.

- Actores: Administrador.
- Tipo: Primario, esencial.
- Referencias:
- Precondición: Disponer de los ficheros de GitHub para lanzarla y una correcta configuración de puertos.
- Postcondición: Se ejecuta el servidor web de RSMap.
- Autor: José Manuel Luque Burgos.

- Versión: 1.0.
- Propósito: Levantar una instancia de la web de RSMap.
- Resumen: El desarrollador puede actualizar la aplicación desde GitHub y lanzarla de manera fácil con nuevas funcionalidades.

Caso de uso 1		
	Actor	Sistema
1	Administrador: descarga el repositorio de RSMap y lanza la aplicación web.	
		2 El sistema sobre el que se instala levanta un servicio web que contiene RSMap.

Tabla 4.1: Secuencia de CU-1

Curso alterno	
2b	Si el servidor está funcionando, no se ejecuta la nueva instancia de la web.

Tabla 4.2: Curso alterno de CU-1.

■ CU-2. Configurar los clientes de RSMap.

- Actores: Administrador.
- Tipo: Primario, esencial.
- Referencias:
- Precondición: Disponer de los servidores necesarios para enviar la información.
- Postcondición: Los clientes remitirán los datos a donde el administrador desee.
- Autor: José Manuel Luque Burgos.
- Versión: 1.0.
- Propósito: Permitir que un administrador defina nuevas localizaciones de almacenamiento.
- Resumen: El desarrollador puede actualizar la aplicación desde GitHub y ponerla a disposición de futuros usuarios aportadores de información.

Caso de uso 2		
	Actor	Sistema
1	Administrador: descarga el repositorio de RSMapPi y lo configura con nuevos parámetros.	
		2 La nueva configuración de los clientes permite enviar información a el destino seleccionado por el administrador .

Tabla 4.3: Secuencia de CU-2

CU-3. Consulta de datos mediante la web.

- Actores: Usuario visitante.
- Tipo: Primario, esencial.
- Referencias:
- Precondición: La aplicación web debe estar en funcionamiento.
- Postcondición: Se muestran identificadores en los lugares en los que se encuentra un dispositivo receptor y detecta tráfico.
- Autor: Cualquier usuario que acceda a la plataforma.
- Versión: 1.0.
- Propósito: Que el usuario tenga conocimiento de por qué zonas existe tráfico.
- Resumen: El usuario entra en la web y puede ver el tráfico en tiempo real.

Caso de uso 3		
	Actor	Sistema
1	Usuario: accede en la web para visualizar el tráfico	
		2 Muestra los marcadores en los lugares en los que existe tráfico en ese momento.

Tabla 4.4: Secuencia de CU-3

Curso alterno	
2b	Si no existen dispositivos retransmitiendo la web informa al usuario.

Tabla 4.5: Curso alterno de CU-3.

■ **CU-4.** Consulta de datos masivos.

- Actores: Usuario visitante.
- Tipo: Primario, esencial.
- Referencias:
- Precondición: La aplicación web debe estar en funcionamiento y la base de datos debe contener registros.
- Postcondición: Se muestran los datos según la consulta enviada.
- Autor: Cualquier usuario que acceda a la plataforma.
- Versión: 1.0.
- Propósito: Que el usuario tenga libre acceso a la información recopilada por los dispositivos que agregan información.
- Resumen: El usuario entra en la web y puede consultar los datos almacenados bajo las condiciones que el desee.

Caso de uso 4		
	Actor	Sistema
1	Usuario: accede en la web y lanza una consulta	
		2 Devuelve todas las entradas que cumplen las condiciones de la consulta.

Tabla 4.6: Secuencia de CU-4

Curso alterno	
2b	Si no existen datos en base a la consulta el sistema no devuelve nada y se muestra un error.

Tabla 4.7: Curso alterno de CU-4.

■ **CU-5.** Añadir nuevos dispositivos.

- Actores: Aportador de información.
- Tipo: Primario, esencial.
- Referencias:
- Precondición: Tiene un dispositivo capaz de remitir información al sistema.
- Postcondición: Los datos que remitidos quedan guardados en el sistema.
- Autor: Administrador, Usuario Aportador.
- Versión: 1.0.
- Propósito: Que cualquier usuario pueda aportar información al sistema.
- Resumen: El usuario se descarga desde GitHub el cliente y lo ejecuta para enviar información del tráfico en su localización.

Caso de uso 5		
	Actor	Sistema
1	Usuario: se descarga el cliente y lo configura para el envío	
		2 Almacena los datos remitidos por el usuario aportador.

Tabla 4.8: Secuencia de CU-5

4.3. Diagramas

4.3.1. Diagrama de paquetes

En este diagrama podemos observar las dependencias que existen entre los distintos paquetes del sistema y como el paquete **IOT** es el elemento central sobre el que se apoyan los demás. El paquete **Raspberry Pi** es dependiente del paquete de **IOT** debido a que **IOT** es el encargado de definir como los clientes van a escribir en el almacenamiento masivo. A su vez el paquete **RSMap web** es dependiente de **Raspberry Pi** debido a que las señales de vehículos se mandarán directamente a la plataforma web gracias a la **API REST**.

Por otra parte tenemos el paquete **Visualizador** el cual usa al paquete **Almacenamiento masivo** para proveerse de información que mostrar. Por último **Almacenamiento masivo** es dependiente de **IOT** debido a que los esquemas (estructura de almacenamiento) se definen en **IOT** y éste se encarga de definirlas en el **Almacenamiento masivo**.

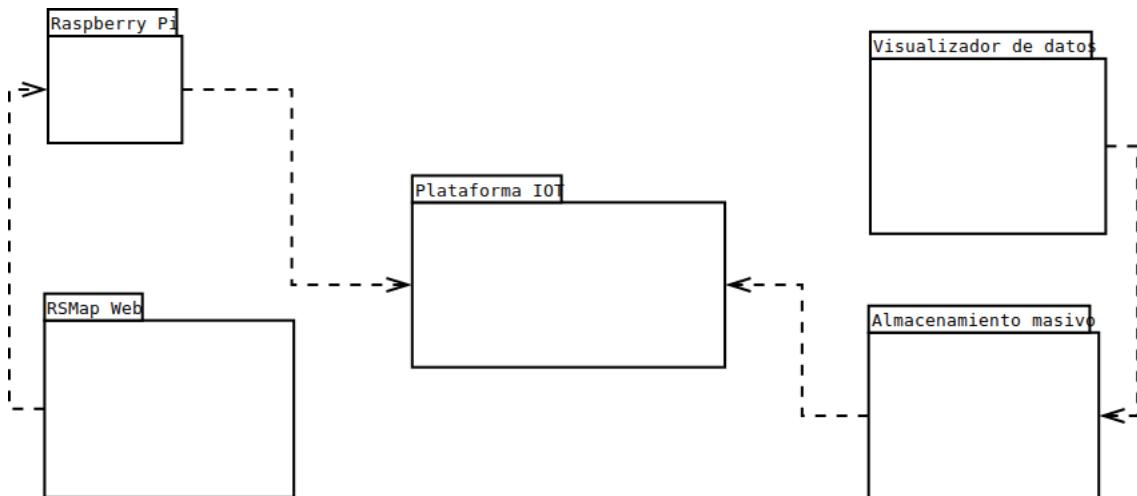


Figura 4.2: Diagrama de paquetes

4.3.2. Diagrama arquitectónico

En ésta sección se detallan los elementos de la estructura que compondrá la aplicación. Debido a que los dispositivos receptores (**Raspberry pi**) poseen una estructura interna la cual merece atención se han detallado por separado el diagrama de los receptores (*Figura 4.3*) y el del sistema completo (*Figura 4.12*).

El sistema recibirá la entrada de audio mediante un micrófono, tras ésto el módulo encargado de recolectar los datos efectuará un filtrado previo evitando así analizar datos innecesarios de ésta manera minimizamos la sobrecarga del dispositivo. Tras ésto se procede a analizar los datos que han pasado el primer filtro. Por último se envían los datos con los resultados a los distintos sistemas de almacenamiento que se detallan en el siguiente diagrama.

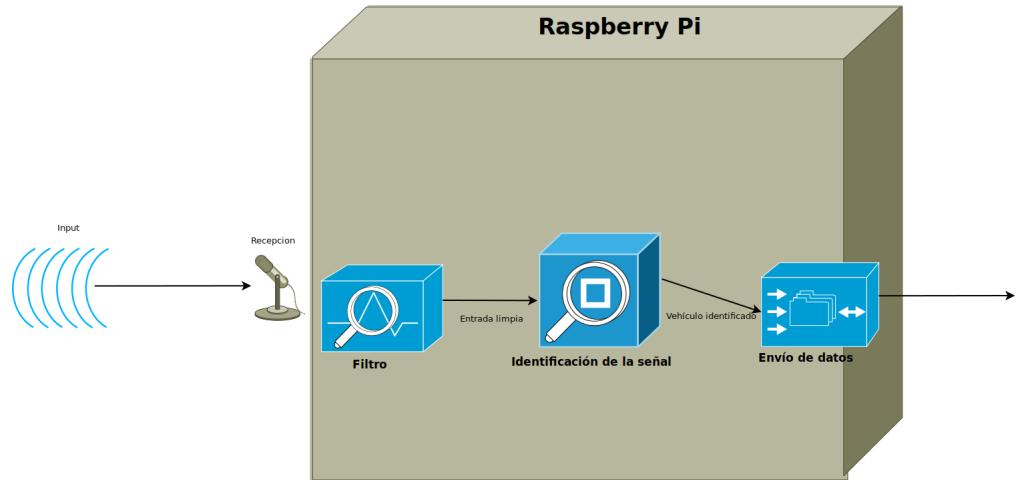


Figura 4.3: Arquitectura Raspberry Pi

Ahora se muestra cual será la estructura en conjunto de todos los elementos que conforman **RSMMap**.

Tras haber analizado el comportamiento de los dispositivos receptores, el siguiente elemento a destacar es la plataforma IOT, la manera de transmitir la información según la configuración establecida, en la que se detalla a donde tienen que remitir los dispositivos la información así como los propios modelos de datos. Los receptores harán uso de ésto para enviar la información a **Almacenamiento masivo**.

Por otra parte, también realizarán envíos a el servidor web indicando cuando pasa un vehículo por la localización en la que se encuentren, el cual atenderá las peticiones de los clientes que soliciten acceder a RSMMap.

Como último elemento nos queda **Visualizador**, que actúa a modo de Notebook y es el encargado de realizar las consultas que el usuario define a **Almacenamiento masivo** para poder acceder a los datos que han almacenado los dispositivos receptores.

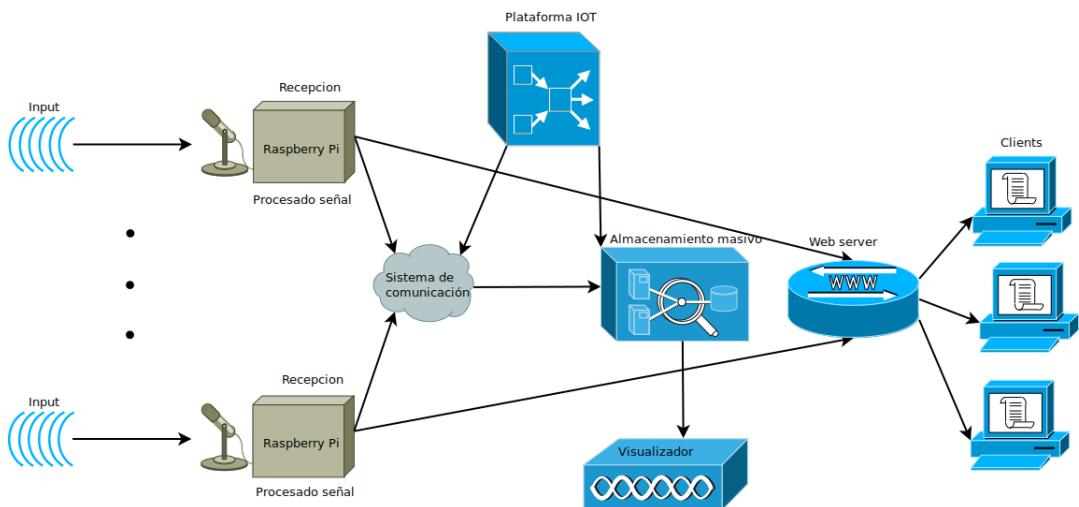


Figura 4.4: Arquitectura de RSMMap

4.3.3. Diagrama de clases

En éste diagrama se presentan los elementos de RSMap como clases y cuales serían los métodos y campos ideales para cada uno de ellos.

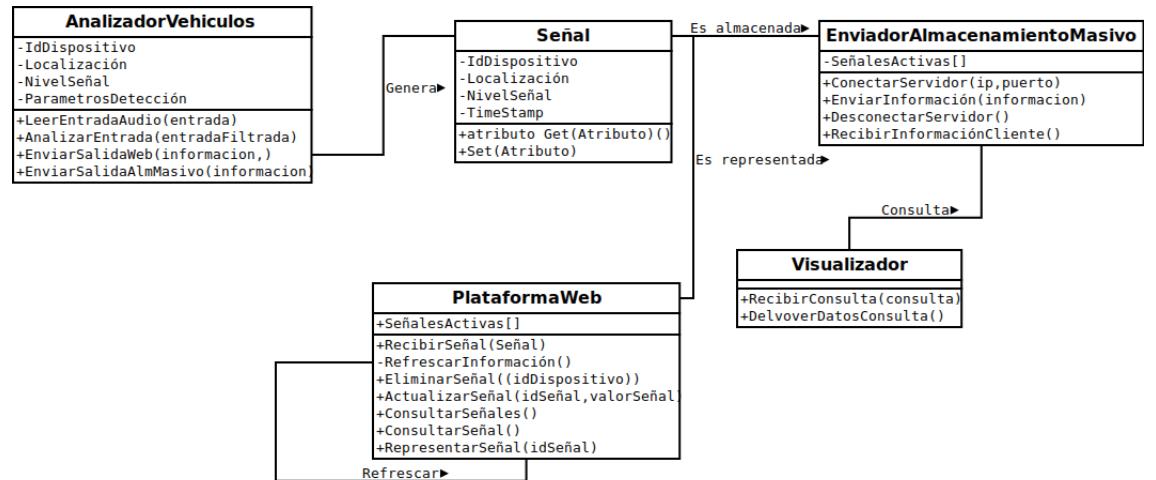


Figura 4.5: Diagrama de clases

4.3.4. Diagramas de casos de uso

Ahora detallaremos de qué manera pueden interactuar los distintos actores con el sistema. A modo de vista general se proporciona un diagrama de Caso de uso global.

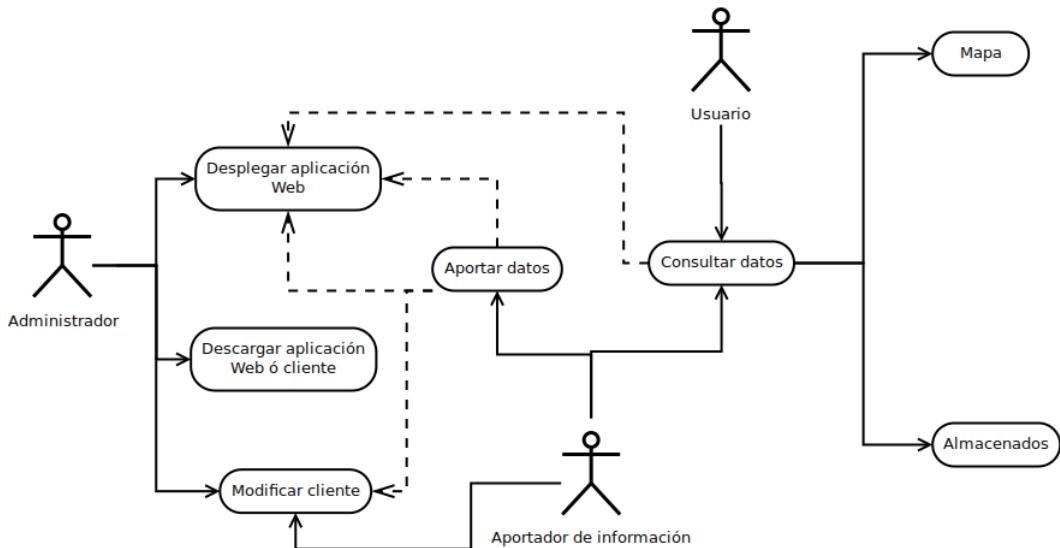


Figura 4.6: Caso de uso general

El administrador tiene capacidad para realizar actividades de mantenimiento y chequeo de los parámetros usados, así como generar nuevas configuraciones.

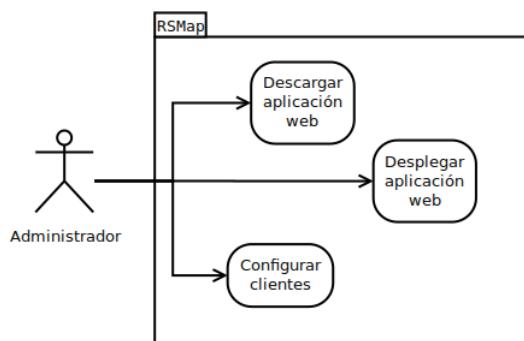


Figura 4.7: Caso de uso del administrador

En segundo lugar tenemos el usuario que puede consultar los datos bien sea mediante el mapa web, el cual sólo indica señales en los puntos en los que pasen vehículos, mientras que mediante el visualizador de datos masivos podrá consultar el conjunto total de información almacenada.

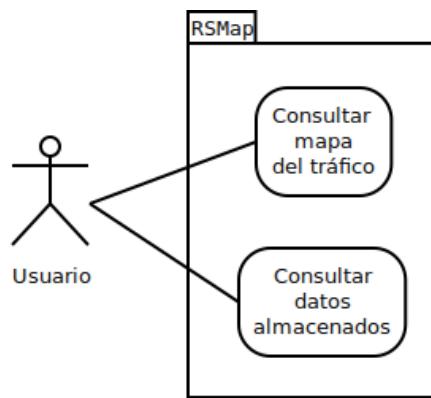


Figura 4.8: Caso de uso del usuario

Por último el usuario aportador podrá bajarse el código fuente desde GitHub y podrá ejecutarlo para enviar la información a RSMap. Como se trata de una especialización del Usuario, se da por entendido que también tendrá acceso a los datos.

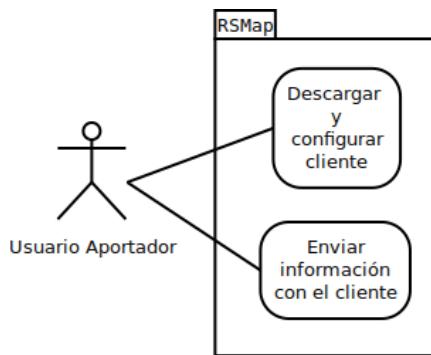


Figura 4.9: Caso de uso del usuario aportador

4.3.5. Diagramas de secuencia

Ahora se detallan los pasos de forma generalizada que cada tipo de usuario realiza y qué efectos desencadenan sobre el sistema.

