Desarrollo de una solución para la simulación de entornos IoT

Máster Universitario en Ingeniería del Software, Métodos Formales y Sistemas de Información

Julio Torres Bataller

Director:
Vicente Pelechano Ferragud

Julio 2014

Agradecimientos

Este Trabajo de Fin de Máster ha supuesto un gran esfuerzo y un desafío muy grande para mí. Esta memoria es el resultado de muchas horas de trabajo y dedicación. Gran parte de las tecnologías y conceptos eran nuevos, y el proceso ha supuesto un aprendizaje muy importante para ampliar mis conocimientos.

En primer lugar, me gustaría agradecer especialmente a mis abuelos José y María Elisa toda la ayuda que me han dado a lo largo de estos años, la cual ha sido importantísima para que pudiese alcanzar mis objetivos y metas.

En segundo lugar, agradecer a mi madre y a mi hermana el apoyo que siempre me dan en todo momento. Sin su ayuda, llegar hasta este punto hubiese sido muy difícil ya que me han ayudado a afrontar las complicaciones que han surgido a lo largo del camino y a pasar los momentos difíciles.

Finalmente, me gustaría agradecer a Nacho Mansanet y Vicente Pelechano su ayuda para llevar a cabo este Trabajo de Fin de Máster. Especialmente a Nacho, que fue quien me propuso este trabajo, el cual en un principio resultó difícil, pero que finalmente me ha resultado muy enriquecedor y del cual aprovecharé muchos de los conocimientos adquiridos. Me gustaría agradecerle la dedicación y paciencia que ha tenido a lo largo de estos meses.

Contenido

Listado de figuras

Listado de tablas

Resumen

Este documento refleja todas las tareas que se han llevado a cabo para el desarrollo de una propuesta para la simulación de entornos IoT (*Internet of Things*).

Se ha llevado a cabo un análisis de las tecnologías disponibles en la actualidad para escoger las más adecuadas para cada función, y se han aplicado con el objetivo de conseguir que un simulador tenga asociado una serie de dispositivos, y éstos sean capaces de cambiar su estado.

El resultado obtenido utilizando la arquitectura basada en los conceptos de *Internet of Things*, ha sido un simulador completamente funcional, que es capaz de registrar un conjunto de dispositivos de nuestro entorno, y generar de forma automática datos de entrada para afectar a su estado.

Se ha llevado a cabo un profundo análisis de aplicaciones similares o pertenecientes a otros campos de influencia en el ámbito en el que nos movíamos. Todo ello, mediante un desarrollo basado en el patrón de diseño Modelo-Vista-Controlador, donde la parte de la vista se ha decidido proponer como trabajo futuro y mejora del simulador.

Los beneficios más destacados que ofrece esta propuesta es la posibilidad de simular un entorno en el que se producen una serie de

cambios físicos que afectan a un conjunto de dispositivos capaces de reconocerlos y asumirlos de forma automática. Evidentemente, todo este proceso ofrece una serie de ventajas económicas que evitan unos gastos muy altos cuando se desea llevar a cabo todas estas actividades en el ámbito físico con hardware.

Abstract

This document shows all the tasks realized in order to develop one proposition to simulate environments based on IoT (Internet of Things) concepts.

A depth analysis has been carried out about the available technologies, in order to choose the most proper ones to perform each function. Moreover, those technologies have been applied to develop a complete simulator with several devices associated which are able to change their state.

The final result achieved using the architecture based on IoT concepts, has been a simulator which is completely functional, and able to registrate a group of devices from our environment and automatically generate input data to change regularly their state.

Similar applications from different knowledge fields have been analyzed and an overview has been given to show their features and how they face the challenges proposed. All the development has been performed following a Model-View-Controller design pattern, where the view part has been proposed as a future work or enhancement for the simulator.

The most remarkable benefits that this proposal offers, is the possibility of simulating an environment where several physic changes are produced affecting the devices which are able to recognize and understand them automatically. Evidently, all this process offers several economic advantages to avoid high investments when all these activities are carried out in the physical area with hardware.

1. Introducción

1.1 Descripción

En los últimos años el número de dispositivos electrónicos que nos rodea en nuestra vida cotidiana ha crecido de una manera significativa. Además, no solamente ha aumentado su número, sino que también ha cambiado la forma en que los utilizamos.

El perfil de los usuarios que utiliza estos dispositivos ha cambiado, y en gran parte es debido a que éstos ahora son móviles y los podemos llevar con nosotros a todas partes. Podemos consultar todo tipo de datos y tenemos acceso a cualquier información que nos resulte interesante.

El concepto que existía hasta hace poco sobre la forma de interactuar con los dispositivos que nos rodean requería de un usuario con un rol activo, donde era él quien debía de iniciar la acción para obtener los servicios que quería. Desde finales de los años 80 [1], se está trabajando en una nueva forma de interactuar con estos dispositivos, aunque es cierto que es desde hace unos pocos años, en los que hemos empezado a ver los frutos de tanta investigación.

El factor que ha influido de forma determinante en esta nueva forma de interacción, es la conectividad que hoy en día viene asociada con cualquier dispositivo desde *smartphones*, televisiones o aparatos de música que podamos encontrar en nuestro hogar haciendo que éstos dejen de actuar de forma aislada y pasen a formar parte de un grupo [2].

Una vez dicho esto, ¿en qué consiste esta nueva forma de utilizar los aparatos que nos rodean? Básicamente consiste en hacer que todo lo que nos rodea sirva como datos de entrada a los dispositivos que tenemos cerca para adecuar su comportamiento en función de la situación y el momento. A esto se le conoce como "entorno", y los dispositivos que están atentos a cambios físicos a su alrededor, siendo capaces de cuantificarlos y procesarlos para adaptar su comportamiento de forma adecuada, se les conoce como **"*context awareness*"**.

De esta forma, se crean las bases para el concepto de lo que se conoce como **"*Internet of Things*"**, donde todos los dispositivos que nos rodean se comunican e intercambian información entre ellos para adaptar su funcionamiento dependiendo de nuestras necesidades, procesando los datos que obtienen. De esta forma, pasan a ser tratados más como servicios que como dispositivos.

Haciendo uso de la tecnología que nos rodea de esta forma, el usuario no estaría siempre obligado a tener un rol activo ya que los dispositivos no estarían esperando a un evento o señal proveniente de ellos, sino que pasaría a tener un rol más pasivo donde los dispositivos del entorno serían capaces de actuar activando, desactivando o regulando los servicios que nos rodean.

Los dispositivos pasarían a tener la capacidad de obtener información mediante sensores que se encargan de medir magnitudes físicas como la temperatura, la luminosidad, el sonido que está a nuestro alrededor o nuestra ubicación.

1.2 Motivación

Una vez establecidas las bases de lo que se entiende por entorno y cómo los dispositivos que nos rodean han cambiado en los últimos años, debemos de pensar cómo agrupar todas estas nuevas funcionalidades y cómo gestionar toda esta cantidad de información que recogen.

Como se indica en el anterior apartado, cada vez nos rodean más dispositivos. Muchas de las funcionalidades que ofrecen ni siquiera las utilizamos debido a que están distribuidos por toda nuestra casa, vecindario o ciudad. En este contexto y estrechamente relacionado con las bases de *Internet of Things*, surgen nuevos conceptos como "Cuidad Inteligente" (*SmartCity*) o "Mundo Inteligente" (*SmartWorld*).

La aplicación de las bases de *Internet of Things* a gran escala en nuestra vida cotidiana supondría cambios muy importantes de tipo social y económico. Estos cambios afectarían primero a entornos reducidos como nuestras propias casas para ir creciendo y llegar a un punto en que todos los objetos del mundo fuesen capaces de conectarse entre sí.

Definimos "Ciudad Inteligente" como aquella ciudad que usa las tecnologías de la información y las comunicaciones para hacer que tanto su infraestructura crítica, como sus componentes y servicios públicos ofrecidos sean más interactivos, eficientes y los ciudadanos puedan ser más conscientes de ellos. Es una ciudad comprometida con su entorno, tanto desde el punto de vista medioambiental como en lo relativo a los elementos culturales e históricos [3].

Como usuarios, todavía seguimos teniendo el papel de actuar sobre los dispositivos y pedir o consultar aquello que necesitamos. Además, se genera el problema de que muchos dispositivos realizan la misma función como por ejemplo, medir la temperatura, abrir una puerta, encender una luz o medir la luminosidad en una zona concreta. Ya no nos sirve con tener un dispositivo de cada tipo, en los hogares cada vez tienen más dispositivos en distintos puntos.

Para poder utilizar de forma eficaz el creciente número de dispositivos que nos rodean, sería muy interesante tener alguna manera de poder gestionarlos fácilmente.

Para un empleado que trabaje en una nave que ocupe una gran superficie, conocer la temperatura de la habitación que se encuentra a 2 plantas de distancia o poder apagar la luz de una habitación que se encuentra en otro edificio, facilitaría el trabajo, ya que además de poder acceder a un listado completo de los dispositivos que hay disponibles y cuales son sus estados, podríamos utilizarlos más cómodamente.

Un sistema capaz de obtener información acerca de todos los dispositivos que nos rodean, sería de gran utilidad. Especialmente en la gestión de grandes edificios o superficies, haciendo que los edificios inteligentes dejen de ser una pesadilla de gestionar, debido a que se puede obtener toda esta información de una manera rápida y sencilla de una forma centralizada. Conocer en un determinado momento el listado de dispositivos capaces de medir la temperatura en un determinado edificio o cuáles son los detalles de un dispositivo en concreto además de poder consultar su estado, permitiría tener las fuentes de información de una manera ordenada y rápida. Si además logramos añadir la capacidad de poder actuar sobre éstos dispositivos a distancia, el avance sería muy importante.

La utilización centralizada no haría más que aprovechar la conectividad que ofrecen los dispositivos a través de servicios web disponibles para que el usuario los utilice.

La creación y desarrollo de aplicaciones capaces de gestionar todos estos entornos suponía el tener acceso a todos los dispositivos físicamente y tener la capacidad de crear estas infraestructuras o arquitecturas. Esto, en la mayoría de las ocasiones supone algo muy difícil e inviable, ya no tanto por el presupuesto que se manejaría, sino porque la gestión y el acceso de todos estos recursos de forma física supondría un gran esfuerzo normalmente muy difícil de asumir.

Además, la creciente necesidad de diseño de este tipo de sistemas no ha hecho más que evidenciar las dificultades que existen para implementarlos evitando grandes gastos.

En este punto es donde el trabajo de los diseñadores y desarrolladores se vuelve complicado. Aparece la necesidad de tener herramientas capaces de simular todos estos elementos y escenarios, para poder realizar la mayor cantidad de pruebas posibles en diferentes entornos antes de dar el salto al mundo físico con el hardware.

Con el objetivo de resolver este problema, se plantea este trabajo fin de máster. La idea consiste en diseñar, desarrollar y probar una herramienta capaz de gestionar un conjunto de dispositivos proporcionados por el usuario, que permitiría realizar todas las tareas englobadas dentro del diseño y desarrollo de un sistema *Internet of Things*, pero sin los gastos asociados al proceso convencional en el mundo físico.

Esta herramienta ofrecerá la posibilidad a los diseñadores y desarrolladores de definir los dispositivos con los que desean trabajar, tanto en sus características como en el número. Posteriormente, podrán simular estos dispositivos e interactuar con ellos de forma remota a través de una librería REST en un servidor local.

Todo este proceso que parece muy complicado, se verá facilitado en gran medida por la herramienta, debido a que los ficheros necesarios para llevar a cabo estas tareas se generarán automáticamente utilizando la información proporcionada por los diseñadores a partir de los requisitos de su sistema.

De esta forma, simular las variables del entorno y ver como responde el sistema en tiempo real, podrá llevarse a cabo trabajando con software y evitando los costes de las pruebas realizadas en entornos físicos.

Para afrontar todas estas limitaciones de medios con las que nos encontramos, el simulador desarrollado ofrece la gran ventaja de permitir al usuario especificar solamente aquellos dispositivos que vaya a necesitar y los eventos que provocan un cambio de estado. En ellos se simularán mediante una función encargada de actualizar el estado y encargarse de proporcionar información de entrada de forma continua, a cada dispositivo. De esta forma, se ofrece un medio sin coste para la realización de pruebas, generando diferentes entornos con los dispositivos especificados.

Los dispositivos serán registrados y dados de alta en el simulador que procesará los detalles y los atributos que definen los diseñadores para cada uno de ellos a través de un fichero de entrada. Este fichero será generado a partir de un servicio de definición de dispositivos, que pondrá a disposición del usuario los tipos de dispositivos con todas sus características para que el usuario los configure, en función de sus necesidades mediante una interfaz sencilla e intuitiva.

Los dispositivos aceptados por el simulador se englobarán en dos grandes grupos dependiendo de la funcionalidad que presentan: actuadores y sensores.

Los actuadores serán aquellos dispositivos que esperan un evento externo para cambiar de estado, mientras que los sensores serán aquellos que se encargan de sensar diferentes magnitudes que se encuentran en el contexto para ofrecer esta información cada vez que sean consultados a través de servicios web.

Finalmente, una vez registrados todos los dispositivos, su funcionalidad se pondrá a disposición de los usuarios mediante una serie de métodos publicados utilizando un patrón REST programado mediante el framework Flask.

Todos estos elementos, nos llevarán a simular la base de una solución *Internet of Things*, los cuales son los dispositivos y que facilitará el desarrollo de aplicaciones dedicadas a estos entornos.

Este Trabajo de Fin de Máster se encuentra en un punto de intersección entre varios campos de investigación. Todos ellos, están relacionados por el objetivo de introducir y adaptar las nuevas tecnologías a nuestra vida cotidiana de la forma más transparente posible.

Por un lado, tenemos los conceptos en los que está basado *Internet of Things*, los cuales ya hemos explicado anteriormente a los largo de este capítulo. Por otro lado, tenemos el campo de la domótica, donde se busca adaptar los dispositivos tecnológicos que tenemos actualmente a nuestros hogares, para ayudarnos a realizar tareas cotidianas que repetimos todos los días. Por último, tenemos el concepto de dispositivos "*context-awareness*" [4], cuya definición ya hemos dado y que básicamente consiste en conseguir que los dispositivos que nos rodean sean capaces de

sensar nuestro entorno físico y adaptarse de forma continua y automática.

Podemos afirmar, que nuestra propuesta recoge ideas de todos estos campos y los une para dar lugar a nuestro TFM.

La siguiente figura, muestra gráficamente como los 3 campos mencionados convergen en un punto que da lugar al espacio en el que se encuentra este trabajo:

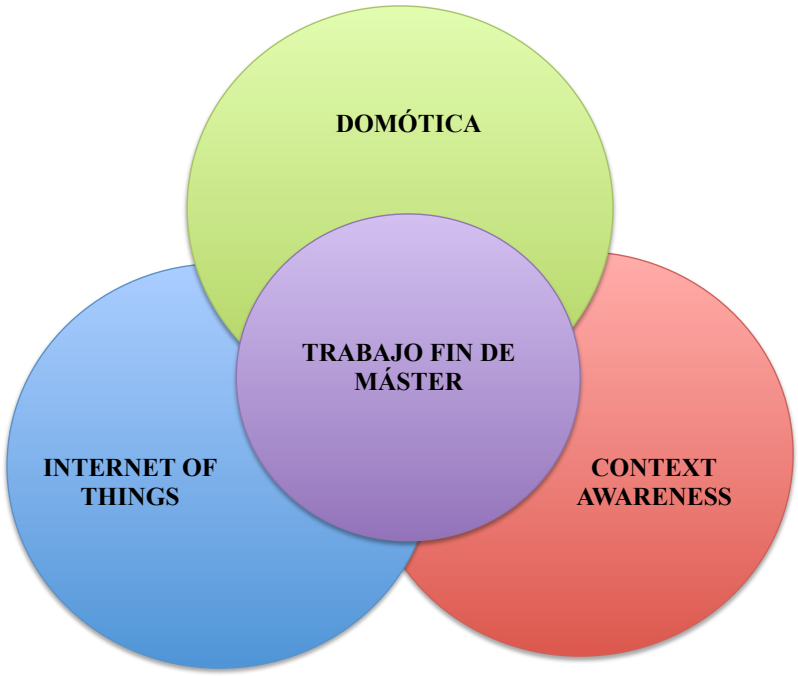

Figura 1: Contexto trabajo Fin de Máster

Una vez expuestas estás ideas, queda claro qué ofrece este TFM, cuáles son las ventajas y posibilidades que ofrece y a quiénes va destinada la herramienta que proponemos.

1.3 Objetivos Globales

Una vez explicado el contexto y la motivación que ha llevado al desarrollo de este simulador, se deben establecer una serie de requisitos que se buscan cumplir una vez terminada la etapa de desarrollo. A continuación se detallan los requisitos principales que debe de cumplir el Trabajo de Fin de Máster (*TFM*):

Fiabilidad: Dado que en el proceso se generan diversos ficheros de forma automática, que servirán de base para luego la ejecución del servidor y el registro de los recursos web, es importante que no se produzcan errores durante la ejecución del simulador que puedan afectar al proceso de generación de estos ficheros, y por consiguiente al funcionamiento del servidor. La forma en que está planteada la solución, implica la ejecución en serie de diversos procesos que dependen uno del otro como se explicará en más detalle en posteriores secciones.

Escalabilidad: el número de dispositivos con los que trabaja el simulador puede variar, por lo tanto, la aplicación debe de estar preparada para asumir un número de dispositivos dentro de un rango amplio.

Utilización de tecnologías estándar: las herramientas utilizadas para el desarrollo del simulador deberán ser abiertas. Esto facilitará la no dependencia de ninguna plataforma en concreto y la facilidad de obtener información sobre el uso y mejoras que puedan afectar a nuestra aplicación.

Este factor afecta desde la selección del lenguaje de programación a utilizar (Python, PHP, Perl..) hasta el tipo de base de datos a utilizar (MySQL).

Lo que conseguimos alcanzando este objetivo, es que la aplicación sea más fácil de mantener y que futuros desarrolladores puedan participar en ella.

En este caso, la elección de Python como lenguaje de programación ha sido enfocada al cumplimiento de este punto, debido a que es un lenguaje de programación de alto nivel cuya sintaxis permite leer y entender con facilidad las tareas que lleva a cabo el programa. Además,

Python ofrece las ventajas que se obtienen de la programación orientada a objetos como puede ser JAVA, lo que supone que la curva de aprendizaje sea asumible para un proyecto de estas características.

1.4 Objetivos Trabajo Fin de Máster

Una vez realizada una introducción y explicados los objetivos globales, se detallarán los objetivos concretos que busca la realización de este Trabajo de Fin de Máster.

Desarrollo de un simulador de dispositivos: se busca el desarrollo de una aplicación en Python que se encargue de procesar los parámetros de los dispositivos registrados mediante una interfaz web y que sea capaz de crear los objetos necesarios en memoria, con todos los atributos especificados.

Generación recursos web asociados: mediante la utilización del microframework Flask desarrollado en Python, se generarán los recursos asociados a cada dispositivo para que estos estén disponibles a través de la web y sea posible interactuar con ellos utilizando los métodos que cada dispositivo tenga asociado.

