UNIVERSIDAD DE OVIEDO

PROGRAMA DE DOCTORADO DE INFORMÁTICA

Tesis Doctoral

Diseño de Componentes para la Web of Things y Evaluación de su Integración en Aplicaciones IoT

Autor: Andrés García Mangas

Director: Francisco José Suárez Alonso

Junio 2022

Universidad de Oviedo

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español: *Diseño de Componentes para la Web of Things y Evaluación de su Integración en Aplicaciones IoT*	Inglés: *Design of Web of Things Components and Assessment of their Integration in IoT Applications*

2.- Autor	
Nombre: Andrés García Mangas	DNI: [redacted]
Programa de Doctorado: Informática	
Órgano responsable: Centro Internacional de Postgrado	

RESUMEN (en español)

El crecimiento sostenido del Internet of Things (IoT) ha dado lugar a múltiples ecosistemas y protocolos dispares, lo cual tiene un impacto negativo en la interoperabilidad y el potencial del IoT. Una de las consecuencias más destacables es la aparición de silos de datos, que son conjuntos de datos aislados que presentan barreras costosas para la integración, es decir, para la extracción de valor. Estas barreras crecen normalmente con el tiempo a medida que se integran nuevas tecnologías y los técnicos que poseen el conocimiento experto dejan de estar activamente involucrados. La Web of Things (WoT) es una iniciativa reciente que persigue la adopción de patrones y tecnologías de la Web para dar solución a este problema de interoperabilidad en el IoT. La Web es uno de los dominios tecnológicos más extendidos y mejor conocidos, por lo que resulta un candidato prometedor.

En esta tesis se busca explorar el paradigma WoT, concretamente las especificaciones W3C WoT, en el contexto de los sistemas IoT modernos. Para ello se han diseñado dos componentes experimentales. En primer lugar, un framework que implementa los bloques fundamentales establecidos por la arquitectura W3C WoT. Concretamente, se incluyen un procesador de documentos Thing Description, un WoT Runtime que expone una API de acuerdo con la especificación WoT Scripting API y cuatro implementaciones de Protocol Binding para los protocolos HTTP, Websockets, CoAP y MQTT. Estas cuatro implementaciones son novedosas y soportan el conjunto completo de verbos del modelo de interacciones WoT. En segundo lugar, un emulador de aplicaciones WoT basadas en arquitectura edge computing. El emulador permite ejecutar código real y utiliza las capacidades de un orquestador de contenedores moderno para garantizar la escalabilidad horizontal de los experimentos, en contraste con otras herramientas dentro del estado de la técnica.

Las contribuciones de los componentes se han validado con dos experimentos. El primer experimento evalúa de manera exhaustiva el rendimiento de las implementaciones de Protocol Binding del framework WoT para los verbos del modelo de interacciones. El segundo experimento demuestra el emulador de aplicaciones WoT en el contexto de un caso de uso realista con componentes de inteligencia artificial y procesamiento de flujos de vídeo. Los resultados demuestran la adecuación de los componentes e ilustran las posibilidades que el paradigma WoT ofrece para afrontar el reto de la interoperabilidad IoT.

Universidad de Oviedo

RESUMEN (en Inglés)

The sustained growth of the Internet of Things (IoT) has resulted in multiple ecosystems and protocols. This heterogeneous nature has a drastic impact on interoperability, which constrains the potential of the IoT. The creation of data silos is one of the most notable impacts in this case. Data silos are isolated datasets with high barriers of entry that complicate integration tasks, that is, it is significantly difficult to extract value from said datasets. Integration barriers grow with the passage of time as the technical personnel leaves the organization or moves on to another projects. The Web of Things (WoT) is a recent paradigm that is focused on solving the IoT interoperability challenge by leveraging well-known Web technologies and architectural patterns.

This thesis explores the WoT paradigm, particularly the W3C WoT specifications, in the context of modern IoT systems. To that end, two distinct experimental components have been designed. The first component is a software framework that includes the building blocks identified by the W3C WoT architecture. More specifically, the framework includes a Thing Description processor, a WoT Runtime exposing an API that implements the WoT Scripting API specification and a set of Protocol Binding implementations for four application-layer protocols: HTTP, Websockets, CoAP and MQTT. These Protocol Bindings include novel contributions and provide support for the entire set of verbs defined in the WoT interaction model. The second component is an emulator for WoT applications based on the edge computing architectural model. The emulator enables the execution of real code and leverages the unique capabilities of a modern container orchestration tool to ensure the horizontal scalability of the experiments, unlike other tools in the state of the art.