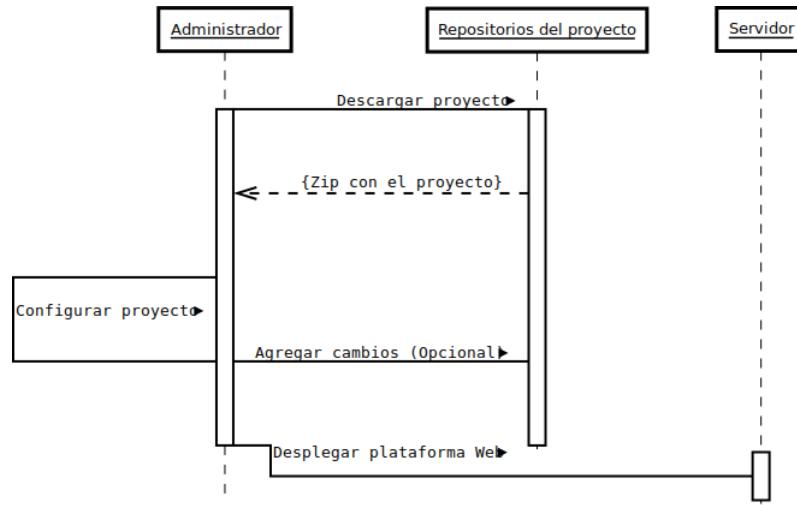


Figura 4.10: Diagrama de secuencia del Administrador

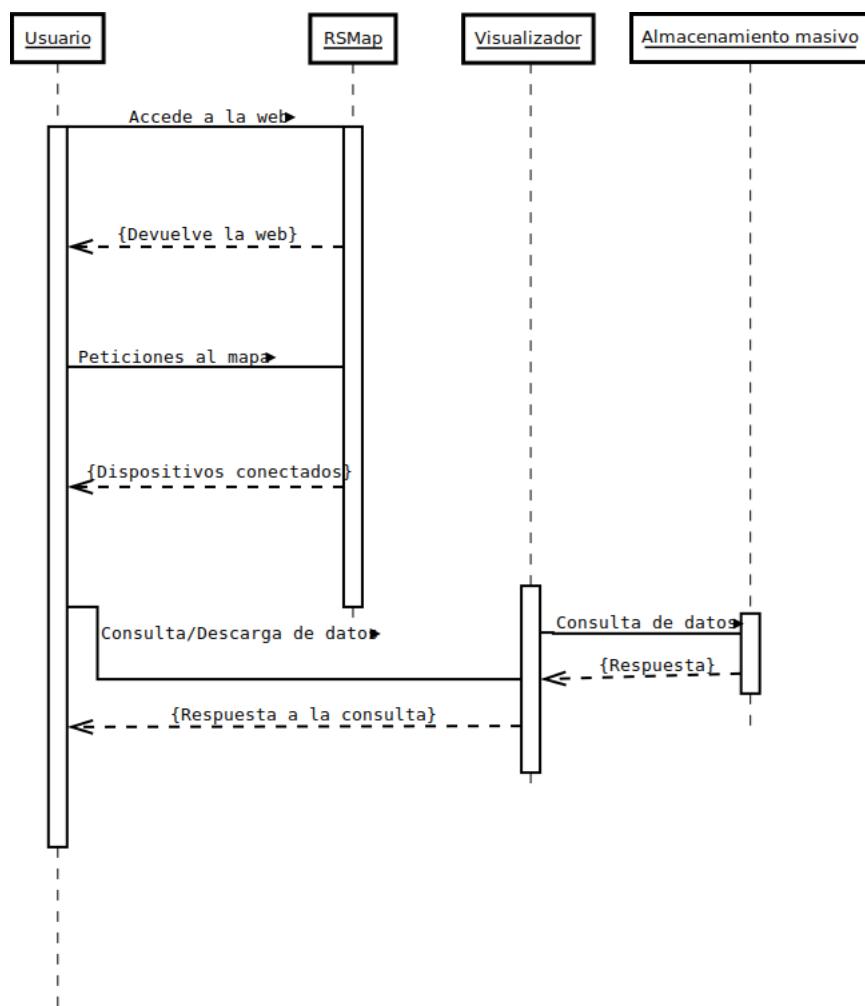


Figura 4.11: Diagrama de secuencia del Usuario

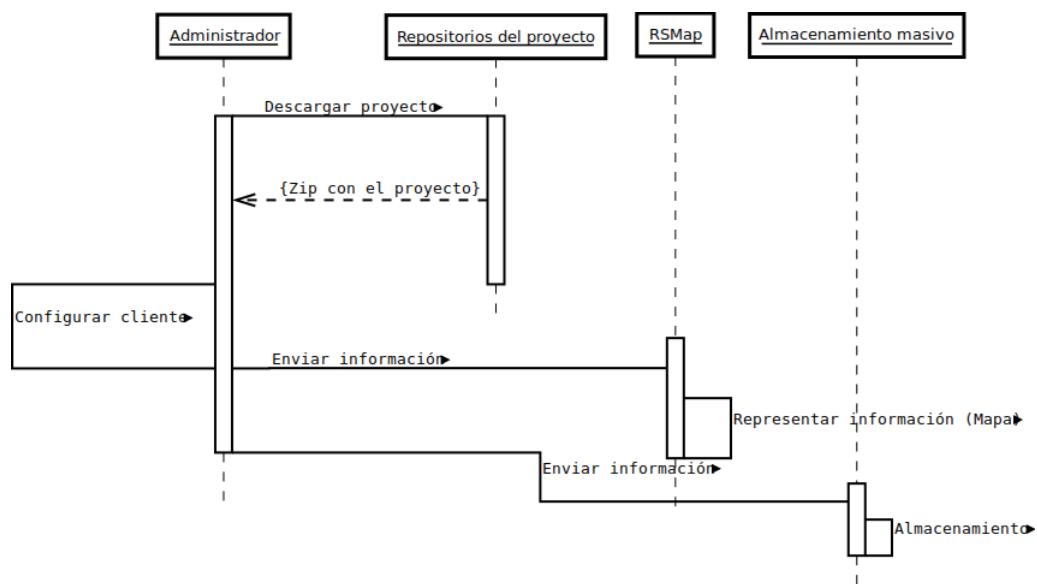


Figura 4.12: Diagrama de secuencia del Usuario Aportador

4.3.6. Diagramas de interfaz

Aquí se presentan dos pequeños esbozos de una aproximación a la interfaz de la web, no se ha entrado en mucho nivel de detalle debido a que la idea es usar una plantilla de Bootstrap y adaptarla a las necesidades, en cualquier caso el aspecto deberá asemejarse al ilustrado en la *Figura 4.13* para la página principal y para la página del mapa el de la *Figura 4.14*.

**Heading**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla quam velit, vulputate eu pharetra nec, mattis ac neque. Duis vulputate commodo lectus, ac blandit elit fincidunt id. Sed rhoncus, tortor sed eleifend tristique, tortor mauris molestie elit, et lacinia ipsum quam nec dui. Quisque nec mauris sit amet elit iaculis pretium sit amet quis magna. Aenean velit odio, elementum in tempus ut, vehicula eu diam. Pellentesque rhoncus aliquam mattis. Ut vulputate eros sed felis sodales nec vulputate justo hendrerit. Vivamus varius pretium ligula, a aliquam odio.



Figura 4.13: Boceto de la página principal

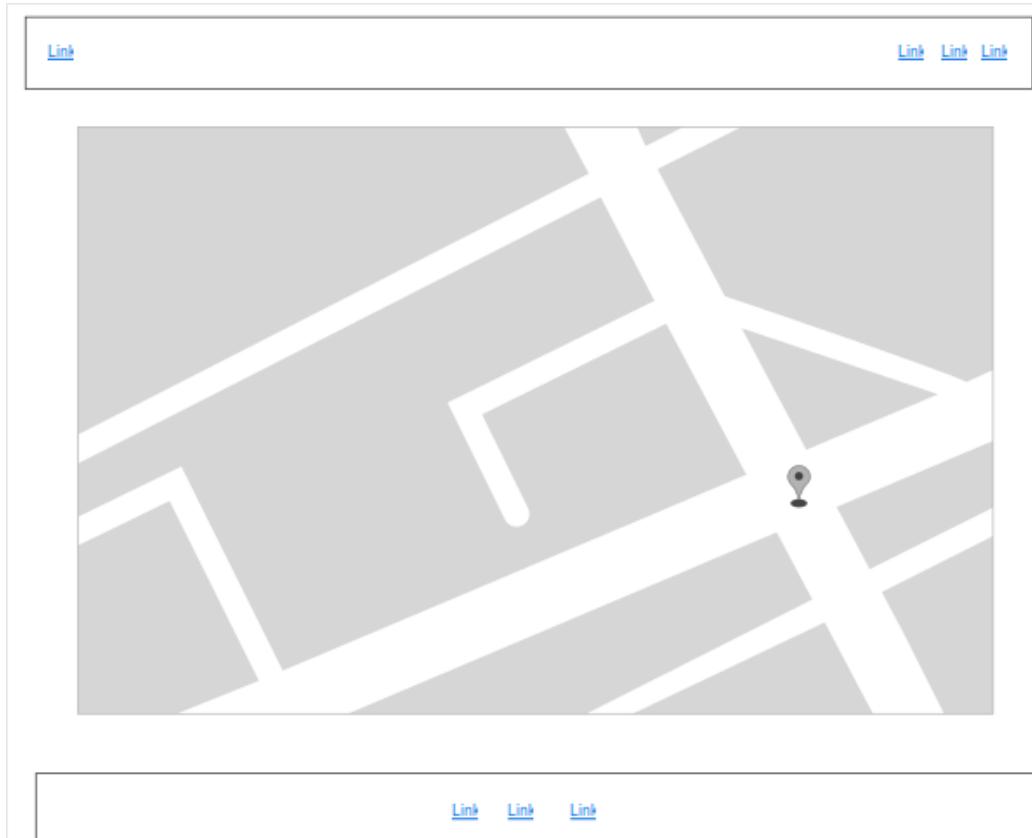


Figura 4.14: Boceto de la página del mapa

Capítulo 5

Diseño

Éste apartado muestra una visión simplificada de como funcionará la aplicación, será en la sección de *Implementación* donde se completen los aspectos restantes cuya explicación resulta más fácil valiéndose del código usado además el porqué de la elección de las tecnologías con las que se llevan a cabo las tareas.

5.1. Diseño del dispositivo receptor

El dispositivo receptor tendrá instalado y en continua ejecución el módulo de reconocimiento de vehículos. Los elementos hardware requeridos se detallaron en la sección *Presupuesto*.

5.2. Diseño del módulo de reconocimiento de vehículos

El lenguaje usado para la recepción de sonido será `Python 3` debido a la facilidad de uso y el número de librerías que presenta. El módulo principal estará compuesto de dos funciones usando como base un patrón típico en sistemas concurrentes, el modelo de Productor-Consumidor. Ésto nos permitirá trabajar con dos hebras independientes (pero pertenecientes al mismo proceso) las cuales trabajan sobre una cola en la que la hebra productora va introduciendo los valores numéricos obtenidos por el micrófono mientras que el consumidor va extrayendo los mismos de la cola para su análisis, el cual se detalla en la subsección Desarrollos algorítmicos.

Una vez el consumidor ha concretado que la información recibida es indicador de que un vehículo ha pasado tiene dos tareas:

- Enviar la información a la web mediante una petición HTTP que incluya un *payload* con la información relativa al dispositivo por ejemplo latitud, longitud, id del dispositivo y nivel de la señal. Las peticiones

serán aceptadas en el servidor web gracias a una api REST que atiende las peticiones de éste tipo.

- Transmitir la información al almacenamiento masivo mediante el soporte que nos provea la plataforma IOT KAA. La forma de proceder aquí es dependiente de cómo se interactúe con la plataforma por tanto los detalles de como el dispositivo envía la información en éste punto serán vistos en el sección de implementación.

5.2.1. Estructuras de datos en el dispositivo receptor

La estructura de datos más remarcable en éste módulo es la cola, sin duda es el elemento más apropiado para la recopilación y extracción de datos de manera simultánea. Además las colas en python están diseñadas para solventar problemas de tipo concurrente de manera interna.

Para transmitir la información a la web se usarán diccionarios de tipo *'clave':valor* en formato JSON cuyo tamaño reducido los hace ideales para enviarlos de manera recurrente. Una aproximación sería:

```

1 {
2   "device_id": "Ronda",
3   "lat": "0.000017",
4   "long": "0.000017",
5   "created": "1473697709",
6   "level": 56.5,
7   "type": "u"
8 }
```

Fragmento de código 5.1: Formato a utilizar

Por otra parte tenemos la estructura de datos Señal, que almacenará toda la información necesaria para ser almacenada/representada en RSMap. Se hace uso de ella en el módulo de envío proporcionado por KAA, en la base de datos de almacenamiento masivo Apache Cassandra y en los modelos definidos en Django para la web.

5.2.2. Desarrollos algorítmicos

A continuación se explica cuál es la forma de proceder a la identificación de un vehículo mediante el sonido recogido, por eso es imprescindible hablar del análisis de señales además del algoritmo usado.

- Digitalización de señales:

- Para obtener una señal discreta a partir de una señal continua, es decir, para convertir los datos del dominio del tiempo al dominio de la frecuencia nos valdremos de la transformada rápida de Fourier (FFT).
 - Con los valores discretizados procedemos a multiplicarlos por una ganancia configurable mediante una constante, que ayudará a aumentar la señal o disminuirla dependiendo del tipo de micrófono y tarjeta de audio que tengamos. Los valores serán contabilizados en forma de bloques (que están definidos en milisegundos), se suman los valores que pertenecen a un mismo bloque descartando aquellos que no entran dentro del rango de cuantización establecido que nos indique que se trata de un vehículo. Éste valor final es puesto en la cola sobre la que la hebra consumidora va extrayendo datos.
- **Identificación de vehículos:** Una vez tenemos la suma de los valores para un bloque, en primer lugar comprobamos que la suma de ese bloque alcanza un nivel mínimo, éste nivel dependerá de las condiciones en las que se encuentre el dispositivo receptor. Por ejemplo, no es lo mismo que se encuentre situado en el primer piso de un edificio que en un tercero.

Una vez tenemos filtrados los valores que superan ese mínimo es momento de contar cuantos bloques seguidos se obtienen por encima de una cota mínima de bloques. Al igual que hemos definido la cota mínima para considerar que existe un nivel de sonido perteneciente a un vehículo debemos definir el número de bloques consecutivos que necesitamos para considerar que un vehículo ha pasado y éste valor será dependiente de igual modo de las características de la carretera ya que el número de bloques necesarios para la identificación está estrictamente relacionado con la velocidad de los vehículos que circulan por la calle, es decir, cuanta más velocidad lleven menos bloques serán necesarios para proceder a la identificación del vehículo.

El valor de cada bloque válido (que supera la cota), es almacenado en una cola, cuando el número de bloques es suficiente para concretar que un vehículo ha pasado los valores almacenados en la cola se mandan a la base de datos de almacenamiento masivo así como a la API y ya estarán listos para su visualización.

Para ilustrarlo con un ejemplo nos valdremos de la *Figura 5.1*.

Cuando el vehículo pasa dentro del rango de captura del micrófono empiezan a contabilizarse los bloques con un valor superior a la cota definida desde **B1** hasta **BN**. Si tenemos un número de bloques válido contabilizaremos que ha pasado un vehículo. Si el número de bloques

se ve interrumpido por 'silencio' o sonido por debajo de la cota, la contabilización de bloques se reinicia y en la cola de envío son descartados todos los elementos pertenecientes a los bloques consecutivos anteriores.



Figura 5.1: Proceso de reconocimiento

5.3. Almacenamiento de información

La información será enviada desde los dispositivos recolectores hasta el servidor que contiene la base de datos de Apache Cassandra que se encontrará instalada en una instancia propia para garantizar que todos los recursos de la máquina se dedican a atender las peticiones de almacenamiento. Por otra parte las peticiones HTTP se realizarán mediante ficheros JSON contra una API Rest construida bajo Django Rest Framework.

Ésta base de datos será de tipo NoSQL debido a que no se requiere ningún tipo de jerarquía sobre los datos y tampoco se precisa una estricta integridad de los mismos ni operaciones de tipo atómico.

Por otra parte tenemos la base de datos de la que hará uso la web de RSMap para la representación en el mapa. Al ser datos de carácter temporal cuyo formato no mantiene ningún tipo de jerarquía relacional se usará SQLite.

5.4. Representación de información

Este módulo trabaja con Apache Zeppelin que también se encontrará en un servidor externo a los demás. En la configuración se establecerá la conexión hacia el servidor de almacenamiento masivo y a partir de ahí, se podrán lanzar consultas para obtener datos.

5.5. Diseño del portal web

El portal estará desarrollado con el framework Django, el cual hace uso del paradigma modelo Vista-Controlador por lo que tendremos por una parte la parte lógica o Backend y por otro lado la parte visual o Frontend. Tendrá la siguiente estructura de archivos y directorios:

- **resources**: archivos complementarios (no necesarios).
- **rsmap**: contiene el proyecto principal de la aplicación en el que se definen los parámetros generales.
 - *urls.py*: definición de las rutas globales de la web.
 - *wsgi.py*: configuración de despliegue.
 - *settings.py*: archivo de configuración general.
- **rsmapweb**: contiene la aplicación web y la API rest.
 - **templates**: contiene las plantillas html de la web.
 - (*index.html*) alberga la página principal.
 - (*map.html*) representa la página que contiene el mapa.
 - **static**: contiene los archivos estáticos de la web (css, fuentes, etc).
 - *js*: es el directorio más importante ya que contiene el script encargado de interactuar con el mapa (*rsmapMapUpdater.js*)
 - *admin.py* archivo para configurar la administración del portal.
 - *apps.py*: archivo en el que se configura las aplicaciones instaladas de la web.
 - *models.py*: modelos de datos que serán usados.
 - *urls.py*: definición de las rutas de la aplicación y la API rest.
 - *serializers.py*: define qué información devolverá al API rest.
 - *views.py*: definición de vistas
- *manage.py*: encargado de ejecutar tareas de lanzar la aplicación además de actualizar los modelos de datos en la base de datos.
- *requirements.txt*: dependencias necesarias para la aplicación.
- *Makefile*: contiene la automatización de las tareas más comunes de la aplicación.

La API en JavaScript de Google Maps nos proporcionará las herramientas necesarias para generar y alimentar un mapa con la información que deseemos.

En lo que a la parte visual se refiere, se usará Bootstrap para garantizar un diseño adaptativo al formato de dispositivo desde el cual se consulte la información.

5.5.1. Diseño de la api REST

La api estará integrada dentro de Django gracias al módulo Django Rest Framework, es por éste motivo por lo que se desarrolla dentro de la sección *Diseño del portal web*.

Una API se caracteriza por ofrecer una serie de URL's sobre las que realizar peticiones HTTP de diverso tipo como podrían ser POST, GET, PATCH, PUT o DELETE entre otras.

Éstas peticiones contienen un *payload* con la información necesaria para que, una vez recibida la petición, la API se encargue internamente de interactuar con los modelos creados en Django que afecten a la misma.

Las rutas tendrán una estructura similar a la siguiente:

- **Consultar listado o añadir dispositivos al mismo:**

`http://urlweb/api/signals/`

Devolverá un JSON con los dispositivos que están retransmitiendo información en ese momento (GET), además ésta ruta debe permitir registrar nuevos dispositivos (POST) por lo que los métodos permitidos en ella serán POST y GET.

- **Actualizar estado de dispositivo:**

`http://urlweb/api/signal/signalID`

Con ésta url podremos obtener los detalles de esta señal en concreto (GET), actualizar el estado de una señal (PATCH) o eliminarla en consecuencia de que el dispositivo deje de retransmitir (DELETE), de ésta manera el servidor web no tendrá que descargar continuamente el JSON de todos los dispositivos para representarlo en el mapa si no únicamente aquellos que se encuentren retransmitiendo en ese momento. Por tanto los métodos permitidos para ésta URL serán GET, PATCH y DELETE.

Capítulo 6

Implementación

A continuación se detallan la instalación, configuración así como el desarrollo de los elementos que componen el sistema. A modo de resumen los puntos serán los siguientes:

- Descripción de las tecnologías seleccionadas.
- Instalación y configuración de Raspberry Pi.
- Instalación y configuración de Kaa.
- Instalación y configuración de Apache Cassandra.
- Desarrollo del módulo de detección de vehículos.
- Desarrollo de la plataforma web.
- Instalación y configuración de Apache Zeppelin.

6.1. Descripción de las tecnologías seleccionadas

Éste apartado detalla las tecnologías seleccionadas así como las que finalmente fueron descartadas.

A lo largo del desarrollo del proyecto el diseño ha ido cambiando progresivamente debido a que el número de elementos que lo componen deben cohesionarse para trabajar sobre la solución, siendo éste uno de los mayores problemas encontrados ya que al elegir una tecnología se deben desarrollar una serie de pruebas para confirmar que es la óptima de cara al diseño final.

6.1.1. Plataforma IOT

Este aspecto ha sido uno de los más relevantes, el cual ha marcado el funcionamiento final de la aplicación. Existen muchas alternativas debido al auge de IOT no obstante la plataforma debía cumplir una serie de premisas enumeradas a continuación:

- **Open Source:** para mantener el licenciamiento de la aplicación final.
- **Independiente:** esto es, debe ser un sistema instalable en un servidor propio sin necesidad de depender de servidores de terceros.
- **Soporte para Raspberry Pi:** es obvio que debe poderse integrar con nuestro dispositivo receptor.
- **Número de dispositivos ilimitado:** esto nos permitirá cumplir con el requisito de que cualquiera pueda proveer datos a la aplicación.
- **Documentación:** es un elemento importante, más aún, cuando se desconoce la materia.

