MÁSTER EN TECNOLOGÍAS Y APLICACIONES EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

Soluciones de integración en la WoT basadas en microservicios

Curso 2020/2021

Alumno:
Juan Francisco Soler Castaño

Director:
Javier Criado Rodríguez

AGRADECIMIENTOS

A la primera persona que se lo quiero agradecer es a mi tutor, Javier Criado Rodríguez, que sin su ayuda y conocimientos no hubiera sido posible el desarrollo de este proyecto.

A mis padres por haberme proporcionado la mejor educación y lecciones de vida.

En especial a mi padre por haberme enseñado que con esfuerzo, trabajo y constancia todo se consigue, y que en esta vida nadie te regala nada.

En especial a mi madre, por cada día hacerme ver la vida de forma diferente y confiar en mis decisiones.

Por último, gracias al resto de mi familia y amigos por su apoyo constante.

“Es importante hacer hincapié en que nadie puede jamás calificar la separación de clases como “bueno” o “malo” para los múltiples en general. La cuestión es si es más apropiado o menos apropiado para este conjunto de múltiples en este momento de su desarrollo”.

(Twins and multiples)

RESUMEN

Este proyecto se centra en el desarrollo de un sistema capaz de integrar dispositivos de la WoT haciendo uso de Ballerina, probando el sistema en un entorno real haciendo uso de una Raspberry Pi y de diversos sensores y actuadores. Ballerina es un lenguaje de programación y framework llamado a ser referencia debido a que facilita la conexión entre aplicaciones y servicios en todo tipo de escenarios de integración. El trabajo desarrollado cubre desde el estudio de la WoT y las diversas formas de integrar Things en ella, hasta la programación y evaluación del producto generando diferentes escenarios reales.

En primer lugar, se ha realizado un estudio bibliográfico de la WoT, la coreografía y orquestación de microservicios, Ballerina y su ámbito de aplicación en la integración, para llegado el siguiente capítulo realizar la selección de la arquitectura del sistema.

Tras la elección de la arquitectura se realiza una implementación individual de cada caso propuesto en función de la manera en la que interactuemos con las Things: propiedades, eventos y acciones.

Finalmente se ha desarrollado el microservicio Parser, encargado de generar los microservicios de integración a partir de un archivo de configuración donde se indican en primer lugar, las Things de las que leemos propiedades o nos suscribimos a sus eventos, en segundo la lógica que debe seguir el sistema y, por último, las Things consecuentes sobre las que se actúa en caso de cumplirse la expresión lógica deseada.

Para la validación del sistema desarrollado se han implementado diversas Things. Para ello se han creado APIs sobre la Raspberry Pi para un sensor ultrasónico, un sensor de temperatura y humedad, un sensor PIR y un Led.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ABREVIATURAS

WoT: Web Of Things

IoT: Internet Of Things

SO: Sistema Operativo

MQTT: Message Queuing Telemetry Transport

HTTP: Protocolo de transferencia de hipertexto

CAPÍTULO 1. INTRODUCCIÓN

A día de hoy, el exponencial crecimiento de dispositivos IoT es evidente. Al finalizar el año 2020, por primera vez, se alcanzaron más conexiones de IoT (automóviles conectados, dispositivos domésticos inteligentes, equipos industriales conectados, etc.) que conexiones que no son de IoT (teléfonos inteligentes, computadoras portátiles y computadoras de sobremesa). De los aproximadamente 21.700 millones de dispositivos conectados activos en todo el mundo, 11.700 millones (o aproximadamente el 54%) serán conexiones de dispositivos IoT a finales de 2020. Para 2025, se espera que haya más de 30.000 millones de conexiones IoT, casi 4 dispositivos IoT por persona de media (véase Figura 1.1) [1]. Si a todo esto se le suma la aparición de la quinta generación de la telefonía móvil (5G), estamos sin duda alguna ante uno de los más importantes desarrollos tecnológicos actuales y de futuro próximo.

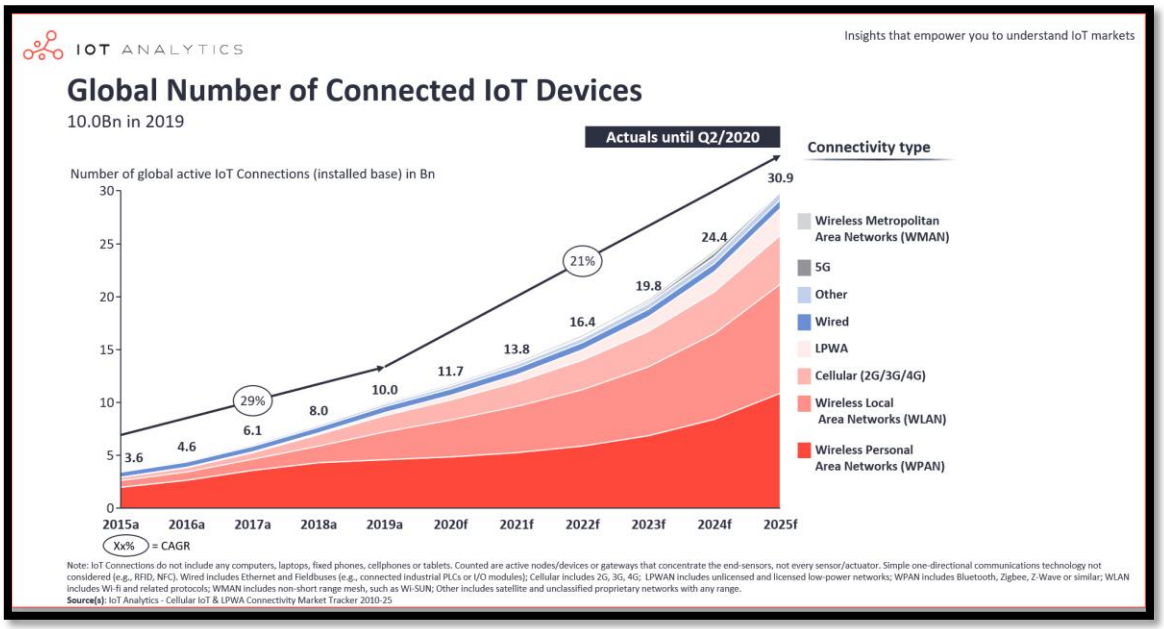


Figura 1.1 Gráfica global dispositivos IoT [1].

Esta creciente evolución de las tecnologías y dispositivos IoT, ligada a las limitaciones ante un posible sistema abierto y a gran escala que combine distintos dispositivos de IoT, ha dado lugar a la aparición de la WoT (Web of Things), definida como un conjunto de estándares del W3C para resolver los problemas de interoperabilidad de diferentes plataformas y dominios de aplicaciones de IoT [2] [3].

La WoT permite así, la interacción de Things y sistemas a través de Interfaces de Programación de Aplicaciones (APIs) exponiendo datos y metadatos de los diferentes dispositivos. Por tanto, la verdadera razón por la que surge la WoT es debido a que la integración de los dispositivos IoT se ve obstaculizada por el hecho de que "muchos dispositivos no hablan el mismo idioma y no pueden intercambiar datos a través de diferentes pasarelas y centros inteligentes" [4].

En este proyecto se pretende aportar soluciones de integración que impliquen la orquestación de diferentes Things de la WoT utilizando microservicios web. Todo ello empleando nuevas herramientas relacionadas con la coreografía y orquestación de servicios como Ballerina.

Para probar las soluciones de integración y mostrar su funcionamiento, se van a crear diferentes Things haciendo uso de una Raspberry Pi e implementando todo lo necesario para simular dispositivos de la WoT accesibles desde sus APIs RESTful. Con ello, podremos generar diferentes escenarios reales donde, haciendo uso de combinaciones lógicas de las propiedades y eventos de las Things, se prueben y validaren las soluciones propuestas.

De esta forma se pretende crear un sistema para gestionar Things de la WoT al que, básicamente, se le pase como entrada una o varias Things denominadas antecedentes y, en función de la lógica deseada (empleando operadores lógicos AND/OR), actúe con la llamada a otras Things denominadas consecuentes.

Un ejemplo claro sobre lo que se pretende desarrollar sería el siguiente. Imaginemos un usuario que dispone de dos dispositivos IoT: un sensor de luz y una bombilla. Éstos, a su vez, están implementados en la WoT. El usuario desea encender la bombilla siempre y cuando el sensor de luz alcance un valor establecido. Para ello, el sistema debe ser capaz de obtener y valorar el antecedente (sensor de luz) y, si se cumple lo especificado, actuar con la llamada al consecuente (encender la bombilla).

1.2 Motivación

La razón principal por la que he elegido el desarrollo de este proyecto, es para mostrar todo lo aprendido a lo largo del máster, sobre todo en lo relacionado a la rama específica del mismo, la rama de desarrollo web/móvil y en concreto al desarrollo e implementación de servicios web y de APIs. De esta forma, pretendo utilizar las diferentes herramientas y tecnologías vistas en el máster y que más interés han causado en mí como son SpringBoot, Node.js o Express, ligadas a la investigación y uso de otras como Ballerina.

Por otro lado, el creciente auge de los dispositivos del *Internet Of Things* y la aparición de la *Web Of Things*, es un motivo más que notable para el desarrollo del proyecto, creando así, una línea de trabajo actual y de futuro, ligada a uno de los sectores punteros de la tecnología e informática.

Ya que cada vez tenemos más dispositivos inteligentes conectados a internet, surge la motivación de poder gestionarlos e integrarlos, creando un sistema que en función de la lógica deseada pueda por ejemplo encender/apagar algún dispositivo, cambiar el valor de alguna de sus propiedades o modificar alguno de sus parámetros de configuración. Supongamos que tiene en casa un sensor de luminosidad y además sus persianas funcionan de manera inteligente y se pueden subir y bajar a través de una App. Imaginémonos ahora que queremos subir o bajar la persiana a la salida y puesta del sol. Este sería un ejemplo

claro y motivacional en el desarrollo del presente trabajo, que pretende crear un sistema que abarque tantos ámbitos y sectores como podamos imaginar.

Por último, y ligado a la labor docente que desarrollo actualmente, me resulta de gran utilidad abordarme en el mundo del IoT y la WoT para poner en práctica lo aprendido en asignaturas como Programación y Computación o Robótica a nivel de E.S.O. y Bachillerato.

1.3 Objetivos

Este trabajo fin de máster tiene como principal objetivo el desarrollar una arquitectura basada en microservicios que permita la integración de diferentes Things de la WoT.

Para desglosar y planificar mejor el trabajo, el objetivo principal se ha dividido en una serie de subobjetivos que se detallan a continuación:

- Realizar un estudio del estado del arte referente a la WoT, a los microservicios web y a la coreografía y orquestación de las Things en la misma, haciendo especial hincapié en arquitecturas de microservicios.
- Realizar un autoaprendizaje de las tecnologías y lenguajes que lleven a la consecución del objetivo principal del proyecto. Entre ellas: Ballerina, SpringBoot, WSO2, MQTT, etc.
- Estudiar y proponer una o varias arquitecturas de microservicios del sistema a desarrollar que sea lo suficientemente consistente y permita el desarrollo óptimo del proyecto.
- Desarrollar y probar la primera versión de la arquitectura. Se generará un caso simulado y sencillo que permita validar la arquitectura para así poder desarrollar con total seguridad un caso de estudio real.
- Desarrollar un caso de estudio real y validar la integración de diferentes Things en la WoT. Se crearán diferentes Things a través de una Raspberry Pi con sus correspondientes APIs RESTful que permitan acceder a sus propiedades, invocar acciones de ellas y suscribirse a los eventos disponibles gracias a la funcionalidad MQTT que se desarrollará en algunas de ellas.

En concreto, para alcanzar el último objetivo, las Things protagonistas de este trabajo serán un LED, un sensor de ultrasonidos (HC-SR04), un sensor de temperatura y humedad (DHT22) y un sensor PIR. Para comprobar la solución propuesta, se han realizado diferentes pruebas que alternen y combinen estas Things.

De esta manera, la finalidad y objetivo del trabajo no es la de buscar solución a un determinado problema en sí, ya que éste no existe como tal, sino la de aportar una idea y un sistema totalmente innovador en el desarrollo tecnológico actual del IoT y la WoT.

1.4 Cronograma

El proyecto comenzó el día 4 de octubre de 2020 y tenía previsto acabar y ser defendido en el mes de febrero del presente año 2021. Durante el desarrollo de las primeras fases centradas en la investigación y una vez ahondados en las primeras horas dedicadas al desarrollo del producto en sí (a mediados del mes de enero), se comprobó la dificultad de llegar a las fechas previstas con el resultado deseado, razón por la cual se decidió aplazar el trabajo hasta la siguiente convocatoria oficial.

Además, otro de los motivos por los que se ha alargado la duración del trabajo ha sido el de querer llegar un paso más de lo que se tenía previsto a comienzos del mismo, ya que en cada avance que se ha realizado, se han ido definiendo y añadiendo mejoras en el trabajo que eran factibles siempre y cuando se ampliara la duración del mismo.

De esta manera el trabajo tenía previsto acabar en el mes de junio y está dividido en dos fases, siendo el objetivo de la primera fase la investigación y el objetivo de la segunda fase el desarrollo de la propuesta junto a un caso de estudio real.

Teniendo en cuenta la normativa del trabajo de fin de máster, el tiempo que se tenía previsto y que se ha dedicado a la realización de este proyecto ha sido de unas 310 horas. Para ello, se identifican las siguientes fases o tareas a realizar y se reparten de forma estimada las horas entre las fases:

- **Fase 1: Establecimiento de objetivos** (3h). El primer paso de este proyecto es establecer los objetivos de investigación y desarrollo que engloban el trabajo fin de máster.
- **Fase 2: Investigación** (40h). Análisis y estudio pormenorizado del estado del arte en el momento del desarrollo, teniendo en cuenta artículos, libros y todo tipo de material relacionado con los objetivos descritos anteriormente.
- **Fase 3: Elaboración del anteproyecto**. (2h)
- **Fase 4: Análisis** (20h). Estudio del problema de una manera minuciosa, haciendo hincapié en el aprendizaje de las tecnologías a aplicar en el mismo, así como metodologías de desarrollo y arquitecturas propuestas para la gestión de microservicios.
- **Fase 5: Desarrollo** (135h). Diseño e implementación de la arquitectura de microservicios propuesta.
- **Fase 6: Validación en caso real** (70h). Diseño e implementación de un caso real con varios dispositivos de la WoT para probar y validar su integración.
- **Fase 7. Redacción** (40h). Elaboración de la memoria final.

En la siguiente tabla se muestran los tiempos de las distintas fases para así tener una visión aproximada del desarrollo del proyecto.

Fase	Fecha								
	2020					2021			
	OCT	NOV	DIC	ENE	FEB	MAR	ABR	MAY	JUN
Establecimiento de los objetivos	3h								
Investigación	12h	15h	13h						
Elaboración del anteproyecto			2h						
Análisis			7h	13h					
Desarrollo				30h	40h	40h	25h		
Validación en caso real							25h	45h	
Redacción								25h	15h
Total: 310h	15h	15h	22h	43h	40h	40h	50h	70h	15h

Tabla 1. Cronograma.

Como vemos en la tabla anterior, el proyecto se ha alargado cuatro meses más de lo previsto, terminando en el mes de junio del año 2021.

1.5 Estructura del documento

El documento ha sido organizado en una serie de capítulos cuya información es la siguiente:

1. **Introducción**: En el primer capítulo se introduce el proyecto aportando las motivaciones que han llevado al desarrollo de éste y reflejando los objetivos así como la duración del mismo.
2. **Estado del arte**: En este capítulo se describe todo el proceso de investigación llevado a cabo acerca de la Web of Things (WoT) y de la orquestación y coreografía de microservicios en la web.
3. **Objetivos y metodología**. Se detallan en profundidad el objetivo general y los objetivos específicos junto con la metodología seguida.
4. **Herramientas y tecnologías**: Capitulo donde se enumeran y comentan las distintas herramientas y tecnologías que se han utilizado en el desarrollo del sistema.
5. **Solución de Integración**: En este capítulo se muestra como se ha desarrollado todo el sistema capaz de realizar la integración de varias Things de la WoT. Se detallan los procesos llevados a cabo, la arquitectura elegida y los microservicios generados.
6. **Casos prácticos reales**: Capítulo donde se muestra el funcionamiento del sistema creado en varios casos prácticos reales, haciendo uso de una Raspberry Pi.
7. **Conclusión y trabajo futuro**: En este capítulo se habla sobre el conocimiento adquirido al desarrollar el proyecto, se valora el mismo y se marcan unas líneas de trabajo futuro.
8. **Bibliografía**: En este capítulo se muestra toda la bibliografía citada.

CAPÍTULO 2. ESTADO DEL ARTE

En este capítulo se habla de la parte de investigación llevada a cabo sobre la WoT, analizando su arquitectura y uno de sus bloques más importantes denominado Thing Description. Además, ligado al objetivo del trabajo de integrar Things de la WoT, se ha hecho un estudio sobre los microservicios, las soluciones de integración y comunicación existentes y sobre la orquestación y coreografía de servicios web.

2.1 WOT

La Web of Things (WoT) o Web de las Cosas es definida como el conjunto de estándares del W3C que busca contrarrestar la fragmentación del IoT mediante el uso y la extensión de las tecnologías web existentes [5]. Con la provisión estandarizada de metadatos y tecnologías reutilizables construidas por bloques, la WoT permite una fácil integración entre plataformas de IoT y dominio de aplicaciones. En pocas palabras, la WoT puede definirse como aquella capa de abstracción entre el dispositivo IoT y el cliente basada en tecnologías Web.

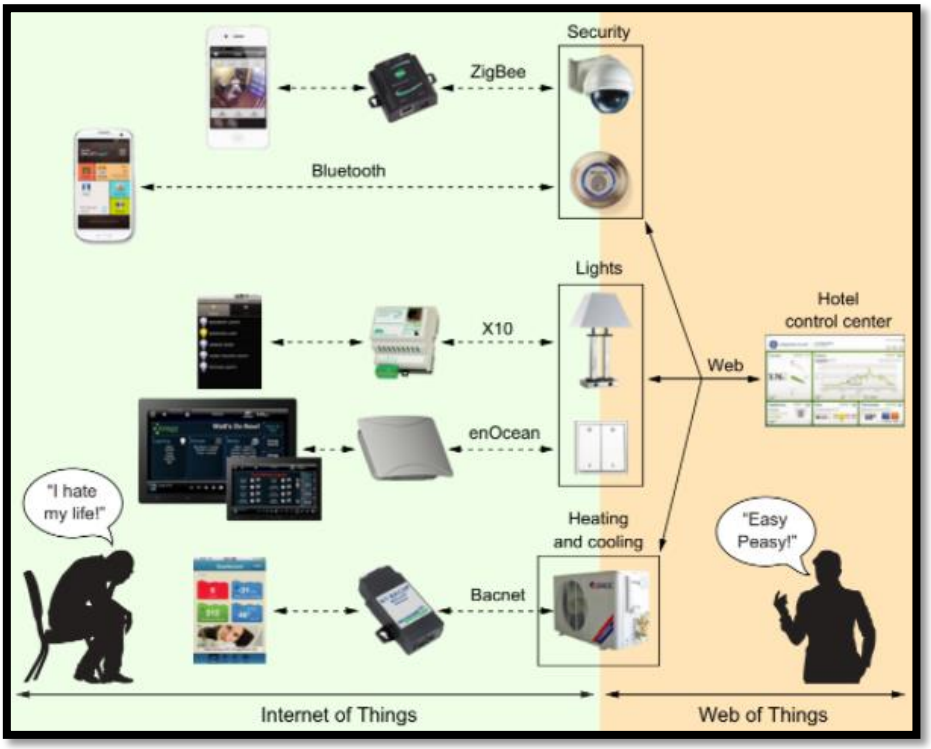


Figura 2.1. WoT vs IoT [6].

Por lo general, en los proyectos típicos de IoT, los programadores se enfrentan a diferentes escenarios tecnológicos en los que intervienen diversos sistemas y servicios de IoT procedentes de diferentes proveedores y fabricantes. Esta diversidad incluye variaciones en los protocolos de comunicación, modelos de datos para el intercambio de información, requisitos de seguridad, etc. Las aplicaciones de IoT generalmente se desarrollan con un gran esfuerzo aplicado a un caso de uso específico y limitado.

Con la Web of Things, aparece un conjunto estandarizado de tecnologías que ayudan a simplificar el desarrollo de las aplicaciones de IoT siguiendo el conocido paradigma web. De esta manera, y como gran ventaja, crece la flexibilidad e interoperabilidad aumentando consigo el potencial comercial que se veía frenado por la fragmentación de IoT.

Además, existen notables ventajas al utilizar la WoT frente al IoT.

En cuanto a facilidad de programación:

- Los protocolos web son más sencillos de aprender que los de IoT.
- Es más sencillo leer y escribir datos en los dispositivos de la WoT.
- Al compartir un mismo tipo de API, la interacción en la WoT es más homogénea.

En cuanto a la utilización de estándares:

- Los protocolos y estándares Web son abiertos y gratuitos. En IoT, algunos de ellos no son neutrales y tienen intereses vinculados a compañías.
- HTTP y REST son tecnologías ampliamente usadas y aceptadas para la interacción con sistemas software.

En cuanto a mantenimiento e integración:

- Los protocolos y estándares Web funcionan de forma correcta desde hace tiempo y sus actualizaciones son fácilmente asumibles.
- En IoT, se requieren múltiples protocolos para la implementación de mecanismos de integración: wrappers, conversores, etc.

En cuanto a seguridad y privacidad:

- Los protocolos y estándares Web contemplan mecanismos de seguridad y privacidad como HTTPS, WSS y MQTTS.
- Determinados protocolos IoT requieren una implementación propia de los mecanismos de seguridad.

El principal objetivo de la WoT es el de interactuar sobre los dispositivos IoT haciendo uso de la tecnología web mediante APIs RESTful (véase figura 2.2). De esta manera la interacción con un dispositivo IoT se basa en una abstracción de propiedades, eventos y acciones. Así se consigue un punto común para acceder a los metadatos de un dispositivo IoT, así como conocer qué y cómo se puede acceder a los datos y funciones del dispositivo IoT.

