ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Desarrollo de un Acceso Asíncrono para la Provisión de Servicios de la Internet de las Cosas en la Ciudad Inteligente

(Development of a Websocket-based Access for a Smart City Internet of Things Platform)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Mario Núñez Callejo

Enero - 2019

AGRADECIMIENTOS

Después de un largo periodo de tiempo desarrollando este proyecto, quisiera agradecer toda la ayuda aportada por parte de *Luis Sánchez* y de *Pablo Sotres* los cuales han estado siempre receptivos a la hora de darme apoyo y poner a mi disposición todos sus conocimientos. Sin ellos la realización de este proyecto hubiera sido más laboriosa y menos gratificante. Siempre me llevaré vuestras enseñanzas.

A nivel más personal agradezco el apoyo moral de mi familia y amigos, el cual siempre me ha dado las fuerzas que necesitaba para seguir adelante incluso en los momentos más difíciles.

RESUMEN

El desarrollo imparable de la tecnología ha llevado a la humanidad a un crecimiento exponencial en la calidad de vida de las personas, facilitando el transporte, la búsqueda de información, las comunicaciones a distancia, entre otras muchas ventajas. En dicho crecimiento, la tecnología ha inundado diferentes ámbitos, uno de los más recientes son las ciudades.

Aparece el paradigma de ciudad inteligente al cual muchas localidades se han sumado y han decidido aprovechar el avance de las tecnologías TIC para ser más sostenibles y ofrecer a sus ciudadanos una serie de servicios de alto valor añadido basados en la capacidad de adaptarse en todo momento a las condiciones de contexto que se dan en la ciudad (p.ej: el tráfico, la contaminación, los recursos naturales, el clima, etc).

Para monitorizar estas condiciones de contexto se emplean una serie de dispositivos electrónicos, comúnmente denominados sensores, que tienen la capacidad de captar ese contexto (temperatura, contaminación atmosférica, estado del tráfico, nivel de llenado de una papelera, etc), y comunicarlo a través de Internet. El concepto de Internet de las Cosas (IoT, por sus siglas en inglés) se acuña precisamente para describir este escenario. El entorno que nos rodea en nuestro día a día, equipado con estos sistemas electrónicos conectados a Internet, incrementa la capacidad de ofrecer servicios en la red usando la información que estos pueden proveer.

El presente proyecto implementa una nueva posibilidad de acceso a los servicios ofertados por SmartSantander (SmS). Con el fin de optimizar la entrega de información a los clientes, se desarrolla un enlace asíncrono para la provisión de servicios en la ciudad inteligente SmartSantander utilizando la tecnología websockets. El uso de este nuevo acceso permitirá a los clientes suscribirse a ciertas observaciones captadas por los sensores, y recibir dicha información tan pronto como esté disponible. Además, utilizando la tecnología websockets se optimiza la interacción entre cliente y la plataforma SmartSantander eliminando también requerimientos tecnológicos antes necesarios para la provisión de dicho servicio, permitiendo así, que prácticamente cualquier cliente pueda recibir la información de contexto que desee.

Palabras Clave— SmartSantander, IoT, redes de sensores, Websockets.

ABSTRACT

The unstoppable development of technology has led the humanity to an exponential growth in the people's quality of life. It facilates trasport, the search of information, remote communications, among many other advantages. During this growth, technology has spread among many fields, one of the lastest are the cities.

The smart city paradigm appears, which many municipalities have joinned it recently. They have decided to take advantage of the development of ICT technologies to be more sustainable and offer their citizens a wide variety of added-value services based on the ability to adapt at all times to the context conditions of the city (e.g: traffic, pollutions, natural resources, climate, etc).

To monitor these context conditions, a series of electronic devices are used, commonly called sensors, which have the capacity to capture that context (temperature, air pollution, traffic conditions, filling level of a wastebasket, etc.), and communicate it through Internet. The Internet of Things concept (IoT) is coined precisely to describe this scenario. The environment that surrounds us in our daily lifes, which is equipped with these electronic systems connected to the Internet, increases the capacity to offer services in the network using the information they can provide.

This project implements a new possibility of access to the services offered by SmartSantander (SmS). In order to optimize the delivery of information to clients, an asynchronous interface is developed for the provision of services in SmartSantander using websockets technology. The use of this new access will allow clients to subscribe to certain observations captured by the sensors, and receive this information as soon as it is available. In addition, using websockets technology, the interaction between the client and the SmartSantander platform is optimized. It also eliminates technological requirements previously necessary for the provision of the same service. The technology used for the development of the interface that has been implemented and integrated in this project is meant to allow almost every client to receive the context information that they want.

Keywords— SmartSantander, IoT, sensor networks, Websockets.

Índice

Índice de Figuras

Índice de Tablas

Lista de Acrónimos

API Application Programming Interface (Interfaz de Programación de Aplicaciones).

API Key Application Programming Interface Key (Clave de interfaz de programación de aplicaciones)

CSS Cascading Style Sheets (Hojas de Estilo en Cascada)

ECMA European Computer Manufacturer Association (Asociación Europea de Fabricantes de Computadoras)

FIFO First In First Out (Primero en entrar, primero en salir)

HTML HyperText Markup Language (Lenguaje de Marcas de Hipertexto)

HTTP Hypertext Transfer Protocol (Protocolo de Transferencia de Hipertexto)

HTTPS Hypertext Transfer Protocol Secure (Protocolo Seguro de Transferencia de Hipertexto)

IoT Internet of Things (Internet de las Cosas)

JS JavaScript

JSON JavaScript Object Notation

LIFO Last In First Out (Último en entrar, primero en salir)

PUB/SUB Publish-Suscribe (Publicación-Suscripción)

REST REpresentational State Transfer (Transferencia de Estado Representacional)

SmS SmartSantander

TIC Tecnologías de la Información y la Comunicación

WS WebSockets

Capítulo 1

Introducción

La velocidad a la que el avance tecnológico, y en particular en el ámbito de las Tecnologías de la Información y las Comunicaciones (TIC), repercute en nuestro día a día, se ha disparado en los últimos años. La aplicación de estos avances en el ámbito de las ciudades ha generado recientemente el concepto de Smart City o ciudad inteligente. Aunque tiene múltiples connotaciones, el concepto por lo general se refiere a un tipo de desarrollo urbano cuyo objetivo es conseguir una mejora en la gestión de las ciudades y los servicios que se ofrecen a los ciudadanos. Con ello se busca crear un entorno de sostenibilidad que funcione eficientemente a todos los niveles.

La ciudad de Santander, a través del proyecto SmartSantander, es pionera en el desarrollo e implantación de este paradigma. Desde el inicio de este proyecto se han desplegado más de 20000 sensores por toda la ciudad con el objetivo de dotarla de capacidades para monitorizar un gran número de los parámetros que influyen de manera decisiva sobre ella y de algún modo, tener la posibilidad de adaptar en tiempo real los servicios que se ofrecen a los ciudadanos y visitantes de la ciudad, de forma que su devenir diario sea más confortable, eficiente y sostenible.

Las aplicaciones del concepto de Smart City están en un crecimiento continuo del mismo modo que lo están las tecnologías que soportan estas aplicaciones y servicios. El objetivo final es común en todos los casos, que los avances tecnológicos redunden en la mejora y ampliación de los servicios ofrecidos a los vecinos y visitantes de la ciudad.

1.1. Motivación

El proyecto SmartSantander se centra en el empleo del concepto de Internet de las cosas (IoT) como base para la consecución de la Smart City. Todos los dispositivos desplegados por la ciudad utilizan el concepto de IoT, que les permite formar redes inalámbricas de sensores que pueden comunicarse entre ellos y más importante, también con otras redes.

Estas redes de sensores que se comunican a través de Internet permiten poner a disposición de cualquiera con los permisos adecuados las observaciones realizadas por estos dispositivos. Será en base a esta información que se puedan generar los servicios de valor añadido que sustentan el paradigma de la Smart City. Para que estas aplicaciones y servicios se puedan proveer, es fundamental que se articulen los mecanismos por los cuales la información obtenida por los sensores sea accesible de manera sencilla y adecuándose a las necesidades de los desarrolladores de estas aplicaciones.

La distribución de los datos obtenidos por los sensores puede realizarse de diversas maneras que se complementan. La realización de una encuesta periódica para solicitar el último dato disponible es la opción más directa, pero también la más ineficiente ya que, si no ha habido ninguna actualización, la petición obtendrá el mismo dato que la petición anterior.

En muchas ocasiones, los sensores emiten sin una estructura definida a priori y funcionan por eventos, por tanto sería más óptimo tener un acceso asíncrono para que cuando se reciban los datos se notifique al usuario sin necesidad de que éste tenga que hacer una petición explícita. A este modo de operación se le denomina comúnmente suscripción-notificación.

En la actualidad, la plataforma de SmartSantander soporta ambas modalidades. La opción asíncrona está basada en la tecnología REST Hooks que requiere que el usuario disponga de un servidor HTTP al cual enviarle las notificaciones cuando se genera una nueva observación. Este requisito puede ser trivial en algunos casos, pero para clientes como puede ser una aplicación en un teléfono móvil, tener un servidor corriendo resultaría ineficiente y perjudicial para el rendimiento del dispositivo.

El proyecto que se ha desarrollado implementa un acceso asíncrono basado en suscripción-notificación a la plataforma SmartSantander mediante la tecnología websockets. Con este nuevo método se crea un socket a nivel de aplicación por el

cual el cliente se puede suscribir a las observaciones que desea recibir de forma que las notificaciones con dichas observaciones vuelven por ese mismo socket sin necesidad de que el cliente disponga de un servidor HTTP.

1.2. Objetivos

Una vez expuesta la motivación del proyecto, se determina el objetivo principal de este Trabajo de Fin de Grado como el desarrollo de un módulo basado en la tecnología websockets que implemente un acceso asíncrono a las observaciones generadas en la plataforma SmartSantander, de esta forma los clientes tienen la posibilidad de suscribirse a determinadas observaciones y recibirlas automáticamente cuando estén disponibles en SmartSantander.

Con el desarrollo de las aplicaciones IoT, el método PUB/SUB se posiciona como uno de los mejores para la obtención de datos en tiempo real. Con el presente Trabajo de Fin de Grado se busca complementar el mecanismo de acceso asíncrono basado en la tecnología de REST Hooks que actualmente ofrece la plataforma de SmartSantander de forma que se ofrezca una alternativa a los clientes para los que la actual no es apropiada.

Además de la creación de suscripciones, se hace énfasis en la gestión del ciclo de vida de las mismas, dotando al cliente de la capacidad para decidir el estado y la perdurabilidad o no de dichas suscripciones en todo momento.

La definición del objetivo principal, que aporta unos propósitos claros a llevar a cabo, es a su vez divisible en dos finalidades claramente diferenciadas, las cuales son:

- Desarrollo de un módulo que tenga como componente principal un servidor que utilice la tecnología websockets y que implemente las interfaces necesarias para manejar todo el ciclo de vida de las suscripciones realizadas para el acceso a servicios IoT según un modelo PUB/SUB.
- Integración de este módulo en la plataforma desplegada por el proyecto SmartSantander para la recepción de las observaciones y su procesado previo a su provisión a los suscriptores correspondientes.

1.3. Estructura del proyecto

Este documento sigue una estructura basada en 4 capítulos principales a través de los cuales se proporciona al lector una visión del trabajo que se ha realizado y de todos los elementos que forman parte del mismo.

Previo a los capítulos principales está definido un resumen y un abstract que aportan un enfoque general de todo el proyecto, facilitando la compresión de los siguientes apartados.

El primer capítulo es introductorio y en él se presentan tanto los objetivos que se pretendían alcanzar como las circunstancias que los motivan.

Seguidamente, el segundo capítulo analiza los conceptos teóricos y el marco en el que se ha llevado a cabo el proyecto. Se hace una revisión del estado del arte explicando todos los componentes necesarios para la implementación de la arquitectura, las herramientas empleadas y las tecnologías utilizadas en el proyecto.

Una vez introducidos todos los aspectos y tecnologías con un papel relevante en el desarrollo del proyecto, el capítulo 3 describe exhaustivamente el componente que se ha implementado en este Trabajo de Fin de Grado, desde su diseño hasta los detalles más importantes de su implementación, así como el proceso de integración de este componente en la plataforma SmartSantander.

Para finalizar, se analizan las conclusiones a las cuales se ha llegado y se exponen ideas para posteriores investigaciones o ampliaciones de este trabajo.

Capítulo 2

Conceptos teóricos

Para poder seguir la descripción del trabajo desarrollado, es necesario introducir con anterioridad las principales tecnologías y elementos que conforman el marco en el que se ha desarrollado el mismo. En este capítulo se hace un breve repaso al estado del arte en el que se encuentran estas tecnologías y se presentan las herramientas que se han utilizado para la implementación.

2.1. IoT

Fue en 1999 la primera vez que se acuñó la expresión Internet de las Cosas y desde entonces ese concepto ha ido evolucionando y su uso ha ido creciendo de manera exponencial.

Al ser algo relativamente reciente, la IoT está aún en desarrollo y es por eso que muchos investigadores se toman la libertad de añadir, quitar o modificar ciertos aspectos de este término. La idea que subyace a la IoT, según Kevin Ashton, el primero en utilizar esta expresión, es básicamente una red de objetos físicos cotidianos que utilizan sensores y APIs para conectarse e intercambiar datos a través de Internet.

Hasta el momento en el que aparece la IoT, los contenidos y servicios a los que se podía acceder en Internet tenían un origen "humano". Los programas informáticos y aplicaciones son desarrolladas por las personas, al igual que páginas web y demás contenido de Internet.

Casi toda esa información contenida en la red es proporcionada por el ser humano. A priori esto puede parecer trivial, pero el problema es que los seres

humanos estamos condicionados, nos limitan muchos factores como son el tiempo, la atención, la precisión y demás imperfecciones intrínsecamente humanas, es por esto que no somos especialmente eficientes a la hora de extraer datos sobre lo que nos rodea.

Claramente se observa la necesidad de conferir poderes a ciertas máquinas para la recopilación de información en nuestro entorno, de modo que por ellas mismas puedan ver, oír y, en definitiva, percibir el mundo. Para conseguir este propósito es necesaria la utilización de dispositivos que sean capaces de tomar este tipo de medidas. Este tipo de dispositivos deberán tener unas características específicas para poder llevar a cabo esta tarea de manera adecuada. No sólo es necesaria la medición de ciertos parámetros, sino que además se requerirá que esos datos se transformen en información y servicios de valor añadido.

Por lo tanto, el verdadero propósito de la IoT no es sólo medir ciertos datos del entorno, sino que es la medición y distribución de los mismos a las aplicaciones y servicios que puedan aportar valor en base a esos datos. Por tanto, el desarrollo de un sistema IoT está al menos formado por cuatro componentes distintos:

1. Sensores:

   Los sensores son dispositivos electrónicos que están capacitados para detectar acciones o estímulos externos y, en algunos casos, reaccionar ante ellos. Un sensor tiene una propiedad sensible a una magnitud del medio, y al variar esta magnitud también varia con cierta intensidad la propiedad, es decir, manifiesta la presencia de dicha magnitud, y también su medida. Para la cuantificación y medición de los resultados deberán transformar a su vez la magnitud física en una señal eléctrica que sea comprensible para otro componente electrónico. Estos dispositivos, al estar desplegados por el entorno y algunos ser inalámbricos, deben estar dotados de cierta autonomía para que no sea necesario reponer su energía en cortos periodos de tiempo.

2. Conectividad:

   La medición de un cierto dato por un sensor no aporta nada por si misma, esa información debe enviarse a un sitio accesible por otras personas o máquinas, es por lo que los sensores deben estar conectados entre sí para enviarse los datos medidos. La información recogida se enviará a un *gateway* que servirá como puerta de enlace para enviarla hacia el exterior, fuera de la red de sensores.

3. Procesamiento de los datos:

   Si bien es posible consumir los datos primarios, el verdadero potencial de la IoT aparece cuando se procesa la ingente cantidad de medidas reportadas por los sensores y de ese procesado extrae un conocimiento de valor añadido. Hay multitud de variantes para este proceso de análisis de datos. Desde la extracción de patrones con fines predictivos hasta la propia detección de datos anómalos que deben eliminarse.