El abanico de posibilidades es muy extenso pero no todas cumplen con los requisitos deseados, las opciones finalmente descartadas fueron:

- **Thinger.io** cubre alguna de las necesidades que se precisan sin embargo la documentación muy es pobre.
- **IBM Watson** es un candidato a tener en cuenta sin embargo trabajar en servidores de terceros y ser de pago ha sido un factor decisivo para descartarlo.
- **AWS IOT** de similares características a IBM Watson, ha sido descartado por las mismas razones que el anterior.
- **Node-RED** es el perfil que más se acerca a lo requerido, no ha sido elegido porque la alternativa finalmente escogida tiene una comunidad más activa y por tanto la documentación es mejor.

Esto nos lleva a **Kaa** (<http://www.kaaproject.org/>) la cual cumple todos los requisitos deseados completamente.



"Kaa is a feature-rich, open-source IoT middleware platform for rapid development of the Internet of Things solutions, IoT applications, and smart products."

<http://www.kaaproject.org/>

Kaa provee una imagen para ser instalada sobre Amazon EC2 lo que simplifica mucho la tarea de instalación y configuración, además cuenta con una extensa documentación, webinars y ejemplos que han sido de gran ayuda a la hora del desarrollo.

Su arquitectura se puede ver en la *Figura 6.1*, actúa como middleware entre nuestra aplicación y los dispositivos que se desean conectar, para ello cuenta con un servidor web (instalado en la imagen que proveen) en el cual podemos configurar las aplicaciones que deseemos. Para cada una de éstas aplicaciones se genera un SDK en el lenguaje deseado el cual será usado para enviar ó recibir información.



Figura 6.1: Arquitectura de Kaa

6.1.2. Dispositivo receptor

En éste aspecto no había muchas dudas en cuanto a las tecnologías a usar. Por una parte tenemos Raspbian, el sistema operativo oficial de Raspberry Pi basado en Debian. Usar algún otro sistema operativo ARM implicaría la incertidumbre sobre cualquier tipo de error en gran parte por el aspecto de los drivers.

Por otra parte tenemos el software desarrollado que se ejecuta en el dispositivo. Al igual que con el sistema operativo, el lenguaje de programación elegido desde un principio fue Python por su sencillez y número de bibliotecas. A medida que el desarrollo ha ido evolucionando Python ha demostrado ser el mejor candidato para el uso que se le pretendía dar, en gran parte gracias a la biblioteca **SoundDevice** que nos proporciona una manera realmente fácil de acceder y capturar datos de las interfaces de audio además de contar con numerosos ejemplos los cuales han sido de gran ayuda.

Por último, el dispositivo necesita hacer uso del SDK generado por Kaa. Existen varias alternativas a la hora de generarla y la más atractiva resultó ser Java. La interacción entre Python y Java se detallará más adelante.

6.1.3. Almacenamiento de información

Para almacenar datos de manera masiva existen varias alternativas como son Apache HBase, MongoDB, Apache Cassandra, Hive ó Redis. En éste aspecto la diferencia entre unas y otras es menores sin embargo hay una de ellas que destaca respecto a las otras por la facilidad de integración con Kaa, es **Apache Cassandra** (<http://cassandra.apache.org/>).



"... is the right choice when you need scalability and high availability without compromising performance... Is best-in-class, providing lower latency for your users..."

<http://cassandra.apache.org/>

Tras una breve documentación acerca de Cassandra se vió que era un software con unas características y una potencia a tener muy en cuenta además de su facilidad integración con Kaa como se ha mencionado anteriormente, también se pueden definir funciones personalizadas en el lenguaje deseado y tiene una línea de comandos amigable. El lenguaje de consultas es CQL y su sintaxis no difiere mucho del resto de SGBD.

6.1.4. Representación de información

En éste apartado las opciones a tener en cuenta era Apache-Zeppelin y Jupyter. Ambas herramientas tienen el papel de representar la información que va a almacenar Cassandra. La integración entre la base de datos y Zeppelin se hace de una manera sencilla (ambas pertenecen a Apache) así como las consultas y visualización de datos, es por eso por lo que finalmente se optó por la opción de **Apache Zeppelin** (<https://zeppelin.apache.org/>).



*"A web-based notebook that enables interactive data analytics.
You can make beautiful data-driven, interactive and collaborative
documents..."*

<https://zeppelin.apache.org/>

Apache Zeppelin nos va a permitir configurar una serie de Notebooks en forma de plataforma web en los que podremos realizar consultas a Cassandra para obtener los datos almacenados de los dispositivos receptores. Los datos se obtendrán mediante consultas CQL y podrán ser representados de varias formas, incluyendo distintos tipos de gráficos así como permitiendo la exportación de los mismos.

6.1.5. Plataforma Web de RSMap

Para la plataforma web aunque las opciones eran varias en términos de lenguajes y frameworks. Finalmente la tecnología usada será Django que es un framework para aplicaciones web en Python.



Las principales causas para la elección son la familiaridad con el mismo, que trabaja bajo el modelo Vista-Controlador, su documentación y el módulo Django-Rest-Framework que permite a los dispositivos comunicarse directamente con la web que será la encargada de representar en un mapa pequeños iconos que indiquen el paso de un vehículo.

El mapa es provisto por Google Maps. Para ofrecer una navegación fluida concretamente en la sección del mapa se hará uso de jQuery y AJAX que nos sirven métodos para actualizar la información en el mismo mapa de manera automática y sin necesidad de recargar la página para ver los cambios.

El aspecto visual queda resuelto gracias a la librería Bootstrap (CSS) y una plantilla (HTML5) predefinida Open Source sobre la que efectuaremos las modificaciones necesarias para adaptarla a las necesidades de RSMap.

6.2. Instalación y configuración de Raspberry Pi

6.2.1. Instalación

En primer lugar se indica como instalar Raspbian en nuestra Raspberry Pi Jessie (basado en Debian), ésta ha sido la opción seleccionada como SO debido a que es el sistema oficial que provee Raspberry Pi.

El primer paso es descargarse la imagen de los servidores oficiales, la imagen usada se puede encontrar en <https://downloads.raspberrypi.org/raspbian/images/raspbian-2016-05-31/>

```

1 # muestra los discos del sistema
2 $ lsblk
3 # copia la imagen en el dispositivo /dev/sdb (tarjeta sd)
4 $ sudo dd bs=1M if=2016-05-27-raspbian-jessie.img of=/dev/sdb
5 # copia la particion del sistema en /dev/sdc (usb)
6 $ sudo dd bs=1M if=/dev/sdb2 of=/dev/sdc
  
```

Fragmento de código 6.1: Copia de Raspbian en tarjeta s y usb

Ahora editamos el archivo **/dev/sdb/boot/cmdline.txt** y añadimos la siguiente línea para que el sistema sólo use la tarjeta SD para arrancar y el dispositivo usb como disco del sistema, ésto aumentará la velocidad de lectura considerablemente:

```
1 smsc95xx.turbo_mode=N dwc_otg.lpm_enable=0 console=ttyAMA0
   ,115200 kgdboc=ttyAMA0,115200 console=tty1 root=/dev/sda1
   rootfstype=ext4 elevator=noop rootwait # /dev/sda1 point to
   our USB drive
```

Fragmento de código 6.2: Modificando el dispositivo de arranque de Raspbian

6.2.2. Configuración

Vamos a instalar el software necesario para interactuar con la tarjeta de sonido además de comprobar si el sistema la soporta y en caso afirmativo, establecerla como dispositivo de audio por defecto lo cual nos permitirá trabajar con ella después de cada reinicio ó apagado del dispositivo.

```
1 $ sudo apt-get install alsa-utils
```

Fragmento de código 6.3: Instalación del paquete alsa-utils

```
1 $ lsusb
2 Bus 001 Device 005: ID 0d8c:000c C-Media Electronics, Inc. Audio
   Adapter
3 Bus 001 Device 004: ID 0781:5567 SanDisk Corp. Cruzer Blade
4 Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.
   SMSC9512/9514 Fast Ethernet Adapter
5 Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
6 Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Fragmento de código 6.4: Comprueban que el dispositivo es reconocido

```
1 $ aplay -l
2 **** List of PLAYBACK Hardware Devices ****
3 card 0: ALSA [bcm2835 ALSA], device 0: bcm2835 ALSA [bcm2835
   ALSA]
4 Subdevices: 8/8
5 Subdevice #0: subdevice #0
6 Subdevice #1: subdevice #1
7 Subdevice #2: subdevice #2
8 Subdevice #3: subdevice #3
9 Subdevice #4: subdevice #4
10 Subdevice #5: subdevice #5
11 Subdevice #6: subdevice #6
```

```

12 Subdevice #7: subdevice #7
13 card 0: ALSA [bcm2835 ALSA], device 1: bcm2835 ALSA [bcm2835
    IEC958/HDMI]
14 Subdevices: 1/1
15 Subdevice #0: subdevice #0
16 card 1: Set [C-Media USB Headphone Set], device 0: USB Audio [
    USB Audio]
17 Subdevices: 1/1
18 Subdevice #0: subdevice #0

```

Fragmento de código 6.5: Identificando los dispositivos de sonido disponibles en el sistema

En éste caso el dispositivo se llama *C-Media USB Headphone Set* cuyo identificador es 1, para establecerla como dispositivo de audio por defecto editamos */home/user/.asoundrc* con el siguiente contenido:

```

1     pcm.!default {
2         type hw
3         card 1
4     }
5
6     ctl.!default {
7         type hw
8         card 1
9     }

```

Fragmento de código 6.6: Comprueban que el dispositivo es reconocido

6.3. Instalación y configuración de Kaa

6.3.1. Instalación

Para la instalación de Kaa nos bastará con Amazon EC2, ya que el sistema de imágenes de SO contiene la versión más actual de Kaa, los pasos a seguir son los siguientes:

Necesitamos una cuenta en Amazon EC2, tras registrarnos nos dirigimos al panel principal, y en el apartado *INSTANCES* seleccionamos, *Launch Instance* para crear una nueva máquina.

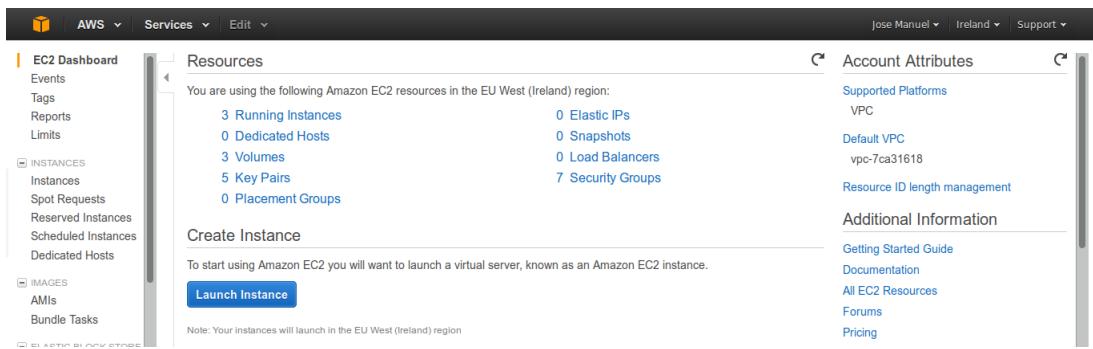


Figura 6.2: Dashboard de AmazonEC2

El segundo paso es dirigirse a *Community AMIs* y en el cuadro de búsqueda escribimos *kaa-sandbox*. Lo recomendable es usar la última versión, en éste caso se ha tomado la **versión 0.9**.

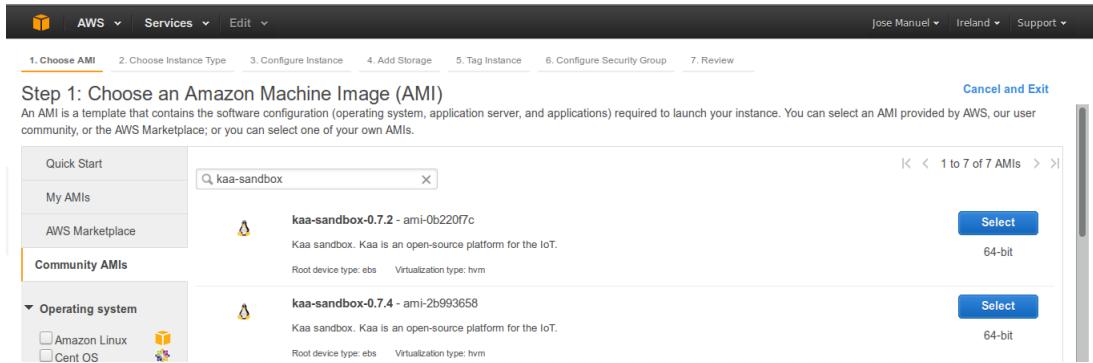


Figura 6.3:

Ahora debemos elegir el tipo de instancia, esto va directamente relacionado con la potencia de la misma. En un principio nos puede valer con las instancias de tipo *free tier* y configurar la escalabilidad a medida de nuestras necesidades.

The screenshot shows the AWS EC2 console interface. The top navigation bar includes 'AWS Services' and 'Edit'. Below it, a progress bar shows steps 1 through 7: '1. Choose AMI', '2. Choose Instance Type' (which is currently selected), '3. Configure Instance', '4. Add Storage', '5. Tag Instance', '6. Configure Security Group', and '7. Review'. The main content area is titled 'Step 2: Choose an Instance Type'. A sub-section title 'Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)' is displayed. A table lists instance types based on 'Family' (General purpose), 'Type' (t2.nano, t2.micro, t2.small), 'vCPUs', 'Memory (GiB)', 'Instance Storage (GB)', 'EBS-Optimized Available', and 'Network Performance'. The 't2.micro' row is highlighted with a green background and has a green 'Free tier eligible' badge. The 't2.micro' row is also selected, indicated by a blue border around the entire row.

Figura 6.4: Selección del tipo de instancia

En el siguiente paso podemos configurar algunas opciones relativas a la máquina, las que están por defecto son adecuadas por tanto no es necesario cambiar ninguna de ellas.

The screenshot shows the AWS EC2 console interface. The top navigation bar includes 'AWS Services' and 'Edit'. Below it, a progress bar shows steps 1 through 7: '1. Choose AMI', '2. Choose Instance Type', '3. Configure Instance' (which is currently selected), '4. Add Storage', '5. Tag Instance', '6. Configure Security Group', and '7. Review'. The main content area is titled 'Step 3: Configure Instance Details'. A sub-section title 'Configure the instance to suit your requirements. You can launch multiple instances from the same AMI, request Spot instances to take advantage of the lower pricing, assign an access management role to the instance, and more.' is displayed. Configuration options include: 'Number of Instances' set to 1; 'Purchasing option' with a checkbox for 'Request Spot instances'; 'Network' dropdown set to 'vpc-7ca31618 (172.31.0.0/16) (default)'; 'Subnet' dropdown set to 'No preference (default subnet in any Availability Z)'; 'Auto-assign Public IP' dropdown set to 'Use subnet setting (Enable)'; 'IAM role' dropdown set to 'None'; and 'Shutdown behavior' dropdown set to 'Stop'.

Figura 6.5: Configurar detalles de la instancia

El siguiente diálogo nos da la opción de configurar el almacenamiento, debido a que trabajamos en una instancia de tipo *free tier* estamos restringidos a un tipo determinado de almacenamiento que al igual que el tipo de instancia, es suficiente por el momento.

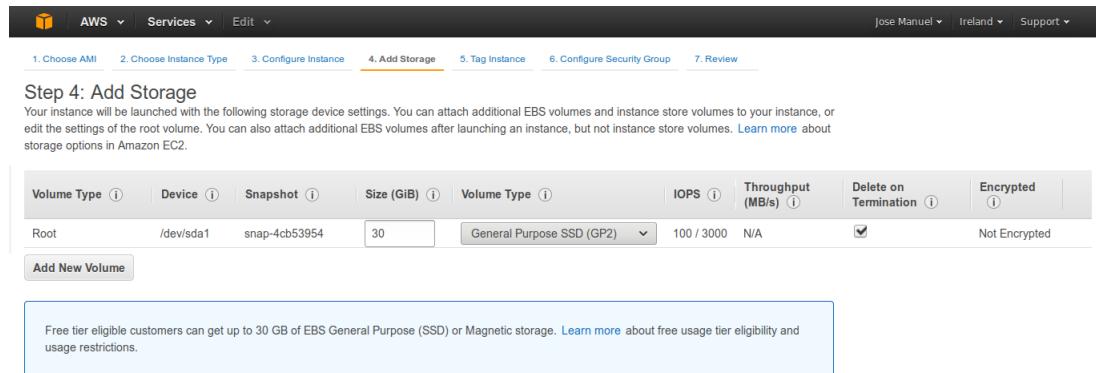


Figura 6.6: Selección de almacenamiento

Los *Security group* hacen referencia a la configuración de puertos, en la documentación de Kaa se especifica cual debe ser, para el correcto funcionamiento y tiene una estructura como la de la figura 6.7.

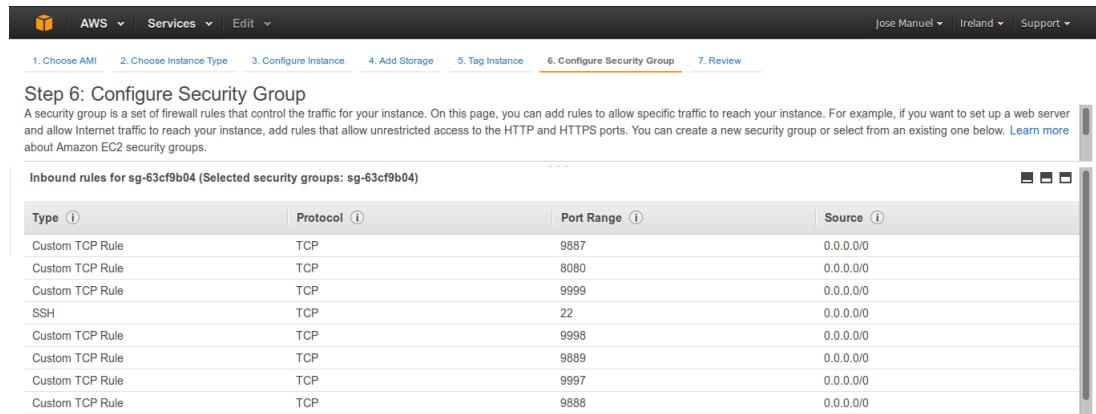


Figura 6.7: Configuración de puertos

Por último y no menos importante, debemos crear un par de llaves ó asignar uno existente. Ésto nos va a permitir conectarnos através de *SSH* a la máquina en cualquier momento, si perdemos este par de claves perderemos el acceso a la máquina.

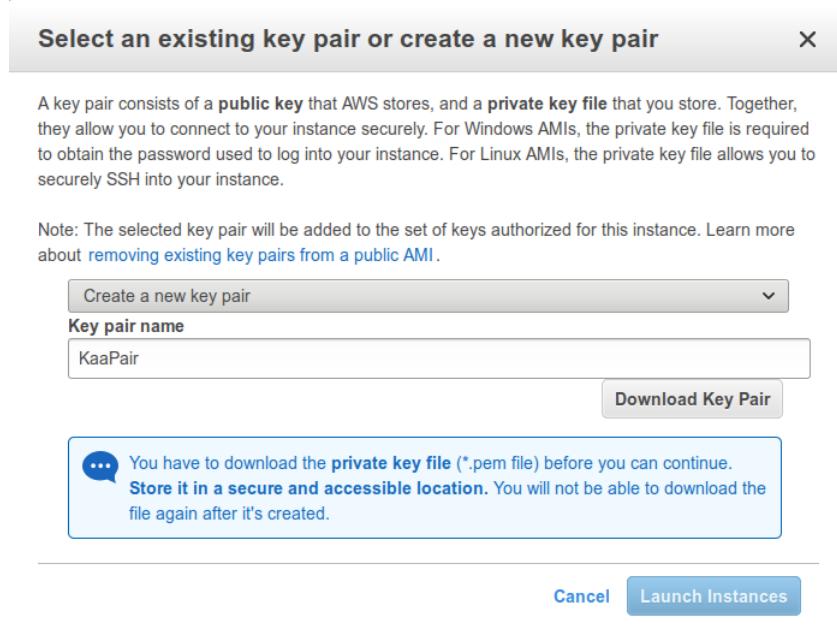


Figura 6.8: Creación de claves

6.3.2. Configuración

Una vez tenemos el servicio instalado la configuración es trivial debido a que Kaa provee un servicio web através del que definimos todo lo necesario.

Antes de empezar merece la pena destacar que existe 3 tipos de usuarios en Kaa:

- **Admin:** puede dar de alta Tenant admins.
- **Tenant admin:** puede dar de alta aplicaciones y Tenant developers.
- **Tenant developer:** puede configurar las aplicaciones y generar SDK's.

El primer paso es establecer la ip pública de Kaa, accediendo desde el menú *management*.

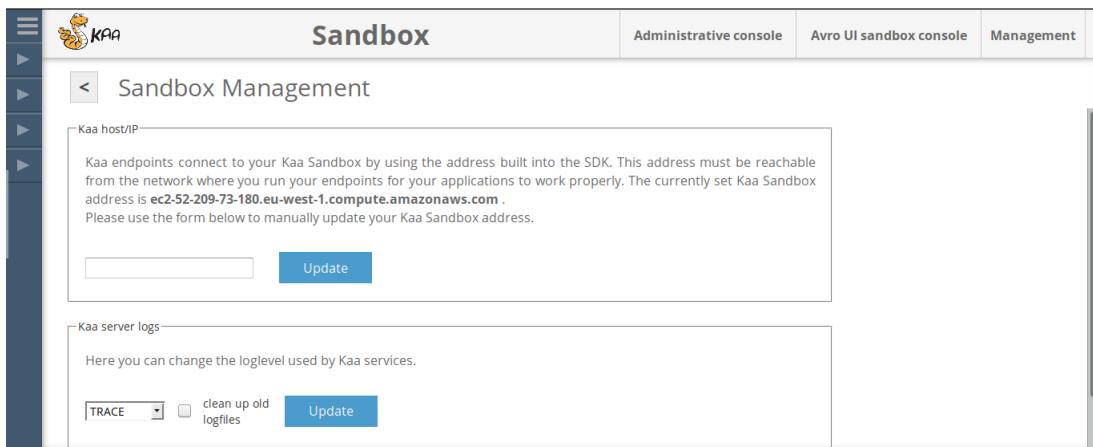


Figura 6.9: Sección Management

Ahora nos dirigimos a *Administrative console* y nos logeamos como Tenant admin para dar de alta una nueva aplicación.

El tipo de credencial seleccionado será *Trusful* de esta forma permitiremos a cualquier cliente conectarse a nuestra aplicación sin necesidad de autenticarse.

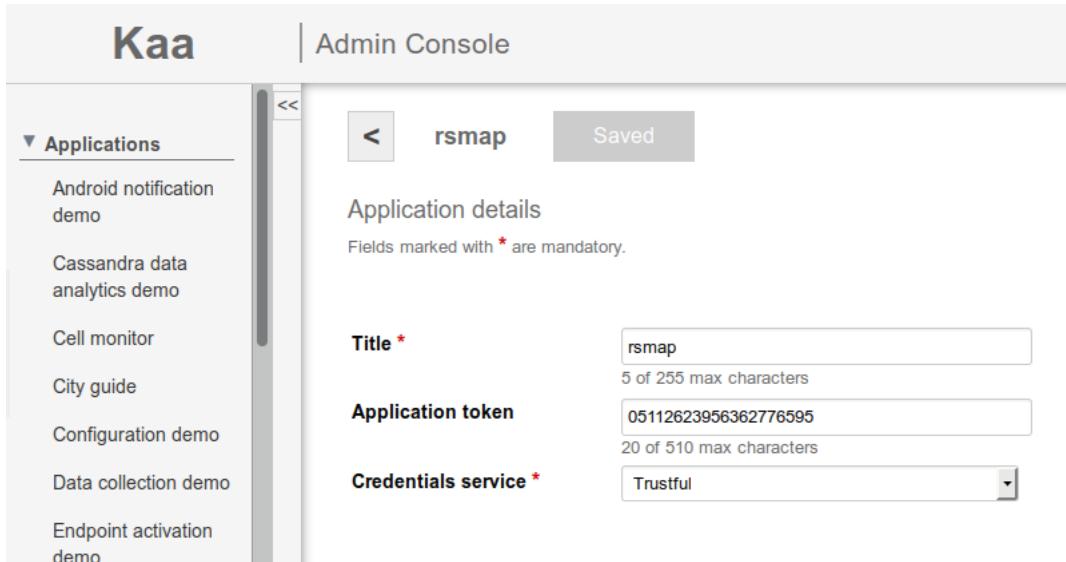


Figura 6.10: Creación de la nueva aplicación

Nos logeamos con una cuenta Tenant Developer, vamos a proceder a crear la estructura de datos que define los campos y el tipo de los mismos

que pertenecerán a nuestra aplicación.

Este paso puede hacerse desde la interfaz web o subiendo un archivo *json*. En nuestro caso vamos a valernos del *json* por comodidad.

The screenshot shows a web-based configuration interface for defining a log schema. At the top, there are buttons for 'Add log schema' (disabled), 'Add' (disabled), and 'Cancel'. Below this, the title 'Log schema details' is displayed, along with a note stating 'Fields marked with * are mandatory.' A table is present with columns: 'Field name', 'Field type', 'Is optional', and 'Delete'. A message in the table body says 'There is no data to display'. At the bottom of the table is an 'Add' button. At the very bottom of the page are three buttons: 'Upload from file', 'Browse...', and 'Upload', with the message 'No file selected.' next to the browse button.

Figura 6.11: Definiendo esquemas de datos

Aquí se definen los campos que Kaa va a proveer en el SDK generado, lo que se traduce en que éste mismo SDK nos ofrecerá una clase con los campos definidos en éste apartado. El campo *namespace* indica en qué paquete se encontrará la clase correspondiente al esquema definido dentro del SDK generado. Kaa hace uso de Apache Avro para definir esquemas de serialización de datos. Ésto le permite manejar internamente la información almacenada en los esquemas.