Two experiments have been used to test the validity and contributions of the components. The first experiment comprehensively evaluates the performance of the Protocol Binding implementations depending on the verbs of the interaction model. The second experiment demonstrates the emulator in the context of a realistic use case based on artificial intelligence and video streaming techniques. Results show the suitability of the components and also serve to demonstrate the possibilities of the W3C WoT architecture in the face of the IoT interoperability challenge.

SR. PRESIDENTE DE LA COMISIÓN ACADÉMICA DEL PROGRAMA DE DOCTORADO EN INFORMATICA

Resumen

El crecimiento sostenido del Internet of Things (IoT) ha dado lugar a múltiples ecosistemas y protocolos dispares, lo cual tiene un impacto negativo en la interoperabilidad y el potencial del IoT. Una de las consecuencias más destacables es la aparición de silos de datos, que son conjuntos de datos aislados que presentan barreras costosas para la integración, es decir, para la extracción de valor. Estas barreras crecen normalmente con el tiempo a medida que se integran nuevas tecnologías y los técnicos que poseen el conocimiento experto dejan de estar activamente involucrados. La Web of Things (WoT) es una iniciativa reciente que persigue la adopción de patrones y tecnologías de la Web para dar solución a este problema de interoperabilidad en el IoT. La Web es uno de los dominios tecnológicos más extendidos y mejor conocidos, por lo que resulta un candidato prometedor.

En esta tesis se busca explorar el paradigma WoT, concretamente las especificaciones W3C WoT, en el contexto de los sistemas IoT modernos. Para ello se han diseñado dos componentes experimentales. En primer lugar, un framework que implementa los bloques fundamentales establecidos por la arquitectura W3C WoT. Concretamente, se incluyen un procesador de documentos Thing Description, un WoT Runtime que expone una API de acuerdo con la especificación WoT Scripting API y cuatro implementaciones de Protocol Binding para los protocolos HTTP, Websockets, CoAP y MQTT. Estas cuatro implementaciones son novedosas y soportan el conjunto completo de verbos del modelo de interacciones WoT. En segundo lugar, un emulador de aplicaciones WoT basadas en arquitectura edge computing. El emulador permite ejecutar código real y utiliza las capacidades de un orquestador de contenedores moderno para garantizar la escalabilidad horizontal de los experimentos, en contraste con otras herramientas dentro del estado de la técnica.

Las contribuciones de los componentes se han validado con dos experimentos. El primer experimento evalúa de manera exhaustiva el rendimiento de las implementaciones de Protocol Binding del framework WoT para los verbos del modelo de interacciones. El segundo experimento demuestra el emulador de aplicaciones WoT en el contexto de un caso de uso realista con componentes de inteligencia artificial y procesamiento de flujos de vídeo. Los resultados demuestran la adecuación de los componentes e ilustran las posibilidades que el paradigma WoT ofrece para afrontar el reto de la interoperabilidad IoT.

Abstract

The sustained growth of the Internet of Things (IoT) has resulted in multiple ecosystems and protocols. This heterogeneous nature has a drastic impact on interoperability, which constrains the potential of the IoT. The creation of data silos is one of the most notable impacts in this case. Data silos are isolated datasets with high barriers of entry that complicate integration tasks, that is, it is significantly difficult to extract value from said datasets. Integration barriers grow with the passage of time as the technical personnel leaves the organization or moves on to another projects. The Web of Things (WoT) is a recent paradigm that is focused on solving the IoT interoperability challenge by leveraging well-known Web technologies and architectural patterns.

This thesis explores the WoT paradigm, particularly the W3C WoT specifications, in the context of modern IoT systems. To that end, two distinct experimental components have been designed. The first component is a software framework that includes the building blocks identified by the W3C WoT architecture. More specifically, the framework includes a Thing Description processor, a WoT Runtime exposing an API that implements the WoT Scripting API specification and a set of Protocol Binding implementations for four application-layer protocols: HTTP, Websockets, CoAP and MQTT. These Protocol Bindings include novel contributions and provide support for the entire set of verbs defined in the WoT interaction model. The second component is an emulator for WoT applications based on the edge computing architectural model. The emulator enables the execution of real code and leverages the unique capabilities of a modern container orchestration tool to ensure the horizontal scalability of the experiments, unlike other tools in the state of the art.