4. Interfaz de usuario:

   Para terminar, la información debe ser proporcionada al usuario final de algún modo. Sería de gran utilidad que, además de aportar esa información, se pudiese avisar a los clientes de cuándo están disponible dichos datos.

Esta catalogación básica es más o menos común para cualquier sistema IoT, pero cada plataforma o aplicación de la IoT puede priorizar u omitir alguna de sus partes según los requerimientos y criterios de diseño que se hayan elegido.

2.2. SmartSantander

SmartSantander es un proyecto de investigación europeo que desarrolla una infraestructura para el desarrollo de arquitecturas, tecnologías facilitadoras esenciales, servicios y aplicaciones para la Internet de las Cosas, orientándose hacia el concepto de ciudad inteligente.

SmartSantander consta de un enorme despliegue de sensores en las ciudades de Belgrado, Guildford, Lübeck y Santander, con mayor importancia en esta última con un despliegue únicamente en esta ciudad de aproximadamente 12000 sensores. La Unión Europea financió el proyecto con 8.7 millones de euros entre 2008 y 2012, con el objetivo de crear la primera y única infraestructura experimental en el mundo en el campo de la Internet de las Cosas. La realización de dicho proyecto fue posible gracias al trabajo conjunto de distintas universidades (del Reino Unido, Alemania, Grecia, Dinamarca, España y Australia) y empresas como Telefónica I+D (España), Alcatel-Lucent S.P.A. (Italia/España), Ericsson D.O.O. (Serbia), y por supuesto, el municipio de Santander. La coordinación del proyecto fue realizada por Telefónica I+D, pero con la inestimable colaboración del Departamento de Ingeniería de Comunicaciones (DICOM) de la Universidad de Cantabria.

El proyecto SmartSantander supone el despliegue de miles de sensores por toda la ciudad y la puesta en en marcha de varias aplicaciones que utilizan la información que recogen para ponerla a disposición de los ciudadanos.

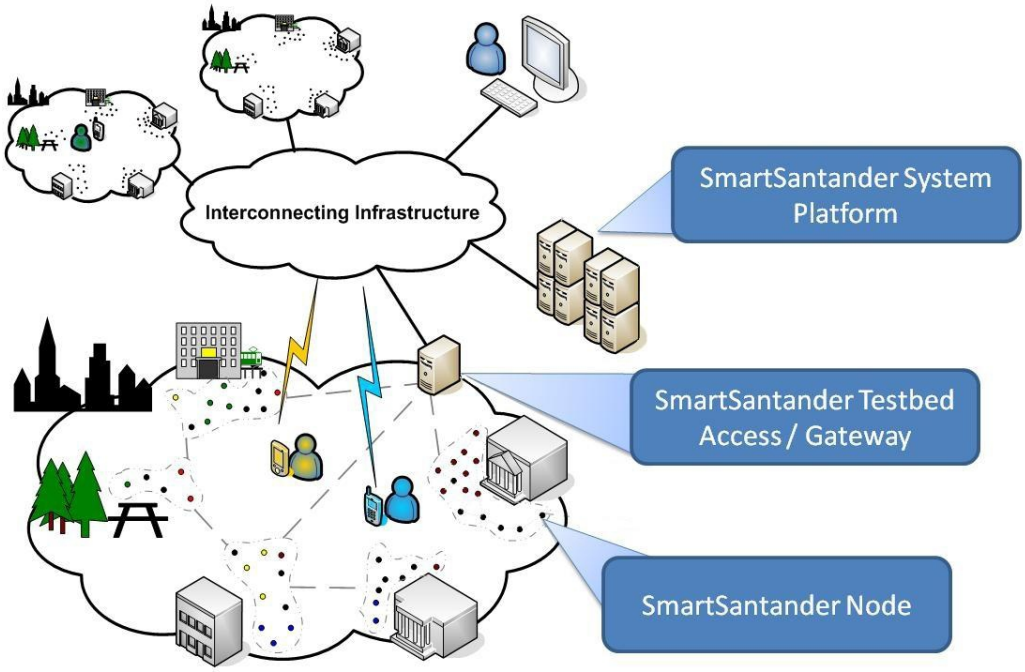

Figura 2.1: Infraestructura de SmartSantander

Tal como se ve en la arquitectura global de SmartSantander en la Figura 2.1, los sensores captan información del entorno, después esa información es enviada a un gateway que es quien finalmente envía a los servidores de SmartSantander los datos recibidos. Ahí se almacenan las observaciones captadas por los sensores a la espera de hacerse efectivas para los clientes que deseen beneficiarse de dicha información.

El objetivo de SmartSantander es el aprovisionamiento de servicios a los ciudadanos y visitantes (como conocer plazas de estacionamiento libres de la ciudad, grados de temperatura en distintas zonas, niveles de contaminación, etc). Asimismo, se implementa una gestión eficiente de los recursos de la ciudad, como el agua y el alumbrado público, gracias a la sensorización del espacio urbano se conocen las situaciones ambientales de todas las partes de la ciudad, lo que permite tomar medidas para lograr un ahorro en recursos y una mayor comodidad de los habitantes de la ciudad. Un claro ejemplo de esto son los sensores lumínicos instalados en el alumbrado público permiten adaptar la frecuencia y la intensidad de la iluminación de cada zona a las necesidades reales.

De este modo, se convierte el espacio urbano en un lugar en el que la comunidad científica puede investigar y en el que el mundo empresarial pueda desarrollar aplicaciones y productos tecnológicos. La ciudad facilita así la generación de nuevos modelos de negocio e ideas empresariales, al tiempo que se generan nuevos servicios para los ciudadanos.

2.3. Asincronismo

A la hora de implementar un módulo que tenga que gestionar medidas generadas por sensores, es necesario plantearse la manera en la que se va a abordar dicha tarea. Para ello, el principal aspecto en el que debemos fijarnos es la forma en la que dichos sensores interactúan con el sistema así como la manera en la que los consumidores de la información demandarán los servicios del módulo a desarrollar.

En particular, es importante destacar el comportamiento temporal tanto de sensores como de usuarios. Por lo general, los sensores tienen un comportamiento temporal periódico, sin embargo cada sensor o tipo de sensor tendrá su propio periodo no sincronizado con el de sus vecinos, tanto si son del mismo tipo como si no lo son. Además, existen sensores que reportan eventos intrínsecamente no sincronizados con ningún reloj. En este contexto, podemos concluir, que de una u otra manera, la plataforma de gestión de los sensores debe ser capaz de manejar este comportamiento fundamentalmente asíncrono de la infraestructura subyacente.
Directamente vinculado con este comportamiento subyacente, las aplicaciones y servicios consumidores de esta información también demandarán que el sistema les informe sin necesidad de hacer peticiones periódicas sino que se les notifique de la llegada de cada nueva medida.

Con estos condicionantes de entre todos los modelos de programación síncrona o asíncrona que existen a la hora de abordar un desarrollo software, parece adecuado decantarse por la programación asíncrona ya que se corresponde de una manera más cercana con el comportamiento que se pretende implementar.
En cualquier caso, en esta sección se hace un repaso a estos dos modelos de programación para que el lector pueda comprender mejor el porqué de la decisión que llevó a que para este TFG se escogiera un entorno de programación asíncrona.

2.3.1. Programación secuencial

Antes de entender la programación asíncrona y todo lo que implica, es necesario entender el concepto de sincronismo. Tradicionalmente, los programas informáticos se programaban y se ejecutaban de un modo secuencial, es decir, que entendiendo un programa como una secuencia de instrucciones, éstas se ejecutarían en el orden que aparecen, de modo que, la ejecución de cada operación no da comienzo hasta que no termina la anterior.

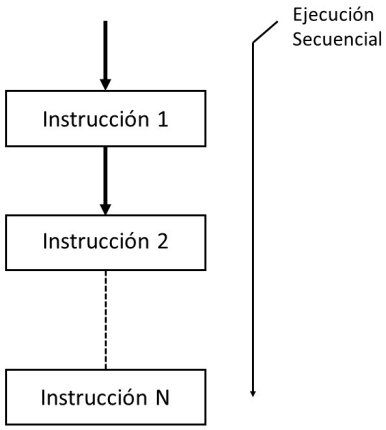

Figura 2.2: Programación Secuencial

En la Figura 2.2, se muestra la ejecución secuencial de las instrucciones en este tipo de programación. Se dice que las operaciones son bloqueantes debido a que bloquean el flujo de ejecución del programa ya que se impediría que la siguiente instrucción comience hasta que no termine la anterior.

2.3.2. Programación asíncrona

Ejecutando un código secuencialmente nos encontramos con operaciones que bloquean la ejecución del código. En primer lugar, hay que diferenciar entre operaciones limitadas por CPU (*CPU-bound*) y operaciones limitadas por entrada salida (*I/O bound*).

Las tareas que consumen de recursos de CPU son aquellas operaciones cuyo código asociado será ejecutado en nuestra aplicación. Esto será limitante, ya que si la CPU está ocupada no puede ejecutar otra tarea hasta que se libere.
Si no son operaciones excesivamente complejas o la potencia de la CPU es

considerablemente baja, el bloqueo de la ejecución daría lugar a tiempos de espera triviales.

Por otro lado, es común encontrar en los programas otro tipo de operaciones que necesitan comunicarse con otros subsistemas, por ejemplo la lectura de un fichero en disco o el acceso a una base de datos externa. Estas operaciones de entrada/salida desencadenan peticiones especiales que son atendidas fuera del contexto de la aplicación.
Al realizar peticiones que dependen de otros sistemas, éstas pueden tardar un tiempo considerable, que muchas veces no se puede predecir, lo que produciría bloquear la ejecución durante todo ese tiempo.

Mediante la programación asíncrona, la ejecución de la aplicación se hace más eficiente. Mientras se espera la respuesta de la petición realizada se puede seguir ejecutando instrucciones del código en segundo plano. Cuando finalmente llegue la respuesta, ésta será procesada.
De esta forma el tiempo que se pasa esperando, en realidad no es tiempo perdido ya que se sigue ejecutando código.

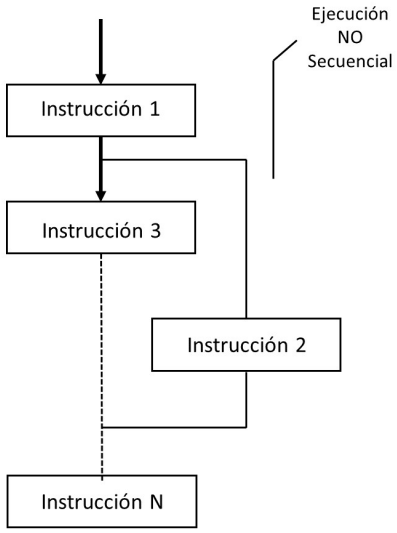

Figura 2.3: Programación Asíncrona

Algunas operaciones son no bloqueantes ya que devuelven el control al programa llamante antes de su finalización y de este modo se siguen ejecutando instrucciones en segundo plano, es por lo que ahora la ejecución no es secuencial.
Por ejemplo, en la Figura 2.3 se ejecuta la instrucción 2, al tener que esperar datos desde fuera de la aplicación, se devuelve el control al programa y, mientras espera

la respuesta, se siguen ejecutando instrucciones. Cuando finaliza la instrucción 2 se trata lo obtenido y se vuelve a seguir la ejecución del programa.

La asincronía, entre otras ventajas, permite que los siguientes procesos a una operación no bloqueante adelanten su ejecución, de este modo sería posible un aumento de la escalabilidad.

2.4. JavaScript

JavaScript es uno de los lenguajes de programación que más ha crecido en los últimos años, esto puede ser debido a que sus usos se han ido adaptando progresivamente a lo largo de los años.

2.4.1. Evolución

En sus inicio, Internet era básicamente una red de ordenadores conectados entre si, esto permitía que una persona desde su ordenador pudiera leer documentos alojados en otra computadora. Usando el lenguaje HTML el navegador de un ordenador permitía mostrar estos documentos. Este es el concepto más básico de web. Adicionalmente se puede añadir CSS para darle estilos a esa página.

En 1995 la empresa Netscape desarrolló JavaScript. Aunque originalmente tuviese otro nombre, este lenguaje de programación fue creado con la finalidad de poder embeber pequeños scripts en las páginas HTML. Inicialmente, estos scripts embebidos son ejecutados en el navegador, por lo que al no tener ninguna interacción con el servidor se dice que se ejecutan en el lado del cliente, y permiten interactuar con las páginas web.

A pesar de ser estos los inicios de JavaScript, hoy en día esto ha evolucionado considerablemente y su uso ha tenido un crecimiento exponencial.
Inicialmente sólo se podía ejecutar en el navegador, pero hoy en día ya permite interacción con el servidor permitiendo el desarrollo de backend o incluso de aplicaciones completas.

2.4.2. Estandarización

Con el éxito inicial de JavaScript, fueron muchas las empresas que decidieron incorporar su propia implementación del lenguaje, lo cual podría suponer una guerra de tecnologías, por lo que se decidió estandarizar el afamado lenguaje JavaScript.

ECMA es el nombre de la asociación europea de fabricantes de computadoras (European Computer Manufacturer Association). Se trata de una organización sin ánimo de lucro que se encarga, entre otras cosas, de regular el funcionamiento de muchos estándares de la industria mundial.
En el año 1997 se consigue estandarizar JavaScript, sus estándares empezaron a ser regidos como ECMAScript. No solo JavaScript se basa en el lenguaje ECMAScript, existen otros como JScript que también lo hacen.

Versión	Nombre oficial	Descripción
ES1	ECMAScript 1	Primera versión
ES2	ECMAScript 2	Sin cambios muy significativos
ES3	ECMAScript 3	Añadido *try/catch*
ES4	ECMAScript 4	Nunca se lanzó
ES5	ECMAScript 5	Soporte de JSON
ES6	ECMAScript 2015	Promesas nativas
ES7	ECMAScript 2016	Función *async*
ES8	ECMAScript 2017	Sin cambios muy significativos

Tabla 2.1: Versiones de ECMAScript

En la Tabla 2.1 se muestran las versiones más importantes de ECMAScript con sus principales funcionalidades añadidas.

2.4.3. Características

La evolución de JavaScript fue intencionadamente orientándose hacia las operaciones de entrada/salida, por esta razón, JavaScript utiliza un modelo asíncrono y no bloqueante, con un loop de eventos implementado con un único thread para sus interfaces de entrada/salida. La asincronía y el no bloqueo permite a JavaScript que, a pesar de utilizar un único hilo, pueda ser concurrente, es decir, que dos o más tareas puedan progresar simultáneamente.

El loop de eventos es un modelo que permite despachar mensajes en concurrencia con las tareas del programa. Para que este modelo tenga sentido es

necesaria la intervención de ciertos componentes:

Call Stack Es una pila de llamadas que se encarga de albergar las instrucciones que deben ejecutarse. Nos indica por qué punto del programa vamos. El funcionamiento de la pila es LIFO (Last In First Out), el último en entrar es el primero en salir, de modo que, por ejemplo cada vez que el programa llame a una función, entra a la pila por arriba generándose un frame que es un bloque de memoria reservado para argumentos y variables locales de esa función, cuando se completa la función y retorna, su frame sale de la pila por arriba también.

Heap Es una región de memoria libre dedicada al alojamiento dinámico de objetos.

Queue Es una cola de mensajes pendientes en la que, cada vez que el programa recibe una notificación de otro contexto distinto, se inserta el mensaje y se registra la función que se ejecutará como respuesta. A esa función se la denomina callback.

Loop de eventos En líneas generales, es el mecanismo para despachar los mensajes. Con cada tick del bucle de eventos se procesa un nuevo mensaje, esto consiste en, llamar al callback asociado a cada mensaje lo que creará un nuevo frame en el Call Stack pudiéndose derivar en varios frames, cuando la pila se vacié habrá terminado de procesar el mensaje.
Cuando el Call Stack está vacío y no queda nada por ejecutar, se procesan los mensajes de la cola.