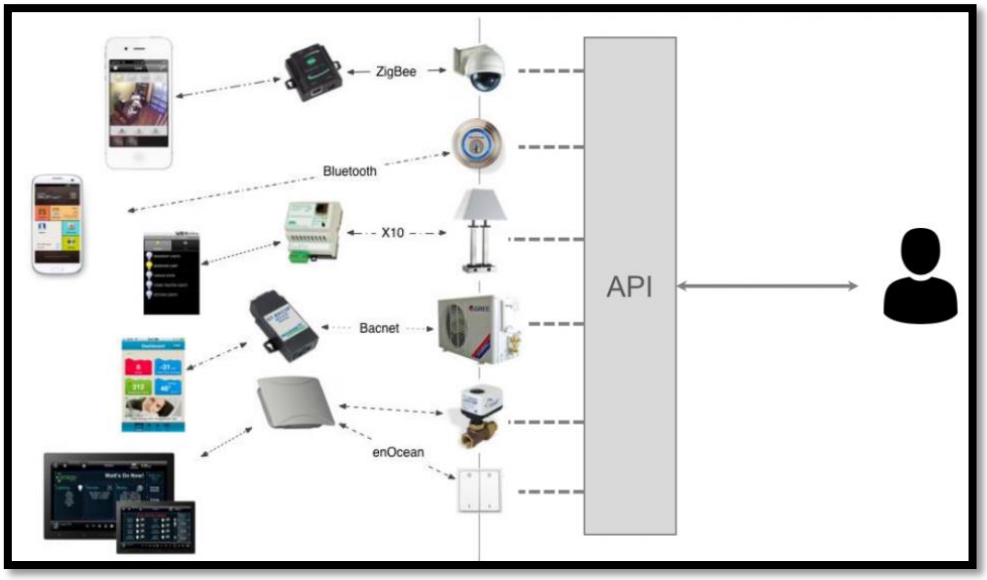


Figura 2.2 WoT API [6].

Pongamos un ejemplo acerca de propiedades, eventos y acciones de una Thing de la WoT. Una propiedad podría ser el valor de un sensor, el estado de una Thing, un parámetro de configuración, etc. Como ejemplo de evento podríamos aportar el aviso de la apertura de una puerta, la presencia de una persona en una habitación o el sobrecalentamiento de una lámpara. Como acciones, activar o detener un proceso (como el de una alarma), imprimir un documento, cerrar una puerta, etc.

La WoT es independiente del protocolo que emplea el dispositivo de IoT y proporciona un mecanismo común para definir cómo se pueden asignar o mapear esos protocolos específicos (MQTT, CoAP, HTTP…) a las propiedades, eventos y acciones de la WoT.

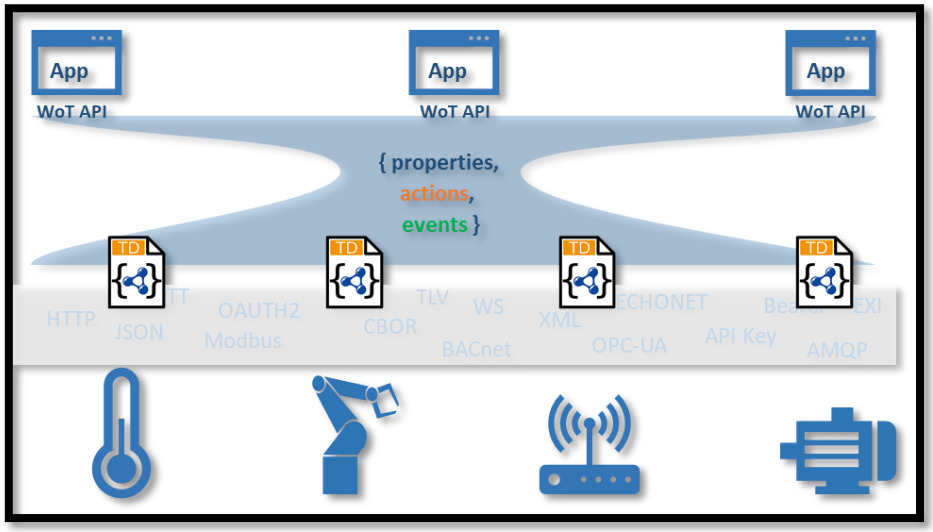


Figura 2.3. Visión genérica de la WoT [5].

2.1.1 WoT Thing Description (TD)

Los metadatos de un dispositivo de IoT, incluida toda la información necesaria para habilitar esta abstracción común, se documentan en lo que se llama **WoT Thing Description (TD)** [3]. El TD, como se ha comentado con anterioridad forma parte de uno de los cuatro pilares fundamentales de la WoT y puede considerarse como el punto de entrada de una instancia de IoT (muy parecido al index.html de un sitio web). Proporciona información sobre qué datos y funciones dispone el dispositivo, qué protocolo de transporte utiliza, cómo se codifican y estructuran los datos, y el mecanismo de seguridad que se usa para controlar el acceso, junto con más metadatos legibles tanto por una máquina como por un ser humano.

Un TD se expresa en JSON-LD y puede ser proporcionado por un dispositivo IoT mismo o alojado externamente en un repositorio a modo de un TD Directory.

La TD provee así, un conjunto de interacciones basadas en un pequeño vocabulario que hace posible la integración de los dispositivos permitiendo la interoperabilidad de diversas aplicaciones.

Como se ha comentado anteriormente, la interacción con un dispositivo IoT se basa en una abstracción de propiedades, eventos y acciones. Estas 3 posibilidades de interacción vienen reflejadas en la TD de la Thing.

- **Propiedades**. Exponen los estados de una Thing. Estos estados pueden ser leídos (GET) y opcionalmente actualizados (PUT). También pueden ser observables para notificar tras un cambio.
- **Eventos**. Describen fuentes de eventos, que emiten datos de forma asíncrona a objetos consumidores.
- **Acciones**. Permiten invocar (POST) funciones o métodos de una Thing, para modificar un estado (por ejemplo, encender/ apagar una luz), o para disparar un proceso (por ejemplo, reducir gradualmente una luz hasta apagarla)

En general, la TD proporciona los metadatos necesarios para las interacciones a través de los posibles diferentes protocolos de IoT (HTTP, MQTT, CoAP) identificados por esquemas URI Proporciona también tipos de contenido en base a los tipos de medios (application/json, application/xml…) y mecanismos de seguridad (autentificación, autorización, confidencialidad, etc.)

A continuación se muestra un ejemplo de una instancia de TD y se ilustra el modelo de interacción con propiedades, acciones y eventos mediante la TD de una lámpara denominada *MyLampThing*.

Ejemplo Thing Description

```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": "urn:dev:ops:32473-WoTLamp-1234",
  "title": "MyLampThing",
  "securityDefinitions": {
    "basic_sc": {"scheme": "basic", "in":"header"}
  },
  "security": ["basic_sc"],
  "properties": {
    "status" : {
      "type": "string",
      "forms": [{"href": "https://mylamp.example.com/status"}]
    }
  },
  "actions": {
    "toggle" : {
      "forms": [{"href": "https://mylamp.example.com/toggle"}]
    }
  },
  "events":{
    "overheating":{
      "data": {"type": "string"},
      "forms": [{
        "href": "https://mylamp.example.com/oh",
        "subprotocol": "longpoll"
      }]
    }
  }
}
```

A partir del ejemplo expuesto, conocemos que existe una propiedad denominada status. Además, se proporciona información que nos indica que se puede acceder a esta propiedad a través del protocolo HTTPS con un método GET a la URI *https://mylamp.example.com/status* (contenida dentro de la estructura forms por el miembro href).

De manera similar se especifica una acción para alterar el estado del interruptor empleando el método POST en el recurso *https://mylamp.example.com/toggle*.

Por último, encontramos el evento "overheating" que indica que podemos suscribirnos para ser notificados sobre un posible sobrecalentamiento de la lámpara, empleando el protocolo HTTP con el subprotocolo long-polling en *https://mylamp.example.com/oh*.

En el ejemplo también se especifica un esquema de seguridad básico que requiere de nombre de usuario y contraseña para acceder.

Para que un dispositivo sea una Thing y un cliente o consumidor pueda interactuar con ella, debe tener disponible una representación de su Thing Description.

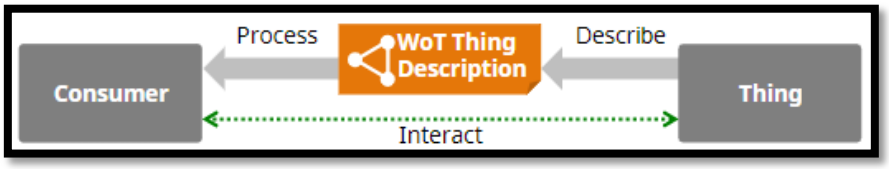


Figura 2.4. Interacción consumidor-Thing [3].

En el desarrollo del proyecto se ha seleccionado el protocolo MQTT para los eventos de la Thing. MQTT (Message Queuing Telemetry Transport) es un protocolo de capa de aplicación inventado en 1999 por Andy Stanford-Clark y Arlen Nipper y que ha ido poco a poco evolucionando hasta convertirse en un importante protocolo para la comunicación máquina a máquina (M2M) [7].

MQTT es un protocolo de publicación-suscripción. De esta manera, los clientes se suscriben a un tema o topic de interés y reciben notificaciones cada vez que se publique un mensaje nuevo para este tema.

En este proyecto la Thing actúa como publicador y el servicio que la integre se suscribe a su evento. Ambos no hablan entre sí directamente, sino a través de un servidor central llamado broker, el cual se puede implementar localmente si tanto publicador como suscriptor se encuentran en la misma red o, en internet, si no están en la misma red local.

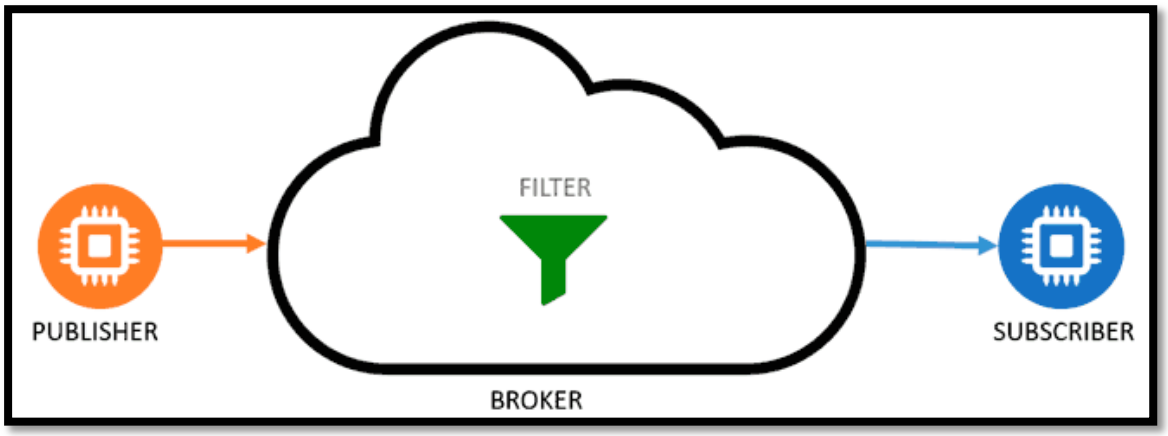


Figura 2.5. MQTT Broker.

En relación a la TD (Thing Description) y al protocolo MQTT, no es posible leer una propiedad sino solo observarla. A continuación se muestra un ejemplo de TD de una lámpara con el evento iluminancia que indica cuánto ilumina la Thing. El cliente MQTT publica con frecuencia los datos de iluminancia en el tema /illuminance que se ejecuta en la dirección 192.168.1.187:1883 por el agente MQTT.

Ejemplo Thing Description con evento MQTT

```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": " urn:dev:ops:32473-WoTIlluminanceSensor-1234",
  "title": " MyIlluminanceSensor",
  "securityDefinitions": {
    "basic_sc": {"scheme": "basic", "in":"header"}
  },
  "events":{
    "illuminance": {
      "data":{"type": "integer"},
      "forms": [
        {
          "href": "mqtt://192.168.1.187:1883/illuminance",
          "contentType": "text/plain",
          "op": "subscribeevent"
        }
      ]
    }
}
```

En el Anexo 2 del presente documento se muestran las TD de las Things creadas para validar el sistema de integración que defiende el trabajo.

2.1.2 Arquitectura de la WoT

La arquitectura de la WoT trata de relacionar los cuatro bloques de construcción que definen la WoT y que se describen a continuación [2]:

- **Thing Description.** Bloque normativo que proporciona un formato de datos legible para describir los metadatos y las interfaces de interacción de las Things.
- **Binding Templates.** Bloque normativo que proporciona las pautas sobre cómo definir interfaces orientadas a las Things para protocolos y ecosistemas de IoT.
- **Scripting API.** Bloque opcional que permite la implementación de la lógica de aplicación de una Thing utilizando una API de JavaScript, de manera similar a las API del navegador web.
- **Security and Privacy Guidelines.** Bloque transversal que proporciona pautas para la implementación y configuración de forma segura en las Things, analizando problemas que deben tenerse en cuenta en cualquier sistema que implemente la WoT.

La arquitectura de la WoT permite cumplir con los siguientes principios comunes:

- Debe permitir el interfuncionamiento mutuo de diferentes ecosistemas utilizando tecnología web.
- Debe basarse en la arquitectura web mediante API RESTful.

- Flexibilidad. Debido a la gran variedad de dispositivos IoT, la arquitectura de la WoT debe ser capaz de cubrir todas las variaciones.
- Compatibilidad. La WoT debe proporcionar un puente entre las distintas soluciones de IoT que ya existen en curso y en que está en desarrollo, con la tecnología web basada en conceptos de WoT. Es decir, debe ser compatible con las soluciones de IoT existentes y los estándares actuales.
- Escalabilidad. La WoT debe poder escalar a soluciones de IoT que incorporen de miles a millones de dispositivos.
- Interoperabilidad. La WoT debe proporcionar interoperabilidad entre los fabricantes de dispositivos y de la nube. Debe ser posible coger un dispositivo habilitado para WoT y conectarlo con un servicio en la nube de diferentes fabricantes.

A su vez, la arquitectura de la WoT debe permitir a las Things tener las siguientes funcionalidades:

- Leer el estado de la Thing.
- Actualizar el estado de la Thing.
- Suscribirse, recibir y darse de baja de notificaciones en el estado de la Thing o en eventos de la Thing.
- Invocar funciones que causen cierta actuación sobre la Thing.

En lo que respecta a la arquitectura de una Thing de la WoT, ésta se desglosa en cuatro bloques:

- **Behavior**. Hace referencia al comportamiento y los datos generales de una Thing: nombre, descripción, etc.
- **Interaction affordances** y **Data Schemas**. Proporcionan un modelo de cómo los consumidores pueden interactuar con la Thing a través de operaciones abstractas, sin hacer referencia al tipo de protocolo.
- **Security Configuration**. Son los mecanismos utilizados para controlar el acceso a las interacciones y controlar la seguridad y privacidad de los metadatos.
- **Protocol Binding(s).** Contiene todo lo necesario para relacionar cada interacción con un mensaje en concreto de un cierto protocolo, como MQTT.

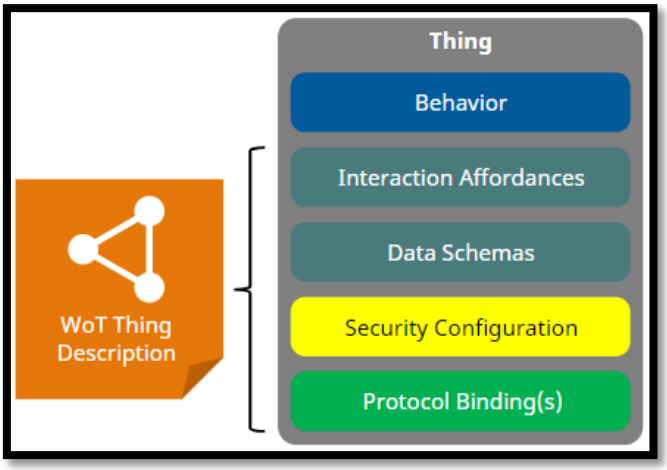


Figura 2.6. Arquitectura de una Thing [2].

Los conceptos de la WoT son aplicables a todos los niveles relevantes para las aplicaciones de IoT: a nivel de dispositivo, a nivel local y a nivel de nube. Esto fomenta interfaces y API comunes en los diferentes niveles y permite varios patrones de integración como Thing-to-Thing, Thing-to-Gateway, Thing-to-Cloud, Gateway-to-Cloud, etc. La siguiente figura ofrece una descripción general de cómo los conceptos de la WoT presentados anteriormente se pueden aplicar y combinar para abordar diferentes casos de uso.

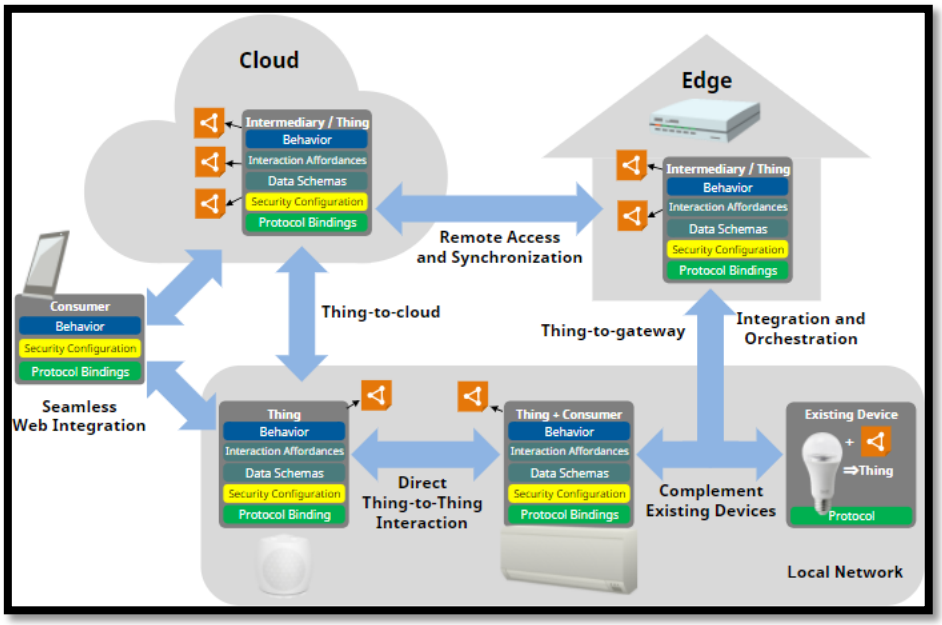


Figura 2.7. Arquitectura abstracta de la WoT [2].

Hasta ahora hemos descrito la arquitectura de la WoT en términos de componentes o bloques abstractos, como Things o consumidores. Cuando estos componentes se implementan como software para asumir un papel específico en la WoT se denominan Servients. Por tanto podemos definir un Servient como un artefacto software que implementa un elemento de la WoT, ya sea Thing, consumidor o un intermediario.

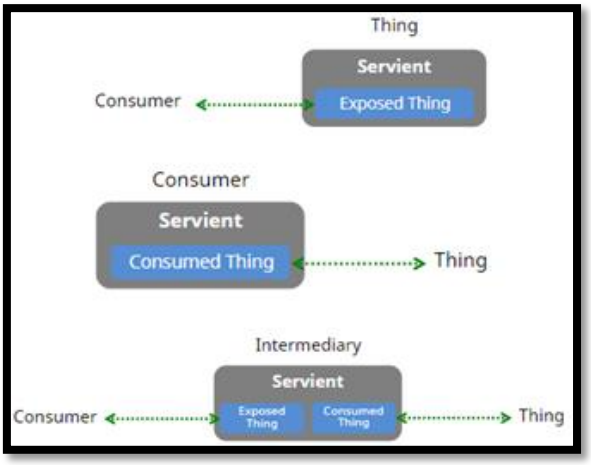


Figura 2.8. Servients [2].

2.2 Microservicios web

Para una mejor comprensión del trabajo realizado, es importante conocer qué es un servicio y un microservicio web.

Podríamos definir un servicio como una unidad funcional que da respuesta y soluciona un problema. De esta forma, se podría desarrollar una aplicación como un conjunto de pequeños servicios donde cada uno se ejecuta como un proceso propio y que se comunica con mecanismos "ligeros", de forma habitual, a través de una API accesible por HTTP [8]. A estos pequeños servicios, encomendados cada uno de ellos a una tarea en concreto, se les conoce como microservicios.

Una de las características propias de los microservicios es su capacidad de ser reemplazables y fácilmente actualizables. Cuando se implementa y desarrolla un sistema cuya arquitectura está basada en microservicios, se tiene la ventaja de tener partes totalmente independientes, lo que favorece a la hora de realizar cambios y modificaciones en él [9]. Caso totalmente opuesto a cuando hablamos de una arquitectura monolítica, en la que todo el sistema está construido con la misma tecnología. De esta forma, un simple y pequeño cambio en el sistema, requiere la prueba y el re-despliegue completo del mismo.

De esta forma, y como ventajas de un sistema basado en microservicios, cada uno de ellos se puede escribir en diferentes lenguajes de programación y no tiene por qué tener como base la misma tecnología de persistencia de datos. Además, sumado a la facilidad de escalar, cada microservicio tiene su propio ciclo de vida, siendo fácil de abarcar, probar, desplegar o versionar.

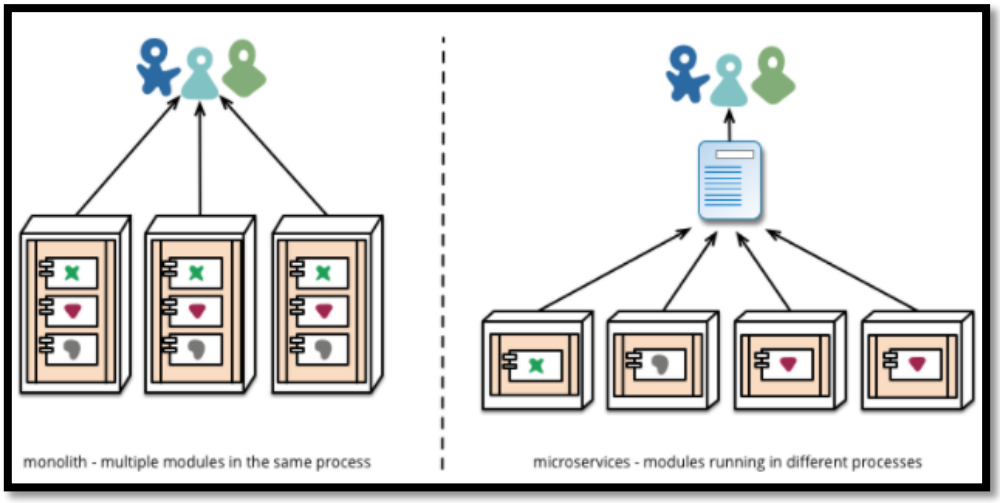


Figura 2.9. Comparativa de arquitecturas [8].

2.3 Soluciones de integración y comunicación entre microservicios

Cuando hablamos de un sistema basado en microservicios, se tiene como objetivo principal la división, integración y comunicación entre todos ellos.

Hay que tener en cuenta que cada microservicio debe ser independiente y no encapsular operaciones del mismo tipo, siendo su modularización ajena al lenguaje, framework o tecnología utilizada. El tamaño debe ser otro de los puntos clave a la hora de dividir un sistema en microservicios, teniendo en cuenta que un desarrollador lo pueda abarcar, así como un equipo gestionar.

Como hemos visto en el apartado anterior, en un software con arquitectura monolítica, los distintos componentes se comunican entre ellos mediante llamadas a nivel de lenguaje ejecutándose todos los objetos bajo un mismo proceso. Al otro lado, el software basado en microservicios se ejecuta en diferentes procesos que, en ocasiones, están alojados en varios servidores. En este caso, los microservicios deben actuar mediante un protocolo de comunicación que, en función de cada servicio, puede ser HTTP, AMQP o un protocolo binario como TCP.