Definición de los canales de comunicación: una vez los dispositivos estén registrados y disponibles a través de la web, se debe de definir cómo se intercambia la información con ellos, ya que habrá unos dispositivos que simplemente devolverán su estado y habrá otros que permitirán realizar acciones sobre ellos para modificar dicho estado. Para ello, se ha utilizado JSON que es un protocolo de intercambio de información ligero que permite especificar cómo será el formato de estos mensajes.

Simulación de cambios en el estado de los dispositivos: la parte final de la aplicación consistirá en simular los cambios que se producen en el estado de los dispositivos y que producen un cambio en el contexto.

1.5 Estructura del documento

El contenido de este documento se organiza en siete secciones cuyo contenido se detalla a continuación:

Capítulo 2: Aplicaciones Relacionadas

Se detallarán varias aplicaciones que están relacionadas con el tema principal del Trabajo de Fin Máster como es el concepto de Computación ubicua e *Internet of Things*. Posteriormente, se mostrará una breve comparación para ver de qué manera cada una de las aplicaciones afronta los problemas que se plantean.

Capítulo 3: Tecnologías involucradas

En este capítulo se explicarán cuáles son las tecnologías utilizadas en el desarrollo del simulador y las razones por las cuales éstas han sido escogidas. Se detallará por qué Python ha sido escogido como lenguaje de programación y cuáles han sido las librerías utilizadas para resolver los problemas encontrados. También se explicará para qué y cómo se utiliza el microframework Flask para la creación de los recursos web de los dispositivos.

Capítulo 4: Solución propuesta

En este capítulo se dará una visión global de cuál es la solución propuesta en este TFM. Se explicará mediante un diagrama cuál es el flujo principal de la aplicación y sus componentes.

Capítulo 5: Desarrollo del simulador

Este capítulo contendrá una extensa explicación técnica de cómo se ha desarrollado el simulador empleando las tecnologías detalladas en el capítulo 3 y cómo se ha aprovechado las ventajas que éstas ofrecen sobre otras tecnologías existentes.

- Desarrollo del Parser para XML
- Desarrollo de las clases involucradas en el Simulador
- Generación de los recursos REST asociados a cada dispositivo

Capítulo 6: Caso de Uso

Se plantea un escenario donde se utiliza la aplicación para explicar su funcionalidad y las utilidades que ofrece. Se detallará un ejemplo práctico en el cual se puede aplicar el uso del simulador para obtener ventajas claras en la realización de tareas pertenecientes a un modelo de negocio concreto.

Capítulo 7: Conclusiones y trabajo futuro

Después de la realización del proyecto, se expondrán las conclusiones obtenidas y cuáles son las mejoras que podrían introducirse en trabajos futuros o puntos que podrían ser mejorados.

1.6 Conclusiones

En este capítulo se ha presentado la temática general de este TFM y cuáles han sido los conceptos e ideas que han influido y motivado su desarrollo. Se ha destacado la importancia de las bases de *Internet of Things* y se ha realizado una pequeña descripción de cuáles son los objetivos más importantes que se busca abarcar con el trabajo. Además se ha realizado una presentación del documento y los puntos que contiene.

En el próximo capítulo, veremos ejemplos de aplicaciones destacadas representativas de los campos de conocimiento expuestos en este primer capítulo. Se mostrarán sus características principales y una visión general de cómo afrontan las tareas que realizan.

2. Aplicaciones relacionadas

En este capítulo se realizará una breve introducción de algunas de las aplicaciones que actualmente ofrece el mercado relacionado con la temática principal de este Trabajo de Fin de Master. Se han buscado ejemplos de aplicaciones que pertenecen a los diferentes campos que influencian este TFM para dar una visión global de las funcionales y características que ofrecen.

2.1 Belkin WeMo Switch

WeMo Switch de la compañía Belkin es un accesorio que proporciona a los elementos del hogar la conectividad necesaria mediante un repetidor wi-fi que permite al usuario actuar sobre estos dispositivos para encenderlos o apagarlos y consultar su estado.

Además, el usuario puede realizar todas estas acciones de forma remota a través de una intuitiva interfaz disponible a través de dispositivos móviles como *smartphones* o *tablets*.

El usuario puede configurar un calendario para programar los dispositivos y que éstos se enciendan o se apaguen según sus necesidades.

Figura 2: Belkin MeWo Switch interfaz

Como se puede observar en la Figura 1 obtenida de *http://www.belkin.com*, cualquier dispositivo del hogar (lámparas, planchas, ventiladores…) puede ser añadido a la lista de dispositivos que deseamos gestionar. Todos estos dispositivos aparecerán listados en la interfaz para que el usuario pueda actuar sobre ellos.

Cierto es que para llevar a cabo los conceptos sobre los que se basa la idea del *Internet of Things,* es necesario que casi todos los dispositivos ofrezcan la posibilidad de poder conectarse a internet. Actualmente no estamos en ese punto, ya que aunque muchos dispositivos sí lo ofrecen, quedan muchos electrodomésticos que no ofrecen esta posibilidad y por lo tanto quedarían fuera de poder ofrecer estas funcionalidades.

Por lo tanto, la solución ofrecida por Belkin puede resultar muy interesante ya que ofrece la parte de la solución que entraría dentro del

hardware y algunas de las opciones dentro del software ofrecidas por el simulador desarrollado en este Trabajo de Fin de Máster, aunque este último ofrece una variedad de dispositivos disponibles mucho más grande y con características asociadas a éstos más detallada.

Tanto nuestra herramienta como el Belkin MeWo Switch serían dos soluciones que ofrecerían un alto nivel de escalabilidad ya que se podrían gestionar tantos dispositivos como se necesite.

2.2 Siafu

Siafu es un simulador de contexto implementado en JAVA. Es una herramienta *open source* en la que se puede modelar un escenario completo con los agentes implicados y en el que se pueden generar cambios de contexto [5].

Siafu permite trabajar a partir de un escenario ya creado previamente o generar uno proprio desde cero. La herramienta contempla un amplio rango de escalabilidad, lo que permite trabajar con escenarios pequeños en los que el número de dispositivos es bajo, aunque está especialmente enfocado a grandes escenarios como ciudades.

En el siguiente diagrama, se presenta la arquitectura de la herramienta Saifu:

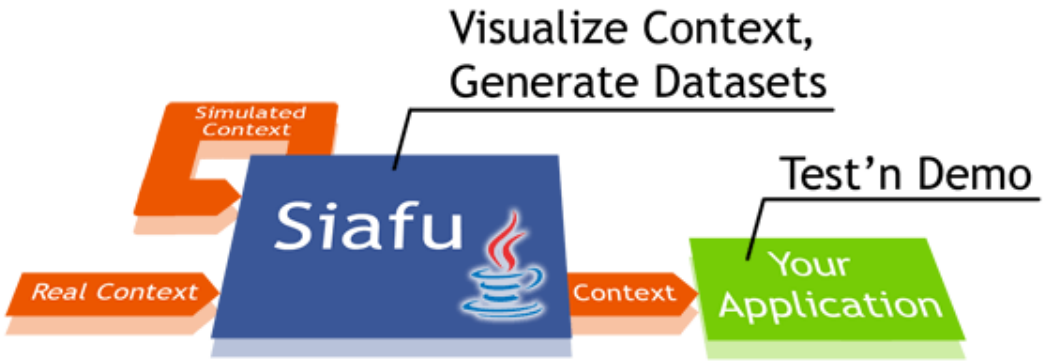

Figura 3: Arquitectura Siafu (http://siafusimulator.org)

En cuanto a las características del contexto que se pueden modelar, Siafu ofrece un amplio abanico para que el usuario pueda personalizar al máximo su simulación. Se puede influir en la posición, la proximidad o cualquier cosa que el usuario desee.

Siafu divide en tres fases el proceso de generar nuevas simulaciones:

- **Definir el entorno**
  - Imagen de fondo para el simulador
  - Mapas para los lugares
  - Mapas para las variables de contexto

- **Programar el comportamiento**
  - Modelado de los agentes (qué hacen y dónde lo hacen)
  - Modelado del escenario
  - Modelado del contexto (cómo evoluciona el contexto)

Toda la implementación y generación de agentes y componentes del escenario, se realiza utilizando un completo framework en JAVA en el cual se pone a disposición del desarrollador clase que debe de extender o implementar para dar forma a sus dispositivos. Las tres clases más importantes y sobre las que se define todo el simulador son:

- Clase Agent
- Clase World
- Clase Context Model

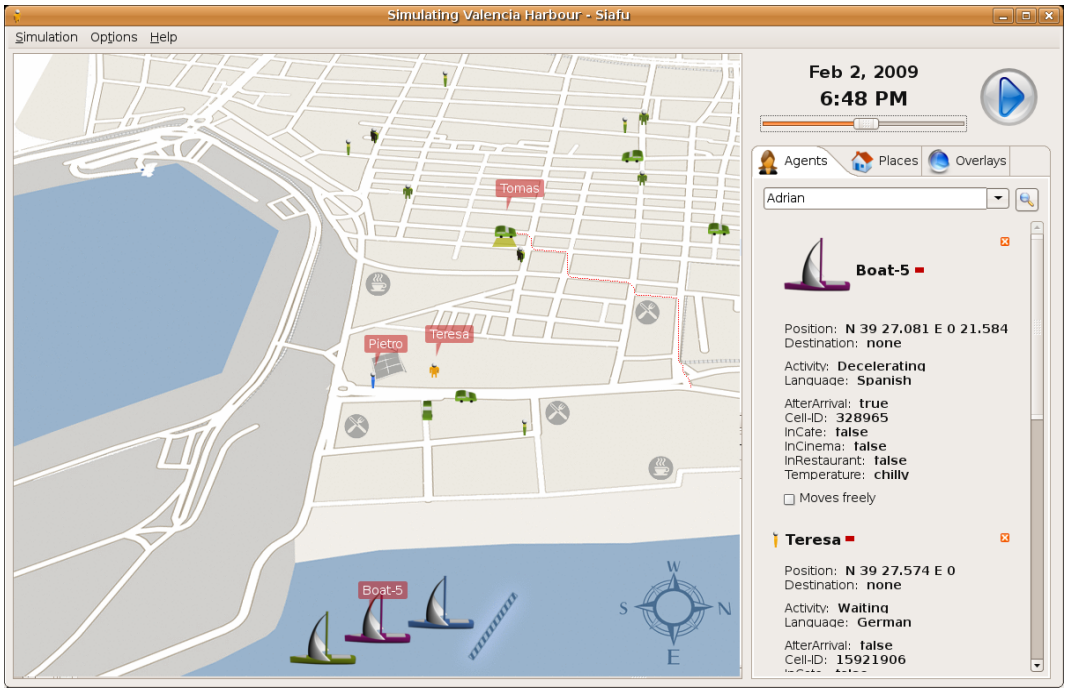

Figura 4: simulación ciudad Valencia (http://siafusimulator.org)

El los siguientes puntos, se define la creación de los agentes del escenario:

- **Unión de los componentes:** en este último paso, se genera un fichero .jar con toda la estructura de ficheros, carpetas, clases, imágenes y metadatos necesarios.

La comparación entre Siafu y la herramienta propuesta en este Trabajo de Fin de Máster nos indica que la primera es más que una aplicación, un framework muy detallado y enfocado al trabajo con grandes escenarios en los que se tienen en cuenta muchos factores. La herramienta desarrollada como parte de este trabajo, es más ligero o trabaja con una lista de dispositivos fija lo que da lugar a una personalización mas limitada.

Por otro lado, y como característica muy destacada en Siafu, hay que indicar la capacidad de asociar variables de entorno dependientes de los dispositivos mediante métodos *get* y *set* o mediante la asignación de estas variables a determinados puntos situados en el mapa.

2.3 INSTEON

INSTEON es una aplicación domótica encargada de conectar los distintos dispositivos del ámbito del hogar. Utiliza una tecnología de control y muestreo de diferentes magnitudes físicas de forma remota.

Algunas de sus aplicaciones más destacadas son:

- Control de las luces
- Detección de movimiento
- Control de las puertas del garaje

Para evaluar INSTEON vamos a analizar algunas de las características importantes de cara a estos dispositivos:

Fiabilidad: INSTEON envía las señales a través de dos medio diferentes, el aire y utilizando el cableado eléctrico disponible en las casas mediante la instalación eléctrica común. Esto facilita la obtención de unas tasas de error más bajas de lo normal. El uso de esta tecnología se conoce como “dual-band”.

Escalabilidad: Actualmente no existe un límite en la cantidad de dispositivos que puede gestionar INSTEON. Es un sistema completamente escalable que ofrece un amplio rango de escenarios en los que es capaz de funcionar correctamente. Normalmente la aplicación es capaz de gestionar 400 nodos.

Compatibilidad: Se garantiza la compatibilidad hacia atrás y hacia adelante con todos los dispositivos gestionados por la aplicación (*Forwards and Backwards Compatibility*).

2.3.1 ¿Cómo funciona INSTEON?

El funcionamiento de INSTEON está basado en 3 principios fundamentales:

- **Uso de un medio de comunicación de doble banda**: Esta característica ha sido explicada con detalle en la sección anterior referente a la forma de conseguir fiabilidad en cuanto al funcionamiento de la aplicación.

- **Repetición de mensajes**: Los dispositivos dentro de la red establecida por la aplicación actúan como repetidores en cuanto reciben un mensaje. De esta manera, no solamente se aumenta la fuerza de la señal, sino que se descubren nuevos caminos dentro de la topología de la red que permite llegar a mayor número de dispositivos. De esta forma la red de dispositivos gana en cohesión y conocimiento del estado del resto de dispositivos.

- **Utilización del estado del enlace (Statelink):** En lugar de enviar una señal para que un dispositivo vaya a un estado concreto, es el mismo dispositivo el que manda su estado en el momento de entrar en la red de dispositivos. Estos mensajes son de tipo genérico, y aseguran un alto grado de compatibilidad.

2.4 Comparativa

En este capítulo, se va a realizar un repaso de por qué se han escogido estas aplicaciones y una comparación con la solución que hemos propuesto para ver cuáles son las ventajas y desventajas que presenta.

Primero de todo, debemos de indicar que las 3 aplicaciones explicadas, son aplicaciones que se encuentran disponibles en el mercado y que vienen respaldadas por complejos desarrollos. Este punto de madurez hace que sean un buen ejemplo para mostrar los puntos y conceptos que toman de las 3 áreas de conocimiento que dan lugar a este trabajo: *Internet of Things*, *context-awareness* y domótica.

Las 3 aplicaciones recogen conceptos de Internet of Things y context awareness, mientras que INSTEON es una aplicación que podría ser más representativa del área de la domótica.

Las aplicaciones mostradas intentan aplicar los conceptos explicados anteriormente a la vida real y a ámbitos de influencia de la vida cotidiana, para obtener información y relacionarnos con el entorno que nos rodea mediante dispositivos *context-awareness*.

¿Cuáles son las ventajas y desventajas en comparación con nuestra herramienta?. Como se ha indicado antes, son aplicaciones maduras y con un proceso de desarrollo amplio a sus espaldas, lo cual no quiere decir que no podamos ver cuáles qué características comparte con nuestra solución propuesta.

La siguiente tabla, muestra una comparativa de las 4 aplicaciones para detallar de forma visual lo anteriormente explicado:

	Belkin WeMo Switch	SIAFU	INSTEON	**NUESTRA HERRAMIENTA**
Escalabilidad	Media	Alta	Alta	Alta
Definición de dispositivos	✖	✓	✓	✓
Fiabilidad	Media	Muy alta	Alta	Alta
Simulación del entorno	✖	✓	✖	✓
Dispositivos móviles	✓	✖	✓	✖
Acceso Remoto	✓	✖	✓	✓
Requiere Hardware	✓	✖	✓	✖
Presupuesto	Alto	Bajo	Alto	Bajo
Complejidad de uso	Básica	Avanzada	Media	Media

Tabla 1: Comparativa de herramientas

Como se puede observar, solamente la aplicación SIAFU y nuestra herramienta, son capaces de simular el entorno. Por otro lado, hemos encontrado que SIAFU es una herramienta mucho más compleja de utilizar, y quizá está más orientada al desarrollo con personal especializado debido al alto grado de personalización que ofrece a través de una librería extensa desarrollada en JAVA.

Por lo tanto, podemos ver que la herramienta propuesta en este trabajo ofrece unas características que no todas las aplicaciones de este entorno ofrecen: bajo coste, sin requisitos hardware y opción de definición de los dispositivos con los que queremos trabajar. Todas ellas, características muy valoradas ala hora de realizar pruebas sobre el diseño de un sistema *Internet of Things*.

2.5 Conclusiones

En el presente capítulo hemos visto ejemplos de aplicaciones disponibles en el mercado relacionadas con las áreas de influencia de este TFM. Se ha intentado que sean aplicaciones representativas de *Internet of Things*, context-awareness y domótica para ver en qué consisten, cuáles son sus principales características y qué es lo que ofrecen con respecto a nuestra herramienta.

En el siguiente capítulo, ofreceremos una visión más técnica de cuales son las tecnologías que se han utilizado en este TFM y cuáles han sido las razones de su elección.

3. Tecnologías involucradas

Durante el desarrollo de la solución propuesta en este Trabajo de Fin de Máster se han utilizado diversas tecnologías que han servido para llevar a cabo aquellos conceptos explicados en capítulos anteriores. Estas tecnologías han sido escogidas después de llevar acabo un análisis de los convenientes e inconvenientes que presenta cada una.

A lo largo de este capítulo, se van a explicar en qué consisten estas tecnologías y por qué se han escogido para alcanzar los objetivos propuestos.

3.1 Python como lenguaje de programación

Python es un lenguaje de programación interpretado, interactivo y orientado a objetos que ofrece una gran cantidad de estructuras de datos de alto nivel por medio de un tipado dinámico y fuerte, además de estas características es multiparadigma y multiplataforma [5], creado por Guido van Rossum en el año 1990 como sustituto del lenguaje de programación ABC.

A continuación se explica un poco más en detalle sus principales características [6]:

Lenguaje **interpretado**: a diferencia de los lenguajes compilados, el código escrito en Python requiere de un intérprete para poder ejecutarse.

Tipado dinámico: en Python no es necesario declarar el tipo de datos que va a contener una variable. El tipo de datos que contiene la variable dependerá de la asignación que tenga en cada momento.

Lenguaje **fuertemente tipado**: a pesar de no requerir declarar el tipo de datos que va a contener una variable, las operaciones que podrán aplicarse sobre esta variable sí que vendrán limitadas por el tipo de datos que contenga. Esto quiere decir, que si los datos son de tipo *integer*, no se podrán aplicar operaciones sobre esta variable propias de datos de tipo string, a no ser que se realice una conversión previa.

Multiparadigma: Python puede ser utilizado en diversos paradigmas de programación como la programación funcional y la programación orientada a objetos, aunque donde es más utilizado es en la programación imperativa orientada a objetos.

Multiplataforma: En el caso de que no utilicemos librerías concretas de una determinada plataforma, solamente será necesario realizar unos cambios menores en nuestros programas para que puedan ejecutarse en todos estos sistemas operativos siempre y cuando tengas disponible un intérprete:

- UNIX

- Solaris
- GNU/Linux
- DOS
- Microsoft Windows

Cuando se ejecuta por primera vez el código fuente de un programa escrito en Python, se generan unos fichero con extensión .pyc o .pyo a partir de los ficheros .py iniciales que son de tipo *bytecode* optimizado y que se utilizarán para sucesivas ejecuciones [6]. Por esta razón, actualmente es considerado como un lenguaje semi-interpretado muy parecido a lo que ocurre con el lenguaje de programación JAVA.