```
1  {
2      "type" : "record",
3      "name" : "AudioReport",
4      "namespace" : "org.kaaproject.kaa.schema.rsmap",
5      "fields" : [ {
6          "name" : "timestamp",
7          "type" : "long"
8      }, {
9          "name" : "zoneId",
10         "type" : {
11             "type" : "string",
12             "avro.java.string" : "String"
13         }
14     }, {
15         "name" : "deviceId",
16         "type" : {
17             "type" : "string",
18             "avro.java.string" : "String"
19         }
20     }, {
21         "name" : "level",
22         "type" : "double"
23     }
24 }
```

Fragmento de código 6.7: Esquema de datos en JSON

Ahora ya tenemos el esquema definido en Kaa como muestran la *figura 6.12* y la *figura 6.13*:

The screenshot shows a 'Log schema details' page with the following fields:

- Name ***: AudioReport
- Namespace ***: org.kaaproject.kaa.schema.rsmap
- Description**: None
- Fields** (Table):

Field name	Field type	Is optional
timestamp	Long	<input checked="" type="checkbox"/>
zoneld	String	<input type="checkbox"/>
deviceld	String	<input type="checkbox"/>
level	Double	<input type="checkbox"/>

Figura 6.12: Esquema de datos definido 1

The screenshot shows a 'Log schemas' list with the following data:

Version	Name	Created by	Date created	Number of EPs	Schema	Library
2	AudioLogReport	devuser	08/25/2016	0		

Figura 6.13: Esquema de datos definido 2

6.4. Instalación y configuración de Apache Cassandra.

6.4.1. Instalación

El proceso para definir la máquina virtual que contiene Cassandra es el mismo que el de Kaa a excepción de dos puntos, el primero es el tipo de instancia. Dentro de *Community AMIs* debemos buscar Cassandra y elegir la versión más reciente.

El otro punto que difiere es el *Security Group*, dado que los puertos que necesitamos son distintos la configuración debe quedar tal que así:

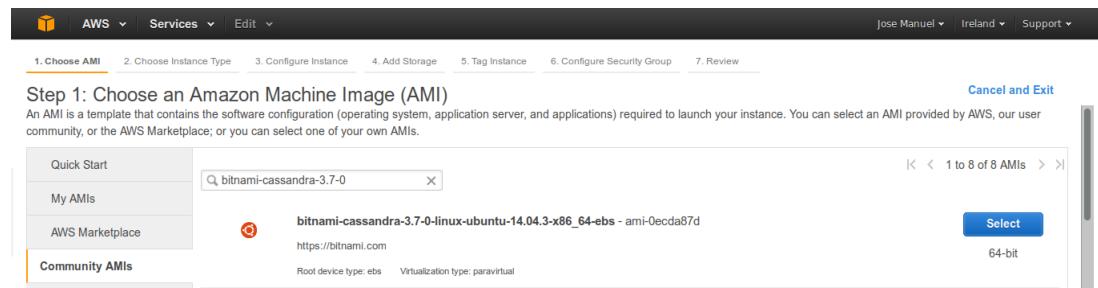


Figura 6.14: Imagen de Cassandra en EC2

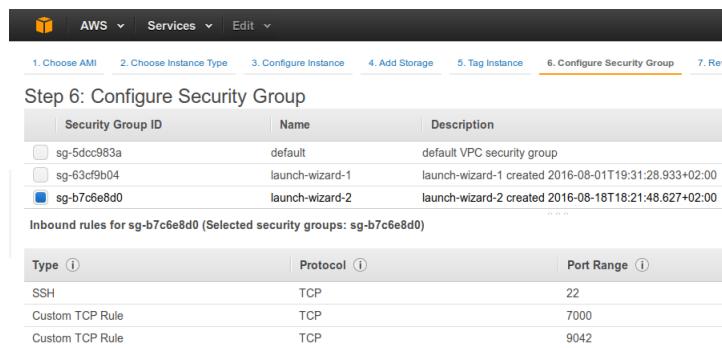


Figura 6.15: Security Group para Cassandra

6.4.2. Configuración

Para comprobar que funciona correctamente accedemos mediante *SSH* y nos logeamos en la shell de Cassandra haciendo uso del comando **cqlsh** como indica la figura 6.16.

A screenshot of a terminal window titled "bitnami@ip-172-31-31-111: ~". The window shows the command "cqlsh -u cassandra" being run, followed by a password prompt. It then displays a connection message: "Connected to Test Cluster at 127.0.0.1:9042. [cqlsh 5.0.1 | Cassandra 3.6 | CQL spec 3.4.2 | Native protocol v4] Use HELP for help." A green cursor is visible at the bottom of the terminal window.

```
bitnami@ip-172-31-31-111:~$ cqlsh -u cassandra
Password:
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.6 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
cassandra@cqlsh> 
```

Figura 6.16: Accediendo al servidor de Cassandra

Vamos a crear un *Keyspace* que contendrá las tablas de nuestra aplicación, para ello usamos la siguiente sentencia:

```
1
2 CREATE KEYSPACE rsmapv0 WITH replication = {
3     'class' : 'SimpleStrategy',
4     'replication_factor' : 1
5 };
```

Fragmento de código 6.8: Keyspace en Cassandra

SingleStrategy indica que sólo usaremos un *Datacenter* y en *replication factor* se indica el número de nodos de copia que queremos establecer.

Ahora es momento de volver a Kaa y definir el *LogAppender* que se encargará de decirle a los clientes cómo y donde tienen que enviar los datos mediante el SDK. En nuestro caso será la base de datos que acabamos de configurar. Ésto lo haremos en la sección de Log appenders mediante un fichero *json* con la siguiente estructura:

```
1 {
2     "cassandraServers": [
3         {
4             "host": "ec2-52-210-20-84.eu-west-1.compute.amazonaws.com",
5             "port": 9042
6         }
7     ],
8     "cassandraCredential": {
9         "org.kaaproject.kaa.server.appenders.cassandra.config.gen.CassandraCredential": {
10            "user": "#####",
11            "password": "#####"
12        }
13    },
14    "keySpace": "rsmapv0",
15    "tableNamePattern": "rows",
16    "columnMapping": [
17        {
18            "type": "EVENT_FIELD",
19            "value": {
20                "string": "zoneId"
21            },
22            "columnName": "zone_Id",
23            "columnType": "TEXT",
24            "partitionKey": true,
25            "clusteringKey": false
26        },
27        {
28            "type": "EVENT_FIELD",
29            "value": {
30                "string": "timestamp"
31            },
32            "columnName": "timestamp",
33            "columnType": "BIGINT",
34            "partitionKey": false,
35            "clusteringKey": true
36        },
37        {
38            "type": "EVENT_FIELD",
39            "value": {
40                "string": "deviceId"
41            },
42            "columnName": "device_Id",
43            "columnType": "TEXT",
```

```

45         "partitionKey": false ,
46         "clusteringKey": true
47     },
48     {
49         "type": "EVENT_FIELD",
50         "value": {
51             "string": "level"
52         },
53         "columnName": "level",
54         "columnType": "DOUBLE",
55         "partitionKey": false,
56         "clusteringKey": false
57     }
58 ],
59 "clusteringMapping": [
60     {
61         "columnName": "timestamp",
62         "order": "DESC"
63     }
64 ],
65 "cassandraBatchType": {
66     "org.kaaproject.kaa.server.appenders.cassandra.config.gen.CassandraBatchType": "UNLOGGED"
67 },
68 "cassandraSocketOption": null,
69 "executorThreadPoolSize": 1,
70 "callbackThreadPoolSize": 2,
71 "dataTTL": 0,
72 "cassandraWriteConsistencyLevel": {
73     "org.kaaproject.kaa.server.appenders.cassandra.config.gen.CassandraWriteConsistencyLevel": "ONE"
74 },
75 "cassandraCompression": {
76     "org.kaaproject.kaa.server.appenders.cassandra.config.gen.CassandraCompression": "NONE"
77 },
78 "cassandraExecuteRequestType": {
79     "org.kaaproject.kaa.server.appenders.cassandra.config.gen.CassandraExecuteRequestType": "SYNC"
80 },
81 "minLogSchemaVersion": 1,
82 "maxLogSchemaVersion": 2147483647,
83 "pluginTypeName": "Cassandra",
84 "pluginClassName": "org.kaaproject.kaa.server.appenders.cassandra.appenders.CassandraLogAppender",
85 "headerStructure": [
86 ],
87 ]
88

```

Fragmento de código 6.9: Esquema de log appender en JSON

Los campos a tener en consideración en este archivo son los siguientes:

- **Host:** hace referencia a la IP de Cassandra.
- **Port:** puerto de Cassandra.
- **User:** usuario de Cassandra.
- **Password:** contraseña de Cassandra.
- **keySpace:** keySpace de Cassandra.
- **tableNamePattern:** nombre de la tabla a crear.
- **columnMapping:** equivalencia entre los campos definidos en el esquema y las columnas que contendrá cada entrada en la base de datos.

Por tanto en éste fichero se especifica el usuario y la contraseña de Cassandra que han sido ocultados por seguridad, la IP pública del servidor Cassandra y campos del final indican la equivalencia entre los campos de esquema creado anteriormente y las tablas en Cassandra.

Una vez generemos el esquema, Kaa se conectará a Cassandra y creará las tablas según el formato que le hemos indicado en el mapa *columnMapping*.

Un factor importante es el diseño de las tablas, pues las consultas en Cassandra son dependientes de la estructura de la base de datos. El tipo de campos que determinan la estructura de la tabla son los siguientes:

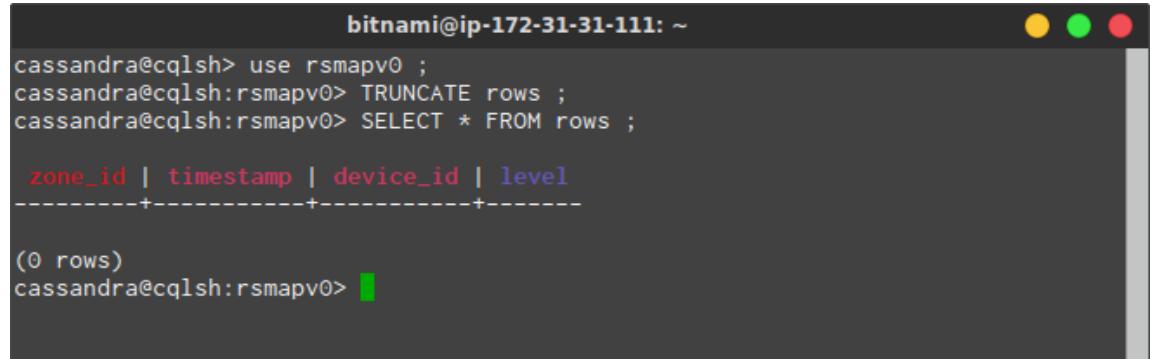
- **partitionKey**, que es la responsable de la distribución de los datos entre los nodos de Cassandra.
- **clusteringKey**, que se usa para ordenar los elementos dentro de una partición.

En el caso de RSMap se usa el campo *zoneId* como *partition key* y *timestamp* y *deviceId* como *clustering key* lo que nos garantiza que podremos hacer consultas del tipo:

```
1
2 SELECT * FROM abc WHERE partitionKey = 'X';
3 SELECT * FROM table WHERE partitionKey = 'X' AND clusteringKey
= 'Y';
```

Fragmento de código 6.10: Mécanica de consultas en CQL según la estructura de tablas

Tras guardar el *log appender* podemos comprobar como se ha creado la tabla *rows* dentro del keyspace *rsmapv0*. Para realizar ésta comprobación accedemos a la shell de Cassandra como muestra la figura 6.17 .

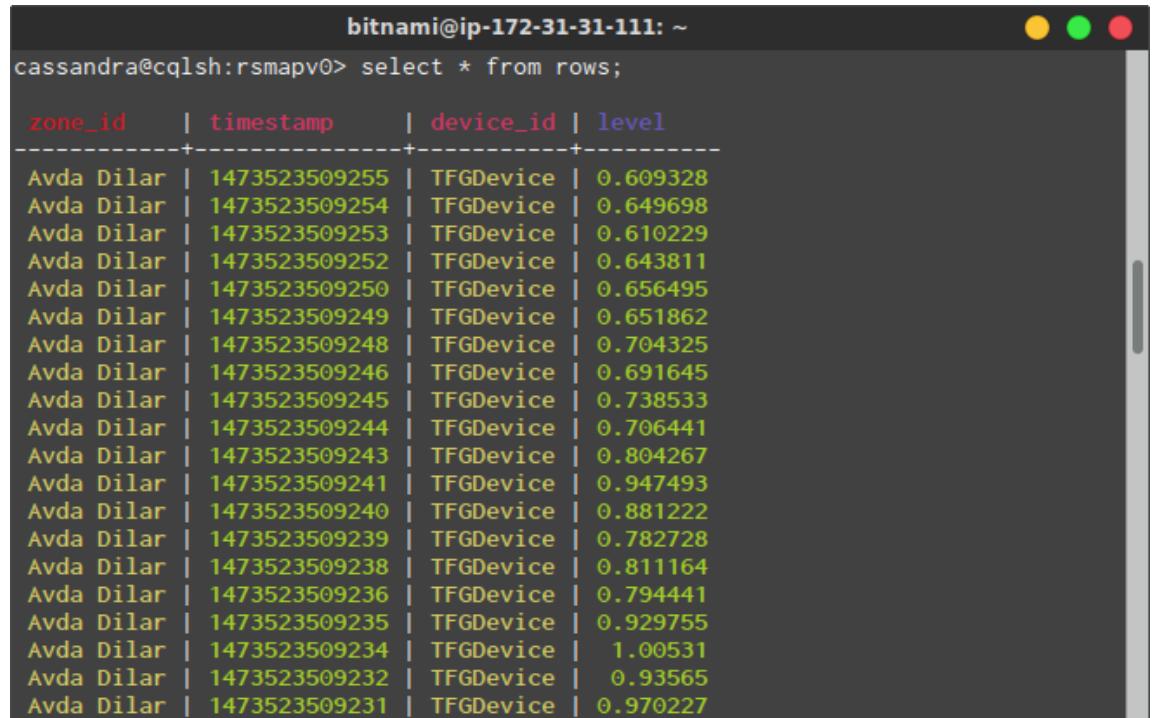


```
bitnami@ip-172-31-31-111: ~
cassandra@cqlsh> use rsmapv0 ;
cassandra@cqlsh:rsmapv0> TRUNCATE rows ;
cassandra@cqlsh:rsmapv0> SELECT * FROM rows ;

zone_id | timestamp | device_id | level
-----+-----+-----+
(0 rows)
cassandra@cqlsh:rsmapv0>
```

Figura 6.17: Comprobación de las tablas creadas

Cuando la tabla contiene datos la consulta se muestra así:



```
bitnami@ip-172-31-31-111: ~
cassandra@cqlsh:rsmapv0> select * from rows;