La comunicación e integración entre el cliente de una aplicación y el conjunto de microservicios que la forman puede realizarse de diferentes formas, cada una de ellas enfocada a un escenario diferente. Por lo general existen dos sistemas de comunicación:

- Por clase de protocolo: sincrónico o asincrónico.
    - **Protocolo sincrónico**, como HTTP/HTTPS. El cliente envía una solicitud (request) y, hasta que no obtenga respuesta (response) del servicio, no puede continuar con su tarea produciéndose un bloqueo de los subprocesos.
    - **Protocolo asincrónico**, como AMQP o MQTT. En este caso el cliente no espera ninguna respuesta del servicio, sino que se envía un mensaje a una cola como

RabbitMQ o a otro agente de mensajes lo que permite no bloquear los subprocesos.

- Por número de receptores: uno o varios.
    - **Receptor único**. Cada solicitud se procesa por un único receptor.
    - **Varios receptores**. Se basa en una interfaz de bus de eventos donde la interfaz de bus de eventos propaga las actualizaciones entre los diferentes microservicios. Un ejemplo es el mecanismo de publicación/suscripción.

Una aplicación basada en microservicios suele usar una combinación de estos estilos de comunicación. El tipo más común es la comunicación de un único receptor con un protocolo sincrónico como HTTP/HTTPS al invocar a un servicio normal HTTP Web API. Además, los microservicios suelen usar protocolos de mensajería para la comunicación asincrónica entre microservicios.

2.4 Orquestación vs Coreografía de microservicios

Hablamos de orquestación y coreografía cuando se habla de la manera de acoplar varios microservicios juntos.

Por un lado la orquestación hace referencia a una instancia, llamada director u orquestador, que tiene control del resto de microservicios de forma centralizada. Por otro lado la coreografía hace referencia a que cada microservicio se gestiona de manera propia y la aplicación resultante aparece con la suma de todos los microservicios [9].

Un sistema basado en coreografía implica un mayor grado de libertad a la hora de gestionar los microservicios. Si aparece un evento determinado, cada microservicio actúa por su parte. Otra ventaja es que al agregar un microservicio nuevo, no es necesario acoplarlo al orquestador (como en la orquestación), simplemente responde si al escuchar el evento, es necesario. Como desventaja, al no existir un director, es difícil rastrear y conocer si todas las acciones requeridas se realizan con éxito. Aunque esto se podría solucionar con un microservicio adicional que compruebe si se activan los microservicios que deben ejecutarse al saltar un evento concreto.

En el desarrollo de este proyecto, y tras detectar un posible hueco de investigación en la integración de Things de la WoT, se ha optado por una orquestación de microservicios, generando un microservicio director (orquestador) que se encarga de realizar la integración comunicándose con el resto de microservicios (las Things) siempre y cuando lo requiera.

De esta manera, se pretende crear un sistema en el que no sea necesario modificar y alterar las Things de la WoT, modificación necesaria e inevitable en caso de implementar una coreografía de microservicios. Se crea así un sistema más sencillo y simple de mantener que proporciona una buena manera de controlar todo el flujo de la aplicación.

CAPÍTULO 3. OBJETIVOS Y METODOLOGÍA

Este capítulo es el puente entre el estudio del dominio y la contribución a realizar. Se detalla el objetivo general y los objetivos específicos. Para concluir se hace un repaso de la metodología de trabajo llevada a cabo.

3.1. Objetivo general

Este proyecto técnico se enmarca en la necesidad de desarrollar un software necesario para lograr y conseguir un impacto concreto y en relación con el objetivo principal: proponer una solución de integración en la WoT. Impacto ligado al mundo tecnológico y de actualidad como es el IoT y, en concreto, la WoT.

Para ello y tras el estudio de las diferentes tecnologías, se apuesta por un nuevo lenguaje de programación especialmente orientado a orquestación y coreografía de servicios como es Ballerina, que será la encargada de orquestar los dispositivos y generar el sistema, demostrando así su efectividad.

Al tratarse de un trabajo de tipo técnico es necesario que el objetivo principal se centre en conseguir un efecto observable y que evidencie la consecución del mismo.

Haciendo uso de nuevas y punteras herramientas tecnológicas ligadas a microservicios, como SpringBoot y Ballerina, se propone un sistema innovador que pretende dar pie a un mercado que actualmente se encuentra en investigación y desarrollo pudiendo de esta manera, continuar y avanzar en esta línea de trabajo.

Para primero probar y segundo demostrar la efectividad del sistema y, en concreto, del funcionamiento de los microservicios web mediante SpringBoot, de la capacidad de integrar mediante Ballerina y de cómo funciona un dispositivo IoT en la WoT, se propone el objetivo de implementar un caso práctico real en el que sensores y actuadores de una Raspberry Pi protagonicen las Things de la WoT y los microservicios generados sean capaces de integrarlas.

3.2. Objetivos específicos

Las metas establecidas en este proyecto van desde la documentación y aprendizaje de la WoT hasta el diseño de un sistema basado en microservicios que logre la integración de las Things. En más detalle, este objetivo general se desglosa en una serie de objetivos específicos y analizables independientemente, que marcan los pasos a seguir para la consecución del mismo.

Los objetivos específicos se dividen en dos categorías. Una sobre el marco teórico o el estado del arte donde se realiza un análisis bibliográfico y la otra sobre el desarrollo específico de la contribución.

Dentro de la primera categoría, se definen dos objetivos específicos:

1. **Estudio del estado del arte del de la WoT y su integración**. Como primera parte del trabajo se realiza una investigación sobre el mercado IoT y su tendencia evolutiva a corto plazo. El eje principal de este objetivo es el estudio de la WoT y cómo se implementa una Thing en ella.

   Para ello, durante toda esta fase de documentación y aprendizaje, se han utilizado principalmente las guías y recomendaciones del W3C, especialmente las que detallan la arquitectura y la descripción de una Thing.

   Además, durante toda esta primera parte del proyecto y siguiendo las guías del libro WoT Guinard [6] se han ido realizando las primeras tomas de contacto con Things de la WoT ya implementadas y alojadas en diferentes servidores.

   Como último apartado, se realiza un estudio sobre las posibilidades y soluciones para la integración en la WoT, donde se investigan soluciones ya aportadas y se descubren las herramientas y tecnologías que pueden dar solución al objetivo general.

2. **Análisis y aprendizaje de Ballerina**. La fase de documentación y aprendizaje de las plataformas, especialmente Ballerina, se ha realizado, en una gran mayoría, haciendo uso de comunidades, foros y manuales online. A su vez, pequeños tutoriales han servido para asimilar conceptos y comprender de manera más eficiente el funcionamiento del lenguaje.

   El objetivo general del trabajo pretende emplear este lenguaje en el microservicio encargado de orquestar las Things del a WoT.

   En lo que refiere a la implementación de las APIs de las Things, junto con Node.js y Express es necesario utilizar librerías propias para la entrada y salida a través de la GPIO de la Raspberry.

Dentro de la segunda categoría, se definen tres objetivos específicos:

1. **Elección de arquitectura**. Cualquier sistema informático requiere de una arquitectura que aporte la estructura y los detalles necesarios para que sea funcional. Por ello, y antes de programar, es necesario determinar la arquitectura del sistema.

   La arquitectura de este trabajo debe fijar el número de microservicios a emplear, cómo están implementados, cómo fluye la información de uno a

otro, quién se encarga de leer y escribir datos en las Things, qué lenguaje utiliza cada uno, etc.

2. **Desarrollo del sistema**. De los cinco, es el objetivo específico más importante y el que abarca la mayor parte del trabajo.

   Una vez determinada la arquitectura del sistema, su desarrollo se desglosa en las siguientes fases:

   - Desarrollar las distintas versiones del orquestador en Ballerina dependiendo de la manera en la que se obtienen los datos: suscripción a un tema MQTT, mediante llamadas GET a la API o en combinación de ambas.
   - Determinar la estructura del fichero de entrada, fichero que será parseado a lenguaje Ballerina.
   - Desarrollar el microservicio Parser, encargado de parsear el fichero de entrada y generar como salida, el orquestador a modo de un fichero en formato Ballerina.

3. **Prueba y evaluación**. Es la parte que engloba todos los test necesarios para validar el correcto funcionamiento del producto generado. Este objetivo podría realizarse utilizando Things ya expuesta en la web. Sin embargo, y como forma de potenciar aún más este trabajo, este objetivo va algo más allá. Se propone como meta la implementación de varias Things propias y expuestas en la WoT de manera que se creen todos los posibles escenarios que abarca el producto generado y de esta manera ser capaces de probar y evaluar de manera positiva su funcionamiento.

La consecución individual y lineal de todos los objetivos específicos lleva al logro y consecución del objetivo general.

3.3. Metodología del trabajo

Tras conocer los objetivos del proyecto, es momento de definir la metodología que se seguirá para culminarlos. La metodología a seguir en el proyecto va a definir el conjunto de técnicas, métodos y procedimientos que se deben seguir durante el desarrollo del mismo para generar el producto y alcanzar los objetivos propuestos.

Una vez obtenida una visión completa del alcance del proyecto y dadas las características del mismo, se elige la metodología ágil, en concreto SCRUM, para llevarlo a cabo. Al tratarse de un sistema totalmente innovador, los requisitos podrán cambiar sobre la marcha, pudiendo de esta manera añadirse o eliminarse funcionalidades si se comprueba que estos cambios añadirán valor al producto final.

Por otra parte y dada la inexperiencia en proyectos de esta índole, se realizará un aprendizaje por descubrimiento, tanto de la WoT como de Ballerina, una característica propia de las metodologías ágiles.

De esta manera, se irán obteniendo versiones funcionales del producto, que podrán ser mejoradas conforme a la retroalimentación obtenida en la fase de pruebas y añadiendo por completo todos los protocolos y funcionalidades de la WoT. De esta manera, tenemos un proyecto en el que el desarrollo es incremental e iterativo.

Una vez concluido el desarrollo del producto se realizarán pruebas que serán aprobadas de manera práctica y visual, donde se provocarán voluntariamente cambios en las propiedades y eventos de la Thing para ver si la respuesta del sistema cumple con lo establecido.

Para lograr los objetivos propuestos y como base de una metodología SCRUM, se definen sprints, que serán las iteraciones a seguir en el desarrollo del producto. De esta manera estableceremos un ritmo de trabajo con un tiempo prefijado, siendo la duración habitual de un Sprint de entre dos y cuatro semanas. En cada Sprint conseguiremos un incremento del producto. En la siguiente tabla se detalla la duración de cada uno de los sprints.

Nº	Definición	Semanas						
		1	2	3	4	5	6	7
1	Creación de varias Things							
2	Funcionalidad Get							
3	Funcionalidad MQTT							
4	Funcionalidad MQTT y GET							
5	Pruebas/tests							
6	Estructura archivo configuración							
7	Implementación del Parser							

Tabla 2. Resumen de Sprints

CAPÍTULO 4. HERRAMIENTAS Y TECNOLOGÍAS

En este capítulo se detallan las herramientas y tecnologías empleadas en la realización de este trabajo fin de máster, explicando en cada una de ellas en qué consiste, cómo se utiliza o sus virtudes y desventajas.

4.1 Spring Boot

Spring Boot [10] puede definirse como una infraestructura ligera que elimina la mayor parte del trabajo de configurar las aplicaciones basadas en Spring, centrando al desarrollador en el propio desarrollo de la solución alejándolo así de la compleja configuración de Spring Core.

Las características de Spring Boot pueden resumirse de la siguiente manera:

- Incorporación directa de aplicaciones de servidores web/contenedores como Apache Tomcat o Jetty, eliminando la necesidad de incluir archivos WAR (Web Application Archive)
- Simplificación de la configuración de Maven gracias a los POM (Project Object Models) “starter”.
- Configuración automática de Spring en la medida de lo posible

Para iniciar un proyecto de Spring Boot se puede emplear Spring Start [11], página web que provee Spring para crear un proyecto inicial, pudiendo elegir, entre otras cosas, los paquetes de dependencias o la versión Java a utilizar.

El lenguaje de programación que emplea Spring Boot es Java. Por tanto, el principal requisito para utilizar esta tecnología es el JRE y el JDK.

Spring Boot se ha empleado en el proyecto para generar el servicio web encargado de crear el microservicio en Ballerina encargado de integrar las Thing de la WoT.

4.2 Ballerina

Ballerina [12] es un lenguaje de programación de código abierto desarrollado por WSO2. Actualmente es el lenguaje más avanzado para el ámbito de las integraciones, utilizado para interconectar diferentes servicios, como los servicios web.

Ballerina dispone de sintaxis tanto textual como gráfica. Al construir el servicio alrededor de un modelo de programación visual, Ballerina permite conectar aplicaciones y servicios y crear cualquier programa a través de un diagrama de secuencia. Es un enfoque que nos permitirá, en el diseño de la lógica de integración, desarrollos más rápidos y efectivos. Pero, en el desarrollo de este trabajo, se ha optado por el diseño textual.

La instalación es simple. Basta con descargar y ejecutar el archivo instalador correspondiente al sistema operativo y a la versión que queramos instalar y, posteriormente, añadir la variable de entorno correspondiente. Algunos de los comandos internos que dispone son: ballerina run, ballerina build o ballerina test.

En el desarrollo de este trabajo se ha empleado una librería adicional, pzfreo/mqtt [13], que permite la integración de clientes MQTT en Ballerina, punto clave en la integración de Things. Esta es una librería soportada para la versión de Ballerina 1.0.0.

4.3 Otras

Como tecnologías y herramientas secundarias se han utilizado Node.js, Express, Eclipse, Visual Studio Code y WSO2 Integration Studio.

Visual Studio Code [14] es un editor de código desarrollado por Microsoft [15] el cual está construido mediante Electron, un framework para el desarrollo de aplicaciones de escritorio desarrollado por GitHub mediante tecnologías web HTML, CSS y Javascript.

Visual Studio Code cuenta con una amplia gama de extensiones que se le pueden añadir para adaptarse a las necesidades del usuario como la extensión Ballerina.

Node.js [16] es un entorno de ejecución del lenguaje Javascript desarrollado sobre el motor V8 de Chrome. Esto permite la creación de aplicaciones de Javascript fuera del entorno web en multiplataforma.

Node.js se ha convertido en una herramienta popular para la creación de aplicaciones web del lado del servidor así como para la empaquetación y “transpiling” de código para aplicaciones web del lado del cliente.

Node.js se usa para el desarrollo de microservicios tanto para la parte del cliente como del servidor. Una de las librerías más importantes que se utilizan para la realización del proyecto es Express [17].

Eclipse [18] es un entorno de desarrollo integrado (IDE), de código abierto y multiplataforma. Es el IDE de desarrollo más popular en la actualidad y el IDE de Java más utilizado.

Su uso es muy sencillo e intuitivo e incluye herramientas de refactorización para hacer el código más legible. También incluye un comprobador de sintaxis en tiempo real que nos avisará si el programa fallará a la hora de compilar.

Una de sus principales ventajas es su extensa colección de plug-ins, unos publicados por Eclipse y otros por terceros. Los hay gratuitos, de pago, bajo distintas licencias, pero casi para cualquier cosa que nos imaginemos se tiene el plug-in adecuado.

Por último, WSO2 [19] Integration Studio [20] es un entorno de desarrollo utilizado para diseñar escenarios de integración y desarrollarlos. Es un entorno de desarrollo gráfico que proporciona un desarrollo de artefactos de integración eficiente y acelera los ciclos de vida del desarrollo.

Incluye una paleta de herramientas visuales, la opción de importar conectores a la paleta de herramientas, construcciones de integración de nivel superior y vistas de propiedades para configuraciones complejas.

WSO2 Integration Studio ofrece diferentes enfoques de implementación, incluida la implementación de CAR (Carbon Application Archive), la implementación de Docker, la implementación de Kubernetes, la implementación de la nube de integración WSO2, etc.

CAPÍTULO 5. SOLUCIÓN DE INTEGRACIÓN

En este capítulo se muestra la solución de integración propuesta y desarrollada que permite la integración de Things de la WoT.

5.1 Arquitectura propuesta

A lo largo del desarrollo del proyecto, se han ido estudiando diferentes modelos arquitectónicos que dieran soporte y solución al objetivo principal del trabajo.

Tras el estudio e investigación sobre posibles arquitecturas para integración, se decidió elegir una arquitectura basada en orquestación de servicios web. Un microservicio que fuera capaz de gestionar la lógica deseada invocando a las distintas Things.

La primera herramienta que se utilizó fue Integration Studio, de WSO2. Utilizando este IDE se podía crear la lógica deseada sin programar nada de manera textual, únicamente utilizando diseño gráfico y sus respectivas configuraciones. Esta arquitectura y solución planteada es válida pero tiene el inconveniente de que sólo puede ser elaborada por una persona con dotes en informática y, en especial, en Integration Studio. De esta forma, se pueden simular e implementar todas las casuísticas posibles pero reprogramando el sistema en cada una de ellas. En el capítulo 6 se muestra un ejemplo de implementación con WSO2.

Esta primera solución aportada cumple con el objetivo del trabajo pero se aleja a la realidad de poder ofrecer un producto para un usuario de nivel básico y estándar. Buscando solución a este inconveniente, se reelaboró la arquitectura donde se aparta Integration Studio a un lado para dar protagonismo a Ballerina.

Se ha diseñado una arquitectura, en la que, a partir de un fichero de configuración, un microservicio web llamado Parser se encarga de realizar el parseo y generar el microservicio en Ballerina capaz de integrar las Things deseadas. La figura 5.1 muestra dicha arquitectura. Este servicio Ballerina recibe el nombre de Orquestador y será el encargado de interactuar con las Things a través de sus propiedades, eventos y acciones. En esta arquitectura existe un receptor, el microservicio orquestador y se utilizan llamadas de tipo sincrónicas, mediante HTTP y suscripciones a eventos de servidores MQTT, sobre las distintas Things de la WoT en función de cómo éstas estén implementadas.

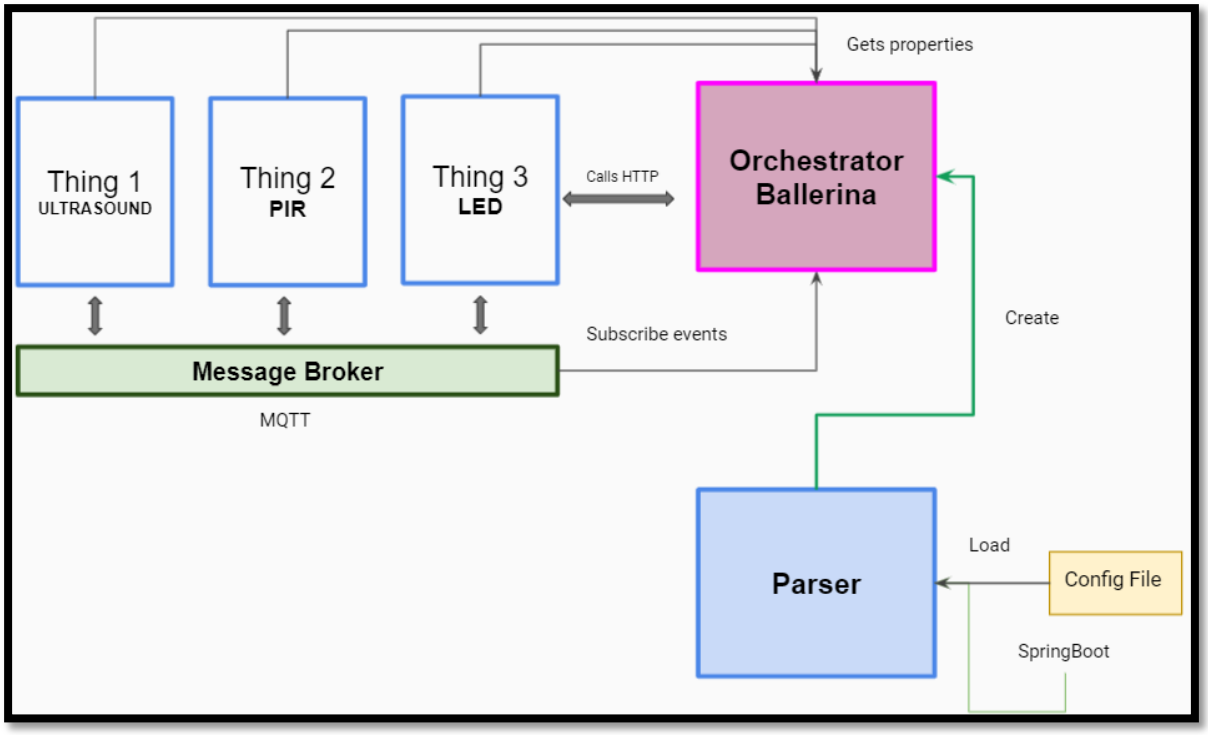


Figura 5.1. Arquitectura propuesta.

El archivo de configuración contiene todo lo necesario acerca de las Things a integrar y de la lógica que se quiere cumplir. Para ello, antes de generar el archivo, se debe conocer y hacer un estudio de la Thing Description de cada Thing que queramos integrar, para saber qué y cómo generar el archivo de configuración. Es necesario conocer la manera en la que se acceden a sus propiedades, cómo se puede suscribir a sus eventos y de qué forma se invocan a sus acciones.

Dado que existen múltiples formas de exponer las propiedades de las Things, de publicar los eventos en función del protocolo utilizado y de invocar acciones, el desarrollo de este proyecto se ha centrado y acotado de la siguiente forma:

- Propiedades: podemos consultar el estado de la Thing a través de una petición a la URI correspondiente utilizando el método GET.
- Eventos: Nos podemos suscribir al evento de una Thing que utilice únicamente el protocolo MQTT, conociendo su host y el topic.
- Acciones: Para invocar cambios en las Things, utilizamos la llamada a la API correspondiente a través del método PUT estableciendo en el body de la petición el payload correspondiente.

Tras esta aclaración, podemos dividir las Things en antecedentes y consecuentes, según vayamos a leer y/o suscribirnos a sus eventos o invocar a sus acciones.

Analizando por partes la arquitectura propuesta se pueden analizar 4 bloques.

- **Things**. El proyecto parte de que las Things ya están desarrolladas y funcionando en la WoT, aunque bien se ha implementado para el desarrollo de un caso práctico real. Se da por hecho que se conoce la dirección de la Thing Description donde conoceremos todo acerca de la Thing.
- **Config File**. Como se puede observar, el archivo de configuración (Config File) es el bloque del que parte el sistema, el que primero debe generarse. Se trata de un fichero en formato .txt que contiene los valores necesarios de las Things para poder interactuar e integrarlas de forma correcta.

  La primera línea del fichero contiene las Things que actúan como antecedentes, de las que se leen propiedades y/o se suscriben a eventos. En función y dependiendo del tipo de interacción, se tiene un formato u otro, detallados y explicados en el Anexo 1 del documento.