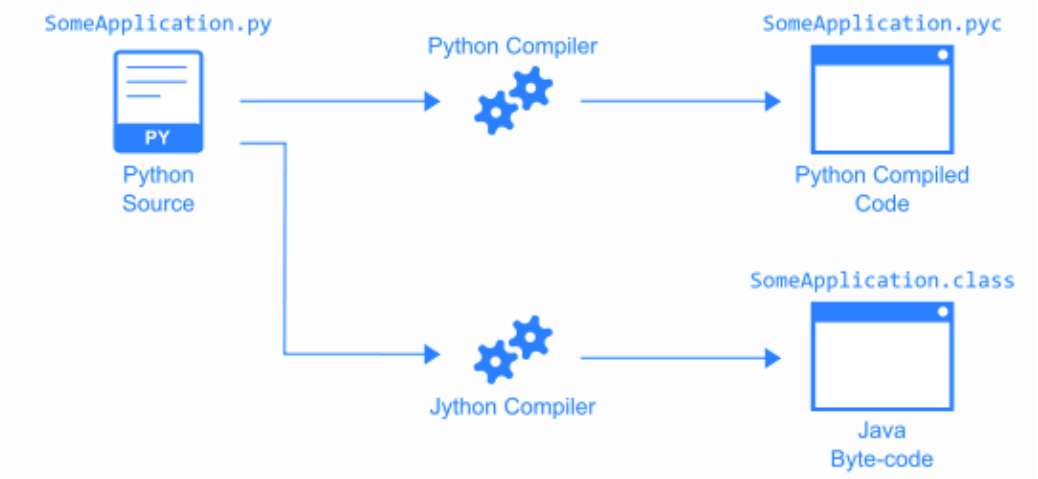

Figura 5: Compilador Python

Existen numerosas implementaciones de Python siendo la implementación de referencia la conocida como CPython basada en el lenguaje de programación C:

- ActivePython
- **CPython**
- Jython (implementación en JAVA)
- IronPython

La filosofía del lenguaje Python quedaría resumida en un fragmento del documento "PEP 20 (*The Zen of Python*)" donde se incluyen algunos puntos destacables sobre este tema:

- Bonito es mejor que feo
- Explícito es mejor que implícito
- Simple es mejor que complejo
- Complejo es mejor que complicado
- La legibilidad cuenta

Actualmente, numerosas aplicaciones utilizan Python como lenguaje de programación, al menos en parte de su código. Algunas de estas aplicaciones son Youtube, Yahoo, Google o Reddit.

Como podemos ver a continuación, el lenguaje Python es utilizado para multitud de tareas e implementaciones, lo que es una señal de su fácil adaptación a diferentes campos o aplicaciones:

En el campo de los *frameworks* para desarrollo web, encontramos Flask. Este ha sido el *framework* utilizado para este trabajo y será explicado con mucho más detalle más adelante. Como breve introducción se puede decir que es un *microframework* ligero y moderno orientado al desarrollo web basado en el *toolkit* Werkzeug y el motor de plantillas Jinja2.

Google por ejemplo utiliza Python para implementar el *backend* de algunas de sus aplicaciones como Gmail o Google Maps.

Por otro lado, aplicaciones como Dropbox o Youtube son algunos de los ejemplos de uso de este lenguaje.

En la siguiente gráfica perteneciente al año 2013, se muestra la comparativa entre los lenguajes de programación más utilizados:

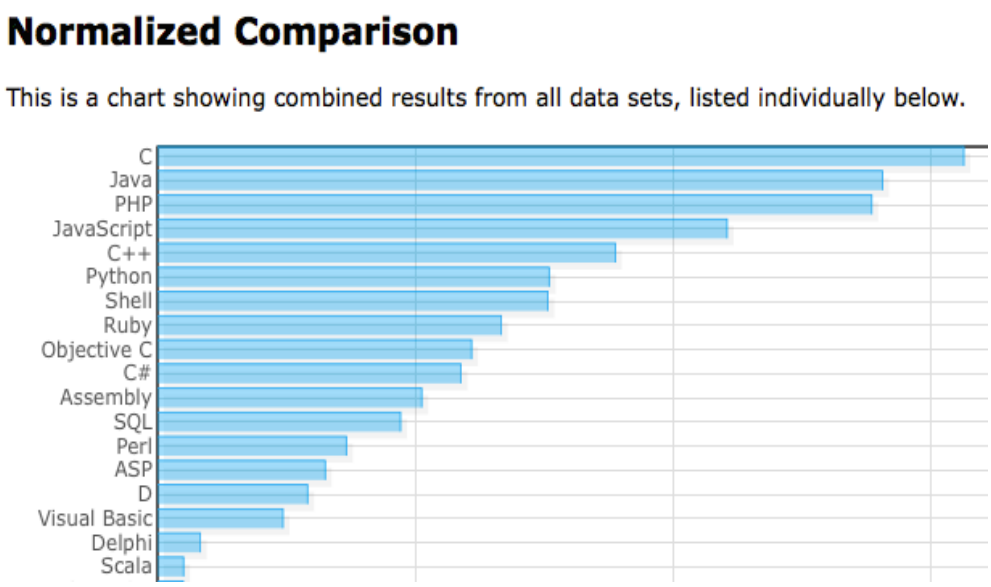

Figura 6: Utilización Python (fuente langpop.com)

3.1.1 ¿Por qué utilizamos Python?

Python es un lenguaje de uso general y ampliamente utilizado en la comunidad de desarrolladores como hemos podido mostrar en la gráfica anterior.

Python es un lenguaje cuya sintaxis busca facilitar la lectura del código generado y que éste sea fácil de entender y de seguir. El desarrollo en python es muy dinámico debido a que el tipado de sus variables es dinámico y permite una mayor facilidad de implementación. Además, es un lenguaje indicado para trabajar con cadenas de texto.

En el inicio del desarrollo de este trabajo de Fin de Máster, se buscaba un lenguaje de programación cuya curva de aprendizaje no fuese muy pronunciada y permitiese cumplir los objetivos marcados. Debido a la facilidad de generar y entender el código escrito en Python, fue desde el principio el lenguaje de programación que presentaba más ventajas.

Por otra parte, es cierto que los lenguajes interpretados pueden presentar algunos problemas de rendimiento debido a que su código debe de ser procesado primero por un intérprete, pero ya que esta característica de tipo no funcional no estaba entre las prioridades de este trabajo, se decidió no tener en cuenta este aspecto.

Uno de los factores más importantes a tener en cuenta, es que Python es un proyecto de código abierto gestionado por la Python Software Foundation, lo cual ofrece mucha facilidad a la hora de encontrar documentación debido a la amplia comunidad de desarrolladores existente, y conseguir cumplir uno de los objetivos marcados en un principio donde las tecnologías empleadas debían de ser estándares y actuales.

Por último, una de las características más interesantes de Python, es que además de tener una sintaxis fácil de entender, ofrecía todas las ventajas que se encuentra en el desarrollo orientado a objetos. Por lo tanto, el diseño de la aplicación pudo realizarse fácilmente aprovechando los conceptos de esta metodología de desarrollo.

Debido a todos estos factores, se escogió Python como lenguaje de programación para implementar el simulador.

3.2 Representational State Transfer (REST)

La Transferencia de Estado Representacional (Representational State Transfer) o REST es una técnica de arquitectura software para sistemas hipermedia distribuidos como la *World Wide Web*. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo.

REST propone una forma de intercambio de mensajes alejada de las abstracciones que se plantean en otros protocolos como SOAP. La idea es comunicar dos máquinas simplemente basándose en el protocolo HTTP y

los métodos que éste tiene definidos. Por lo tanto, REST es un protocolo más ligero (lightweight) y no tan encorsetado como SOAP o CORBA. Los sistemas que siguen los principios REST se llaman con frecuencia *RESTful*.

3.2.1 Características de REST

Utilizando los métodos especificados por el protocolo HTTP, REST ofrece la posibilidad de crear, leer, actualizar y borrar. Este conjunto de operaciones se conoce como CRUD *(Create, Read, Update and Delete)* y conforman las operaciones más básicas que se realizan sobre un objeto disponible.

Características de REST:

- Es un protocolo cliente/servidor sin estado
- Proporciona un conjunto de operaciones bien definidas
- Sintaxis universal
- Utilización de hipermedios
- Independiente de plataforma
- Independiente de lenguaje de programación utilizado
- Basado en estándares conocidos (HTTP)

3.2.2 Métodos HTTP en REST

En REST los objetos disponibles a través de la red son tratados como recursos. Sobre estos recursos se aplican las operaciones definidas e implementadas por HTPP. Estos métodos son:

- GET – Recupera la representación definida de un recurso
- PUT – Actualiza una propiedad que afecta a un recurso
- DELETE – Borra un recurso existente
- HEAD – Obtiene las cabeceras del recurso utilizado
- POST – Transfiere información a un programa en una URL dada

Mediante estos métodos se implementan las operaciones sobre los recursos que posteriormente se publican a través de la red. Estos métodos forman lo que sería una API *RESTful* y facilita una uniformidad en todas las API basadas en REST.

3.2.3 Mensajes HTTP

El protocolo http además, proporciona una línea de estado muy útil para comprobar el estado de la petición realizada al servidor. Esta línea esta compuesta por la versión del protocolo HTPP seguido de un código numérico junto con el texto asociado a éste. Éste código numérico indica el estado de la solicitud realizada, estos códigos van desde 200 en el caso de que el resultado obtenido de la petición sea correcto o 404 en el caso de que el recurso solicitado no se haya podido encontrar, hasta llegar al código 504 donde se indica que el servidor no responde. Estos tres campos van separados por espacios en blanco.

CÓDIGO	NOMBRE	EXPLICACIÓN
200	OK	PETICIÓN CORRECTA
201	CREATED	PETICIÓN COMPLETADA
202	ACCEPTED	PETICIÓN ACEPTADA
206	PARTIAL CONTENT	LA PETICIÓN SERVIRÁ PARCIALMENTE EL CONTENIDO SOLICITADO
303	METHOD	MÉTODO DE LA PETICIÓN
400	BAD REQUEST	SOLICITUD INCORRECTA
403	FORBIDDEN	EL SERVIDOR REHÚSA RESPONDER
404	NOT FOUND	RECURSO NO

		ENCONTRADO
500	INTERNAL ERROR	ERROR INTERNO EN EL SERVIDOR
503	SERVICE UNAVAILABLE	SERVICIO NO DISPONIBLE
504	GATEWAY TIMEOUT	TIEMPO DE ESPERA AGOTADO

Tabla 2: Mensajes HTTP

Respecto a las características detalladas anteriormente, hay que explicar la referente a que REST es un protocolo cliente/servidor sin estado. Para ello, debemos de entender que hay una diferencia entre el estado del recurso y el estado de la aplicación. El estado de la aplicación va variando según evoluciona la ejecución mientras que el estado del recurso nunca varía [7] El estado de la ejecución varía debido a que representa el cambio que ha seguido el usuario a través del flujo de la aplicación para alcanzar el recurso. Por otro lado, como hemos indicado anteriormente, el estado del recurso no varía.

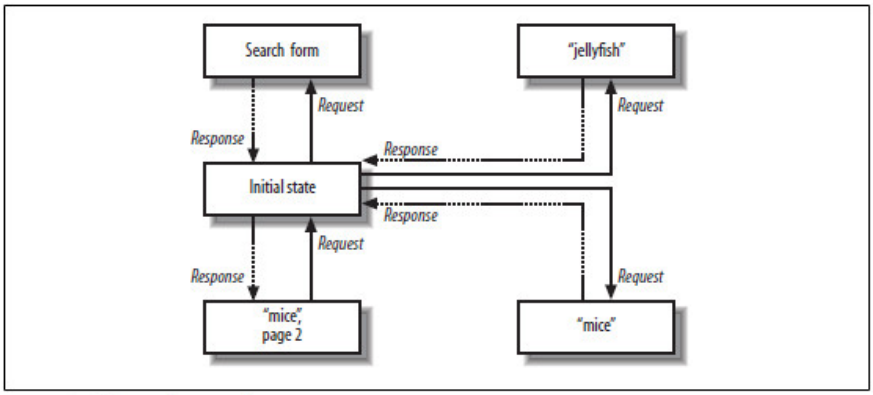

Figure 4-1. A stateless search engine

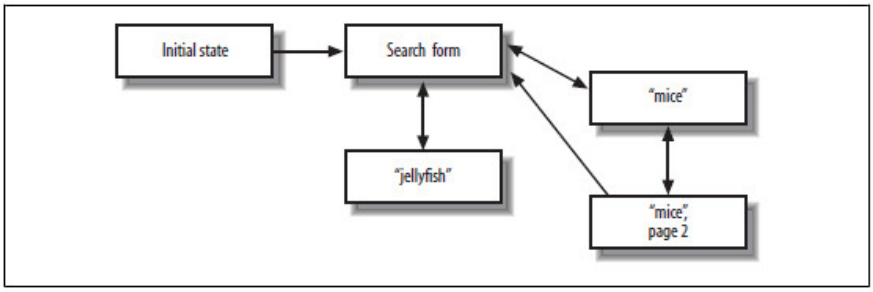

Figure 4-2. A stateful search engine

Figura 7: búsqueda con y sin estado

En REST una petición es completa y se realiza sin necesidad de ningún dato perteneciente a otra petición anterior, todo lo necesario para que el servicio web gestione la petición lo recibe en una sola vez y se evalúa contra el recurso sobre el que ejecuta la petición. Además, cada recurso es accesible únicamente por una única URI.

Esta característica es muy importante porque afecta directamente sobre la capacidad de escalabilidad de las aplicaciones. En aquellas aplicaciones donde las peticiones están interrelacionadas, resulta muy complicada la escalabilidad, ya que en caso de requerir información de otras peticiones no gestionadas por el mismo servidor, es imposible continuar. Manteniendo el protocolo REST sin estado, permite poner a disposición de la aplicación varios servidores que pueden gestionar todo el tráfico simplemente con la mediación de un balanceador de carga ya que cada petición es individual e independiente.

3.2.4 Arquitectura Orientada a Recursos (ROA)

Los recursos en REST se exponen en la red a través de una API en la que se conjugan las URIs, el protocolo HTTP y XML o JSON.

La arquitectura orientada a recursos ofrece una forma sistemática y definida de cómo utilizar las tecnologías disponibles para cumplir una serie de requisitos y objetivos. Estas metodologías son muy útiles para los desarrolladores ya que establecen las bases a seguir para ofrecer sus servicios a través de la red.

La necesidad de definir una arquitectura viene dada porque REST establece un conjunto de patrones de diseño software, pero no define una arquitectura. Por lo tanto, podemos decir que no existe el concepto de arquitectura REST.

Esta arquitectura ofrece las bases para:

- Establecer y definir los recursos
- Los nombres de los recursos
- La representación de estos recursos

- Los enlaces entre los recursos

Por otra parte, el recurso que se expone en internet se hace con una determinada representación que compone la definición del recurso. Esta representación se envía utilizando protocolos de intercambio de mensajes como XML o JSON.

3.2.5 URI's de los recursos

Para que un recurso sea un recurso, debe de tener una URI asociada. Mediante esta URI, el recurso pasa a ser tal y es accesible a través de la red.

No existen reglas fijas a la hora de establecer la nomenclatura de estos enlaces, pero sí que deben de ser descriptivos de aquello a lo que representan. El recurso al que apunta el enlace debe de poderse deducir solamente viendo la estructura de la URI. Tenemos que tener en cuenta que estos enlaces pueden ser utilizados por otras aplicaciones o publicados en otros medios, por lo que deben de ser fáciles de entender.

Además, esta URI debe de ser única y solamente representa a un recurso. Una misma URI no puede ser utilizada para dos recursos distintos.

Un ejemplo sería la siguiente dirección web:

- *http://www.ejemplo.com/pedidos/2010/enero*

En la URI anterior, se puede observar fácilmente como mediante esa dirección se busca acceder al recurso en el que están los datos de ventas del año 2010 pertenecientes al mes de Enero.

Esta relación no es bidireccional, ya que un mismo recurso sí que puede estar accesible por medio de dos URI's distintas que apuntan a una misma información. Por ejemplo:

- *http://www.ejemplo.com/pedidos/2010/enero*
- *http://www.ejemplo.com/pedidos/2010en*

En este caso, las dos URI's enlazan con el mismo recurso, pero son distintas. Esto es algo que los desarrolladores deben de prevenir y aplicando una serie de medidas a la hora de organizar la estructura de los recursos.

3.2.6 Seguridad e Idempotencia

Ambas son propiedades que deben de cumplir las operaciones definidas sobre los recursos mediante los métodos HTTP.

Seguridad: GET y HEAD

Esto implica que cuando se utilizan estos métodos de un recurso concreto, no se debe de producir un cambio de estado en la parte servidor. Esto quiere decir, que debería de producir el mismo efecto en el servidor ejecutar cualquiera de estas dos peticiones una o infinitas veces.

Idempotencia: PUT y DELETE

De la misma forma que la propiedad de seguridad, aplicar estos dos métodos sobre un mismo recurso una o más de una vez no debería de producir un cambio en el servidor. Evidentemente, el método PUT afectará al estado del recurso porque actualizará su estado, pero el servidor no se verá afectado por ello.

3.2.7 Comparativa SOAP y REST

La siguiente tabla muestra una comparativa entre SOAP y REST:

SOAP	REST
Diseñado para su utilización en entornos de computación distribuida	Comunicación Punto a Punto
Requiere herramientas y middleware extenso	Solamente requiere soporte del protocolo HTTP
El contenido del mensaje nos indica qué recurso se accede	La URL nos indica el recurso al que se quiere acceder
Alto nivel de expresividad	Bajo nivel de expresividad
Alto nivel de fiabilidad	Bajo nivel de fiabilidad
El formato de las respuestas no es fácil de entender	El formato de las respuestas es fácil de entender debido a que son texto plano (JSON o XML)
Las peticiones trabajan con un stado	Las peticiones no tienen ningún estado asociado
Mayor uso del ancho de banda disponible para realizar sus operaciones	Menor uso del ancho de banda disponible para las operaciones

Tabla 3: Tabla comparativa SOAP y REST

Un punto importante a tener en cuenta, es que la depuración de la aplicación utilizando el protocolo REST puede realizarse con un simple navegador web tal y como explicaremos en próximos capítulos. Esto contrasta con el protocolo SOAP donde se requiere de software específico para poder realizar pruebas con la aplicación desarrollada y analizar el contenido de las peticiones y respuestas.

3.2.8 ¿ Por qué utilizamos REST ?

El uso de REST para el desarrollo del simulador vino determinado por su sencillez en cuanto a la sintaxis utilizada.

Por otra parte, hay que destacar que debido a la elección de Python como lenguaje de programación de la aplicación, REST es más recomendable debido a que continúa con la filosofía "*lightweight*" o ágil que se buscaba alcanzar.

La elección entre SOAP y REST podría definirse como decantarse entre la estructura, organización y jerarquía que se establece con SOAP y el alto grado de accesibilidad que ofrece REST.

3.3 Lenguajes de intercambio de datos: JSON y XML

Una vez explicados con que lenguaje de programación y con que framework se ha desarrollado este Trabajo de Fin de Máster, hay que explicar el uso que se le ha dado a dos lenguajes de intercambio de datos ampliamente conocidos y utilizados como son JSON y XML.