Es por esto que es posible la asincronía, ya que, cuando se hace una petición de operación de entrada/salida, se devuelve el control de la ejecución al programa mientras se ejecuta esa operación fuera del contexto de la aplicación y cuando acaba se señaliza enviando una notificación que se pondrá en la cola de mensajes, Queue, quedando pendiente de procesar, por lo que como el loop de eventos va procesando mensajes cada vez, cuando se vacíe el Call Stack pasará a la cola donde está el mensaje de la operación entrada/salida y será procesado cuando llegue su turno.

De esta forma, el modelo asíncrono funciona, ya que ante una operación de entrada/salida, no se bloquea el código mientras se espera respuesta, y cuando ésta

llega se pone a la cola de mensajes y será procesada cuando se finalice lo que estaba pendiente.

2.4.4. Mecanismos para controlar la ejecución asíncrona

Para controlar la asincronía en el código se pueden utilizar diferentes mecanismos, los más conocidos son los callbacks, las promesas y async/await.

Los callbacks es la forma más primitiva para permitir que JavaScript sea asíncrono. Es la base y, en lo que se fundamentan el resto de patrones asíncronos.
Un callback es simplemente una función que se pasa como argumento de otra función, y que será ejecutada cuando la función a la que es pasada haya terminado de realizar todo lo que está pendiente. Básicamente un callback es lo que determina qué se hace cuando una operación asíncrona notifica que ha terminado.

Uno de los mayores problemas de los callbacks es el conocido callback hell, que es una situación en la cual se anidan suficientes llamadas asíncronas como para dificultar la legibilidad del código y crear cierto grado de confusión.
Es por eso que existen otros métodos que mejoran esta deficiencia de los callbacks.

Por otro lado están las promesas, una promesa es un objeto que representa el resultado de una operación asíncrona. Las promesas se basan en callbacks pero consiguen hacerlos más comprensibles y amenos.

Cuando se llame a una función asíncrona trabajando con este mecanismo, nos devolverá inmediatamente una promesa como garantía de que la operación asíncrona finalizará en algún momento, ya sea exitosamente o por un error.
Cuando se obtiene el objeto promesa, se definen dos callbacks, uno para indicar lo que debe hacer si todo fue correcto, promesa resuelta, y el otro para tomar acciones en caso de fallo, promesa rechazada.

Las promesas supusieron un gran avance frente a los callbacks permitiendo mayor elegancia en el código, y por tanto, mejor legibilidad, pero a veces el encadenamiento de promesas puede resultar también tedioso. El nacimiento de las palabras clave *async* y *await* tiene como objetivo hacer las promesas más amigables, pero en lo fundamental siguen siendo lo mismo.

Se puede declarar una función asíncrona con la etiqueta *async* e indica que una promesa será automáticamente devuelta. Por otro lado, *await* debe ser usado siempre dentro de una función declarada como async y esperará automáticamente,

de forma asíncrona y no bloqueante, a que una promesa se resuelva.

2.5. Node.js

Una de las claves de la evolución del uso de JavaScript fue cuando en 2009, el ingeniero Ryan Dahl, innovando y saliéndose de los inicios de JavaScript desarrolló Node.js.

Originalmente, JavaScript fue diseñado para ejecutarse solo en navegadores, de modo que cada navegador tenía un motor específico que permitía ejecutar código JavaScript, aunque en realidad no está limitado solo a los navegadores, ya que JavaScript puede ejecutarse en cualquier entorno que tenga instalado el intérprete para sus instrucciones

Ryan Dahl tomó el motor de Chrome, llamado V8, y lo incrustó dentro de un programa C++, a todo el conjunto lo llamó Node. De modo que Node es simplemente un programa en C++ que lleva embebido el motor V8 de Chrome, de esta forma se puede ejecutar código JavaScript fuera del navegador. Dicho de otro modo, Node.js es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome que nos brinda la posibilidad de pasarle código JavaScript para que lo ejecute. Esto permite ejecutar JavaScript en el lado del servidor, ofreciendo la posibilidad de construir el backend de una web o incluso aplicaciones móviles. Con este avance se puede tener el mismo lenguaje en el cliente y en el servidor permitiendo unificar lenguajes y reutilizar código.

Node.js emplea un único hilo y un bucle de eventos asíncrono. Las nuevas peticiones son tratadas como eventos en este bucle. Estas características son el motivo por el que JavaScript encaja tan bien con la filosofía de Node.js. Además, esto hace que Node.js sea capaz de gestionar múltiples conexiones y peticiones de forma muy eficiente. La comunicación no bloqueante y los eventos asíncronos lo hacen idóneo para el desarrollo de aplicaciones que manejan datos en tiempo real.

2.6. Arquitectura de software

La construcción de un sistema software puede llegar a ser extremadamente laboriosa. Con el tiempo se han ido desarrollando formas y guías mediante las cuales se facilita la forma de abordar los posibles problemas que surjan. A estas

directrices se las conoce como arquitectura de software. Cada sistema podrá utilizar ciertas pautas que es lo que acabará definiendo el estilo arquitectónico.

Además, para el correcto funcionamiento de cualquier sistema software se debe conseguir la comunicación entre los distintos bloques funcionales que componen el sistema, formándose así una interacción efectiva entre todos los componentes.

2.6.1. API

Una API (Application Programming Interface) es un conjunto de reglas que describen cómo una aplicación puede interactuar con otra.
Por lo general en la red, hay distintos computadores con diferentes sistemas operativos diversos lenguajes de programación con los que fueron construidos, como es lógico estos sistemas de alguna manera se tienen que comunicar con otros, el concepto de API es una especie de software intermedio que permite comunicarse a dos softwares distintos. Una API representa la capacidad de comunicación entre componentes de software.

2.6.2. REST

A la hora de definir un API se pueden elegir distintos tipos de arquitecturas en base a los requerimientos.
REST (REpresentational State Transfer) es un estilo de arquitectura vinculado al protocolo HTTP. Consiste en una serie de directrices que se deben cumplir en el diseño de la arquitectura de una API.

Los principios más importantes que definen la arquitectura REST son los siguientes:

1. Todo los datos que se transfieren a través de Internet son un recurso representado por un formato.

2. Cada recurso tiene un identificador único, existe una URI única que los identifica. Los recursos podrán ser manipulados a través de la URI.

3. El protocolo de transmisión de datos utiliza los métodos de HTTP para indicar la acción que se desea realizar sobre un recurso determinado. Los más comunes son *GET* para obtener del servidor un recurso, *POST* para crear un recurso

del servidor, *DELETE* para borrar recursos del servidor, y *PATCH* para la modificación parcial de un recurso.

4. Protocolo cliente/servidor sin estado, cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla. El servidor trata a cada petición como algo independiente y la comunicación siempre devuelve un estado final, es decir, devolvería el recurso completo.

Todo desarrollo basado en API REST separa el cliente del servidor, lo que, entre otras cosas, permite tener independencia de la tecnología con la que se implemente y el lenguaje con el que se desarrolle, es decir, siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una gran libertad a la hora de cambiar o probar nuevos entornos dentro del desarrollo. Con una API REST se pueden tener servidores en cualquier lenguaje, lo único que es indispensable es que las respuestas a las peticiones se hagan siempre en el lenguaje de intercambio de información usado, normalmente XML o JSON.

2.6.3. Tecnologías para APIs asíncronos

Por su dependencia del protocolo HTTP, los API basados en REST tienen un comportamiento eminentemente síncrono. Esto es, se basan en el paradigma petición-respuesta y por ello no se adaptan correctamente a la extracción de información que se genera de manera asíncrona.

Existen, sin embargo, tecnologías que también se basan en el protocolo HTTP y que se utilizan para completar los API REST en aquellas funciones con un comportamiento asíncrono.

2.6.3.1. Long Polling

Una de las primeras tecnologías orientadas al concepto asíncrono es la conocida como Long Polling. Para su correcta explicación es necesario hacer una pequeña revisión de cómo se llegó a su desarrollo.

Internet utiliza un modelo cliente-servidor, ya que reparte las tareas en dos roles, el servidor que es un proveedor que proporciona recursos o servicios y el cliente que es un consumidor que contacta al servidor demandando los recursos que

este provee. Para la transferencia de información entre clientes y servidores se usa el protocolo de comunicación HTTP o HTTP seguro (HTTPS).

La interacción entre clientes y servidores puede hacerse de varias formas, la inicial fue el método petición-respuesta, el cliente pide algo al servidor y éste le responde. Cada vez que se mande una petición al servidor web habrá una respuesta asociada a ella, por tanto, cuando una petición se manda habrá una conexión TCP creada entre el cliente y el servidor, y cuando se reciba la respuesta esa conexión será cerrada. Básicamente, por cada petición que se haga al servidor habrá una respuesta y también una conexión.

En los casos que se necesite que los datos se actualicen sin interacción del usuario cabe la posibilidad de usar el método Polling. Mediante este mecanismo los cambios en los datos son buscados en intervalos de tiempo, y se mantendrá la búsqueda independientemente de si los datos han cambiado o no.

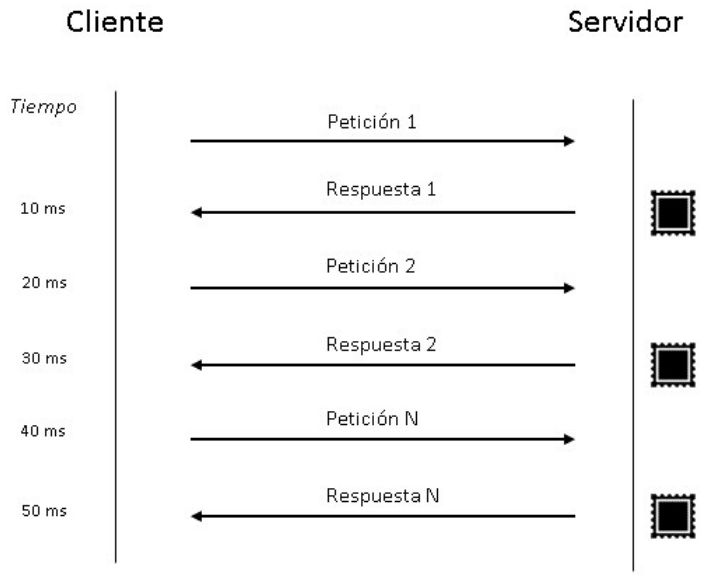

Figura 2.4: Polling

Como se ve en la Figura 2.4, cada cierto tiempo se hace una petición y se obtiene una respuesta, los datos enviados por el servidor pueden ser los mismos en varias respuestas si se da el caso de que no ha habido cambios. Esto provocaría llamadas innecesarias al servidor, abriendo una conexión y luego cerrándola cada vez.

Una variante de este método sería Long Polling. En este caso cuando el servidor

recibe una petición desde el cliente si no hay cambios en los datos solicitados en lugar de devolver una respuesta vacía guarda la petición y espera a que haya algún cambio en los datos para enviarlos al cliente, o después de un periodo de tiempo determinado, en ese momento el cliente volverá a lanzar otra petición que abrirá una conexión que se mantendrá abierta durante un cierto periodo de tiempo, el cual puede que sea suficiente para que se realicen cambios en el servidor y puedan ser reportados en su respuesta.

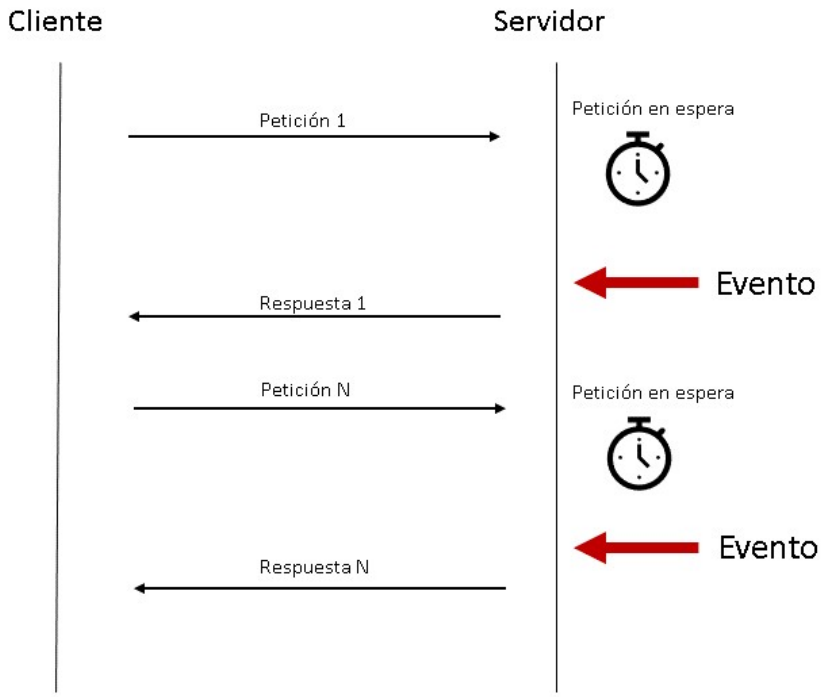

Figura 2.5: Long polling

En la Figura 2.5 se puede observar como, mediante el método Long Polling, el cliente realiza una petición al servidor con el objetivo de recibir cierta información, al no estar esos datos disponibles o no haberse actualizado desde el último requerimiento de información, la petición se mantiene en espera hasta que ocurra el evento que haga tener al servidor los datos requeridos por el cliente, después se envían esos datos en la respuesta.
Nada más recibir los datos, el cliente realiza otra petición manteniendo nuevamente una conexión abierta para que el servidor le pueda mandar nuevos datos cuando los tenga.

Dependiendo de la intensidad del tráfico de datos será mejor usar un mecanismo

u otro. Para aplicaciones de procesamiento intensivo, el método *long polling* puede llegar a encontrarse con mayores latencias en la transmisión continua de datos al cliente. Además, se debe configurar para que las conexiones no permanezcan vivas indeterminadamente, lo que podría causar re-conexiones constantes.

2.6.3.2. RestHooks

El método Long Polling, a pesar de ser más efectivo que el Polling para entornos asíncronos, sigue presentando ineficiencias claras, debido a que solamente puede mantener abierta una conexión por cierto tiempo, lo que desencadenará a reconexiones o peticiones que no obtienen respuesta por no haberse recibido ningún dato nuevo.
Para solucionar este problema se cuenta con el método publicación-suscripción (PUB/SUB), el cual permite a los clientes suscribirse a determinada información e inmediatamente cuando esté disponible les será enviada.

Los REST Hooks son una tecnología que permite al cliente recibir notificaciones enviadas por el servidor, por medio de suscripciones, justo en el momento que son publicadas .
La arquitectura REST está basada en HTTP, el cual es síncrono, para mejorar su rendimiento en entornos en los cuales la variación de los datos se debe a eventos, se utilizan los REST Hooks, implantando la asincronía mediante el método PUB/SUB.

El uso de los REST Hooks consiste en la suscripción del cliente a una determinada información, en esa suscripción se le indica al servidor qué información desea recibir y una dirección (*hook*) a la cual se deberán enviar dichos datos. El servidor responderá a esta petición indicando al cliente si es posible llevar a cabo la suscripción. Si la respuesta es negativa se cerrará la conexión y el cliente no estará suscrito a nada, si por el contrario es positiva, el servidor cerrará la conexión efectuando la suscripción del cliente.
Cuando la información requerida esté disponible en el servidor, este la enviará al *hook* que se aportó en la suscripción y de este modo, el cliente recibirán los datos que en un principio solicitó.