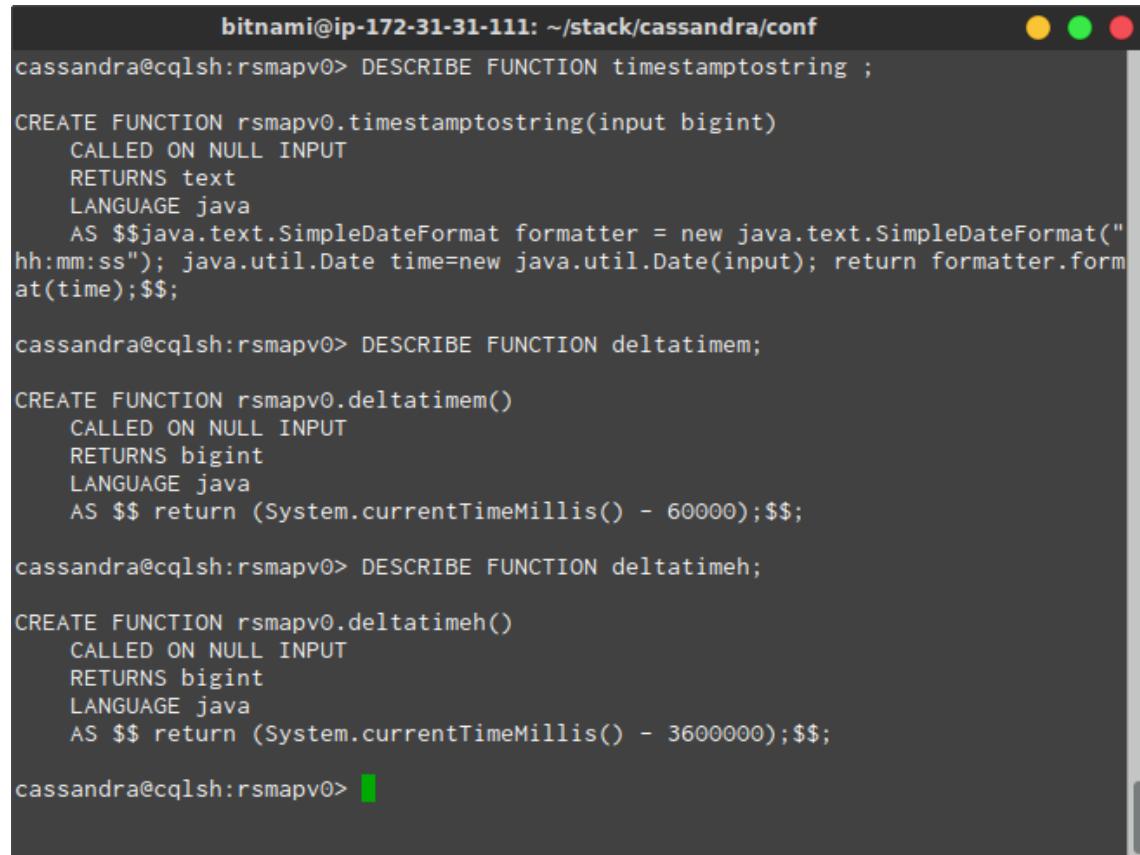
zone_id | timestamp | device_id | level
-----+-----+-----+
Avda Dilar | 1473523509255 | TFGDevice | 0.609328
Avda Dilar | 1473523509254 | TFGDevice | 0.649698
Avda Dilar | 1473523509253 | TFGDevice | 0.610229
Avda Dilar | 1473523509252 | TFGDevice | 0.643811
Avda Dilar | 1473523509250 | TFGDevice | 0.656495
Avda Dilar | 1473523509249 | TFGDevice | 0.651862
Avda Dilar | 1473523509248 | TFGDevice | 0.704325
Avda Dilar | 1473523509246 | TFGDevice | 0.691645
Avda Dilar | 1473523509245 | TFGDevice | 0.738533
Avda Dilar | 1473523509244 | TFGDevice | 0.706441
Avda Dilar | 1473523509243 | TFGDevice | 0.804267
Avda Dilar | 1473523509241 | TFGDevice | 0.947493
Avda Dilar | 1473523509240 | TFGDevice | 0.881222
Avda Dilar | 1473523509239 | TFGDevice | 0.782728
Avda Dilar | 1473523509238 | TFGDevice | 0.811164
Avda Dilar | 1473523509236 | TFGDevice | 0.794441
Avda Dilar | 1473523509235 | TFGDevice | 0.929755
Avda Dilar | 1473523509234 | TFGDevice | 1.00531
Avda Dilar | 1473523509232 | TFGDevice | 0.93565
Avda Dilar | 1473523509231 | TFGDevice | 0.970227
```

Figura 6.18: Cassandra con datos almacenados

Para finalizar vamos a crear dos funciones en java dentro del Keyspace que nos ayudarán a visualizar y consultar los datos posteriormente. La necesidad de definir estas funciones viene dada porque el campo TimeStamp se almacena bajo el formato UnixTimestamp que nos proporciona los milise-

undos actuales transcurridos desde 1970. Es importante destacar que estas funciones pueden ser definidas en el lenguaje que queramos, en nuestro caso vamos a usar Java.

La primera convierte un *timestamp* en una hora legible, las otras dos nos ayudarán a seleccionar los datos de un minuto atrás hasta la fecha actual y de una hora atrás hasta la fecha actual.



The screenshot shows a terminal window with a dark background and light-colored text. At the top, it displays the command prompt: `bitnami@ip-172-31-31-111: ~/stack/cassandra/conf`. Below the prompt, there are three colored window control buttons (yellow, green, red) on the right. The terminal content consists of several `DESCRIBE FUNCTION` commands followed by their definitions:

```
bitnami@ip-172-31-31-111: ~/stack/cassandra/conf
cassandra@cqlsh:rsmapv0> DESCRIBE FUNCTION timestampToString ;
CREATE FUNCTION rsmapv0.timestampToString(input bigint)
  CALLED ON NULL INPUT
  RETURNS text
  LANGUAGE java
  AS $$java.text.SimpleDateFormat formatter = new java.text.SimpleDateFormat("hh:mm:ss"); java.util.Date time=new java.util.Date(input); return formatter.format(time);$$;

cassandra@cqlsh:rsmapv0> DESCRIBE FUNCTION deltaTimem;
CREATE FUNCTION rsmapv0.deltaTimem()
  CALLED ON NULL INPUT
  RETURNS bigint
  LANGUAGE java
  AS $$ return (System.currentTimeMillis() - 60000);$$;

cassandra@cqlsh:rsmapv0> DESCRIBE FUNCTION deltaTimeh;
CREATE FUNCTION rsmapv0.deltaTimeh()
  CALLED ON NULL INPUT
  RETURNS bigint
  LANGUAGE java
  AS $$ return (System.currentTimeMillis() - 3600000);$$;

cassandra@cqlsh:rsmapv0>
```

Figura 6.19: Funciones dentro de Cassandra

Si queremos hacer uso de éstas funciones debemos editar el fichero **/home/bitnami/stack/cassandra/conf** y cambiar la línea:

enable user defined functions: false
por:
enable user defined functions: true

Ya sólo nos queda reiniciar el servicio:

```
1 $ sudo ./stack/ctlscript.sh stop cassandra
2 $ sudo ./stack/ctlscript.sh start cassandra
```

Fragmento de código 6.11: Reiniciando Cassandra

A modo de resumen, hasta el momento tenemos la plataforma Kaa desplegada, en ella se encuentra el esquema de datos a usar y el LogAppender que indica a donde se van a remitir los datos. Además tenemos Apache Cassandra desplegado y con las tablas necesarias creadas para empezar a enviar información de las señales detectadas. Ésto quiere decir estamos en disposición de generar el SDK que usarán los clientes para conectarse a Cassandra.

Para generar el SDK accedemos a la interfaz web de Kaa. Los SDK's se generan bajo un sistema de versiones lo que nos posibilita tener varios para una misma aplicación con distinto funcionamiento. El control de éstas versiones se gestiona mediante perfiles SDK por lo que si queremos obtener el nuestro debemos definir en primer lugar un perfil en el que también se deben especificar las versiones para los esquemas y log appenders.

Una vez nos encontramos en la interfaz buscamos la aplicación creada que se detalló en la sección de *Configuración de Kaa* y nos situamos en la opción *SDK profiles / Add SDK Profile*.

Admin Console

Add SDK profile Add Cancel

SDK profile details
Fields marked with * are mandatory.

Name * profile1
8 of 256 max characters

Configuration schema version * 1

Client-side EP profile schema version * 0

Notification schema version * 1

Log schema version * 2

► Event class families

Default user verifier

Figura 6.20: Definición del SDK

Por último pulsamos la opción *Generate SDK* la cual nos muestra una ventana en la que debemos escoger sobre qué lenguaje se generará el SDK. RSMap usa el SDK generado en Java.

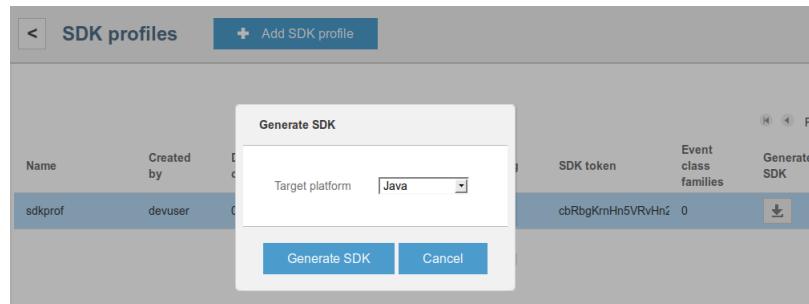


Figura 6.21: Definición del SDK

6.5. Desarrollo del módulo de detección de vehículos.

El código completo de éste módulo se encuentra en el repositorio <https://github.com/RSMMap/RSMMapPi>, dentro de la organización <https://github.com/RSMMap> que alberga el proyecto RSMMap completo.

Para analizar la estructura podemos hacer una clara división en dos partes, por un lado tenemos el módulo en python que se encarga de interactuar con la tarjeta de sonido, tomar los datos y enviarlos. La segunda parte es el módulo en java que se encarga de usar el SDK generado en Kaa para enviarlos a Cassandra.

6.5.1. Módulo de detección

Para detallarlo de una manera clara se analizarán fragmentos de código separadamente, aunque todos ellos pertenecen al mismo archivo (*RSMMapPi/analyzer/ VehicleDetection.py*) pero antes a modo de resumen se enumeran las librerías más reseñables y la secuencia que sigue el programa a lo largo de su ejecución.

Dependencias

Las dependencias externas y necesarias de éste módulo se encuentran especificadas en el archivo (*RSMMapPi/analyzer/requirements.txt*). Las más importantes son:

- **sounddevice==0.3.4:** es la más importante ya que mediante ella se accede al dispositivo para capturar los datos. Hay que destacar que los ejemplos de su documentación han sido de gran ayuda a la hora del desarrollo de éste módulo.
- **numpy==1.11.1:** es la librería por excelencia para el análisis científico de datos en Python.
- **requests==2.11.1:** nos proporciona una cómoda interfaz para efectuar peticiones HTTP que serán usadas para interactuar con la API Rest.

Además se han usado algunas otras dependencias pertenecientes a python, su instalación no es necesaria pues vienen con el compilador del lenguaje, ellas son:

- **queue:** contiene la estructura de datos de cola, sobre la que se insertaran y leerán los datos obtenidos.
- **threading:** nos permite crear hebras y procesos.
- **socket:** nos servirá para enviar los datos obtenidos desde Python al SDK de Kaa que, recordemos que está en Java.

Resumen de ejecución

En primer lugar se realiza una petición a la API para dar de alta el dispositivo en la lista de los receptores que se encuentran enviando datos. Si esa petición nos devuelve el código 400 significa que ese dispositivo ya se encuentra dado de alta, por lo que en ese caso vamos a realizar otra petición, ésta vez sobre la ruta asignada a ese dispositivo en concreto para indicar que el dispositivo se ha conectado y empezará a retransmitir datos en breve. En caso de que el dispositivo no exista, se da de alta con el payload que contiene toda la información asociada a él.

Una vez el dispositivo se identifica mediante la API, se abre un socket sobre el mismo equipo (localhost), éste servirá para enviar los datos obtenidos desde Python a Java. Los dos primeros mensajes que se envían son la cadena de localización del dispositivo y su identificador.

A continuación se definen una serie de variables que serán usadas para la detección y análisis del tráfico. Tras ésto se lanzan dos hebras, una trabajará con la función *producer* y la otra con la función *consumer*.

La hebra *producer* hace uso de la librería **Sound device** a la que llama con los parámetros definidos anteriormente, el más destacable es callback, que hace referencia a la función *callback* la cual tiene el cometido generar los bloques con los datos numéricos extraídos desde la tarjeta de sonido y ponerlos en la cola.

La hebra *consumer* es la encargada de analizar la información puesta en la cola y enviar las señales tanto al SDK (valores que se almacenarán) como a la API Rest (valores que se representarán en el mapa). Cuando el programa recibe una señal de interrupción (*enter*, '*q*' o '*Q*') el productor pondrá en la cola un objeto de llamado *_sentinel* el cual le indicará al consumidor que el programa va a finalizar. El consumidor envía las señales de parada a la API Rest, que borrará el dispositivo de la lista de dispositivos retransmitiendo así como al programa en Java que cerrará la conexión con Cassandra.

Código

Al comienzo del fichero se definen una serie de variables que serán usadas cuando el dispositivo envíe información. Gracias a ellas éste quedará identificado por un id (*device_id*), una localización en formato texto (*device_location*) y una localización geográfica (*latitude* y *longitude*).

A continuación se generan los payloads iniciales que se usarán para realizar las peticiones sobre la API, éstos tiene estructura de map {‘clave’ : valor} y tenemos dos tipos:

- **new_device_payload:**

contiene toda la información asociada al dispositivo, es usado cuando el dispositivo no está registrado en la plataforma web.

- **existing_device_payload:**

contiene únicamente el nivel iniciado a -1, ésto permitira identificarlo como nuevo dispositivo y representarlo en el mapa con un icono que indique una nueva conexión.

En la parte final de éste bloque se definen las URLs que se necesitarán para comprobar el estado del dispositivo en el sistema.

- **signals_list_url:**

referencia a la url que contiene todos los dispositivos registrados.

- **signal_url:**

referencia a la url que identifica un dispositivo en concreto.

```

1      # device id (case sensitive)
2      device_id = 'TFGDevice'
3      # device location (name)
4      device_location = 'Avda Dilar'
5      # device coordinates
6      latitude = '37.177336'
7      longitude = '-3.598557'
8      # signal_type (default unknown)
9      signal_type = 'u'
10     # level -1 means device is connected and it will send
11     #       data soon.
12     level = '-1.0'
13     # new_device_payload contains all related info with a
14     #       map structure
15     new_device_payload = {'device_id':device_id, 'lat':
16                           latitude, 'long':longitude, 'level':level, 'type':
17                           signal_type}
18     # if device exists, only level is needed
19     existing_device_payload = {'level': level}
```

```

16     # rest urls
17     signals_list_url = 'http://52.210.3.41/api/signals/'
18     signal_url = 'http://52.210.3.41/api	signal/' + device_id +
      '/',

```

Fragmento de código 6.12: Parámetros usados para la comunicación con la API

El siguiente bloque se encarga de hacer uso de las URL's y Payloads definidos anteriormente

```

1 # rest first request
2 req = requests.post(signals_list_url, new_device_payload)
3
4 # check rest response
5 if(req.status_code == 400):
6     print("Device already exists, sending connect signal")
7     req = requests.patch(signal_url, existing_device_payload)
8     if(req.status_code == 200):
9         print("Connection successful")
10    else:
11        print("Can't send connected signal, exiting")
12        sys.exit()
13 elif(req.status_code == 201):
14     print("Connection successful, device added to device's
           database")
15 else:
16     print("Device can't connect to rest service, check your
           connection.")
17 sys.exit()

```

Fragmento de código 6.13: Identificación de dispositivo mediante la API

Cuando realizamos una petición HTTP obtenemos un valor numérico que nos indica el resultado de dicha petición. En base a los códigos obtenidos podemos saber si el dispositivo se encontraba dado de alta, por lo que no se puede sobreescibir con una petición POST (*código 400*) o por el contrario si se añadió satisfactoriamente (*código 201*).

En el caso de no poder realizar la petición POST usamos la petición PATCH que básicamente es un *update* sobre el objeto en Django que contiene la información del dispositivo, es aquí donde se usa la URL definida en la variable *signal_url* que hace referencia directa al dispositivo en cuestión.

En cualquier caso, el campo *level* será establecido a -1, lo que sobre el mapa se traducirá en el ícono de conexión.

El siguiente paso es conectarse al Socket creado por *DataSender.java* que deberá estar lanzado y esperando conexiones.

```

1      # Connecting to DataSender socket on localhost
2      socket = socket.socket()
3      socket.connect(( 'localhost' , 5000))
4      # Send device location
5      device_location_sock = device_location+\n"
6      device_location_bytes = bytes(device_location_sock , 'utf
7          -8')
8      socket.send(device_location_bytes)
9      # Send device id
10     device_id_sock = device_id+\n"
11     device_id_bytes = bytes(device_id_sock , 'utf-8')
12     socket.send(device_id_bytes)

```

Fragmento de código 6.14: Conexión con Java mediante un Socket

Se le envían dos mensajes iniciales que contienen la cadena de localización y el id del dispositivo los cuales serán usados a posteriori por *DataSender.java* para añadirlos a Cassandra. No se puede pasar una cadena como tal, es por eso por lo que se convierten en Bytes antes de enviarlos.

Ahora se definen las variables necesarias para la captura e identificación de vehículos.

```

1      # multiplier factor
2      gain = 10
3      # number of cuantization levels
4      levels = 100
5      # system device id
6      device = 2
7      # block time (ms)
8      block_duration = 100
9      # sample rate
10     samplerate = 44100
11     # high sample rate
12     high = 2000
13     # low sample rate
14     low = 450
15     # cuantization value
16     delta_f = (high - low) / levels
17     # window will divided in bands, fftsize defines the resolution
18         on freq domain
19     fftsize = np.ceil(samplerate / delta_f).astype(int)
20     # window freq resolution
21     low_bin = np.floor(low / delta_f)
22     # vehicle threshold
23     threshold = 0.59
24     # consequutive blocks, its may depend of the road conditions
25     consequutive_blocks = 50

```

Fragmento de código 6.15: Variables correspondientes al análisis

Las variables `threshold` y `consecutive_blocks` tienen relación directa con la identificación de vehículos. La primera indica el umbral mínimo que deben tener los valores obtenidos por cada bloque obtenido mientras que la segunda indica qué número de bloques necesitamos por encima de ese umbral para considerar que un vehículo ha pasado.

El resto poseen la misma estructura que las del ejemplo que proporciona la librería `SoundDevice` para cuantizar los valores obtenidos a través de una entrada de audio. (<http://python-sounddevice.readthedocs.io/en/0.3.4/examples.html#real-time-text-mode-spectrogram>)

La esencia de la hebra productora es la llamada a `InputStream` que abre un canal de comunicación con la tarjeta de sonido. Toma como argumentos los valores definidos anteriormente y una función llamada `callback`, que es la encargada de generar cada bloque y ponerlo en la cola. Cada bloque consta de una serie de valores que tras aplicarle la Transformada Rápida de Fourier son sumados obteniendo así el valor global para cada bloque.