3.3.1 JSON

JSON o JavaScript Object Notation es una forma de guardar e intercambiar la información con un formato que resulta ligero y fácil de leer y entender para los seres humanos.

JSON es independiente del lenguaje de programación utilizado para desarrollar la aplicación lo que facilita su adaptación para cualquier desarrollo que llevemos a cabo sin tener que realizar ningún cambio [8].

Básicamente, JSON está basado en dos estructuras:

Una colección de pares nombre/valor: en la mayoría de lenguajes, esto se considera como un objeto, hash table, diccionario, struct..

Una lista ordenada de valores: en muchos lenguajes esto se considera un array, lista o vector.

El hecho de poder implementar JSON con numerosas estructuras de datos presentes en cualquier lenguaje de programación moderno, es uno de los factores más importantes a la hora de elegir JSON como lenguaje de intercambio de datos. En el caso de este Trabajo de Fin de Máster, nos ha ayudado a cumplir uno de los objetivos globales marcados en el inicio de utilizar tecnologías independientes de lenguaje y plataforma y que supongan un estándar.

Para visualizar mejor como está organizada una estructura de datos JSON, a continuación se muestran unos diagramas explicativos [8].

Un objeto JSON se expresaría por un conjunto de pares nombre/valor comprendidos entre llaves "{}", donde cada nombre está separado por dos puntos ":" y cada valor por una coma ",".

Un array es una colección ordenada de valores que se encuentran comprendidos entre llaves "[]" y separados por comas ",".

Además de toda la información dada, hay que indicar que el rango de valores que pueden tomar van desde un integer hasta un string:

Respecto al formato de un string, debemos de decir que es un conjunto de caracteres en formato UNICODE englobados por dobles comillas (" "). Un string es muy parecido a lo que se considera string en JAVA o C.

En lo que respecta a los números, el formato que se utiliza es el mismo que en JAVA o C excepto que el formato octal y hexadecimal no son utilizados:

3.3.2 Correspondencia JSON – Python

Como ya hemos explicado en el apartado anterior, JSON es independiente de cualquier lenguaje de programación, debido a que puede ser representado por diferentes estructura de datos disponibles en muchos lenguajes.

En nuestro caso, nos interesa conocer como se relacionan las estructuras explicada de JSON con el lenguaje de programación Python que hemos utilizado para desarrollar este Trabajo de Fin de Máster.

Las siguientes tabla, está obtenidas de la documentación publicada sobre Python y muestra como las clases *Encoder* y *Decoder* trabajan en la conversión de Python a JSON y de JSON a Python.

Tabla correspondencia Encoder:

PYTHON	JSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	integer
True	true
False	false
None	null

Tabla 4: Encoder Python – JSON

3.3.3 XML

Extensible Markup Language (XML) es un lenguaje de marcado que define un conjunto de reglas para organizar la información en un formato que sea fácil de entender y de leer para las personas y las máquinas.

Las bases de XML están definidas en XML 1.0 por la W3C (*World Wide Web Consortium*) como un estándar libre.

Para el desarrollo del simulador, el lenguaje XML se utiliza para codificar la información obtenida a través de la aplicación web donde el usuario puede registrar los dispositivos. Esta información se guarda como XML y se utiliza como datos de entrada para el procesador escrito en python.

XML es un lenguaje basado en etiquetas que engloban valores asociados. Existen etiquetas (tags) de apertura y etiquetas de cierre, y la información contenida entre éstas, se considera asociada al nombre que se le da a la etiqueta.

- Etiqueta de apertura: <dispositivo>
- Etiqueta de cierre: </dispositivo>

Un ejemplo de fragmento de XML sería el que define a aun sensor de sonido:

```
<SimulatedDevice      deviceID="sound1"
                      name="sound1" type="1"
                      subtype="SOUND_LEVEL_METER"
                      initialState="off">

<SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationMode"
value="es.upv.pros.coolSimulator.sensorNotificationModeAutomatic"/>

<SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationInterval"
value="90"/>
```

<SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationMinValue"
value="3"/>
<**SimulatedDeviceProperty**
property="es.upv.pros.coolSimulator.sensorNotificationMaxValue"
value="3"/>

</SimulatedDevice>

En este fragmento, podemos ver en negrita las etiquetas de apertura y de cierre. En este caso la etiqueta <SimulatedDevice> contiene una serie de valores dentro de la etiqueta que conforman lo que serían los atributos asociados a esa etiqueta en concreto. Además, tenemos etiquetas de tipo <SimulatedDeviceProperty> que definen propiedades del dispositivo y que se encuentran englobadas dentro de la etiqueta de cierre </SimulatedDevice>.

Como se puede observar, XML es un lenguaje bastante intuitivo y fácil de entender. Esta forma de codificar la información resulta muy útil de cara a ser procesada por máquinas debido a que en todos los lenguajes de programación existen librerías capaces de leer este tipo de información para extraer los datos necesarios en cada momento.

3.3.4 Comparación XML y JSON

Durante muchos años, el lenguaje de marcado XML ha sido el estándar y el formato más utilizado en el desarrollo web. XMl surgió como un derivado del Standard Generalized Markup Language (SGML), más simple y fácil de trabajar con él. Además, XML era propuesto y promocionado por la *World Wide Web Consortium* (W3C) lo que favorecía su adopción por parte de la industria.

En su momento, XML suponía tres grandes ventajas:

- Estaba basado en texto
- Se podía entender
- La posición del código era independiente

- Interoperabilidad y estándar abierto

Por otro lado, tenía una gran desventaja, y es que no era un lenguaje que funcionaba bien para el intercambio de datos. Principalmente, porque acarrea un montón de información extra que provoca que las comunicaciones de datos sean demasiado lentas y complejas.

Con el auge de la web 2.0, se produjo un aumento de la cantidad de datos que se necesitaba intercambiar entre las distintas aplicaciones que funcionaban a través de internet. XML era algo complejo para realizar cosas que en principio eran sencillas y no requerían de todo el potencial que ofrecía. De esta forma, empezó a quedar al descubierto el punto débil de XML y surgió JSON (Javascript Object Notation).

Es cierto que JSON ofrece ciertas ventajas sobre XML, aunque también tiene sus desventajas. La principal desventaja de JSON es la falta de extensibilidad, esto quiere decir que así como en XML podemos transferir imágenes, tablas o gráficos, en JSON estamos limitados a texto y números.

El aumento de uso de JSON no quiere decir que sea mejor que XML, solamente que hay otra opción más disponible a la hora de desarrollar aplicaciones web y en cada caso el diseñador debe de conocer cuales son los requisitos que debe de cumplir para elegir uno u otro.

3.3.5 ¿ Por qué utilizamos JSON?

En el caso concreto de éste Trabajo de Fin de Máster, se decidió que el input proporcionado al simulador sería más interesante que fuese en XML por la facilidad de lectura debido a la existencia de las etiquetas, mientras que para la transferencia de archivos era más conveniente JSON, ya que la transferencia de datos iba a ser muy sencilla y solamente con formato de texto y números.

Además, la curva de aprendizaje es más corta debido a la simplicidad de la sintaxis, lo que favoreció la elección de JSON como el lenguaje que se utilizaría para el intercambio de mensajes en el simulador.

3.4 Microframework Flask

Flask es un microframework para Python basado en Werkzeuf y Jinja2.

El término “micro” viene dado no por el bajo número de líneas de código o por la cantidad de ficheros utilizados para nuestra aplicación. El término “micro” viene dado porque se centra en mantener un núcleo básico de funcionalidades dejando cosas como la validación de datos o la capa de abstracción de la base de datos para otras de las muchas librerías ya existentes. Por otra parte, Flask se caracteriza por ser extensible y ser capaz de ofrecer soluciones al usuario para desarrollar sus aplicaciones con un alto grado de personalización a la vez.

Las características más importantes de Flask son:

- Capaz de desplegar un servidor de desarrollo y un depurador de código
- Soporte para testing unitario
- Trabaja con peticiones REST
- Compatible con Jinja2
- Utilización de codificación Unicode
- Compatibilidad con Google App Engine
- Extensiones disponibles para desarrollar funcionalidades específicas
- Gran cantidad de documentación disponible

3.4.1 ¿ Por qué utilizamos Flask ?

La decisión de utilizar Python como lenguaje de programación y la adopción de tecnologías ligeras para el desarrollo de la aplicación llevó a decidirse por Flask como framework de desarrollo web.

Flask sigue con la misma idea que representan REST,JSON y Pyhon. Una metodología fácil de entender y cuya curva de aprendizaje es razonable.

Trabajar con Flask es sencillo y permite definirse direcciones URL de los recursos claras y que no dan pie a confusión. Una vez declarada la ruta de nuestro recurso, solo queda asociarle la clase que se encarga de gestionar ese recurso mediante los métodos http que tenga implementados:

Por ejemplo, la siguiente línea :

api.add_resource(**LightRest,'/simulator/SimJulio/LightRest'**)

Añade a nuestra API un recurso llamado "LightRest" que a su vez esta siendo manejado por la clase LightRest, donde tenemos los métodos http GET y HEAD definidos.

Esta forma de dar de alta nuestros recursos en el servidor que ofrece Flask resulta muy intuitiva y permite depurar el código fácilmente. Para ello, simplemente hay que utilizar un navegador web y utilizar los métodos definidos sobre nuestro recurso

3.5 Conclusiones

En el presente capítulo se ha presentado de forma detallada cuáles son las tecnologías utilizadas para la realización de la solución propuesta. Se han mostrado comparativas de las ventajas y desventajas que ofrecen con respecto a otras tecnologías similares y finalmente se ha justificado porque se han escogido.

En el siguiente capítulo veremos una visión global de la arquitectura de la solución propuesta y unos rasgos generales de los componentes del simulador.

4. Solución Propuesta

4.1 Descripción global

En este capítulo se mostrará una visión global de la solución que hemos propuesto para que la aplicación ofrezca la funcionalidad para la que ha sido diseñada.

En primer lugar, tenemos un primer módulo donde se lleva a cabo todo el procesamiento de datos a partir de los cuales se trabajará en los siguientes pasos. Esta función se lleva a cabo por una aplicación web llamada “Simulator” (Servicio de especificación de dispositivos) donde el usuario es capaz de proporcionar la definición de los dispositivos con los que desea realizar la simulación mediante un formulario web, y a su vez gestionar todos los componentes que quiera.

Como resultado del primer módulo, se obtiene un fichero de entrada con toda la información necesaria, la cual será gestionada a través de otro módulo en el cual intervienen una serie de clases implementadas en python que se encargarán de trasladar toda esta información recibida a objetos en memoria que representarán al simulador y todos los dispositivos.

Con toda esta información disponible en memoria, se generarán de forma automática mediante una serie de funciones disponibles en la clase Simulator, un conjunto de ficheros que serán la base para implementar la solución necesaria para utilizar el microframework Flask.

Los ficheros resultantes del proceso anterior nos permitirán inicializar un servidor local mediante el cual podremos acceder a nuestros dispositivos mediante una serie de recursos REST que facilitarán la interacción con éstos.

El siguiente diagrama muestra la solución propuesta, y como los diferentes componentes interactúan entre sí, para dar como resultado una simulación de un entorno, en el cual se encuentran diferentes dispositivos que además sufren cambios en sus estados.

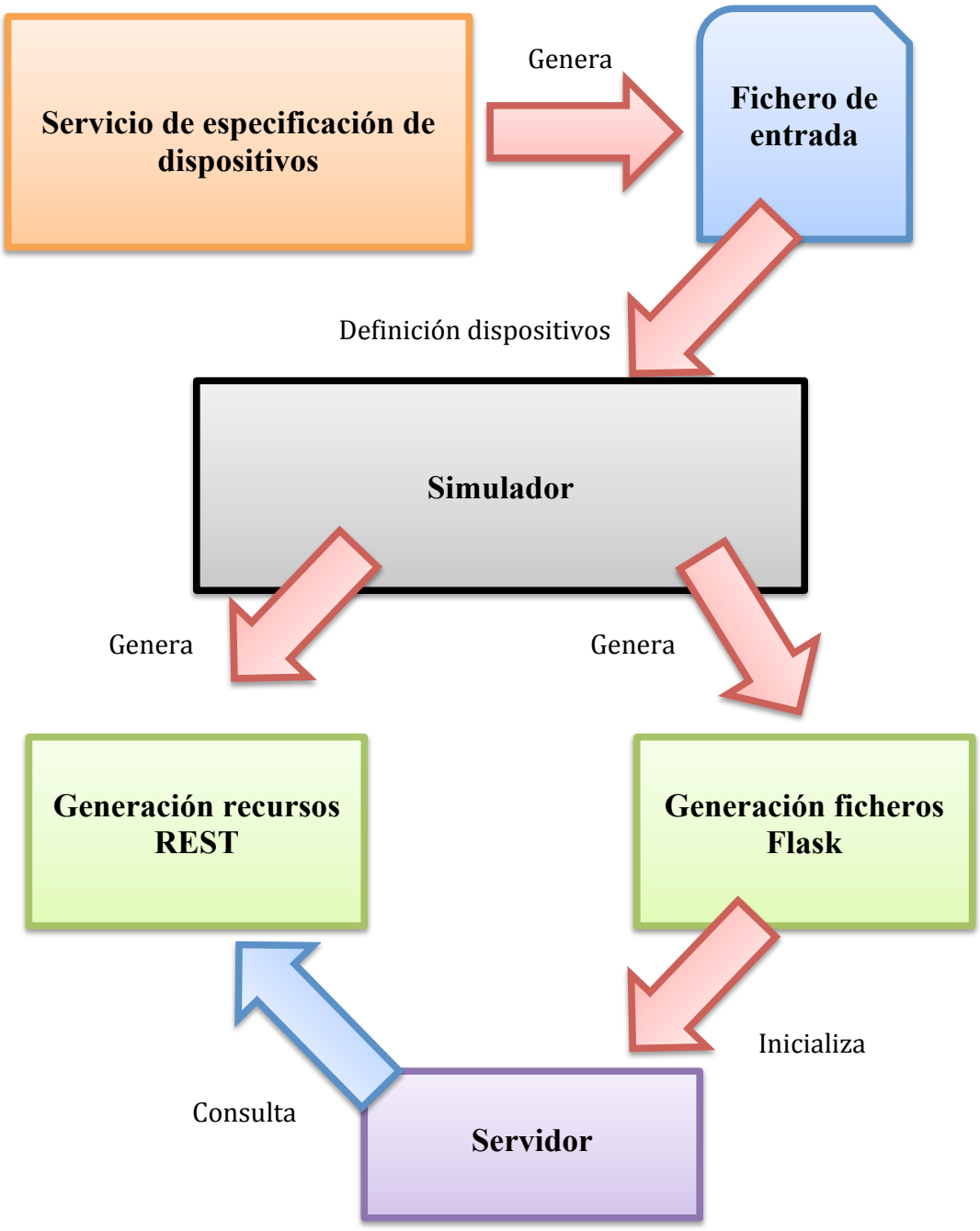

Figura 8: Arquitectura de la solución

El diagrama anterior muestra una visión general de la solución propuesta en este Trabajo de Fin de Máster en la cual se combinan todas las tecnologías previamente explicadas en el capítulo 3.

El desarrollo se ha llevado a cabo basándose en un patrón de diseño software Modelo-Vista-Controlador donde se diferencian 3 partes claramente para hacer que la aplicación sea fácil de mantener y de evolucionar en un futuro:

1. **El modelo** detalla la forma en que los datos con los que trabaja la aplicación está representada.

2. **El controlador** representa los eventos que se activan para realizar consultar o trabajar con la información proporcionada con el modelo.

3. **La vista** es la representación visual que tiene el usuario final de la aplicación y la interfaz a través de la cual interactúa con el simulador. En el caso de este TFM, el componente vista no está presente debido a que no existe ninguna interfaz a través de la cual interactuar con el simulador. El TFM cubre los procesos que se llevan a cabo para obtener los resultados definidos en secciones anteriores.

El siguiente diagrama, muestra la relación que se establece entre los 3 componentes que conforman el patrón:

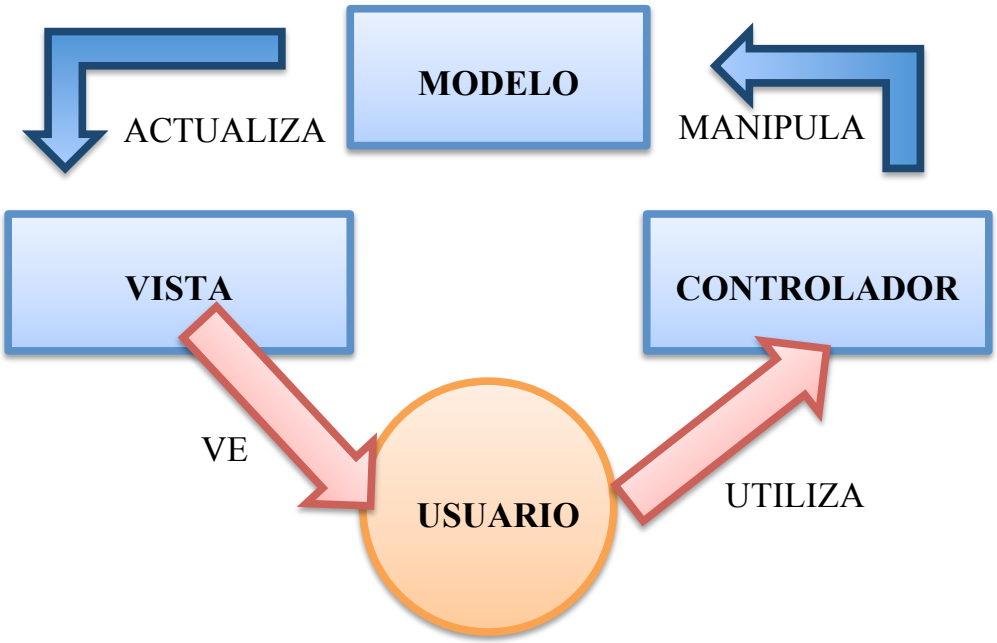

Figura 9: Diagrama Modelo-Vista-Controlador

4.2 Conclusiones

En el presente capítulo se ha mostrado el planteamiento de la solución planteada. Se ha dado una visión global mediante un diagrama de todos los componentes y el flujo básico de la aplicación. Además, se ha descrito el patrón de diseño utilizado para establecer las bases de este TFM.

En el siguiente capítulo, veremos una visión técnica de cómo se utilizan las tecnologías escogidas, para implementar la arquitectura propuesta en este capítulo de forma detallada.

5. Desarrollo del Simulador

El primer paso en el desarrollo del simulador, es la generación de un fichero XML con todas las especificaciones de los dispositivos que el usuario ha definido en su arquitectura.

Toda esta información de entrada se registra a través de una aplicación web que se encuentra en esta dirección: galanica.dsic.upv.es/simulator

Esta aplicación se encuentra disponible en un servidor alojado en el departamento del DSIC de la Universidad Politécnica de Valencia.

Las tecnologías utilizadas para el desarrollo de esta aplicación son:

- PHP
- Python
- JQuery
- Javascript
- Twitter Bootstrap
- Scripts de Shell

Para el desarrollo de la aplicación web Simulator, se ha utilizado una arquitectura Modelo-Vista-Controlador explicado anteriormente con detalle y un patrón de diseño Maestro-detalle.