  Si se tienen dos o más Things antecedentes, es necesario establecer en la segunda línea del documento la expresión lógica que se tiene que cumplir para que se realicen las llamadas a las Things consecuentes. Por ejemplo, si se tienen dos Things antecedentes con sus expresiones definidas, bien se puede llamar al consecuente cuando se cumpla una de las dos o bien cuando se cumplan las dos. Para ello se hace uso de los operadores lógicos AND y OR.

  En la última línea del fichero se establecen las Things consecuentes, marcando todos los parámetros necesarios (host, payload…) así como el orden de los mismos. Todo ello, se encuentra documentado en el Anexo 1.
- **Parser**. Es el servicio encargado de crear el microservicio de Ballerina (orquestador) el cuál integra todas las Things. Se puede definir como el microservicio que traduce el archivo de configuración a código Ballerina. Según el formato detectado en el archivo de configuración se crea un tipo de Orquestador u otro. Esto depende del número de Things a integrar y de si se quiere acceder a sus propiedades, a sus eventos o a ambos.
- **Orquestador**. Es el eje y pieza principal de la arquitectura del sistema. Se crea desde el Parser y existen cinco tipos de plantillas en función del número de Things y la manera de interactuar con ellas (evento o propiedad).

5.2 Implementación de microservicios

Como se ha comentado, el sistema dispone básicamente de dos microservicios: Parser y Orquestador. El primero encargado de generar el segundo a través del archivo de configuración.

5.2.1 Parser

El microservicio Parser se ha implementado bajo la tecnología de Spring Boot y es un microservicio web que parsea el archivo de configuración y genera el correspondiente archivo/microservicio en Ballerina, el orquestador.

Este archivo de configuración, detallado en el Anexo 1, determina las Things que actúan como antecedentes, la lógica que debe cumplir las propiedades y/o eventos de estos antecedentes y las Things consecuentes a las cuales llamaremos en caso de cumplir con la lógica deseada.

Se trata de un servicio web que emplea la dependencia Thymeleaf [21], un motor de plantillas de HTML. El servicio dispone de una serie de rutas y plantillas que se describen a continuación:

- **Index, "/".** Es la ruta principal del servicio, la Home. Desde aquí se realiza la carga del fichero de configuración a partir de formulario web.

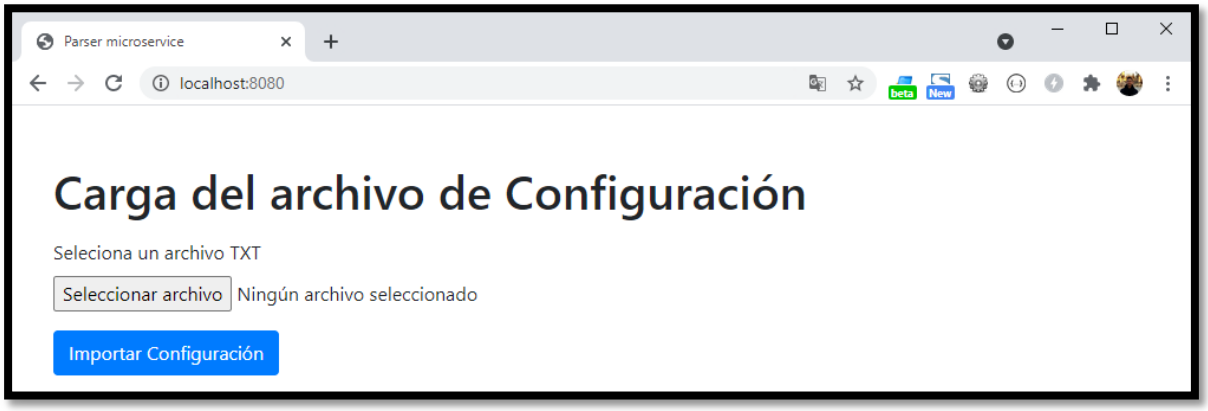


Figura 5.2. Index del microservicio Parser.

Al clicar sobre el botón azul, Importar Configuración, se envía el fichero a través del método POST a la ruta /parser.

- **Parser, "/parser".** El controlador de esta ruta recibe un fichero como parámetro de entrada.
  En caso de que el fichero esté vacío o mal formado, o simplemente haya ocurrido algún otro problema, nos devuelve la plantilla de *error*, con la posibilidad de regresar al index para cargar un nuevo archivo.

Figura 5.3. Error del microservicio Parser.

Por el contrario, si el archivo es correcto, el controlador realiza el parseo, genera el servicio Ballerina guardándolo en la carpeta *output* y nos devuelve la plantilla *ok*, que nos permite descargarnos o ejecutar el servicio.

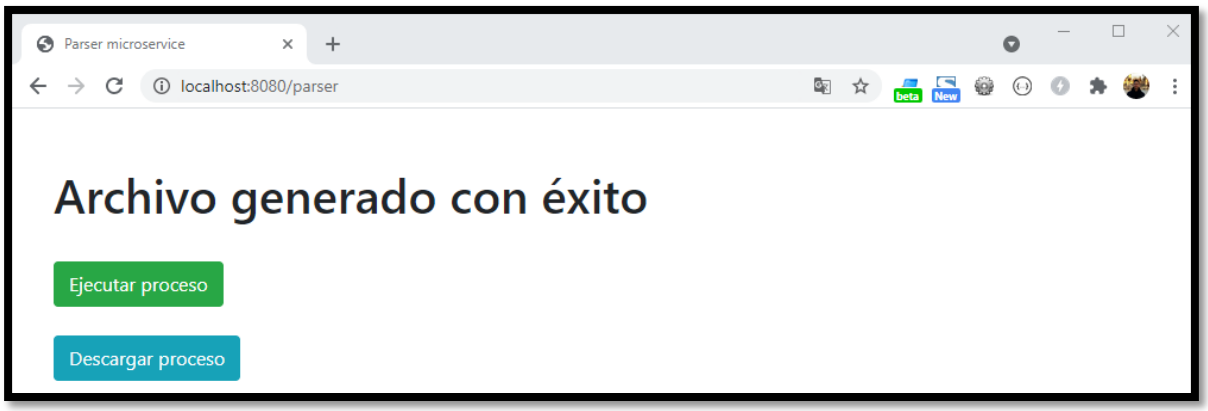


Figura 5.4. Ruta OK del microservicio Parser.

Si hacemos clic en Ejecutar proceso, el servicio nos redirecciona a la ruta /run donde el servicio Ballerina se pondrá en marcha. Al clicar en Descargar proceso, el archivo con nombre salida.bal será descargado.

- **Ejecutar Proceso, "/run".** Al acceder a esta ruta, su correspondiente controlador se encarga de ejecutar el último servicio Ballerina generado. Para ello, el servicio inicia el terminal y ejecuta el servicio: *ballerina run salida.bal*
  Al cabo de unos 15-20 segundos, tiempo necesario para la carga de todo el sistema y dependencias, se inicia el servicio, comenzando así la integración de las Things.

Figura 5.5. Ruta run del microservicio Parser.

- **Detener proceso, "/stop".** Durante la ejecución tenemos la posibilidad de detener el servicio, haciendo clic en Detener proceso. Si esto ocurre, se redirecciona a la ruta /stop, donde se detiene el servicio de Ballerina avisándonos a través de su plantilla correspondiente. Desde aquí tenemos la posibilidad de volver al index para cargar un nuevo archivo de configuración.

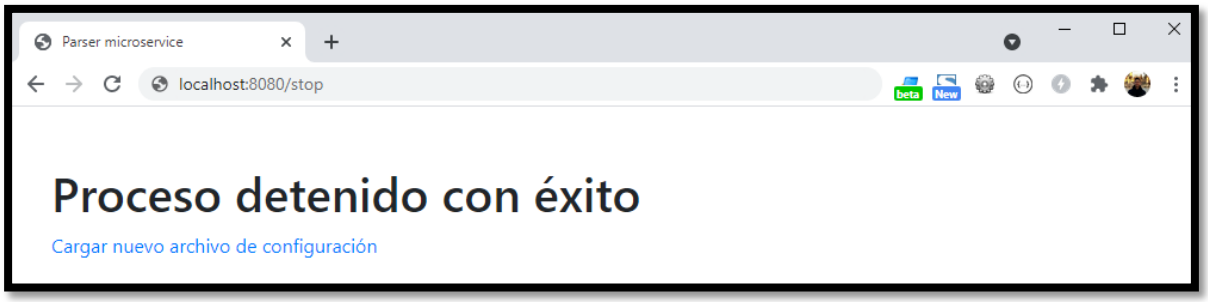


Figura 5.6. Ruta Stop del microservicio Parser.

El microservicio completo se encuentra alojado en un proyecto Github [22].

5.2.2 Orquestador

Es el microservicio encargado de llevar a cabo la integración de las Things. Utiliza el lenguaje de programación Ballerina en su versión 1.0.0 y utiliza los siguientes paquetes o librerías:

- **Http**. Este paquete proporciona una implementación para conectarse e interactuar con endpoints HTTP.
- **Runtime**. Uno de los paquetes de bibliotecas estándar de Ballerina. Proporciona funciones para interactuar con el tiempo de ejecución y el contexto de y para gestionar errores.
- **String**. Este paquete proporciona una serie de operaciones de cadena.

- **Task**. Proporciona la funcionalidad para programar una tarea de Ballerina, administrando la ejecución de las tareas ya sea una sola vez o de forma periódica.
- **Log**. Librería utilizada para la entrada y salida de texto por pantalla.
- **Mqtt**. Librería que proporciona una serie de operaciones para la conexión y suscripción sobre el protocolo mqtt.

El orquestador no es más que un archivo con extensión .bal llamado salida.bal que contiene todas las interacciones con las Things necesarias.

Como se ha comentado anteriormente, podemos acceder a una Thing a través de un GET, suscribirnos a un evento a través del protocolo MQTT e invocar acciones a través de un POST. Con estas posibilidades surgen diferentes casuísticas que determinan diferentes implementaciones del orquestador. Por ejemplo, podríamos invocar una acción de una Thing en función del estado leído de una propiedad de otra Thing o de la notificación llegada al suscribirnos a uno de sus eventos. Todos estos posibles casos se detallan en el apartado siguiente, 6.3, donde se estudia de manera individualizada cada una de ellos.

Por tanto, el orquestador se encarga de accionar las Things consecuentes siempre y cuando se cumpla la expresión lógica establecida. Esta expresión lógica estará compuesta por las condiciones de las Things antecedentes basadas en eventos o en propiedades. De esta forma se pueden utilizar el operador lógico OR (||) cuando la expresión lógica dependa de una u otra propiedad/evento y el operador lógico AND (&&) cuando la expresión lógica dependa de propiedad y evento.

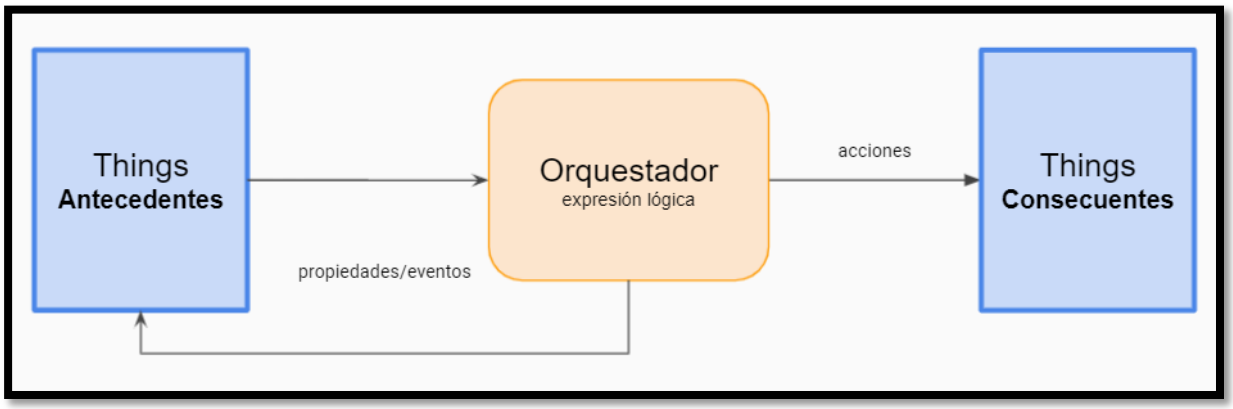


Figura 5.7. Funcionalidad del Orquestador.

Este trabajo fin de máster y, en concreto, el microservicio orquestador, se ha acotado al uso de un máximo de 3 Things como posibles antecedentes, pudiéndose combinar antecedentes GET y/o antecedentes MQTT de la forma deseada. Como consecuentes, no se determina el número máximo, lo que permite invocar tantas acciones, POST, como se quiera.

5.3 Orquestación de Things

En este apartado se detalla el proceso seguido por el microservicio Orquestador para realizar la integración de las Things.

Dependiendo de si leemos propiedades (GET) o nos suscribimos a un evento (MQTT) nos encontramos con los siguientes cinco escenarios, detallados en apartados independientes.

5.3.1 Una propiedad - GET

Se trata del caso considerado más básico ante una posible integración de Things. Partimos de una Thing *antecedente* de la cual, a través de su Thing Description, sabemos que podemos leer el estado de una de sus propiedades haciendo una llamada a su API empleando el método GET. A continuación, necesitamos que esa propiedad alcance un valor determinado, definido como *condición.* Si esa condición es cierta, se cumple, el sistema es capaz de realizar la llamada a la acción de una o varias Things definidas como *consecuentes*.

El pseudocódigo que implementa el microservicio de Ballerina sería el siguiente:

```
Definir el timer
Definir el antecedente
Definir el/los consecuente(s)
nuevoServicio()
        onTrigger()
                get del antecedente
                if(condicion==true)
                        put al consecuente(s)
```

Como en este caso estamos ante un tipo de comunicación síncrona entre el Orquestador y la Thing antecedente, debemos definir el *timer,* que es la frecuencia con la que queremos realizar las llamadas GET sobre la API de la Thing antecedente. A menor frecuencia, más llamadas se realizarán en un mismo intervalo de tiempo. Para establecer este timer, es necesario conocer la Thing Description del dispositivo y saber cada cuánto tiempo se actualizan las propiedades, lo que nos puede llevar a un ahorro de recursos por parte del sistema Orquestador.

5.3.2 Un evento - MQTT

En este caso, nos encontramos con otro de los escenarios básicos. Partimos de una Thing *antecedente* de la cual, a través de su Thing Description, sabemos que podemos suscribirnos a un evento a través del protocolo MQTT, necesitando para ello conocer el topic o tema en el que la Thing publica dicho evento. A continuación, necesitamos suscribirnos al comentado topic y esperar a ser notificados por la API. Cuando somos notificados, si el evento y el valor recibido son los esperados, el sistema es capaz de realizar la llamada a la acción de una o varias Things definidas como *consecuentes*.

El pseudocódigo que implementa el microservicio de Ballerina sería el siguiente:

```
Definir el/los consecuente(s)
Main()
        Definir el cliente mqtt
        Suscripción al topic del cliente
Callback()
        onMessage()
                save payload
                if(condicion==true)
                        put al consecuente(s)
```

En este caso no necesitamos de un *timer* que gestione los tiempos de ejecución ya que estamos ante una comunicación asíncrona. A partir de la suscripción al evento de la Thing, establecemos un canal abierto de comunicación en el que el servicio únicamente tiene que esperar a ser notificado.

5.3.3 Dos o más propiedades - GETs

Puede darse el escenario en el que necesitemos como antecedentes dos o más propiedades de una Thing o propiedades de varias distintas.

Para ello, partimos de varias Things *antecedentes* de las cuales, a través de sus Thing Description, sabemos que podemos leer el estado de una de sus propiedades haciendo una llamada a su API empleando el método GET. A continuación, necesitamos que esas propiedades alcancen un valor determinado, definido como *condiciones.* Si la expresión lógica que integra a todas las Things todas las condiciones son es ciertas, se cumplen, el sistema es capaz de realizar la llamada a la acción de una o varias Things definidas como *consecuentes*.

El pseudocódigo que implementa el microservicio de Ballerina sería el siguiente:

```
Definir el timer
Definir los antecedentes
Definir el/los consecuente(s)
nuevoServicio()
        onTrigger()
                get de loss antecedentes
                if(condiciones==true)
                        put al consecuente(s)
```

Como en este caso sí que estamos ante un tipo de comunicación síncrona entre el Orquestador y las Things antecedentes, se define el *timer.*

Las peticiones a todas las propiedades se realizan en el mismo intervalo de tiempo, no pudiéndose definir en esta versión del proyecto, diferentes frecuencias para las distintas Things. Además, si en una iteración una de las condiciones ya es cierta y las otras no, en la

siguiente iteración se vuelve a comprobar de nuevo el estado de todas las propiedades y condiciones, incluida la que ya era cierta.

5.3.4 Dos o más eventos - MQTTs

En este caso, el sistema quiere suscribirse a dos o más eventos de una misma Thing o a dos o más eventos de diferentes Things.

El algoritmo en este caso realiza lo siguiente. Primero se generan unas variables booleanas, una por cada evento, establecidas a *falsas* y que almacenan si el valor recibido en cada notificación es cierto o no. A continuación se definen los clientes mqtt, el servicio se suscribe a todos los eventos y espera a recibir notificación. Cuando se recibe una notificación se comprueba a qué evento pertenece y, en caso de que el valor sea el deseado, establece la variable booleana a *verdadero.* Cada vez que se reciba un evento y después de actualizar dichas variables, se comprueba el estado de todas ellas. Si la expresión lógica que integra a todas las Things es cierta, el sistema procede a la llamada a la acción de una o varias Things definidas como *consecuentes*.

El pseudocódigo que implementa el microservicio de Ballerina sería el siguiente:

```
Definir el/los consecuente(s)
Definir variables booleanas por cada antecedente
Main()
        Definir los clientes mqtt
        Suscripción a los topics de los clientes
Callback()
        onMessage()
                save payload
                if(condicion==true)
                        variableBooleana = true
                if(expresión=true)
                        put al consecuente(s)
```

En este escenario no se ha definido un intervalo de tiempo en el que, una vez concluido, restablezca a falso el valor de las variables booleanas. Esto quiere decir que, si la notificación de un evento lleva consigo el establecimiento a verdadero de una variable, ésta no va a cambiar a falso a menos que reciba otro evento del mismo topic con un valor distinto del deseado.

Este escenario se ha programado así ya que dependiendo de cómo este implementado el evento podemos encontrarnos ante dos casos. Que la API notifique cada vez que se produce un cambio de estado. Por ejemplo, una Thing que avise de la presencia de una persona en una sala. Podría notificar cuando la persona entra y cuando abandona la sala o sólo notificar cuando entra.

Por ello, en esta primera versión del proyecto, se ha optado por desarrollar el caso en el que la Thing notifica cuando se produce cambio de estado, dejando como trabajo futuro considerar más escenarios.

5.3.5 Propiedades/eventos – GETs/MQTTs

Como quinto, último y caso más complejo tenemos la posibilidad de querer combinar los escenarios anteriores. En este caso tenemos una o más Things de la que queremos leer sus propiedades (GET) y una o más a las que queremos suscribirnos a alguno de sus eventos. Todas ellas denominadas *antecedentes*.

Para este escenario se ha establecido el siguiente algoritmo. Primero se definen unas variables booleanas, una por cada antecedente e inicializadas a *falsas,* que almacenan si la propiedad alcanza el valor establecido o si se recibe la notificación deseada. A continuación aparecen dos vertientes; por un lado se definen los clientes mqtt, el servicio se suscribe a todos los eventos y espera a recibir notificación. Por otro, se crea un *timer* que dispara un servicio con las llamadas a las propiedades.

Tanto cuando se recibe una notificación de evento como cuando se hace una llamada GET a la API, las variables booleanas se actualizan en función de los valores recibidos y, si la expresión lógica definida y que integra a todas ellas es cierta, el sistema procede a la llamada a la acción de una o varias Things definidas como *consecuentes*.

El pseudocódigo que implementa el microservicio de Ballerina sería el siguiente:

```
Definir el timer
Definir el/los consecuente(s)
Definir el/los antecedentes(s)
Definir variables booleanas por cada antecedente
Main()
        Definir los clientes mqtt
        Suscripción a los topics de los clientes
Callback()
        onMessage()
                save payload
                if(condicion==true)
                        variableBooleana = true
                if(expresión=true)
                        put al consecuente(s)
nuevoServicio()
        onTrigger()
                get de loss antecedentes
                if(condiciones==true)
                        variableBooleana = true
                if(expresión=true)
                        put al consecuente(s)
```

CAPÍTULO 6. CASOS PRÁCTICOS REALES

Para probar el sistema implementado se ha desarrollado un escenario con varias Things de la WoT. Una serie de dispositivos que, conectados a una Raspberry Pi, actúan como dispositivos de la WoT. Por un lado, disponemos de una serie de Things antecedentes, Things de las cuales podemos leer propiedades y/o suscribirnos a eventos: Sensor de ultrasonidos, sensor de temperatura, sensor PIR. Por otro, una Thing que actúa como consecuente, un LED, el cual podemos encender/apagar simulando acciones de la Thing.

6.1. Sensores y actuadores

A continuación, detallamos el funcionamiento de cada Thing:

- Sensor de ultrasonidos HC-SR04. Se trata de un sensor que permite detectar distancia a un objeto mediante ultrasonido en un rango de 2 a 450 centímetros. Destaca por su pequeño tamaño, bajo consumo energético y buena precisión [23].

  El sensor HC-SR04 posee dos transductores: un emisor (trigger) y un receptor (echo) piezoeléctricos, además de la electrónica necesaria para su operación. El emisor emite ultrasonidos y cuando rebotan en un objeto u obstáculo que se encuentran en el camino serán captados por el receptor. El circuito se encargará de hacer los cálculos necesarios de ese eco para determinar la distancia.

Figura 6.1. Sensor HC-SR04 [23].

- Sensor de temperatura y humedad relativa DHT. Utiliza un sensor capacitivo de humedad y un termistor para medir el aire circundante y solo un pin para la lectura de los datos. Además dispone de un procesador interno que realiza el proceso de medición, proporcionando la medición mediante una señal digital. El rango de medición de temperatura es de -40°C a 80 °C con precisión de ±0.5 °C y rango de humedad de 0 a 100% RH con precisión de 2% RH, el tiempo entre lecturas debe ser de 2 segundos [24].

Figura 6.2. Sensor DHT [24].

- Sensor PIR (Pasivo Infrarrojo). Los módulos más comunes de este sensor se denominan HC-SR501. Consisten en 2 elementos fundamentales, un lente que hace que los rayos incidentes recaigan sobre un punto y el sensor PIR como tal. El sensor PIR tiene dos elementos sensibles a la luz, que al medirse el retardo de detección entre cada sensor, es posible calcular la distancia a la que se encuentra el objeto [25].