Two experiments have been used to test the validity and contributions of the components. The first experiment comprehensively evaluates the performance of the Protocol Binding implementations depending on the verbs of the interaction model. The second experiment demonstrates the emulator in the context of a realistic use case based on artificial intelligence and video streaming techniques. Results show the suitability of the components and also serve to demonstrate the possibilities of the W3C WoT architecture in the face of the IoT interoperability challenge.

Índice general

Índice de figuras

Índice de cuadros

Índice de bloques de código

Prefacio

La presente Tesis Doctoral ha sido el resultado de cinco años de trabajo, siempre con la ayuda inestimable del Prof. Francisco José Suarez Alonso y mis compañeros en CTIC Centro Tecnológico (presentes y pasados) a los que me siento afortunado de poder llamar amigos también. A todos ellos les debo mucho por allanarme el camino y estar siempre dispuestos a ofrecer su asistencia, moral y técnica.

En el plano familiar, mis padres Manoli y Mario, mi pareja Tania, mi hermano Carlos, mi *otra hermana* Cris, mi abuela Eloína, mis tíos Susana y Moisés, mis *padres adoptivos* Pili y Julio, y por supuesto el resto de mi familia, me han apoyado incondicionalmente siempre. Cualquier éxito que haya podido conseguir es gracias a ellos.

En el plano personal, tengo una deuda impagable con mis amigos Buster, Castri, Celia, Juan, Maider, Minerva, Rebo, Sonia y Voby. Hay un montón de personas a la que les debo muchas cosas como para mencionarlas a todas sin extenderme demasiado, así que ruego que nadie se sienta ofendido. La importancia que tienen en mi vida es complicada de explicar, por lo que prefiero ser breve en mi agradecimiento. No podría haber pedido gente mejor a mi alrededor.

Finalmente, también quiero hacer mención al Prof. Pablo Alonso García, que ha sido un mentor y un amigo desde el primer momento en el que le conocí.

Todas estas personas son una parte inseparable de mi mismo, por lo que este trabajo les pertenece a ellas tanto como me pertenece a mí.

Capítulo 1

Introducción

El *Internet of Things* (IoT) es el nombre general que recibe el conjunto interconectado de dispositivos comúnmente conocidos como *Things* en Internet. Dentro de esta categoría de Things se suelen incluir sensores, maquinaria industrial especializada, electrodomésticos, y cualquier otro tipo de dispositivo conectado que no actúa principalmente como terminal directo de usuario. Las Things son mayoritariamente físicas, aunque la definición es amplia y podría incluir entidades virtuales. Además, tienden a tener propósitos específicos, en contraste con el propósito general de un ordenador personal. En otras palabras, el IoT no es un producto ni una tecnología concreta bien delimitada, es más adecuado definirlo como un concepto de alto nivel basado en la interconexión de dispositivos diversos en Internet.

En términos de arquitectura, el IoT puede caracterizarse [1] en tres capas: la capa de *percepción*, que incluye los sensores y dispositivos que hacen interfaz con el mundo real; la capa de *red*, que comprende la infraestructura y tecnologías de red que habilitan la interconexión; y la capa de *aplicación*, que implementa la lógica de negocio. Adicionalmente, la tendencia actual apunta a un crecimiento de la relevancia de la arquitectura conocida como *edge computing* [2, 3], basada en la redistribución de servicios de las capas altas de nube a las capas intermedias en la frontera de los despliegues. La arquitectura *edge computing* se describe con más detalle en la sección 2.3.

Desde el punto de vista de bloques fundamentales, el IoT se puede dividir en seis componentes formados por una combinación de tecnologías, patrones, buenas prácticas y técnicas:

Identificación Tecnologías para direccionamiento (e.g. IPv4) y asignación de nombres para componentes de redes IoT.

Sensorización Principalmente, dispositivos sensores y actuadores que hacen de interfaz con el mundo real, midiendo métricas físicas o modifi-

cando su entorno.