El patrón Maestro-Detalle es un patrón compuesto por otras unidades de interacción. Consiste en mostrar los objetos de la interfaz como una relación en la que se establece un nodo principal del que surgen otros nodos con los detalles de nodo principal [9].

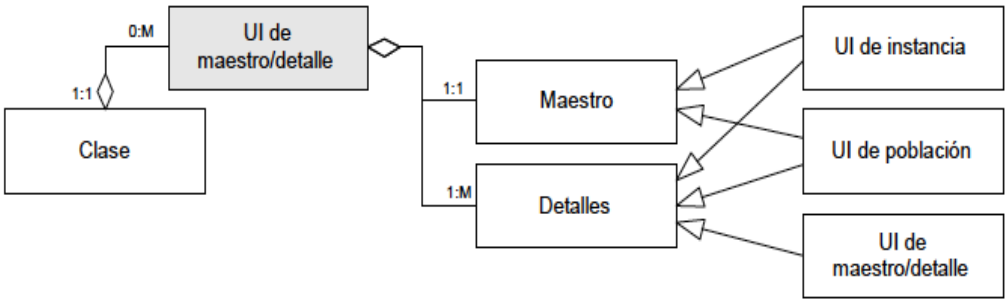

Figura 10: Metamodelo de la unidad de interacción de maestro/detalle[9]

Para acceder a la aplicación es necesario un usuario y una contraseña cuya configuración se guarda en una base de datos SQL.

Una vez estamos dentro de la aplicación, debemos de crear un nuevo simulador al que posteriormente asociaremos una serie de dispositivos que tendrán un conjunto de características definidas. La definición de los dispositivos se lleva a cabo a través de un formulario como el siguiente:

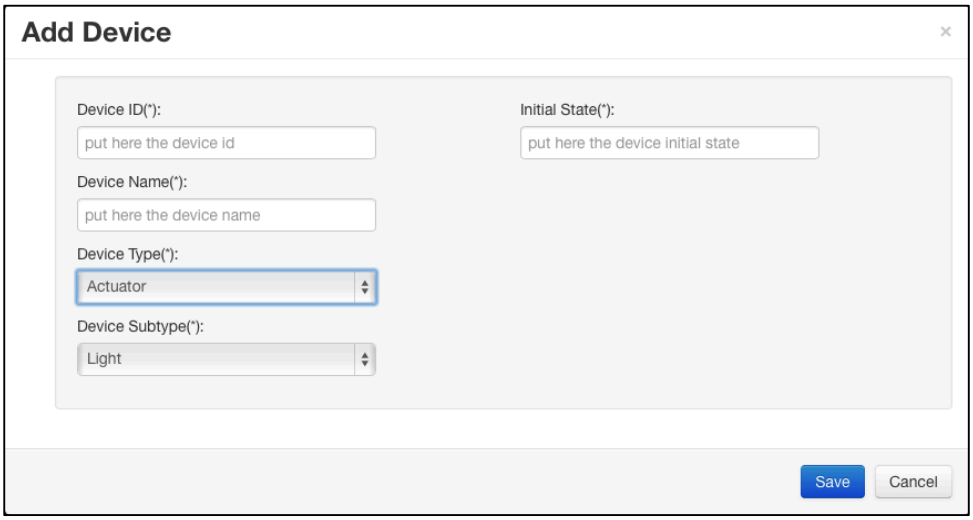

Figura 11: Formulario dispositivos

5.1 Los dispositivos

Hemos dividido los dispositivos admitidos por el simulador, en dos grandes grupos dependiendo de la forma en que el usuario debe de interactuar con ellos: actuadores y sensores.

Los dispositivos actuadores son aquellos en los que el usuario debe de tener un rol activo e inicializar el evento que provoca un cambio en el estado del dispositivo. Los dispositivos de tipo actuador utilizados en el simulador son los que se muestran en la siguiente tabla:

DISPOSITIVO	DESCRIPCIÓN
Luz	Dispositivo que enciende o apaga la luz
Luz Gradual	Dispositivo para encender o apagar la luz y además asignarle una determinada luminiscencia
Puerta	Dispositivo que abre o cierra una

	puerta
Calefacción	Dispositivo de encendido y apagado de la calefacción
Persianas	Dispositivo de subida o baja de las persianas de una ventana
Alarma de luz	Dispositivo de encendido o apagado de una alarma visual
Alarma de sonido	Dispositivo de encendido o apagado de una alarma sonora

Tabla 5: Tabla de dispositivos actuadores

Respecto a los dispositivos de tipo sensor, son aquellos que miden determinadas magnitudes físicas que se encuentran en el contexto del simulador. Estos dispositivos no requieren ninguna acción previa del usuario ya que su estado va cambiando en función de los cambios que se producen en el entorno. Los dispositivos sensores son:

DISPOSITIVO	DESCRIPCIÓN
Sensor Brillo	Sensor que mide la luminosidad de la luz
Sensor Presión	Sensor que mide la presión atmosférica
Sensor Viento	Sensor que mida la velocidad del viento
Sensor Temperatura	Sensor que mide la temperatura
Sensor Sonido	Sensor que mide la potencia del sonido
Sensor Magnetismo	Sensor que detecta campos magnéticos
Sensor Movimiento	Sensor que detecta el movimiento

Tabla 6: Tabla de dispositivos sensores

Los dispositivos actuadores y sensores comparten una serie de atributos que son comunes para los dos, pero sin embargo cada tipo tiene una serie de atributos específicos que los diferencia entre si.

Las características que definen a los dispositivos actuadores y sensores son:

ATRIBUTOS	DESCRIPCIÓN
Identificador	
Nombre	
Tipo	En el caso de un actuador su valor será “0”, mientras que para un sensor será “1”.
Subtipo	Cualquiera de los previamente definidos
Estado	

Tabla 7: Atributos comunes de los dispositivos

Por otro lado, las características que son particulares de los dispositivos sensores son:

ATRIBUTOS	DESCRIPCIÓN
Modo	Indica si el refresco del estado del dispositivo es automático o manual. Este atributo es **obligatorio**.
Intervalo	Indica cada cuando se actualiza el estado de un dispositivo. Solamente disponible para un modo automático. Este atributo es **obligatorio** cuando el modo es automático.

Valor Máximo	Indica el valor numérico máximo que puede tomar el estado. Este atributo es **opcional**.
Valor Mínimo	Indica el valor numérico mínimo que puede tomar el estado. Este atributo es **opcional**.

Tabla 8: Atributos de los sensores

Algunos de los atributos indicados pueden estar definidos o no en un dispositivo sensor. Dependerá de los datos de entrada facilitados.

En el caso del atributo "Modo", influirá en los atributos "Intervalo", "Máximo" y "Mínimo", ya que si este tiene el valor "Automático", estos 3 valores no serán necesarios, mientras que si el valor es "Manual", entonces deberemos de especificar los valores.

En la siguiente figura, se muestra un diagrama de clases que establece las relaciones entre un simulador y los dispositivos:

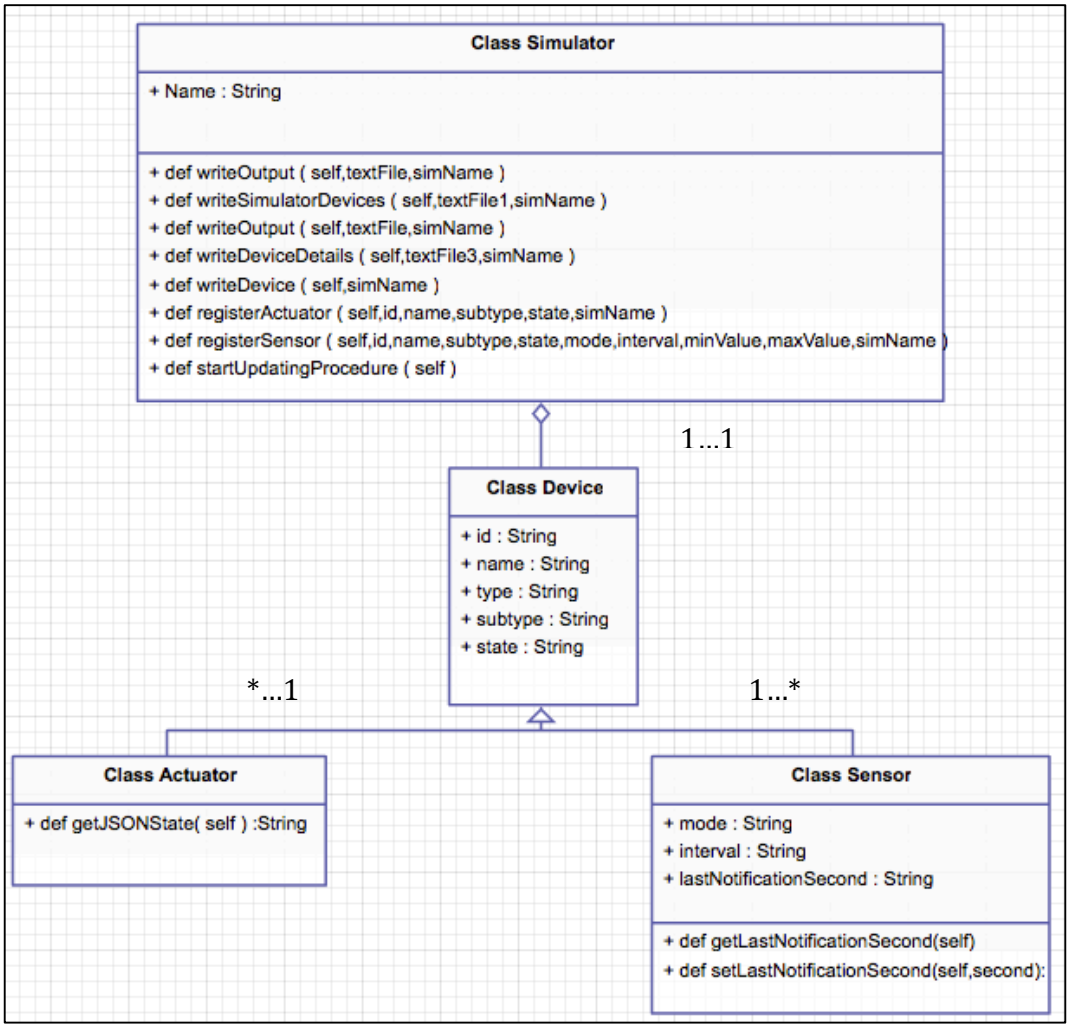

Figura 12: Diagrama clases simulador

5.2 Procesado del fichero de entrada

Como se ha indicado en secciones anteriores, toda la información con las especificaciones sobre los dispositivos que vamos a poder utilizar en nuestro simulador, se indican a partir de un fichero XML generado a través del servicio de definición de dispositivos (Simulator) disponible en Galanica.upv.es/simulator.

En la implementación que hemos llevado a cabo, este fichero es un fichero en formato XML, pero podría estar en cualquier formato. Lo importante es que supone los datos de entrada de la solución mediante los

cuales el simulador obtiene la definición de los dispositivos con los que vamos a trabajar a través de una serie de metadatos que contiene.

La lectura y extracción de toda la información del fichero de entrada, para generar los objetos y los ficheros necesarios para que funcione el simulador se lleva a cabo en módulo **Parser.py**.

El módulo Parser.py lleva a cabo las siguientes funciones:

- Conversión del fichero de una estructura XML a una estructura DOM
- Lectura del contenido
- Validación datos entradas
- Generación de los objetos en memoria

Este módulo recibe como parámetro de entrada el fichero XML con toda la información y lo procesa mediante una librería externa disponible para Python llamada **MiniDom**.

Cuando se incializa el módulo Parser.py a través de consola, recibe como argumento el nombre del fichero XML generado. Esto se ve reflejado en el código en las siguientes líneas:

```
arguments = str(sys.argv)
xmlFile = sys.argv[1]
```

Donde se ve como se especifican los argumentos de la entrada de datos por consola y nos quedamos con el primer elemento el cual guardamos en la variable **xmlFile**.

Una vez ya tenemos el fichero de entrada, este es procesado mediante la librería **MiniDom**, la cual convierte el contenido en formato XML en un árbol a través del cual podemos movernos utilizando las diferentes etiquetas disponibles.

El proceso de conversión se vería reflejado en el siguiente diagrama:

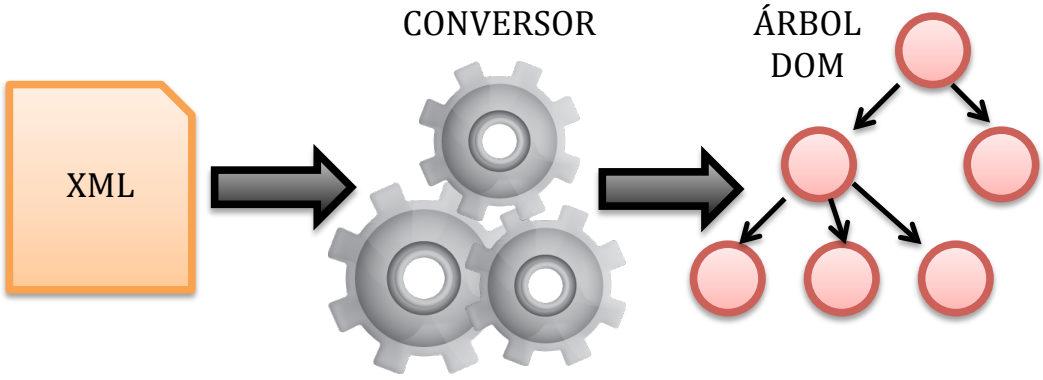

Figura 13: diagrama conversión XML - DOM

La conversión de la estructura XML a DOM (Document Object Model) para poder trabajar con el contenido se realiza en las siguientes líneas:

```
DOMTree = xml.dom.minidom.parse(xmlFile)
collection = DOMTree.documentElement
```

En este punto, tenemos todos los elementos del documento XML en una estructura de datos (collection), a través de la cual ya podemos trabajar.

Para poder explicar la forma en que se procesa el fichero XML, es importante conocer el formato en el que se encuentran definidos los dispositivos y las etiquetas utilizadas.

El fragmento de código que encontramos en el Anexo A, sería el correspondiente a un simulador en el que se han definido dos dispositivos, uno de tipo actuador y otro de tipo sensor.

El código resaltado en rojo pertenece a la definición del simulador mientras que el verde pertenece a un dispositivo sensor de temperatura y el azul a uno actuador de puerta.

Como se puede observar, los dispositivos sensores tienen más atributos definidos que los actuadores. Tal y como se ha definido el

fichero de entrada, se deben de tener en cuenta 2 etiquetas fundamentales utilizadas para definir los dispositivos:

- **SimulatedDevice:** indica la definición de un nuevo dispositivo. El simulador contendrá tantos dispositivos como etiquetas con este nombre aparezcan.
- **SimultedDeviceProperty:** indica una propiedad asignada a un dispositivo. Estas etiquetas solo aparecerán relacionadas con dispositivos de tipo sensor.

De esta forma, podemos recuperar todos los dispositivos definidos en el XML de una forma rápida y sencilla mediante esta línea de código:

```
simulatedDevices = collection.getElementsByTagName("SimulatedDevice")
```

Posteriormente, los atributos asociados a cada dispositivo se validan primero si existen y luego recuperando su valor con estas dos funciones:

- hasAttribute()
- getAttribute()

Quedando el código para recuperar el atributo “tipo de dispositivo” de la siguiente manera:

```
simulatedDevice.hasAttribute("type")
type = simulatedDevice.getAttribute("type")
```

En el proceso de lectura de los dispositivos desde el fichero de entrada, se realiza la validación de que no se registran dispositivos iguales. La solución que hemos propuesto era comprobar mediante dos funciones implementadas en el módulo **Simulator.py**:

- actuatorRepeated(self,id):
- sensorRepeated(self,id):

El criterio que se ha decidido establecer para determinar si un dispositivo es igual que otro ha sido por medio del atributo “identificador”. Estas dos funciones comprueban si alguno de los

dispositivos registrados previamente tienen el mismo identificador que el dispositivo que se pretende registrar en este momento.

Por último, desde este módulo se inicializan y cierran los ficheros que se generan al mismo tiempo que se lee el fichero de entrada y que representan los recursos REST y los ficheros necesarios para inicializar el servidor facilitado por el microframework Flask.

Estos ficheros se generan mediante una serie de funciones que se encuentran en el módulo Simulator.py.

5.3 El módulo Simulator

En la arquitectura de la solución que hemos propuesto, el módulo **Simulator.py** es la clase central desde la cual se realizan las tareas más importantes en el flujo principal de la aplicación.

En esta clase se llevan a cabo las siguientes funciones:

- Registro de los dispositivos definidos en el fichero de entrada
- Escritura del contenido de los ficheros de la API REST
- Generación de los cambios en el contexto de los dispositivos

El funcionamiento de esta clase se basa en las funciones que se definen en ella y que son llamadas desde el módulo Parser.py. Durante el diseño del simulador, se pensó que era óptimo tener toda esta funcionalidad en esta clase debido a que son funciones que afectan directamente al comportamiento del simulador.

El registro de los dispositivos que se obtienen del fichero de entrada se lleva a cabo mediante 2 funciones definidas:

- registerActuator(self,id,name,subtype,state,simName):
- registerSensor(self,id,name,subtype,state,mode,interval,minValue,maxValue,simName):

Estas dos funciones son las encargadas de crear en memoria los objetos asociados a los dispositivos, una vez se realiza la comprobación de que no son dispositivos repetidos, y de la obtención de todos sus atributos desde el fichero XML.

Una vez los objetos que representan a los dispositivos están creados en memoria, se almacenan en dos listas para llevar un control sobre la cantidad de dispositivos registrados y hacer que éstos sean fácilmente accesibles desde otros métodos.

El siguiente código representaría el registro de un dispositivo actuador:

```
if (subtype == "ACTUATOR") or (subtype == " ACTUATOR "):
                    ACTUATOR = ACTUATOR (id,name,state)
                    Simulator.actuators.append(ACTUATOR)
```

La siguiente función que realiza este módulo es la de generar el contenido de los ficheros donde se encuentran definidos los recursos REST correspondientes a cada dispositivo, además de todos los ficheros de la API REST basada en el microframework Flask necesarios para hacer que nuestros dispositivos sean accesibles desde la red.

5.3.1 Generación automática de ficheros

Como parte de la solución propuesta, lo que hemos hecho ha sido que los dispositivos sean accesibles a través de la red mediante unos servicios implementados utilizando REST, a través de los cuales podemos obtener información acerca del simulador y de los dispositivos que tiene asociados.

Para poder llevar a cabo estas tareas, es necesario el uso del microframework Flask y de las clases necesarias para implementar la API REST. Todos los ficheros necesarios para poner en funcionamiento el servidor local y poder acceder a los dispositivos mediante una serie de

direcciones web se generarán automáticamente desde la clase **Simulator.py**

Mediante la arquitectura que se ha planteado, el usuario será capaz de generar todos los ficheros necesarios de forma automática para poder implementar REST, utilizando una arquitectura basada en el microframework Flask.

Los ficheros resultantes generados automáticamente son:

- Output.py
- simulatorDevices.py
- deviceDetails.py
- Un fichero por cada subtipo de dispositivo registrado

Las funciones que generan el contenido de estos ficheros son:

- writeOutput
- writeSimulatorDevices
- writeDeviceDetails

Estos ficheros se generan automáticamente en el directorio junto con el resto de módulos definidos durante el desarrollo y de forma transparente al usuario.

