iasturias INNOVACIÓN
Gobierno del Principado de Asturias
Consejería de Empleo, Industria y Turismo
Unión Europea
Fondo Europeo de Desarrollo Regional

RESULTADOS WORMS

WEB OF THINGS REFERENCE-IMPLEMENTATION FOR A MICROSERVICES-BASED SERVIENT

En Gijón , a 31 de Diciembre de 2020

1. MEMORIA TÉCNICA

Introducción

El proyecto WoRMS (*Web of Things Reference-implementation for a Microservices-based Servient*) es ejecutado por Fundación CTIC – Centro Tecnológico entre el 1 de enero de 2018 y el 31 de diciembre de 2020.

El proyecto WoRMS se concibió con el objetivo principal de investigar un conjunto de tecnologías, componentes y herramientas software *open source* con el fin de diseñar una implementación de referencia que facilite a la industria el desarrollo de sistemas y aplicaciones compatibles con los estándares WoT definidos por el W3C *Web of Things Working Group*.

En el ámbito del proyecto se definirán y desarrollarán dos versiones de la implementación de referencia con el fin de abarcar el mayor número de dispositivos IoT posibles: Una versión completa destinada a dispositivos y sistemas con capacidades de computación elevadas (WoRMS) y otra versión restringida orientada a dispositivos y sistemas con capacidades de computación muy reducidas (WoRMS Lite).

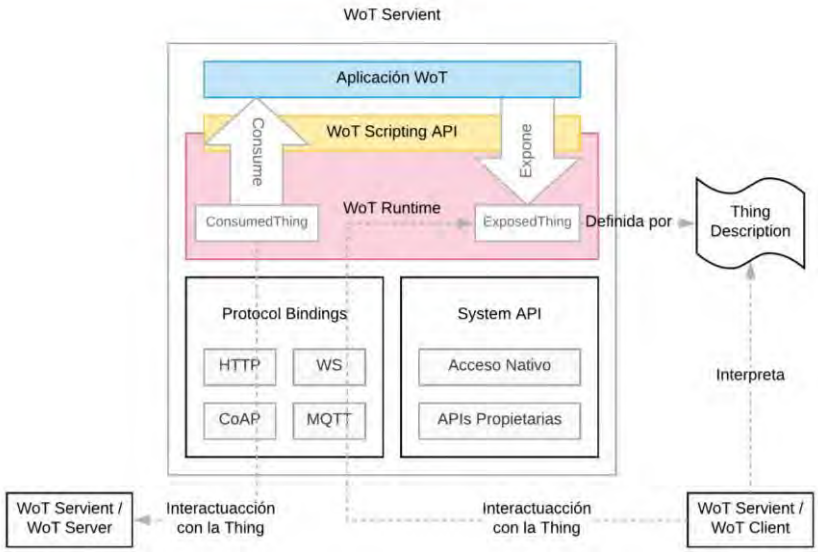


Figura 1: Diagrama de alto nivel de un WoT Servient.

El plan de trabajo del proyecto WoRMS se ha dividido en 3 fases a lo largo de las cuales se desarrollan 4 hitos o paquetes de trabajo:

- Fase 1: Diseño.
  - Hito 1: Análisis de dispositivos IoT y diseño de bajo nivel de la implementación de referencia.
- Fase 2: Desarrollo
  - Hito 2: Diseño y desarrollo de la implementación de referencia WoRMS.
  - Hito 3: Diseño y desarrollo de la implementación de referencia WoRMS Lite.
- Fase 3: Validación.

- Hito 4: Despliegue de la aplicación WoT demostradora y validación de la implementación de referencia.

Justificación 2018

Durante el periodo que cubre este informe, se ha finalizado por completo la ejecución de las tareas correspondientes al Hito 1, y se ha comenzado a trabajar en el Hito 2.

A continuación, se describen las tareas realizadas en cada uno de los hitos de trabajo mencionados anteriormente y los resultados conseguidos.

Hito 1: PT1: Análisis de dispositivos IoT y diseño de bajo nivel de la implementación de referencia.

El objetivo principal de este hito fue tanto la especificación de requisitos del sistema como el diseño técnico del mismo, desglosando la actividad en dos tareas principales:

- T1.1: Investigación del ecosistema de dispositivos IoT de baja capacidad de procesamiento.
- T1.2: Diseño en bajo nivel de la implementación de referencia WoRMS Lite.
- T1.3: Diseño en bajo nivel de la implementación de referencia WoRMS Lite.
- T1.4: Diseño general de la aplicación WoT demostradora.

T1.1: Investigación del ecosistema de dispositivos IoT de baja capacidad de procesamiento.

En esta tarea se ha realizado un análisis de los dispositivos hardware que serán objetivo de la implementación "Lite" del WoT Runtime. Estos dispositivos se caracterizan por ser microcontroladores de bajas capacidades (CPU, memoria RAM y memoria interna), pero con una serie de protocolos de conexión de periféricos a través de I2C, UART, GPIO… El coste reducido y la versatilidad de los mismos permiten su despliegue en aplicaciones IoT.

Un aspecto a tener en cuenta a la hora de seleccionar los dispositivos objetivo, es que estos deben tener algún tipo de conexión a internet (Ethernet o Wifi son las más comunes), ya que la propia implementación requiere del uso de servicios y aplicaciones web, que de otra manera serían inaccesibles.

A continuación, se resumen las características de los dispositivos que se han analizado:

Arduino MKR1000[1]

[1] https://store.arduino.cc/arduino-mkr1000

Figura 2: Arduino MKR1000

Arduino MKR1000 es una potente placa que combina la funcionalidad del Zero y el Wi-Fi Shield. Es la solución ideal para los fabricantes que desean diseñar proyectos de IoT con una experiencia previa mínima en redes. Arduino MKR1000 ha sido diseñado para ofrecer una solución práctica y rentable para los fabricantes que buscan agregar conectividad Wi-Fi a sus proyectos con una experiencia previa mínima en redes. Está basado en el Atmel ATSAMW25 SoC (System on Chip), que forma parte de La familia SmartConnect de dispositivos inalámbricos Atmel, diseñada específicamente para proyectos y dispositivos IoT. El ATSAMW25 está compuesto por tres bloques principales:

- SAMD21 Cortex-M0 + MCU de bajo consumo de 32 bits
- WINC1500 de baja potencia de 2.4GHz IEEE® 802.11 b / g / n Wi-Fi
- ECC508 CryptoAuthentication

El ATSAMW25 incluye también una sola antena de PCB de flujo 1x1. El diseño incluye un circuito de carga Li-Po que permite que el Arduino / Genuino MKR1000 funcione con batería o con 5 V externos, cargando la batería Li-Po mientras funciona con energía externa. El cambio de una fuente a otra se realiza de forma automática. Una buena potencia de cómputo de 32 bits similar a la placa Zero, el amplio conjunto habitual de interfaces de E / S, Wi-Fi de baja potencia con un Cryptochip para una comunicación segura y la facilidad de uso del software Arduino (IDE) para desarrollo y programación de código.

Todas estas características hacen que esta placa sea la opción preferida para los proyectos emergentes de baterías IoT en un factor de forma compacta. El puerto USB se puede usar para suministrar alimentación (5 V) a la placa. El Arduino MKR1000 puede funcionar con o sin la Batería Li-Po conectada y tiene un consumo de energía limitado. El módulo Wifi MKR1000 es compatible con el certificado SHA-256.

Las principales características técnicas de este dispositivo son:

- Basado en el microcontrolador: SAMD21.
- Arquitectura: 32 bits Cortex M0+.
- Interfaz inalámbrica: WiFi.
- Voltaje de alimentación: 5 volts, USB, batería LiPo.
- Entradas / salidas digitales: 8.
- Salidas PWM: 12.
- Entradas analógicas: 7.
- Salidas analógicas: 1.
- Corriente máxima en pin de 3.3 volts: Ninguna.

- Corriente máxima en pines de IO: 7 mA.
- Memoria flash para programa: 256 KB.
- Memoria RAM: 32 KB.
- Memoria EEPROM: Ninguna.
- Frecuencia de reloj: 48 Mhz.
- Led multipropósito: pin 6.

RTL8710[2]

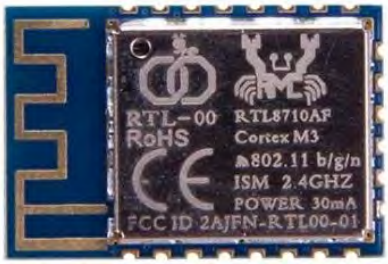


Figura 3: RTL8710

El módulo RTL8710 Wifi es una plataforma de IoT inalámbrica de bajo coste basada en el procesador Realtek RTL8710 ARM Cortex-M3 a 166MHz. A pesar de su tamaño, se postula como una alternativa interesante por su potente MCU, 1MB ROM, 512KB RAM, 1MB Flash incorporado y un rendimiento más rápido de Wifi.

El módulo RTL8710 Wifi usa FreeRTOS como sistema operativo, y puede ser depurado y programado utilizando el SDK del RealTek RTL8710 sobre una conexión micro USB. Contiene hasta 17 GPIO, hasta 3 I2C, hasta 2 PCM, 4 PWM y dos UART de alta velocidad y uno de baja velocidad. Para Wi-Fi, tiene una velocidad de hasta 150 Mbps en una conexión 802.11n y una velocidad máxima de 54 Mbps en 802.11g.

El módulo RTL8710 Wifi ofrece un procesador económico y potente, lo que le convierte en una gran opción para plataformas IoT WIFI para fabricantes, aficionados, ingenieros de hardware e ingenieros de software.

Principales características:

- 802.11 b / g / n, CMOS MAC.
- CPU incorporada de 32 bits.
- Protocolo TCP / IP incorporado.
- WiFi a 2.4 GHz, soporta WPA / WPA2.
- Soporte STA / AP / STA + AP.
- Soporta configuración inteligente (incluyendo dispositivos Android e iOS).

[2] https://www.seeedstudio.com/RTL8710-WiFi-Module-p-2793.html

- SPI, UART, I2C, I2S, GPIO.
- Potencia de salida 802.11b + 17 dBm.
- Parámetros inalámbricos.
  - Estándar inalámbrico: 802.11 b / g / n.
  - Frecuencia: 2.4GHz - 2.5GHz (2400M - 2483.5M).
- Parámetros de hardware:
  - CPU: ARM Cortex M3 (166MHz).
  - ROM / RAM / Flash: 1MB / 512KB / 1MB.
  - SPI: Hasta 2.
  - I2C: 3.
  - GPIO: 17.
  - Tensión de trabajo: 3.0 - 3.6 (3.3V).
  - Temperatura de trabajo: -20 a 85.
  - Tamaño: 24mm * 16mm * 0.8mm.
  - Modelo de red inalámbrica: Station/SoftAP/SoftAP+Station.
- Parámetros de software:
  - Wifi: 802.11n hasta 150 Mbps, 802.11g hasta 54 Mbps.
  - Protocolod de red: TCP / UDP / HTTP / FTP.
  - Seguridad: WPA / WPA2.
  - Actualización de firmware: Puerto serial local / OTA / Descarga de host.
  - Configuración de usuario: Conjunto de instrucciones AT +, servidor en la nube.

ESP8266[3]

Figura 4: ESP8266

ESP8266EX es capaz de funcionar de manera consistente en entornos industriales, debido a su amplio rango de temperatura de operación. Con características altamente integradas en el chip y una cantidad mínima de componentes externos discretos, el chip ofrece confiabilidad, compacidad y robustez.

Diseñado para dispositivos móviles, dispositivos electrónicos portátiles y aplicaciones de IoT, ESP8266EX logra un bajo consumo de energía con una combinación de varias tecnologías patentadas. La arquitectura

[3] https://www.espressif.com/en/products/hardware/esp8266ex/overview

de ahorro de energía presenta tres modos de operación: modo activo, modo de suspensión y modo de suspensión profunda. Esto permite que los diseños alimentados por batería funcionen por más tiempo.

El ESP8266EX de Espressif ofrece una solución Wi-Fi SoC altamente integrada para satisfacer las continuas demandas de los usuarios de un uso eficiente de la energía, un diseño compacto y un rendimiento fiable en la industria del IoT.

Con las capacidades de red Wi-Fi completas y autónomas, el ESP8266EX puede funcionar como una aplicación independiente o como esclavo de un host MCU. Cuando el ESP8266EX aloja la aplicación, se inicia rápidamente desde la memoria flash. La caché de alta velocidad integrada ayuda a aumentar el rendimiento del sistema y a optimizar la memoria del sistema. Además, el ESP8266EX puede aplicarse a cualquier diseño de microcontrolador como adaptador Wi-Fi a través de interfaces SPI/SDIO o UART.

El ESP8266EX integra interruptores de antena, balun RF, amplificador de potencia, amplificador de recepción de bajo ruido, filtros y módulos de gestión de potencia. El diseño compacto minimiza el tamaño de la PCB y requiere circuitos externos mínimos.

Además de las funcionalidades Wi-Fi, el ESP8266EX también integra una versión mejorada del procesador de 32 bits de la serie L106 Diamond de Tensilica y de la SRAM en el chip. Se puede interconectar con sensores externos y otros dispositivos a través de los GPIOs. Su SDK proporciona códigos de ejemplo para varias aplicaciones.

La Plataforma de Conectividad Inteligente de Espressif Systems (ESCP) permite características sofisticadas, incluyendo:

- Cambio rápido entre el modo de suspensión y el modo de despertador para un uso eficiente de la energía.
- Sesgo de radio adaptativo para un funcionamiento de baja potencia.
- Procesamiento avanzado de la señal.
- Mecanismos de cancelación de estimulación y coexistencia de RF para la mitigación de interferencias comunes de celulares, Bluetooth, DDR, LVDS y LCD.

Las principales especificaciones técnicas del dispositivo son:

- Wi-Fi:
  - Certificación: Alianza Wi-Fi.
  - Protocolos: 802.11 b/g/n (HT20).
  - Rango de frecuencia: 2.4G ~ 2.5G (2400M ~ 2483.5M).
  - Potencia TX:
    - 802.11 b: +20 dBm.
    - 802.11 g: +17 dBm.
    - 802.11 n: +14 dBm.
  - Sensibilidad de Rx:
    - 802.11 b: -91 dbm (11 Mbps).
    - 802.11 g: -75 dbm (54 Mbps).

        - 802.11 n: -72 dbm (MCS7).
    - Antena: PCB Trace, Externo, Conector IPEX, Chip de cerámica.
- Hardware:
    - CPU: Procesador de 32 bits Tensilica L106.
    - Interfaz de periféricos:
        - Control remoto UART/SDIO/SPI/I2C/I2S/IR.
        - Luz y botón GPIO/ADC/PWM/LED.
    - Voltaje de operación: 2.5V ~ 3.6V.
    - Corriente de funcionamiento: Valor medio: 80 mA.
    - Rango de temperatura de funcionamiento: -40°C ~ 125°C.
    - Tamaño del envase: QFN32-pin (5 mm x 5 mm).
    - Interfaz externa: -.
- Software:
    - Modo Wi-Fi: Estación/SoftAP/SoftAP+Estación.
    - Seguridad: WPA/WPA2.
    - Encriptación: WEP/TKIP/AES.
    - Actualización de firmware: Descarga UART / OTA (a través de la red).
    - Desarrollo de Software: Soporta Desarrollo de Servidor de Nube / Firmware y SDK para una rápida programación en el chip.
    - Protocolos de red: IPv4, TCP/UDP/HTTP.
    - Configuración de usuarios: Conjunto de instrucciones AT, Servidor en nube, Aplicación Android/iOS.

ESP32[4]

Figura 5: ESP32.

El ESP32 es un único chip combinado Wi-Fi y Bluetooth de 2,4 GHz diseñado con la tecnología TSMC de ultra baja potencia de 40 nm. Está diseñado para lograr la mejor potencia y rendimiento de RF, mostrando

[4] https://www.espressif.com/en/products/hardware/esp32/overview

robustez, versatilidad y fiabilidad en una amplia variedad de aplicaciones y escenarios de potencia. La serie de chips ESP32 incluye ESP32-D0WDQ6, ESP32-D0WD, ESP32-D2WD y ESP32-S0WD.

El ESP32 está diseñado para aplicaciones móviles, electrónicas y de Internet de las cosas (IoT). Cuenta con todas las características de última generación de los chips de baja potencia, incluyendo la compuerta de reloj de grano fino, múltiples modos de potencia y el escalado dinámico de potencia. Por ejemplo, en un escenario de aplicación de concentrador de sensor de IoT de baja potencia, el ESP32 se despierta periódicamente y sólo cuando se detecta una condición específica. El ciclo de trabajo bajo se utiliza para minimizar la cantidad de energía que gasta el chip. La salida del amplificador de potencia también es ajustable, contribuyendo así a un equilibrio óptimo entre el rango de comunicación, la velocidad de datos y el consumo de energía.

El ESP32 es una solución altamente integrada para aplicaciones de IO de Wi-Fi y Bluetooth, con unos 20 componentes externos. El ESP32 integra un conmutador de antena, un balun de RF, un amplificador de potencia, un amplificador de recepción de bajo ruido, filtros y módulos de gestión de potencia. Como tal, la solución completa ocupa un área mínima de placa de circuito impreso (PCB).

El ESP32 utiliza CMOS para radio y banda base totalmente integradas de un solo chip, a la vez que integra circuitos de calibración avanzados que permiten que la solución elimine las imperfecciones de los circuitos externos o se ajuste a los cambios en las condiciones externas. Como tal, la producción en masa de soluciones ESP32 no requiere equipos de prueba Wi-Fi caros y especializados.

Las principales especificaciones técnicas del dispositivo son:

- Wi-Fi:
  - 802.11 b/g/n.
  - 802.11 n (2.4 GHz), hasta 150 Mbps.
  - WMM.
  - TX/RX A-MPDU, RX A-MSDU.
  - Bloqueo inmediato ACK.
  - Desfragmentación.
  - Supervisión automática de balizas (hardware TSF).
  - 4 × interfaces Wi-Fi virtuales.
  - Soporte simultáneo para los modos Infrastructure Station, SoftAP y Promiscuous. Tenga en cuenta que cuando el ESP32 está en modo Estación, al realizar una búsqueda, el canal SoftAP cambiará.
  - Diversidad de antenas.
- Bluetooth:
  - Cumple con las especificaciones Bluetooth v4.2 BR/EDR y BLE.
  - Transmisor clase 1, clase 2 y clase 3 sin amplificador de potencia externo.
  - Control de potencia mejorado.
  - 12 dBm potencia de transmission.
  - Receptor NZIF con sensibilidad BLE de -97 dBm.
  - Salto de frecuencia adaptativa (AFH).

  - HCI estándar basado en SDIO/SPI/UART.
  - UART HCI de alta velocidad, hasta 4 Mbps.
  - Bluetooth 4.2 BR/EDR BLE controlador de modo dual.
  - Conexión síncrona orientada/extendida (SCO/eSCO).
  - CVSD y SBC para códecs de audio.
  - Bluetooth Piconet y Scatternet.
  - Multi-conexiones en BT y BLE clásico.
  - Publicidad y escaneado simultáneos.
- CPU y memoria:
  - Microprocesador(es) Xtensa® de 32 bits LX6 de uno o dos núcleos, hasta 600 MIPS (200 MIPS para ESP32-S0WD, 400 MIPS para ESP32-D2WD).
  - 448 KB ROM.
  - 520 KB SRAM.
  - 16 KB SRAM en RTC.
  - QSPI soporta múltiples chips flash/SRAM.
- Relojes y timers:
  - Oscilador interno de 8 MHz con calibración.
  - Oscilador RC interno con calibración.
  - Oscilador de cristal externo de 2 MHz ~ 60 MHz (40 MHz sólo para funcionalidad Wi-Fi/BT).
  - Oscilador de cristal externo de 32 kHz para RTC con calibración.
  - Dos grupos de temporizadores, incluyendo 2 × 64-bit timers y 1 × watchdog principal en cada grupo.
  - Un temporizador RTC.
  - RTC watchdog.
- Interfaces avanzadas para periféricos:
  - 34 GPIOs programables.
  - SAR ADC de 12 bits de hasta 18 canales.
  - 2 × DAC de 8 bits.
  - 10 × sensores táctiles.
  - 4 × SPI.
  - 2 × I^2S.
  - 2 × I^2C.
  - 3 × UART.
  - 1 host (SD/eMMC/SDIO).
  - 1 esclavo (SDIO/SPI).
  - Interfaz Ethernet MAC con soporte dedicado para DMA e IEEE 1588.
  - CAN 2.0.
  - IR (TX/RX).
  - Motor PWM.