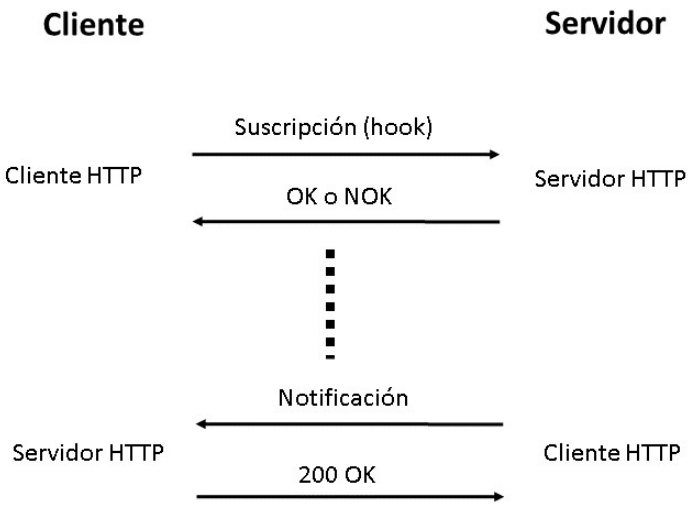

Figura 2.6: REST Hooks

En la implementación de los REST Hooks de la Figura 2.6, se muestra el proceso de suscripción de cierta información y su posterior recepción.
Se aprecia la necesidad de que el cliente disponga de un servidor HTTP debido a que para el envío de información se necesita un servidor como *endpoint*, el cual reciba los datos. Por tanto, se requiere de la utilización de un servidor HTTP por parte del cliente.

Esta es la principal desventaja del uso de los REST Hooks, que se le pone la condición al cliente de que disponga de un servidor. Este requerimiento supondrá un obstáculo para clientes que no puedan permitirse tener corriendo un servidor, como por ejemplo es el caso de dispositivos con limitadas capacidades como son los teléfonos móviles.

2.6.3.3. Websockets

Los websockets son una tecnología que permite una comunicación bidireccional entre clientes y servidores, lo que significa que ambas partes se comunican e intercambian datos al mismo tiempo.

Las anteriores tecnologías para APIs asíncronos cumplen la función de proporcionar los datos que el cliente solicita, pero tienen aspectos en los que se hace posible una mejora notoria de sus características. Long Polling daba lugar a peticiones innecesarias y muchas reconexiones. Por otro lado los REST Hooks solucionaban este problema mediante el método PUB/SUB, pero requerían que el

cliente tuviese en todo momento un servidor HTTP ejecutándose, lo que llevaba a un gran consumo de recursos que algunos clientes no podían permitirse.
La utilización de los websockets tiene como objetivo la solución de todos estos problemas aportando una nueva posibilidad de aprovisionamiento de servicios que mejore la experiencia de usuario.

El uso de los websockets permite establecer un canal de comunicación bidireccional y full-dúplex sobre un único socket TCP. Dicho canal permanece abierto permitiendo la conexión entre cliente y servidor websockets para el intercambio simultáneo de información entre ambos.

Los websockets utilizan su propio protocolo, el cual fue estandarizado por la IETF (Internet Engineering Task Force) [RFC 6455] en 2011. Originalmente, estaba diseñado para ser implementado en navegadores y servidores web, aunque su evolución ahora permite implementarlo en cualquier otro tipo de aplicación que siga el modelo cliente/servidor.

A pesar de utilizar su propio protocolo, los websockets se basan en HTTP, esto hace necesaria la actualización de protocolo, es decir, durante el intercambio de información, pasar de la utilización de HTTP a websockets por parte tanto del cliente como del servidor.

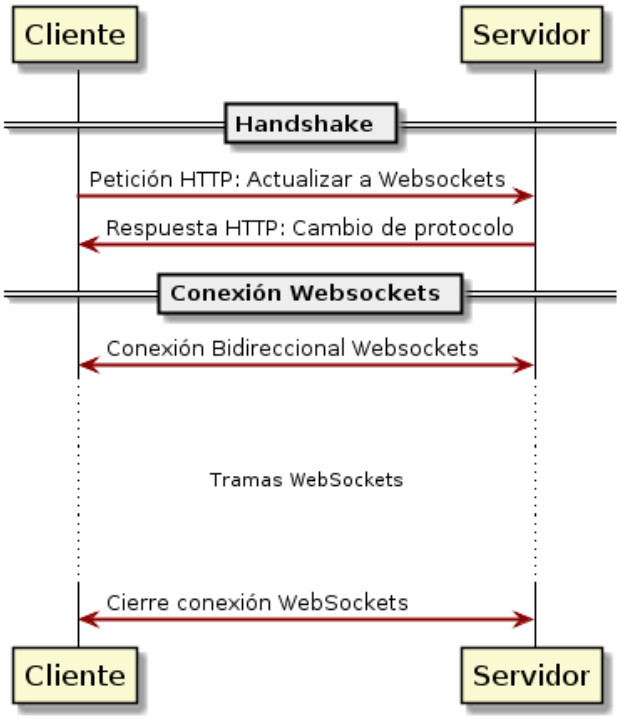

Figura 2.7: Handshake de HTTP a WS

En referencia a los mostrado en la Figura 2.7, se muestra cómo se produce la actualización de protocolo. Partiendo del intercambio de tramas HTTP, el cliente realiza una petición para cambiar el protocolo de HTTP a WS, si el servidor lo soporta se efectuará dicho cambio. Al efectuarse dicha actualización se crea una conexión websockets ful-dúplex bidireccional que permite el envío en tiempo real de mensajes entre cliente y servidor. Todas las tramas que se envíen desde ese momento serán WS. La conexión será cerrada por el cliente si se desconecta o el servidor si se produce una caída, cerrando a su vez todas las conexiones abiertas.

En un entorno asíncrono, como es el caso de SmartSantander, se hacen uso de los websockets basándose en el método PUB/SUB. El cliente tendrá la posibilidad de pedir cierta información al servidor, suscribiéndose a los datos que quiera recibir. El servidor, en vez de mandarle una respuesta inmediatamente indicando si tiene esos datos o no, espera a recibir el evento en el cual contenga esa información requerida por el cliente y cuando esté disponible la envía en una notificación debido a que la conexión websockets inicial con el cliente permanece abierta.

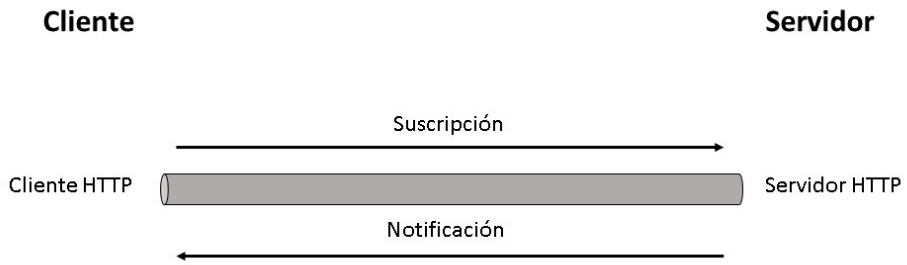

Figura 2.8: Websockets aplicados al modelo PUB/SUB

Tal y como se ve en la Figura 2.8, se crea una conexión entre el cliente y el servidor, la cual permanece abierta para el envío de información tanto del cliente al servidor como viceversa.
Esa conexión abierta permite al cliente mandar la suscripción de los datos que desea recibir al servidor, este espera a recibir esos datos que le llegan mediante un evento y cuando los tenga en su poder se los envía al cliente por la misma conexión que se creó al principio y que sigue abierta.
Con la desconexión del cliente la conexión se cierra también, si ese cliente desease volver a mandar una suscripción a ciertos datos debería abrir una nueva conexión para ese fin.

Mediante el uso de la tecnología websockets, al contrario que con el uso de los REST Hooks, no es necesario tener un servidor HTTP en cliente debido a que para enviar las observaciones que se pedían en la suscripción no hay que crear otra conexión, sino que se usa la conexión websockets ya abierta entre el cliente y servidor para enviar esos datos.

La utilización de websockets puede llevarse a cabo de diferentes maneras. Una de ellas, y la que se implementa en este Trabajo de Fin de Grado, es mediante la utilización de librerías de código que implementan un pequeño protocolo por encima de websockets para facilitar el desarrollo.

En este proyecto se hace uso de la librería Socket.io que, básicamente, es un módulo que permite una comunicación en tiempo real, bidireccional y basada en eventos entre cliente y servidor. Socket.io establece una capa por encima de websockets que será la encargada de negociar el establecimiento de la conexión entre cliente y servidor, si ambos soportan websockets se establecerá una conexión de este tipo, si por el contrario uno de los dos no soporta websockets se opta por la negociación de otro método soportado por ambos, como por ejemplo Long Polling.

2.7. Almacenamiento de información

Los servidores de SmartSantander, al recopilar mucha información de los sensores deben de mantenerla en algún sitio para que se accesible para los clientes. La persistencia de los datos que deben prevalecer en el tiempo, para poder acceder a ellos en un futuro, se garantiza almacenando dichas medidas en bases de datos.

2.7.1. Bases de datos

Una base de datos es un almacén que permite guardar grandes cantidades de información de forma organizada para que luego se pueda encontrar y utilizar fácilmente.

Existen varios tipos de bases de datos, uno de los más conocidos y extendidos son las bases de datos relacionales que le dan uso a un lenguaje de consulta estructurado llamado SQL. Este tipo de base de datos almacenan y organizan la información en tablas y permite establecer interconexiones entre esos datos mediante las cuales se relacionarán los datos de las tablas.

Este tipo de base de datos siguen las reglas llamadas ACID (atomicity, consistency, isolation, durability), con estas propiedades garantizan que un dato sea almacenado de manera inequívoca y con una relación definida sobre una estructura basada generalmente en tablas que contienen filas y columnas.

El gran crecimiento de Internet supuso un auge en el almacenamiento de datos, muchos más usuarios podían almacenar información, lo que supuso un aumento exponencial de la cantidad de información que se subía a las bases de datos.
Para mantener las propiedades ACID con un gran número de datos supondría un reto. La escalabilidad de este tipo de bases de datos es vertical y suele hacerse mediante un hardware más potente, lo que puede suponer fuertes inversiones de capital. Además, ante un número muy elevado de peticiones se podrían generar cuellos de botella, debido a la consistencia y el aislamiento, ralentizando todo el sistema.

Debido a estos inconvenientes es que surgen las llamadas bases de datos no relacionales o bases de datos NosQL (Not Only SQL), las cuales permiten resolver los problemas de escalabilidad y rendimiento que se presentan con los grandes volúmenes de información, mediante nuevos entornos de manejo de datos distribuidos y escalables de forma horizontal. Ofrecen una forma de almacenar y manipular los datos sin necesidad de ser restrictivo como es el caso de SQL, con el objetivo de sacrificar integridad por velocidad.

Las bases de datos NoSQL son sistemas de almacenamiento de información que no cumplen el esquema entidad-relación de las bases de datos relacionales. Para el almacenamiento de datos no utilizan estructuras de datos en forma de tablas sino que usan otro tipo de formatos, según ese formato utilizado serán de un tipo de bases de datos o de otro.

Dentro de las bases de datos NoSQL existen diferentes tipos, entre los cuales destacan:

Base de datos clave-valor Es un tipo de base de datos no relacional almacena datos utilizando un método simple de clave-valor, el almacenamiento de los datos se hace como un conjunto de pares clave-valor en los que una clave sirve como un identificador único.
Más concretamente, el modelo de datos se basa en un tabla hash en donde cada clave está asociada con un solo valor en una colección.

Base de datos documentales Se caracterizan por almacenar la información como documentos, los cuales encapsulan y codifican datos o información siguiendo algún formato estándar, generalmente usaran una estructura simple como JSON o XML, y donde se utiliza una clave única para cada registro.

Base de datos en grafo En estas bases de datos la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se puede hacer uso de la teoría de grafos para recorrerla.

2.7.2. Redis

Una de las más populares bases de datos NoSQL es Redis, la cual se incluye dentro del tipo de clave-valor. La importancia de este tipo de bases de datos le ha llevado a ser utilizada por grandes compañías, tales como Github, para desarrollar tareas que involucran grandes cantidades de datos, pero también para tareas de un alcance menor. Basándose en su prestigiosa fama se puede pensar que sólo es usada por grandes empresas que puedan permitírselo, pero la realidad es que es de código abierto y cualquiera puede hacer uso de ella, de hecho, es muy común su utilización en proyectos, un claro ejemplo de ello es SmartSantander.

Redis (REmote DIctionary Server) es una base de datos de tipo clave-valor en memoria. Es considerado de software libre ya que tiene otorgada la licencia BSD. Soporta estructuras de datos como cadenas de caracteres, hashes, listas, conjuntos de elementos y conjuntos ordenados entre otros.

Uno de sus rasgos característicos es que es una base de datos en memoria, por lo que mantiene el conjunto de datos en memoria RAM. Esto significa que cada vez que se haga una consulta a la base de datos o se actualicen valores, sólo se accederá a la memoria principal, por lo que realizando este tipo de operaciones no se involucraría el disco, esto generará una mayor velocidad y por consiguiente se reducirán los tiempos de respuesta. Además, Redis permite la persistencia en el disco, dependiendo de los requerimientos necesarios.

Redis incluye también ciertas funcionalidades, una de las más importantes es Pub/Sub. Básicamente, esto es un patrón de mensajería que separa las partes que interactúan sobre los mensajes en dos roles, publicadores y suscriptores.

En esta arquitectura de paso de mensajes existen publicadores que envían mensajes ante el suceso de un evento y sin conocer lo que sucederá después con el mensaje. Los mensajes publicados son caracterizados en canales, sin saber los suscriptores que hay a ese canal. Por otro lado están los suscriptores que expresarán interés en uno o más canales, y sólo recibirán mensajes publicados por esas fuentes de interés, sin saber si quiera si hay publicadores. Esta división en publicadores y suscriptores puede permitir mayor escalabilidad y una topología de red más dinámica.

Capítulo 3

Implementación

Una vez tratados los conceptos teóricos necesarios para el desarrollo del proyecto, se continua con el siguiente paso que es la implementación propiamente dicha. En este capítulo, se detalla el sistema que se ha desarrollado a lo largo de este TFG. Se hace especial énfasis en los componentes funcionales que se han implementado, así como aquellos módulos que ha sido necesario incluir para la integración del sistema en la plataforma SmartSantander

3.1. Arquitectura del sistema

Como ya se introdujo en el Capítulo 1, el principal objetivo de este TFG es el desarrollo de un sistema que permita el acceso asíncrono a las medidas producidas por los sensores desplegados en la plataforma SmartSantander.

Para ello se ha diseñado un módulo, denominado *SmS Websockets Manager*, que se integrará con el resto de la plataforma de SmartSantander para proveer dicho acceso asíncrono. El *SmS Websockets Manager* consta de 3 componentes principales, los cuales se utilizan para realizar acciones específicas

En general, incluyendo en la arquitectura de nuestro sistema no sólo el módulo desarrollado sino también los otros componentes con los que interactúa podemos distinguir los siguientes estratos en nuestra arquitectura:

Clientes Son los que harán uso de los servicios proporcionados por SmartSantander. Decidirán la información que desean recibir, y la consumirán cuando les llegue.

SmS WebSockets Manager Es el módulo desarrollado y se ha integrado en la plataforma SmartSantander. Interactuará por un lado con los clientes exportando un interfaz basado en websockets y por otro lado con SmartSantander empleando los interfaces que ya existen en la plataforma.

SmS Son los componentes ya existentes en esta plataforma y con los que es necesario interactuar para el aprovisionamiento del servicio.

Comenzando por el nivel inferior de la arquitectura tenemos los sensores. Son el elemento más básico de la plataforma SmartSantander. En base a las medidas captadas, generan observaciones que reportan hacia los servidores centrales de la plataforma que se encargan de su gestión y almacenamiento.

La plataforma SmartSantander ya dispone de un sistema de acceso asíncrono a la información basado en suscripciones, que permite que los clientes se suscriban a él y cuando se produzca una medida que cumpla con los criterios definidos en esa suscripción, la plataforma de SmartSantander emita una notificación a dicho cliente.

El módulo desarrollado, *SmS WebSockets Manager*, aporta una nueva forma de emitir esas notificaciones. Para ello consta de 3 componentes principales: un servidor HTTP con soporte websockets, un WS Subscription Broker y un listener de Redis.

El servidor HTTP con soporte websockets se encarga de establecer la conexión websockets con los clientes que están solicitando cierta suscripción, esa conexión websockets es un canal bidireccional por el cual es posible mandar información. Las observaciones generadas por los sensores se enviarán a su cliente correspondiente a través de dicha conexión websockets.