Comunicación Protocolos de red que habilitan la comunicación entre dispositivos IoT. Existen protocolos para múltiples casos de uso, incluyendo alternativas de baja potencia y área local, como Bluetooth, y protocolos para comunicaciones de largo alcance, como LTE.

Computación Conjunto de plataformas e infraestructura hardware que proporciona los recursos de computación, así como las pilas software que se utilizan para construir las aplicaciones y servicios que se ejecutan sobre estos recursos.

Servicios Módulos que implementan lógica de negocio; por ejemplo, para exposición de datos extraídos de las capas bajas, análisis de los datos y actuación en función de los resultados obtenidos.

Semántica Conjunto de tecnologías que se utilizan para añadir conocimiento de *sentido común* a las interacciones entre máquinas. Esto habilita el descubrimiento de servicios de manera automática incluyendo conocimiento del contexto (e.g. RDF, OWL).

La relevancia del IoT emana de su clara tendencia de crecimiento a lo largo del siglo XXI. De hecho, los estudios apuntan a que en un futuro cercano existirán decenas de billones de Things en un mercado del orden de trillones de euros [4].

Este crecimiento explosivo ha dado lugar a su vez a un conjunto de retos. Uno de los retos más relevantes, que se ve notablemente impactado por el volumen de dispositivos instalados, es del de la *interoperabilidad*. De manera general, el objetivo de interoperabilidad se enfoca en reducir la fricción en despliegues heterogéneos. Es decir, en optimizar los costes de mantenimiento y desarrollo que surgen de la existencia de diferentes ecosistemas IoT, que a su vez implican diferencias significativas en herramientas de desarrollo, capacidades, protocolos e interfaces.

La aparición de *silos de datos* es uno de los impactos negativos más relevantes causados por los problemas de interoperabilidad. Los silos de datos son conjuntos de datos aislados que presentan barreras costosas para la integración, es decir, para la extracción de valor. Estas barreras crecen normalmente con el tiempo a medida que se integran nuevas tecnologías y los técnicos que poseen el conocimiento experto dejan de estar activamente involucrados. La cuestión no es que los datos dentro del *silo* no tengan interés, si no que el coste de implementar las herramientas necesarias para la integración y explotación es lo suficientemente alto como para que no resulte eficiente.

Se pueden trazar paralelismos claros entre Internet y el IoT. En la década de los 90, Internet se enfrentó a retos serios en cuestión de fragmentación. Diversos proveedores ofrecían servicios online que estaban notablemente alejados del ecosistema que hoy forma Internet. Para solucionar esta problemática nació el World Wide Web Consortium (W3C), que tuvo un papel instrumental para que las organizaciones acordasen asentar el desarrollo de Internet sobre dos cimientos tecnológicos: HTML, como lenguaje de marcado para definición de la estructura las páginas web; y HTTP, como protocolo de capa de aplicación transportado sobre TCP para formalización de las interacciones entre nodos en Internet.

La experiencia de Internet es un indicativo claro de que adoptar un conjunto tecnologías y prácticas comunes, compartidas y validadas por las organizaciones integradas en el dominio del IoT, puede ser una solución efectiva al problema de la interoperabilidad en el IoT. Concretamente, las tecnologías de la Web pueden volver a resultar un candidato efectivo para proporcionar los bloques fundamentales que aseguren la interoperabilidad. La Web es uno de los ecosistemas tecnológicos mejor conocidos y documentados, probablemente el más popular a día de hoy. Es además un ecosistema en constante crecimiento, y que se ha visto revitalizado por la popularidad y las mejoras de experiencia de usuario conseguidas en el dominio de las aplicaciones Web enfocadas a dispositivos móviles, que están acortando la distancia de funcionalidad con las aplicaciones nativas.

El campo de la *Web of Things* (WoT) nace [5, 6, 7] precisamente de esta idea principal: la aplicación de las tecnologías y técnicas de la Web al dominio del IoT para proporcionar solución al problema de interoperabilidad. Las lecciones aprendidas en el pasado reciente nos proporcionan un nivel razonable de confianza para confirmar que resulta de interés invertir esfuerzos en avanzar este campo de conocimiento. Afrontar el reto de interoperabilidad en el IoT es un imperativo para permitir su crecimiento, y para que el IoT consiga asentarse de manera estable a largo plazo en el día a día de la sociedad.