- LED PWM hasta 16 canales.
- Sensor Hall.

- Seguridad:
  - Arranque seguro.
  - Encriptación Flash.
  - OTP de 1024 bits, hasta 768 bits para clientes.
  - Aceleración de hardware criptográfico:
    - AES.
    - Hachís (SHA-2).
    - RSA.
    - CCE.
    - Generador de números aleatorios (RNG).

Pycom[5]

En esta familia se han analizado los dispositivos WiPy, SiPy, LoPy4, GPy y FiPy.

Si bien cada dispositivo tiene sus particularidades, que se indicarán a continuación, todos ellos poseen las siguientes características y especificaciones comunes:

Características:

- Potente CPU, BLE y radio WiFi de última generación. Rango WiFi 1KM.
- Habilitado para MicroPython.
- Cabe en una protoboard estándar (con cabezales).
- Consumo de energía ultra bajo: una fracción en comparación con otros microcontroladores conectados.
- Disponible con o sin cabezales de pines soldados.

Especificaciones:

- CPU:
  - Xtensa® dual-core 32-bit LX6 microprocessor(es), hasta 600 DMIPS.
  - Aceleración de hardware en coma flotante.
  - Multihilo Python.
  - Un coprocesador ULP extra que puede monitorizar GPIOs, los canales ADC y controlar la mayoría de los periféricos internos durante el modo de espera profunda, mientras que sólo consume ~25uA.
- WiFi: 802.11b/g/n 16mbps.
- Bluetooth: Bajo consumo de energía y clásico.

[5] https://pycom.io/

- RTC: Funcionando a 150kHz.
- Seguridad:
  - Compatibilidad con SSL/TLS.
  - Seguridad empresarial WPA.
- Hash / encriptación:
  - CSA.
  - MD5.
  - DES.
  - AES.

WiPy.

Figura 6: Pycom WiPy

El WiPy 3.0 es una pequeña plataforma de desarrollo MicroPython permite el uso de Wifi y Bluetooth IoT. Con un rango WiFi de 1KM, un chipset Espressif ESP32 de última generación y un procesador dual, el WiPy 3.0 se trata de llevar el Internet de las cosas al siguiente nivel.

Especificaciones diferenciales:

- RAM: 520KB + 4MB.
- Memoria Flash: 8MB.

SiPy.

Figura 7: Pycom SiPy

Con Sigfox, Wifi y BLE, el SiPy es el único microcontrolador con triple portador habilitado para MicroPython en el mercado hoy en día. Es una plataforma IoT de grado empresarial perfecta para sus objetos conectados. Con el último chipset Espressif, el SiPy ofrece una combinación perfecta de potencia, facilidad de uso y flexibilidad.

Características diferenciales:

- Dos años de conectividad Sigfox gratuita.

Especificaciones diferenciales:

- RAM: 520KB.
- Memoria Flash: 4MB.

LoPy4.

Figura 8: Pycom LoPy4

LoPy4 es una tarjeta de desarrollo cuádruple con capacidad para MicroPython (LoRa, Sigfox, WiFi, Bluetooth), la plataforma perfecta de nivel empresarial para tus objetos conectados. Con el último chipset Espressif, el LoPy4 ofrece una combinación perfecta de potencia, facilidad de uso y flexibilidad.

Características diferenciales:

- Conectividad LoRa y Sigfox simultánea.
- También se puede utilizar como una puerta de enlace Nano LoRa.

Especificaciones diferenciales:

- RAM: 520KB + 4MB.
- Memoria Flash: 4MB.

GPy.

Figura 9: Pycom GPy

Con WiFi, BLE y LTE-CAT M1/NB1, el GPy es el último microcontrolador con triple portador Pycom habilitado para MicroPython en el mercado hoy en día, presentándose como la plataforma perfecta de IoT de calidad empresarial para sus objetos conectados.

Características diferenciales:

- También se puede utilizar como una puerta de enlace Nano LoRa.
- Listo para todo el mundo, un solo producto cubre todas las bandas LTE-M.

Especificaciones diferenciales:

- RAM: 520KB + 4MB.
- Memoria Flash: 8MB.
- LTE CAT-M1/MB-IoT:
  - Un solo chip para CAT M1 y NB1.
  - 3GPP versión 13 LTE Advanced Pro.
  - Soporta las categorías de UE LTE de banda estrecha M1/NB1.
  - Banda base integrada, RF, memoria RAM y gestión de energía.
  - Opción de clase de potencia de transmisión reducida.
  - Estimaciones de potencia máxima:
    - Corriente TX = 420mA pico @1.5Watt.
    - Corriente RX = 330mA pico @1.2Watt.
  - Funciones DRX (eDRX) y PSM ampliadas para casos de uso de larga duración de suspensión.

FiPy.

Figura 10: Pycom FiPy

Con Sigfox, LoRa, WiFi, BLE y LTE-CAT M1/ NB1, FiPy es el último microcontrolador compatible con Pycom MicroPython del mercado actual, siendo la plataforma IOT perfecta para tus objetos conectados.

Características diferenciales:

- Cinco redes: WiFi, BLE, celular LTE-CAT M1/NB1, LoRa y Sigfox.
- También se puede utilizar como una puerta de enlace Nano LoRa.
- Listo para todo el mundo, un solo producto cubre todas las bandas LTE-M.

Especificaciones diferenciales:

- RAM: 520KB + 4MB.
- Memoria Flash: 8MB.
- LoRa:
  - LoRaWAN stack - Dispositivos de clase A y C.
  - Rango de nodos: Hasta 40 km.
  - Nano-gateway: Hasta 22km (Capacidad hasta 100 nodos).
- Sigfox:
  - Dispositivo de clase 0. Potencia máxima de transmisión:
    - +14dBm(Europa).
    - +22dBm (América).
    - +22dBm (Australia y Nueva Zelanda).
  - Rango de nodos: Hasta 50 km.
- LTE CAT-M1/MB-IoT:
  - Un solo chip para CAT M1 y NB1.
  - 3GPP versión 13 LTE Advanced Pro.
  - Soporta las categorías de UE LTE de banda estrecha M1/NB1.
  - Banda base integrada, RF, memoria RAM y gestión de energía.
  - Opción de clase de potencia de transmisión reducida.
  - Estimaciones de potencia máxima:
    - Corriente TX = 420mA pico @1.5Watt.
    - Corriente RX = 330mA pico @1.2Watt.
  - Funciones DRX (eDRX) y PSM ampliadas para casos de uso de larga duración de suspensión.

Pyboard[6]

[6] https://store.micropython.org/

Figura 11: Pyboard.

El MicroPython pyboard es una placa de circuito electrónico compacto que ejecuta MicroPython en el *bare metal*, lo que le brinda un sistema operativo Python de bajo nivel que se puede usar para controlar todo tipo de proyectos electrónicos.

El pyboard es la placa oficial de microcontroladores MicroPython con soporte completo para funciones del software MicroPython.

Sus principales características hardware son:

- Microcontrolador STM32F405RG.
- CPU Cortex M4 a 168 MHz con hardware de punto flotante.
- ROM de 1024KiB y RAM de 192KiB.
- Conector micro USB para alimentación y comunicación serie.
- Ranura para tarjeta micro SD, compatible con tarjetas SD estándar y de alta capacidad.
- Acelerómetro de 3 ejes (MMA7660).
- Reloj en tiempo real con batería de respaldo opcional.
- 24 GPIO en los bordes izquierdo y derecho y 5 GPIO en la fila inferior, más LED e interruptor GPIO disponibles en la fila inferior.
- Convertidores analógicos a digitales de 3x 12 bits, disponibles en 16 pines, 4 con blindaje de tierra analógico.
- Convertidores 2x 12 bits de digital a analógico (DAC), disponibles en los pines X5 y X6.
- 4 LEDs (rojo, verde, amarillo y azul).
- 1 interruptor de reinicio y otro de usuario.
- Regulador de voltaje LDO incorporado de 3.3V, capaz de suministrar hasta 250 mA, rango de voltaje de entrada de 3.6V a 16V.
- Cargador de arranque DFU en la ROM para actualizar fácilmente el firmware.

La siguiente tabla resume las principales características de estos dispositivos:

Tabla 1: Resumen características dispositivos WoT de baja capacidad de procesamiento.

Dispositivo	Descripción	Entorno de desarrollo	Precio
Arduino MKR1000	Modelo MKR de Arduino con conexión WiFi integrada gracias al chip WINC1500. CPU: 26MHz, ROM: 4MB o 8MB, RAM: 224KB	Arduino	31€
RTL8710	Chip WiFi con procesador ARM Cortex M3 (166MHz), ROM: 1MB, RAM: 512K.	Arduino	<5€

ESP8266	CPU: Tensilica L106 32-bit 80MHz, ROM: 512KB hasta 16MB, RAM: 80K.	Arduino/MicroPython	5€
ESP32	CPU: Tensilica Xtensa LX6 160 o 240 MHz, ROM: 512KB hasta 16MB, RAM: 520KB.	Arduino/MicroPython	<10€
Pycom	Existen múltiples versiones (WiPy, GPy, FiPy), pero todas se basan en procesadores ESP32 modificados con mayor capacidad y funcionalidades específicas.	MicroPython	Entre 20 € y 54 €
Pyboard	CPU: 168 MHz Cortex M4, ROM: 1024KiB, RAM: 192KiB. Necesita de un complemento para tener conexión WiFi.	MicroPython	~60€

En base a los dispositivos previamente analizados, y teniendo en cuenta tanto capacidades como entornos de desarrollo, se ha optado por realizar la implementación utilizando la familia de los ESP32, los cuales incluyen también los microcontroladores Pycom. En concreto se utilizarán las placas: Node MCU, GPy y FiPy.

En todos estos dispositivos se puede desarrollar utilizando MicroPython, lo que unifica y simplifica las tareas de programación. De acuerdo a la definición del sitio oficial[7], MicroPython es un pequeño pero eficiente interprete del Lenguaje de Programación Python 3 que incluye un subconjunto mínimo de librerías y que además está optimizado para que pueda correr en microcontroladores y ambientes restringidos.

MicroPython ofrece la posibilidad de escribir códigos más simples, en lugar de usar Lenguajes de Programación de más bajo nivel como C o C++, que es el que utiliza Arduino, por ejemplo. MicroPython está repleto de características avanzadas como un prompt interactivo, números enteros de precisión arbitraria, cierres, comprensión de listas, generadores, manejo de excepciones y más. Sin embargo, es lo suficientemente compacto para caber y funcionar con sólo 256k de espacio de código y 16k de RAM.

Otra muestra de las características que tiene MicroPython y que lo hacen diferente a otros sistemas embebidos son:

- Tiene una RELP Interactiva (Read-Eval-Print Loop por sus siglas en Ingles). Es un pequeño programa que lee e interpreta los comandos del usuario, los evalúa y después imprime el resultado. Esto permite conectar alguna tarjeta (microcontrolador que soporte Python) y esta tiene que ejecutar el código sin necesidad de compilar ni cargar el programa.
- Posee muchas librerías de desarrollo. Así como el Lenguaje de Programación Python cuenta con un sin fin de librerías para la ejecución de tareas, MicroPython también viene bien cargado con

[7] http://micropython.org/

bastantes paquetes para ahorrar trabajo. Es posible ejecutar análisis de datos JSON desde un servicio web, búsqueda de texto en expresiones regulares o hasta levantar un Socket dentro de una red tan solo con las funciones ya precargadas.

- Extensibilidad. Para los usuarios avanzados de MicroPython, pueden extender de Python a funciones de más bajo nivel como C o C++, pudiendo mezclar códigos que requieran de ejecución más rápida a bajo nivel con MicroPython que es de más alto nivel.

T1.2: Diseño en bajo nivel de la implementación de referencia WoRMS.

En tarea se ha realizado el diseño general de WoRMS, una implementación experimental de un Web of Things (WoT) Servient para permitir el desarrollo ágil de WoT Applications. El diseño se ajusta a las recomendaciones generadas por el grupo de trabajo de WoT en el W3C, y la API pública sigue la interfaz descrita en el documento de la W3C WoT Scripting API.

Las características más destacables de WoRMS se describen a continuación. La información sobre cada uno de estos puntos se extenderá a lo largo del presente informe, y se encuentra más detallada en el entregable *E1.1: Diseño de la implementación de referencia WoRMS*.

- Soporte para Python 2.7, 3.6 y 3.7.
- Implementación completa de la Scripting API.
- Auto-descubrimiento multicast basado en mDNS.
- Modelo de programación I/O asíncrono basado en corrutinas.
- Implementación de plantillas de Protocol Binding (cliente y servidor) para los protocolos MQTT, HTTP, CoAP y WebSockets.

Antes de comenzar la descripción, se definen algunos términos propios del dominio de la W3C WoT que se mencionan repetidamente a lo largo de este proyecto en todas sus tareas. Los términos se incluyen en inglés por homogeneidad y fidelidad a la fuente.

- **Thing Description**: Una Thing Description (TD) es la representación formal de una Thing. Una TD se serializa en formato JSON o JSON-LD y contiene toda la información necesaria para operar con las interacciones de una Thing (Properties, Actions y Events).
- **Thing**: Una Thing es cualquier elemento virtual o físico que puede ser descrito por una TD. El objetivo principal de la WoT es establecer una serie de patrones (basados en los actualmente encontrados en la Web) para que las Things puedan interoperar entre sí de manera escalable y con mínimos requisitos de configuración por parte de operarios humanos.
- **Servient**: Un conjunto de componentes software que permite exponer (actuar como servidor) y consumir (actuar como cliente) Things. Un Servient incluye un WoT Runtime, una API pública según las especificaciones de la Scripting API y un conjunto arbitrario de plantillas de Protocol Binding. A continuación, se muestra un diagrama de los bloques que componen un WoT Servient. Se indican, así mismo, las funcionalidades que aporta o implementa WoRMS. WoRMS (o un WoT Servient cualquiera) puede interpretarse de manera paralela a un servidor Web. Ambos actúan

como la base software encargada de contener y exponer aplicaciones, así como de gestionar las operaciones de bajo nivel para permitir las comunicaciones de dichas aplicaciones en la Web.

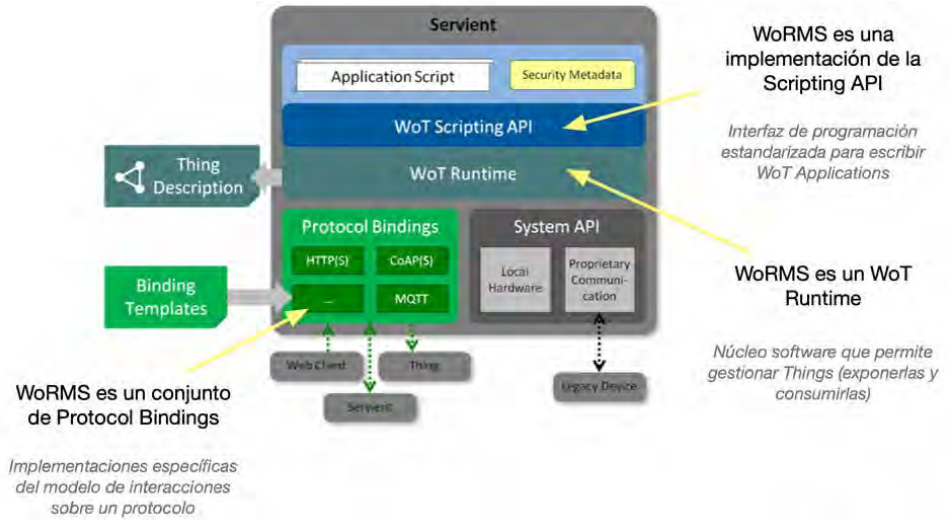


Figura 12: Diagrama de bloques de un WoT Servient.

- **Interaction Model**: Modelo teórico que engloba todas las interacciones que se pueden llevar a cabo sobre una Thing. Cualquier funcionalidad pública de una Thing (consultar la temperatura, reiniciar un sensor, etc) puede modelarse según el Interaction Model. Dentro de este modelo se describen tres patrones diferenciados: Event, Property y Action.
- **Property**: Representan atributos o valores (piezas indivisibles de información) que pueden ser leídas y/o escritas. Estos procesos de lectura y escritura se espera que no tengan una larga duración en el tiempo. Ejemplos:
  - La temperatura de un sensor de temperatura.
  - El peso que está midiendo una báscula.
- **Action**: Procedimientos de larga duración que no tienen cabida dentro de una Property y que suelen requerir múltiples pasos para ser llevados a cabo. Ejemplos:
  - Bajar una persiana automatizada.
  - Reiniciar una máquina virtual.
- **Event**: Ocurrencias observadas por la Thing que son comunicadas a los clientes que tienen una subscripción. La principal diferencia con las Property o Action es que la comunicación no es iniciada por el cliente, si no que responde a un cambio en el entorno de la Thing. Ejemplos:
  - Un sensor de presencia ha detectado una persona.
  - Una aspiradora robot se ha conectado a su dock de carga.
- **Protocol Binding**: Conjunto de patrones y plantillas para traducir operaciones conceptuales de alto nivel en el Interaction Model (leer una Property, invocar una Action) a operaciones concretas dentro de un protocolo. Es decir, una plantilla de Protocol Binding es la especificación que

determina cuestiones de bajo nivel tales como los formatos de serialización, los contenidos de los paquetes o los patrones de comunicación.

Modelo de programación I/O asíncrono.

La API de WoRMS se organiza en torno a un modelo de programación concurrente asíncrono basado en un **loop de eventos**. Concretamente se utiliza la implementación proporcionada por el módulo built-in **asyncio** de Python 3. Dado que en Python 2.7 no está disponible asyncio, se utiliza el framework **Tornado**. Este framework tiene la característica de que proporciona su propia implementación del loop de eventos en Python 2.7, pero utiliza asyncio siempre que esté disponible. De esta manera se consigue soportar ambas versiones con la misma base de código. La implementación de Tornado para Python 2.7 es una opción menos adecuada que asyncio (Tornado es un framework de terceras partes mientras que asyncio es un módulo oficialmente soportado del núcleo de Python), pero resulta un compromiso aceptable dada la falta de alternativa.

El loop de eventos es la entidad principal en una aplicación basada en este modelo de programación. El loop se ejecuta en un bucle infinito sobre el hilo principal. Cuando se realiza una petición de red, el loop salta a otro punto de la ejecución del programa y comprueba periódicamente por la recepción de la respuesta, cuando esta respuesta llega, la ejecución se retoma. La principal ventaja es que este tipo de operaciones asíncronas no malgastan ciclos de CPU de manera innecesaria esperando por respuestas de la red.

Las operaciones asíncronas se programan utilizando constructos similares a una función denominados **corrutinas**. Una corrutina se define y comporta igual que una función con una excepción notable: dentro de una corrutina se puede utilizar **await** o **yield** para invocar a otra corrutina de manera asíncrona. Cuando se invoca un corrutina de esta manera, se produce el salto dentro del loop que se comentaba anteriormente. Cuando la corrutina retorna, se retoma la ejecución donde se había dejado. Esto simplifica notablemente la programación de código asíncrono, ya que se puede escribir como si fuese código bloqueante sin necesidad de escribir callbacks o gestionar placeholders futuros.

Este modelo asíncrono es especialmente interesante para aplicaciones con computaciones ligeras y alta carga de interacción de red (como es el caso de WoRMS) pero no resultan tan adecuadas para cargas de trabajo computacionalmente costosas (después de todo, solo se dispone de un hilo).

A continuación se puede ver un diagrama de alto nivel en el que se comparan las dos aproximaciones comentadas hasta el momento

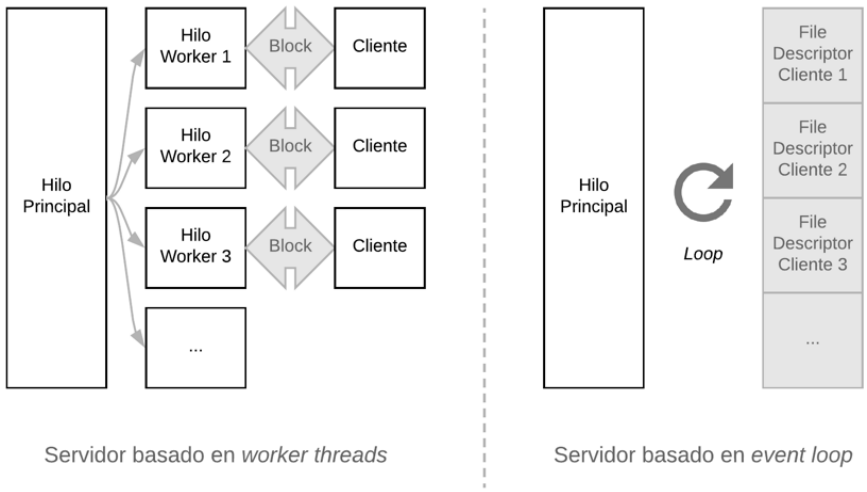


Figura 13: Comparativa servidor basado en *worker threads* vs servidor basado en *event loop*

Módulos de WoRMS.