```

1 with sd.InputStream(
2     device=device,
3     channels=1,
4     callback=callback,
5     blocksize=int(samplerate * block_duration / 1000),
6     samplerate=samplerate
7 )

```

Fragmento de código 6.16: Hebra productora

El código perteneciente a la hebra consumidora consiste en un bloque que se ejecuta mientras no obtenga una señal de parada, en tal caso se envía una señal de desconexión a la API usando el método `DELETE` que eliminará el modelo asociado al dispositivo en Django y otra a `DataSender.java`. El código encargado de filtrar los datos es el siguiente:

```

1 if(data > threshold):
2     # global block consecutive count
3     consecutive += 1
4     local_consecutive += 1
5     local_data_sum += data
6     if(local_consecutive == consecutive_blocks):
7         # if detected > consecutive_blocks send to api
8         rest
9         existing_device_payload = {
10             'level': str(local_data_sum)
11         }

```

```

11         req = requests.patch(signal_url ,
12                           existing_device_payload)
13
14         print(
15             str(consecutive_blocks) +
16             " consecutive blocks, sending to rest
17             API "
18             + str(local_data_sum)
19             )
20
21         local_consecutive = 0
22         local_data_sum = 0.0
23
24     # add representative values to send_queue
25     send_queue.put(data)
26
27 else :
28     if(consecutive > consecutive_blocks):
29         print("Sending data to cassandra")
30         while not send_queue.empty():
31             # detected case, sending items to Kaa
32             # SDK via TCP socket
33             item_to_send = send_queue.get()
34             linestr =str(item_to_send)+"\n"
35             linebytes = bytes(linestr , 'utf-8')
36             socket.send(linebytes)
37
38     # consecutive was not bigger than consecutive_blocks ,
39     # cleaning resources
40     send_queue = Queue()
41     send_queue.queue.clear()
42     consecutive = 0
43     local_consecutive = 0
44     local_data_sum = 0.0

```

Fragmento de código 6.17: Hebra consumidora

Si el dato obtenido de la cola supera el umbral definido incrementaremos la variable que indica los bloques consecutivos válidos hasta el momento, además añadiremos el valor a una suma parcial. En cuanto tengamos un número de bloques válido efectuaremos una petición PATCH a la API que actualizará el valor para el dispositivo con la suma total de los bloques obtenidos. Por último éste valor se sitúa en otra cola que alimentará a la aplicación *DataSender.java*.

El motivo de enviar ésta señal cuando detectamos los bloques necesarios es para garantizar que la aplicación trabaja en tiempo real sobre el mapa, los datos que se envían a *DataSender.java* pueden sufrir pequeños retrasos debio a que éstos se envían cuando se obtiene un valor que no supera el umbral. Tras enviarlos todos los recursos que ocupan son liberados para proceder a la detección de un nuevo vehículo.

Las funciones *producer* y *consumer* son llamadas cuando se inicializan los Threads correspondientes a cada una de ellas.

```
1     queue = Queue()
2
3     # thread instances
4     thread_prod = Thread(target=producer, args=(queue, ))
5     thread_cons = Thread(target=consumer, args=(queue, ))
6
7     # thread init
8     thread_prod.start()
9     thread_cons.start()
```

Fragmento de código 6.18: Inicialización de las hebras

6.5.2. Módulo de envío

Este módulo se encargará de inicializar el Socket necesario para que *VehicleAnalyzer.py* le proporcione los datos que posteriormente enviará a Cassandra. La estructura es más simple que la del módulo anterior en gran medida a que el SDK que proporciona Kaa tiene una interfaz muy cómoda para trabajar con ella.

Dependencias

Las dependencias necesarias son el SDK generado por Kaa en formato *.JAR* así como otras dependencias ajenas pertenecientes a Kaa. Todas ellas se encuentran bajo el directorio *sender* y se pueden identificar con los siguientes nombres:

- kaa-java-ep-sdk.jar
- log4j-over-slf4j-1.7.7
- logback-classic-1.1.2
- logback-core-1.1.2

Resumen de ejecución

La primera tarea inicia el cliente Kaa *kaaClient* sobre el cual se enviarán los datos.

Lo siguiente es inicializar el Socket y esperar la llegada de mensajes, los dos primeros indicarán localización e id como se indicó anteriormente.

Mientras no se detecte una señal de parada (*carácter 'q'*) el Socket estará aceptando conexiones y leyendo los datos de cada una.

Para cada conexión crea un objeto de la clase *AudioReport*, éste nombre y sus atributos vienen definidos por los esquemas que definimos anteriormente sobre la plataforma Kaa. Tras asociarle los valores pertinentes el objeto es enviado gracias a la función *addLogRecord(report)*; que añadirá éste objeto en forma de entrada a Cassandra.

Código

En el siguiente fragmento de código se detalla como se abre el Socket y se leen las dos primeras entradas asociandolas a las variables pertinentes

```

1      ServerSocket Server = new ServerSocket(5000);
2
3      System.out.println("Waint for VehicleAnalyzer on TCP
4          socket at port 5000");
5
6      // Listening connections
7      Socket connected = Server.accept();
8      System.out.println("VehicleAnalyzer with " + connected.
9          getInetAddress() + ":" + connected.getPort() + " is
10         connected!");
11
12     // First two packets comes with device location and
13     // device id
14     String location = inFromClient.readLine();
15     String device = inFromClient.readLine();

```

Fragmento de código 6.19: Inicialización del Socket

Para finalizar se muestra la creación y envío del objeto que se traducirá en una fila dentro de nuestra base de datos.

```

1      // create new AudioReport object which is appended to
2      // Cassandra
3      AudioReport report = new AudioReport();
4      long timestamp = System.currentTimeMillis();
5      report.setZoneId(location);
6      report.setDeviceId(device);
7      report.setLevel(Double.parseDouble(fromclient));
8      report.setTimestamp(timestamp);
9
10     // send Audio Report object
11     kaaClient.addLogRecord(report);

```

Fragmento de código 6.20: Creación y envío de datos a Cassandra

6.6. Desarrollo de la plataforma web.

La estructura de la plataforma web se detalló en el capítulo de diseño, por tanto vamos a proceder a analizar los archivos más relevantes que la componen. Al igual que el módulo de detección podemos hacer una distinción clara entre las dos partes que la componen, la web de RSMap y la API Rest.

6.6.1. Plataforma web

Dependencias

Se encuentran especificadas en el archivo (*RSMaprequirements.txt*) dentro del repositorio <https://github.com/RSMap/RSMap>

- **Django==1.10.1**

Código

Los archivos de Django que disponen el comportamiento de la aplicación (parte lógica) son **models.py** en el que se define la estructura de datos que será almacenada, tiene un formato como el que se muestra a continuación:

```

1 class Signal(models.Model):
2     device_id = models.TextField(primary_key=True)
3     lat = models.DecimalField(max_digits=9, decimal_places=6)
4     long = models.DecimalField(max_digits=9, decimal_places=6)
5     created = models.DateTimeField(auto_now=True, blank=True)
6     level = models.FloatField()
7
8     type = models.CharField(
9         max_length=1,
10        choices=VEHICLE_TYPE,
11        default=UNKNOWN,
12    )
13
14     def __str__(self):
15         return self.device_id

```

Fragmento de código 6.21: Modelo definidos en Django

Un campo a tener en consideración es el llamado *created*, en concreto el argumento **auto_now=True** que nos va a permitir que cada vez que un objeto almacenado del tipo *Signal* sea creado o actualizado, el valor de éste campo será actualizado con el *timestamp* relativo a la fecha de dicho cambio. Así podremos tener constancia de cual fué el último momento en el se actualizó, por ejemplo, el campo *level*.

El siguiente fragmento de código pertenece al archivo *urls.py* en el que se definirán las rutas que va a proporcionar RSMap en su web:

```
1 urlpatterns = format_suffix_patterns([
2     url(r'^api/signals/$', views.SignalList.as_view(), name='
3         signal-list'),
4     url(r'^api	signal/(?P<pk>[a-zA-Z0-9]+)/$', views.SignalDetail.
5         as_view(), name='signal-detail'),
6     url(r'^$', TemplateView.as_view(template_name='index.html')),
7     url(r'^map/$', TemplateView.as_view(template_name='map.html'))
8 ])
9 
```

Fragmento de código 6.22: Definición de URL's

Éstas URL's se definen mediante expresiones regulares, si en el navegador se introduce una URL que entra del patrón de una de éstas expresiones, se llamará a la vista correspondiente para esa URL.

Tanto las URL's pertenecientes a la web, como las pertenecientes a la API Rest son definidas en éste mismo fichero.

Los directorios *static* y *templates* son los encargados de crear la parte visual de la web de RSMap.

En *templates* podemos encontrar las plantillas que sirve nuestra aplicación, ambas usan HTML5 y Bootstrap:

- **index.html:** contiene la página principal.
- **map.html:** contiene la página con el mapa de tráfico.

Dentro del fichero *map.html* se incluye el archivo *static/js/rsmapMapUpdater.js* que se encargará de actualizar dinámicamente el contenido del mapa.

La función *checkDevices* se ejecuta cada 30 segundos y permite a un visitante saber si existen dispositivos retransmitiendo en ese momento, en caso de que no se le mostrará una indicación de ello en la esquina superior derecha.

```

1 function checkDevices () {
2     \$(function () {
3         \$.getJSON( 'http://52.210.3.41/api/signals.json' , function (
4             data) {
5                 if(data.length == 0){
6                     bootstrap_alert.warning( '<strong>INFO:</strong> No' ,
7                         'devices sending data right now' , 'warning' , 4000)
8                 ;
9             }
10            });
11        });
12    });
13 }

```

Fragmento de código 6.23: Función checkDevices

En el siguiente fragmento se muestra como se actualiza el mapa. El proceso consta de dos pasos.

Cada segundo se comprueba la ruta `http://52.210.3.41/api/signals.json` que devuelve la lista de dispositivos conectados a RSMap. Una vez se ha obtenido se recorre esa lista y se comprueban los tiempos en los que fueron añadidos, si éstos tiempos pertenecen al intervalo (*tiempo actual - 2 segundos*) $\geq timestampSeñal \geq (tiempo actual)$ se procede a representarla en el mapa.

```

1 // map update via AJAX
2 \$(document).ready(
3     function worker(){
4         \$.ajax(
5             {
6                 // retrieve updated json with last valid signals
7                 url: "http://52.210.3.41/api/signals.json" ,
8
9                 complete: function(){
10                     // next call will be in 1 second
11                     setTimeout(worker, 1000);
12                 }
13             }
14         ).then(
15             function(data)
16             {
17                 //clean map
18                 deleteMarkers();
19
20                 // run all json markers and add them to map
21                 for(var i = 0; i < data.length; i++){
22                     var signal = data[i];
23                     //console.log(signal);
24
25                     signal_timestamp = parseFloat(signal[ 'created' ]);
26                     // convert now to seconds and 2 seconds less
27                     // (python timestamp comes in seconds)

```

```

28         // that's the reason to divide by 1k
29         var now = ((new Date().getTime() - 2000) / 1000|0) ;
30
31         // update 'last update' field
32         $('.last-update').empty();
33         var now_date = new Date();
34         $('.last-update').append(now_date);
35
36         // now was reduced 2 seconds so if signal_timestamp is
37         // greater than
38         // now means it was in a interval between now and 2
39         // secs later
40         if(signal_timestamp > now){
41             var location = new google.maps.LatLng(parseFloat(
42                 signal['lat']), parseFloat(signal['long']));
43             var level = signal['level'];
44             addMarker(location, level);
45         }
46     }
47 );

```

Fragmento de código 6.24: Actualización dinámica del mapa

Para cada señal que se va a representar se tiene su valor *level*. Según el tamaño de éste valor se representará la señal en el mapa con distintos iconos.



Figura 6.22: Icono para valor -1: conexión de dispositivo



Figura 6.23: Icono para valor >55.5: vehículo pesado



Figura 6.24: Icono para valor 45.5 <x <55.5: vehículo medio



Figura 6.25: Icono para valor $x < 45.5$: vehículo ligero

En un principio se pensó en efectuar ésta comprobación en la parte lógica de la aplicación pero como los datos se obtienen de todas formas en el cliente, se aprovecha ésta situación para realizar la comprobación en el navegador de éste modo se libera carga del servidor web.

Aspecto Final

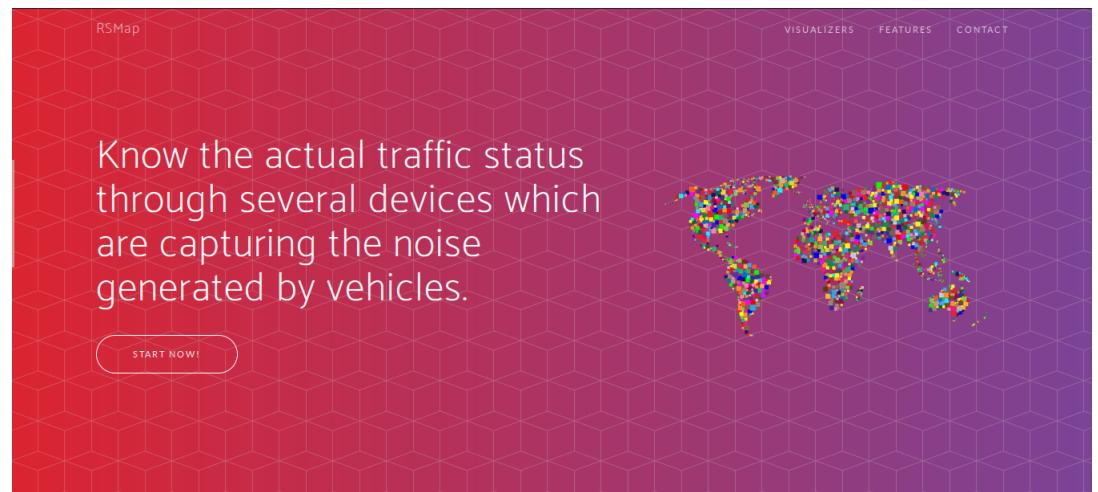


Figura 6.26: Home de la web

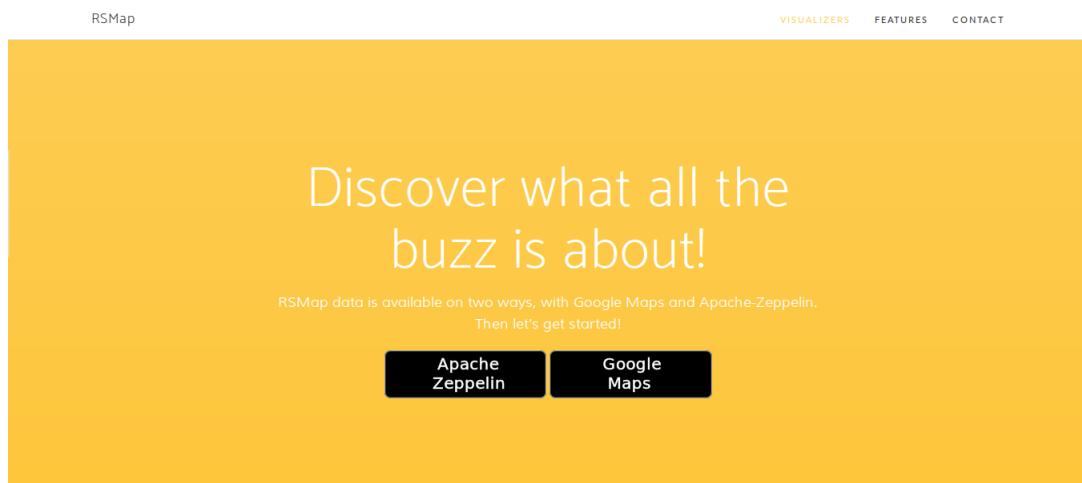


Figura 6.27: Sección de visualizadores

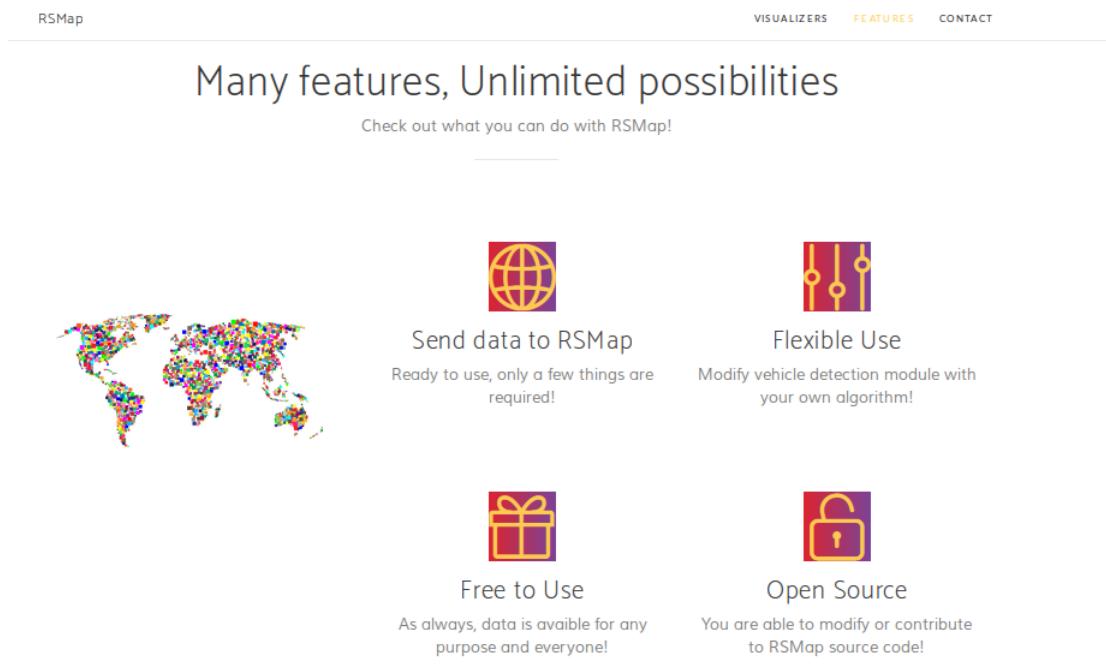


Figura 6.28: Sección de características

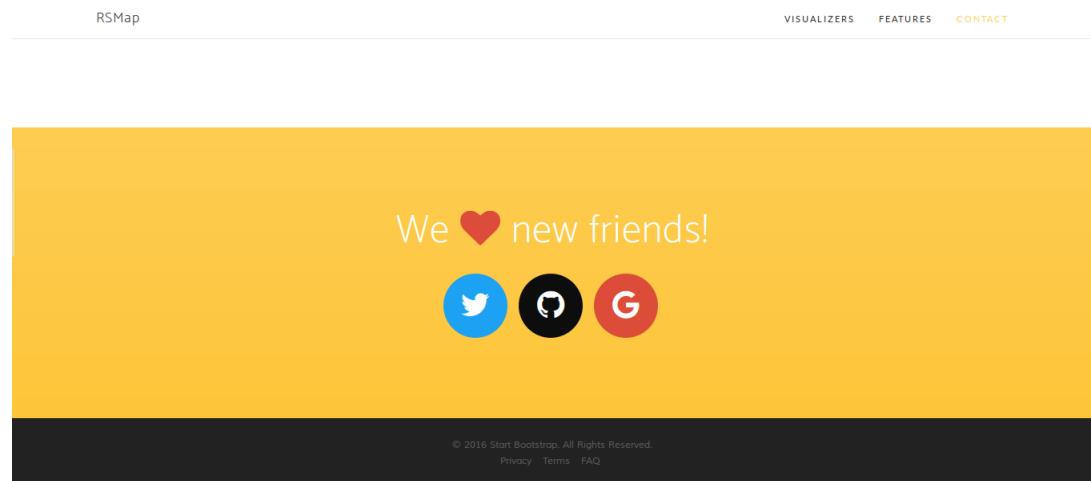


Figura 6.29: Sección de contacto



Figura 6.30: Sección del mapa

6.6.2. API Rest

Dependencias

Dentro del mismo repositorio, en el mismo archivo *requirements.txt* se encuentra la linea necesaria para la API.

- **djangorestframework==3.4.6**

Código

Tanto el archivo *models.py* como el *urls.py* son usados por la API, que forma parte del ecosistema que genera Django, por tanto vamos a definir los archivos que componen la API.

El primer archivo se llama *serializers.py* y en él se detalla cómo se va a usar el modelo *Signal* en la API, es decir, cuando una petición solicite un objeto de tipo *Signal* qué campos se devolverán en el JSON.

```
1 class SignalSerializer(serializers.ModelSerializer):
2     created = serializers.DateTimeField(format="%s", required=False)
3
4     class Meta:
5         model = Signal
6         fields = ('device_id', 'lat', 'long', 'created', 'level', 'type',
7                   ,)
```

Fragmento de código 6.25: Serializador del modelo Signal

Para finalizar el archivo *views.py* definimos las vistas que tendrá nuestra API, dos como se ha mencionado anteriormente. La primera de ellas devuelve un JSON con todos los dispositivos conectados a RSMap y la segunda los campos para un dispositivo en concreto.

Éstas vistas son llamadas desde las urls que servirán la API y que fueron descritas anteriormente cuando se expuso el archivo *urls.py*.