En el resto de este documento se describen un conjunto de trabajos de investigación en el dominio de la Web of Things. De manera específica, se persigue explorar las especificaciones WoT del W3C en el contexto de los sistemas IoT modernos. Con este objetivo, se elaboran implementaciones experimentales de especificaciones y herramientas basadas en los bloques fundamentales de la W3C WoT, proponiendo contribuciones novedosas donde se detecta la necesidad. Esto permite a su vez extraer experiencias y conclusiones de interés. La organización del documento se describe a continuación:

- En el **capítulo 2** se presenta una visión amplia del estado de la técnica

en los dominios de la interoperabilidad IoT, la Web of Things, y la simulación y emulación de despliegues IoT/WoT. De este estado de la técnica se derivan las **motivaciones** de la tesis.

- En el **capítulo 3** se presenta el diseño de una implementación experimental de los bloques fundamentales de la W3C Web of Things.
- En el **capítulo 4** se presenta el diseño de una herramienta basada en orquestación de contenedores para la emulación de despliegues IoT/WoT basados en arquitectura edge computing.
- En el **capítulo 5** se describe un conjunto de experimentos para validación de los diseños previamente presentados en los capítulos 3 y 4.
- En el **capítulo 6** se presentan las conclusiones que emanan de la visión de alto nivel conseguida en el transcurso de los trabajos.
- En el **apéndice A** se adjunta la versión aceptada de una publicación de impacto elaborada a partir del diseño experimental de los bloques fundamentales W3C WoT (capítulo 3).
- En el **apéndice B** se adjunta la versión aceptada de una publicación de impacto elaborada a partir de la herramienta de emulación WoT (capítulo 4).
- En el **apéndice C** se adjunta la documentación de la API para la implementación experimental de los bloques fundamentales W3C WoT.

Capítulo 2

Trabajo previo

En este capítulo se describe el conjunto de publicaciones, estándares y especificaciones más relevantes que constituyen la base del estado de la técnica en los dominios que sirven de cimiento para este trabajo. Concretamente, el análisis se ha dividido en las siguientes secciones:

- En la sección 2.1 se describen las características técnicas de estándares relevantes de interoperabilidad IoT. Estas especificaciones buscan reducir los costes de desarrollo, despliegue y mantenimiento de soluciones IoT, y resultan un punto de partida e inspiración para la W3C WoT. De hecho, el grupo de trabajo del W3C ha manifestado que uno de los objetivos principales de las especificaciones W3C WoT es proporcionar un puente entre estándares de interoperabilidad. Es decir, la W3C WoT no persigue sustituir los estándares que ya se encuentran implantados en campo.

- En la sección 2.2 se proporciona una vista general de esfuerzos académicos relacionados en mayor o menor medida con la WoT. Por ejemplo, se incluyen casos de uso reales, artículos de análisis del ecosistema, frameworks software y comparativas de protocolos. En la sección 2.2.1 se describen las particularidades técnicas de las especificaciones W3C WoT, que resultan uno de los cimientos principales del presente trabajo. Concretamente, tienen mayor impacto en el diseño del framework de aplicaciones WoT presentado en la sección 3.

- En la sección 2.3 se analiza el patrón de arquitectura IoT conocido como *edge computing*. Además, en la sección 2.3.1 se incluye un análisis del estado de la técnica en la simulación y emulación de despliegues IoT, que sirve de base para el diseño del emulador presentado en la sección 4.

- Basado en las conclusiones extraídas de las secciones anteriores, en la sección 2.4 se proporcionan las motivaciones generales que dan lugar a los trabajos realizados en el contexto de esta tesis.

2.1. Estándares de interoperabilidad

En esta sección se presentan especificaciones que tienen paralelismos con la W3C WoT. Es decir, que abordan el problema de la interoperabilidad en el IoT de alguna manera. Todas ellas son notablemente complejas, así que por claridad se ha decidido adoptar una perspectiva enfocada en los detalles técnicos que las caracterizan: modelos de datos, protocolos, formatos de serialización y arquitecturas de comunicación.

Es importante destacar que la W3C WoT no persigue *sustituir* a las especificaciones existentes, si no proporcionar un conjunto de bloques fundamentales de interoperabilidad IoT lo suficientemente flexibles como para *incluir* a las demás. De hecho, los documentos W3C WoT hacen referencia explícita [8] a ecosistemas como OCF o oneM2M.