En esta sección se presentan los módulos de mayor relevancia que componen la implementación de WoRMS:

- El conjunto de plantillas de Protocol Binding. Cada protocolo soportado incluye una implementación de cliente y una implementación de servidor. Dada su entidad, los detalles de implementación se incluyen en una sección dedicada posterior.
- Las interfaces principales de la W3C WoT Scripting API: los módulos de Exposed Thing, Consumed Thing y el objeto de entrada WoT.
- El módulo de descubrimiento multicast basado en mDNS (Multicast DNS) y DNS-SD (DNS Service Discovery). Multicast DNS difiere del protocolo DNS normal en que no existe una autoridad central que sirve como traductor de nombres. Los elementos de la red disponen de un agente mDNS que es capaz de responder a preguntas multicast sobre traducción de nombres y servicios cuando dichos elementos se dan por aludidos. Los nombres en una red mDNS son escogidos personalmente por los elementos de la red, sin necesidad de registrarlos en un servidor central. Estos nombres siguen una estructura formal descrita en la especificación de DNS-SD.
- El módulo Servient que proporciona la entidad principal que actúa como orquestador de los demás módulos.

A continuación, se puede ver un diagrama en el que se observan las relaciones entre módulos y sus interacciones desde un punto de vista general.

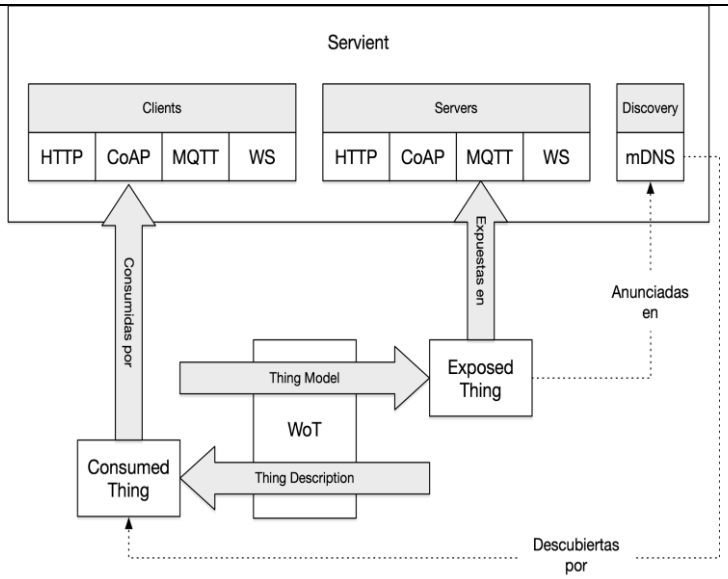


Figura 14: Relaciones e interacciones entre módulos de WoRMS.

Interfaz WoT

El objeto WoT proporciona un punto de entrada para la interacción con Things, ya sea su consumición o su exposición. El Servient actúa normalmente a modo de factoría de este objeto, ya que se retorna como resultado de la corrutina de arranque del Servient. Su interfaz viene determinada por la especificación de la W3C WoT Scripting API.

Interfaz Consumed Thing

Los objetos Consumed Thing sirven a modo de proxy local para acceder a Things remotas contenidas en otros Servients. Una Consumed Thing proporciona todos los metadatos necesarios para acceder a los puntos de entrada de cada uno de los protocolos para cada tipo de interacción. Los clientes instanciados a partir de las plantillas de Protocol Binding son capaces de interpretar esos metadatos para seleccionar dinámicamente el protocolo que se utilizará para acceder a la Thing remota. De esta manera el protocolo de comunicación resulta transparente para el programador que está manipulando la Consumed Thing.

Interfaz Exposed Thing

Los objetos Exposed Thing representan las Things locales contenidas en un Servient. Al contrario que con las Consumed Thing, las Exposed Thing son editables, por lo que el programador puede añadir o eliminar interacciones una vez creadas. El Servient es la entidad encargada de añadir la Exposed Thing a cada uno de los servidores instanciados a partir de las plantillas de Protocol Binding, es decir, todas las Exposed Thing contenidas en un Servient se exponen en cada uno de los protocolos configurados en dicho Servient.

Plantillas de Protocol Binding.

En esta sección se presentan los detalles del diseño de cada una de las plantillas de Protocol Binding para cada uno de los protocolos soportados en WoRMS. En el entregable E1.1 se pueden ver los mensajes

intercambiados para cada uno de los verbos soportados por cada uno de los elementos del interaction model (Property Read, Property Write, Property Observe, Action Invoke y Event Observe).

WebSockets

WebSockets es un protocolo nativo de la Web soportado por la mayoría de los navegadores actuales que permite establecer comunicaciones bidireccionales entre un cliente y un servidor.

Su principal aportación al contexto de la Web es el soporte nativo de patrones de mensaje push, donde es el servidor el que inicia la comunicación. HTTP no resulta especialmente adecuado en estos escenarios por estar basado en un patrón request-response.

Todas las interacciones en esta plantilla se llevan a cabo intercambiando mensajes serializados en JSON siguiendo la especificación JSON-RPC. Este protocolo está diseñado para permitir la invocación de métodos de manera remota utilizando mensajes JSON. Esto nos permite utilizar un patrón request-response en WebSockets como base, aprovechando a la vez sus capacidades push ahí donde sea necesario.

HTTP

HTTP es uno de los protocolos de capa de aplicación más relevantes dentro del contexto de Internet. Es la piedra angular sobre la que se asienta la Web y es, por consiguiente, un ciudadano de primer orden dentro de la Web of Things.

Resulta de especial utilidad para implementar patrones de comunicación request-response, aunque carece de mecanismos incorporados para permitir iniciar la comunicación desde el servidor e implementar patrones bidireccionales. Es por esto que se deben utilizar soluciones compromiso como el patrón long-polling para solventar este problema.

El patrón long-polling implica establecer una conexión desde el cliente al servidor de manera recurrente para esperar por nuevos eventos o mensajes. Estas conexiones permanecen abiertas durante un tiempo prolongado hasta que el servidor envía una respuesta u ocurre un timeout. De esta manera se minimiza la latencia y el coste de establecimiento de conexión derivado de lanzar peticiones constantes al servidor para recibir respuestas negativas.

Finalmente destacar que todos los mensajes intercambiados dentro de la plantilla HTTP se serializan en formato JSON.

MQTT

MQTT es un protocolo ligero nativo del dominio de la IoT basado en un patrón de comunicación publish-subscribe. En este patrón de comunicación existen un conjunto de categorías denominadas tópicos que sirven como primer nivel de organización de la comunicación. Los clientes se conectan a una entidad central encargada de gestionar los mensajes denominada broker. Una vez conectados, los clientes pueden publicar mensajes en tópicos o suscribirse a ellos. Todos los mensajes publicados en un tópico por cualquiera de los clientes son recibidos por todos los suscriptores de dicho tópico.

Dadas las características de MQTT, se necesita llevar a cabo un trabajo de adaptación para poder soportar patrones request-response dentro de este protocolo. La traducción de verbos de observación y suscripción a mensajes generados por el servidor es inmediata, lo cual no puede aplicarse a verbos que requieren comunicaciones síncronas.

Al igual que ocurre con las demás plantillas, los mensajes intercambiados dentro de esta plantilla se serializan en formato JSON.

CoAP

CoAP es un protocolo ligero de capa de aplicación transportado sobre UDP. Esto le permite tener una huella menor en términos de computación y volumen de transferencia de datos frente a alternativas basadas en TCP (por ejemplo, HTTP, MQTT o WebSockets).

CoAP está diseñado alrededor del patrón REST y presenta múltiples similitudes con HTTP desde el punto de vista arquitectural. Diferencias destacables con HTTP incluyen el soporte para multicast, lo que permite el autodescubrimiento, entre otras cosas, y el soporte para observación de recursos, lo que habilita el patrón de comunicación push.

Requisitos funcionales y no funcionales.

Como resultado de la fase inicial de diseño y planificación de la implementación, se han obtenido los siguientes requisitos funcionales y no funcionales.

T1.3: Diseño en bajo nivel de la implementación de referencia WoRMS Lite.

En esta tarea se detalla el diseño de la implementación de las recomendaciones del W3C sobre WoT para la interoperabilidad de sistemas IoT. Esta descripción se encuentra más ampliada en el correspondiente entregable *E1.2 Diseño de la implementación de referencia WoRMS Lite*.

Esta implementación se dirige a sistemas embebidos de capacidades limitadas, que puedan ejecutar MicroPython (George, 2019) y que posean capacidades de red (WiFi, Ethernet…). La arquitectura se basa en las recomendaciones del W3C y se limita a las capacidades software de los microcontroladores que ejecutan MicroPython.

Como se ha mencionado previamente, el sistema objetivo para esta implementación es un microcontrolador que ejecute MicroPython y que tenga capacidades de red, ya sean inalámbricas o cableadas. MicroPython es un sistema operativo embebido a la par que una implementación reducida de Python3 que proporciona una terminal, interfaces con los distintos pines del microcontrolador, árbol de directorios, gestión de excepciones y control de las interfaces de red. MicroPython es una implementación de código abierto y existen múltiples librerías diseñadas por la comunidad para ampliar sus funcionalidades.

Existen varios microcontroladores capaces de ejecutar MicroPython además de aquellos que aun pudiendo hacerlo, no poseen interfaces de red. Entre ellos destacan:

- Pycom (GPy, FiPy, WiPy, etc.).

- ESP8266 (Múltiples versiones).
- ESP32 (Múltiples versiones).
- Pyboard.

Teniendo en cuenta las restricciones anteriormente mencionadas, se ha realizado la implementación de una librería en MicroPython que permite la creación de un servidor WoT en uno de los dos protocolos implementados (HTTP y MQTT). Este servidor expondrá de manera asíncrona una serie de recursos definidos por una TD autogenerada en base a los mismos. La implementación se divide en varios paquetes o "Building Blocks" (BBs) como se definen en las recomendaciones del W3C (ver figura inferior).

Para esta implementación, se han diseñado tres de estos BBs: "Thing Description" (TD), "Scripting API" y "Protocol Bindings". Además, de la denominada "System API" que permite interactuar con el hardware local, sensores, actuadores…

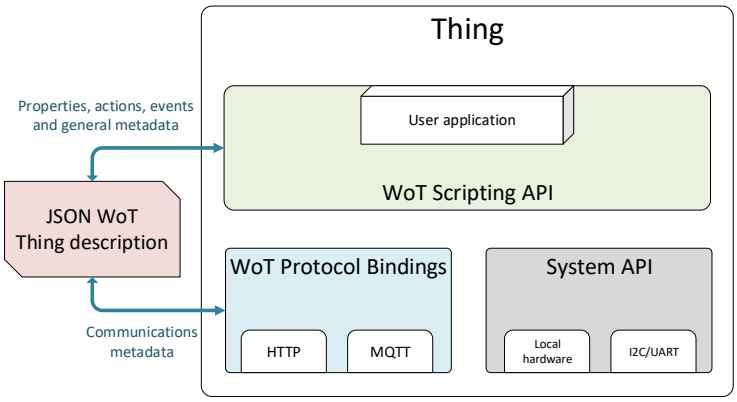


Figura 15: Arquitectura de los Building Blocks

MicroPython

MicroPython es una implementación del lenguaje de programación Python 3 que incluye una pequeña parte de sus funcionalidades, permitiendo su ejecución en entornos de reducidas capacidades, como los microcontroladores. MicroPython intenta ser lo más compatible posible con Python, para permitir la traducción de código lo más simple posible.

La librería de WoT creada soporta la versión de MicroPython 1.9 (George, The MicroPython project, 2019) o superior, aunque cabe destacar que este lenguaje está constantemente en desarrollo y futuras versiones podrían romper partes de la misma como la conexión WiFi o las librerías de acceso al hardware.

Esta librería está implementada utilizando un modelo asíncrono, basado en una implementación de la librearía asyncio para MicroPython denominada uasyncio (Sokolovsky & George, uasyncio, 2019). Esta implementación se basa en la utilización de un bucle infinito (loop), en el que se ejecutan las corrutinas. Estas corrutinas se encargan de tareas diversas, como la gestión de mensajes entrantes, llamada a los controladores de las propiedades y acceso al hardware.

Todas las funciones que se ejecutan durante el funcionamiento de la librería han de ser funciones asíncronas que se definen añadiendo "async" a la definición de las mismas. Como elemento a destacar,

todas las funciones que sirven como controladores de los recursos expuestos han de ser asíncronas, y por tanto se definirán así:

```
async def read_accelerometer_data(self):
      await asyncio.sleep_ms(10)
      return accelerometer.acceleration()
```

Todas las funciones asíncronas deben incluir una instrucción *await* o *yield* que permita al bucle seguir ejecutándose. En caso de no necesitarla, como es el caso en el ejemplo anterior, se utilizará un pequeño *sleep* asíncrono proporcionado por la propia librería uasyncio.

Thing Description

La descripción asociada a una Thing es un modelo formal que representa una Thing Web, generalmente en formato JSON. Una TD describe los metadatos, las interfaces y los recursos que una Thing expone de manera inteligible tanto para una máquina como para un usuario. Las TD proveen una serie de interacciones basadas en un pequeño vocabulario que permite la integración con otras Things o aplicaciones de manera simple. Estas interacciones se dividen en:

- Propiedades: Exponen valores internos de la Thing, permitiendo o no su modificación.
- Acciones: Permiten invocar acciones en la Thing, que pueden o no modificar su estado interno.
- Eventos: Permite definir alertas o mensajes que se generan bajo ciertas condiciones, este tipo de interacción se inicia desde la propia Thing.

Cada una de las interacciones definidas en estos tres tipos posee metadatos referentes a las comunicaciones denominados *forms,* estos contienen información tanto del tipo de protocolo que se utiliza para exponer la Thing, el tipo de respuesta que se espera, seguridad, etc.; además de una URI que indica la dirección a consultar para obtener los datos deseados.

Scripting API

WoT provee una capa de interoperabilidad basada en la forma en la que las Things se definen, pudiendo estas ser expuestas y/o consumidas. Para exponer o consumir una Thing, es necesaria una Thing Description y el software necesario para exponer sus recursos o accederlos. La implementación realizada de este BB permite servir tanto la TD como los recursos. Para ello ofrece una serie de métodos que permiten añadir, eliminar y crear funcionalidades para cada uno de los recursos.

En primer lugar, tenemos la clase *WoT* que permite crear un objeto de la clase *ExposedThing* a partir de una TD previamente construida mediante el método *produce*. Por otro lado, la clase *ExposedThing* ofrece métodos para la creación de propiedades, acciones y eventos, así como para asignar *handlers* o controladores a cada uno de ellos. Cada recurso solo podrá tener un controlador, exceptuando las propiedades que pueden tener dos, uno para lectura y otro para escritura en caso de permitirse esta última, por lo que si se añaden varios controladores para un recurso sólo se mantendrá el último.

Las restricciones de este tipo de microcontroladores sólo permiten tener un servicio expuesto, por lo que una vez se comienza a exponer los recursos no es posible iniciar otro servidor o hacer cambios en el actual de manera interna, solo se podría interaccionar mediante las URIs expuestas y el protocolo seleccionado.

En cuanto a la definición de eventos, cambia de un protocolo a otro. En MQTT es posible implementar un método asíncrono que publique el evento cuando se cumpla cierta condición, pero en HTTP es necesario hacer *long polling* para "suscribirse" al evento, ya que ese protocolo no se ha pensado para ello. Esto puede resultar en una distinta implementación del *event_handler* en función del protocolo a utilizar

Protocol Bindings

En esta implementación, los *Protocol Bindings* nos permitirán enlazar los distintos recursos con sus respectivos métodos de acceso a través de un protocolo determinado. Una Thing solo puede tener un protocolo asociado, y todos sus recursos se expondrán a través del mismo. Las implementaciones son completamente asíncronas, lo que permite responder a múltiples peticiones al mismo tiempo, así como mantener procesos ejecutándose en paralelo, lo cual es muy conveniente para la monitorización de eventos. Las particularidades de cada uno se explican a continuación.

HTTP REST

La implementación del protocolo HTTP REST se ha realizado mediante la librería Picoweb (Sokolovsky, 2019) que permite la creación de un servidor web asíncrono en MicroPython. Este se crea tras la invocación del método *expose* de la clase *ExposedThing* pasando el parámetro "HTTP". Internamente, se utiliza la *ExposedThing* para recabar los distintos recursos a exponer, asignarles una URL, asociar los distintos *handlers* y añadirla al servidor, así como almacenarlas en la TD. Finalmente, se genera la TD por última vez con todas las URIs incluidas y se expone en el directorio raíz ("http://mything/").

MQTT

La implementación del protocolo MQTT sea ha realizado mediante la librería mqtt_as (Hinch, 2019) que implementa el protocolo MQTT de manera asíncrona. Este se crea tras la invocación del método *expose* de la clase *ExposedThing* pasando el parámetro "MQTT". Los tópicos o URIs MQTT asociados a cada propiedad serán el doble que en HTTP, ya que es necesario uno para recibir las peticiones (tópicos de entrada) y otro para publicar las respuestas (tópicos de salida). La *ExposedThing* se utiliza para recabar los distintos recursos a exponer, asignarles sus respectivos tópicos, asociar los distintos *handlers* y añadirlos al servidor que escuchará los tópicos de entrada, así como almacenarlas en la TD. Finalmente, se genera la TD por última vez con todas las URIs incluidas y se expone en el directorio raíz ("mqtt://mything"). En cuanto al bróker MQTT que se utilizará, queda a elección del usuario indicar cuál es su IP y puerto.

T1.4 Diseño general de la aplicación WoT demostradora.

En esta tarea se ha realizado el diseño de alto nivel de la aplicación WoT que se utilizará para demostrar las funcionalidades de las implementaciones WoRMS y WoRMS Lite. Más adelante, en la tarea T4.1 se realizará el diseño en detalle de la aplicación demostradora.

El resultado de este trabajo ha sido el diseño de alto nivel de la aplicación, de forma que se contemple la incorporación de una instancia de cada una de las versiones de la implementación de referencia, además

de diferentes sensores y otros componentes o dispositivos habituales en un despliegue IoT en el ámbito industrial, como pueden ser PLCs o servidores OPC.

A partir de este diseño general, y de los diseños de las implementaciones realizados en las tareas anteriores, se ha obtenido también un listado de requisitos no funcionales de alto nivel.

El diseño de la aplicación WoT demostradora se basa en un conjunto de sensores conectados a un PLC que a su vez está expuesto por un servidor OPC UA, uno de los protocolos más comunes en entorno industrial para interoperabilidad de procesos industriales.

El servidor OPC UA está encapsulado por un servidor WoT implementado sobre WoRMS configurado con el rango completo de implementaciones de Protocol Binding.

Un cliente WoT también implementado sobre WoRMS se encarga de hacer de pasarela y expone una API consumida por la aplicación Web que será la interfaz para los usuarios.

Con el objetivo de demostrar también la validez y fiabilidad de la implementación WoRMS Lite, se conecta un microcontrolador directamente a un subconjunto de los sensores. Sobre este microcontrolador se despliega un servidor WoT que se integra con el resto de la aplicación a través MQTT usando el broker común. La aplicación Web interactúa directamente con la interfaz Websockets del broker MQTT sin necesidad de pasar por la API de traducción.

A continuación, se puede ver un diagrama que describe la arquitectura de la aplicación WoT demostradora.