La función del WS Subscription Broker es lanzar los comandos al IoT API de SmartSantander, mediante los cuales se gestiona el ciclo de vida de las suscripciones. Esta gestión incluye la creación de suscripciones, así como la modificación y eliminación según las necesidades del usuario.

En la Figura 3.1 se muestra la arquitectura del sistema con todos los componentes que toman parte en ella:

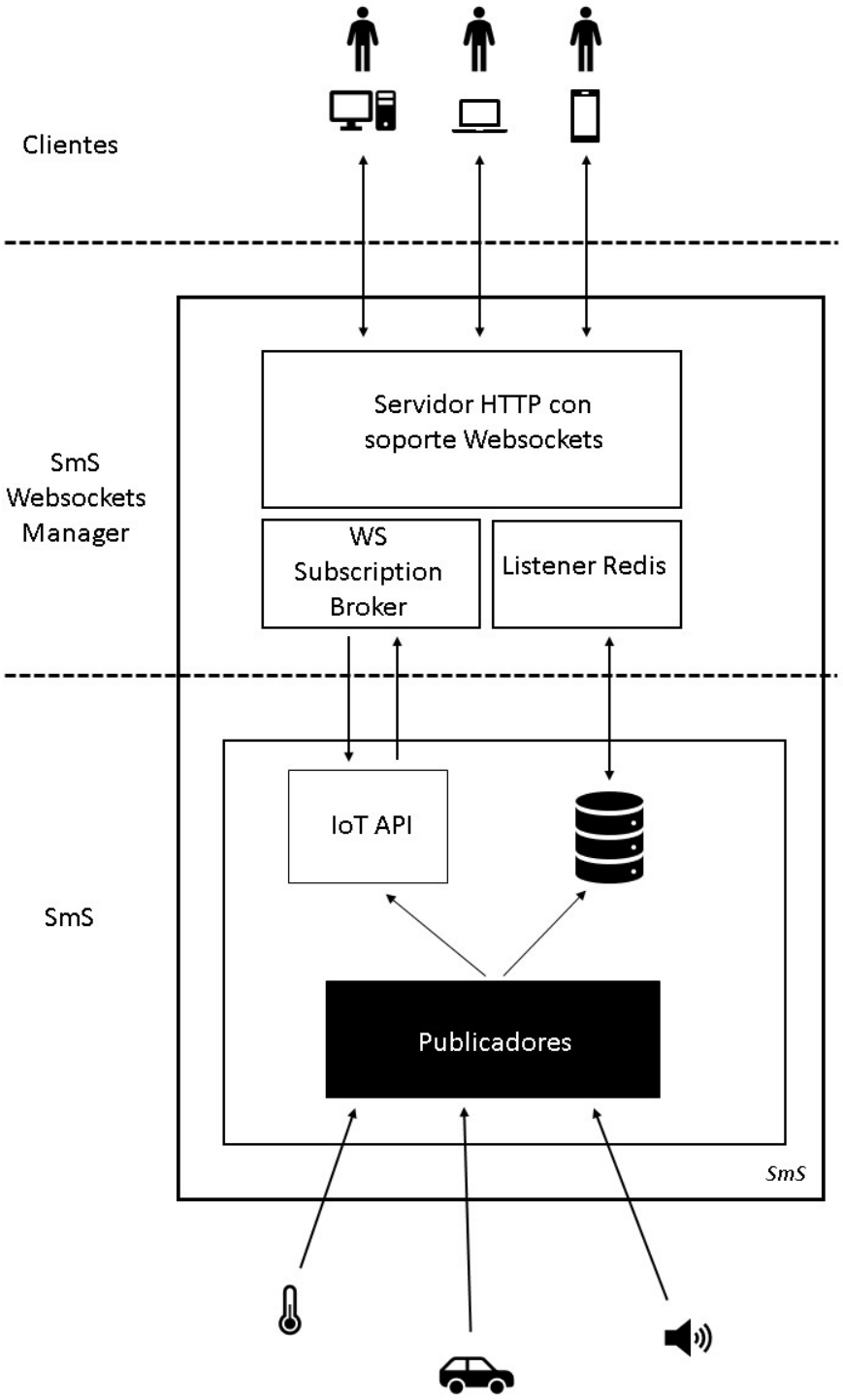

Figura 3.1: Arquitectura del sistema

El listener de Redis es una conexión que está constantemente escuchando las notificaciones que llegan a la base de datos Redis que es dónde se publican las notificaciones de las suscripciones. Cuando llega una notificación, el listener de Redis la recogerá y la redirigirá al cliente que la había solicitado.

Por tanto, los clientes son los que primeramente deciden la suscripción que desean y la solicitan al servidor websockets. El WS Subscription Broker publicará en el IoT API dicha suscripción para hacerla efectiva. Cuando observaciones que resultan de interés a esa suscripción lleguen a SmartSantander, el módulo las recogerá mediante el listener Redis y las redirigirá al cliente, o clientes, que la solicitaron.

3.2. Gestión del ciclo de vida de una suscripción

La idea básica del acceso asíncrono a observaciones de SmartSantander se basa en que los clientes desean ser notificados con cada una de las observaciones que cumplan con los criterios definidos. Para ello, al hacer la suscripción fijarán dichos criterios y cada vez que, de modo asíncrono, se produzca una observación de interés, esta será redirigida a esos clientes que la solicitaron.

Las suscripciones son la base de todo este sistema, gracias a ellas SmartSantander sabe que datos exactamente son requeridos y por qué cliente, o clientes, en concreto.

Las suscripciones pueden encontrarse en diferentes estados, eso sí, nunca podrán estar en dos o más estados al mismo tiempo. Según el sistema definido en SmartSantander, sus suscripciones pueden estar en tres estados diferentes:

Inexistente Se entiende como la no existencia de cierta suscripción. Antes de que un cliente se conecte y decida que desea recibir cierta información, no se habría generado ninguna suscripción por tanto no existirían esas suscripciones.
En realidad esto no sería un estado de la suscripción propiamente dicho, ya que la misma ni tan siquiera estaría creada, pero podría considerarse un estado que representa la ausencia de estado.

Activa Cuando una suscripción está activa implica que es efectiva y por tanto se notificarán al cliente las observaciones correspondientes a dicha suscripción.

Expirada Es el estado en el cual una suscripción está creada, pero no está activa. De este modo, a pesar de existir, al cliente no le llegarán notificaciones de dicha suscripción. Coloquialmente también se le puede hacer referencia como estado inactivo.

3.2.1. Estructura de una suscripción

Para poder hacerse efectivas en el sistema, las suscripciones deben tener un formato determinado, sino no serían válidas y por tanto SmartSantander nunca las aceptaría como correctas y no se almacenarían en la plataforma.

Básicamente, una suscripción es un objeto JSON en el que se distinguen dos elementos fundamentales, *target* y *query*.

La primera parte es el objeto *target*, dicho elemento consta a su vez de dos objetos con información relevante para la notificación de observaciones.
El primero de ellos es el atributo *technology*, el cual se indica la tecnología que se usará para hacer las notificaciones. En este caso es WS, websockets. Ya se comentó que el acceso asíncrono a observaciones de SmartSantander existente podía hacerse utilizando varias tecnologías. Lo que hace el módulo desarrollado es permitir el acceso con una nueva, websockets.
Además de la tecnología, en el *target* aparece el atributo *parameters*. Este campo incluye los detalles necesarios para hacer llegar la notificación al cliente. Para el caso de la tecnología websockets, se incluye el *socketID*, que es el identificador de la conexión websockets de cada cliente.

Por otra parte está el objeto *query*, en el cual consta la información a la que se suscribe el cliente. Esto es, los criterios que deben cumplir las observaciones en las que el cliente está interesado.

Cuando se tiene una suscripción compuesta correctamente, es cuando se puede empezar a interactuar con SmartSantander. El diseño del *SmS WebSocket Manager*, en concreto el componente funcional WS Subscription Broker, permite realizar diferentes operaciones con las suscripciones, de modo que determinado cliente puede crear suscripciones, modificarlas parcialmente o eliminarlas.

La Figura 3.2 muestra un ejemplo de una suscripción:

```
{
    "target": {
        "technology": "ws",
        "parameters": {
            "socketid": socketID
        }
    },
    "query": {
        "what": {
            "format": "observation",
            "_allOf": [
                {
                "phenomenon": "temperature:ambient",
                "filter": {
                    "uom": "degreeCelsius",
                    "value": {
                        "_gt": 5
                    }
                }
                },
                {
                "phenomenon": "relativeHumidity"
                }
            ]
        },
        "where": {
            "_anyOf": [
            {
            "area": {
                "type": "Circle",
                "coordinates": [-3.810011,43.462403],
                "radius": 1.0,
                "properties": {"radius_units": "km"}
            }
            },
            {
            "area": {
                "type": "Circle",
                "coordinates": [-3.799162,43.471705],
                "radius": 500,
                "properties": {"radius_units": "m"}
            }
            }
            ]
        }
    }
}
```

Figura 3.2: Formato de una suscripción

3.2.2. Creación de una suscripción

Es la operativa básica a través de la cual los clientes definen a qué quieren suscribirse, esto es, que medidas y observaciones quieren empezar a recibir.

Para crear correctamente la suscripción, en primer lugar, ha de generarse con el formato adecuado para que el sistema la reconozca, y posteriormente ha de solicitarse mediante una petición HTTP formulada al IoT API de SmartSantander.

Para generar la suscripción el cliente debe mandar una serie de datos para que después, el *SmS WebSocket Manager* pueda componer la suscripción y generarla correctamente. El flujo de envío de información para crear una suscripción, el contenido de dichos mensajes, así como los participantes en dicho flujo, se muestra en la figura 3.3.

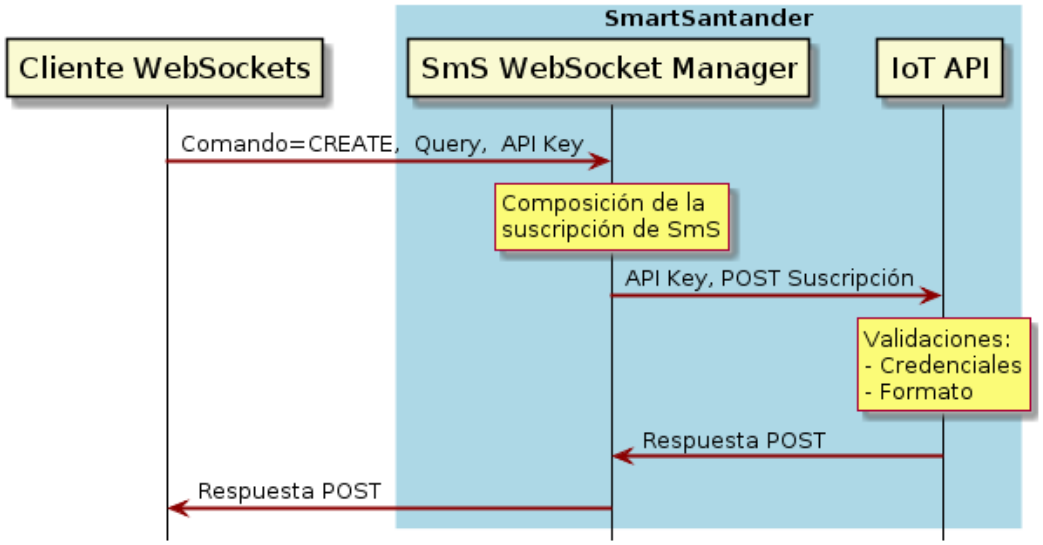

Figura 3.3: Creación de una suscripción

El envío de información comienza con el cliente. Este envía un mensaje con la información que necesita el *SmS WebSocket Manager* para poder generar la suscripción. Lo que primeramente debe indicar el cliente es qué operación es la que desea realizar. Para ello, en el campo *comando* indicará que desea crear una suscripción, fijándolo al valor CREATE. Con esto, el *SmS WebSocket Manager* ya sabrá lo que el cliente desea hacer, y por tanto podrá actuar en consecuencia.

La creación de una suscripción básicamente consiste en que el cliente fija los criterios que deben cumplir las observaciones que desea recibir. Por lo tanto, además de mandar el comando CREATE, es necesario que indique cuales son esos criterios.

Esto es, la *query*. Como ya se indicó en el apartado 3.2.1, la *query* es un objeto JSON, y en los diferentes pares de clave-valor del mismo se van a estructurar los datos específicos de la información que el cliente está requiriendo.

El último de los parámetros que se deben incluir para iniciar la creación de una suscripción es el API Key. Se trata de las credenciales del usuario y deben incluirse para que la plataforma SmartSantander permita completar la operación. Este API Key lo utilizará el *SmS WebSockets Manager* a la hora de utilizar el IoT API de SmartSantander.

Cuando el *SmS WebSocket Manager* reciba el comando CREATE, la *query* y el API Key enviados por el cliente, ya será capaz de formar la suscripción y posteriormente hacer la petición HTTP POST al IoT API.

La composición de la suscripción por parte del *SmS WebSocket Manager* se basa en crear un objeto JSON mediante la unión de otros dos.

El primero de ellos es el *target*. Este elemento lo genera el *SmS WebSocket Manager* de forma que cuando lleguen las observaciones asociadas a la suscripción, poder notificársela al cliente que la solicitó.

Como se ha dicho anteriormente, el atributo *technology* tiene el valor prefijado "WS"que indica que las notificaciones se harán utilizando websockets. Por otra parte, dentro del atributo *parameters* se indica el *socketID* que se fija al valor del identificador de la conexión websocket del cliente. Será, precisamente, el websocket con esté identificador el que se emplee en la notificación.

Para completar la suscripción, se utiliza la *query*. Este elemento es enviado por el cliente junto con el comando CREATE y el API Key

Al final mediante la formación del *target* y la inclusión de la *query*, el *SmS WebSocket Manager* compone la suscripción, teniendo además el API Key enviado por el cliente, ya está listo para realizar el POST al IoT API de SmartSantander.

La creación de la suscripción en la plataforma de SmartSantander simplemente requiere hacer una petición HTTP con el método POST, incluir en el *body* de dicha petición el objeto JSON que se ha compilado con anterioridad y utilizar el API Key como usuario del método de autenticación básica de HTTP.

Cuando el IoT API reciba dicha petición deberá realizar una serie de validaciones para hacerla efectiva. La primera de ellas es validar las credenciales, es decir, el API Key, cuando esto sea correcto significará que el *SmS WebSocket*

Manager se ha logueado correctamente, y por tanto puede realizar peticiones contra el API. La segunda validación es la del formato, es decir, si se le ha enviado una suscripción con formato válido. Aun teniendo una autenticación correcta con el API Key, si la suscripción no sigue las reglas de formato de SmartSantander no será creada.

De esta validación es de lo que dependerá la respuesta del IoT API. Las posibles respuestas ante la petición de creación de una suscripción son:

Código de Respuesta	Descripción
201	Suscripción creada correctamente
400	El cuerpo del mensaje no contiene una suscripción de SmS válida
401	Autenticación incorrecta en el sistema
403	Permiso denegado para realizar operación

Tabla 3.1: Respuestas del API ante creación de suscripción

Como se muestra en la Tabla 3.1, solamente con la devolución de un código de respuesta 201 se crea correctamente una suscripción. En este caso el IoT API manda la respuesta en la cual se certifica que se ha creado correctamente la suscripción, incluyendo además, un campo en el cual indica el *subscriptionID*, que es el identificador único de la suscripción. Las suscripciones se crean inicialmente en estado inactivo, por lo que no se recibirían notificaciones todavía, a no ser que en la petición POST se indique la creación de la misma en estado activo, lo cual supondría que en cuanto se reciban observaciones de esa suscripción serán notificadas.

El resto de códigos de respuesta, no finalizan correctamente la creación de la suscripción, por lo que el *SmS WebSocket Manager* deberá notificar al cliente qué fue lo que falló en base a la respuesta recibida por parte del API.