```
1 class SignalList(generics.ListCreateAPIView):
2     queryset = Signal.objects.all()
3     serializer_class = SignalSerializer
4
5 class SignalDetail(generics.RetrieveUpdateDestroyAPIView):
6     queryset = Signal.objects.all()
7     serializer_class = SignalSerializer
```

Fragmento de código 6.26: Vistas de la API

6.7. Instalación y configuración de Apache Zeppelin.

6.7.1. Instalación

Es hora de instalar Apache-Zeppelin. Como en los servicios anteriores, lo primero que debemos hacer es configurar e iniciar una nueva instancia en Amazon EC2. Los pasos a seguir son los mismos con la salvedad de la selección de la imagen y la configuración de *Security groups*.

El SO usado ésta vez será Ubuntu 14.04.4 LTS, para seleccionarlo lo podemos buscar dentro de las imágenes disponibles al configurar la nueva instancia.

Como tráfico entrante abriremos el puerto 8080 que es donde se va a ejecutar Apache-Zeppelin.

Una vez que tengamos la máquina disponible accedemos por SSH y procedemos a instalar Apache-Zeppelin con las siguientes órdenes:

```

1 $ cd /opt
2 $ sudo wget http://apache.rediris.es/zeppelin/zeppelin-0.6.1/
     zeppelin-0.6.1.tgz
3 $ sudo tar -zxf zeppelin-0.6.1.tgz
4 $ sudo /opt/zeppelin-0.6.1-bin-all/zeppelin-daemon.sh

```

Fragmento de código 6.27: Instalación de Apache-Zeppelin

Debemos asegurarnos de descargar el binario que trae los intérpretes instalados, entre ellos el de Cassandra para poder hacer llamadas a la base de datos en los Notebooks.

Si ahora abrimos el navegador a nuestra dirección pública deberíamos encontrarnos con lo siguiente:

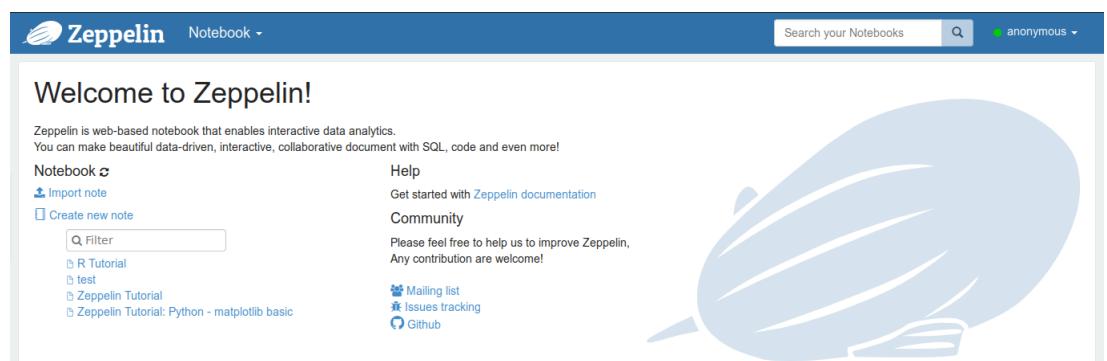


Figura 6.31: Apache Zeppelin en funcionamiento

6.7.2. Configuración

Vamos a configurar es el intérprete de Cassandra, para ello nos dirigimos a la parte derecha de la interfaz y entramos en la opción *interpreters*. Tras localizar el intérprete de Cassandra pulsamos el botón *editar* y configuraremos los siguientes campos:

- **cassandra.credentials.password**, aquí indicamos la contraseña de Cassandra.
- **cassandra.credentials.username**, aquí el usuario.
- **cassandra.hosts**, aquí la dirección pública del servidor de Cassandra.
- **cassandra.keyspace**, y aquí el Keyspace que definimos anteriormente en Cassandra.

para finalizar guardamos los cambios efectuados.

Ahora estamos preparados para crear un nuevo Notebook para RSMap mediante el menú desplegable *Notebooks* y la opción *create new notebook* al cual deberemos establecer un nombre. Una vez hecho ésto entraremos en el Notebook que tiene una estructura como muestra la *figura 6.20*.

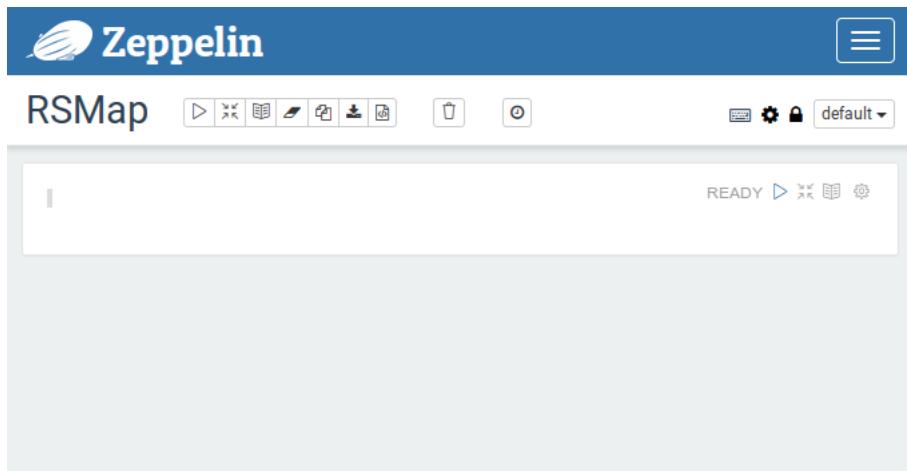


Figura 6.32: Estructura de un Notebook en Zeppelin

Sólo queda una tarea por realizar, que es configurar cuales de los intérpretes estarán disponibles para usar bajo este Notebook. Existen muchos como el de *Scala*, *Python*, *Java* o *Markdown* entre otros.

Para configurarlos nos dirigimos a la rueda dentada situada en la parte derecha y desactivamos todos los intérpretes a excepción de el de Cassandra. La configuración debe quedar tal que así:

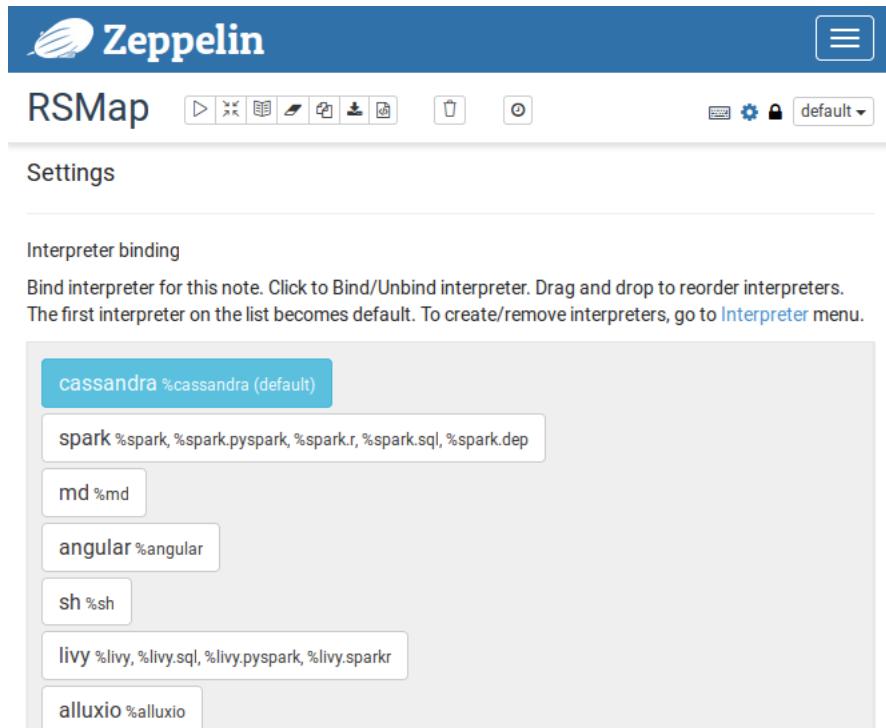


Figura 6.33: Cassandra como intérprete para Zeppelin

Ya tenemos todos los elementos preparados para realizar consultas en *CQL* a nuestro servidor Cassandra y poder visualizar los datos. La interfaz de Zeppelin está compuesta de varias celdas en las cuales podemos introducir el código deseado.

En nuestro caso estas sentencias pertenecen al intérprete de Cassandra luego la primera línea de cada celda debe contener la palabra clave **%cassandra**. Una vez tengamos escrita la consulta podemos pulsar el botón de *Play* y esperar unos segundos para obtener los resultados.

Vale la pena destacar que Zeppelin posee un sistema de programación de tareas al que podemos acceder pulsando el pequeño ícono del reloj llamado *Run scheduler* con una estructura similar a crontab, ésto nos permite programar un tiempo predeterminado para que se ejecuten las celdas del Notebook de manera automática y poder tener los resultados actualizados en todo momento.

6.7.3. Ejemplo de consulta CQL

Detallamos un ejemplo de consulta a modo de ilustración de como funciona CQL, veamos que forma tiene y posteriormente se comentarán las

diferentes partes de la misma.