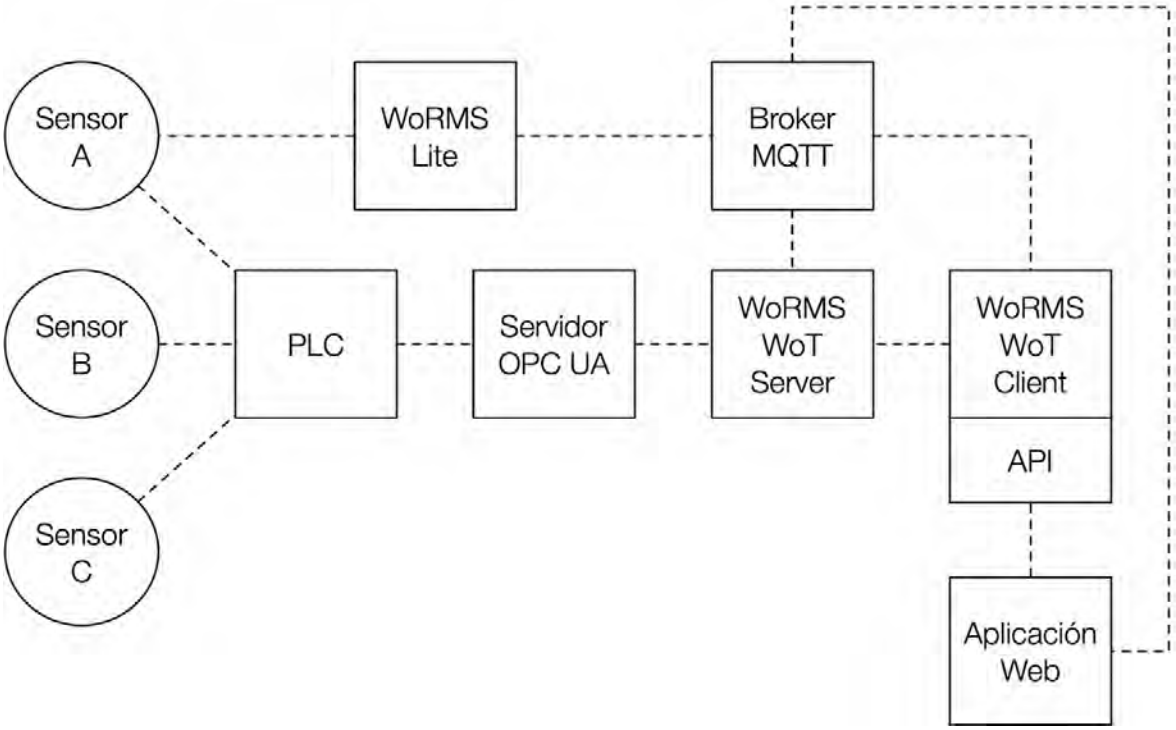


Figura 16: Diagrama de la arquitectura de la aplicación WoT demostradora.

Requisitos no funcionales.

En la siguiente tabla se indican los requisitos no funcionales de alto nivel para el despliegue de la aplicación WoT demostradora:

Tabla 2: Requisitos no funcionales de la aplicación WoT demostradora.

ID	Descripción	Prioridad
RNF-1	La aplicación Web puede interactuar con el servidor WoT sobre WoRMS a través de HTTP, Websockets y MQTT.	Alta
RNF-2	La aplicación Web puede interactuar con el servidor WoT sobre WoRMS a través de CoAP.	Media
RNF-3	La aplicación Web puede interactuar con el servidor WoT sobre WoRMS Lite a través de MQTT.	Alta
RNF-4	La aplicación Web puede interactuar con el servidor WoT sobre WoRMS Lite a través de HTTP.	Baja
RNF-5	Se utiliza OPC UA como tecnología de interoperabilidad industrial.	Alta
RNF-6	Se utilizan sensores y/o actuadores reales (no simulados).	Alta
RNF-7	Se incluyen ejemplos de todos los tipos de interacción (Event, Action y Property) en la aplicación Web.	Alta
RNF-8	La aplicación Web está apropiadamente adaptada a móvil y desktop.	Alta

Hito 2: Diseño y desarrollo de la implementación de referencia WoRMS

El objetivo principal de este hito consiste en diseñar y desarrollar la versión completa de la implementación WoRMS, incluyendo todos los Protocol Bindings y los módulos que dan soporte a las funcionalidades expuestas por el WoT Runtime a través de la Scripting API. Este hito se divide en 5 tareas,

de las cuales las tres primeras se han iniciado en la anualidad 2018, si bien todas finalizan en el año 2019, por lo que en este informe se resumen los trabajos realizados durante 2018 en dichas tareas:

- T2.1: Diseño y desarrollo del WoT Runtime.
- T2.2: Diseño y desarrollo del Protocol Binding HTTP.
- T2.3: Diseño y desarrollo del Protocol Binding WebSsockets.
- T2.4: Diseño y desarrollo del Protocol Binding CoAP.
- T2.5: Diseño y desarrollo del Protocol Binding MQTT.

T2.1: Diseño y desarrollo del WoT Runtime

El WoT Runtime proporciona funcionalidades comunes para la construcción de WoT Servients, actuando como el punto de unión del resto de componentes. Sus tareas más relevantes incluyen las siguientes:

- Gestión de las Things expuestas al exterior y consumidas desde Servients remotos.
- Inicialización, control y destrucción de los servidores que implementan las plantillas de Protocol Binding. Estos servidores deben ser notificados de cambios en las Things para adaptarse consecuentemente.
- Exposición del catálogo de Things actualmente contenido en el Servient.
- Validación y serialización de Thing Descriptions.
- Integración con mecanismos de descubrimiento.

La entidad de mayor relevancia dentro del WoT Runtime es la clase Servient, en la que se invirtieron la mayoría de los esfuerzos de desarrollo para este periodo.

El esquema mostrado a continuación proporciona una vista general del diseño y los componentes existentes. Aunque los componentes aquí indicados se encuentran en un estado de avance notable, se reservan trabajos de optimización y refinamiento de la funcionalidad, así como la integración de mecanismos de descubrimiento.

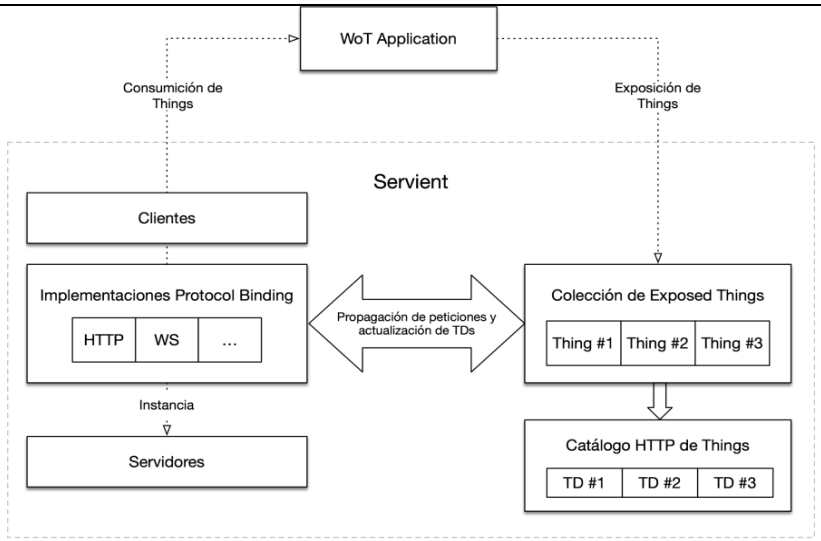


Figura 17: Esquema de diseño y componentes del Servient.

A continuación, se adjunta un extracto de código en Python que representa la interfaz de la clase Servient para clarificar su funcionalidad. La implementación de los métodos no se proporciona por ser de excesiva longitud.

```
class Servient(object):
    """An entity that is both a WoT client and server at the same time.
    WoT servers are Web servers that possess capabilities to access underlying
    IoT devices and expose a public interface named the WoT Interface that may
    be used by other clients.
    WoT clients are entities that are able to understand the WoT Interface to
    send requests and interact with IoT devices exposed by other WoT servients
    or servers using the capabilities of a Web client such as Web browser."""

    def __init__(self, hostname=None, catalogue_port=9090,
                 clients=None, clients_config=None,
                 dnssd_enabled=False, dnssd_instance_name=None):
        pass

    @property
    def is_running(self):
```

```
        """Returns True if the Servient is currently running
        (i.e. the attached servers have been started)."""
        pass

    @property
    def hostname(self):
        """Hostname attached to this servient."""
        pass

    @property
    def exposed_thing_set(self):
        """Returns the ExposedThingSet instance that
        contains the ExposedThings of this servient."""
        pass

    @property
    def exposed_things(self):
        """Returns an iterator for the
        ExposedThings contained in this Servient."""
        pass

    @property
    def servers(self):
        """Returns the dict of Protocol Binding
        servers attached to this servient."""
        pass

    @property
    def clients(self):
        """Returns the dict of Protocol Binding
        clients attached to this servient."""
        pass
```

```
    @property
    def catalogue_port(self):
        """Returns the current port of the HTTP
        Thing Description catalogue service."""
        pass

    @catalogue_port.setter
    def catalogue_port(self, port):
        """Enables the servient TD catalogue in the given port."""
        pass

    @property
    def dnssd(self):
        """Returns the DNS-SD instance linked to
        this Servient (if enabled and started)."""
        pass

    @property
    def dnssd_instance_name(self):
        """Returns the user-given DNS-SD service instance name."""
        pass

    def select_client(self, td, name):
        """Returns the Protocol Binding client instance to
        communicate with the given Interaction."""
        pass

    def add_client(self, client):
        """Adds a new Protocol Binding client to this servient."""
        pass
```

```
    def remove_client(self, protocol):
        """Removes the Protocol Binding client
        with the given protocol from this servient."""
        pass

    def add_server(self, server):
        """Adds a new Protocol Binding server to this servient."""
        pass

    def remove_server(self, protocol):
        """Removes the Protocol Binding server
        with the given protocol from this servient."""
        pass

    def refresh_forms(self):
        """Cleans and regenerates Forms for all the
        ExposedThings and servers contained in this servient."""
        pass

    def enable_exposed_thing(self, thing_id):
        """Enables the ExposedThing with the given ID.
        This is, the servers will listen for requests for this thing."""
        pass

    def disable_exposed_thing(self, thing_id):
        """Disables the ExposedThing with the given ID.
        This is, the servers will not listen for requests for this thing."""
        pass

    def add_exposed_thing(self, exposed_thing):
        """Adds an ExposedThing to this Servient.
        ExposedThings are disabled by default."""
```

```
        pass

    def remove_exposed_thing(self, thing_id):
        """Disables and removes an ExposedThing from this Servient."""
        pass

    def get_exposed_thing(self, thing_id):
        """Finds and returns an ExposedThing contained
        in this servient by Thing ID. Raises ValueError
        if the ExposedThing is not present."""
        pass

    def disable_td_catalogue(self):
        """Disables the servient TD catalogue."""
        pass

    def start(self):
        """Starts the servers and returns an instance of the WoT object."""
        pass

    def shutdown(self):
        """Stops the server configured under this servient."""
        pass
```

T2.2: Diseño y desarrollo del Protocol Binding HTTP

La plantilla de HTTP permite trasladar el modelo de interacciones de alto nivel de Things a mensajes HTTP. El trabajo invertido en esta tarea durante la anualidad 2018 se centró en lo siguiente:

- Extender el diseño de alto nivel de la plantilla HTTP generada durante la fase de diseño inicial.
- Desarrollar implementaciones concretas para el servidor y el cliente de la plantilla HTTP.

La implementación de ambos componentes (cliente y servidor) se está llevando a cabo sobre el framework Tornado. Dentro del contexto del servidor se desarrollan cinco controladores diferenciados para soportar cada uno de los verbos de interacción (véase diagrama a continuación).

Se dispone además de una implementación básica del cliente HTTP pendiente de optimizaciones (por ejemplo, ajuste de los tiempos por defecto de timeout para la aplicación del patrón long-polling).

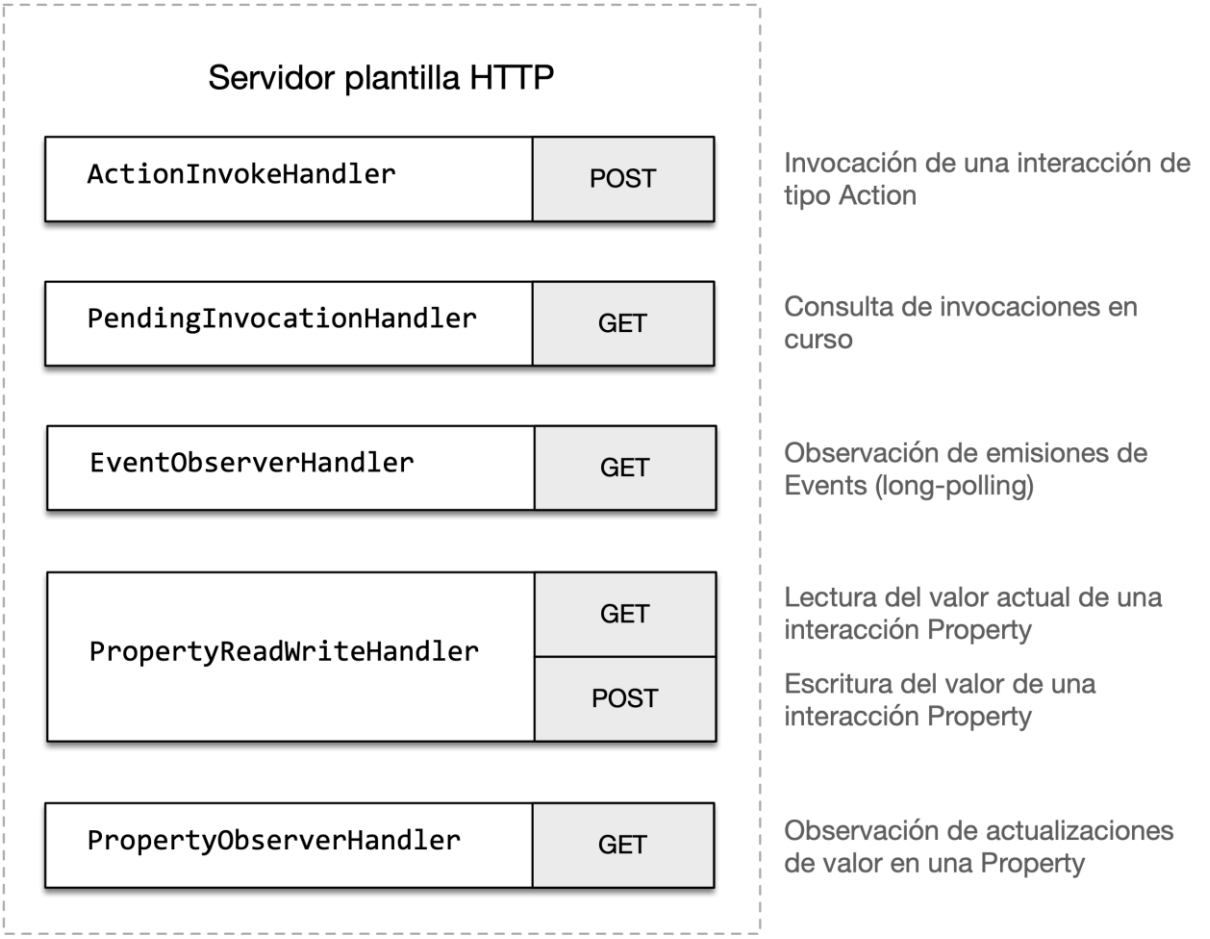


Figura 18: Diagrama controladores plantilla HTTP

Diagramas de secuencia servidor HTTP

A continuación, se muestra el conjunto de diagramas de secuencia que representan las operaciones de cada uno de los verbos del modelo de interacciones en el contexto del servidor HTTP.

Lectura y escritura de Property

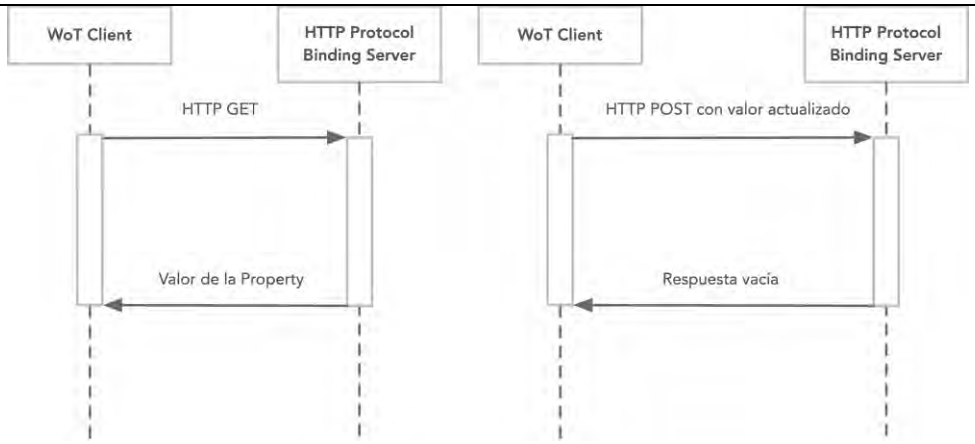


Figura 19: Diagrama lectura y escritura de Property

Observación de actualizaciones de Property

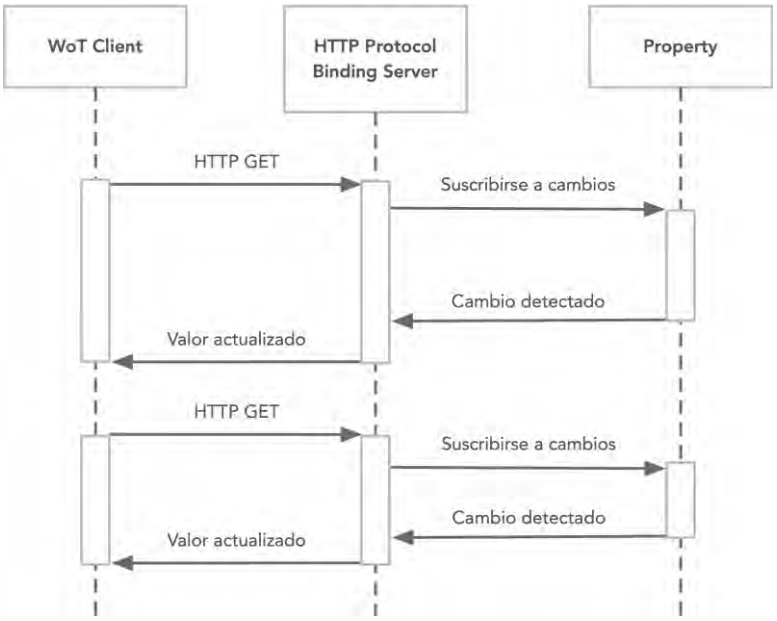


Figura 20: Diagrama observación de actualizaciones de Property

Invocación de Action

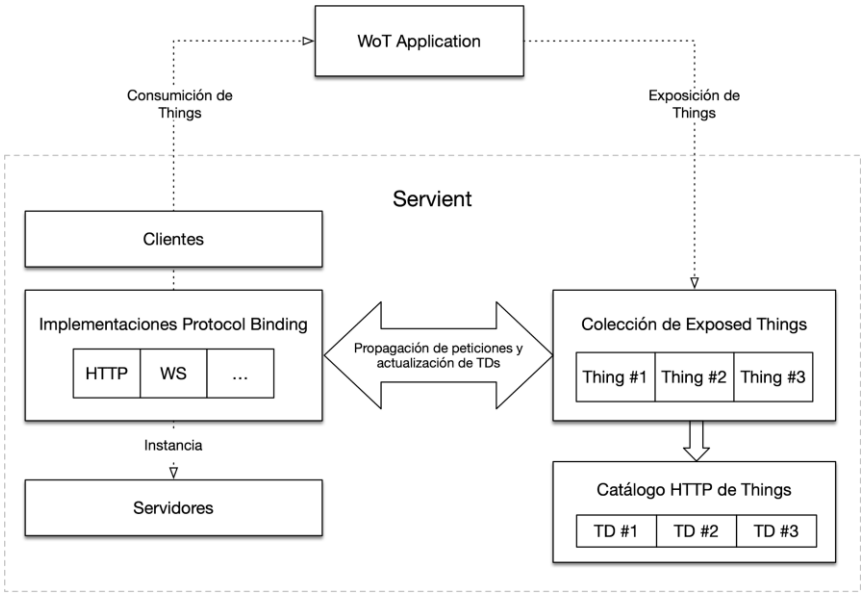


Figura 21: Diagrama invocación de Action

Observación de emisiones de Event

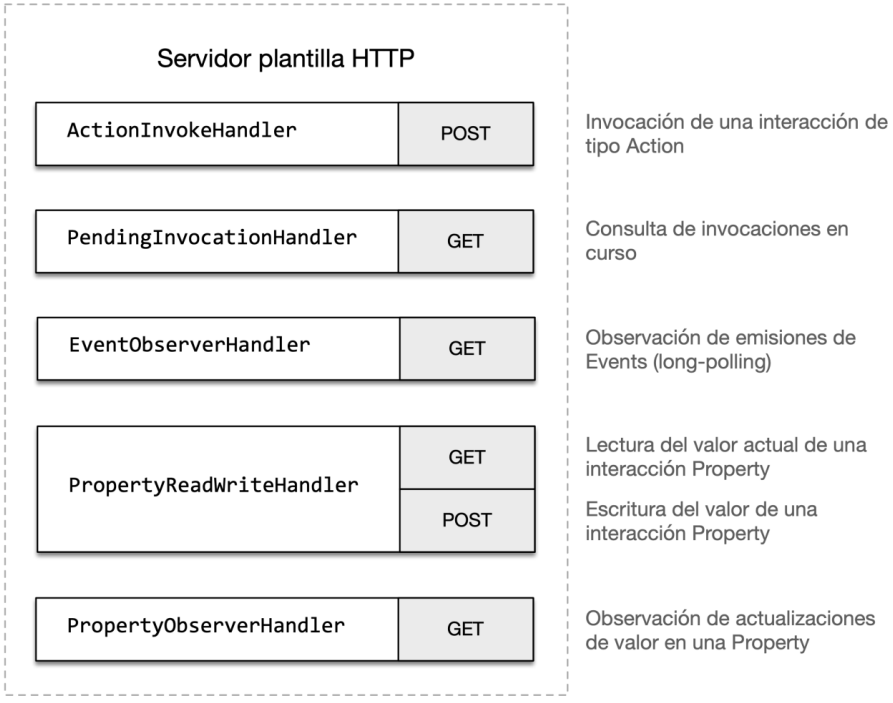


Figura 22: Diagrama observación de emisiones de Event

T2.3: Diseño y desarrollo del Protocol Binding WebSsockets

De manera similar a las demás plantillas, este módulo permite traducir el modelo de interacciones de Thing al protocolo Websockets.