3.2.3. Listado de suscripciones

La posibilidad de suscribirse a determinado evento es un servicio del que muchos clientes pueden beneficiarse, debido a que el conocimiento de esa información les puede resultar útil para su vida cotidiana. Es lógico pensar que, dependiendo de las circunstancias en las que se encuentren dichos clientes, les serán útiles unos datos u otros. Es por esto que la suscripción a más de un evento es un servicio del que muchos clientes podrían hacer uso.

El servicio de suscripciones proporcionado por SmartSantander permite a los clientes suscribirse a tantos eventos como deseen. Basándose en la necesidad de los clientes de recibir información de utilidad en su ciudad, es lógico pensar que cada uno puede llegar a tener muchas suscripciones activas a la vez. Con el paso del tiempo, la cantidad de suscripciones pertenecientes a un cliente será cada vez mayor, por tanto, sería de gran utilidad para ellos, en un momento determinado, conocer cuantas y cuales son las suscripciones que ellos tienen.

Para cubrir esta necesidad el *SmS WebSocket Manager* soporta la funcionalidad de listar todas las suscripciones de un cliente, si este lo requiere.
Los mensajes necesarios para llevar a cabo esta operación se muestran en la figura 3.4.

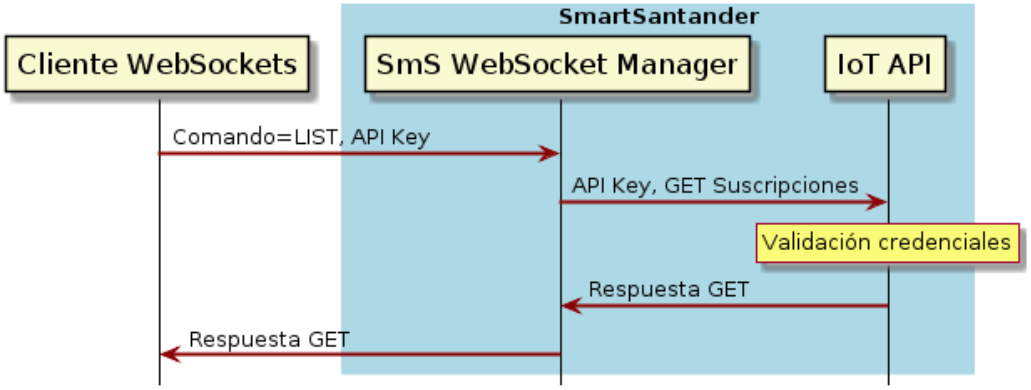

Figura 3.4: Listado de suscripciones

Para listar todas las suscripciones, un cliente solamente debe indicar al *SmS WebSocket Manager* el comando LIST y el API Key para acceder al IoT API de SmartSantander. Las respuestas a este comando por parte del API pueden ser:

Código de Respuesta	Descripción
200	Se muestran las suscripciones del cliente
401	Autenticación incorrecta en el sistema
403	Permiso denegado para realizar operación

Tabla 3.2: Respuestas del API ante listado de suscripciones

Como muestra la Tabla 3.2, la única respuesta del API que aportará todas las suscripciones del cliente es la 200. El resto de respuestas deberán ser devueltas al cliente para que si lo desea solucione dichos impedimentos que no le permiten ejecutar la operación solicitada.

Cuando se realiza la operación LIST, el IoT API se le muestran al cliente todas sus suscripciones, tanto las que están activas como las que están expiradas, incluyendo campos como los datos a los que hacen referencia esas suscripciones y el *subscriptionID* para identificar a cada una de ellas.

Si en ese listado no aparecen suscripciones en determinado estado, significa que el cliente actualmente no dispone de dichas suscripciones en ese estado.

3.2.4. Eliminación de suscripciones

Del mismo modo que un cliente decide suscribirse a cierta información, el mismo cliente puede decidir también que ya no desea recibir esos datos a los que inicialmente se suscribió. Del mismo modo que el *SmS WebSocket Manager* soporta la creación de una suscripción, también permite su eliminación.

Los clientes tienen la posibilidad de crear tantas suscripciones como ellos deseen, por lo tanto a la hora de decidir eliminar determinada suscripción deben saber cual de ellas es la que desean eliminar. Haciendo uso del comando LIST, anteriormente explicado, se mostrarían todas las suscripciones, así como sus respectivos *subscriptionID*.

La Figura 3.5 muestra el intercambio de mensajes para el borrado de suscripciones:

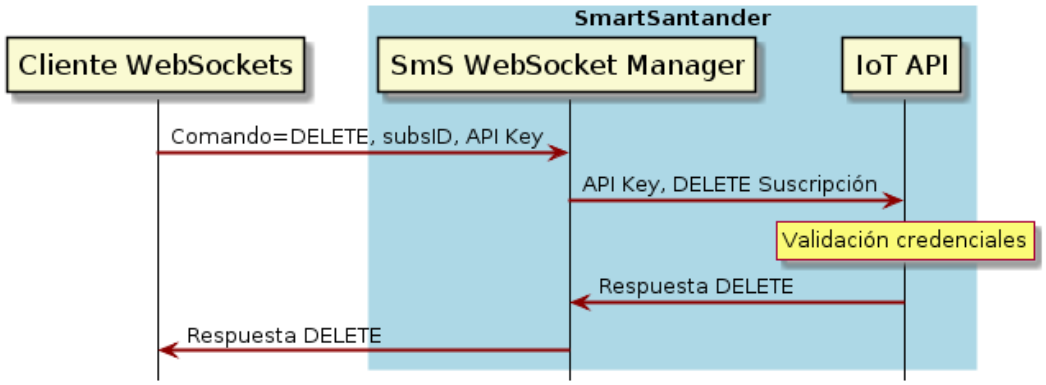

Figura 3.5: Eliminación de una suscripción

El cliente deberá enviar al *SmS WebSocket Manager* los parámetros que sirvan para identificar qué operación quiere hacer, sobre qué suscripción desea hacerla y para validarse en el API.

El comando DELETE indicará que el cliente desea eliminar una suscripción.

Para identificar el recurso a borrar, el cliente envía también el *subscriptionID*, que es un identificador único de la suscripción sobre la que se va a operar y por lo tanto permitirá que mediante el comando DELETE se borre esa, y sólo esa, suscripción.

Una vez que el *SmS WebSocket Manager* tiene los datos necesarios realiza una petición HTTP al IoT API, haciendo uso del método DELETE. Como siempre, se valida con el API Key y se ejecuta la operación si es posible. La petición DELETE puede generar las siguientes respuestas:

Código de Respuesta	Descripción
200	Suscripción eliminada correctamente
401	Autenticación incorrecta en el sistema
403	Permiso denegado para realizar operación
404	Suscripción indicada no existe

Tabla 3.3: Respuestas del API ante eliminación de suscripción

Como se indica en la Tabla 3.3, cuando el código de respuesta es 200, significa que la suscripción fue borrada correctamente. El proceso de eliminar una suscripción conlleva su paso a un estado inexistente, por lo que en el futuro si se trata de buscar dicha suscripción jamás será encontrada y no podrá ser recuperada. Para volver a recibir los datos de esa suscripción la única opción sería volver a suscribirse a la misma información. Esto es, misma *query* pero distinto *subscriptionID*.

En versiones anteriores de la plataforma, este comando sería realmente útil, ya que cada cliente podía tener como máximo cierto número de suscripciones, de modo que si ya cumplía ese cupo no podría realizar más. El borrado permitiría regular la cantidad de suscripciones para que un cliente no alcanzase su máximo. Actualmente el sistema tiene un diseño que permite al cliente tener tantas suscripciones como quiera, por lo que este comando DELETE tiene utilidad únicamente para la gestión adecuada de las suscripciones de un cliente.

3.2.5. Activación y desactivación de suscripciones

Otra funcionalidad que permite el *SmS WebSocket Manager* es la activación y desactivación de las suscripciones ya existentes. Esta característica permite modificar el estado de las suscripciones.

Las suscripciones tienen dos estados básicos, activas y expiradas. La principal característica que los distingue es que cuando una suscripción está activa le llegan

notificaciones al cliente de esos datos, mientras que estando expirada no se recibiría nada.

A través de esta operativa, el cliente tendrá en todo momento el control sobre cuando desea empezar a recibir notificaciones e igualmente cuando deja de interesarle dicho envío.

En el caso de querer desactivar una suscripción, el principal motivo de los clientes es que, temporalmente, no desean recibir información de ella, pero no quieren eliminarla porque puede que en un futuro vuelvan a querer recibir esos datos. Una solución es desactivar esa suscripción, de este modo, seguiría perteneciendo a sus suscripciones, es decir ejecutando el comando LIST se mostraría dicha suscripción, pero no llegarían notificaciones de la misma.

El flujo de mensajes para desactivar una suscripción se muestra en la Figura 3.6.

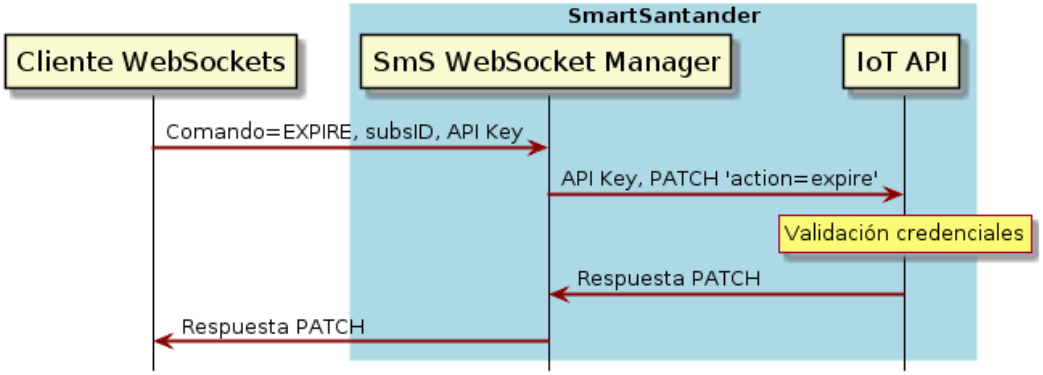

Figura 3.6: Desactivación de una suscripción

Primero el cliente mandará los datos correspondientes al *SmS WebSocket Manager*, en los cuales indicará que desea hacer un comando EXPIRE para desactivar una suscripción, el *subscriptionID* para indicar qué suscripción es la que se quiere desactivar y, como siempre, el API Key para loguearse en el IoT API.
Después el *SmS WebSocket Manager* hará una petición PATCH para aplicar una modificación parcial al recurso, en este caso cambiar el estado de la suscripción a expirada.

Por otro lado, se puede activar una suscripción, es decir, hacerla pasar de estado expirada a activa. Esta funcionalidad cobra especial relevancia si se tiene en cuenta que, tal y como está definido el sistema, las suscripciones una vez creadas caducan a los siete días, es decir, una semana después de ser creadas cambian

automáticamente su estado de activas a inactivas. De modo que, para evitar que una suscripción caduque se puede ejecutar semanalmente esta operación para activar de nuevo la suscripción.

La renovación de una suscripción sigue los siguientes pasos:

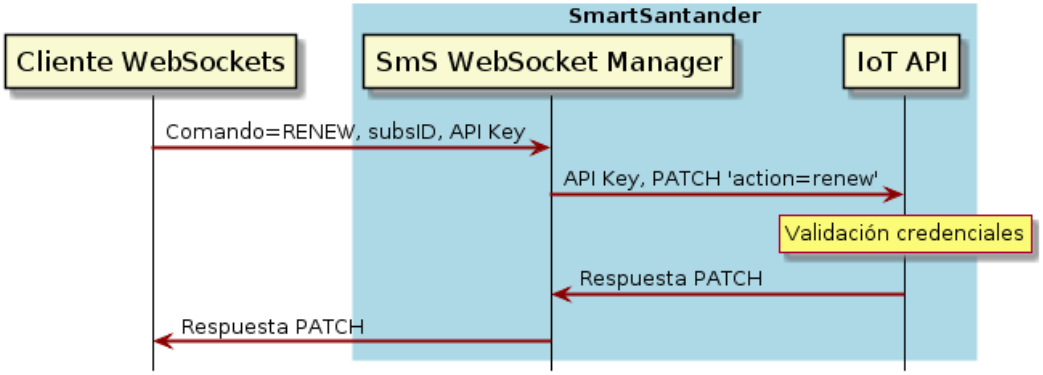

Figura 3.7: Renovación de una suscripción

Como se observa en la Figura 3.7, el procedimiento para renovar una suscripción es prácticamente el mismo que para desactivarla, con la única diferencia que aquí al usar el método PATCH se indica que el estado de la suscripción debe modificarse a activo. El comando empleado en este caso es el RENEW.

Las posibles respuestas a la activación o desactivación de una suscripción son las siguientes:

Código de Respuesta	Descripción
200	Se ha modificado el estado de la suscripción
401	Autenticación incorrecta en el sistema
403	Permiso denegado para realizar operación
404	Suscripción indicada no existe
409	Estado actual de la suscripción entra en conflicto con la petición

Tabla 3.4: Respuestas del API ante modificación del estado de una suscripción

Los posibles errores de la modificación de suscripciones, mostrados en la Tabla 3.4, son iguales a las otras operaciones, a excepción del error con código de respuesta 409, que indica que no se ha podido desactivar una suscripción que ya estaba expirada.

3.3. Notificación de una observación

El propósito final del módulo desarrollado es el envío de observaciones a los clientes que las soliciten. Por tanto, una vez tratada la forma en la que, a través del *SmS WebSockets Manager*, se gestiona el ciclo de vida de las suscripciones, el paso siguiente es extraer las observaciones de SmartSantander y redirigirlas a los clientes interesados en ellas.

3.3.1. Publicación de observaciones en SmartSantander

Los sensores son dispositivos electrónicos encargados de medir ciertas condiciones de la ciudad (aparcamientos desocupados, nivel de ruido, etc) y publicar dichas observaciones en la plataforma SmartSantander para su posterior redirección a los clientes.

Internamente, dentro de la plataforma SmartSantander, la publicación de los datos se hace en una base de datos Redis. La organización de dicha base de datos sigue una estructura de colas, que básicamente, son listas de *strings* ordenadas por orden de inserción. La plataforma SmartSantander aprovisiona gran cantidad de servicios, para los cuales se han desarrollado numerosos módulos que deben de acceder a la Redis anteriormente mencionada para obtener la información allí publicada. Por lo tanto, cada uno de estos servicios accederá a una cola de Redis diferente, en la cual se almacena la información correspondiente a dicho servicio. Esto facilita en gran parte la extracción de información y, por tanto, la eficiencia del sistema.
Concretamente, para el caso de los websockets, las observaciones serán publicadas en la cola *ws* de Redis.

Los sensores, dispositivos encargados de captar información del medio, toman gran cantidad de medidas en periodos cortos de tiempo, debido a la frecuente variación de los fenómenos atmosféricos o humanos. La publicación de todas las medidas tomadas en Redis podría suponer una carga innecesaria del sistema, y más si no hay clientes que quieran aprovecharse de dichas mediciones.

La publicación de observaciones en la plataforma SmartSantander sigue un determinado procedimiento, el cual está diseñado específicamente para no sobrecargar al sistema con publicaciones de información que no sean requeridas por ningún cliente. Las observaciones generadas por los sensores pasan un control

antes de ser publicadas en la Redis, para cerciorarse que hay algún cliente suscrito a dichos datos.

La arquitectura que gestiona la publicación de observaciones en la base de datos Redis se muestra en la Figura 3.8.