Con los ficheros generados, debemos de explicar su contenido y cuáles son sus funciones.

Output.py: Es el fichero de configuración básico de Flask. Contiene la definición de la API REST que vamos a publicar junto con los recursos. Además, contiene las llamadas a funciones necesarias para volver a registrar en memoria los dispositivos detectados por el módulo Parser.py. Esto es debido, a que una vez finaliza el hilo de su ejecución, se pierden las referencias a estos objetos y debemos de volver a darlos de alta con la misma información.

Básicamente, esta clase contiene toda la información detectada del fichero XML para dar de alta un nuevo simulador en memoria a partir de cual, poder obtener la información haciendo uso de los recursos REST.

Para entender un poco el contenido y la sintaxis de este fichero, la siguiente línea muestra como se vuelve a crear el simulador con el registro de un nuevo dispositivo:

```
sim=Simulator("SIM_NAME")
sim.registerActuator("DEVICE_ID","DEVICE_NAME","SUBTYPE","S
TATE","SIM_NAME")
```

simulatorDevices.py: En este fichero se define la clase simulatorDevices, la cual realiza un función básica que consiste en mostrar todos los dispositivos que están registrados en el simulador, clasificándolos por el subtipo al que pertenecen y mostrando únicamente su identificador. En este caso, la clase simulatorDevices es uno de los recursos REST que exponemos a través de la red y solamente tiene implementado el método GET.

deviceDetails.py: en este módulo se encuentra la clase deviceDetails, la cual es la más compleja de implementar de un recurso REST en el simulador. La función que realiza esta clase es muy importante, ya que devuelve todos detalles de un dispositivo dado a través de la URL además de su estado en ese momento y ofrece la posibilidad de actualizar su estado.

Este recurso REST tiene implementados los métodos GET y PUT. Mediante el método GET, recuperamos el estado actual del dispositivo y mediante el método PUT podemos cambiar el estado del dispositivo.

Debemos de indicar, que el método GET podrá ser utilizado tanto con dispositivos actuadores como sensores, mientras que por otro lado el método PUT solamente podrá ser utilizado sobre dispositivos actuadores.

Una vez hemos explicado los primeros componentes del simulador, el siguiente diagrama muestra la relación que existe entre ellos y la forma que tienen de integrarse con más detalle:

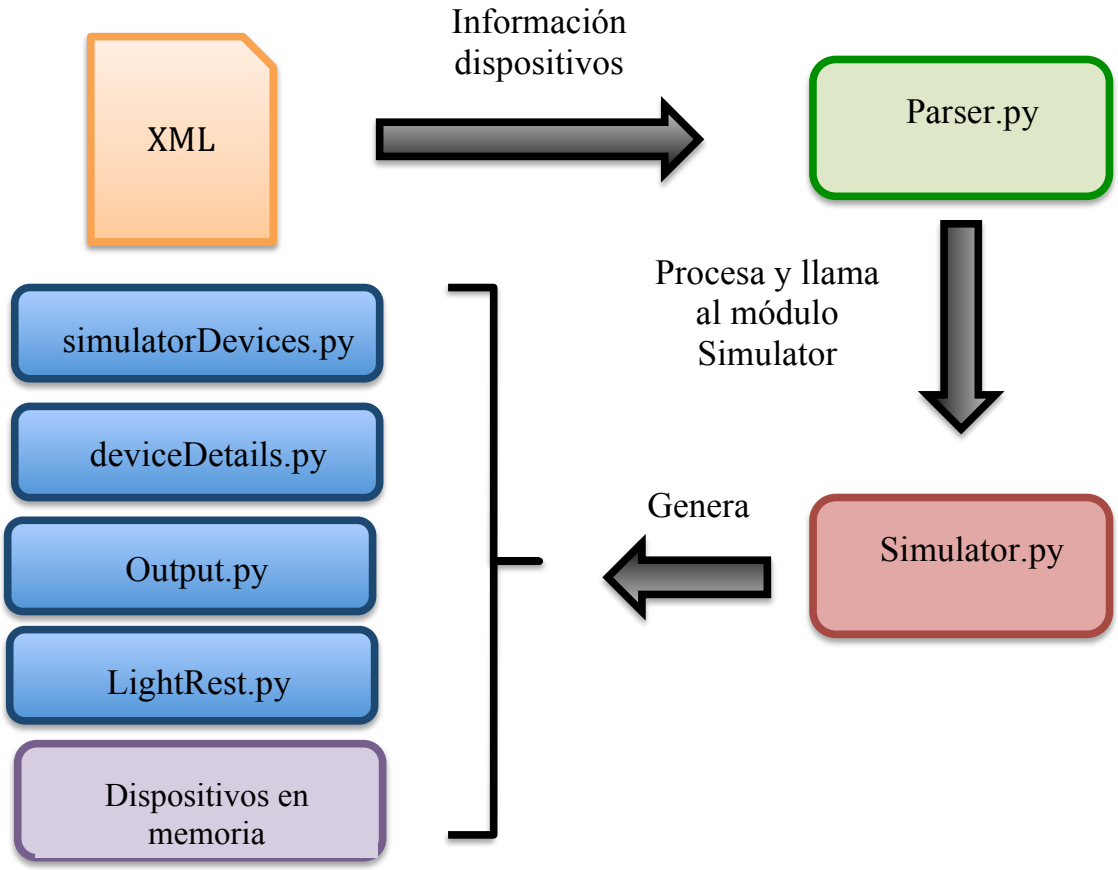

Figura 14: Integración módulos Parser-Simulator

5.4 Direccionamiento recursos REST

En el apartado anterior hemos explicado cómo generamos los recursos REST necesarios, y el contenido de los ficheros resultantes.

A lo largo de este apartado, explicaremos cómo hemos implementado las direcciones de los recursos REST y la forma en que éstos han sido dados de alta mediante el microframework Flask para ser expuestas a través de un servidor local.

Utilizando las bases explicadas anteriormente a lo largo del apartado 3.3, se han decidido exponer estos recursos mediante una serie de URL's que tienen las siguientes estructuras:

[gateway]/simulator/[nombre simulador]
Mostrará todos los dispositivos registrados en el simulador clasificándolos por actuadores y sensores.

[gateway]/simulator/[nombre simulador]/[subtipo de dispositivo]
Mostrará todos los dispositivos que pertenecen al subtipo especificado en la URL.

[gateway]/simulator/[nombre simulador]/[subtipo de dispositivo]/[dispositivo]
Mostrará el estado actual del dispositivo en el momento de hacer esta llamada junto con información adicional acerca del dispositivo indicado mediante un identificador.

En total, tenemos 3 tipos de recursos REST que debemos de dar de alta, los cuales están gestionados por una serie de clases generadas automáticamente.

Hay que recordar, que se genera un recurso REST por cada subtipo de dispositivo que se registra, por lo tanto el número de recursos REST asociados a los subtipos podrá ir de 1 a 14.

La siguiente tabla, muestra la relación entre las clases REST, su dirección web y los métodos que acepta:

CLASE	DIRECCIÓN	MÉTODOS
simulatorDevices	[gateway]/simulator/[nombre simulador]	GET, HEAD, OPTIONS
deviceDetails	[gateway]/simulator/[nombre simulador]/[subtipo de dispositivo]/[dispositivo]	GET, PUT, HEAD, OPTIONS

[SUBTIPO]rest	[gateway]/simulator/[nombre simulador]/[subtipo de dispositivo]	GET, HEAD, OPTIONS

Tabla 9: Tabla de recursos REST

Como podemos observar, no todas las clases implementan todos los métodos HTTP disponibles. Debido a la función que tienen que desempeñar, para la clase simulatorDevices y [SUBTIPO]rest, solamente implementaremos el método GET ya que simplemente mostrarán información sobre el simulador y los dispositivos.

Por otro lado, la clase deviceDetails sí que implementa el método GET y PUT, debido a que mediante esta clase se podrá influir sobre el estado de los dispositivos, por lo que se necesitará enviar un bloque de información al servidor para actualizar el estado. Esto solamente es posible con el método PUT de HTTP.

Por lo demás, los métodos HEAD y OPTIONS estarán disponibles para todos las clases.

Para dar de alta estos recursos en la API REST y especificar la clase que gestiona los métodos que éstos admiten, se utilizan estas líneas de código dentro del fichero de configuración de Flask:

```
api.add_resource(RECURSO_REST, '/simulator/SIM_NAME'/[PARAMETROS])
```

5.5 Peticiones y respuestas

En este capítulo explicaremos la solución propuesta para la gestión de las peticiones y respuestas que hemos utilizado en este trabajo.

Una vez los servicios REST están disponibles a través del servidor local, podemos enviar peticiones y recibirlas a través del navegador.

En este punto debemos resaltar que el uso de las tecnologías REST y JSON juega un papel importante. Decidimos utilizar REST por la facilidad a la hora de desarrollar y probar nuestra aplicación desde un simple navegador web, y luego JSON debido a la sencillez y libertad con la que nos permite definir cómo van a ser los mensajes que vamos a intercambiar con el servidor.

Primero de todo, debemos de explicar la diferencia entre lo que es la definición de un recurso (en este caso dispositivos) y lo que es su estado. Ambas informaciones vienen dadas en las respuestas devueltas por el servidor, la diferencia está en la parte del mensaje donde vienen reflejadas.

En el caso de la definición, nos proporciona información sobre los atributos del recurso y de qué cosas nos indican las características que éste posee. Por otro lado, el estado del recurso nos dice el valor que uno o algunos de estos atributos tiene en un momento determinado. Podríamos decir que la definición sería cómo está representado el recurso y el estado sería una imagen concreta del recurso en un momento dado.

Las peticiones se realizan utilizando las direcciones explicadas en el apartado 4.6, donde se indica la dirección y el método que se desea aplicar sobre el recurso REST.

Respecto a las respuestas, es la parte que generamos en nuestro simulador y que más en detalle vamos a explicar.

Para la implementación de las respuestas se ha decidido utilizar la librería "*Response*" disponible en el microframework Flask. La utilización de esta librería es sencilla, ya que solamente debemos de crear un objeto de tipo Response y posteriormente asignar a sus dos atributos los valores que queremos. En concreto un objeto Response tiene dos atributos:

- **headers:** contiene todas las cabeceras de la respuesta
- **data:** contiene los datos que se encuentran en el cuerpo de la respuesta

Las respuestas se generan en los diferentes métodos de las clases que gestionan los recursos REST que hemos dado de alta.

En el caso de la clase **simulatorDevices,** solamente implementa el método GET debido a que el HEAD y el OPTIONS le vienen heredados y no necesitan ser implementados de nuevo.

Para el mensaje de respuesta del método GET, se ha definido un mensaje JSON en el cual se agrupan los identificadores de los dispositivos según el subtipo al que pertenecen. La estructura del mensaje JSON propuesta es la siguiente:

```
{‘devices’:
            {   ‘subtipo’ : [lista identificadores],
                ‘subtipo’:[ lista identificadores],
                .
                .
                ‘subtipo’:[ lista identificadores],
            }
}
```

Como se puede observar, no es más que una etiqueta “device” seguido de pares de datos en las que la clave es el subtipo de los dispositivos y el valor es una lista en la cual se agrupan todos los identificadores de dispositivos del mismo subtipo.

En este caso, no se ha añadido ninguna cabecera y solamente se ha definido el campo “**data”** de la respuesta. Es necesario indicar el tipo de respuesta que se envía desde el servidor, y para ello se indica la propiedad “mimetype”, con lo que la respuesta quedaría implementada de la siguiente forma:

```
response = Response(mimetype='application/json')
res2={}
res2['devices'] = res
response.data = res2
return response
```

En las líneas de código previas vemos como generamos 2 diccionarios anidados, uno perteneciente a la etiqueta ‘devices’ y otro donde hemos ido insertando los subtipos y las listas de dispositivos. Finalmente se le asigna este diccionario al campo **“data”** de la respuesta.

Como se ha indicado en el apartado 4.6, se ha generado un fichero por cada subtipo de dispositivo detectado en el fichero de entrada. Estos ficheros se generan automáticamente y solamente tienen implementado el método GET el cual devuelve solamente los identificadores de los dispositivos registrados. En este caso, la estructura del mensaje JSON que hemos propuesto es la siguiente:

{‘subtipo’ : [lista identificadores] }

Como podemos observar, la forma de definir el mensaje con JSON es muy sencilla, ya que solamente hay que tener en cuenta los diferentes niveles que queremos representar, y saber identificar si queremos añadir valores únicos expresados con los símbolos ‘’ o listas de valores expresadas con los símbolos “[]”.

En este caso, sí que se ha decidido añadir información en la cabecera de la respuesta. En concreto se devuelve el nombre del subtipo con la etiqueta “Device – Subtype”. La siguiente línea de código indica como se implementa este cambio en la respuesta:

response.headers.add(TAG_NAME’,'TAG_VALUE')

Por último, pasamos a explicar la implementación de la respuesta generada por la clase **“deviceDetails”** la cual es la más compleja de los 3 tipos debido a que implementa los métodos GET y PUT. En este caso, el método PUT solamente se podrá aplicar sobre dispositivos de tipo actuador.

En la solución planteada, se decide que para mostrar los detalles de un dispositivo en concreto se necesitan 2 parámetros: el subtipo y el identificador del dispositivo que se desea recuperar. Estos 2 parámetros se recuperan de la dirección web introducida por el usuario en el navegador.

Para que los métodos GET y PUT reconozcan estos dos parámetros, se debe de indicar en la cabecera del método que se van a utilizar:
def get(self,subType,id):
def put(self,subType,id):

Para la implementación del método GET de esta clase, se ha decidido crear un método en cada una de las clases que definen los subtipos de dispositivos. Este método se encarga de generar el estado del dispositivo en un determinado momento, pero utilizando un mensaje JSON. La cabecera del método es la siguiente:

def getJSONState(self):

Este método se encuentra definido en todas y cada una de las clases que definen los subtipos de dispositivos ya que para cada subtipo, es necesario implementar de una forma diferente los estados que puede presentar.

La siguiente tabla muestra la relación entre los subtipos de dispositivos y los posibles estados que se contemplan para cada uno de ellos:

SUBTIPO DE DISPOSITIVO	**ESTADOS**
Luz	Encendida / Apagada
Luz Gradual	Encendida / Apagada + Valor numérico
Puerta	Abierta / Cerrada
Calefacción	Encendida / Apagada
Persianas	Abierta / Cerrada
Alarma de Luz	Encendida / Apagada
Alarma de Sonido	Encendida / Apagada
Sensor Brillo	Valor numérico

Sensor Presión	Valor numérico
Sensor Viento	Valor numérico
Sensor Temperatura	Valor numérico
Sensor Sonido	Valor numérico
Sensor Magnetismo	Detectado / No Detectado
Sensor Movimiento	Detectado / No Detectado

Tabla 10: Estados de los dispositivos

En el **Anexo B**, se puede encontrar la implementación del método getJSONState() para diversos dispositivos diferentes.

Tal y como se puede observar en la tabla, el dispositivo con subtipo Luz Gradual, tiene definido su estado por dos atributos: el primero nos indica si está encendido o apagado, y el segundo nos indica la intensidad con la que la luz está encendida en el caso de estarlo. La implementación de su método getJSONState es algo diferente de los demás dispositivos. Esta implementación se puede encontrar en el **Anexo B**.

Por lo tanto, ante una petición de tipo GET, solamente se deberá de filtrar por el subtipo y el identificador recibidos para obtener los datos del dispositivo requerido y generar una respuesta JSON válida.

En el cuerpo del mensaje de respuesta se devolverá el estado del dispositivo en el momento de la petición, mientras que en la cabecera, se añadirán otra serie de propiedades que conforman la definición del dispositivo. Para ello, se añaden las siguientes cabeceras a la respuesta HTTP en el caso de los dispositivos actuadores y sensores:

CABECERA	VALOR
Device-Id	Identificador actuador
Device-Name	Nombre actuador
Device-Type	Actuador
Device-SubType	Subtipo actuador

Tabla 11: Tabla cabeceras actuadores

Y las siguientes cabeceras en el caso de los dispositivos sensores:

CABECERA	VALOR
Device-Mode	Modo de actualicacióndel estado
Device-Interval	Intervalo de actualización del estado

Tabla 12: Tabla cabeceras sensores

Respecto al cuerpo del mensaje, solamente se debe de asignar al atributo "data" del objeto "Response" el valor del atributo estado del dispositivo, que como acabamos de explicar, ya se encuentra en formato JSON y por lo tanto no debemos de realizar ninguna gestión.

La estructura del mensaje JSON de respuesta propuesto del método GET es el siguiente:

{'state': [{'
VARIABLE_ESTADO:'VALOR_VARIABLE_ESTADO'}]}

A continuación, se detallará la parte de la solución correspondiente a la implementación del método PUT, mediante el cual se cambia el estado de un dispositivo de tipo actuador.

La solución que hemos propuesto consiste en enviar en el cuerpo de la petición un mensaje JSON en el cual se especifica el nuevo estado del dispositivo.

Para asegurarse de que no se intenta aplicar un método PUT sobre un dispositivo de tipo sensor o que el estado especificado en el mensaje no corresponde con uno de los válidos indicados en la anterior tabla, se realizan una serie de validaciones.

La estructura del mensaje JSON que se envía en el cuerpo de la petición es la siguiente:

```
{‘operation’:
        {
                ‘name’:’OPERATION_NAME’,
                ‘arguments’:[{‘name’:’ARGUMENT_NAME’,
                        ‘value’:’VALUE’}]
        }
}
```

Como se puede observar, tenemos 3 etiquetas:

- **operation**: indica el inicio de una operación
- **name**: indica el nombre de la operación que se realiza
- **arguments**: es una lista donde se indica el nombre y el valor del nuevo estado. Este campo solo aparecerá en caso de que la operación invocada requiera argumentos.

Una vez hemos definido los campos que va a tener nuestro mensaje JSON, pasamos a recoger los valores que tiene asignados. Esta función se lleva a cabo utilizando la librería **reqparse,** la cual nos ofrece la funcionalidad que necesitamos y nos permite procesar la petición recibida para obtener los datos que queremos de ella.

Las líneas de código que se encuentran en el **Anexo C** muestran como se procesan estos parámetros.

Como se puede observar, primero obtenemos la lista de argumentos a través de la petición y posteriormente nos quedamos con los valores asociados a la etiquetas **“operation”** y **“name”**.

Una vez tenemos el valor asociado a la etiqueta “name” solamente queda procesar el subtipo y el id que nos viene indicado en la dirección para cambiar el estado del dispositivo actuador.

Un ejemplo de cambio de estado con el método PUT sobre un dispositivo actuador de subtipo “Puerta” se encuentra en el **Anexo D**.

Por otro lado, tal y como hemos explicado anteriormente, el dispositivo con subtipo “Luz Gradual” es un caso especial ya que se debe

de procesar una petición con un mensaje JSON particular. En este caso, debemos de procesar más etiquetas ya que se obtienen más parámetros en el cuerpo del mensaje. El código encargado de procesar un mensaje de este tipo específico se encuentra en el **Anexo E**:

Básicamente se comprueba que el subtipo y el identificador del dispositivo son los adecuados, para después procesar una lista de argumentos y extraer el nombre y el valor de cada uno de ellos. Finalmente estos valores se asignan al estado del dispositivo.