Los recursos de esta anualidad 2018 se han destinado principalmente en llevar a cabo la implementación del servidor para el diseño obtenido en la primera fase, así como en desarrollar todos los mecanismos de validación y control de flujo del protocolo de comunicación.

Este protocolo especifica el formato y flujo de los mensajes JSON-RPC intercambiados sobre Websockets por cliente y servidor. La necesidad de implementar esta capa de abstracción es una desventaja con respecto al protocolo HTTP, que ya define un conjunto limitado y bien conocido de verbos, aun así, las capacidades de comunicación bidireccional de Websockets hacen que ciertos patrones (por ejemplo, observación de eventos) suponga una implementación casi directa, en contraste con los mecanismos más complejos que hay que utilizar sobre HTTP (long-polling).

Justificación 2019

Durante el periodo que cubre este informe, se ha finalizado por completo la ejecución de las tareas correspondientes al Hito 2 y se ha comenzado a trabajar en el Hito 3, ejecutándose aproximadamente la mitad de los trabajos previstos para ese hito.

A continuación, se describen las tareas realizadas en cada uno de los hitos de trabajo mencionados anteriormente y los resultados conseguidos.

Hito 2: Diseño y desarrollo de la implementación de referencia WoRMS

El objetivo principal de este hito consiste en diseñar y desarrollar la versión completa de la implementación WoRMS, incluyendo todos los Protocol Bindings y los módulos que dan soporte a las funcionalidades expuestas por el WoT Runtime a través de la Scripting API. Este hito se divide en 5 tareas que se han finalizado a lo largo de la presente anualidad 2019.

- T2.1: Diseño y desarrollo del WoT Runtime.
- T2.2: Diseño y desarrollo del Protocol Binding HTTP.
- T2.3: Diseño y desarrollo del Protocol Binding WebSsockets.
- T2.4: Diseño y desarrollo del Protocol Binding CoAP.
- T2.5: Diseño y desarrollo del Protocol Binding MQTT.

T2.1: Diseño y desarrollo del WoT Runtime

Esta tarea se inició en la anualidad 2018 y se finalizó en la anualidad 2019 según la planificación temporal prevista. En el informe de justificación de la anualidad 2018 se detallaron los trabajos realizados en ese periodo, por lo que aquí se resumen únicamente los trabajos ejecutados en 2019.

El WoT Runtime es la capa que contiene al Servient e implementa los detalles de orquestación de Things. Sobre esta capa se despliegan las aplicaciones WoT, de forma que puedan gestionar y descubrir nuevas Things.

La entidad de mayor relevancia dentro del WoT Runtime es la clase Servient sobre la que se han llevado a cabo trabajos de optimización y refinamiento de la funcionalidad, así como la integración de mecanismos de descubrimiento.

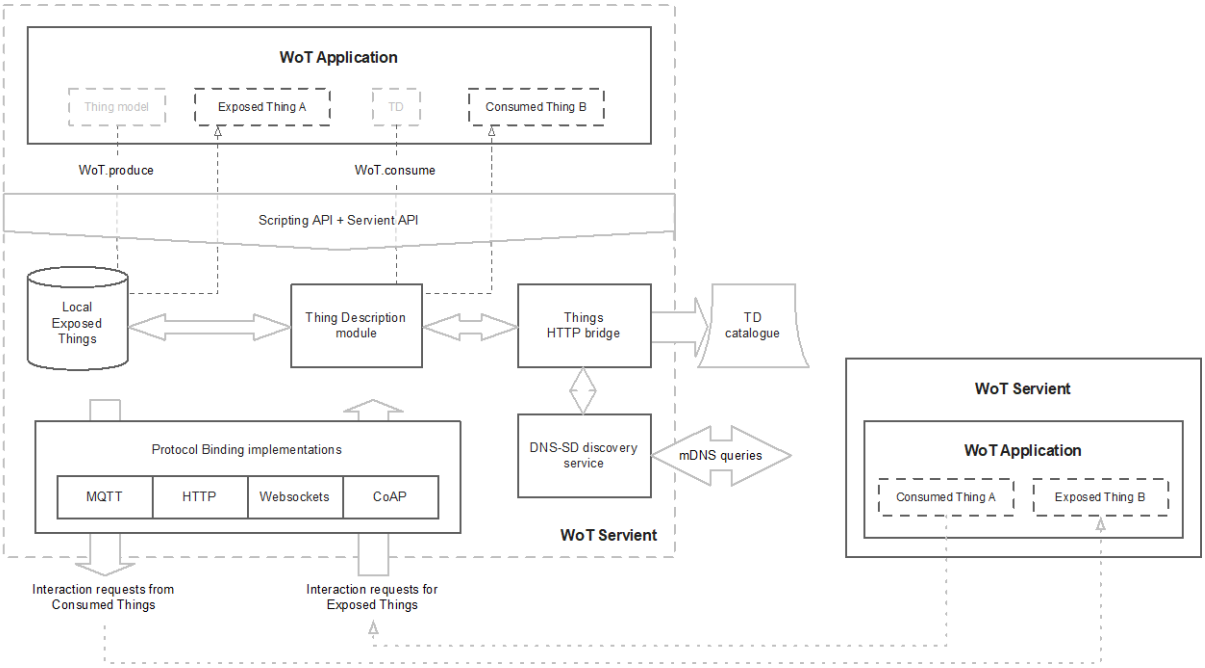


Figura 23. Arquitectura de WoTPy (WoRMS dentro del ecosistema Python)

El módulo de implementaciones de Protocol Bindings contiene a su vez cuatro submódulos que implementan una interfaz común que se expone de manera interna al resto de componentes de WoTPy.

El Servient (la clase principal que actúa de contenedor de Things y orquestador de clientes y servidores) observa el conjunto de Exposed Things actualmente disponibles e invoca a los Protocol Bindings para regenerar las URIs y los parámetros de conexión para cada una de las interacciones, lo que ocurre cada vez que se detectan cambios. Todas las implementaciones de los Protocol Bindings operan de la misma manera a alto nivel, es decir, en el lado de la red exponen y consumen interacciones de Things siguiendo los detalles de sus protocolos, mientras que en el lado del Servient se comunican con las Things siguiendo la interfaz de la Scripting API.

Las aplicaciones WoT pueden descubrir nuevas Things usando dos métodos:

1) Un método local que restringe la búsqueda a las Things registradas en el propio Servient.
2) Un método multicast basado en las tecnologías DNS Service Discovery (DNS-SD) y Multicast DNS (mDNS).

Las aplicaciones WoT pueden además interactuar con las capas inferiores del Servient utilizando dos APIs diferenciadas:

1) Servient API. Permite que el programador interaccione con la base de datos local que contiene las Things, así como configurar los clientes y servidores del Servient.
2) Scripting API. Interfaz común definida en las especificaciones que encapsula el funcionamiento interno del WoT Runtime.

El módulo de Thing Description (TD) contiene la lógica para validar documentos TD de acuerdo con la especificación del W3C.Todas las TD externas son validadas como paso previo a la construcción de una nueva Consumed Thing para asegurarse que siguen el formato esperado. Este módulo también tiene la capacidad de serializar Things internas a representación textual para su transmisión a través de la red (y viceversa).

Los Servient dependen del módulo Things HTTP Bridge para exponer el catálogo interno de Things de cara a la red. Este servicio es el que se registra bajo DNS-SD y que puede ser descubierto de manera dinámica utilizando descubrimiento multicast. Las Things se exponen en formato JSON sobre HTTP, y se recuperan utilizando peticiones GET a una URL que contiene el identificador de la Thing en cuestión. Los identificadores se pueden recuperar de un mapa global que está disponible en la raíz de este servidor HTTP.

T2.2: Diseño y desarrollo del Protocol Binding HTTP

Al igual que en la tarea anterior, esta tarea se inició en la anualidad 2018 y se finalizó en la anualidad 2019 según la planificación temporal prevista. En el informe de justificación de la anualidad 2018 se detallaron los trabajos realizados en ese periodo, por lo que aquí se resumen únicamente los trabajos ejecutados en 2019.

El trabajo invertido en esta tarea durante la anualidad 2019 se centra en la implementación entre las acciones de alto nivel que pueden ser ejecutadas sobre una Thing y los mensajes intercambiados con el servidor, utilizando el Protocol Binding HTTP.

HTTP es un protocolo sin estado basada en el modelo de petición y respuesta que sirve como cimiento de la Web. Esto quiere decir que es una parte fundamental de la mayoría de los entornos WoT. La mayoría de las implementaciones de APIs HTTP en el dominio de la IoT se encuentran actualmente basadas en el modelo REST, por lo que esta es la aproximación tomada en WoTPy.

HTTP REST presenta algunos problemas cuando se aplica en el contexto de la WoT, siendo uno de los más notables la ausencia de una solución con soporte extendido para mensajería iniciada desde el lado del servidor. Una alternativa bastante común es el patrón long-polling, en el que el cliente lanza peticiones de manera constante al servidor para esperar por la aparición de nuevos datos; estas peticiones se mantienen abiertas en el servidor durante un tiempo prolongado cuando no hay nuevos datos para intentar ahorrar el coste de abrir y cerrar la conexión constantemente.

Se identifican cinco recursos HTTP diferenciados que se pueden ver a continuación. Los verbos de interacción se mapean a métodos HTTP de acuerdo con los requisitos del patrón REST (uniform interface constraint).

- **Property.** Representa el valor de una propiedad. Se utilizan peticiones GET para leer el valor (read), mientras que se usan peticiones PUT para actualizarlo (write).
- **Property Subscription.** Representa una suscripción a notificaciones de actualización de propiedades (observe). Se utiliza el patrón long-polling, de manera que los clientes lanzan peticiones GET que permanecen abiertas hasta que el servidor dispone de una notificación; la suscripción se destruye automáticamente una vez que se ha generado una respuesta.
- **Action.** Representa procedimientos que pueden ser invocados (invoke) usando peticiones POST. Una petición exitosa retorna automáticamente con la URI de la Action Invocation que representa el procedimiento en curso.
- **Action Invocation.** Representa instancias de acciones que han sido previamente invocadas. Los clientes utilizan peticiones GET para conocer el estado actual de la invocación y recuperar sus resultados, o la causa de error.
- **Event Subscription.** Representa una suscripción a notificaciones de un evento en particular. La implementación es similar a la de Property Subscription, es decir, se utiliza el patrón long-polling.

T2.3: Diseño y desarrollo del Protocol Binding WebSsockets

La tarea T2.3 fue la última tarea iniciada en la anualidad 2018. Por ello, i al igual que sucedió con las dos tareas anteriores, en este informe se resumen únicamente los trabajos ejecutados en 2019, estando los trabajos de 2018 documentados en el correspondiente informe de justificación.

El trabajo invertido en esta tarea durante la anualidad 2019 se centra en la implementación entre las acciones de alto nivel que pueden ser ejecutadas sobre una Thing y los mensajes intercambiados con el servidor, utilizando el Protocol Binding WebSockets.

Websockets es un protocolo de comunicación bidireccional soportado en todos los navegadores modernos; resulta especialmente adecuado para comunicaciones iniciadas desde el servidor (en contraste con HTTP). Se puede usar en combinación con HTTP para cubrir la mayoría de las necesidades de los escenarios de comunicación posibles en la Web.

La implementación está organizada alrededor de un modelo de llamada a procedimientos remotos (RPC) representada con JSON-RPC 2.0. Se identifican cuatro mensajes que pueden ser intercambiados entre cliente y servidor durante una sesión:

- **Request.** Mensajes enviados por los clientes para iniciar la llamada a un procedimiento; contienen el nombre del método y los parámetros de la llamada. Existe un método independiente para cada verbo de interacción con el mismo nombre excepto para el verbo unsubscribe, que se cubre con el método dispose.

- **Response.** Mensajes enviados por el servidor que contienen las respuestas de las llamadas realizadas en peticiones request.
- **Error.** Es un tipo específico de respuesta que genera el servidor cuando surge un error durante el procesamiento de una request. Contiene detalles del error en vez del resultado de la llamada.
- **Emitted Item.** Estos mensajes notifican al cliente sobre nuevos eventos generados en el contexto de una suscripción. Pueden ser emitidos por el servidor en un número indefinido sin previa intervención del cliente (excepto para el establecimiento inicial de la suscripción).

Todos los mensajes request incluyen un identificador único que se referencia en la response asociada para permitir su identificación. Esto es necesario porque todos los mensajes se intercambian en el mismo canal (conexión) y pueden llegar fuera de orden con una latencia indeterminada; lo cual supone una de las características más interesantes del módulo Websockets, ya que permite simplificar la gestión de peticiones de larga duración (invocación de acciones) sin requerir la utilización de long-polling u otros patrones poco óptimos. Algo similar ocurre en el caso de las suscripciones, ya que todos los eventos de una suscripción incluyen su identificador único.

T2.4: Diseño y desarrollo del Protocol Binding CoAP

El trabajo invertido en esta tarea durante la anualidad 2019 se centra en la implementación entre las acciones de alto nivel que pueden ser ejecutadas sobre una Thing y los mensajes intercambiados con el servidor, utilizando el CoAP Protocol Binding.

CoAP es un protocolo de petición – respuesta especialmente indicado para dispositivos con capacidades limitadas (por ejemplo, microcontroladores). Presenta múltiples similitudes con HTTP (los métodos soportados son un subconjunto de los de HTTP) por lo que el diseño de esta implementación es bastante similar.

En comparación con los demás protocolos considerados, CoAP se transporta sobre UDP en vez de TCP. Esto tiene como consecuencia un gasto menor de transferencia de datos y unas necesidades de computación reducidas. CoAP presenta además un mecanismo para salvar la falta de fiabilidad asociada a UDP; para ello se utilizan mensajes de tipo CON, que deben ser confirmados por el otro extremo de la comunicación usando mensajes RST o ACK. También existe la posibilidad de usar mensajes sin confirmación (mensajes NON).

Una de las diferencias más notables con HTTP es la existencia de un mecanismo para iniciar comunicaciones desde el lado del servidor. Esta funcionalidad se implementa bajo la extensión CoAP Observe. Los clientes pueden establecer un parámetro en las peticiones para indicar al servidor que les gustaría recibir notificaciones siempre que el recurso se vea actualizado; el servidor enviará un mensaje sin necesidad de intervención del cliente a partir de ese momento.

Se identifican tres recursos dentro de la implementación CoAP. La opción de utilizar CoAP Observe simplifica notablemente el diseño cuando se compara con el caso de HTTP:

- **Property.** El verbo read se implementa sobre peticiones GET, mientras que el verbo write utiliza peticiones PUT. El verbo observe aprovecha CoAP Observe para proporcionar una solución directa

sin necesidad de soluciones como long-polling. La diferencia entre read y observe es básicamente la presencia del parámetro Observe dentro de la petición GET.

- **Action.** Los procedimientos de acción se pueden lanzar (verbo invoke) usando peticiones POST, lo que a su vez crea una nueva invocación dentro del servidor que recibe un identificador único. Los clientes pueden observar estas invocaciones a través de una petición GET (pasando el identificador como parámetro); posteriormente son notificados de los resultados en el momento en el que la invocación termina.
- **Event.** CoAP Observe se utiliza una vez más para implementar los verbos subscribe y unsubscribe. Los clientes pueden crear nuevas suscripciones observando el recurso con una petición GET; nuevos mensajes son enviados desde el servidor en cada una de las emisiones del evento. La suscripción puede destruirse simplemente dejando de observar el recurso.

T2.5: Diseño y desarrollo del Protocol Binding MQTT

El trabajo invertido en esta tarea durante la anualidad 2019 se centra en la implementación entre las acciones de alto nivel que pueden ser ejecutadas sobre una Thing y los mensajes intercambiados con el servidor, utilizando el MQTT Protocol Binding.

MQTT es un protocolo ligero transportado sobre TCP que está basado en el modelo de mensajería publish-subscribe. Esto quiere decir que no existen peticiones punto a punto (petición – respuesta) como en HTTP, si no que los nodos cliente se suscriben a un conjunto de tópicos para recibir todos los mensajes publicados en ese tópico por cualquier otro nodo. El broker es el nodo central que se encarga de gestionar las suscripciones y redistribuir los mensajes.

A continuación, se puede ver la lista de tópicos de la implementación MQTT. La parte denotada como <servient_id> representa el Servient ID obtenido a partir del nombre de la máquina; actuando como un espacio de nombres para evitar colisiones dentro del mismo broker.

Topic	Patrón
`Property request`	<servient_id>/property/requests/<thing_name>/<property_name>
`Property update`	<servient_id>/property/updates/<thing_name>/<property_name>
`Property write ACK`	<servient_id>/property/ack/<thing_name>/<property_name>
`Action invocation`	<servient_id>/action/invocation/<thing_name>/<action_name>
`Action result`	<servient_id>/action/result/<thing_name>/<action_name>
`Event emission`	<servient_id>/event/<thing_name>/<event_name>

Los verbos read y write se mapean a mensajes publicados en el tópico property request. Los clientes pueden recibir de manera opcional una confirmación de la actualización suscribiéndose al tópico property write ACK. Todas las actualizaciones de valores de propiedades son a su vez publicadas en el tópico

property update, sin tener en cuenta el origen de la actualización. Las operaciones read se traducen en forzar la publicación dentro del tópico property update.

Las acciones (verbo invoke) se gestionan con dos tópicos. Los clientes publican primero un mensaje en el tópico action invocation para iniciar la invocación; los resultados se publican posteriormente dentro del tópico action result. Todas las invocaciones llevan asociadas un identificador único para permitir que los clientes puedan distinguirlas.

Los eventos se gestionan de manera similar a las actualizaciones de propiedades. Todas las notificaciones de un evento en particular se publican en su tópico event emission sin intervención por parte del cliente, es decir, no hace falta establecer una suscripción previa. Esto simplifica notablemente la gestión del ciclo de vida de las suscripciones (el verbo unsubscribe no tiene interpretación en este caso).

Hito 3: Diseño y desarrollo de la implementación de referencia WoRMS Lite

El objetivo principal de este hito consiste en diseñar y desarrollar una versión de la implementación WoRMS, que sea apropiada para dispositivos con una capacidad de procesamiento limitada. Este hito se divide en 2 tareas cuya ejecución se ha iniciado a lo largo de la presente anualidad 2019 y que tienen prevista su finalización en la anualidad 2020. Por ello, en este informe se resumen los trabajos realizados durante la anualidad 2019 en dichas tareas.