En la tabla 2.1 se muestra un resumen de las características técnicas más relevantes de las especificaciones de interoperabilidad. Se identifican cinco características en cada ecosistema: la adopción del patrón arquitectural REST para la construcción de servicios; la capacidad de implementar modelos de mensajería basados en publicación-suscripción; la integración de vocabularios bien conocidos según los principios de *Linked Data*; el número de *Protocol Bindings* u opciones para traducción de interacciones de alto nivel a mensajes de red; y finalmente si el ecosistema se enfoca en los principios y patrones de la Web, es decir, en la WoT, o por el contrario toma una vista más centrada en el IoT *clásico*.

Ecosistema	REST	Pub-Sub	Linked Data	Protocol Bindings	*Web Focus*
OCF	•	-	-	2	-
LwM2M	•	-	-	1	-
oneM2M	•	•	-	4	-
Web Thing	•	•	•	2	•
Project Things	•	•	-	2	•
OGC	•	•	-	2	•

Cuadro 2.1: Resumen de ecosistemas IoT

2.1.1. Open Connectivity Foundation (OCF)

Esta especificación [9] está desarrollada por la *Open Connectivity Foundation* (OCF) y define una arquitectura y protocolo REST sobre CoAP, así como modelos de datos y extensiones para diferentes verticales (e.g. *Smart Home*):

- Las comunicaciones ocurren sobre el protocolo CoAP con contenido codificado en CBOR (JSON en formato binario). El stack soporta CoAP sobre UDP (encriptación con DTLS) y CoAP sobre TCP (encriptación con TLS). Opcionalmente se puede soportar también HTTP.
- IoTivity [10] es la implementación de referencia propuesta por la OCF. Presenta dos versiones (*full-featured* y *lite*) que cubren la versión 1.3 de la especificación.
- Está basada en una arquitectura cliente - servidor que sigue el patrón REST. Las entidades exponen sus funcionalidades a través de *recursos* identificados por URIs. Dichos recursos se manipulan con un conjunto restringido de verbos CRUDN (*Create, Read, Update, Delete, Notify*).
- Existen tres actores diferenciados dentro de OCF:
  - *OCF Server*: Entidad que expone recursos.
  - *OCF Client*: Entidad que interactúa con los recursos a través de acciones CRUDN.
  - *OCF Device*: Entidad que actúa como cliente y servidor al mismo tiempo. Es interesante destacar que un mismo dispositivo físico puede contener varios *OCF Device* lógicos.
- OCF está basado en un modelo de datos denominado *Resource Model*:
  - Una de las entidades principales es el *Resource*, que está identificado por una URI y reside en un *Device*. Un *Resource* es una instancia de uno o más *Resource Type*.
  - Un *Resource Type* define el conjunto de *Property* esperables en un *Resource* de dicho tipo. Las *Property* representan piezas de información relacionadas con el *Resource*. Pueden ser obligatorias u opcionales, así como *writable* o *read only*.
  - Un *Device* pertenece a una categoría o tipo denominada *Device Type*. El *Device Type* define un conjunto mínimo de *Resource*

Types que deben ser soportados por el *Device*. Un *Device* puede exponer otros *Resource Types* arbitrariamente más allá de los mínimos requeridos.

- Existe un conjunto de *Device Types* definidos por el OCF [11]. La especificación también admite la definición de *Device Types* por terceras partes. Algunos ejemplos de *Device Type* definidos por el OCF: *Air Purifier*, que debe soportar el *Resource Type* `oic.r.switch.binary` o *Camera*, que debe soportar `oic.r.media`.
- Existe un conjunto de *Resource Types* denominados los *Core Resources* que resultan transversales a todas las verticales. Tres de ellos deben ser soportados de manera obligatoria por todos los *Device*: *Platform* (`oic.wk.p`) contiene información de plataforma como el fabricante o la versión del hardware; *Discoverable Resources Link List Interface* (`oic.wk.res`) permite el descubrimiento de otros recursos; *Device* (`oic.wk.d`) contiene información del *Device* como la versión del software.