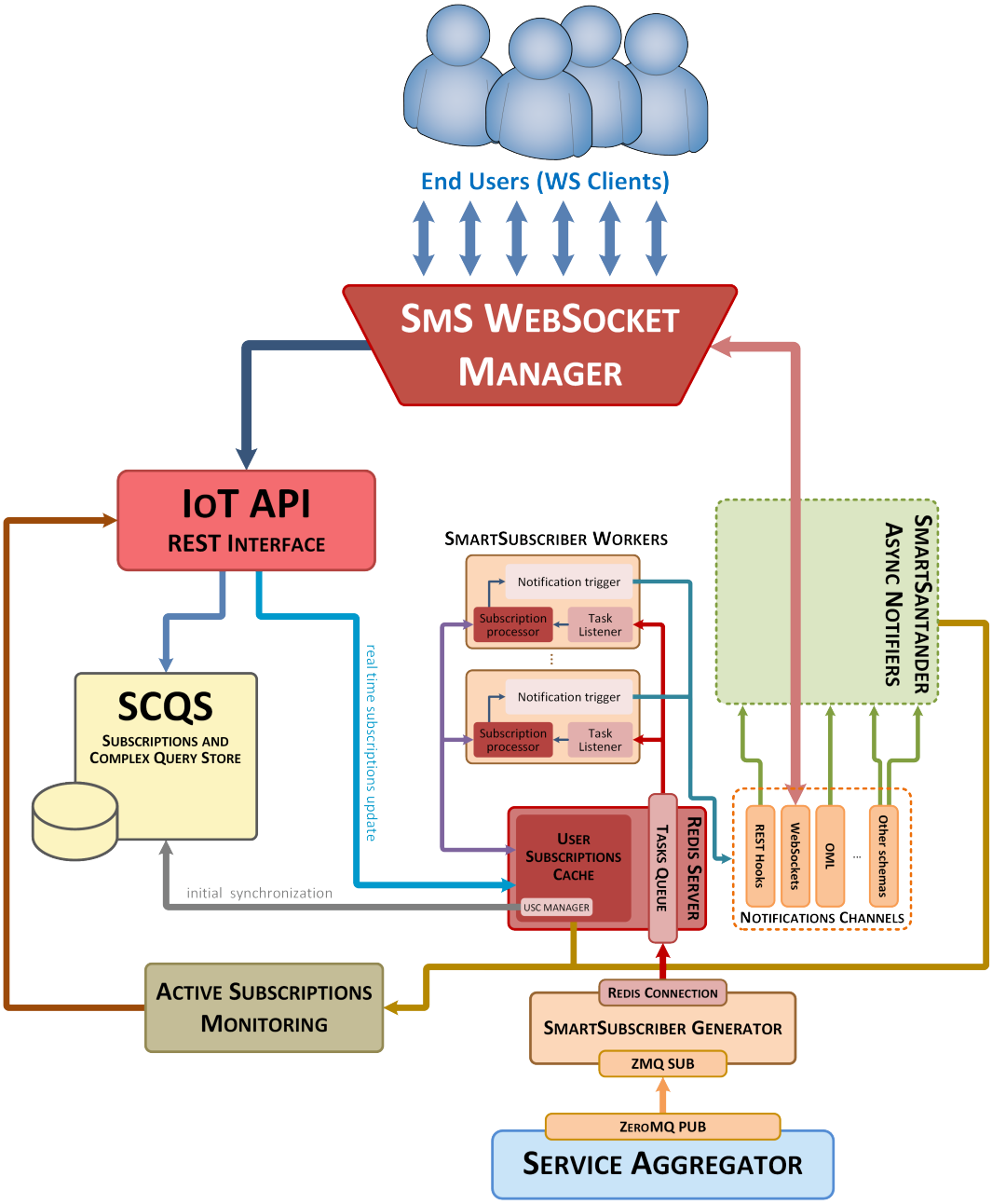

Figura 3.8: Publicación de observaciones en SmartSantander

Uno de los elementos con menos interacción con el *SmS WebSocket Manager*, pero igualmente necesario, es el *Service Aggregator*, que es el componente de SmartSantander donde se reciben todas las observaciones tomadas por los sensores.

Tal como se vio en la Sección 3.2, el *SmS WebSocket Manager* establece con los clientes una conexión persistente y bidireccional, sobre la cual se le envían ciertos comandos para realizar las operaciones con suscripciones. Se ha visto que lo subyace a estas operaciones son peticiones HTTP al IoT API. Dichas peticiones HTTP producirán cambios los cuales deberán registrarse de algún modo en SmartSantander.

La información añadida o modificada del API por el *SmS WebSocket Manager* se verá reflejada principalmente en dos bases de datos, una de ellas para almacenamiento únicamente y otra para interacción con los datos contenidos en la misma.

La primera de esas bases de datos es la referida en la Figura 3.8 como *SCQS*. Es del tipo MongoDB y es utilizada para almacenar todas las suscripciones de los clientes, así como las modificaciones que se hacen sobre ellas.
Por otra parte también se utiliza una base de datos Redis, que a fin de cuentas es una copia de la *SCQS* sobre la cual se harán ciertas comprobaciones antes de publicar ninguna observación. Ésta no es la misma Redis donde se publican las observaciones que debe notificar el *SmS WebSocket Manager*. Por ello, en adelante, para diferenciarlas, denominaremos *Redis Server* a esta base de datos sincronizada con el SCQS y *Redis SmS* a la base de datos de la cual el *SmS WebSocket Manager* recoge las observaciones que debe notificar.

Cuando se hace una petición contra el *IoT API*, los cambios que conlleva se hacen efectivos en la *SCQS*, dejando en ella un registro de las suscripciones del sistema. Automáticamente, se actualiza la caché de *Redis Server* en tiempo real con los mismos datos que en la *SCQS*.
La *Redis Server* tiene además una cola llamada *Task Queue* en la cual se almacenan todas las observaciones tomadas por los sensores y que previamente habían sido enviadas al *Service Aggregator*.

Previo a la publicación de las observaciones en la cola *ws* de la *Redis SmS*, se ha de verificar que dichas observaciones son requeridas por algún cliente websockets. El único escenario en que los clientes reciben observaciones es que tengan suscripciones activas. Por lo tanto, antes de publicar una observación en la *Redis SmS*, se comprueba si dicha observación es solicitada en alguna suscripción activa.

Las suscripciones activas se encuentran en la *Redis Server* debido a la sincronización con la *SCQS*. Las observaciones generadas se encuentran también en la *Redis Server*, concretamente almacenadas en la *Task Queue*.

Los *SmartSuscriber Workers* son módulos pertenecientes a la plataforma SmartSantander y que tienen como función principal acceder a la *Redis Server* y comparar cada observación generada con cada suscripción activa para decidir que observaciones se publican en la *Redis SmS* y cuales no. En el caso de que en la comparación, suscripción activa y observación se correspondan, el *SmartSuscriber Worker* publicará dicha observación en la *Redis SmS*.

Por último, los *SmartSuscriber Workers* acceden al campo *target* de la suscripción para identificar la tecnología utilizada para la creación de dicha suscripción, y así, publicar la observación en la cola de la *Redis SmS* correspondiente. Además, extraen ciertos parámetros como el *subscriptionID* y el *socketID* asociados a la suscripción, y los añaden en una cabecera a la observación antes de publicarla en la *Redis SmS*.
Están desarrollados numerosos *SmartSantander Workers*, de modo que cada uno de ellos tratará una observación publicada en la *Task Queue*, logrando así la publicación en *Redis SmS* de todas las observaciones requeridas.

3.3.2. Envío de notificaciones a los clientes

El despliegue de los miles de sensores por toda la ciudad da como resultado la recopilación de cantidades ingentes de datos, dicha información puede llegar a ser proporcionada como servicio a clientes interesados en ella.

En el apartado 3.3.1, se ha explicado el procedimiento de generación de observaciones, así como el posterior filtrado de las mismas para, finalmente, publicar en SmartSantander solamente los datos estrictamente necesarios.

SmartSantander es una plataforma que presta una gran variedad de servicios, los cuales provee mediante la utilización de tecnologías como por ejemplo REST Hooks, OML, websockets, etc. Para el almacenamiento de dichos datos utiliza la *Redis SmS*. Dentro de esta base de datos, existen colas específicas de cada tecnología para publicar la información relativa a cada servicio.

Para el caso de los websockets, existe una cola dónde se publican las observaciones. Estas se irán insertando en dicha cola y el *SmS WebSocket Manager* irá recogiéndolas para su posterior redirección al cliente que la solicitó. Básicamente, se trata de una cola FIFO, en la que lo primero que entra es a su vez lo primero que sale.

Para el correcto funcionamiento del servicio es necesaria la utilización de tres colas dentro de la *Redis SmS*, las cuales son *ws*, *ws_processing* y *failed*, cada una de ellas con una función determinada.

La cola *ws* es la principal. Es dónde se publican las observaciones que más tarde serán enviadas a los clientes. El *SmS WebSocket Manager* será el encargado de recoger dichos datos.

Para evitar la pérdida de información debida a errores se hace uso de la cola *ws_processing*, la cual se utilizará como un respaldo para salvaguardar los datos hasta que sean enviados al cliente que los solicitó.

Por último, la cola *failed* es utilizada para desactivar una suscripción, en el caso de que el cliente se desconecte y, por tanto, no exista receptor de una determinada observación.

El sistema de notificación de observaciones tiene dos componentes principales, la base de datos donde se almacenan las observaciones, *Redis SmS*, y el *SmS WebSocket Manager* encargado de recoger dichas observaciones y notificarlas a los clientes que las solicitaron.
El componente del *SmS WebSocket Manager* encargado de interactuar con la *Redis SmS* y sus colas es el listener de Redis, anteriormente mencionado en la Sección 3.1.

Dicha interacción, básicamente consiste en la inserción y extracción de elementos en las distintas colas de Redis anteriormente mencionadas. Para operar con varias colas a la vez es probable que sea necesario utilizar ciertos comandos simultáneamente para la obtención o modificación de datos. Es por ello que es necesaria la creación de dos conexiones entre el *SmS WebSocket Manager* y *Redis SmS*.

La primera toma de contacto entre estos dos componentes es iniciada por parte del *SmS WebSocket Manager*. Motivado por la obtención de la información requerida en el cliente, el listener de Redis, utiliza una de las conexiones con *Redis SmS* para recoger observaciones, si las hubiese, de la cola *ws*.
En el caso que dicha cola estuviese vacía, el listener Redis se queda escuchando hasta que alguna observación sea publicada. Si por el contrario, en dicha cola hubiese alguna observación, cogería la primera de ellas y comenzaría a procesarla para posteriormente enviarla al cliente que la solicitó.

El listener Redis emitirá comandos para operar sobre las colas de la *Redis SmS*.

El comando utilizado para recoger observaciones de la cola *ws* de la *Redis SmS* es BRPOPLPUSH. Su funcionalidad consta de tres partes:

B Hace referencia a la naturaleza bloqueante del comando. Es lo que permite quedarse escuchando la cola en el caso de que no hubiese ninguna observación disponible en ella. Al estar escuchando se bloquea una conexión, y por tanto, no podrían utilizarse otros comandos mientras tanto. Este el motivo por el que inicialmente se crearon dos conexiones entre *SmS WebSocket Manager* y *Redis SmS*

RPOP Sacar por la derecha. El objetivo de esta operación es obtener la primera observación que se ha introducido en la cola *ws*, para que el *SmS WebSocket Manager* tenga en su poder dicha observación.

LPUSH Meter por la izquierda. A fin de conservar la observación sacada de la cola *ws* mientras se procesa, es decir, mientras es enviada al cliente, se mete dicha observación en la cola *ws_processing* como copia de seguridad hasta que finaliza su envío al cliente.

Tal como se muestra en la Figura 3.9, el *SmS WebSocket Manager* utiliza una de las conexiones con *Redis SmS* para lanzar el comando BRPOPLPUSH, que se quedará escuchando la cola *ws* hasta que llegue algún mensaje. En el caso de que llegue más de uno, sólo se actuaría sobre el más antiguo y secuencialmente se operaría sobre los restantes.

Cuando se recoge la notificación se lleva a dos sitios distintos. El primero es el *SmS WebSocket Manager*, quien la guarda para poder enviársela al cliente. El segundo es la cola *ws_processing*, en dónde se almacena dicha notificación como medida de seguridad hasta su envío al cliente. A su vez, de la cola *ws* desaparece la notificación tratada, quedando solo las otras observaciones publicadas en dicha cola.

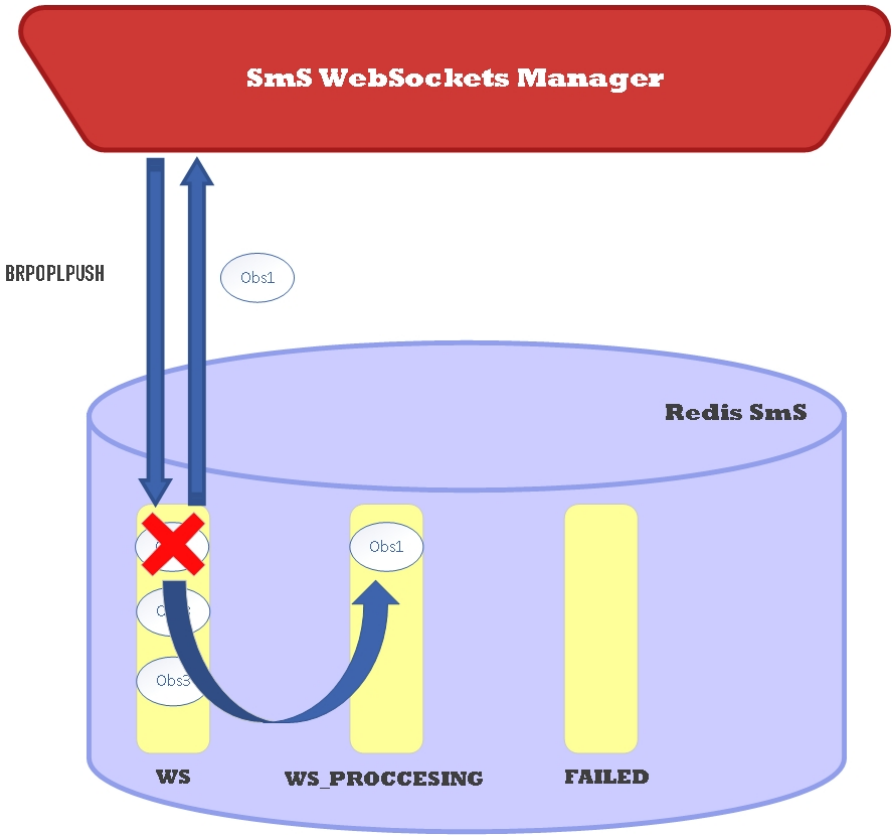

Figura 3.9: BRPOPLPUSH

Una vez llegado a este punto, es necesario que el *SmS WebSocket Manager* envíe la notificación extraída de la cola *ws* a su cliente correspondiente. Para realizar dicha acción es importante recordar que, tal como se comentó en la Sección 3.3.1, las notificaciones que llegan a la cola *ws* constan de dos partes, una de ellas la es observación generada, y la otra es una cabecera la cual incluye entre otros campos el *subscriptionID*, que permite identificar la suscripción, y el *socketID*, que hace referencia al cliente websockets.

El aprovisionamiento de este servicio implica que cada cliente posea una conexión websockets con el *SmS WebSocket Manager*. Éste es el encargado de redirigir las observaciones a su cliente correspondiente. Cada conexión websockets entre un cliente y el *SmS WebSocket Manager* viene identificada por un *socketID*, siendo este un valor único que permite identificar a cada cliente de manera inequívoca.

Al extraer una notificación de la cola *ws*, el *SmS WebSocket Manager* separa la cabecera de la observación propiamente dicha. De este modo, obtiene por un lado la observación en sí, y por otro, el *socketID* asociado al cliente que solicitó

dicha información y, por tanto, sabe a que conexión websockets debe enviar esa observación.

La observación obtenida corresponde a una suscripción previa que determinado cliente realizó, debido a que los fenómenos en la ciudad son aleatorios, es posible que para cuando dicho fenómeno sea medido y publicada su observación, el cliente que la solicitó ya esté desconectado, por tanto no habría a quien mandarle dicha información. Es por lo que, previamente al envío de una observación, se realiza una comprobación para verificar si el receptor de la misma continúa conectado o no.