- T3.1: Diseño y desarrollo del WoT Runtime reducido
- T3.2: Diseño y desarrollo del Protocol Binding seleccionado.

T3.1: Diseño y desarrollo del WoT Runtime reducido

Esta versión del WoT Runtime está pensada para su despliegue en microcontroladores con una capacidad de procesamiento limitada, lo que amplia enormemente el espectro de dispositivos IoT susceptibles de ser utilizados con WoRMS.

El trabajo invertido en esta tarea ha sido la implementación de una librería en MicroPython que permite la creación de un servidor WoT en uno de los dos protocolos implementados (HTTP y MQTT). Este servidor expone de manera asíncrona una serie de recursos definidos por una TD autogenerada en base a los mismos. La implementación se divide en varios paquetes o *Building Blocks* como se definen en las recomendaciones del W3C.

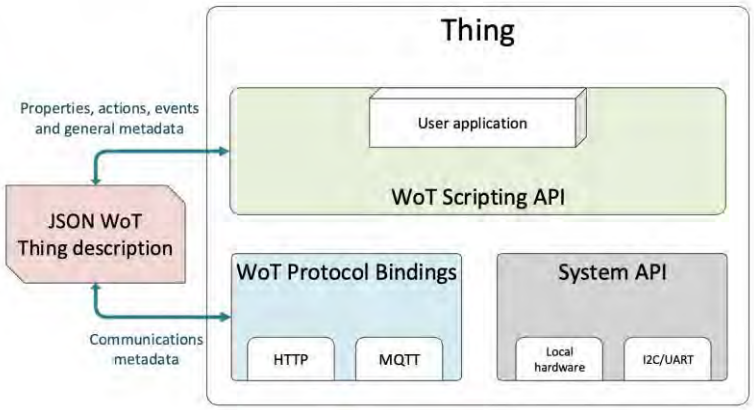


Figura 24. Arquitectura de los *Building Blocks*

Para esta implementación, se han desarrollado tres de estos *Building Blocks:* Thing Description, Scripting API y Protocolo Bindings. Además de la denominada "System API" que permite interactuar con el hardware local, sensores, actuadores, etc.

A continuación, se describen estos tres *Building Blocks* de manera general:

Thing Description.

La descripción asociada a una Thing es un modelo formal que representa una Thing Web, generalmente en formato JSON. Una TD describe los metadatos, las interfaces y los recursos que una Thing expone de manera inteligible tanto para una máquina como para un usuario. Las TD proveen una serie de interacciones basadas en un pequeño vocabulario que permite la integración con otras Things o aplicaciones de manera simple. Las interacciones se dividen en:

- Propiedades: Exponen valores internos de la Thing, permitiendo o no su modificación.
- Acciones: Permiten invocar acciones en la Thing, que pueden o no modificar su estado interno.
- Eventos. Permiten definer alertas o mensajes que se generan bajo ciertas condiciones. Este tipo de interacción se inicia desde la propia Thing.

Cada una de las interacciones definidas en estos tres tipos posee metadatos referentes a las comunicaciones denominados forms, estos contienen información tanto del tipo de protocolo que se utiliza para exponer la Thing, el tipo de respuesta que se espera, seguridad, etc.; además de una URI que indica la dirección a consultar para obtener los datos deseados.

Scripting API

WoT provee una capa de interoperabilidad basada en la forma en la que las Things se definen, pudiendo estas ser expuestas y/o consumidas. Para exponer o consumir una Thing, es necesaria una Thing Description y el software necesario para exponer sus recursos o accederlos. La implementación realizada en este *Building Block* permite servir tanto la TD como los recursos. Para ello ofrece una serie de métodos que permiten añadir, eliminar y crear funcionalidades para cada uno de los recursos.

En primer lugar, la clase *WoT* permite crear un objeto de la clase *ExposedThing* a partir de una TD previamente construida mediante el método *produce.*

Clase *WoT*	
Método	Descripción
`WoT.create_empty_thing(id, name, description, security)`	Crea una Thing con los parámetros aportados y retorna un objeto de la clase ExposedThing.
`WoT.produce(thingDescription)`	Genera una Thing en base a una TD que se le pase como parámetro.

Por otro lado, la clase *ExposedThing* ofrece métodos para la creación de propiedades, acciones y eventos, así como para asignar handlers o controladores a cada uno de ellos. Cada recurso sólo podrá tener un controlador, exceptuando las propiedades que pueden tener dos: uno para lectura y otro para escritura en caso de permitirse esta última, por lo que si se añaden varios controladores para un recurso sólo se mantendrá el último.

Clase *ExposedThing*	
Método	Descripción
`ExposedThing.expose(server)`	Expone mediante el protocolo definido en el parámetro de entrada todos los recursos previamente añadidos a la Thing. Este método llama a los *Protocol Bindings* para crear los distintos puntos de acceso. En caso de ser MQTT, habrá que pasar también la IP y puerto del bróker, así como los datos de la red WiFi como ***kwargs*.
`ExposedThing.add_property( name, description):`	Añade un recurso del tipo propiedad a la Thing.
`ExposedThing.remove_property(name):`	Elimina una propiedad de la Thing.
`ExposedThing.add_action( name, input_type, safe, idempotent, description)`	Añade un recurso del tipo acción a la Thing.
`ExposedThing.remove_action( name)`	Elimina una acción de la Thing.
`ExposedThing.add_event( name, description)`	Añade un recurso del tipo evento a la Thing.
`ExposedThing.remove_event( name)`	Elimina un evento de la Thing.
`ExposedThing.set_property_read_handler( property_name, read_handler)`	Añade un método controlador (*read_handler*) de lectura a una de las propiedades.
`ExposedThing.set_property_write_handler( property_name, write_handler)`	Añade un método controlador (*write_handler*) de escritura a una de las propiedades. Si se añade este controlador, la propiedad pasa a ser editable, y así se reflejará en la TD.
`ExposedThing.set_action_handler( action_name, action_handler)`	Añade un método controlador (*action_handler*) a una de las acciones.
`ExposedThing.set_event_handler( event_name, event_handler)`	Añade un método controlador (*event_handler*) a uno de los eventos.

`ExposedThing.generate_thing_description()`	Genera la TD de la Thing en su estado actual y la almacena en una variable interna de la misma.

Protocol Bindings

La implementación de los Protocol Bindings permite enlazar los distintos recursos con sus respectivos métodos de acceso a través de un protocolo determinado Una Thing sólo puede tener un protocolo asociado, y todos sus recursos se expondrán a través del mismo. Las implementaciones son completamente asíncronas, lo que permite responder a múltiples peticiones al mismo tiempo, así como mantener procesos ejecutándose en paralelo, lo cual es muy conveniente para la monitorización de eventos.

T3.2: Diseño y desarrollo del Protocol Binding seleccionado

El trabajo invertido en esta tarea ha sido la implementación, por una parte, del mapeo entre las acciones de alto nivel llevadas a cabo sobre una Thing, y los mensajes intercambiados con el servidor a través del Protocol Binding HTTP. Por otra parte, se implementa el mapeo entre las acciones de alto nivel que pueden ejecutarse sobre una Thing y los mensajes intercambiados con el bróker, cuando se utiliza MQTT Protocol Binding.

La implementación del protocolo HTTP se realiza a través de la librería Picoweb (P. Sokolovsky) que permite la creación de un servidor web asíncrono en MicroPython. Este se crea tras la invocación del método *expose* de la clase *ExposedThing* pasando el parámetro HTTP. Internamente se utiliza la Exposed Thing para recabar los distintos recursos a exponer, asignarles una URL, asociar los distintos handlers y añadirla al servidor, así como almacenarlas en la TD. Finalmente, se genera la TD por última vez con todas las URIs incluidas y se expone en el directorio raíz.

A continuación, se describe la manera en las que los verbos de interacción WoT se mapean a mensajes de bajo nivel a través del HTTP Protocol Binding.

Verbo	Tipo de interacción	Descripción
`Read`	Propiedad *(Property)*	Leer el valor de la propiedad.
`Write`	Propiedad *(Property)*	ACtualiza el valor de la propiedad.
`Invoke`	Acción (Action)	Invocar una acción.
`Observe`	Propiedad (Property)	Suscribirse a notificaciones de actualización de valores de propiedad.

La implementación del protocolo MQTT se realiza mediante la librería mqtt_as (P. Hinch) que implementa el protocolo MQTT de manera asíncrona. Este se crea tras la invocación del método *expose* de la clase *ExposedThing* pasando el parámetro "MQTT". La interface *ExposedThing* se utiliza para recabar los distintos recursos a exponer, asignarles sus respectivos tópicos, asociar los distintos *handlers* y añadirlos al servidor que escuchará los tópicos de entrada, así como almacenarlas en la TD. Finalmente, se genera

la TD por última vez con todas las URIs incluidas y se expone en el directorio raíz. En cuanto al bróker MQTT que se utilizará, queda a elección del usuario indicar cuál es su IP y puerto.

A continuación, se puede ver la lista de tópicos de la implementación MQTT. La parte denotada como <servient_id> representa el *Servient ID* obtenido a partir del nombre de la máquina; actuando como un espacio de nombres para evitar colisiones dentro del mismo bróker.

Tópico	Patrón
Property request	`<servient_id>/property/<property_name>`
Property update	`<servient_id>/property/<property_name>/response`
Action invocation	`<servient_id>/action/<action_name>`
Action result	`<servient_id>/action/<action_name>/response`
Event emission	`<servient_id>/event/<event_name>`

Justificación 2020

Durante el periodo que cubre este informe, se ha finalizado por completo la ejecución de las tareas correspondientes al Hito 3 y 4.

A continuación, se describen las tareas realizadas en cada uno de los hitos de trabajo mencionados anteriormente y los resultados conseguidos.

Hito 3: Diseño y desarrollo de la implementación de referencia WoRMS Lite

El objetivo principal de este hito consiste en diseñar y desarrollar una versión de la implementación WoRMS, que sea apropiada para dispositivos con una capacidad de procesamiento limitada. Este hito se divide en 2 tareas cuya ejecución se ha iniciado en la anualidad 2019 y se ha complementado a lo largo de la presente anualidad 2020. Por ello, en este informe se resumen los trabajos realizados durante la anualidad 2020 en dichas tareas.

- T3.1: Diseño y desarrollo del WoT Runtime reducido.
- T3.2: Diseño y desarrollo del Protocol Binding seleccionado.

T3.1: Diseño y desarrollo del WoT Runtime reducido

Esta versión del WoT Runtime está pensada para su despliegue en microcontroladores con una capacidad de procesamiento limitada, lo que amplia enormemente el espectro de dispositivos IoT susceptibles de ser utilizados con WoRMS. A continuación, se resume el desarrollo implementado sobre esta versión de WoT en la anualidad 2020, desde dos perspectivas: a nivel general y a nivel específico.

A nivel general:

La implementación de esta versión WoT Runtime, finalizada en esta anualidad 2020, proporciona:

- Un W3C WoT Runtime que contiene la funcionalidad base para consumir y exponer Things en la red.
- Ofrece un API según las especificaciones de la W3C WoT Scripting API que estandariza la interfaz utilizada por los desarrolladores para utilizar el Runtime.
- Proporciona un conjunto de W3C WoT Protocol Bindings que permiten traducir entre las interacciones de alto nivel en una Thing y los mensajes de bajo nivel que se transmiten en la red.
- Soporta MicroPython, una implementación del lenguaje de programación Python 3 que incluye una pequeña parte de sus funcionalidades, permitiendo su ejecución en entornos de reducidas capacidades, como los microcontroladores.
- Modelo de programación asíncrono basado en corutinas. Estas últimas permiten exponer recursos web, leer y escribir sensores y generar acciones y eventos de manera simultánea.

Dado que esta implementación está desarrollada en MicroPython, el resultado es una serie de librerías necesarias para su funcionamiento (picoweb, uasyncio, uloggin y uMQTTas) junto a la propia librería de WoRMS Lite. Estas librerías adicionales se encargan de los servicios de Protocol binding de manera asíncrona, así como de los logs.

A nivel específico:

La implementación se divide en varios paquetes o *"Building Blocks" (BBs)* como se definen en las recomendaciones del W3C.

Durante la anualidad 2020 se continúa con la implementación y refinamiento de tres de estos BBs:

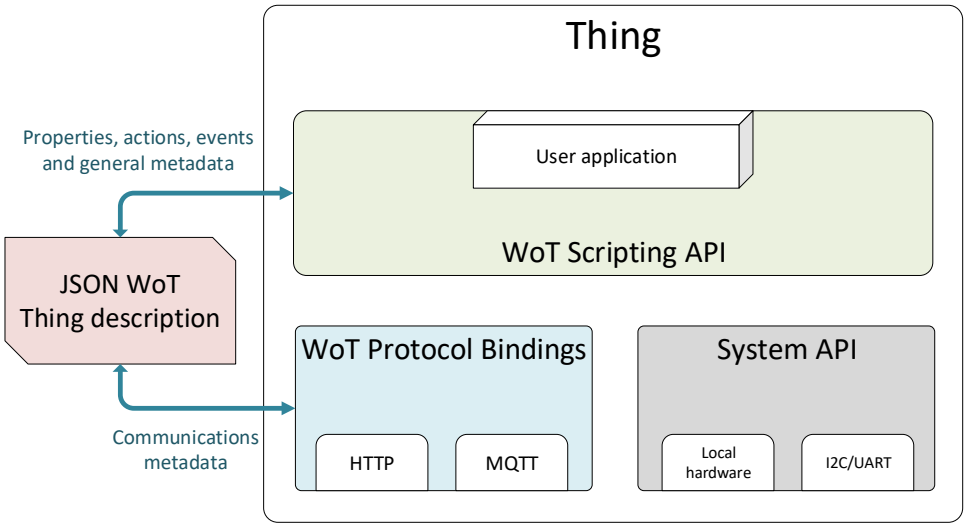


Ilustración 1. Arquitectura de los *Building Blocks*

- ***Thing Description (TD).***

Como modelo formal que describe los metadatos, las interfaces y los recursos que una Thing expone de manera inteligible tanto para una máquina como para un usuario.
Las TD proveen de una serie de interacciones basadas en un pequeño vocabulario que permite la integración con otras Things o aplicaciones de manera simple.
El espacio de nombres *uWoT.Resources* contiene funciones y clases de utilidad para la manipulación de Things.

- ***Scripting API***
  La implementación llevada a cabo sobre esta interfaz permite encapsular el funcionamiento interno del WoT Runtime. También proporciona un objeto WoT que permite crear un objeto de la clase *ExposedThing* a partir de una TD previamente construida mediante el método *produce*.

  La clase *ExposedThing* ofrece métodos para la creación de propiedades, acciones y eventos, así como para asignar *handlers* o controladores a cada uno de ellos.

- ***Protocol Bindings.***
  Esta implementación permite enlazar los distintos recursos con sus métodos de acceso a través de un protocolo determinado, en este caso: HTTP y MQTT. La Thing sólo puede tener un protocolo asociado, y todos sus recursos se expondrán a través del mismo. Las implementaciones son asíncronas, lo que permite responder a múltiples peticiones al mismo tiempo, así como mantener procesos ejecutándose en paralelo. Esto permite monitorizar eventos de manera efectiva.

T3.2: Diseño y desarrollo del Protocol Binding seleccionado

El trabajo invertido durante la anualidad 2020 se centra en la implementación y refinamiento en la que los verbos de interacción se mapean a mensajes de bajo nivel en los diferentes protocolos de capa de aplicación soportados.

Verbo	Tipo de interacción	Descripción
`Read`	`Propiedad (Property)`	*Lee el valor de la propiedad*
`Write`	Propiedad (Property)	*Actualizar el valor de la propiedad*
`Invoke`	Acción (Action)	*Invocar una acción*
`Observe`	Propiedad (Property)	*Suscribirse a notificaciones de actualización de valores de propiedad*

- **Protocol Binding HTTP**
  El trabajo invertido en esta tarea durante la anualidad 2019 se centra en la implementación entre las acciones de alto nivel que pueden ser ejecutadas sobre una Thing y los mensajes intercambiados con el servidor, utilizando el Protocol Binding HTTP.

La implementación de este protocolo se apoya en la librería Picoweb**.** Esta librería permite crear un servidor Web asíncrono en MicroPython a través del método *expose* de la clase *ExposedThing.* Esta clase se utiliza internamente para recabar los distintos recursos a exponer, asignarles una URL y asociar los manejadores para añadirla al servidor. Finalmente se genera la TD con todas las URIs incluidas, y se expone sobre el directorio raíz.

En la implementación se identifican los siguientes recursos HTTP. Los verbos de interacción se mapean a métodos HTTP de acuerdo con los requisitos del patrón REST *(uniform interface constraint).*

- **Property.** Representa el valor de una propiedad. Se utilizan peticiones GET para leer el valor *(read),* mientras que se usan peticiones PUT para actualizarlo *(write).*
  Ejemplo de solicitud de lectura:
  GET *http://<host>:<port>/<thing_name>/<property_name>*

  Ejemplo de solicitud de escritura:
  PUT *http://<host>:<port>/<thing_name>/<property_name>*

- **Action invocation.** Representa procedimientos que pueden ser invocados *(invoke)* usando peticiones POST.
  Ejemplo:
  POST http://<host>:<port>/<thing_name>/<action_name>

- **Event Subscription.** Representa una suscripción a notificaciones de un evento particular *(observe)*. El cliente lanza una petición GET que permanece abierta hasta que el servidor dispone de una notificación.
  Ejemplo de suscripción:
  GET http://<host>:<port>/<thing_name>/<action_name>

A continuación, se adjunta un extracto de código que representa la clase donde se implementa el protocolo HTTP. En la mayoría de los métodos se omite su implementación por ser de excesiva longitud.

uWoT/Protocols/HTTP_Bind.py
Extracto de clase

```
import ujson, network, os, sys
import picoweb

class HTTPBind(object):
    def __init__(self):
        self.app = picoweb.WebApp(__name__)
        self.response_pattern = {"value": None}
```

```
    async def read_req_body (self, req):
        """
        Receives a request and parses its body
        :param req: an HTTP request
        :return: The body of the request
        """
        ...

    def bind(self, exposed_thing):
        """
        recives an ExposedThing object, reads its properties and exposes them in
an HTTP server
        :param exposed_thing: e ExposedThing object
        :return: Nothing
        """
        ...

        #Expose the TD in the root directory:
        @self.app.route("/")
        def index(req, resp):
            yield from picoweb.start_response(resp,
content_type="application/json")
            yield from resp.awrite(exposed_thing.thingDescription)

        print("Starting HTTP server in {}, port {}".format(self.current_IP, 80))
        self.app.run(host=self.current_IP, port=80, debug=-1)
```

- **Protocol Binding MQTT**

  El trabajo invertido en esta tarea durante la anualidad 2019 se centra en la implementación entre las acciones de alto nivel que pueden ser ejecutadas sobre una Thing y los mensajes intercambiados con el servidor, utilizando el MQTT Protocol Binding.

  MQTT es un protocolo ligero transportado sobre TCP que está basado en el modelo de mensajería *publish-subscribe.* Esto significa que no existen peticiones punto a punto (petición – respuesta) como en HTTP, si no que los nodos cliente se suscriben a un conjunto de tópicos para recibir todos los mensajes publicados en ese tópico por cualquier otro nodo. El bróker es el nodo centran que se encarga de gestionar las suscripciones y redistribuir los mensajes.

  La implementación del protocolo MQTT sobre la versión WoRMS Lite, se apoya en la librería *mqtt_as* que implementa el protocolo de manera asíncrona. Este se crea tras la invocación del método *expose* de la clase *ExposedThing* pasando el parámetro “MQTT”. Los tópicos o

URIs MQTT asociados a cada propiedad serán el doble que en el protocolo HTTP, ya que es necesario un tópico para recibir las peticiones de entrada, y otro para las respuestas.
La *ExposedThing* se utiliza para recabar los distintos recursos a exponer, asignarles sus respectivos tópicos, asociar los distintos handlers y añadirlos al servidor que escuchará los tópicos de entrada. Finalmente se almacenan en la TD y esta se genera con todas las URIs incluidas para exponerla en el directorio raíz. En cuanto al bróker MQTT que se utilizará, queda a elección del usuario indicar cuál es su IP y puerto.

A continuación, se puede ver la lista de tópicos de la implementación MQTT. La parte denotada como *<servient_id>* representa el *Servient ID* obtenido a partir del nombre de la máquina; actuando como un espacio de nombres para evitar colisiones dentro del mismo broker.