```
1 %cassandra
2 SELECT rsmapv0.timestampToString(timestamp) AS time, level from
rsmapv0.rows WHERE zone_id={{zone_id='Granada'}} and
timestamp > rsmapv0.deltatimeH() ORDER BY timestamp ASC;
```

Fragmento de código 6.28: Ejemplo de consulta CQL

Como se vió en la parte de configuración de Cassandra habíamos creado ciertas funciones para usarlas a la hora de consultar los datos, es el momento de verlas en funcionamiento. *rsmapv0.timestampToString* es la función que nos formatea un *timestamp* a una fecha, rsmapv0 indica en qué Keyspace se encuentra dicha función.

En la cláusula WHERE tenemos **zone id = {{zone id='Granada'}}** lo que nos permitirá modificar el valor de ese campo mediante la interfaz de Zeppelin si necesidad de reescribir la consulta.

En último lugar **rsmapv0.deltatimeH()** hace referencia a una de las otras funciones definidas en Cassandra, en este caso se indica en la consulta que el campo *timestamp* debe ser mayor que **rsmapv0.deltaTimeH**. Ésta función nos devuelve la hora actual restándole un minuto, por tanto obtendremos todos los valores de un minuto antes de la fecha actual.

Tras poner el sistema en funcionamiento durante un minuto en el que pasan algunos vehículos podemos ver como Zeppelin representa los datos mediante la consulta mencionada:

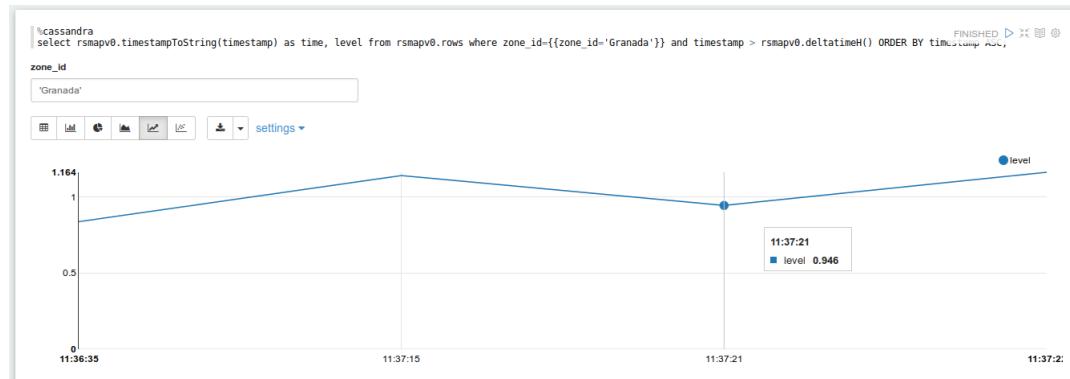


Figura 6.34: Muestra de resultados en Zeppelin

Capítulo 7

Pruebas

Las pruebas realizadas en RSMap han supuesto cambios mínimos sobre los que se ha ido mejorando lo anterior por lo que no se tiene una perspectiva completa de todos las pruebas realizadas ya que serían cientos, sin embargo el conjunto de esos pequeños cambios se puede agrupar en problemas a los cuales se le ha encontrado una solución en la mayoría de los casos.

Se ha monitorizado la carga que sufren los servidores cuando se encuentra sometidos a trabajo. Debido a que Amazon EC2 proporciona un sistema autoescalable, éste aspecto no es de gran preocupación. No obstante los resultados obtenidos indican que la aplicación RSMap en su conjunto no supone un gran coste de recursos como podemos ver en las siguientes imágenes:

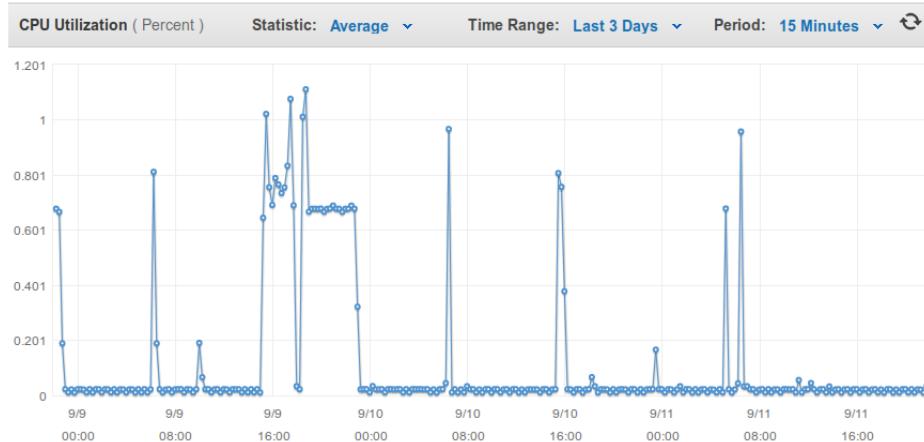


Figura 7.1: Carga CPU del servidor web y API

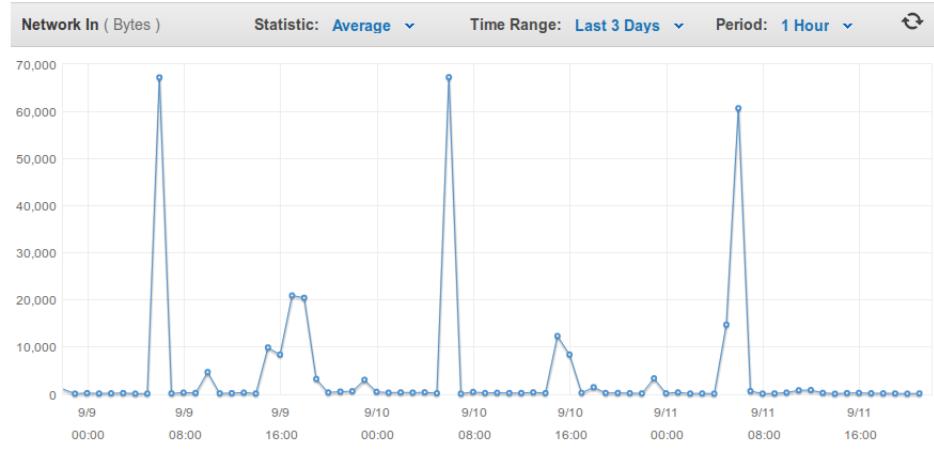


Figura 7.2: Carga de entrada de red del servidor web y API

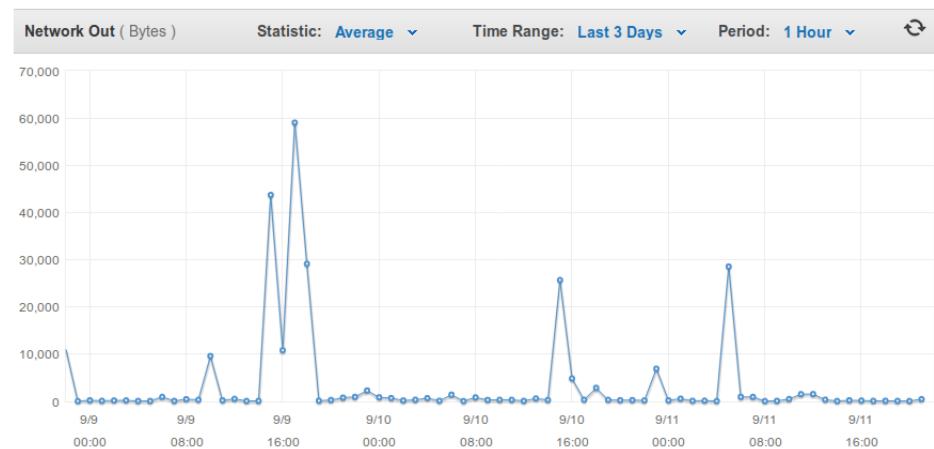


Figura 7.3: Carga de salida de red del servidor web y API

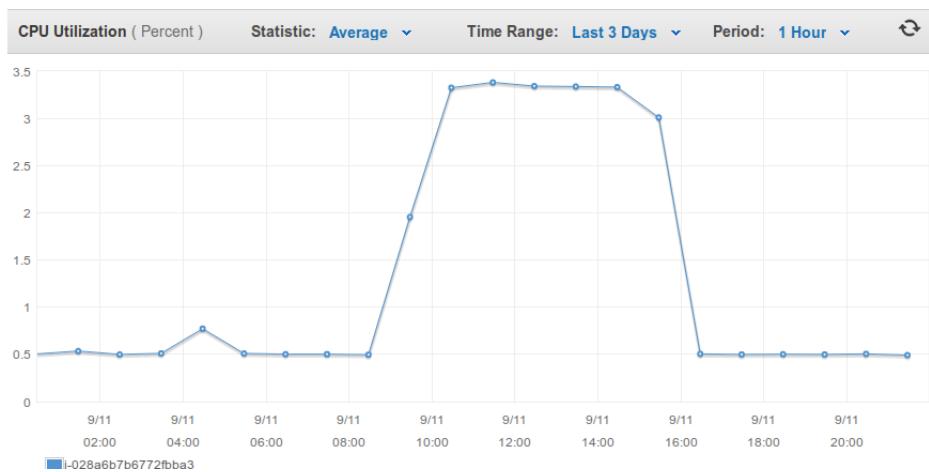


Figura 7.4: Carga CPU del servidor de Cassandra

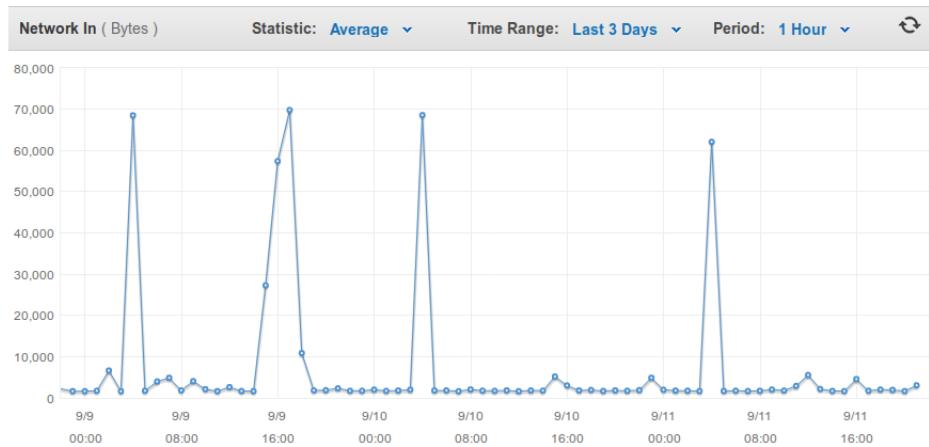


Figura 7.5: Carga de entrada de red del servidor de Cassandra

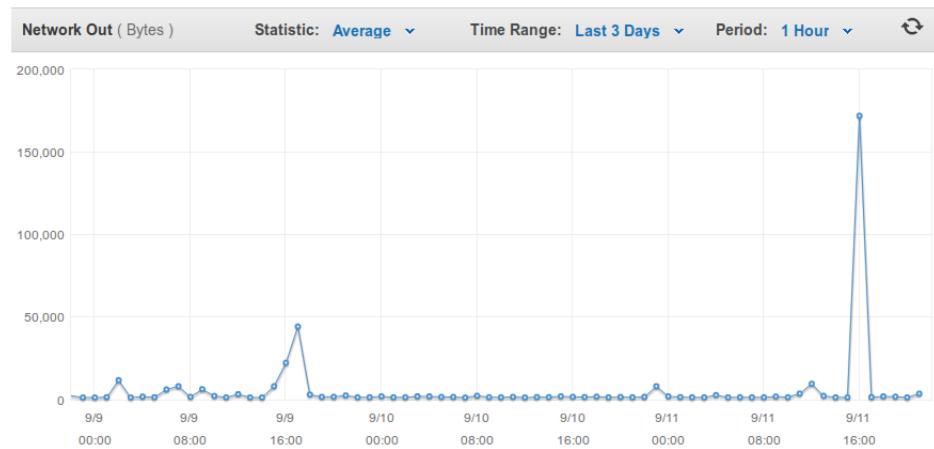


Figura 7.6: Carga de salida de red del servidor de Cassandra

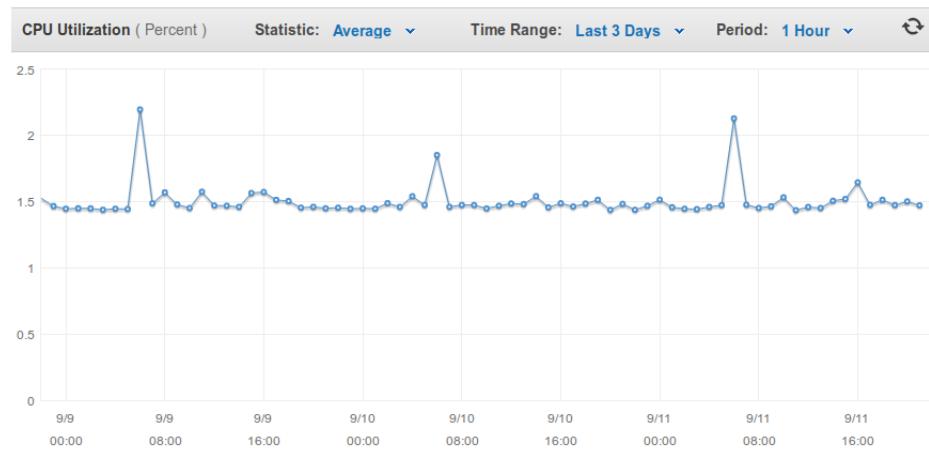


Figura 7.7: Carga CPU del servidor de Zeppelin

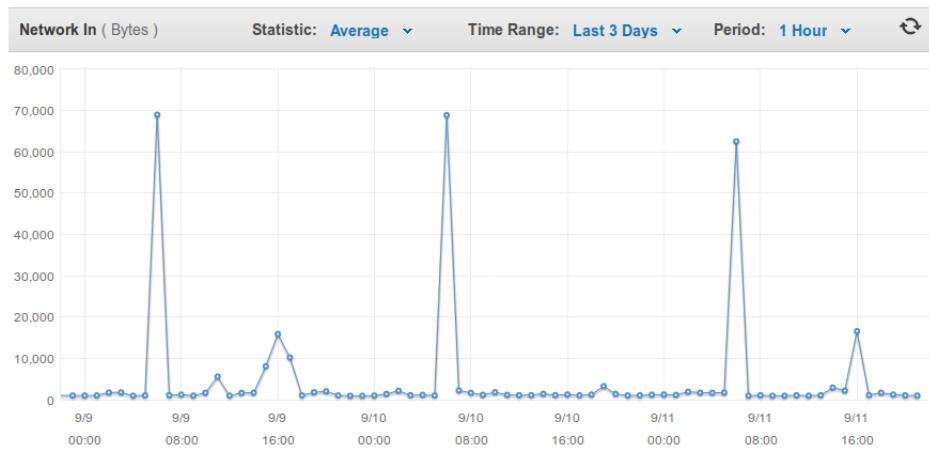


Figura 7.8: Carga de entrada de red del servidor de Zeppelin

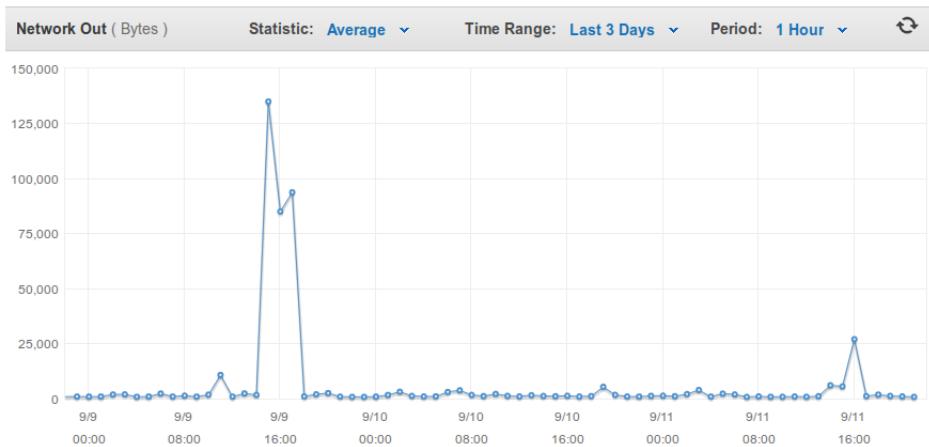


Figura 7.9: Carga de salida de red del servidor de Zeppelin

El dispositivo receptor Raspberry Pi ha demostrado haber sido una elección correcta ya que en ningún momento se ha presentado algún problema relacionado con el rendimiento, es más, los recursos usados cuando la aplicación está funcionando no superan un 30 % del total por lo que éste dispositivo tiene cabida para actividades computacionalmente mucho más complejas de las que se han llevado a cabo finalmente.

La siguiente tabla muestra las fases críticas en el desarrollo, configuración y construcción del proyecto y los errores y soluciones que se han propuesto para combatirlos.

Nombre	Descripción	Resultado	Solución
P1	Conexión con la tarjeta de sonido	OK	
P2	Obtención de valores que representen el sonido obtenido	OK	
P3	Comunicación entre hebras	OK	
P4	Identificación de vehículos	OK	
P5	Generación de SDK en KAA	OK	
P6	Comunicación entre Python y Java	No se pudo creando un subprocesso desde Python.	Crear un Socket en Java.
P7	Integración entre Java y KAA SDK	OK	
P8	Envío de datos a Cassandra	ERROR	Abrir los puertos correctos.
P9	Uso de Scala para definir las de tiempo en Zeppelin	ERROR	Definir las funciones dentro de Cassandra, en Java.
P10	Conexión de Zeppelin y Cassandra	OK	
P11	Consultas en Cassandra	ERROR	Corregir el esquema de datos definido en Cassandra
P12	Escritura mediante rutas en la API	OK	
P13	Lectura mediante rutas de la API	OK	
P14	Actualización mediante rutas de la API	ERROR	El método a usar es PATCH y no PUT.
P15	Eliminación mediante rutas de la API	OK	
P16	Chequeo de tiempos de las señales recibidas	OK	
P17	Diseño responsive en la web	ERROR, Para web y tablets tiene una correcta visualización, para smartphones el mapa se redimensiona incorrectamente.	No solucionado.

Tabla 7.1: Pruebas realizadas sobre RSMap

Capítulo 8

Conclusiones y trabajos futuros

8.1. Conclusiones

En un resumen general del estado de RSMap se puede determinar que está listo para poner en producción. Seguramente surgirían algunos inconvenientes y esa es una de las *competencias* que bajo mi punto de vista se adquieren cuando se estudia en la Universidad, más en concreto en mi experiencia siendo alumno de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**. Me he demostrado la capacidad para resolver problemas, buscar información y alternativas ante los diferentes obstáculos que surgen cuando uno tiene que hacer combinar las diferentes piezas que conforman la aplicación.

El hecho de desarrollar un proyecto en el cual se usan muchos de los conceptos aprendidos durante el paso por las asignaturas hace que sea una experiencia enriquecedora.

El proceso de análisis sin duda alguna ha sido uno de los más duros debido a la incertidumbre de si el camino elegido será la aproximación más acertada de cara al avance y finalización del proyecto.

Por otra parte, la elección de L^AT_EX como soporte para la documentación ha acelerado por una parte el desarrollo de la misma generando muchos de los contenidos como índices, pies de foto o numeración sin embargo me ha supuesto un tiempo entender como funciona. No obstante ha sido una elección más que acertada en cuanto la calidad visual del documento.

El proyecto se encuentra alojado bajo el dominio www.rsmap.es y todo el código y documentación se encuentran en <https://github.com/RSMMap>. Si el lector desea ver la plataforma en funcionamiento, en el caso de que no le sea posible configurar su dispositivo de envío puede escribirme a luqueburgosjm@gmail.com y estaré encantado de activar mi dispositivo para que vea con más detalle el funcionamiento de RSMMap.

8.2. Trabajos futuros

RSMMap está abierto a muchas posibilidades que pasan desde anexar un sistema que estudie y saque conclusiones de los datos almacenados en Cassandra así como de monitorizar otro tipo de elementos ambientales gracias a que se pueden definir nuevas estructuras de almacenamiento y representación de una manera sencilla.

Haciendo un repaso por los objetivos planteados en un principio y los obtenidos finalmente RSMMap cuenta con las siguientes características:

- Permite reconocer cuano un vehículo pasa por un receptor si bien factores de ruio ambiental pueden influir de manera negativa sobre el resultado, sin embargo bajo condiciones óptimas las cuales son alcanzables en zonas no muy ruidosas o cuando existe menos ruido ajeno al tráfico los resultados son más que aceptables.
- Tras una serie de pasos, cualquier usuario puede aportar información al sistema.
- Los datos se encuentran almacenados en una base de datos con una potencia muy a tener en cuenta por tanto las técnicas de análisis de datos masivos son aplicables a RSMMap.
- La información que RSMMap contiene es representada de una manera limpia y elegante.
- Todos los componentes usados son Open Source.

Como reto personal me he propuesto seguir mejorando el algoritmo de detección de vehículos en ambientes con mucho ruido.

Glosario de términos

Big Data: Concepto que se refiere al almacenamiento de inmensas cantidades de datos.

IOT: Del inglés Internet of Things se refiere a la capacidad de dispositivos que poseen conexión a Internet y pueden comunicarse entre ellos así como con otros servicios

Raspberry Pi: Ordenador de bajo coste, con un procesador ARM cuyo consumo es bajo.

API Rest: Arquitectura que trabaja sobre el protocolo HTTP y sirve para interactuar con bases de datos mediante ficheros como pueden ser JSON ó XML.

Django Rest Framework: Módulo de Python que permite crear APIs REST de una manera fácil e intuitiva.

Django: Framework en Python para crear páginas webs.

Ajax: Del inglés, Asynchronous JavaScript And XML provee métodos para el desarrollo de webs interactivas y sin recargas.

Kaa: Plataforma de IOT Open Source.

SGBD: Acrónimo de Sistema Gestor de Bases de Datos.

Apache Cassandra: SGBD orientado al almacenamiento masivo de información.

Apache Zeppelin: Visualizador web que soporta diversos formatos y lenguajes.

Notebook: Cuaderno de Zeppelin, en el se pueden escribir tareas para que sean ejecutadas y cuya salida es devuelta al usuario.

ZigBee: Conjunto de tecnologías que permiten la comunicación entre dispositivos de manera inalámbrica y con baja latencia.

Open Source: Dícese del software cuyo código es ofrecido públicamente para su estudio, modificación o simplemente uso.

DAFO: (FODA) es una análisis de debilidades, amenazas, fortalezas y oportunidades.

Smart City: concepto que se refiere al uso de tecnologías aplicadas sobre una ciudad.

Apache Spark: sistema de procesado de datos masivo que permite aplicar técnicas como machine learning sobre los mismos.

Diagrama de PERT: diagrama para organizar tareas y actividades por tiempo.

Diagrama de GANTT: al igual que el diagrama de Pert ofrece una visión del tiempo a emplear en las actividades programadas y como se encadenan unas con otras.

RJ45: conector de red común entre los equipos informáticos.

NoSQL: base de datos sin estructura relacional. Normalmente los datos son almacenados en formato de diccionario.

Desarrollo Ágil: Metodología que define la forma de trabajo sobre un proyecto en forma de iteraciones de manera que en cada una de éstas iteraciones se completa un requisito u objetivo del proyecto.

Fork: Abrir un nuevo camino dentro de un proyecto para orientar el desarrollo hacia otra dirección.

Roadmap: Camino a seguir para completar ciertos objetivos definidos.

SO: acrónimo de Sistema Operativo

Python: Lenguaje de programación cuya sintaxis se caracteriza por su flexibilidad y su legibilidad.

LOPD: acrónimo de Ley Orgánica de Protección de Datos.

GitHub: sistema de control de versiones que contribuye al desarrollo del software libre.

Bootstrap: librería que hace uso de jQuery y CSS para ofrecer interfaces amigables.

Productor-Consumidor: Paradigma dentro de la computación paralela que trata un problema mediante un elemento que genera datos mientras otro los procesa.

Payload: Información asociada a una petición.

HTTP: Del Inglés, HyperText Transfer Protocol que utilizan las direcciones de internet para interactuar con los servidores.

VPS: Del Inglés, Virtual Private Server. Servidor personal normalmente conectado a internet.

JSON: Del Inglés, JavaScript Object Notation, archivo con estructura de diccionario usado para el paso de datos de un sistema a otro.

SQLite: Motor de base de datos implementado en C cuyo formato es únicamente un archivo sobre el que se escribe la información.

GET: Dentro de HTTP, petición que demanda información.

POST: Dentro de HTTP, petición que envía información.

PATCH: Dentro de HTTP, petición que actualiza información.

DELETE: Dentro de HTTP, petición que elimina información.

Backend: Parte lógica de un software.

Fronted: Parte visual de un software.

Middleware: Pasarela intermedia por la que diversos elementos intercambian información.

Webinars: Pequeñas conferencias Online orientadas generalmente a la formación sobre un tema.

CQL: Acrónimo de Cassandra Language Query, es usado por Cassandra para interactuar con el motor de datos.

CSS: Del Inglés, Cascading Style Sheet, es un lenguaje estructurado y que da formato a elementos HTML.

HTML5: Lenguaje de marcas orientado a la creación de páginas webs estáticas.

jQuery: Biblioteca en JavaScript que provee funciones para interactuar con el código HTML de manera intuitiva.

Anexo. Manual de Usuario para aportar datos a RSMap

En éste anexo se detallan los pasos a seguir para configurar un dispositivo y contribuir al aporte de datos a RSMap.

En primer lugar debemos asegurarnos de tener instalado Git cuya versión no es realmente importante, Python en su versión 3.5.2, pip en su versión 8 y Java en su versión 8.

```
1 $ git --version
2 git version 2.9.3
3 $ python --version
4 3.5.2
5 $ pip --version
6 pip 8.1.2 from /opt/virtualenvs/py3/lib/python3.5/site-packages
    (python 3.5)
7 $ java --version
8 openjdk version "1.8.0_102"
9 OpenJDK Runtime Environment (build 1.8.0_102-b14)
10 OpenJDK 64-Bit Server VM (build 25.102-b14, mixed mode)
```

Fragmento de código 8.1: Versiones requeridas

En caso de no tener alguna de las dependencias, será necesario instalarlas mediante el gestor de paquetes de la distribución que se esté usando.

Ahora nos descargamos el repositorio situado en la GitHub <https://github.com/RSMMap/RSMMapPi>.

```
1 $ git clone https://github.com/RSMMap/RSMMapPi
```

Fragmento de código 8.2: Descarga de repositorio

A continuación vamos a proceder a configurar los parámetros necesarios para que el receptor detecte los vehículos analizando las condiciones de la vía en la que se situa, para ello accedemos a la carpeta *resources* donde encontraremos el archivo *spectrogram.py*. Éste programa es un ejemplo que ofrece la librería SoundDevice y se ha modificado con el propósito de extraer el núme-

ro de bloques y cota mínima necesarios para una correcta identificación de vehículos.

También será necesario identificar el id del dispositivo asociado a nuestra tarjeta de sonido, para ello usamos:

```
1 $python spectrogram.py -l
```

Fragmento de código 8.3: Obtener los dispositivos de audio conectados

Que nos devuelve la lista de los dispositivos de audio. Tras identificar el correcto (en la mayoría de casos, denominado Default) procedemos a llamar a el script.

```
1 $python spectrogram.py -d <id del dispositivo>
```

Fragmento de código 8.4: Ejecutar la aplicación de configuración

Tras ésto se nos mostrará un espectrograma que representa mediante caracteres el nivel de ruido detectado. Al final de la línea aparecen dos números, el primero indica los bloques consecutivos que se han obtenido antes de volver al silencio y el segundo la suma de los valores para cada bloque.

Dichas variables las podemos identificar dentro del código como *consecutive_blocks* y *threshold*. Jugando con éstos valores podremos deducir cuando pasa un vehículo cuantos bloques ocupa así como el valor mínimo para tener en consideración que son válidos.

Una vez que las tengamos editamos el archivo *VehicleDetection* y sustituimos los valores obtenidos por los que el programa trae por defecto. Una vez hecho ésto ya estamos en condiciones de remitir datos a RSMap, sólo necesitamos compilar el programa *DataSender.java*, para ello nos dirigimos al directorio *sender* e introducimos las siguientes órdenes:

```
1 $ javac -cp kaa-java-ep-sdk.jar DataSender.java
```

Fragmento de código 8.5: Compilar *DataSender.java*

Ahora ya podemos ejecutar, convenientemente unos 45 segundos antes de lanzar *VehicleDetection.py* para dar tiempo a inicializar todo lo necesario.

```
1 $ java -cp .:/kaa-java-ep-sdk.jar:log4j-over-slf4j-1.7.7.jar:logback-classic-1.1.2.jar:logback-core-1.1.2.jar DataSender
```

Fragmento de código 8.6: Lanzar *DataSender*

Cuando esté listo, mostrará un mensaje de que se están esperando conexiones TCP.

Por último nos dirigimos al directorio *analyzer*, lo primero que vamos a hacer es instalar las dependencias con:

```
1 $ pip install -r requirements.txt
```

Fragmento de código 8.7: Instalar dependencias

Modificamos el archivo con los parámetros obtenidos anteriormente y lo lanzamos con:

```
1 $ python VehicleDetection.py
```

Fragmento de código 8.8: Ejecución de VehicleDetection.py

Si todo funciona correctamente las señales que envía éste módulo serán visibles en el mapa de RSMap y serán almacenadas en Cassandra al que se podrá acceder mediante Zeppelin.

Bibliografía

Fuentes principales (APIS y Docs):

- [1] Kaa docs. <http://docs.kaaproject.org/display/KAA/Kaa+IoT+Platform+Home>
- [2] Django docs. <https://docs.djangoproject.com/en/1.10/>
- [3] Zeppelin docs. <https://zeppelin.apache.org/docs/0.5.0-incubating/docs.html>
- [4] Python docs. <https://docs.python.org/3/c-api/>
- [5] Django Rest Framework docs. <http://www.django-rest-framework.org/#api-guide>
- [6] Java docs. <https://docs.oracle.com/javase/8/docs/api/>
- [7] Amazon docs. <https://aws.amazon.com/es/documentation/>

Webinars:

- [8] Kaa IOT. <https://www.youtube.com/watch?v=wL5vXuD-3nw>
- [9] Data Ingestion with Kaa. <https://www.youtube.com/watch?v=VHwJLHQj150>
- [10] Data Representation with Kaa. <https://www.youtube.com/watch?v=MFdQvlnptsM>

Artículos:

- [11] Data Modeling in Cassandra. 16/08/16 - Autor: DataStax http://docs.datastax.com/en/landing_page/doc/landing_page/dataModeling.html

- [12] How to Choose a Sensible Sampling Rate. 2/06/16 - Autor: David B. Stewart. <http://www.embedded.com/design/configurable-systems/4006414/How-to-Choose-A-Sensible-Sampling-Rate>
- [13] Music Database Retrieval Based on Spectral Similarity. 4/06/16 - Autor: Cheng Yang. <http://ilpubs.stanford.edu:8090/489/1/2001-14.pdf>
- [14] Sound Object Localization and Retrieval in Complex Audio Environments . 6/06/16 - Autores: Derek Hoiem, Yan Ke, Rahul Sukthankar. http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/dhoiem/publications/ICASSP2005_Hoiem.pdf

Dudas sobre el desarrollo de código:

- [15] StackExchange. <http://tex.stackexchange.com/>
- [16] StackOverflow. <http://stackoverflow.com/>
- [17] JSFiddle. <https://jsfiddle.net/>
<http://www.planetcassandra.org>
<https://github.com/kaaproject/sample-apps>

Software utilizado

- [18] Arch. <https://www.archlinux.org/>
- [19] Dia. <http://dia-installer.de/index.html.es>
- [20] Gimp. <http://www.gimp.org.es/>
- [21] Inkscape. <https://inkscape.org/es/>
- [22] Atom Editor. <https://atom.io/>
- [23] Latex. <https://www.latex-project.org/>

Material de asignaturas

Prácticas, entregas y transparencias pertenecientes a las asignaturas de **Procesamiento digital de señales, Desarrollo de aplicaciones para Internet, Fundamentos de Ingeniería del Software, Ingeniería de Servidores, Infraestructura Virtual y Sistemas concurrentes y distribuidos** impartidas en **Grado en Ingeniería Informática** en la Universidad de Granada.