Finalmente, en la parte de las cabeceras del mensaje de respuesta se insertan una serie de datos que definen el dispositivo tal y como ya hemos explicado anteriormente con el método GET. Las cabeceras que corresponden en este caso son las mostradas previamente en la Figura 24.

5.6 Simulando el contexto

A lo largo de este capítulo hemos explicado la solución que hemos propuesto, y cómo hemos implementado todos los componentes de ésta haciendo uso de las tecnologías escogidas.

Una vez tenemos el simulador implementado, con todas las clases y módulos necesarios para realizar las funcionalidades descritas, vamos a explicar la solución que hemos propuesto para el factor más importante del simulador: los cambios en el contexto.

Como se ha explicado en el capítulo inicial, la característica principal que ofrece la solución propuesta es la capacidad de simular entornos en los que los dispositivos cambian de estado en función de eventos externos que se encuentran en el entorno que los rodea.

La forma en que se ha pensado la implementación de estos cambios en el estado de los dispositivos, está basada en la creación de un hilo que se ejecuta junto con el simulador y que se encarga de generar nuevos datos que producen que se modifique el estado de los dispositivos sensores.

Debemos de especificar, que este hilo solamente afectará a los dispositivos de tipo sensor, ya que los dispositivos de tipo actuador cambiarán su estado mediante acciones iniciadas con el usuario utilizando peticiones con el método PUT.

Para la implementación de este hilo se ha decidido la implementación de 2 métodos encargados de su creación e inicialización:

def start(self): es un método encargado de inicializar el simulador y que a su vez enlaza con el método que inicializa el hilo que se ejecutará en segundo plano a la misma vez que el simulador.

def startUpdatingProcedure(self): este método es el que crea el hilo y lo inicializa controlando las veces que debe de refrescarse el estado de un sensor.

Debido a la importancia de estos métodos en el conjunto global de la solución se indicará su implementación completa para explicar su funcionamiento:

```
def start(self):
        print 'Starting Simulator!!!'
        if not self.isRunning:
                self.isRunning=True
                self.startUpdatingProcedure()
```

El método **start** se encarga de inicializar el simulador, tiene un atributo llamado "isRunning" de tipo booleano que inicializa a True cada vez que el simulador se inicia. Además, realiza una llamada a la función **startUpdatingProcedure** para que el hilo comience a ejecutarse en segundo plano.

Este método es llamado cuando se inicia el módulo Flask que inicializa el servidor local, que atiende a las peticiones justo antes del método inicializador del servidor, debido a que si no se realiza de esta forma, existen problemas y conflictos entre los procesos del servidor local y el hilo.

A continuación, la implementación del método **startUpdatingProcedure** es la siguiente:

```
def startUpdatingProcedure(self):
        guardamos tiempo actual

        para cada uno de los sensores:
                si el sensor tiene un modo automático:
                        si el tiempo actual > última actualización + intervalo:
                                        generamos un nuevo estado
                        última notificación = tiempo actual

                si el simulador se está ejecutando:
                        creamos el hilo
                        el hilo se ejecuta en segundo plano
                        inicializamos el hilo
```

Este método, primero captura el momento en el que se inicializa y a continuación realiza una revisión por todos los sensores registrados en el simulador. A continuación, comprueba si el modo de actualización de estos sensores es automático o manual. En el caso de ser manual, no realizaría ninguna acción, pero en el caso de ser automático, comprobaría si ya ha pasado el intervalo definido en el dispositivo que nos indica cada cuánto tiempo se debe de actualizar. Si este tiempo ya ha pasado, se generará un nuevo estado y por último se le asignará al sensor en cuestión.

El hilo que ejecutamos, se mantiene en segundo plano con el simulador generando datos continuamente para los dispositivo.

Como se ha podido observar, para que este método pueda funcionar correctamente, se necesita guardar una serie de datos para saber si el estado de un dispositivo debe de ser actualizado o no. La solución propuesta para ello, ha sido añadir un nuevo atributo en las clases “actuator” y “sensor” donde se definen las diferentes clases de los subtipos disponibles.

El atributo se llama **LastNotificationSecond y** guardará el momento en que se actualizó por última vez el estado del dispositivo sensor.

Por otro lado, debido a que cada subtipo de sensor manejaba una magnitudes distintas, no fue posible añadir un método encargado de generar el nuevo valor que representa al nuevo estado, por lo que se decidió añadir un método en cada clase de cada subtipo.

De esta forma, cada vez que el tiempo que ha transcurrido entre la última actualización y el tiempo actual es mayor que el intervalo que tiene definido el sensor, se llama a la función **generateNewStatus** que se encarga de generar un nuevo valor el cual será guardado en el atributo **"status".**

5.7 Conclusiones

A lo largo de este capítulo se ha realizado una explicación detallada de todos los componentes de la aplicación desarrollada. Se ha mostrado como se ha implementado cada funcionalidad con las tecnologías seleccionadas. Además, se ha demostrado como se ha realizado la integración de todos estos componentes para dar como resultado el simulador.

En el siguiente capítulo, se va a exponer un caso de uso en el cual se lleva a la práctica todas las características y funcionalidades que se han descrito hasta el momento.

6. Caso de uso

6.1 Presentación del caso

A lo largo de este capítulo, se va a exponer un ejemplo de caso de uso en el que las funcionalidades de la aplicación desarrollada, pueden ayudar a facilitar las tareas que se deben de llevar a cabo en la vida real en un proceso de negocio.

Pensemos en una instalación botánica, en la cual se encuentran numerosas especies distintas de plantas, que ocupan una gran extensión de terreno. Este terreno está dividido en diferentes zonas dependiendo de las características que presenta cada especie.

En este caso, debemos de diseñar un sistema basado en una arquitectura *Internet of Things* donde se nos remarca la necesidad de tener controlado en todo momento las condiciones atmosféricas y físicas que rodea a las plantas para evitar problemas en el desarrollo de éstas.

Además, se nos pide que debido a la separación de las plantas y a la gran extensión del terreno que queremos cubrir. Sería muy importante

tener la facilidad de consultar esta información de forma remota, y tener la capacidad de llevar a cabo ciertas acciones para adaptar las condiciones que rodean a cada grupo de plantas. Por ejemplo, algunas de estas acciones, podrían ser encender la luz de una determinada zona, subir las persianas para permitir que entre más luz o encender la calefacción para aumentar la temperatura en las temporadas de invierno para aquellas especies que son sensibles a la temperatura.

En condiciones normales, se realizaría un diseño del sistema y se pasaría a llevar a cabo diversas pruebas con dispositivos físicos para comprobar como responde el sistema a los cambios en el entorno. Por lo tanto, nos veríamos obligados a realizar las pruebas directamente en el terreno, con el coste económico añadido que esto supone (compra de dispositivos, desplazamiento al terreno, instalación del sistema…).

Utilizando la solución propuesta en este TFM, tendremos la capacidad de definir nuestros dispositivos, generar un sistema que simulará los dispositivos y los cambios en el entorno. Además, generará automáticamente una API REST que permitirá gestionar todos estos dispositivos de forma remota.

El primer paso, sería la definición de los dispositivos. En este caso en concreto debido a los requisitos del sistema, necesitaremos dispositivos sensores que puedan sensar las condiciones físicas de cada zona, con lo cual necesitaremos más de un dispositivo de cada subtipo.

El diagrama de la distribución de las plantas en la instalación botánica sería el siguiente:

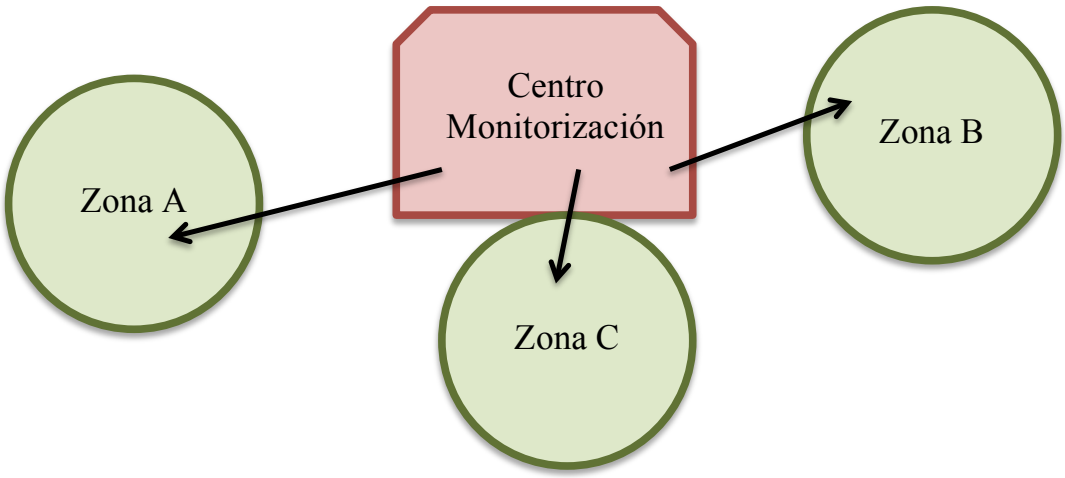

Figura 15: Distribución terrero instalación botánica

En concreto, en este caso nos indican que cada una de las agrupaciones de plantas presenta unas características especiales:

- **Zona A**: Plantas especialmente sensibles a la temperatura
- **Zona B**: Plantas que necesitan estar expuestas a la luz
- **Zona C**: Plantas que se ven especialmente afectadas por la presión atmosférica y la velocidad del viento.

6.2 Selección de los dispositivos

Después de analizar la distribución y necesidades de las plantas, estos serían los dispositivos sensores que se van a necesitar:

DISPOSITIVO	ZONA
Sensor Temperatura 1	A
Sensor Temperatura 2	
Sensor Temperatura 3	
Sensor Temperatura 4	
Sensor Temperatura 5	
Sensor Brillo 1	
Sensor de Presión 1	
Sensor Brillo 2	B
Sensor Brillo 3	
Sensor Brillo 4	
Sensor Temperatura 6	
Sensor de Presión 2	C
Sensor de Presión 3	
Sensor de Presión 4	
Sensor de Viento 1	
Sensor de Viento 2	
Sensor de Viento 3	
Sensor de Viento 4	

Tabla 13: Dispositivos sensores del caso de uso

Viendo los requisitos del escenario propuesto, los dispositivos actuadores que se van a utilizar para favorecer que las condiciones sean las recomendables son los siguientes:

DISPOSITIVO	ZONA
Calefacción 1	A
Calefacción 2	
Calefacción 3	
Luz Gradual 1	B
Luz 1	
Luz Gradual 2	
Luz 2	
Persianas 1	C
Persianas 2	
Persianas 3	
Persianas 4	
Puerta 1	
Puerta 2	

Tabla 14: Dispositivos actuadores del caso de uso

¿Qué buscamos con estos dispositivos actuadores?. Lo que se intenta, es tener cierta capacidad de influencia sobre el entorno para favorecer que las condiciones sean las que queremos. Por ejemplo, en el caso de que la temperatura sea baja en una determinada zona, y la especie de plantas que se encuentra en esa zona sea especialmente sensible, tener la capacidad de poder hacer que aumente la temperatura de forma remota.

Una vez tenemos decididos los dispositivos que vamos a utilizar, debemos de trasladar todas estas especificaciones al fichero XML que sirve de entrada a nuestra aplicación.

Como hemos indicado en apartados anteriores, este proceso se realiza a través de un servicio web de definición de dispositivos llamado Simulator, disponible en galanica.dsic.upv.es.

Por medio de esta aplicación, seremos capaces de indicar todas las características de los dispositivos. En nuestro caso, jugaremos con los atributos de "*intervalo*", "*máximo*" y "*mínimo*" de los sensores. Esto nos permitirá establecer un tiempo de actualización del estado de nuestros sensores en función de la criticidad que le queramos dar a la magnitud física que sensa. También podremos establecer un rango aceptable de valores que nos servirá para poder simular los cambios en el entorno.

La aplicación web nos permite mediante una serie de formularios y vistas, especificar las características de los dispositivos con los que queremos trabajar. Una vez toda esta información está registrada, la aplicación ofrece la opción de exportar todos estos datos a un fichero XML, que como hemos indicado antes, servirá de entrada para nuestro simulador.

De esta forma, el usuario puede especificar todos estos datos a través de una interfaz visual y posteriormente exportar los metadatos de los dispositivos definidos.

Esta característica es muy útil debido a que podremos asignar un intervalo de actualización más bajo a aquellos sensores que nos resulten especialmente interesantes de conocer y otro más bajo para aquellos valores que no sean tan importantes.

6.3 Fichero de entrada

Para todos los sensores, se establecerá un modo de actualización automático para favorecer la simulación del entorno y trabajar con la mayor cantidad de datos generados posibles.

Por lo tanto, el contenido del fichero resultante de la exportación de las especificaciones de los dispositivos, tendrá bloques de información como el siguiente para un dispositivo sensor de temperatura:

<SimulatedDevice deviceID="temp1" name="Temperature1" type="1" subtype="TEMPERATURE_SENSOR" initialState="on">

<SimulatedDeviceProperty property="es.upv.pros.coolSimulator.sensorNotificationMode" value=" es.upv.pros.coolSimulator.sensorNotificationModeAutomatic"/>

<SimulatedDeviceProperty property="es.upv.pros.coolSimulator.sensorNotificationInterval" value="5"/>

<SimulatedDeviceProperty property="es.upv.pros.coolSimulator.sensorNotificationMinValue" value="25"/>

<SimulatedDeviceProperty property="es.upv.pros.coolSimulator.sensorNotificationMaxValue" value="35"/>

</SimulatedDevice>

Por lo que respecta a los dispositivos actuadores, su configuración es algo más sencilla debido a que tienen menos atributos. El contenido del fichero XML para un actuador de subtipo "*Luz*" sería el siguiente:

<SimulatedDevice deviceID="light1" name="Light" type="0" subtype="LIGHT_ACTUATOR" initialState="on"/>

Cuando el fichero de entrada se procesa por el simulador, los dispositivos quedan registrados y se generan objetos en memoria de todos ellos.

6.4 Generación API REST

Como resultado de la ejecución del simulador, obtenemos los ficheros descritos en secciones anteriores, los cuales suponen las clases

necesarias correspondientes a los recursos REST que van a ofrecer la opción de operar de forma remota con nuestros dispositivos.

De esta forma, tendremos un recurso REST por cada subtipo de dispositivo con su método GET implementado. El contenido del fichero generado automáticamente GradualLightRest es el siguiente:

```
class GradualLightRest(Resource):
    def get(self):
        devices = []
        for device in SharedData.simulator.actuators:
            if (device.getSubtype() == "GradualLight"):
                deviceId = device.getId()
                devices.append(deviceId)
        res={}
        res['GradualLight']=devices
        response = Response(mimetype='application/json')
        response.headers.add('Device-Subtype','GradualLight')
        response.data = res
        return response
```

Además, tenemos las clases correspondientes a los recursos que hemos descrito en el capítulo 5:

- simulatorDevices
- DeviceDetails

Observando el contenido del fichero Output.py donde se registran todas estas clases en el servidor local, tendríamos líneas de código como estas:

```
api.add_resource(simulatorDevices, '/simulator/SimJulio')
api.add_resource(HeatingRest,'/simulator/SimJulio/HeatingRest')
```

Como podemos ver, en cada línea se asocia una clase generada a una dirección de acceso.

Cuando ejecutemos el módulo, se registrarán todos los dispositivos y estarán disponibles en memoria para acceder a ellos a través de las direcciones asociadas.

6.5 Operaciones sobre los dispositivos

Una vez los dispositivos ya se encuentran registrados, la API REST ha sido generada automáticamente y se ha lanzado el módulo basado en Flask para inicializar el servidor local, solamente nos quedaría utilizar las operaciones que vienen definidas en la tabla 21 para cada recurso.

Para acceder a la información que queremos, solamente necesitaremos un navegador y un plugin mediante el cual podremos enviar cualquier petición HTTP.

Por ejemplo, en el caso de que queramos comprobar todos los dispositivos que estamos utilizando en nuestra simulación, solo tenemos que realizar una petición GET al recurso **simulatorDevices** para obtener este resultado con todos los dispositivos agrupados por el subtipo correspondiente:

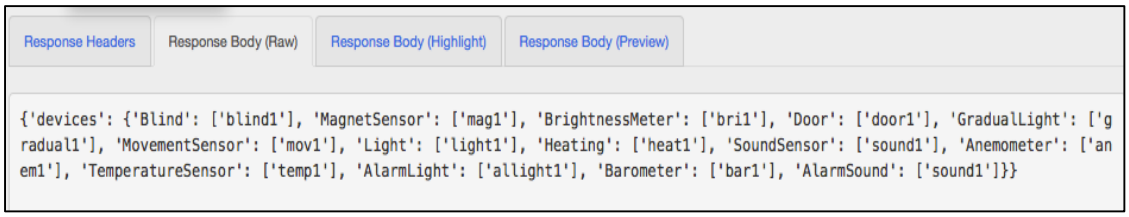

Figura 16: Listado de dispositivos

Por ejemplo, en nuestro caso, si queremos acceder a la temperatura que existe en un punto concreto de la zona A, realizaremos una consulta GET a un sensor de temperatura. La dirección a la que accederíamos es la siguiente:

http://127.0.0.1:5000/simulator/SimJulio/TemperatureSensorRest/temp1

Y el resultado obtenido sería un mensaje como este:

{'state': {'currentTemperature': 29.52}}

En este caso, el cambio de estado del sensor de temperatura se encontrará entre los valores que hemos definido en el XML de entrada e irá variando basándose en el intervalo especificado.

Ahora supongamos que hemos detectado una disminución de la temperatura en una zona que está cubierta por un sensor de temperatura "**temp2**" y queremos hacer que ésta baje para no perjudicar a las plantas de esa zona.

Para ello, utilizaremos un dispositivo actuador de tipo calefacción que se pondrá en marcha y favorecerá el aumento del calor en la zona. Para llevar a cabo esta operación, solamente tenemos que enviar una petición PUT al actuador "**heat1**" a la dirección que se especifica con la siguiente información:

http://127.0.0.1:5000/simulator/SimJulio/HeatingRest/heat1

{"operation":{"name":"switchOn"}}

A lo que se nos devolverá un mensaje con el nuevo estado del dispositivo:

{'state': {'isOn': True}}

De esta forma, estamos simulando todo el entorno con el que los dispositivos físicos se encontrarán cuando estén trabajando en el terreno, y podremos comprobar si hay cualquier problema en el funcionamiento de cambios de estado o notificaciones.

Además del estado, también obtendremos como respuesta una serie de cabeceras HTTP descritas en el capítulo 5, que nos proporcionarán información importante sobre la definición del dispositivo.

En un escenario real, solamente tendríamos que dejar pasar un tiempo y volver a consultar la temperatura de **"temp2"** para comprobar si el aumento de la temperatura es suficiente o debemos de activar otro dispositivo de calefacción.

6.6 Utilización de la herramienta

Una vez ya se ha explicado cómo se aprovecha la funcionalidad que ofrece esta aplicación, vamos a mostrar como ejecutar los procesos que han sido implementados para llegar a este punto.

Debemos de explicar que en el desarrollo de esta propuesta hemos utilizado la herramienta **virtualenv**. Esta herramienta básicamente lo que nos permite, es generar una arquitectura con una copia operativa de python, la cual nos ofrece la posibilidad de eliminar dependencias con cualquier librería asociada a una versión concreta de este lenguaje de programación.