Tópico	**Patrón**
Property request	`<servient_id>/property/<property_name>`
Property update	`<servient_id>/property/<property_name>/response`
Action invocation	`<servient_id>/action/<action_name>`
Action result	`<servient_id>/action/<action_name>/response`
Event emission	`<servient_id>/event/<event_name>`

En la implementación se identifican los siguientes recursos HTTP. Los verbos de interacción se mapean a métodos HTTP de acuerdo con los requisitos del patrón REST *(uniform interface constraint).*

- **Property.** Representa el valor de una propiedad.
  Los verbos *read* y *write* se mapean a mensajes publicados en el tópico *property request.* Las actualizaciones de valores de propiedades son a su vez publicadas en el tópico *property update,* sin tener en cuenta el origen de la actualización. Las operaciones *read* se traducen en forzar la publicación dentro del tópico *property update.*
  Ejemplo:
  ```
  {
    “value”: <property_value>
  }
  ```

- **Action invocation.** Representa acciones que pueden ser invocadas *(invoke).* Estas acciones se gestionan con dos tópicos: los clientes publican primero un mensaje en el tópico *action invocation* para iniciar la invocación; los resultados se publican posteriormente dentro del tópico *action result.* Todas las invocaciones llevan asociadas un identificador único para permitir que los clientes puedan distinguirlas.
  Ejemplo:
  ```
  {
    “value”: <new_value>
  ```

```
}
```

- **Event Subscription.** Representa una suscripción a notificaciones de un evento particular *(observe)*. Los eventos se gestionan de manera similar a las actualizaciones de propiedades. Todas las notificaciones de un evento en particular se publican en su tópico *event emission* sin intervención por parte del cliente, es decir, no hace falta establecer una suscripción previa. Esto simplifica notablemente la gestión del ciclo de vida de las suscripciones.
  Ejemplo:
  ```
  {
    "value": <True/False>
  }
  ```

A continuación, se adjunta un extracto de código que representan la clase donde se implementa el protocolo HTTP. En la mayoría de los métodos se omite su implementación por ser de excesiva longitud.

uWoT/Protocols/MQTTP_Bind.py
Extracto de clase

```
import ujson, network, os, sys
import picoweb

class HTTPBind(object):
    def __init__(self):
        self.app = picoweb.WebApp(__name__)
        self.response_pattern = {"value": None}

    async def read_req_body (self, req):
        """
        Receives a request and parses its body
        :param req: an HTTP request
        :return: The body of the request
        """
        ...

    def bind(self, exposed_thing):
        """
        recives an ExposedThing object, reads its properties and exposes them in
an HTTP server
        :param exposed_thing: e ExposedThing object
        :return: Nothing
        """
        ...
```

```
        #Expose the TD in the root directory:
        @self.app.route("/")
        def index(req, resp):
            yield from picoweb.start_response(resp,
content_type="application/json")
            yield from resp.awrite(exposed_thing.thingDescription)

        print("Starting HTTP server in {}, port {}".format(self.current_IP, 80))
        self.app.run(host=self.current_IP, port=80, debug=-1)
```

Hito 4: Despliegue de la aplicación WoT demostradora y validación de la implementación de referencia

El objetivo principal de este hito consiste en diseñar y desarrollar una versión de la implementación WoRMS, que sea apropiada para dispositivos con una capacidad de procesamiento limitada. Este hito se divide en 2 tareas cuya ejecución se ha iniciado en la anualidad 2019 y que se ha finalizado en la anualidad 2020. Por ello, en este informe se resumen los trabajos realizados durante la anualidad 2020 en dichas tareas.

- T4.1: Diseño de detalle de la aplicación WoT demostradora
- T4.2: Implementación de la aplicación WoT demostradora
- T4.3: Despliegue de la aplicación WoT demostradora y validación

T4.1: Diseño de detalle de la aplicación WoT demostradora

En esta tarea se ha llevado a cabo el diseño de un despliegue cuyo objetivo ha sido validar y demostrar la funcionalidad de los componentes desarrollados hasta el momento. Este diseño incluye:

- **Escenario de validación**. Donde se modelan situaciones reales de funcionamiento en un entorno controlado, para garantizar así el correcto funcionamiento de los componentes desarrollados.
- **Diseño conceptual**. A través de él se presentarán los actores y dispositivos involucrados, así como los flujos de comunicación entre los mismos.
- **Funcionalidad principal**. Servirá para definir las características y funcionalidad de las aplicaciones requeridas en cada dispositivo del escenario de validación.
- **Requisitos funcionales y no funcionales**. Que permitan validar el correcto funcionamiento del sistema.

Esta tarea se desarrolla sobre dos versiones de la implementación de referencia con el fin de abarcar el mayor número de dispositivo IoT posibles: una versión completa destinada a dispositivos y sistemas con capacidades de computación elevadas (WoRMS) y otra versión restringida, orientada a dispositivos y sistemas con capacidades de computación muy reducidas (WoRMS Lite).

Se explica a continuación, el despliegue en cada una de las versiones:

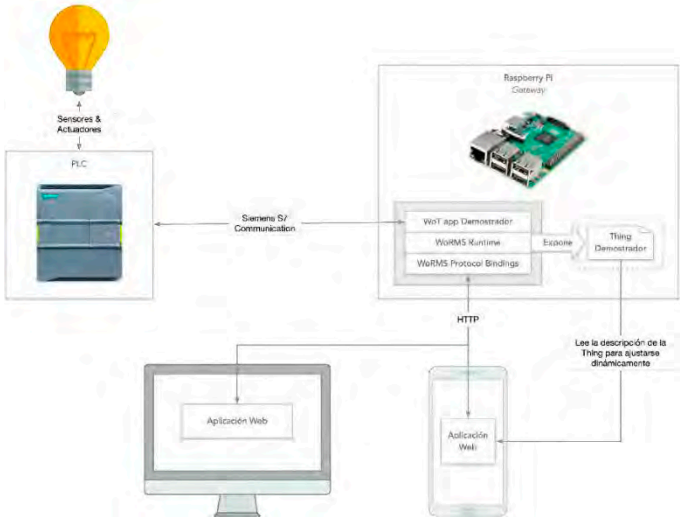
- **WoRMS - Diseño de detalle de la aplicación WoT demostradora**
  - **Escenario de validación**

    El escenario de validación tiene como objetivo el acceso de lectura y escritura a una serie de sensores dispuestos en una planta industrial, cuya conexión está centralizada en un PLC. Este escenario simularía una situación de emergencia en el que se produce una fuga de gas en una planta industrial. La situación de emergencia, activada a voluntad de manera remota, dispara un protocolo de emergencia en el que se activarán de manera secuencial una alarma y un ventilador. Alternativamente, se incluyen sensores de presencia asociados a dos luces que determinan si un puente grúa puede realizar movimientos.

    El acceso de lectura y escritura a cada una de las variables controladoras definidas en el PLC debe realizarse utilizando una librería específica en cada cliente. Sobre este escenario se introduce un Gateway que implementa el entorno de ejecución de WoRMS, de esta forma, es posible ofrecer una interfaz web estandarizada que puede ser utilizada por múltiples clientes remotos.

  - **Diseño conceptual**

    A través del siguiente diagrama se representa el diseño conceptual de la aplicación demostradora, exponiendo los diferentes dispositivos físicos que la conforman:



Sobre el diseño anterior se observan dos componentes principales, que simbolizan dispositivos físicos remotos:

- **PLC**. Implementa la lógica de situación de emergencia en la planta industrial.

- **Gateway**. Implementa la interfaz Web of Things y la comunicación directa con el PLC, utilizando el runtime de WoRMS. Adicionalmente, cuenta con un servidor web que sirve una aplicación web que permitirá a cualquier usuario interactuar con la interfaz WoT desde cualquier navegador.

El principal actor involucrado será el usuario, que podrá monitorizar desde su dispositivo el estado de los sensores de la planta industrial, además de poder disparar la situación de emergencia de manera remota. El usuario interactúa con la interfaz WoT mediante una aplicación Web adicional, que interpretará la Thing Description del dispositivo modelado y realizará peticiones HTTP a cada una de sus propiedades de manera periódica.

De esta manera, la aplicación Web no requiere de ningún conocimiento sobre las particularidades de cada dispositivo, pudiendo automatizarse la consulta de cada uno de los sensores a partir de la Thing Description.

- **Funcionalidad principal**

A continuación, se identifica la funcionalidad requerida en cada una de las aplicaciones definidas para cada uno de los dispositivos físicos remotos:

- **PLC (de la marca Siemens):** El protocolo a implementar en el PLC será el siguiente:
1) La situación de emergencia es activada, bien de manera manual o mediante un pulsador o remotamente.
2) Se enciende la luz de emergencia durante 5 segundos.
3) Se enciende un ventilador durante 5 segundos.
4) Se para el ventilador.
Alternativamente, debe de existir un sensor de presencia asociada a un semáforo que determine si el puente grúa puede moverse.

- **Gateway:** El Gateway es el componente principal, puesto que implementa la interfaz que facilita la interacción con el dispositivo final. Implementará lógica de comunicación específica con el PLC y ofrecerá una interfaz HTTP basada en los *protocol bindings* desarrollados a lo largo del proyecto WoRMS.
El Gateway proporcionará acceso a la descripción del dispositivo a través de peticiones HTTP y expondrá *endpoints* dedicados para la lectura y escritura de cada una de las variables del dispositivo presentes en la descripción del mismo.

- **Interfaz Web:** Se proporcionará una interfaz gráfica mediante una aplicación que monitorizará en tiempo real el estado de la planta industrial, realizando peticiones HTTP

al Gateway WoT. También permitirá activar el protocolo de emergencia de manera remota.

- **Requisitos funcionales y no funcionales**

  Se exponen a continuación el listado de requisitos definidos para el escenario de validación:

Requisitos funcionales		
ID	**Descripción**	**Prioridad**
RF-1	La *Thing Description* del dispositivo es accesible de manera remota	Alta
RF-2	Es posible activar el protocolo de emergencia desde la aplicación Web a través de la interfaz *WoT*	Alta
RF-3	Todas las variables definidas en la *TD* del dispositivo son accesibles a través de la interfaz Web	Alta
RF-4	La aplicación Web representa el estado actual de la planta en todo momento	Alta
RF-5	Todas las interacciones se realizan mediante peticiones HTTP	Alta

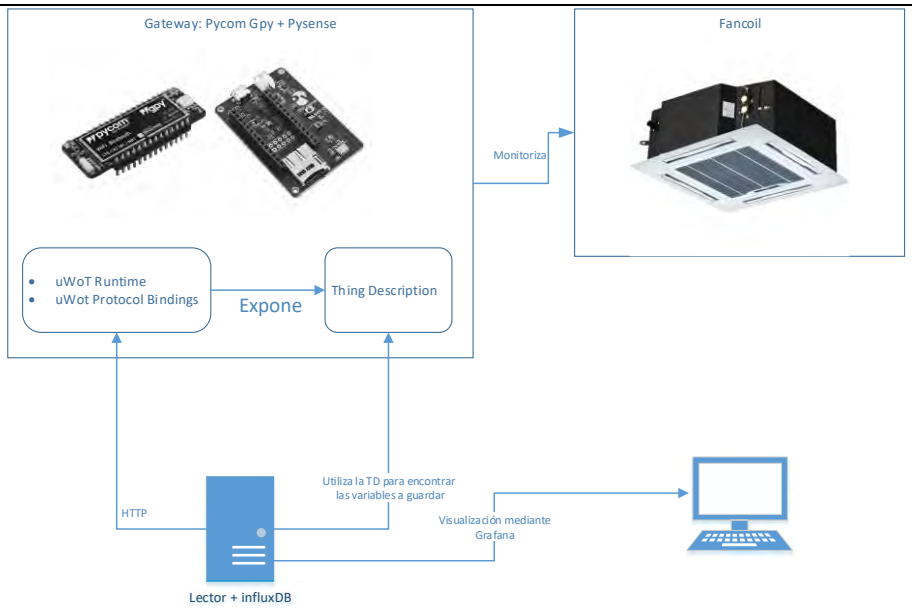
Requisitos no funcionales		
ID	**Descripción**	**Prioridad**
RF-6	La aplicación Web debe reflejar el estado de cada una de las variables en tiempo real.	Alta
RF-7	La aplicación Web debe presentar un diseño adaptativo.	Alta

- **WoRMS Lite – Diseño de detalle de la aplicación WoT demostradora**
  - **Escenario de validación**

    El escenario de validación tiene como objetivo el acceso de lectura a una serie de sensores dispuestos en una serie de fancoils dentro de un edificio de oficinas. El escenario monitoriza el estado, consumo y gasto energético de los fancoils. Los sensores de temperatura monitorizan la temperatura del agua y aire que circula por el sistema, también se monitoriza el consumo eléctrico, el flujo de agua (caliente y fría) y se calcula el gasto energético.

  - **Diseño conceptual**

    A través del siguiente diagrama se representa el diseño conceptual de la aplicación demostradora, exponiendo los diferentes dispositivos físicos que la conforman:



El acceso de lectura de las variables se realiza mediante la instalación y ejecución de la librería *uWoT* en una placa *Pycom Gpy,* mientras que los componentes hardware se distribuyen de la siguiente manera:

- **Fancoil:** Dispositivo a monitorizar.

- **Gateway:** Pycom Gpy y Pysense. Implementa la interfaz *WoT* y la monitorización directa con el Fancoil, utilizando el runtime de *uWoT.*

- **Servidor:** Implementa una base de datos temporal en la que un script almacena cada minuto los datos leídos de la interfaz *WoT,* así como un cuadro de mando implementado en Grafana donde los datos pueden ser visualizados.

El principal actor involucrado será el usuario, que podrá monitorizar desde su dispositivo el estado de los fancoils.

El responsable de acceder de manera directa a la interfaz *WoT* será el script de lectura, que realizará peticiones HTTP basadas en la *TD,* a todas las propiedades cada minuto, y después las almacenará en la Influx DB.

- **Funcionalidad principal**

A continuación, se identifica la funcionalidad requerida en cada una de las aplicaciones definidas para cada uno de los dispositivos físicos remotos.

- **Fancoil:** Lleva integrados los diferentes sensores que aportan medidas al sistema.

- **Gateway:** Representa el componente principal puesto que implementa la interfaz que facilita la interacción con el dispositivo final. Implementará lógica de comunicación

específica con los sensores del fancoil y ofrecerá una interfaz HTTP basada en los *Protocol Bindings* desarrollados a lo largo del proyecto WoRMS Lite.
El Gateway proporcionará acceso a la descripción del dispositivo a través de peticiones HTTP y expondrá endpoints dedicados para la lectura y/o escritura de cada una de las variables del dispositivo presentes en la descripción del mismo.

- **Interfaz Web (Grafana):** Se propocionará una interfaz gráfica mediante una aplicación Web que monitorizará en tiempo real el estado de los fancoils, realizando peticiones HTTP al Gateway WoT.

- **Requisitos funcionales y no funcionales**
  Se exponen a continuación el listado de requisitos definidos para el escenario de validación:

Requisitos funcionales		
ID	**Descripción**	**Prioridad**
RF-1	La *Thing Description* del dispositivo es accesible de manera remota	Alta
RF-2	Es posible consultar los datos de los sensores desde la interfaz Web	Alta
RF-3	Todas las variables definidas en la *TD* del dispositivo son accesibles a través de la interfaz Web	Alta
RF-4	La aplicación Web representa el estado actual de los fancoils en todo momento	Alta

Requisitos no funcionales		
ID	**Descripción**	**Prioridad**
RF-6	La aplicación Web debe reflejar el estado de cada una de las variables en tiempo real.	Alta
RF-7	La aplicación Web debe presentar un diseño adaptativo.	Alta

T4.2: Implementación de la aplicación WoT demostradora

Esta tarea se desarrolla sobre dos versiones de la implementación de referencia con el fin de abarcar el mayor número de dispositivo IoT posibles: una versión completa destinada a dispositivos y sistemas con capacidades de computación elevadas (WoRMS) y otra versión restringida, orientada a dispositivos y sistemas con capacidades de computación muy reducidas (WoRMS Lite).

Como resultado de la ejecución de esta tarea, se han generado dos entregables que se adjuntan en este expediente de justificación: por una parte, el software de la aplicación demostradora de WoRMS y, por otra, el software de la aplicación demostradora de WoRMS Lite.

A continuación, se adjuntan unos extractos de código que representan las entidades más representativas de cada una de las versiones de la implementación. En la mayoría de los métodos se omite su implementación por ser de excesiva longitud.

- **WoRMS – Extractos de la implementación de la aplicación demostradora**

Extracto de opc-wot/opc/server.py

```
"""
WoT application to expose a Thing that provides current host CPU usage levels.
"""

import asyncio
import json
import logging
import os
from asyncua import Client, Node, ua
import pymysql.cursors
import time
import datetime


from wotpy.protocols.http.server import HTTPServer
from wotpy.protocols.ws.server import WebsocketServer
from wotpy.wot.servient import Servient

logging.basicConfig()
LOGGER = logging.getLogger("opcmonitor")
LOGGER.setLevel(logging.INFO)

PORT_CATALOGUE = int(os.environ.get("PORT_CATALOGUE", 9090))
PORT_WS = int(os.environ.get("PORT_WS", 9191))
PORT_HTTP = int(os.environ.get("PORT_HTTP", 9292))
DEFAULT_CPU_CHECK_SEC = float(os.environ.get("CPU_CHECK_SEC", 2.0))


OPC_SERVER_URL = os.environ.get(
    "OPC_SERVER_URL", "opc.tcp://192.168.0.102:49320")

OPC_SERVER_URL = 'opc.tcp://0.0.0.0:4840/freeopcua/server/'


DESCRIPTION = {
    "id": "urn:org:fundacionctic:thing:opcwot",
    "name": "OPC Monitor Thing",
    "properties": {
            "inicio": {
                "description": "Emergency situation trigger",
                "type": "number",
                "observable": True
            },
        "luz-roja": {
```

```
                "description": "Red light state",
                "type": "number",
                "observable": True
        },
        "luz-azul": {
                "description": "Blue light state",
                "type": "number",
                "observable": True
        },
        "luz-emergencia": {
                "description": "Emergency light state",
                "type": "number",
                "observable": True
        },
        "ventilador": {
                "description": "Fan state",
                "type": "number",
                "observable": True
        }
    }
}

OPC_VARIABLES = {
    "inicio":           "ns=2;s=Channel1.S7-1200.Inicio",
    "luz-roja":         "ns=2;s=Channel1.S7-1200.Luz_roja",
    "luz-azul":         "ns=2;s=Channel1.S7-1200.Luz_azul",
    "luz-emergencia":   "ns=2;s=Channel1.S7-1200.Luz_emergencia",
    "ventilador":       "ns=2;s=Channel1.S7-1200.Ventilador"
}

OPC_VARIABLES = {
    "inicio": "ns=2;i=2",
    "luz-roja": "ns=2;i=3",
    "luz-azul": "ns=2;i=4",
    "luz-emergencia": "ns=2;i=5",
    "ventilador": "ns=2;i=6"
}


opc_values = {}
connection = pymysql.connect(host='localhost',
                             user='root',
                             password='******',
                             db='showroom_opc',
                             charset='utf8mb4',
                             cursorclass=pymysql.cursors.DictCursor)
```

```
def build_opc_read_handler(nodeId):
    """Factory for OPC Property read handlers."""

def build_opc_write_handler(nodeId):
    """Factory for OPC Property write handlers."""

def create_opc_monitor_task(exposed_thing):
    global opc_values

    async def opc_monitor_loop():

        while True:
            try:
                data = []
                ts = time.time()
                timestamp = datetime.datetime.fromtimestamp(
                    ts).strftime('%Y-%m-%d %H:%M:%S')
                async with Client(url=OPC_SERVER_URL) as client:
                    for key, value in OPC_VARIABLES.items():
                        var = client.get_node(value)
                        opc_values[value] = int(await var.get_value())
                        data.append((timestamp, key, opc_values[value]))
                insert_into_mysql(data)
            except Exception:
                LOGGER.exception('error')

            await asyncio.sleep(DEFAULT_CPU_CHECK_SEC)

    event_loop = asyncio.get_event_loop()
    event_loop.create_task(opc_monitor_loop())


def insert_into_mysql(values):
    global connection
    print(values)
    try:
        with connection.cursor() as cursor:
            sql = "INSERT INTO `variable_values` (`time`, `variable`, `value`) VALUES
(%s, %s, %s)"
            cursor.executemany(sql, values)
        connection.commit()
    except Exception:
        LOGGER.exception('error')
```

```python
async def main():
    """Main entrypoint."""

    LOGGER.info("Creating WebSocket server on: {}".format(PORT_WS))
    ws_server = WebsocketServer(port=PORT_WS)

    LOGGER.info("Creating HTTP server on: {}".format(PORT_HTTP))
    http_server = HTTPServer(port=PORT_HTTP)

    LOGGER.info(
        "Creating servient with TD catalogue on: {}".format(PORT_CATALOGUE))

    servient = Servient(catalogue_port=PORT_CATALOGUE)
    servient.add_server(ws_server)
    servient.add_server(http_server)

    LOGGER.info("Starting servient")

    wot = await servient.start()

    LOGGER.info("Exposing System Monitor Thing")

    exposed_thing = wot.produce(json.dumps(DESCRIPTION))

    for key, value in OPC_VARIABLES.items():
        exposed_thing.set_property_read_handler(
            key, build_opc_read_handler(value))

    exposed_thing.set_property_write_handler(
        "inicio", build_opc_write_handler(OPC_VARIABLES["inicio"]))

    create_opc_monitor_task(exposed_thing)

    exposed_thing.expose()


if __name__ == "__main__":
    LOGGER.info("Starting loop")
    loop = asyncio.get_event_loop()
    loop.create_task(main())
    loop.run_forever()
```