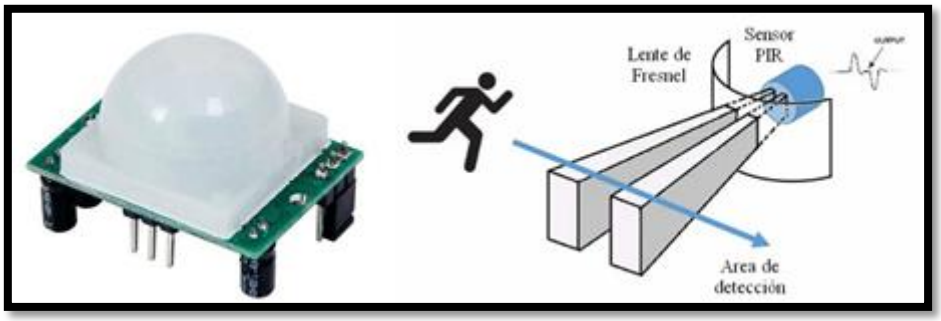


Figura 6.3. Sensor PIR [25].

Los sensores PIR son de tipo ópticos, es decir, se basan en cambios en la radiación electromagnética para sensar el entorno. En lo particular, los sensores PIR trabajan en el rango de la luz infrarroja. Se les conoce como piroeléctricos debido a que los objetos que emiten mayor radiación infrarroja son aquellos cuya temperatura es mayor.

- Led. Se trata de un tipo de diodo que además de permitir el paso de la corriente solo un un sentido, en el sentido en el que la corriente pasa por el diodo, este emite luz. Los Leds tienen dos patillas de conexión una larga y otra corta. Para que pase la corriente y emita luz se debe conectar la patilla larga al polo positivo y la corta al negativo. En caso contrario la corriente no pasará y no emitirá luz [26].

Figura 6.4. LED [26].

En lo que refiere a la Raspberry, se ha utilizado una Raspberry Pi 3 modelo B+ [27]. Lo primero que debemos hacer, antes de realizar las conexiones de nuestras Things, es poner en marcha la raspberry, realizando una serie de configuraciones necesarias.

Empezamos instalando el sistema operativo necesario. La Raspberry es capaz de soportar diversos sistemas operativos pero por razones prácticas para el correcto desarrollo del proyecto, es necesario utilizar una versión Raspbian, siendo aconsejable utilizar la WoT versión de Raspbian [28] que integra por defecto node.js y npm, entre otras tecnologías. Para su instalación, descargamos la imagen para copiarla en una memoria SD. La manera más fácil es utilizar Etcher [29], herramienta empleada para escribir archivos de imagen.

A continuación es necesario conectar a la red la Raspberry, ya sea vía Ethernet o Wi-Fi y habilitar el acceso remoto vía SSH para tener acceso remoto y poder gestionar las APIS de nuestras Things. Para ello, abrimos la ventana de configuración y habilitamos SSH:

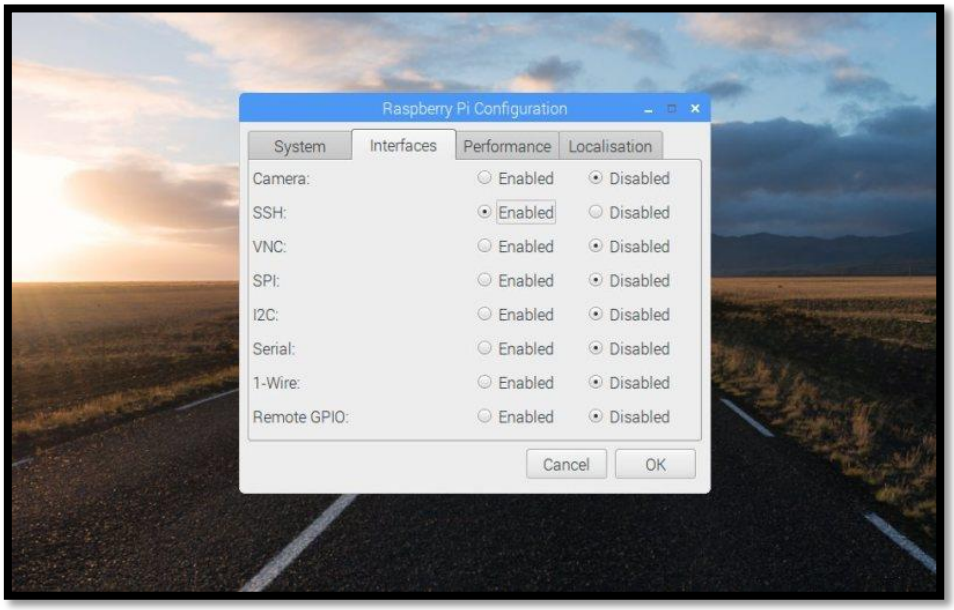


Figura 6.5. Configuración SSH Raspberry.

Como último paso de configuración, es necesario establecer una dirección ip estática para nuestra Raspberry, de forma que siempre esté accesible a través del mismo host. Para ello, necesitamos modificar el archivo de configuración DHCP ubicado en la carpeta etc:

```
sudo nano /etc/dhcpcd.conf
```

En este archivo viene por defecto un ejemplo de configuración de IP estática. Simplemente se tienen que descomentar las líneas y añadir en la propiedad static ip_address la IP que queremos darle.

```
# Example static IP configuration:
interface wlan0
static ip_address=192.168.0.37/24
#static ip6_address=fd51:42f8:caae:d92e::ff/64
static routers=192.168.0.1
static domain_name_servers=192.168.0.1 8.8.8.8 fd51:42f8:caae:d92e::1
```

Figura 6.6. IP estática Raspberry.

Aunque podríamos utilizar el editor de la Raspberry, es más eficiente y cómodo para la implementación del código trabajar desde otro PC y sincronizar dicho código siempre que se quiera probar. Para ello haremos uso de GitHub, instalándolo en nuestra Raspberry desde la terminal y con los siguientes comandos:

```
$ sudo apt-get install git
$ git config --global user.name "soler5"
$ git config --global user.email "juanfrasc5@gmail.com"
```

Por último, son necesarias ciertas librerías para que Node.js pueda acceder y trabajar con la GPIOs. Los valores que se leen o escriben en los pines están disponibles a través de archivos. Lo más cómodo es utilizar un de las diversas librerías que ofrecen la capa de abstracción y funcionalidad necesaria para trabajar con estos archivos. Una de ellas es *onoff* [30] la cual instalaremos haciendo uso de NPM [16].

```
$ npm install onoff –save
```

Para trabajar con el sensor de temperatura y humedad, necesitamos instalar la librería *dht* [31] de node.

```
$ npm install node-dht-sensor –save
```

Una vez concluidos los pasos previos, se reinicia la Raspberry Pi concluyendo todo lo necesario respecto a su configuración propia. La Raspberry ya está lista para ser utilizada en la Web Of Things aunque no tiene mucho con lo que trabajar en el mundo real. Para ello se realizan las conexiones de sensores y actuadores citados previamente.

Cabe destacar y realizar un breve análisis sobre los puertos GPIO (General Purpose Input/Output) y la manera en la que se van a emplear en este proyecto. GPIO es, en esencia, un pin que puede actuar para leer o escribir datos. Por ello tienen dos modos: un modo de entrada y un modo de salida. Cuando se selecciona el modo de salida, el pin se puede configurar en ALTO, lo que significa que produce 3.3 voltios; cuando el pin está configurado en BAJO, está apagado y no emite ningún voltaje.

La numeración del GPIO varía en función del modelo de la Raspberry. En la siguiente figura, se enumeran todos los pines disponibles de este modelo de Raspberry donde se puede comprobar que la numeración de los pines no corresponde con el número de GPIO. Por ejemplo, el pin 12 no se refiere al GPIO12, si no al GPIO18.

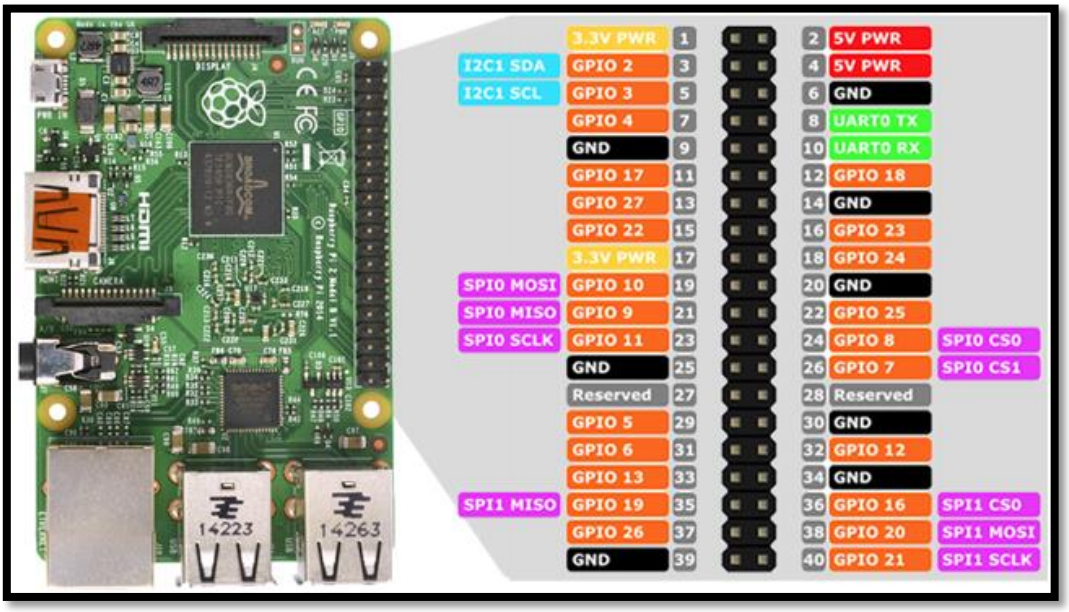


Figura 6.7. GPIO Raspberry [6].

Para realizar el montaje y conexión de todos los componentes se han utilizado protoboards, jumpers y resistencias eléctricas. A continuación se detalla el proceso de conexión de todas las Things.

En la figura 6.8 se puede observar la conexión de la Thing que actúa como consecuente, el LED, donde se han utilizado los pines 7 y 6 correspondientes al GPIO4 y GND (tierra). Además, se ha utilizado una resistencia de 220 Ohmios para evitar que el LED se sobrecaliente y quede inutilizable.

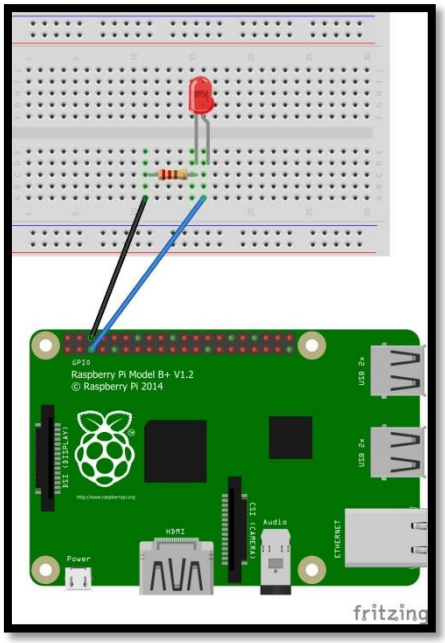


Figura 6.8. Montaje actuador LED.

Para el sensor de ultrasonidos HC-SR04 se han utilizado los pines 1, 16, 18 y 39, correspondientes a 3.3V, GPIO23, GPIO24 y GND junto con 2 resistencias de 220 Ohmios. La salida del Trigger (que manda el pulso) se utiliza en el GPIO23 y el echo (quien recibe el pulso) en el GPIO24.

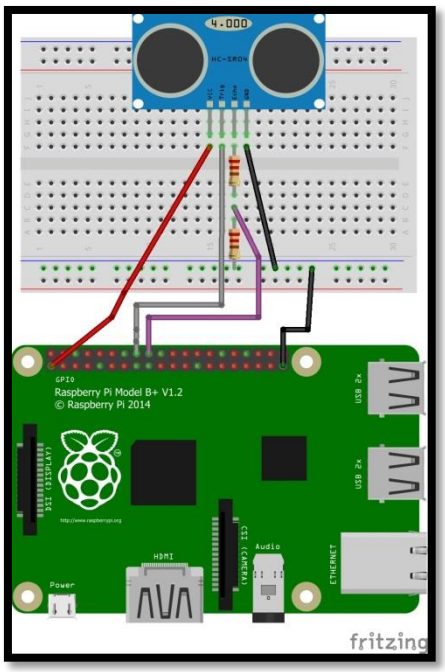


Figura 6.9. Montaje sensor HC-SR04.

Como se puede ver en la siguiente figura, el sensor de infrarrojos pasivos (PIR) tiene tres pines: uno para la fuente de alimentación de 5V (VCC) uno para el valor digital del sensor (OUT) y otro para tierra (GND). Estos pines se han conectado a los pines 4, 13 y 39 de la Raspberry correspondientes a 5V, GPIO27 y GND.

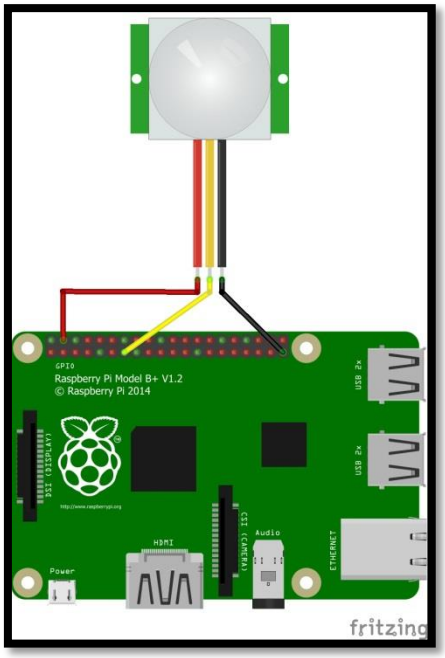


Figura 6.10. Montaje sensor PIR.

Por último, el sensor de temperatura y humedad DHT22 como podemos ver en la siguiente figura dispone de cuatro pines aunque solo se emplean tres. De derecha a izquierda: el primer pin para tierra (GND), el tercero para la señal conectado al pin32 (GPIO12) y a 5V a mediante una resistencia y el último a 5V (VCC).

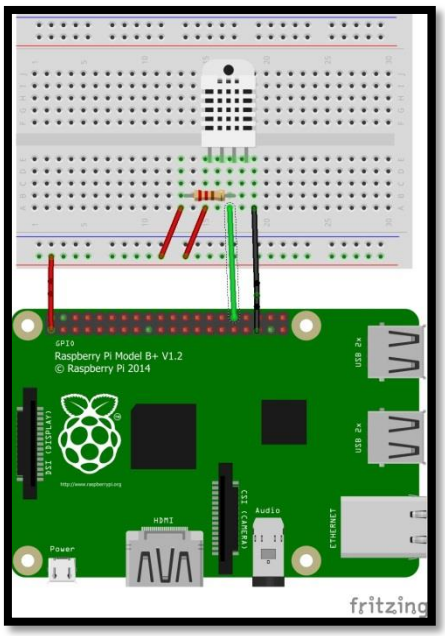


Figura 6.11. Montaje sensor DHT.

A modo de resumen (figura 6.12) podemos apreciar en su conjunto todos los sensores y el actuador que simulan las cuatro Things.

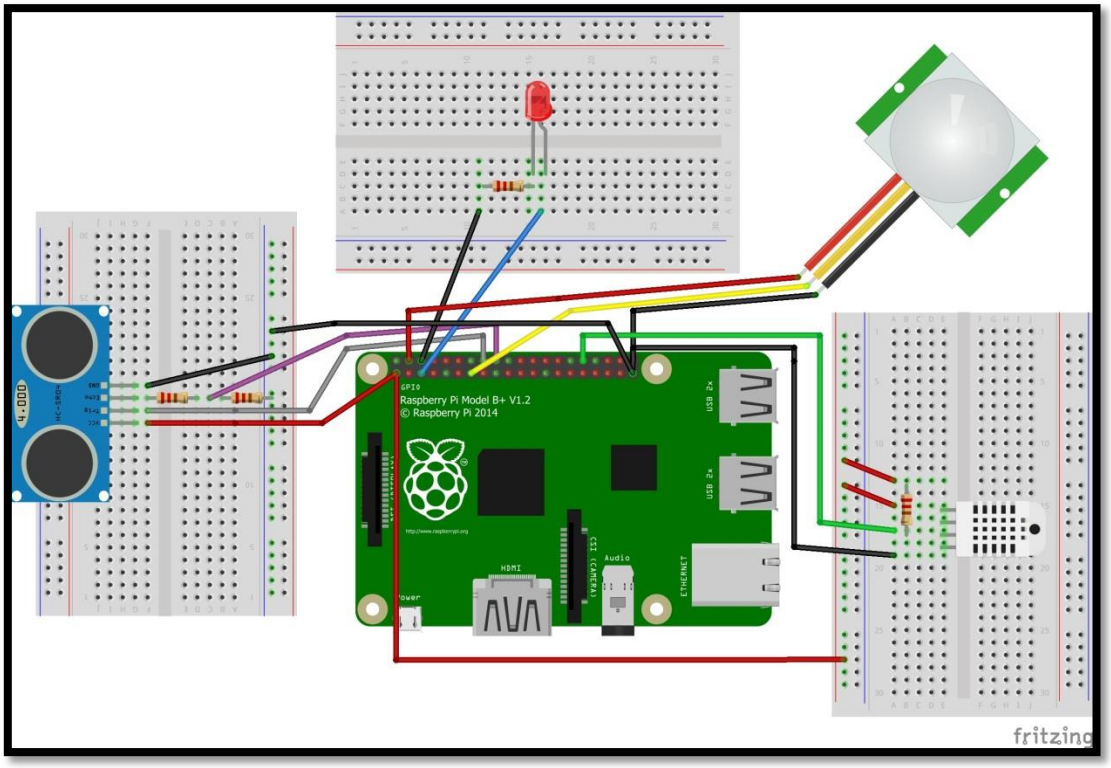


Figura 6.12. Montaje global simulado.

Por último y para hacer uso del protocolo MQTT, se necesita implementar un servidor MQTT en la Raspberry Pi. Para ello instalamos la librería *Mosquitto* [32] con el siguiente comando.

```
$ sudo apt install mosquitto
$ sudo systemctl enable mosquitto
$ sudo systemctl start mosquitto
```

Con el segundo comando, indicamos que se inicie automáticamente cuando la raspberry se reinicia. Con el tercero, iniciamos MQTT.

En el anexo 3 están disponibles las fotografías reales del circuito.

6.2 Implementación de las APIs.

Todos y cada uno de estos dispositivos se han conectado a la web y por ello son dispositivos de la WoT empleando APIs RESTful y siguiendo las pautas marcadas por Guinard [6]. Estas APIs se han construido utilizando las tecnologías Node.js y Express, tecnologías vistas en el desarrollo del máster.

Cada una de las Things se ha implementado de manera independiente. Por ello se dispone de una API por cada Thing ejecutada por consiguiente en un puerto distinto de la Raspberry Pi. De esta manera cada Thing será accesible a través de protocolos de Internet (TCP/IP) ya que están alojador en un servidor HTTP.

La estructura de todas las Apis es la siguiente:

- **wot-server.js**. Es el fichero principal de nuestro proyecto. Actúa como punto de entrada de nuestra API y es el encargado de arrancar el servidor con la correcta configuración.
- **package.json**. En este fichero se definen los módulos que vamos a necesitar en nuestra aplicación.
- **servers/**. Contiene el fichero encargado de cargar las vistas en el servidor HTTP. Se trata básicamente de un servidor HTTP dentro del framework Express.
- **routes/**. Los controladores se definen con convenio en los ficheros route en donde establecemos la función que se debe ejecutar en función del método y la url invocada.
- **resources/**. En esta carpeta se alojan archivos JSON como la Thing Description o el archivo resources.json que contiene y almacena información (como puerto a utilizar y el GPIO asociado) y el estado de las propiedades de la Thing.
- **plugins/.** Contiene el fichero encargado de actualizar el modelo de datos cada vez que estos se leen del sensor.
- **middleware/**. Contiene el fichero encargado de realizar las conversiones de representación para poder utilizar tanto HTML como JSON según desde donde se realice la petición.

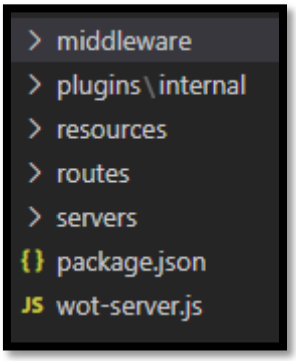


Figura 6.13. Estructura de la API.

A modo de ejemplo estudiaremos la API RESTful del sensor de ultrasonidos HC-SR04 ya que se trata del sensor más completo al implementar MQTT y tener una propiedad accesible a través de una ruta.

Lo primero de todo es definir el modelo de datos (figura 6.14), archivo que guardará el nombre de la Thing, la descripción, el puerto en el que se ejecuta y el sensor que dispone. El sensor a su vez contiene el nombre, descripción, un valor inicial y el cual se modificará cuando el valor del sensor varíe, los pines GPIO que emplea y la unidad de medida.

```
{
  "pi": {
    "name": "HC Sensor Pi",
    "description": "A simple WoT-connected Raspberry PI for the WoT book.",
    "port": 8488,
    "sensors": {
      "hc": {
        "name": "US sensor",
        "description": "A passive ultrasonic sensor. Send distance",
        "value": 0,
        "unit": "mm",
        "trigger": 23,
        "echo": 24
      }
    }
  }
}
```

Figura 6.14. Modelo de datos de una Thing.

A continuación definimos la Thing Description en otro fichero JSON. Aquí se indican, como datos más importantes, las propiedades, eventos y acciones de la Thing y la manera de interactuar con ellos. En la siguiente figura, observamos que la Thing tiene una propiedad numérica y de sólo lectura llamada distancia y que es accesible a través de una URL. A su vez dispone de un evento llamado short-distance el cual emplea el protocolo MQTT y notifica en caso de detectar una distancia inferior a 50mm.

```
{
    "title": "MyPirSensor",
    "id": "urn:dev:ops:32473-WoTPirSensor-1234",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "properties": {
        "distance": {
          "type": "number",
          "forms": [
            {
              "href": "http://192.168.0.37:8488/sensors/pir",
              "op": ["readproperty"],
              "contentType": "application/json"
            }
          ]
        }
      },
    "events": {
        "short-distance": {
            "data":{
              "type": "json",
              "description":"Devuelve un JSON con dos atributos: value y date. Value es un booleano que indica corta distancia y date contiene la fecha de la notificación"
            },
            "forms": [
                {
                    "href": "mqtt://192.168.0.37:1883",
                    "contentType" : "text/plain",
                    "op" : "subscribeevent"
                }
            ]
        }
    }
}
```

Figura 6.15. Thing Description HC-SR04.