Mediante **virtualenv**, podemos olvidarnos de la versión de python que estemos utilizando. Ya que todas las librerías o características asociadas a la versión del lenguaje python que utilizamos para el desarrollo, estarán disponibles mediante la generación de diferentes "entornos".

Por lo tanto, el primer paso es activar el entorno en el que queremos trabajar e instalar Flask. A partir de este momento, solamente deberemos activar el entorno en el que vamos a trabajar y todas las librerías de Flask y python estarán accesibles.

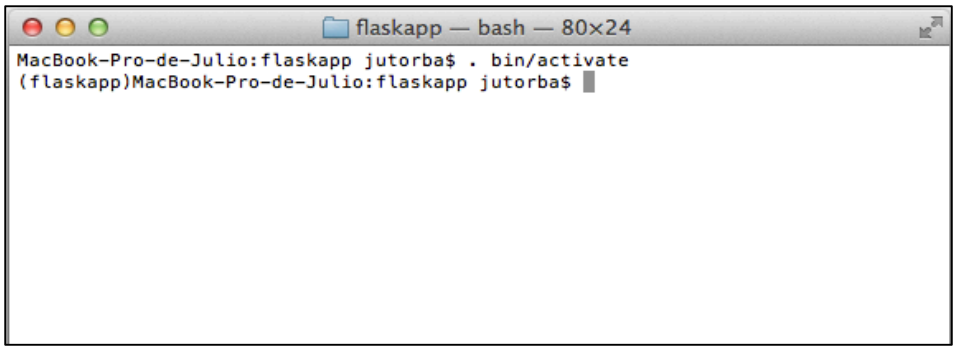

Figura 16: Activación virtualenv

Como podemos observar, una vez introducido el comando, la línea de comandos cambia y nos indica que ya estamos trabajando con Flask. Todas los ficheros que componen las librerías disponibles en el entorno están contenidas en la carpeta **bin**.

A continuación empezamos ejecutando el módulo Parser.py explicado con detalle en el capítulo anterior. Para ello, debemos de acceder a la carpeta **app**, donde se encuentran todos los ficheros relacionados con la herramienta.

Precisamente en la carpeta **app** es donde debemos de situar nuestro fichero XML generado y descargado del servicio de definición de dispositivos, junto con los ficheros python que implementan nuestro simulador para que sea posible el acceso a su contenido.

La ejecución del módulo Parser.py se lleva a cabo desde una consola, indicándole como parámetro de entrada el nombre del fichero XML que hemos generado.

La ejecución del módulo, nos devuelve unos mensajes indicándonos que el fichero ha sido leído, junto con el identificador asignado al simulador y el número de dispositivos detectados.

En estos momentos, si comprobamos el contenido de la carpeta **app**, podremos observar como nos aparecerán todos los ficheros generados automáticamente pertenecientes a los recursos REST y a los ficheros necesarios para trabajar con Flask, tal y como se ha descrito en el capítulo 4.

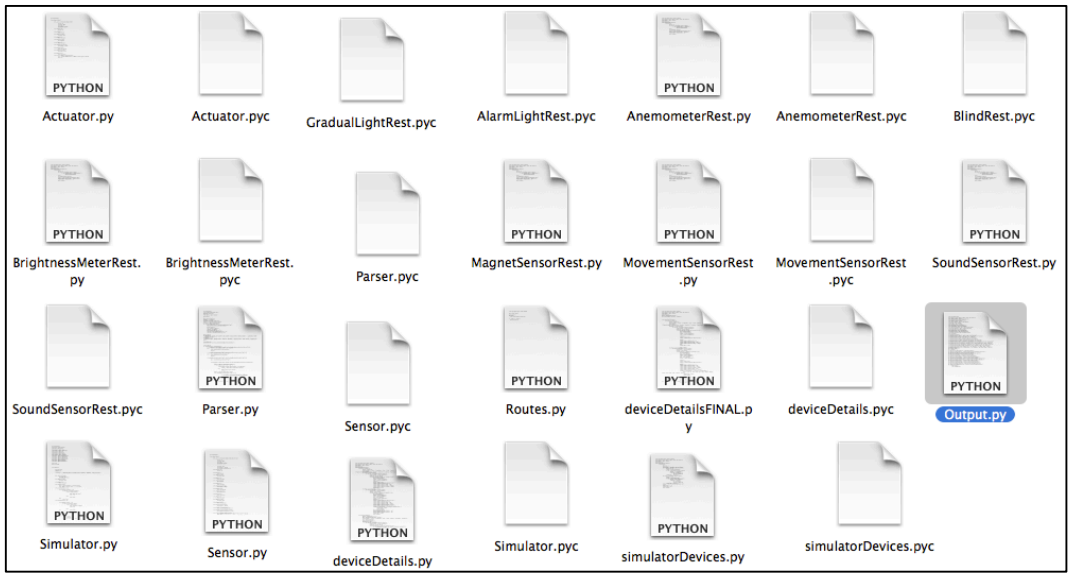

Figura 17: Ficheros generados automáticamente

Una vez ya tenemos todos los ficheros generados, el siguiente paso es ejecutar el módulo Output.py.

Cuando termina la ejecución, ya estamos en disposición de poder acceder a nuestros recursos a través de nuestro navegador web accediendo al servidor local desplegado por la ejecución del módulo Output.py.

De esta forma, mediante un plugin como el mencionado en la sección anterior, ya podríamos acceder a toda la funcionalidad de la aplicación.

6.7 Conclusiones

En el presente capítulo, hemos visto un caso práctico de cómo la herramienta que hemos desarrollado puede utilizarse en un caso real. Se ha llevado a cabo un flujo básico de la aplicación, cumpliendo con unos requisitos expuestos para conseguir obtener una serie de ventajas detalladas a lo largo del capítulo. Además, se han dado los pasos necesarios para la utilización de la herramienta por parte de cualquier usuario.

Se ha demostrado cuáles son las utilidades que ofrece la solución propuesta y cómo ponerlas en práctica.

En el capítulo siguiente, y como cierre de este TFM. Se van a exponer las conclusiones finales acerca de este trabajo exponiendo cual ha sido su principal contribución en el campo en el que se enmarca.

7. Conclusiones y trabajo futuro

En este capítulo, vamos a realizar un análisis final de todo el trabajo realizado basándose en las ideas iniciales que llevaron a realizar esta propuesta de Trabajo de Fin de Máster y en los resultados obtenidos, así como las futuras posibles mejoras que podrían implementarse.

7.1 Conclusiones

A lo largo de este documento, se ha presentado una propuesta relacionada con el ámbito de context – awareness, englobado en el sector de *Internet of Things* y sus conceptos.

Como inicio, se ha realizado un análisis del punto en el que se sitúa este TFM y cuáles son las áreas de influencia de las que recoge ideas. Además, se ha realizado una revisión del momento en el que surgen estos nuevos conceptos y de las necesidades que deben de cubrir.

Dentro de estas nuevas necesidades se ha situado la propuesta en la que se basa este TFM, revisando los objetivos que busca cubrir, las

respuestas que da a alguno de los problemas planteados y las mejoras que propone.

Para la implementación de los conceptos expuestos, se han utilizado tecnologías actuales, tecnologías que se utilizan hoy en día en el ámbito de la ingeniería informática. Estamos hablando de tecnologías web y conceptos de diseño utilizados para resolver los retos que se nos han ido planteando conforme se avanzaba con el TFM.

Las tecnologías empleadas junto con sus características, ventajas y desventajas, han sido explicadas con detalle al igual que los motivos que han servido para decantarse por ellas para desarrollar esta propuesta. Todos estos puntos han quedado mostrados en el capítulo 3 en profundidad.

Respecto a la propuesta final, todo queda relatado en los capítulos 4 y 5. Donde se lleva a cabo una explicación mucho más técnica de cómo se han llevado a cabo las tareas necesarias para alcanzar los objetivos declarados en el capítulo 1.

Como resultado final, se ha obtenido una herramienta que permite resolver algunos de los problemas que presentan el planteamiento de escenarios inteligentes con gran cantidad de dispositivos que necesitan ser administrados y creados de una manera sencilla y con un bajo coste.

El bajo coste y el ahorro de recursos, es la principal ventaja que ofrece esta solución con respecto a otras. Esta solución intenta trasladar al apartado del software muchas de las tareas que se vienen realizando en el mundo físico para la realización de pruebas y generación de escenarios personalizados, con el consiguiente coste asociado que supone.

La herramienta desarrollada, permite simular un entorno cambiante con un conjunto de dispositivos que han sido definidos siguiendo unas especificaciones concretas que permiten gestionarlos agrupando sus funcionalidades y operaciones de una forma lógica.

Además, no solamente se genera un simulador con una serie de dispositivos, sino que se genera de forma automática una API REST que permite acceder a la información de estos dispositivos a través de la red.

En concreto, a través de un servidor mediante el cual se pueden realizar numerosos tests y poner a prueba la funcionalidad de la herramienta de una forma práctica.

Durante el desarrollo de este Trabajo de Fin de Máster, se han combinado una serie de tecnologías que correctamente integradas han conseguido alcanzar los objetivos marcados.

En el proceso de realización de este TFM, se han utilizado conceptos y tecnologías novedosas para mí, las cuales han contribuido a ampliar los conocimientos que poseía sobre las tecnologías web y las arquitecturas enfocadas a *Internet of Things*.

Además, durante la investigación de los conceptos e ideas relacionadas con domótica y *context-awareness*, he conseguido obtener una visión diferente de cómo plantear una arquitectura destinada a plataformas web.

7.2 Contribuciones

Como bien se ha indicado en líneas anteriores, la principal contribución de este TFM es la propuesta de una solución que permite a los usuarios configurar y simular entornos basados en arquitecturas *Internet of Things*.

El usuario es capaz de definir aquellos dispositivos que van a componer su escenario de una completa lista, en la cual se ven reflejados la mayoría de los dispositivos base que podemos necesitar. Unos funcionan a partir de eventos que inicializan sus funciones y otros son sensores del entorno físico.

No solamente se permite elegir los elementos del escenario libremente y en la cantidad que se quiera, sino que también se ha implementado la principal característica que define a esta herramienta. La capacidad de simular cambios en el entorno, los cuales suponen un cambio en el estado de los dispositivos sensores. Se trata de generar una entrada continua de datos para estos dispositivos que son capaces de interpretarlos y cambiar en función de estas entradas.

Esta característica supone una gran contribución, ya que se elimina la necesidad de componentes físicos o de hardware en las etapas iniciales de diseño y planteamiento de los escenarios, lo que supone una gran libertad a la hora de personalizar los entornos en los que queremos trabajar y probar nuestros dispositivos. Además del ahorro que supone al no tener que dedicar parte del presupuesto a la compra de hardware.

Como contribución final, toda esta información está disponible a través de internet mediante la generación automática de una API REST, la cual permite acceder a los estados actuales de todos estos componentes de una forma remota, además de dar la posibilidad de poder actuar sobre ellos.

Como característica supone una gran mejora, ya que en entornos de una gran extensión donde el número de dispositivos es muy alto, la capacidad de gestionar todos estos componentes de una forma remota supone una gran ventaja y facilita el trabajo.

7.3 Trabajo Futuro

Al igual que muchos otros trabajos, esta propuesta no está cerrada y pueden realizarse numerosas mejoras en el futuro.

A continuación se detallan algunas de las mejoras que podrían resultar interesantes de cara a trabajar en futuras propuestas:

Implementación de una capa de persistencia: Una mejora de esta aplicación podría ser la utilización de un sistema de bases de datos donde los dispositivos se almacenasen de forma permanente y se pudiese trabajar con su información a través de la API REST desarrollada. De esta forma, se evitaría tener que tener en memoria el simulador con todos sus dispositivos. Una base de datos, mejoraría la capacidad de almacenaje de objetos y permitiría una mayor estabilidad de la información, de una manera independiente.

Además, resultaría especialmente interesante y útil el poder tener la opción de que los estados de los dispositivos se almacenasen de forma permanente. De esta forma antes posibles problemas de caídas del sistema

o fallos en la ejecución, no se perderían los datos, ya que ahora mismo toda esta información se encuentra almacenada en memoria.

Desarrollo de una interfaz: Actualmente, algunos de los procesos que se llevan a cabo en el simulador, deben de ser ejecutados directamente desde un terminal. Sería una buena opción el poder tener una interfaz que permitiese al usuario utilizar las funcionalidades que ofrece el simulador a través de un medio más visual e intuitivo.

Compatibilidad con Arduino: Arduino es una plataforma open source de prototipado de dispositivos electrónicos. Una posible mejora sería establecer una integración con esta arquitectura que permitiese la implementación de estos dispositivos con sus definiciones asociadas en hardware.

Arduino proporciona una solución económica para poner en práctica aquello que hemos definido en la parte software. Nos permitiría detectar en una fase temprana, posibles problemas presentes en el mundo físico y que mediante el desarrollo prototipos son detectados con mayor facilidad.

Especialmente resultaría útil esta mejora, debido a que la mayoría de los dispositivos que componen nuestros escenarios, muy posiblemente se conectarán de forma inalámbrica. Las ondas que emiten para comunicarse entre sí, puedes verse afectadas por elementos físicos que pueden llegar a bloquear estas ondas. Por lo tanto, un prototipo físico podría darnos una idea de cómo influye el entorno en el funcionamiento de estos dispositivos.

8. Bibliografía

[1] M. Weiser, R.Gold, J.S. Brown (1999), The origins of ubiquitous computing research at PARC in the late 1980s

[2] W. Keith Edwards and Rebecca E. Grinter (2001), At Home with Ubiquitous Computing: Seven Challenges

[3] Telefónica: http://smartcity-telefonica.com/?p=373

[4] Miquel Martin and Petteri Nurmi. A generic large scale simulator for ubiquitous computing. In Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, 2006 (MobiQuitous 2006), San Jose, California, USA, July 2006. IEEE Computer Society

[5] Milton R. Mazzarri S. (2011), http://www.slideshare.net/doknos/qu-es-python

[6] C. C. Celia, F. I. Lara, C. R. Valentín, G. N. Amanda (2007-2008), Python,http://www.it.uc3m.es/spickin/docencia/comsoft/presentations/spanish/doc/Python.pdf

[7] LeonardDavidJeromeJacob, RichardsonHeinemeier , HanssonLouvelKaplan-Moss (2007),O'Reilly –RESTful WebServices

[8] Documentación oficial JSON*:* json.org/

[9] Pedro J. Molina, Ismael Torres, Oscar P*ƒ*astor (2002), Patrones de Interfaz de Usuario para la exploración orientada a objetos.

Anexo A

El siguiente código muestra el contenido del fichero XML de definición de dispositivos. Era importante mostrar el uso de las diferentes etiquetas detallas en anteriores apartados, y de esta forma ver como es la estructura, y poder mostrar con un ejemplo práctico de como se organiza la información que constituye la definición de cada dispositivo.

En este ejemplo de código, se puede ver como la parte resaltada en rojo especifica la información que define al simulador **"SimJulio"** y posteriormente como en color verde se resalta la definición de un dispositivo sensor de temperatura con identificador **"temp1"**, junto con todas sus propiedades.

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulator name="SimJulio"
tunnelURL="http://galanica.dsic.upv.es/simulator/simjulio">

    <SimulatedDevice deviceID="temp1" name="temperature" type="1"
    subtype="TEMPERATURE_SENSOR" initialState="on">
```

```
    <SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationMode"
value="es.upv.pros.coolSimulator.sensorNotificationModeAutomatic"/>
    <SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationInterval"
value="3"/>
    <SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationMinValue"
value="1"/>
    <SimulatedDeviceProperty
property="es.upv.pros.coolSimulator.sensorNotificationMaxValue"
value="2"/>
  </SimulatedDevice>

<SimulatedDevice deviceID="door1" name="door1" type="0"
subtype="DOOR_ACTUATOR" initialState="on"/>

</Simulator
```

Anexo B

El objetivo de este anexo era mostrar varios ejemplos con diferentes tipos de dispositivos (Actuadores y sensores) de implementación del método **getJSON()**, mediante el cual se obtiene el estado del dispositivo en dicho formato. Este método es muy importante debido a que es el método al que se le consulta el estado del dispositivo a través de las peticiones que gestionan los recursos REST.

El poder ver ejemplos de cómo se ha implementado este método con diferentes dispositivos, muestra como se ha tenido que personalizar el método teniendo en cuenta las características propias de cada uno de ellos.

Ejemplo implementación del método getJSONState() para un dispositivo de subtipo "**Luz**":

```
def getJSONState(self):
        res={}
        isOn=False
        if self.getState()=='on':
            isOn=True
        res['isOn']=isOn
        return res
```

Ejemplo implementación del método getJSONState() para un dispositivo de subtipo "**Viento**":

```
def getJSONState(self):
        res={}
        currentWindSpeed= self.getState()
        res['currentWindSpeed']=currentWindSpeed
        return res
```

Ejemplo implementación del método getJSONState() para un dispositivo de subtipo "**Luz Gradual**":

```
def getJSONState(self):
        res={}
        isOn=False
        if int(self.getState())>0:
            isOn=True
        res['currentIntensity']=int(self.getState())
        res['isOn']=isOn
        return res
```

Anexo C

En este anexo, se busca mostrar como se gestiona un mensaje en formato JSON, y como se extrae la información de la respuesta basándose en los campos que contiene.

Se busca mostrar como se utiliza la librería **reqparse** para procesar el mensaje JSON que recibe la clase del recurso REST. Este mensaje viene asociado con una petición HTTP de tipo PUT, y contiene la información organizada por una serie de etiquetas, las cuales utilizamos para recuperar los valores que tiene asociados, y posteriormente operar con ellos.

```
parser = reqparse.RequestParser()
args = parser.parse_args()
operation=args['operation']
name=operation['name']
```

Anexo D

En estas líneas de código se muestra como se establece el estado de un dispositivo actuador de tipo "**Puerta**".

Primero se realiza la comprobación de que el parámetro recibido a través de la petición sea el nombre de un subtipo concreto. Esto es debido a que tal y como hemos explicado anteriormente, cada subtipo de dispositivo permite realizar una serie de operaciones asociadas a sus características.

En este caso, el dispositivo "Puerta" permite las operaciones **"Abrir"** y **"Cerrar"**. Dependiendo del valor de la operación que se recibe a través del método PUT, se cambia el estado del dispositivo a este nuevo valor.

```
elif ((actuatorSubtype == 'Door') and (name in {'open','close'})):
        if (name == 'open'):
                actuator.setState('open')
        else:
                actuator.setState('close')
```

Anexo E

Este anexo se ha decidido añadir debido a que muestra como se gestiona un mensaje JSON de una petición PUT que cambia el estado de un dispositivo actuador de subtipo "**Luz Gradual**".

La decisión de añadir un anexo con este código reside en que el mensaje JSON es el más complejo que gestiona la aplicación de todos los que reciben los actuadores.

En este caso, el actuador **"Luz Gradual"** recibe una lista de argumentos los cuales hay que procesar para obtener el valor del nuevo estado. A diferencia del resto de actuadores, que reciben solamente una etiqueta con un valor asociado, en este mensaje JSON debemos de gestionar la lista de argumentos mediante un bucle.

```
elif ((actuatorSubtype == 'GradualLight') and (name == 'setIntensity')):
        arguments=operation['arguments']
        for argument in arguments:
                argName=argument['name']
                argValue=argument['value']
                actuator.setState(argValue)
```