Extracto de opc-client/src/components/ScadaMain.vue

```
<script>
import axios from "axios";
import "vue-awesome/icons/circle";
import "vue-awesome/icons/fan";
import "vue-awesome/icons/people-carry";

import Icon from "vue-awesome/components/Icon";

export default {
  components: {
    "v-icon": Icon
  },
  data: () => ({
    operatorColor: "grey",
    lightbulbColor: "grey",
    fanColor: "grey",
    lightOn: false,
    operatorPresent: false,
    lightVarKey: "LUZ EMERGENCIA",
    headers: [
      { text: "Variable OPC", value: "opc_var" },
      { text: "Valor", value: "opc_val" }
    ],
    opcVariables: [
      {
        opc_var: "INICIO",
        opc_val: 0.0
      },
      {
        opc_var: "LUZ EMERGENCIA",
        opc_val: 0.0
      },
      {
        opc_var: "VENTILADOR",
        opc_val: 0.0
      },
      {
        opc_var: "ALARMA PUENTE GRUA",
        opc_val: 0.0
      },
      {
        opc_var: "PASO PUENTE GRUA",
        opc_val: 0.0
      }
    ]
```

```
  }),
  computed: {
    activeEmergency: function() {
      return false;
      return (
        this.getVarValue("LUZ EMERGENCIA").opc_val !== 1 ||
        this.getVarValue("VENTILADOR").opc_val !== 1
      );
    }
  },
  created() {},
  methods: {
    activateEmergency() {
      axios
        .put("http://192.168.0.103:3001/api/emergency", '{ "value": "30" }', {
          headers: {
            "Content-Type": "application/json"
          }
        })
        .then(function(response) {})
        .catch(function(error) {
          console.log(error);
        });
    },
    changeButtonColor() {
      this.buttonColor = this.buttonColor === "grey" ? "red" : "grey";
    },
    changeLightbulbColor() {
      if (this.lightOn) {
        this.lightbulbColor = this.lightbulbColor === "grey" ? "red" : "grey";
      }
    },
    changeOperatorColor() {
      if (this.operatorPresent) {
        this.operatorColor = this.operatorColor === "grey" ? "red" : "grey";
      }
    },
    changeVarValue(varId, varValue) {
      for (var i in this.opcVariables) {
        if (this.opcVariables[i].opc_var == varId) {
          this.opcVariables[i].opc_val = varValue;
          break;
        }
      }
    },
    getVarValue(varId) {
```

```
      return this.opcVariables.find(element => element.opc_var === varId);
    },
    retrieveOPCValues() {
      let that = this;
      axios
        .all([
          axios.get("http://192.168.0.103:3001/api/emergency"),
          axios.get("http://192.168.0.103:3001/api/light"),
          axios.get("http://192.168.0.103:3001/api/fan"),
          axios.get("http://192.168.0.103:3001/api/redlight"),
          axios.get("http://192.168.0.103:3001/api/greenlight")
        ])
        .then(
          axios.spread(
            (inicioRes, luzRes, ventiladorRes, redlightRes, greenlightRes) => {
              that.changeVarValue("INICIO", inicioRes.data.value);
              that.changeVarValue("LUZ EMERGENCIA", luzRes.data.value);
              that.changeVarValue("VENTILADOR", ventiladorRes.data.value);
              that.changeVarValue("ALARMA PUENTE GRUA", redlightRes.data.value);
              that.changeVarValue("PASO PUENTE GRUA", greenlightRes.data.value);

              if (luzRes.data.value === 0) {
                that.lightOn = false;
                that.lightbulbColor = "grey";
              } else {
                that.lightOn = true;
              }

              if (redlightRes.data.value === 0) {
                that.operatorPresent = false;
                that.operatorColor = "grey";
              } else {
                that.operatorPresent = true;
              }

              that.fanColor = ventiladorRes.data.value === 0 ? "grey" : "blue";
            }
          )
        );
    }
  },
  mounted() {
    let that = this;
    setInterval(function() {
      that.retrieveOPCValues();
    }, 1000);
```

```
    setInterval(function() {
      that.changeLightbulbColor();
      that.changeOperatorColor();
    }, 1000);
  }
};
</script>
```

- **WoRMS Lite – Extractos de la implementación de la aplicación demostradora**

Extracto de resource_definitions/properties.py

```
def property_declaration(exposed_thing):
    exposed_thing.add_property(
        name="temperature-air-in",
        description="An AM2315 Temperature sensor, returns the current temperature of
the air comming in the fan coil in Celsius.",
        type="number")

    exposed_thing.add_property(
        name="temperature-air-out",
        description="An AM2315 Temperature sensor, returns the current temperature of
the air comming out of the fan coil in Celsius.",
        type="number")

    exposed_thing.add_property(
        name="temperature-cold-water-in",
        description="A DS18b20 temperature sensor measuring temperature of water
coming into the system.",
        type="number")

    exposed_thing.add_property(
        name="temperature-cold-water-out",
        description="A DS18b20 temperature sensor measuring temperature of water
coming out of the system.",
        type="number")
...
```

Extracto de main.py

```
...
```

```
class main(object):
    def __init__(self, logger, broker, ssid, ssid_password, id, name, context, type,
description, server="HTTP"):
        self.logger = logger
        self.calibrating = False
        self.Accelerometer = None
        self.Pysense_Humidity =  None

        try:
            self.Air_In_Probe = DS18X20(OneWire(machine.Pin('P10')))
        except Exception as ex:
            self.logger.error(ex)

        try:
            self.Air_Out_Probe = DS18X20(OneWire(machine.Pin('P11')))
        except Exception as ex:
            self.logger.error(ex)

        try:
            self.Cold_Water_Out = DS18X20(OneWire(machine.Pin('P2')))
        except Exception as ex:
            self.logger.error(ex)
...



        self.mywot = WoT()
        self.mything = self.mywot.create_empty_thing(id=id, name=name,
context=context, type=type, description=description)

        ###### STORED VALUES ######
        self.Temperature_Air_In = 0
        self.Temperature_Air_Out = 0
        self.Temperature_Air_Facade = 0
...

        self.restart = False

        ###### PROPERTY DEFINITION ######
        property_declaration(self.mything)
        self.mything.set_property_read_handler(property_name="temperature-air-in",
read_handler=self.get_Temperature_Air_In)
        self.mything.set_property_read_handler(property_name="temperature-air-out",
read_handler=self.get_Temperature_Air_Out)
        self.mything.set_property_read_handler(property_name="humidity-air-in",
read_handler=self.get_Humidity_Air_In)
...
```

```
        ###### ACTION DEFINITION ######
        self.mything.add_action(name="restart", safe=True, idempotent=True,
description="restarts the Pycom")
        self.mything.set_action_handler(action_name="restart",
action_handler=self.action_restart)


        ###### ASYNCHRONOUS TASKS ######
        self.loop = asyncio.get_event_loop()
        self.loop.create_task(self.read_Air_Temperature_Probes())
        self.loop.create_task(self.read_Air_Humidity())
        self.loop.create_task(self.read_Water_Temperature_Probes())
...

        ###### START WOT SERVER ######
        self.mything.expose(server=server, broker=broker, ssid=ssid,
ssid_password=ssid_password, port=1883)


    ###### UTILITIES ######
    async def action_restart(self, value):

    async def check_network_status(self):

    async def DS18b20_Read(self, sensor):

    def median(self, list):


    ###### AIR TEMPERATURE AND HUMIDITY SENSORS ######
    async def get_Temperature_Air_In(self):

    async def get_Humidity_Air_In(self):

    async def get_Temperature_Air_Out(self):
    ...

    async def read_Air_Humidity(self):

    async def read_I2C_Air_Temperature_Humidity_Facade(self):

    async def read_Air_Temperature_Probes(self):
    ...

    ###### WATER SENSORS ######
```

```
    async def get_Temperature_Hot_Water_In(self):

    async def get_Temperature_Hot_Water_Out(self):

    async def get_Temperature_Cold_Water_In(self):
    ...

    ###### ACCELEROMETER ######
    async def get_Accelerometer_Data(self):

    async def read_Accelerometer_Data(self):


    ###### CURRENT CONSUMPTION IN PULSES ######
    async def get_Current_Power_Consumption(self):

    async def read_Current_Power_Consumption(self):


    ###### CURRENT CONSUMPTION GCBC100-2A ######
    async def get_GCBC1002A_Amps(self):

    async def action_GCBC1002A_base(self, value):

    async def read_GCBC1002A_Amps(self):
    ...

    ###### WATER FLOW SENSORS ######
    async def get_Cold_Water_Flow(self):

    async def get_Hot_Water_Flow(self):

    async def read_Cold_Water_Flow(self):
    ...

    ###### THERMAL ENERGY CALCULATION ######
    async def get_Cold_Thermal_Energy_Output(self):

    async def get_Hot_Thermal_Energy_Output(self):
    ...
```

T4.3: Despliegue de la aplicación WoT demostradora y validación

En esta tarea se ha representado un escenario de validación que modela la interacción con dispositivos industriales sin opciones de conectividad estandarizadas. Es posible ofrecer una interfaz estandarizada al

dispositivo industrial implementando una aplicación basada en el entorno de ejecución proporcionado por WoRMS.

Este escenario de validación incluye:

- **Despliegue.** Identifica las aplicaciones que conforman el escenario de validación y la relación entre ellas.
- **Configuración de red.** Identifica los dispositivos físicos del sistema que formarán parte de una red privada local.

Esta tarea se desarrolla sobre dos versiones de la implementación de referencia con el fin de abarcar el mayor número de dispositivo IoT posibles: una versión completa destinada a dispositivos y sistemas con capacidades de computación elevadas (WoRMS) y otra versión restringida, orientada a dispositivos y sistemas con capacidades de computación muy reducidas (WoRMS Lite).

Se explica a continuación, el despliegue en cada una de las versiones:

- **WoRMS - Despliegue de la aplicación WoRMS demostradora y validación**
  - **Despliegue**

    El escenario propuesto está compuesto por tres aplicaciones independientes:

    1) **Gateway Web of Things.** Servicio que expone una interfaz HTTP a las variables definidas en el dispositivo. Se comunica con el PLC Siemens mediante librerías específicas.
    2) **Aplicación Web.** Interfaz gráfica que permite monitorizar la planta industrial mediante comunicación directa con el Gateway WoT realizando peticiones HTTP.
    3) **Proxy.** Puesto que el Gateway WoT no añade cabeceras CORS a las peticiones de manera automática, se introduce un proxy intermedio que facilita la comunicación entre navegadores web remotos y el Gateway WoT.

    Todas las aplicaciones se definen como contenedores docker independientes y pueden desplegarse de manera conjunta con el siguiente comando:

    ```
    docker-compose -f docker-compose-lite-i86.yml up -d --build
    ```

    La aplicación Web principal será accesible en el puerto 8080, el proxy en el 3001 y el Gateway en el 9292, siendo todos los puertos configurables en el fichero yml. A su vez el Gateway es dependiente de dos servicios remotos que se corresponden con el PLC industrial y una base de datos en la que persistir los datos del dispositivo. Ambos servicios deben encontrarse en la misma red que los servicios desplegados, siendo su URL configurable en el fichero yml.

    Se presentan a su vez diversos ficheros yml para cubrir diferentes entornos de despliegue:

    - Despliegue simple en Raspberry Pi
    - Despliegue simple en arquitectura i86
    - Despliegue completo en Raspberry Pi (incluye base de datos mySQL y Grafana)

- Despliegue completo en arquitectura i86 (incluye base de datos mySQL y Grafana)

- **Configuración de red**

  Se espera que todos los componentes del sistema formen parte de una red local privada para garantizar la conectividad. Los dispositivos físicos utilizados fueron los siguientes:
  - Router con red WiFi
  - PLC conectado al router por Ethernet
  - Raspberry Pi conectada vía WiFi al router
  - Servidor remoto accesible por red: contiene la base de datos mySQL y un Grafana en caso de que corresponda.

- **WoRMS - Despliegue de la aplicación WoRMS Lite demostradora y validación**
  - **Despliegue**

    El escenario propuesto está compuesto por tres aplicaciones independientes:

    1) **Gateway Web of Things.** Servicio que expone una interfaz HTTP a las variables definidas en el dispositivo. Se comunica con el PLC Siemens mediante librerías específicas.
    2) **Aplicación Web.** Interfaz gráfica que permite monitorizar la planta industrial mediante comunicación directa con el Gateway WoT realizando peticiones HTTP.
    3) **Proxy.** Puesto que el Gateway WoT no añade cabeceras CORS a las peticiones de manera automática, se introduce un proxy intermedio que facilita la comunicación entre navegadores web remotos y el Gateway WoT.

    Todas las aplicaciones anteriores han sido definidas como contenedores docker independientes y pueden desplegarse de manera conjunta con el siguiente comando:

    ```
    docker-compose -f docker-compose-lite-i86.yml up -d --build
    ```

    La aplicación web principal será accesible en el puerto 8080, el proxy en el 3001 y el Gateway en el 9292, siendo todos los puertos configurables en el fichero yml. A su vez el Gateway es dependiente de dos servicios remotos que se corresponden con el PLC industrial y una base de datos en la que persistir los datos del dispositivo. Ambos servicios deben encontrarse en la misma red que los servicios desplegados, siendo su URL configurable en el fichero yml.

    Se presentan a su vez diversos ficheros yml para cubrir diferentes entornos de despliegue:
    - Despliegue simple en Raspberry Pi
    - Despliegue simple en arquitectura i86
    - Despliegue completo en Raspberry Pi (incluy base de datos mySQL y Grafana)
    - Despliegue completo en arquitectura i86 (incluye base de datos mySQL y Grafana)

  - **Configuración de red**

Se espera que todos los componentes del sistema formen parte de una red local privada para garantizar la conectividad. Los dispositivos físicos utilizados fueron los siguientes:

- Router con red WiFi
- PLC conectado al router por Ethernet
- Raspberry Pi conectada vía WiFi al router
- Servidor remoto accesible por red: contiene la base de datos mySQL y un Grafana en caso de que corresponda.

2. RESULTADOS CONSEGUIDOS

Justificación 2018

Durante el periodo de tiempo que cubre este informe, se han conseguido todos los resultados planificados, tal y como se había planteado al inicio del proyecto y que se detallan a continuación, relacionándolos con los objetivos específicos del proyecto:

- OE1 – Implementación de referencia de WoT Runtime y WoT Scripting API:
  - Durante la anualidad 2018, se ha obtenido el diseño general de la implementación de referencia WoRMS, y se ha comenzado a trabajar en el diseño de detalle e implementación del WoT Runtime y de los Protocol Binding HTTP y Websockets.
- OE2 – Implementación de WoT Runtime y WoT Scripting API para dispositivos limitados:
  - Durante la anualidad 2018, se ha obtenido el diseño general de la implementación de referencia WoRMS Lite, estando previsto el inicio de su desarrollo e implementación para la anualidad 2019.
- OE3 – Despliegue WoT demostrador:
    - Si bien el diseño de detalle y la implementación de la aplicación WoT demostradora no se iniciarán hasta la anualidad 2020, en esta anualidad 2018 se han sentado las bases de dicha aplicación con el diseño general de dicha aplicación, lo que orienta los trabajos del proyecto hacia el cumplimiento de los siguientes indicadores.

Justificación 2019

Durante el periodo de tiempo que cubre este informe, se han conseguido todos los resultados planificados, tal y como se había planteado al inicio del proyecto y que se detallan a continuación, relacionándolos con los objetivos específicos del proyecto:

- OE1 – Implementación de referencia de WoT Runtime y WoT Scripting API:
    - *Se pueden crear y gestionar interacciones de WoT Things con capacidad de procesamiento completo.*

      A través del modelo teórico *Interaction Model* se engloban todas las interacciones que se pueden llevar a cabo sobre una Thing. Cualquier funcionalidad pública de una Thing puede modelarse según el Interaction Model y los 3 patrones que describe: Property (atributos o valores que pueden ser leídos o escritos), Action (procedimientos de larga duración que no tienen cabida dentro de una Property), Event (ocurrencias observadas por la Thing que so comunicadas a los clientes que tienen una suscripción).
    - *Las WoT Things pueden ser expuestas y consumidas a través de los cuatro Protocol Bindings desarrollados.*

      A través del WoT Runtime se proporciona la funcionalidad base que permite consumir y exponer Things en la red, y a través de WoT Protocol Bindings se permite traducir entre las interacciones de alto nivel en una Thing y los mensajes de bajo nivel que se transmiten en la red. Se ha validado la compatibilidad de los cuatro Protocol Bindings desarollados (HTTP, MQTT, CoAP y Websockets) con el WoT Runtime a través de varios tests y benchmarks.
- OE2 – Implementación de WoT Runtime y WoT Scripting API para dispositivos limitados:
    - Durante la anualidad 2019, se ha comenzado a trabajar en el diseño de detalle de la implementación del WoT Runtime y del Protocol Binding de la implementación de referencia WoRMS Lite, lo que está encaminando el proyecto hacia el cumplimiento de los siguientes indicadores de consecución:
- OE3 – Despliegue WoT demostrador:
    - Si bien el diseño de detalle y la implementación de la aplicación WoT demostradora no se iniciarán hasta la anualidad 2020, en esta anualidad 2019 se ha confirmado la viabilidad del diseño general de dicha aplicación realizado en la anualidad 2018.

Justificación 2020

Durante el periodo de tiempo que cubre este informe, se han conseguido todos los resultados planificados, tal y como se había planteado al inicio del proyecto y que se detallan a continuación, relacionándolos con los objetivos específicos del proyecto:

- OE2 – Implementación de WoT Runtime y WoT Scripting API para dispositivos limitados:
    - Se pueden crear y gestionar interacciones de WoT Things con capacidad de procesamiento limitado.
      Las *Thing Description* proveen de una serie de interacciones basadas en un pequeño vocabulario que permite la integración con otras Things o aplicaciones de manera simple. A su vez, el conjunto de W3C WoT Protocolo Bindings, permitirá traducir entre las interacciones de alto nivel en una Thing y los mensajes de bajo nivel que se transmiten en la red.

    - Las WoT Things pueden ser expuestas y consumidas a través del protocolo.

      Los objetos que implementan la interfaz *ExposedThing,* disponen del método *expose* donde se especifica el parámetro asociado al protocolo a utilizar: HTTP o MQTT. Estos objetos se añaden al servidor y desde ahí se exponen. Este servidor escuchará las peticiones a través del protocolo, pudiendo así consumir las WoT Things.
- OE3 – Despliegue WoT demostrador:
    - *Se han conseguido desplegar, instalar y configurar los dispositivos hardware definidos en una infraestructura de red.* Las aplicaciones demostradoras de la versión WoRMS y WoRMS Lite se han desplegado, instalado y configurado correctamente según se ha indicado en apartados anteriores.
    - *El funcionamiento de este despliegue es conforme a los requisitos establecidos.* Según se ha descrito, el funcionamiento de las aplicaciones demostradoras da cumplimiento a los requisitos que se establecieron en la fase de diseño de dichas aplicaciones, con lo que se da por cumplido este indicador.