Como paso siguiente, asignamos los recursos de la Thing a la URL que responde el servidor. En Express, la URL de un recurso de define mediante una ruta, la cual se ha definido dentro de la carpeta rutas/ en el archivo sensors.js. Aquí podemos ver que disponemos de dos rutas, la primera corresponde a la raíz (/) a través de la cual está accesible la Thing Description. La segunda (/hc) devuelve la propiedad distance de nuestro sensor.

```
var express = require('express'),
  router = express.Router(),
  resources = require('./../resources/model');
  var td = require('./../resources/td.json');

router.route('/').get(function (req, res, next) {
  req.result = td;
  next();
});

router.route('/hc').get(function (req, res, next) {
  req.result = resources.pi.sensors.hc;
  next();
});
module.exports = router;
```

Figura 6.16. Archivo sensors.js.

Una vez que las rutas están listas, se deben cargar dentro del servidor HTTP, lo cual se hace en el archivo servers/http.js. Este archivo es, en definitiva, el servidor HTTP envuelto en el framework Express.

```
var express = require('express'),
  sensorRoutes = require('./../routes/sensors'),
  converter = require('./../middleware/converter'),
  cors = require('cors'),

var app = express();

app.use(cors());

app.use('/sensors', sensorRoutes);

// For representation design
app.use(converter());
module.exports = app;
```

Figura 6.17. Archivo http.js.

Como hemos visto anteriormente, la estructura de la API tiene como punto de entrada el archivo wot-server.js el cual inicia el servicio en el puerto deseado y se encarga de lanzar el servidor y el plugin encargado de actualizar los datos captados por el sensor.

```
// Final version
var httpServer = require('./servers/http'),
resources = require('./resources/model');

// Internal Plugins
var hcPlugin = require('./plugins/internal/hcPlugin'); //#A

// Internal Plugins for sensors/actuators connected to the PI GPIOs
// If you test this with real sensors do not forget to set simulate to 'false'
hcPlugin.start({'simulate': false, 'frequency': 10000}); //#B

// HTTP Server
var server = httpServer.listen(resources.pi.port, function () {
  console.log('HTTP server started...');
  console.info('Your WoT Pi is up and running on port %s', resources.pi.port);
});
```

Figura 6.18. Archivo wot-server.js.

En la siguiente figura podemos ver el plugin del sensor. Se trata del archivo más importante de la API ya que es el encargado de actualizar los datos en el modelo y de publicar el evento en el servidor MQTT. Al comienzo se definen todas las variables, donde destaca *client* que es el cliente MQTT ubicado en el puerto 1883 de la Raspberry Pi. Contiene la función *start* y *stop* encargadas de arrancar y detener el plugin y la función *connectHardware* la cual establece la comunicación con los pines GPIO correspondientes y actualiza el modelo en base a lo que capta el sensor utilizando los modelos matemáticos correspondientes. Además, si se cumple con el evento establecido, hace un *publish* con el topic *short-distance* alertando de ésta a los suscriptores. Por último se crea un intervalo para determinar la frecuencia con la que el sensor comprueba la nueva distancia.

```
var mqtt = require('mqtt');
var resources = require('./../../resources/model');
var client = mqtt.connect("mqtt://192.168.0.37:1883");
var interval, sensor;
var model = resources.pi.sensors;
var pluginName = 'US sensor';
var localParams = {'simulate': false, 'frequency': 5000};

exports.start = function (params) {
  localParams = params;
  client.on('connect', function () {
    connectHardware();
  });
};

exports.stop = function () {
  if (params.simulate) {
    clearInterval(interval);
  } else {
    sensor.unexport();
  }
  console.info('%s plugin stopped!', pluginName);
};

function connectHardware() {

  const Gpio = require('pigpio').Gpio;
  const MICROSECONDS_PER_CM = 1e6/34321;

  const trigger = new Gpio(23, {mode: Gpio.OUTPUT});
  const ECHO = new Gpio(24, {mode: Gpio.INPUT, alert: true});

  trigger.digitalWrite(0);

  const warchHCSR04 = () =>{
    var starTick;
    ECHO.on('alert', (level, tick)=>{
      if (level==1){
        starTick = tick;
      }else{
        const endTick = tick;
        const diff = (endTick>>0) - (starTick>>0);
        model.hc.value = diff/2/MICROSECONDS_PER_CM;
        var date = new Date();

        if(model.hc.value<50){
          client.publish('short-distance', '{"value":1,"date":"'+ date.getDay()+'/'+date.getMonth()+'/'+date.getFullYear()+' '+date.getHours()+':'+date.getMinutes()+':'+date.getSeconds()+'"}');

        }else{
          client.publish('short-distance', '{"value":0,"date":"'+ date.getDay()+'/'+date.getMonth()+'/'+date.getFullYear()+' '+date.getHours()+':'+date.getMinutes()+':'+date.getSeconds()+'"}');
        }
      }
    })
  }
  warchHCSR04();

  setInterval(()=>{
    trigger.trigger(10, 1);
  }, 1000);
};
```

Figura 6.19. Plugin hcPlugin.js.

Para poner en marcha la API de la Thing debemos ejecutar el comando *node wot-server.js*

Ahora se puede realizar la petición GET para obtener la propiedad. Dependiendo de si realizamos la petición desde un navegador o desde un gestor de peticiones como Postman, obtendremos los datos con un formato distinto.

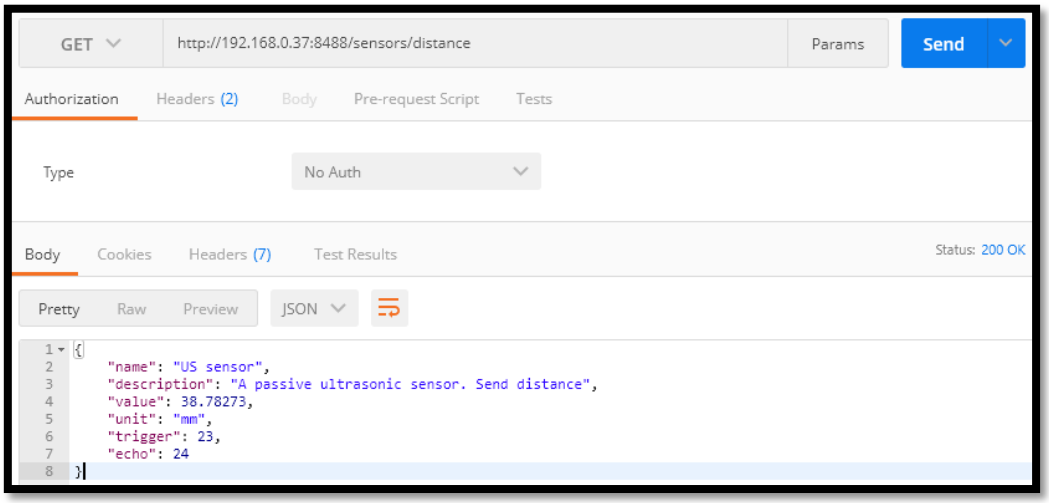


Figura 6.20. Petición desde Postman.

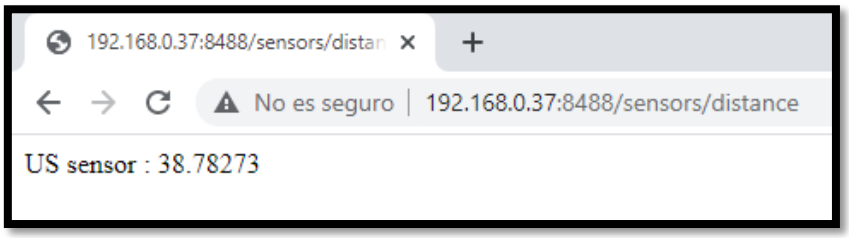


Figura 6.21. Petición desde navegador.

Para probar que la notificación del evento mediante MQTT funciona correctamente se hace uso de Mosquitto, uno de los broker MQTT más populares. Una vez instalado utilizamos el siguiente comando (Figura 6.22) para suscribimos al evento *short-distance* del host de la Raspberry Pi y comprobar que, efectivamente, estamos siendo notificados.

```
C:\Program Files\mosquitto>mosquitto_sub -h 192.168.0.37 -t short-distance
{"value":0,"date":"1/5/2021 13:51:30"}
{"value":1,"date":"1/5/2021 13:51:31"}
```

Figura 6.22. Suscripción desde Mosquitto.

De forma similar se implementan las demás APIs RESTFUL de las Things, quedando el esquema de la figura siguiente como resumen a nivel de URLs de todas las APIs.

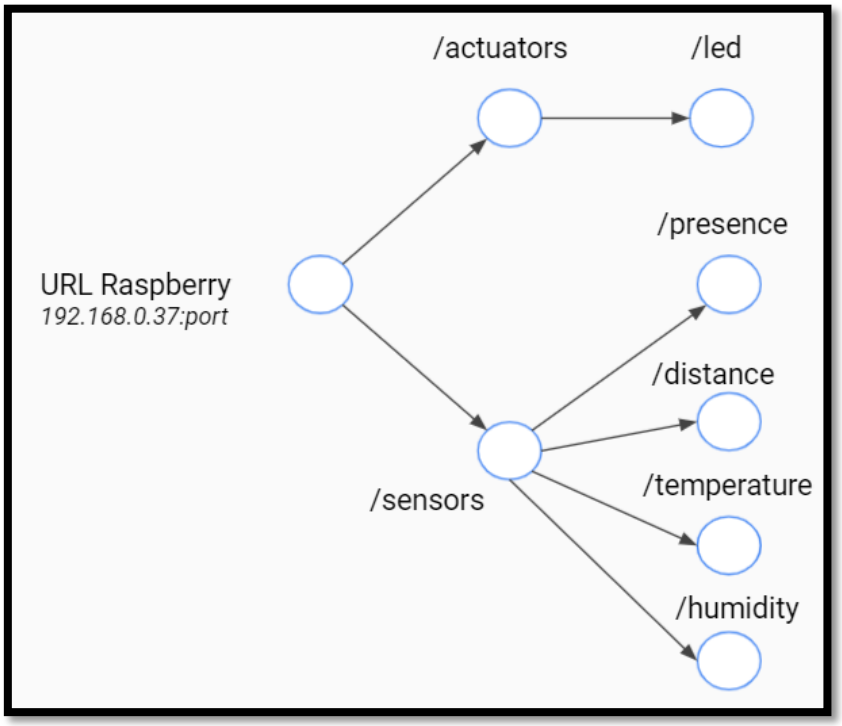


Figura 6.23. Esquema APIs Raspberry.

Una vez implementadas las APIs, las Things ya son accesibles a través de la WoT. Es momento de llevar a cabo los ensayos prácticos y reales utilizando diferentes escenarios. Por ello, y como muestra la figura 5.24 los sensores serán las Things antecedentes y el LED La Thing consecuente. Teniendo esto en cuenta, el orquestador deberá llamar al consecuente siempre y cuando se cumpla la expresión lógica impuesta que engloba propiedades y/o eventos de los sensores.

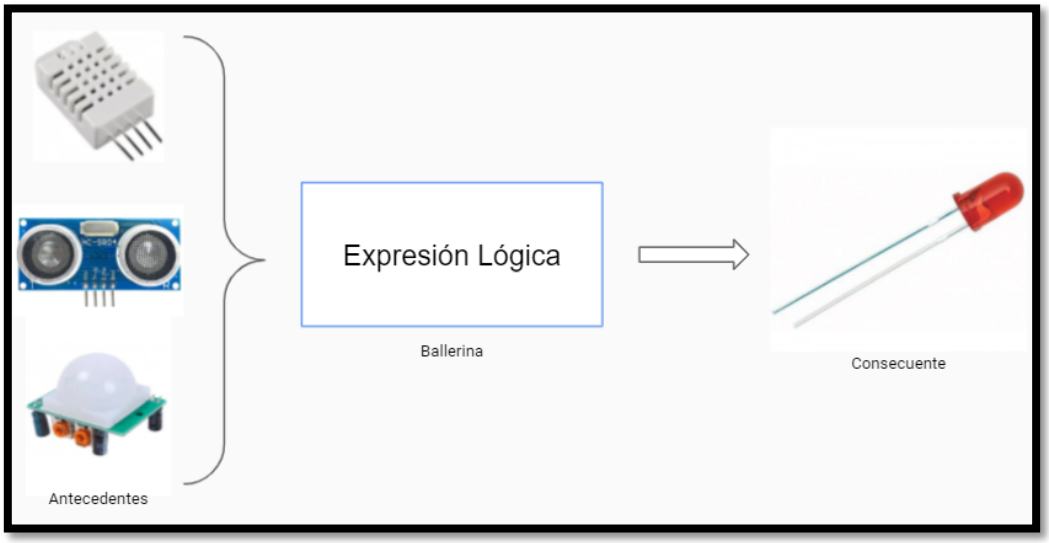


Figura 6.24. Esquema Things.

Todas las APIs se encuentran alojadas en un proyecto Github [33].

6.3 Escenarios propuestos

En este punto se muestran todas las pruebas llevadas a cabo para validar las distintas orquestaciones de las Things en función de sus propiedades, eventos y acciones. Se analizan los cinco casos independientes y desarrollados en el Orquestador.

6.3.1 Ejemplo en WSO2 Integration Studio.

Para demostrar el funcionamiento de la primera arquitectura probada del proyecto, se generó un caso práctico con Integration Studio de WSO2.

En este ejemplo se muestra un caso práctico en el que el sistema se suscribe a un evento MQTT de la Thing que emplea el sensor PIR y publica en el broker cada vez que detecta presencia en una sala. Cuando se detecte presencia se quiere llamar a la acción de otra Thing, en concreto encender un LED.

Lo primero que necesitamos es definir el endpoint a utilizar, el del LED, estableciendo el nombre la URI y el método, tal y como se puede apreciar en la figura siguiente.

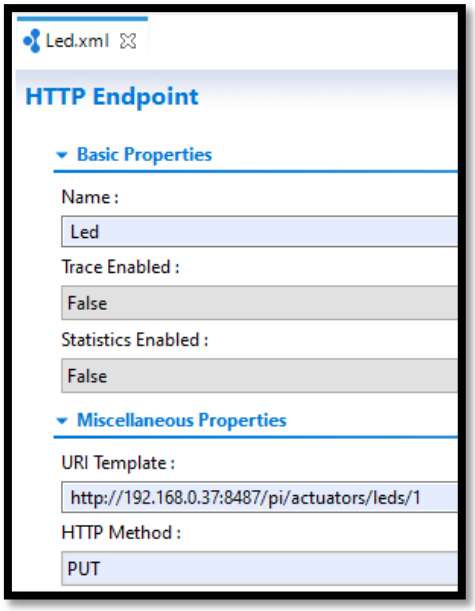


Figura 6.25. Endpoints LED.

A continuación, definimos el inbound-endpoint del sensor PIR. Como se puede apreciar en la siguiente figura (Figura 6.26), se establecen los parámetros del Host, puerto, tema o topic y el tipo (MQTT). Este tipo de inbound debe ir proseguido de una secuencia en caso de recibir correctamente notificación y otra secuencia de error en caso de algún fallo en el sistema.

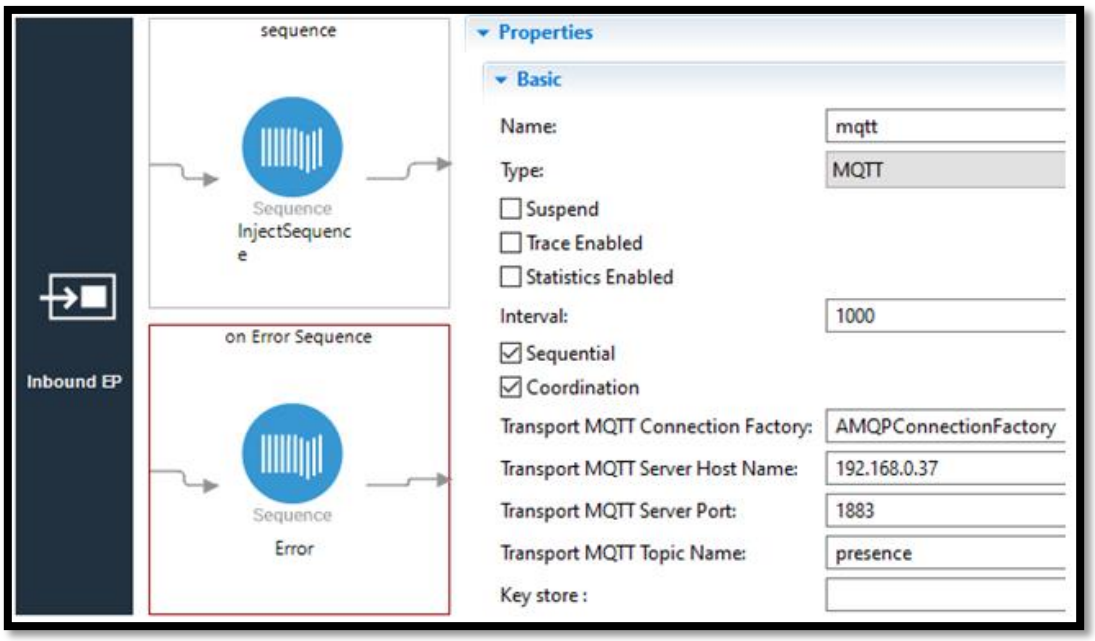


Figura 6.26. Indbound-enpoints sobre MQTT.

Una vez recibido el evento, el sistema procede a ejecutar la secuencia llamada *InjectSequence,* la cual tal y como se puede ver en la Figura 6.27 realiza como primer paso un LOG sobre la propiedad *presencia* que almacena el valor recibido por el evento. A continuación y, mediante un filtro, si el valor recibido es *true* reaiza la llamada al endpoint

del LED, pasándole como payload el valor *true* con lo que el LED se encenderá. En caso de recibid un valor de presencia falso, se manda a la acción un valor de payload *false* y el LED se apaga.

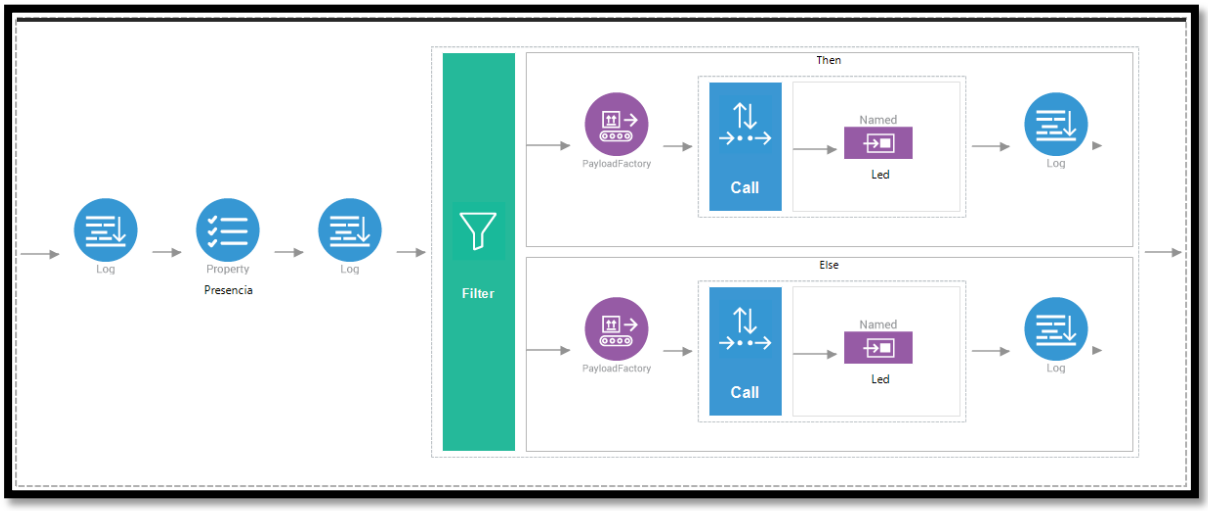


Figura 6.27. Secuencia InjectSequence.

6.3.2 Una propiedad - GET.

Como primer escenario práctico que valide el sistema, se ha utilizado el sensor de ultrasonidos y el LED como Things. El sensor actúa como Thing antecedente y el LED como Thing consecuente. Se pretende que el LED se encienda cuando la distancia obtenida por el sensor sea inferior a 10 cm.

El valor de la distancia se es una propiedad de la Thing y se obtiene haciendo una petición GET a su API. Si la propiedad obtenida es inferior a 10cm, el LED se enciende. Además como se trata de un caso sincrónico, se establece un intervalo de 1000 ms como frecuencia entre peticiones.

Los pasos a seguir son los siguientes:

1. Crear el archivo de configuración siguiendo el manual del Anexo 1 y analizando las Things Descriptions correspondientes.
2. Acceder a la URL del servicio PARSER y cargar el archivo de configuración.
3. Una vez aquí hay dos opciones:
   a) Ejecutar directamente el servicio ORQUESTADOR.
   b) Descargar el archivo y ejecutar por nuestra cuenta.
4. Comprobar funcionamiento. El LED se enciende al aproximar un objeto a menos de 10 cm.

El archivo de configuración contiene dos líneas. La primera indicando la Thing antecedente y la segunda, la consecuente. En este caso no es necesario introducir una línea con la lógica ya que no podemos utilizar expresiones lógicas al tener un solo antecedente, un solo argumento de entrada.

Como se observa a continuación, en la Thing antecedente se incluye el GET seguido de la URL y de la expresión que queremos que cumpla. *Value* es el argumento con el valor de la propiedad que se recibe al hacer el GET sobre la API. El último número (1000) corresponde a la frecuencia con la que se harán peticiones al antecedente. Por tanto, si se cumple la expresión de la propiedad, se ejecuta el POST sobre la Thing pasando como cuerpo (body) de la petición el JSON *"value":true*, lo que hará encender el LED.

get http://192.168.0.37:8485/sensors/ultrasounds value<70 1000

post http://192.168.0.37:8487/actuators/leds/1 {"value":true}

En la figura 6.28 se puede ver el diagrama del ejemplo práctico donde el Orquestador hace la petición y, una vez evaluada la propiedad, procede a realizar la llamada (POST) a la acción de encender el LED o volver a hacer el GET a la API para obtener la nueva distancia captada por el sensor.