En el caso de que la observación vaya dirigida a un cliente ya desconectado no se le notificará. Es lógico pensar, que al haberse desconectado, el cliente no será capaz de recibir ninguna de la posibles notificaciones futuras, por tanto, es innecesario la publicación de las observaciones relativas a esa suscripción. En este caso se procede a la desactivación de dicha suscripción. Para ello se utiliza el comando LPUSH para insertar el *subscriptionID* de la suscripción en cuestión, sacado de la cabecera de la notificación, en la cola *failed* y, por medio de un módulo interno de SmartSantander se desactivará la suscripción con dicho identificador. Además, al no necesitar enviar la observación a ningún cliente se debe borrar la copia de seguridad hecha en la cola *ws_processing*, por medio del comando LREM.

Tal como se muestra en la Figura 3.10, al no tener un cliente a quien mandarle la observación, se desactiva la suscripción insertando el *subscriptionID* en la cola *failed*. Esto provoca un cambio de estado de dicha suscripción, de activa a expirada, evitando así que se publiquen observaciones relativas a ella en la *Redis SmS* y, por consiguiente, consiguiendo un ahorro de tráfico de datos innecesario.
Por otro lado se borra de la cola *ws_processing* la observación de respaldo, ya que no se va a procesar por no haber ningún cliente interesado en ella.
Por último, el *SmS WebSocket Manager* vuelve a ejecutar el comando BRPOPLPUSH a la cola *ws* para quedarse escuchando a la llegada de futuras notificaciones o, si ya las hubiera, para el comenzar tratamiento de la próxima.

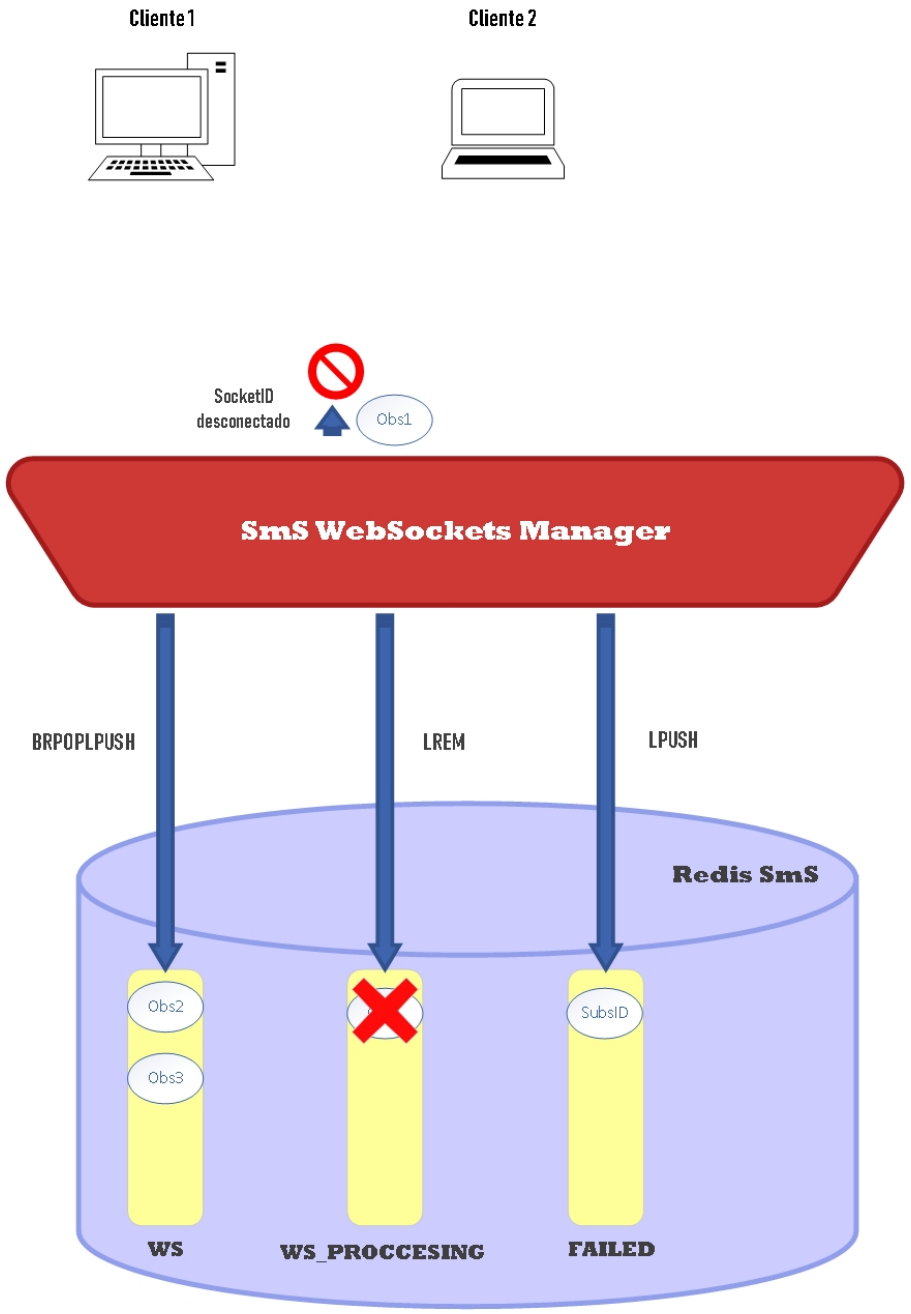

Figura 3.10: Tratamiento de observación de cliente desconectado

Por otra parte, está el caso de que el cliente a quien se debe enviar la observación esté conectado. En esta situación se deberá redirigir la notificación exclusivamente al cliente que la solicitó. Además, una vez enviada la observación, debe ser eliminar de la cola *ws_processing* y volver a escuchar la cola *ws* para tratar el resto de observaciones.

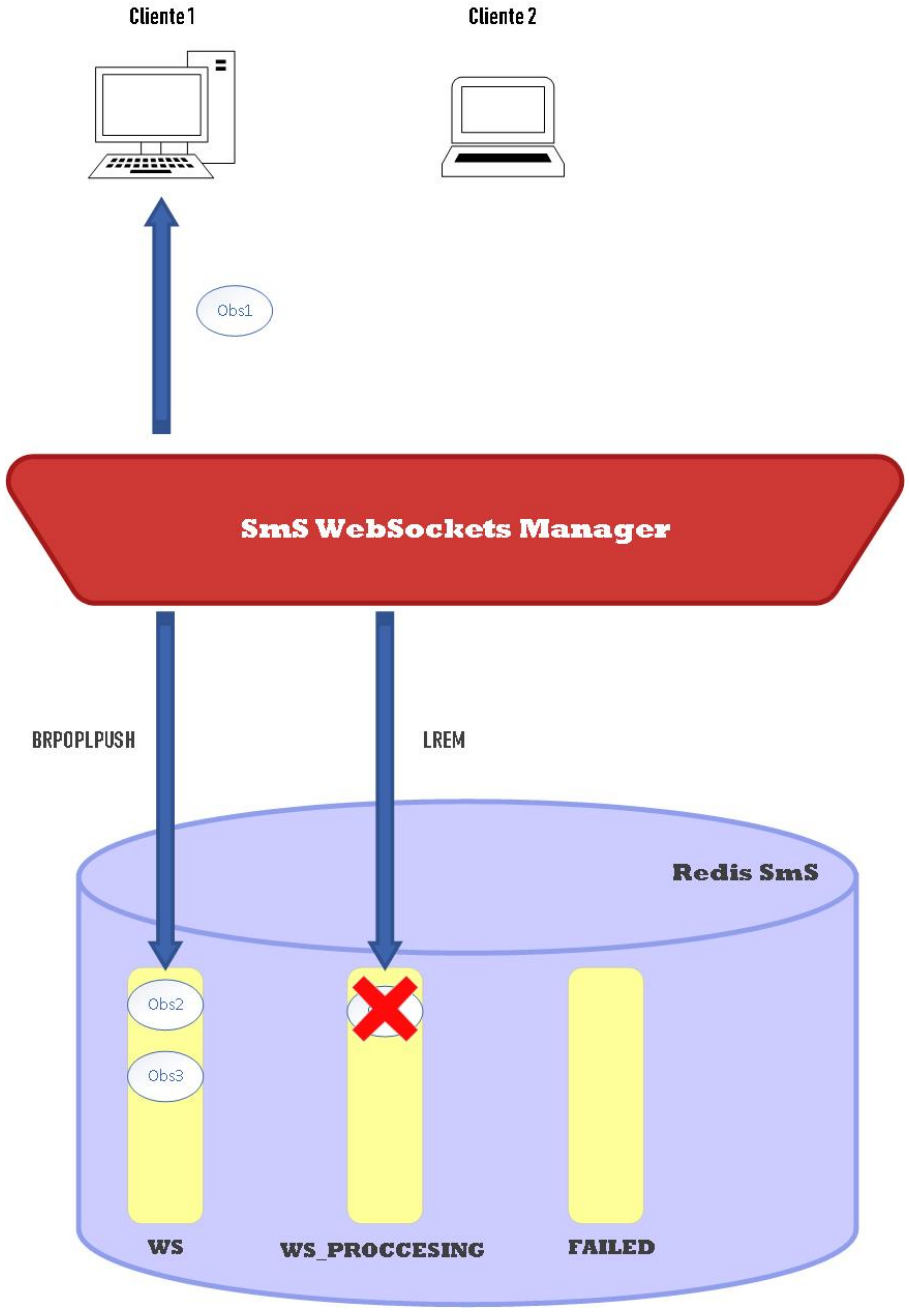

Figura 3.11: Envío de observación a cliente conectado

La Figura 3.11 muestra el proceso de envío de una notificación a un cliente. Tan pronto como el *SmS WebSocket Manager* tiene en su poder la observación extraída de la cola *ws* y verifica que el cliente que la quiere recibir está conectado, se la envía a través de la conexión websockets establecida con el mismo. Después, lanza el comando LREM para borrar de la cola *ws_processing* la observación, debido a que ya se notificó al cliente y no sería necesario guardarla más tiempo. Finalmente, el *SmS WebSocket Manager* vuelve a hacer un BRPOPLPUSH para volver a escuchar

la cola *ws* y recoger la siguiente notificación que llegue o que ya esté en dicha cola.

El método anteriormente descrito es el procedimiento de interacción entre el *SmS WebSocket Manager* y la plataforma SmartSantander. Dependiendo de si los clientes que solicitan información están conectados o no, actúa de una forma o de otra, tratando siempre de proveer el servicio al usuario de la manera más óptima.

Capítulo 4

Conclusiones y líneas futuras

Los websockets son una tecnología que se ha utilizado para el desarrollo de un enlace asíncrono a la plataforma SmartSantander. El presente proyecto proporciona una nueva forma de ofertar los servicios de notificación de observaciones de SmartSantander basada en websockets. En este capítulo se resumen las conclusiones extraídas del desarrollo del trabajo, así como posibles nuevas versiones del mismo.

4.1. Conclusiones

La implantación de la IoT para la mejora de la eficiencia y administración de las ciudades inteligentes supone un avance tecnológico sustancial del cual salen beneficiados tanto las ciudades como sus habitantes y visitantes. Mediante el despliegue y la interconexión de sensores se obtienen parámetros relativos a la ciudad, los cuales son accesibles mediante aplicaciones para los clientes con permisos.

A lo largo del presente documento se han detallado las características del sistema IoT de SmartSantander, así como las tecnologías ya implementadas para el acceso asíncrono a las observaciones generadas. El desarrollo de este Trabajo de Fin de Grado se complementa con dichos accesos ya implementados y crea una nueva forma de acceder a las observaciones de SmartSantander.

En este trabajo se ha realizado la implementación en JavaScript de un módulo cuyo componente principal es un servidor con soporte para la tecnología websockets consiguiendo de este modo una comunicación bidireccional y en tiempo real con los clientes para el acceso asíncrono a los servicios IoT. Para ello se utiliza el modelo PUB/SUB, así como interfaces para la gestión del ciclo de vida de las suscripciones

realizadas.

Se ha realizado la integración del módulo desarrollado con la plataforma SmartSantander. La incorporación de este componente al sistema IoT permite por un lado la creación de suscripciones así como la gestión de su ciclo de vida, y por otro lado la publicación de observaciones ligadas a suscripciones activas y su posterior notificación a los clientes correspondientes.

La generación de un nuevo acceso asíncrono a las observaciones permite a los usuarios la utilización del servicio mediante un nuevo método sin los inconvenientes asociados a los mecanismos que ya existían.. La creación de un nuevo método de acceso complementa el sistema IoT desplegado por SmartSantander dotándolo de nuevas prestaciones orientadas a mejorar de manera considerable la experiencia de los usuarios de la plataforma de SmartSantander.

4.2. Líneas futuras

El desarrollo del proyecto y su integración con SmartSantander pone a disponibilidad de los clientes el uso de websockets como medio principal para la notificación de observaciones medidas por los sensores. Dicho envío de información es un servicio que se proporciona a los clientes, por ello, ante la posible variación de las preferencias de los usuarios siempre se está dispuesto a la implementación de funcionalidades adicionales al sistema desarrollado en este proyecto.

La primera modificación que podría hacerse en un futuro es la forma en que se utilizan los websockets. En este proyecto se utiliza la librería socket.io, la cual es soportada en la mayoría de los lenguajes de programación pero no en todos, por lo tanto una posible variación sería utilizar websockets puros en lugar de esta librería, proporcionando así una implementación pura de websockets y mejorando también los tiempos de latencia.

También, se podría implementar seguridad basada en certificados. De este modo, se pasaría del uso de websockets (WS) a websockets seguros (WSS), tratando así de securizar el entorno de intercambio de información para evitar posibles ataques de terceros.

Por otra parte, y más centrado en la funcionalidad y posible comodidad del usuario, es la implementación de una nueva operación que permita a los usuarios la modificación completa de una suscripción. Hasta ahora estaba implementada la

posibilidad de realizar una modificación parcial de suscripciones, de modo que, se puede cambiar su estado. La nueva funcionalidad sería crear la operación MODIFY que permitiese modificar los datos a los que hace referencia una suscripción la cual ya no se desea, sin necesidad de crear otra nueva suscripción.

El enriquecimiento de funcionalidades de los proyectos se basa en las preferencias y opiniones de los clientes, para nuevas funciones sería conveniente hacer un estudio de aceptabilidad una vez implementada esta primera versión del servicio, y en base a ella y contrastando opiniones de los clientes, incluir posibles adaptaciones sugeridas para la mejora del aprovisionamiento del servicio.

Bibliografía

[1] Kevin Ashton. That 'Internet of Things' Thing. *RFID journal*, 2009.

[2] Hakim Cassimally Adrian McEwen. *Designing the Internet of Things*. 2014.

[3] Leo Turi. Contribution to the Federation of the asynchronous SmartSantander service layer within the European fed4fire context. Trabajo fin de grado, University of Padova, 2015.

[4] SmartSantander. SmartSantander IoT API Documentation. https://api.smartsantander.eu/docs. [Última consulta 20/10/2018].

[5] Javier Vélez Reyes. *Programación Asíncrona en Node JS*. Mayo 2014.

[6] Javier Calzado. JavaScript Asíncrono. *https://lemoncode.net*, Enero 2018.

[7] Carlos Azaustre. Formas de manejar la asincronía en JavaScript. *https://carlosazaustre.es*, Junio 2016.

[8] Async/Await. https://javascript.info/async-await. [Última consulta 05/11/2018].

[9] Lindsay Bassett. *Introduction to JavaScript Object Notation*. Agosto 2015.

[10] NodeJS Documentation, https://nodejs.org/en/docs/. [Última consulta 13/09/2018].

[11] Desarrollo de Aplicaciones Web, https://juanda.gitbooks.io/webapps/content/. [Última consulta 23/09/2018].

[12] Mariano Furriel. WebSockets vs Long Polling. *https://blog.intive-fdv.com.ar/*, Octubre 2016.

[13] Internet Engineering Task Force (IETF). *RFC6455 WebSockets*. Diciembre 2011.

[14] Socket.io Documentation, https://socket.io/docs/. [Última consulta 21/08/2018].

[15] acens. Bases de Datos NoSQL. *https://www.acens.com*, Febrero 2014.

[16] Redis.io Documentation, https://redis.io/documentation/. [Última consulta 17/11/2018].