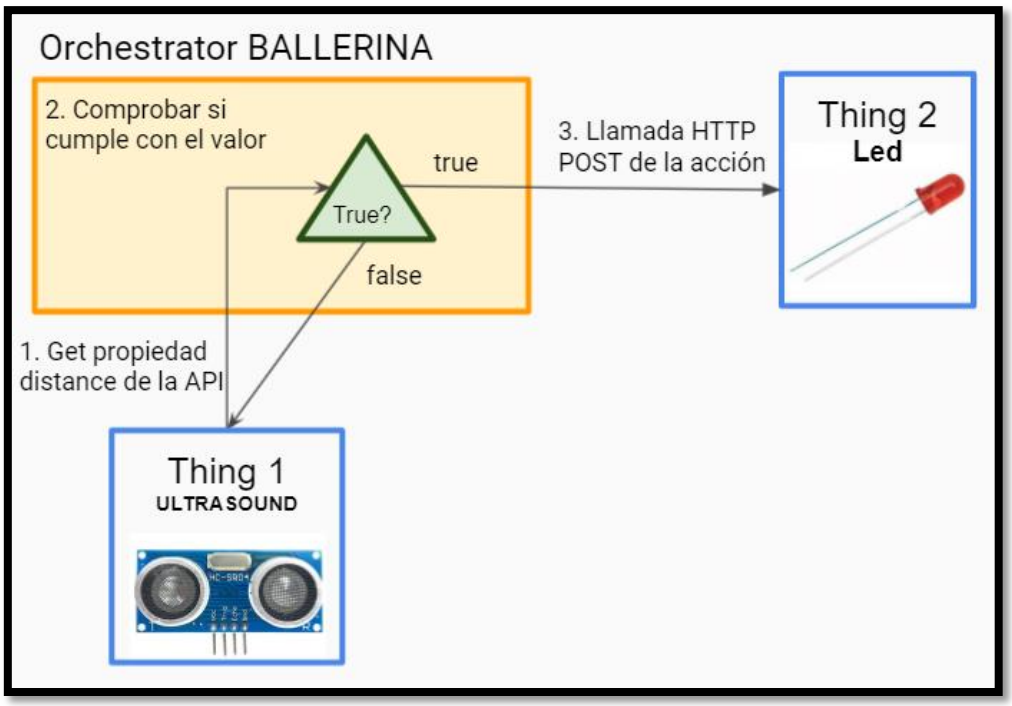


Figura 6.28. Ejemplo práctico GET.

Para demostrar y validar este caso práctico en concreto, se ha grabado un video [34] explicativo dónde se puede comprobar el funcionamiento del Parser y del servicio Orquestador de Ballerina.

6.3.3 Un evento – MQTT

Como segundo escenario práctico que valide el sistema, se ha utilizado el sensor de ultrasonidos y el LED como Things. El sensor actúa como Thing antecedente y el LED como Thing consecuente. Se pretende que el LED se encienda cuando el sistema sea notificado mediante el evento short-distance que notifica cuando la distancia detectada por el sensor es inferior a 50mm.

Este caso se desarrolla suscribiéndonos al topic short-distance de la THing. En el momento de ser notificados, si el valor de la notificación es true, el LED se enciende. Además como se trata de un caso asincrónico, no se necesita establecer un intervalo de tiempo ya que siempre estaremos preparados para recibir la notificación.

El archivo de configuración contiene dos líneas. La primera indicando la Thing antecedente y la segunda, la consecuente. En este caso no es necesario introducir una línea con la lógica ya que no podemos utilizar expresiones lógicas al tener un solo antecedente, un solo argumento de entrada.

Como se observa a continuación, en la Thing antecedente se incluye el MQTT seguido del host y de la expresión que queremos que cumpla. *Value* es el argumento con el valor del evento recibido que se recibe como notifiación al suscribirnos al topic short-distance. Por tanto, al recibir notifiación y comprobar que el valor es el deseado, se ejecuta el POST sobre la Thing pasándo como cuerpo (body) de la petición el JSON *"value":true*, lo que hará encender el LED.

mqtt 192.168.0.37 short-distance value==1

post http://192.168.0.37:8487/actuators/led {"value":true}

En la figura 6.29 se puede ver el diagrama del ejemplo práctico donde el Orquestador se suscribe al evento y, una vez recibida y evaluada la notificación, procede a realizar la llamada (POST) a la acción de encender el LED.

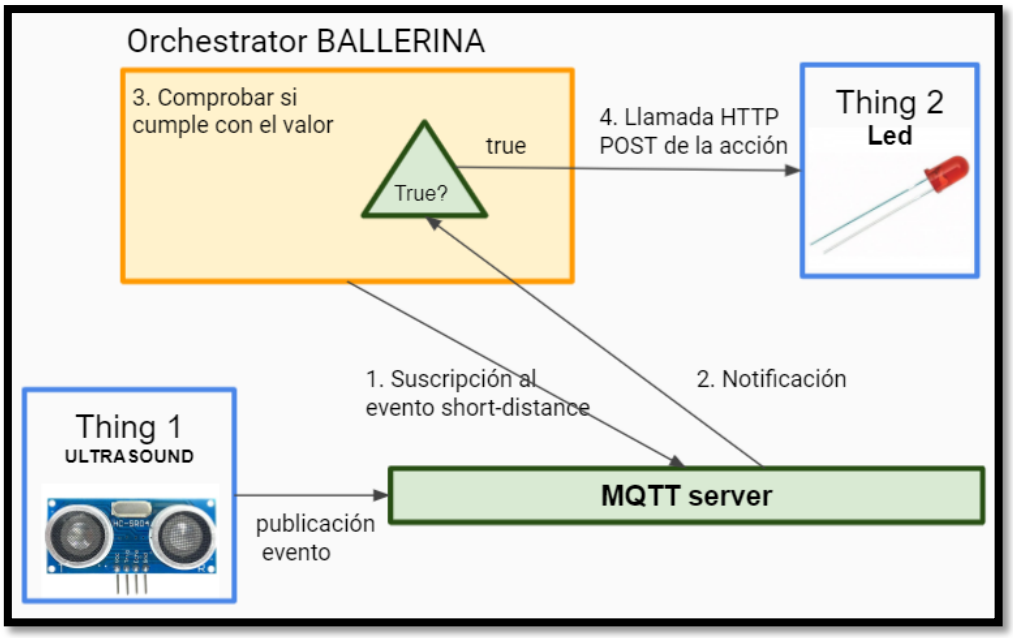


Figura 6.29. Ejemplo práctico MQTT.

Para demostrar y validar este caso práctico en concreto, se ha grabado un video [35] explicativo dónde se puede comprobar el funcionamiento del Parser y del servicio Orquestador de Ballerina.

6.3.4 Dos propiedades – GETs

Como tercer escenario práctico que valide el sistema, se ha utilizado el sensor de ultrasonidos, el sensor de temperatura y el LED como Things. Los dos sensores actúan como Things antecedentes y el LED como Thing consecuente. Se pretende que el LED se encienda cuando la distancia obtenida por el sensor sea inferior a 10 mm y la temperatura sea inferior a 30ºC.

Estas dos propiedades se obtienen haciendo una petición GET a la API de la Thing. Además como se trata de un caso sincrónico, se establece un intervalo de 1000 ms como frecuencia entre peticiones.

El archivo de configuración contiene tres líneas. La primera indicando las Things antecedentes, la segunda con la expresión lógica deseada (operador &&) y la tercera con la Thing consecuente.

Como se observa a continuación, en las Things antecedentes se incluye el GET seguido de la URL y de la expresión que queremos que cumpla. *Value* es el argumento con el valor de la propiedad que se recibe al hacer el GET sobre la API. Por tanto, si se cumple la expresión indicada (1&&2) es decir, ambas propiedades cumplen con lo esperado, se ejecuta el POST sobre la Thing pasando como cuerpo (body) de la petición el JSON *"value":true*, lo que hará encender el LED.

get http://192.168.0.37:8485/sensors/temperature value<30 1000 , get http://192.168.0.37:8488/sensors/distance value<10 1000

1 && 2

post http://192.168.0.37:8487/actuators/led {"value":true}

En la figura 6.30 se puede ver el diagrama del ejemplo práctico donde el Orquestador hace las dos peticiones y, una vez evaluadas las propiedades, procede a realizar la llamada (POST) a la acción de encender el LED o volver a hacer los GETs a las APIs para obtener la nueva distancia y la nueva temperatura captada por los sensores.

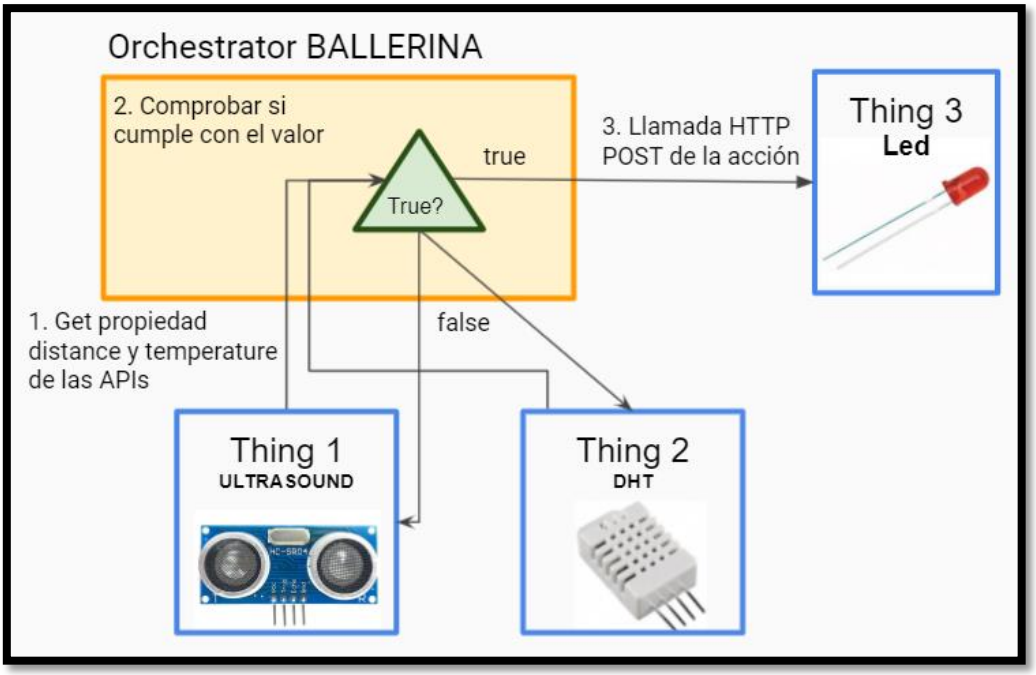


Figura 6.30. Ejemplo práctico GETs.

Para demostrar y validar este caso práctico en concreto, se ha grabado un video [36] explicativo dónde se puede comprobar el funcionamiento del Parser y del servicio Orquestador de Ballerina.

6.3.5 Dos eventos – MQTTs

Como cuarto escenario práctico que valide el sistema, se ha utilizado el sensor de ultrasonidos, el sensor PIR y el LED como Things. Los sensores actúan como Things antecedentes y el LED como Thing consecuente. Se pretende que el LED se encienda cuando el sistema sea notificado mediante el evento short-distance del sensor de ultrasonidos y el evento presence del sensor PIR. El primero notifica cuando la distancia detectada por el sensor es inferior a 50mm, el segundo notifica cuando detecta presencia.

Este caso se desarrolla suscribiéndonos al topic short-distance y presence de ambas Things. En el momento de ser notificados, si ambos valores de notificación son true, el LED se enciende. Además como se trata de un caso asincrónico, no se necesita establecer un intervalo de tiempo ya que siempre estaremos preparados para recibir la notificación.

El archivo de configuración contiene tres líneas. La primera indicando las Things antecedentes, la segunda con la expresión lógica y la tercera con la Thing consecuente.

Como se observa a continuación, en las Things antecedentes se incluye el MQTT seguido del host y de la expresión que queremos que cumpla. *Value* es el argumento con el valor del evento recibido que se recibe como notificación al suscribirnos al topic short-distance. Por tanto, si se cumple la expresión indicada (1&&2) es decir, ambos eventos notifican y cumplen con lo esperado, se ejecuta el POST sobre la Thing pasando como cuerpo (body) de la petición el JSON *"value":true*, lo que hará encender el LED.

mqtt 192.168.0.37 presence value==1 , mqtt 192.168.0.37 short-distance value==1

1 && 2

post http://192.168.0.37:8487/actuators/led {"value":true}

En la figura 6.31 se puede ver el diagrama del ejemplo práctico donde el Orquestador se suscribe a los dos eventos y, una vez recibidas y evaluadas las notificaciones, procede a realizar la llamada (POST) a la acción de encender el LED.

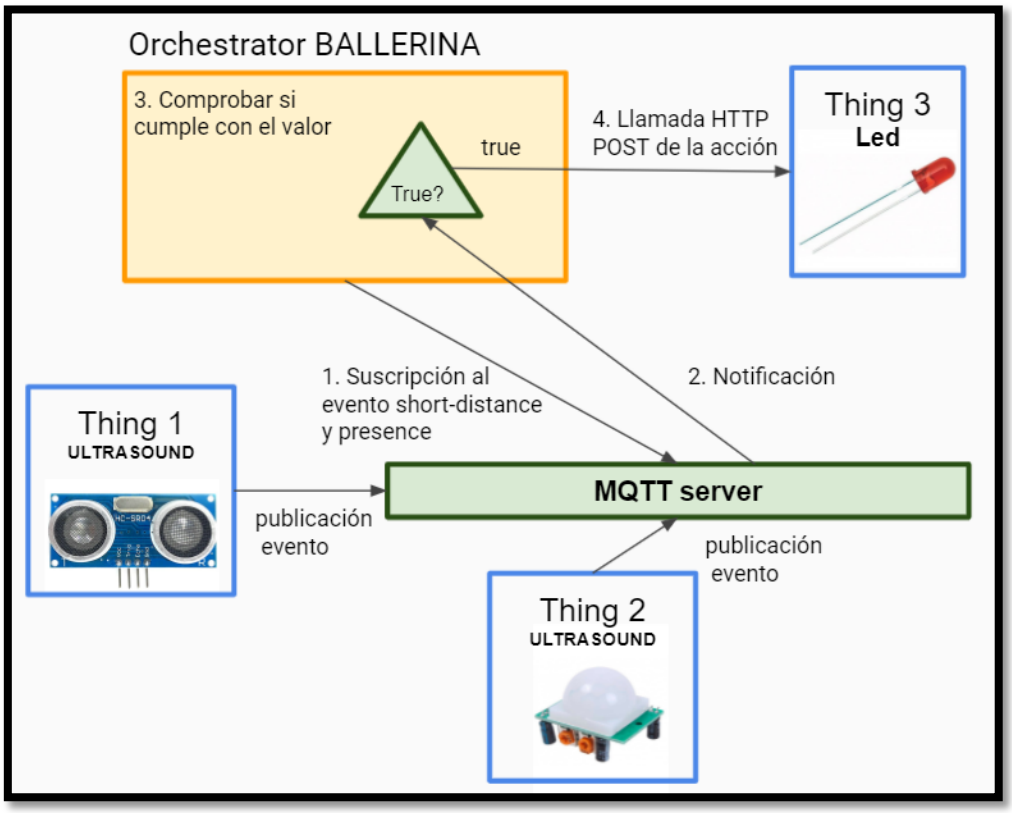


Figura 6.31. Ejemplo práctico MQTTs.

Para demostrar y validar este caso práctico en concreto, se ha grabado un video [37] explicativo dónde se puede comprobar el funcionamiento del Parser y del servicio Orquestador de Ballerina.

6.3.6 Propiedades/eventos – GETs/MQTTs

Como quinto y último escenario práctico que valide el sistema, se ha utilizado el sensor de ultrasonidos, el sensor PIR y el LED como Things. Los sensores actúan como Things antecedentes y el LED como Thing consecuente. Se pretende que el LED se encienda cuando el sistema sea notificado mediante el evento short-distance del sensor de ultrasonidos y la propiedad temperature del sensor dht tenga un valor inferior a 30º.

Este caso se desarrolla suscribiéndonos al topic short-distance y simultáneamente realizando peticiones GET sobre la API del sensor de ultrasonidos. En el momento de ser

notificados o de obtener la propiedad, si ambos valores (notificación y evento) son true y se cumplen, el LED se enciende.

El archivo de configuración contiene tres líneas. La primera indicando las Things antecedentes, la segunda con la expresión lógica y la tercera con la Thing consecuente.

Como se observa a continuación, en las Things antecedentes se incluye el MQTT seguido del host y de la expresión que queremos que cumpla y, por otro lado, el GET seguido de la URL y de la expresión que queremos que cumpla. Por tanto, si se cumple la expresión indicada (1&&2) es decir, el eventos recibido y la propiedad leída cumplen con lo esperado, se ejecuta el POST sobre la Thing pasando como cuerpo (body) de la petición el JSON *"value":true*, lo que hará encender el LED.

mqtt 192.168.0.37 presence value==1 , get 192.168.0.37 HC-SR04 value==1

1 && 2

post http://192.168.0.37:8487/actuators/led {"value":true}

En la figura 6.32 se puede ver el diagrama del ejemplo práctico donde el Orquestador se suscribe a los dos eventos y, una vez recibidas y evaluadas las notificaciones, procede a realizar la llamada (POST) a la acción de encender el LED.

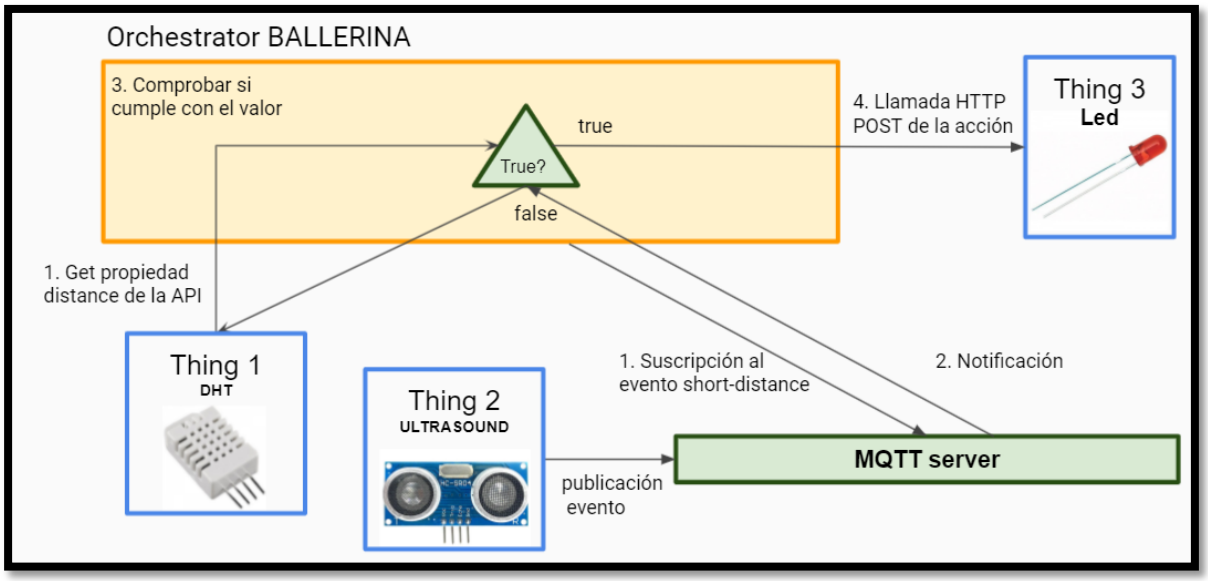


Figura 6.32. Ejemplo práctico GETs/MQTTs.

Para demostrar y validar este caso práctico en concreto, se ha grabado un video [38] explicativo dónde se puede comprobar el funcionamiento del Parser y del servicio Orquestador de Ballerina.

CAPÍTULO 7. CONCLUSIÓN Y TRABAJO FUTURO

7.1 Conclusión

Es cierto que la WoT es un hecho reciente y de corta historia, pero con una curva de alto desarrollo en estos momentos. El potencial de ésta, junto con el IoT, es enorme y su ámbito de aplicación crece día a día. No se puede estimar los muchos de los campos donde continúan creciendo y expandiéndose.

Al término de este proyecto, se ha logrado desarrollar un sistema para la integración de Things de la WoT. También se ha logrado realizar pruebas en entorno real utilizando Things reales tras haber implementado sus APIs correspondientes. Con todo este trabajo de investigación y construcción también se ha logrado un amplio conocimiento en Ballerina y en orquestación de microservicios, así como afianzar y potenciar en diversos lenguajes y herramientas tecnológicas como node.js, Express, Spring Boot y JAVA.

Se ha demostrado la capacidad del sistema en integrar Things de diversa manera. Primero a partir de la lectura de sus propiedades mediante llamadas GET a su API. Segundo, mediante la suscripción a eventos del tipo MQTT. En ambos casos o en combinación de ellos y siempre y cuando se cumpla la lógica deseada, se produce la llamada a una Thing consecuente, mediante un POST a su API.

El sistema de integración de este proyecto se ha creado a partir de código abierto, información y librerías, foros públicos y gratuitos. Aunque no es simple, y lleva un largo tiempo, es posible desarrollar numerosas mejoras que doten al producto de más valor.

7.2 Trabajo futuro

El trabajo que se puede hacer en durante el periodo del trabajo es limitado, y por ello la labor que queda por delante es enorme.

Una vez resuelto el principal objetivo del proyecto, de crear un sistema capaz de integrar un conjunto de Things de la WoT, se ha investigado en los diferentes caminos de ampliación.

El principal trabajo futuro consiste en dotar al sistema de la posibilidad de trabajar con más de 3 Things que actúen como antecedentes, de las cuales el sistema lee propiedades o se suscribe a sus eventos.

Además, actualmente el comportamiento del servicio Ballerina es cíclico ejecutándose de manera infinita hasta que el proceso se detiene. De esta manera comprueba una y otra vez el valor de los antecedentes aunque estos ya cumplan con el valor deseado. Es por eso por lo que se propone esta línea de mejora añadiendo la posibilidad de indicar el número de ejecuciones en el proceso.

Otra de las vías de ampliación se centra en añadir el protocolo AMQP como protocolo de eventos. De esta manera se incluiría al ya existente MQTT.

En relación con el microservicio Parser, se disponen como objetivos de trabajo futuro los siguientes:

- Mejorar la interfaz gráfica.
- Añadir la posibilidad de crear más de un orquestador y poder ejecutarlos simultáneamente.
- Integrar, cuando se ejecuta el servicio de Ballerina, la terminal en el navegador web.

BIBLIOGRAFÍA

[1] Lueth, K. L. IoT Analytics. (2020). Recuperado de: https://bit.ly/3dnMmDn

[2] Kovatsch, M., Matsukura, R., Lagally, M., Kawaguchi, T., Toumura, K., Kajimoto, K. (2020). *Web of Things (WoT) Architecture*, W3C Recommendation. Recuperado 20 de obtubre de 2020, de https://www.w3.org/TR/wot-architecture/

[3] Kaebisch, S., Kamiya, T., McCool, M., Charpenay, V., Kovatsch, M. (2020). *Web of* Things (WoT) Thing Description, W3C Recommendation. Recuperado 20 de obtubre de 2020, de https://www.w3.org/TR/wot-thing-description/

[4] Sheng, M., Qin, Y., Yao, L. y Benatallah, B. (2017). *Managing the Web of Things*. Cambridge: Morgan Kaufmann, Elsevier.

[5] *Web of Things (WoT)*. (s.f.). Recuperado 20 de octubre de 2020, de https://www.w3.org/WoT/

[6] Guinard, D. y Trifa, V. (2016). *Building the Web of Thing: whit examples in Node.js and Raspberry Pi.* Shelter Island: Manning Publications Co.

[7] Thangavel, D., Ma, X., Valera, A., Tan, HX. y Tan, CKY. *Performance evaluation of MQTT and COAP via a common middleware*. In: 2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP). 2014. p. 1–6.

[8] Fowler, M. y Lewis, J. (2014, Marzo 25). *Microservices: a definition of this new architectural term.* Recuperado de https://martinfowler.com/articles/microservices.html

[9] Singhal, N., Sakthivel, U. y Raj, P. (2019, Enero). *Selection Mechanism of micro-services orchestration vs choreography.* International Journal of Web & Semantic Technology (IJWest), volumen 10 No. 1.

[10] Spring Boot. (s.f.). Recuperado 10 de enero de 2021, de https://spring.io/projects/spring-boot.

[11] Spring Start. (s.f.). Recuperado 10 de enero de 2021, de https://start.spring.io.

[12] Ballerina. (s.f.). Recuperado 18 de enero de 2021, de https://ballerina.io.

[13] Fremantle, P. (2019). *MQTT-Ballerina*. Recuperado 2 de febrero de 2021, de https://bit.ly/3jvs8eZ.

[14] Visual Studio Code. (s.f.). Recuperado 18 de noviembre de 2020, de https://code.visualstudio.com/.

[15] Microsoft. (s.f.). Recuperado 18 de noviembre de 2020, de https://www.microsoft.com/es-es.

[16] Mead, A. (2018). *Learning Node.js Development*. Packt Publishing.

[17] Augarten, B., Kuo, M., Lin, E., Shaikh, A., Pereira, F., Tisserand, G., Zhang, C. y Zhang, K. (2015). *Express.js Blueprints*. Packt Publishing.

[18] Eclipse. (s.f.). Recuperado 18 de noviembre de 2020, de https://www.eclipse.org/.

[19] WSO2. (s.f.). Recuperado 20 de enero de 2020, de https://wso2.com/.

[20] Integration Studio. (s.f.). Recuperado 20 de enero de 2020, de https://bit.ly/3hhHe4T.

[21] Thymeleaf. (s.f.). Recuperado 19 de noviembre de 2020, de https://www.thymeleaf.org/.

[22] Soler Castaño, J.F. (2021). *WoTParser*. Recuperado 20 de junio de 2021, de https://github.com/soler5/TFM/tree/master/wotparser

[23] Electro Schematics HC-SR04 Datasheet. (s.f.). Recuperado 15 de mayo de 2021, de https://bit.ly/3hibKvz.

[24] Electro Schematics DHT22 Datasheet. (s.f.). Recuperado 15 de mayo de 2021, de https://bit.ly/3y7eKBx.

[25] Homemade Circuit Projects. Pir Sesnsor Datasheet. (s.f.). Recuperado 15 de mayo de 2021, de https://bit.ly/361ZolX.

[26] AreaTecnología. Diodo LED. (s.f.). Recuperado 15 de mayo de 2021, de https://bit.ly/3hm31bv.

[27] Raspberry Pi. (s.f.). Recuperado 10 de enero de 2021, de https://www.raspberrypi.org/.

[28] WoT book PI image. (s.f.). Recuperado 10 de enero de 2021, de https://bit.ly/3w4iHFF.

[29] Balena Etcher. (s.f.). Recuperado 10 de enero de 2021, de https://www.balena.io/etcher/.

[30] OnOff Library. (s.f.). Recuperado 11 de enero de 2021, de https://github.com/fivdi/onoff.

[31] NPM. Node-dht-sensor. (s.f.). Recuperado 11 de enero de 2021, de https://bit.ly/3w8Z7bv.

[32] Mosquitto. (s.f.). Recuperado 10 de febrero de 2021, de https://mosquitto.org/.

[33] Soler Castaño, J.F. (2021). *WoT*. Recuperado 20 de junio de 2021, de https://github.com/soler5/TFM/tree/master/wot

[34] Soler Castaño, J.F. (2021). *TFM - Soluciones de integración en la WoT. Una propiedad – GET.* Recuperado 20 de junio de 2021, de https://youtu.be/yj48Jk4JhHs.

[35] Soler Castaño, J.F. (2021). *TFM - Soluciones de integración en la WoT. Un evento – MQTT.* Recuperado 20 de junio de 2021, de https://youtu.be/a1ejEnMC2To.

[36] Soler Castaño, J.F. (2021). *TFM - Soluciones de integración en la WoT. Dos o más propiedades – GETs.* Recuperado 20 de junio de 2021, de https://youtu.be/4zpWPWuLOdQ.

[37] Soler Castaño, J.F. (2021). *TFM - Soluciones de integración en la WoT. Dos o más eventos – MQTTs.* Recuperado 20 de junio de 2021, https://youtu.be/dqMMmZsEXF4.

[38] Soler Castaño, J.F. (2021). TFM - Soluciones de integración en la WoT. Una propiedad – GET Propiedades/eventos – GETs/MQTTs. Recuperado 20 de junio de 2021, de https://youtu.be/Ph2aqCMNyCY

ANEXO 1. ARCHIVO DE CONFIGURACIÓN

Para generar el servicio de Ballerina, emplearemos un archivo de configuración en formato de texto (.txt) formado por 2 o 3 líneas.

En la primera línea se incluyen, separados por comas, los antecedentes de nuestra orquestación con el siguiente formato:

MQTT: mqtt {host} {topic} {condicion}

GET: get {host} {condicion} {time_ms}

En la segunda línea se incluye la condición o expresión que tienen que cumplir los antecedentes para ejecutar los consecuentes. (Sólo es necesaria si hay 2 o más antecedentes)

Ejemplos:

1 && 2

1 || 2

(1 && 2) || 3

En la tercera y última línea se incluyen, separados por comas, los consecuentes (POST) con el siguiente formato.

POST: **post {host} {payload}**

Se pueden dar las siguientes casuísticas:

- Un solo antecedente.
  - GET (a)
  - MQTT (b)
- Varios antecedentes.
  - 2 o más GET (c)
  - 2 o más MQTT (d)
  - 2 o más MQTT+GET (e)

Ejemplos del archivo de configuración según casuística:

(a) GET

```
get http://192.168.0.37:8485/pi/sensors/temperature value<25 1000

post http://192.168.0.37:8487/pi/actuators/leds/1 {"value":true}
```

(b) MQTT

```
mqtt 192.168.0.37 presence value==1

post http://192.168.0.37:8487/pi/actuators/leds/1 {"value":true}
```

(c) 2 o más GET

```
get http://192.168.0.37:8485/pi/sensors/temperature value<25 1000 , get http://192.168.0.37:8485/pi/sensors/humedad value<100 1000

1 || 2

post http://192.168.0.37:8487/pi/actuators/leds/1 {"value":true}
```

(d) 2 o más MQTT

```
mqtt 192.168.0.37 presence value==1 , mqtt 192.168.0.37 HC-SR04 value==1

1 && 2

post http://192.168.0.37:8487/pi/actuators/leds/1 {"value":true}
```

(e) 2 o más MQTT+GET

```
mqtt 192.168.0.37 presence value==1 , mqtt 192.168.0.37 HC-SR04 value==1 , get http://192.168.0.37:8485/pi/sensors/temperature value<25 1000

(1 || 3) && 2

post http://192.168.0.37:8487/pi/actuators/leds/1 {"value":true}
```

ANEXO 2. THINGS DESCRIPTIONS

SENSOR ULTRASONIDOS


```
{
    "title": "MyPirSensor",
    "id": "urn:dev:ops:32473-WoTPirSensor-1234",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "properties": {
        "distance": {
          "type": "number",
          "forms": [
            {
              "href": "http://192.168.0.37:8488/sensors/distance",
              "op": ["readproperty"],
              "contentType": "application/json"
            }
          ]
        }
      },
    "events": {
        "short-distance": {
            "data":{
              "type": "json",
              "description":"Devuelve un JSON con dos atributos: value y date.
 Value es un booleano que indica corta distancia y date contiene la fecha de l
a notificación"
            },
            "forms": [
                {
                    "href": "mqtt://192.168.0.37:1883",
                    "contentType": "text/plain",
                    "op": "subscribeevent"
                }
            ]
        }
    }
  }
```

SENSOR DHT


```
{
    "title": "MyDHTSensor",
    "id": "urn:dev:ops:32473-WoTDHTSensor-1234",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "properties": {
        "temperature": {
          "type": "number",
          "forms": [
            {
              "href": "http://192.168.0.37:8485/sensors/temperature",
              "op": ["readproperty"],
              "contentType": "application/json"
            }
          ]
        },
        "humidity": {
          "type": "number",
          "forms": [
            {
              "href": "http://192.168.0.37:8485/sensors/humidity",
              "op": ["readproperty"],
              "contentType": "application/json"
            }
          ]
        }
      }
  }
```

SENSOR PIR

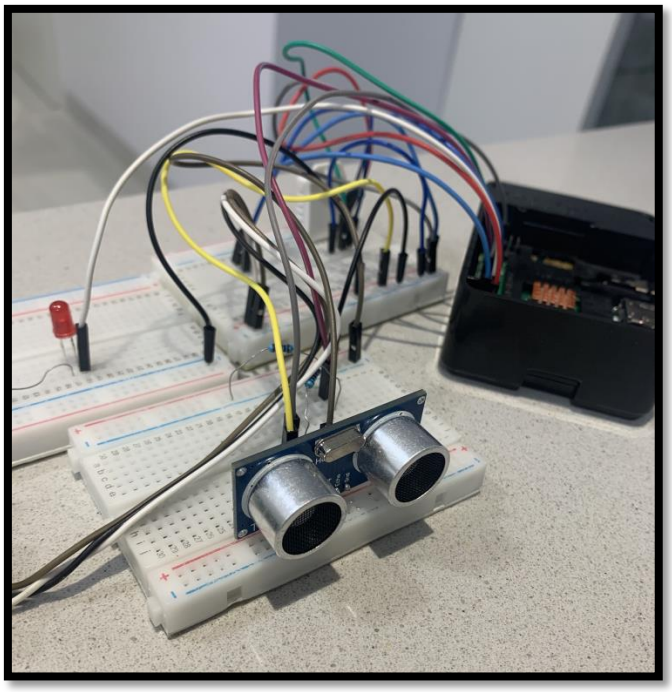
```
{
    "title": "MyPirSensor",
    "id": "urn:dev:ops:32473-WoTPirSensor-1234",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "properties": {
        "presence": {
          "type": "boolean",
          "forms": [
            {
              "href": "http://192.168.0.37:8486/sensors/pir",
              "op": ["readproperty"],
              "contentType": "application/json"
            }
          ]
        }
      },
    "events": {
        "presence": {
            "data":{
                  "type": "integer",
                 "description":"Devuelve un JSON con dos atributos: value y
date. Value es un booleano que indica presencia y date contiene la fecha de la
 notificación"

            },
            "forms": [
                {
                    "href": "mqtt://192.168.0.37:1883",
                    "contentType": "text/plain",
                    "op": "subscribeevent"
                }
            ]
        }
    }
  }
```

LED

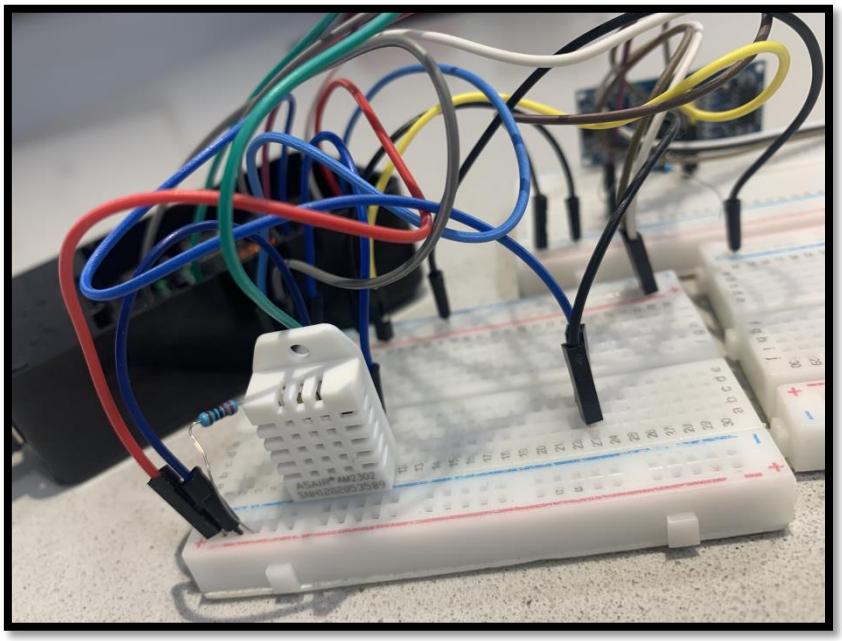
```
{
    "title": "MyPirSensor",
    "id": "urn:dev:ops:32473-WoTPirSensor-1234",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "properties": {
      "status": {
        "type": "boolean",
        "forms": [
          {
            "href": "http://192.168.0.37:8487/actuators/led",
            "op": ["readproperty"],
            "contentType": "application/json"
          }
        ]
      }
    },
    "actions": {
      "encender" : {
        "input":  {
          "type": "json",
          "properties": {
            "value":true
          }
        },
        "forms": [{
          "op": ["invokeaction"],
          "href": "http://192.168.0.37:8487/actuators/led",
          "contentType":"application/json",
          "http:methodName": "PUT"
        }]
      },
      "apagar": {
        "input": {
          "type": "json",
          "properties": {
            "value":false
          }
        },
        "forms": [{
          "op": ["invokeaction"],
          "href": "http://192.168.0.37:8487/actuators/led",
          "contentType":"application/json",
          "http:methodName": "PUT"
        }]
    }
  }}
```

ANEXO 3. IMÁGENES CASO REAL

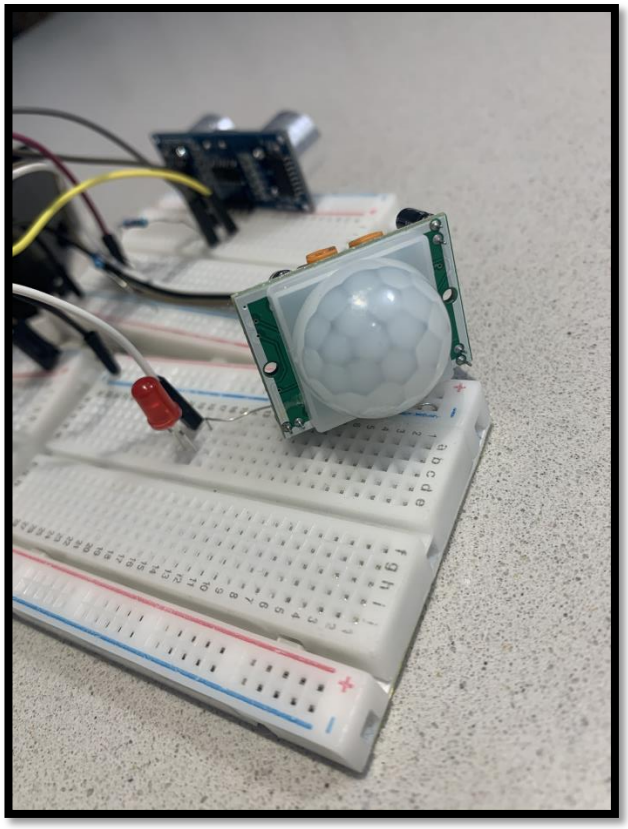
SENSOR ULTRASONIDOS



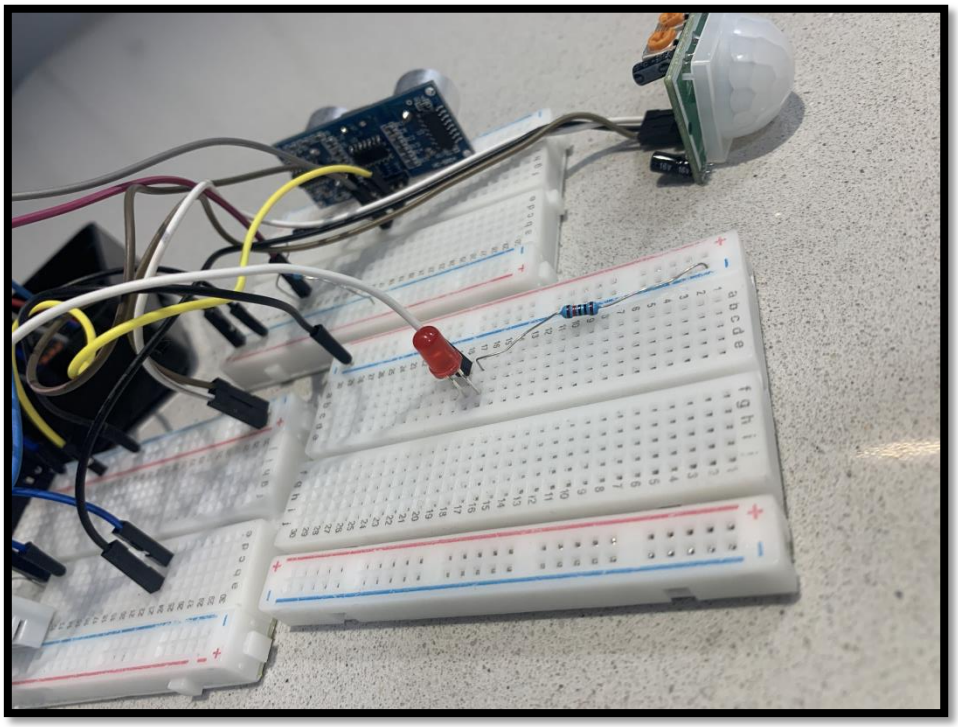
SENSOR DHT



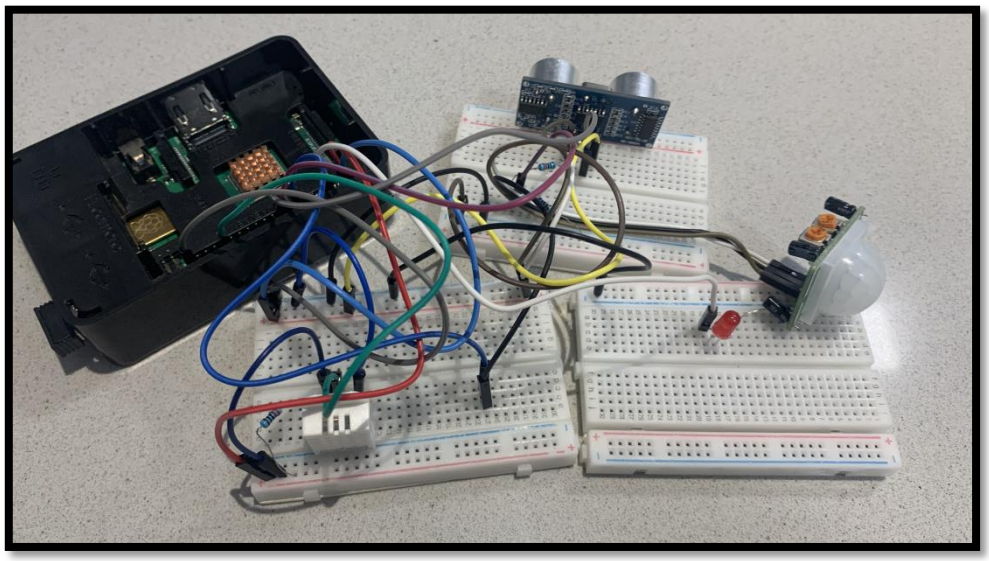
SENSOR PIR



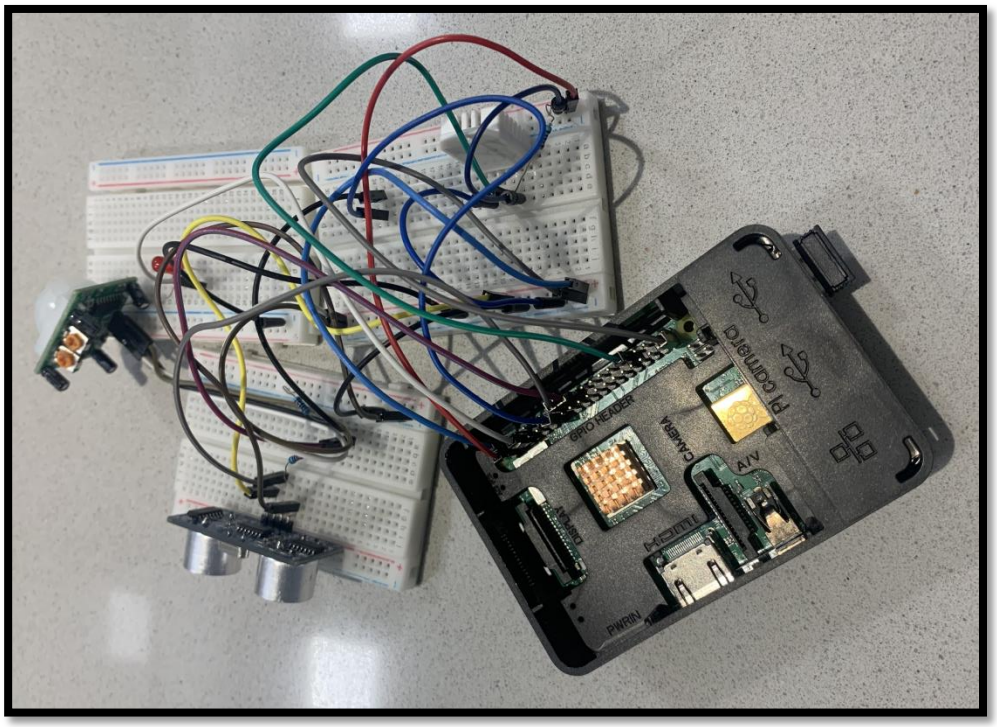
LED



VISIÓN GLOBAL 1



VISIÓN GLOBAL 2



Resumen/Abstract

Este proyecto se centra en el desarrollo de un sistema capaz de integrar dispositivos de la WoT haciendo uso de Ballerina, probando el sistema en un entorno real haciendo uso de una Raspberry Pi y de diversos sensores y actuadores. Ballerina es un lenguaje de programación y framework llamado a ser referencia debido a que facilita la conexión entre aplicaciones y servicios en todo tipo de escenarios de integración. El trabajo desarrollado cubre desde el estudio de la WoT y las diversas formas de integrar Things en ella, hasta la programación y evaluación del producto generando diferentes escenarios reales.

En primer lugar, se ha realizado un estudio bibliográfico de la WoT, la coreografía y orquestación de microservicios, Ballerina y su ámbito de aplicación en la integración, para llegado el siguiente capítulo realizar la selección de la arquitectura del sistema.

Tras la elección de la arquitectura se realiza una implementación individual de cada caso propuesto en función de la manera en la que interactuemos con las Things: propiedades, eventos y acciones.

Finalmente se ha desarrollado el microservicio Parser, encargado de generar los microservicios de integración a partir de un archivo de configuración donde se indican en primer lugar, las Things de las que leemos propiedades o nos suscribimos a sus eventos, en segundo la lógica que debe seguir el sistema y, por último, las Things consecuentes sobre las que se actúa en caso de cumplirse la expresión lógica deseada.

Para la validación del sistema desarrollado se han implementado diversas Things. Para ello se han creado APIs sobre la Raspberry Pi para un sensor ultrasónico, un sensor de temperatura y humedad, un sensor PIR y un LED.