- AllJoyn es otro framework para interoperabilidad en entornos IoT que se encuentra bajo el paraguas de la OCF. AllJoyn se encuentra en su *major release* final [12] y los esfuerzos de desarrollo de ahora en adelante se centrarán en IoTivity.

- Se pueden integrar otros ecosistemas basados en comunicaciones propietarias a través de *Bridges*. La interoperabilidad entre AllJoyn y OCF está explícitamente definida en uno de estos documentos (e.g. equivalencias entre *Resources* de OCF e *Interfaces* de AllJoyn).

2.1.2. LwM2M

LwM2M [13] es un protocolo de capa de aplicación basado en CoAP desarrollado por la *Open Mobile Alliance* (OMA):

- El stack de protocolos utiliza CoAP en su capa más alta con dos opciones de transporte: UDP o SMS (ambas con opción de ser encriptadas con DTLS o en claro). El contenido se serializa en varios formatos dependiendo del contexto (texto plano, binario o JSON).

- Dentro de la arquitectura de LwM2M se pueden identificar tres actores: *LwM2M Client*, *LwM2M Server* y *LwM2M Bootstrap-Server* (en los siguientes puntos se describe con mayor detalle su función).

- Está basado alrededor del denominado *Resource Model*. En este modelo un *LwM2M Client* contiene y expone *Object Instances* instanciadas a partir de modelos base *Object*. Estos *Object* están listados en un catálogo oficial que puede recibir contribuciones de terceras partes. Cada *Object* expone a su vez una serie de *Resources*.

  A continuación se presenta una lista de los *Object* normativos definidos por la especificación.

  - *LwM2M Security*: Credenciales para que un *LwM2M Client* pueda autenticarse en un *LwM2M Server*. Solo pueden ser manipulados por un *LwM2M Bootstrap-Server* y su acceso está bloqueado a los *LwM2M Server*. Ejemplo de *Resource*: *LwM2M Server URI* (URI CoAP que identifica unívocamente al servidor).
  - *LwM2M Server*: Datos relativos a un *LwM2M Server* en el que el cliente está registrado. Ejemplo de *Resource*: *Lifetime* (tiempo de vida del registro).
  - *Access Control*: Autorización de acciones de un *LwM2M Server* sobre el cliente. Ejemplo de *Resource*: *ACL* (desglose de permisos de escritura y lectura de un servidor).
  - *Device*: Información relacionada con el dispositivo que contiene el *LwM2M Client* y funciones para reiniciarlo. Ejemplo de *Resource*: *Firmware Version*.
  - *Connectivity Monitoring*: Información y métricas relacionadas con la conexión de red del cliente. Ejemplo de *Resource*: *Network Bearer* (clase de red sobre la que opera el dispositivo, por ejemplo, Bluetooth o GSM).
  - *Firmware Update*: Gestión del ciclo de vida del firmware del cliente. Un *LwM2M Server* puede transferir un paquete de firmware a un cliente o puede solicitarle que lo descargue de un tercer servidor. Ejemplo de *Resource*: *Package URI* (URI del paquete del paquete firmware que debe ser descargado).
  - *Location*: Información acerca de la localización del dispositivo cliente. Se pueden definir distintas geometrías en función del nivel de incertidumbre. Ejemplo de *Resource*: *Latitude*.
  - *Connectivity Statistics*: Recolección de estadísticas y agregados relacionados con las conexiones de red. Un *LwM2M Server* puede arrancar o parar el proceso de recolección, y consultar los resultados. Ejemplo de *Resource*: Tx Data (total de KB transmitidos).

- Define una serie de interfaces que modelan los intercambios entre cliente y servidor. Cada interfaz está formada por un conjunto de operaciones enfocado a dar soporte para una necesidad común de despliegues IoT. Las interfaces se comentan a continuación:

    - *Bootstrap*: Provisionado de información básica en un *LwM2M Client* con el objetivo de habilitarlo para un posterior registro en un *LwM2M Server*. Las comunicaciones se llevan a cabo entre un *LwM2M Client* y un *LwM2M Bootstrap-Server*, que es un actor específico para esta interfaz que no interviene en las demás interfaces.
    - *Client Registration*: Notificación por parte de un *LwM2M Client* a un *LwM2M Server* para comunicar su existencia y funcionalidades soportadas. La comunicación solo se produce en *uplink* (de cliente a servidor).
    - *Device Management and Service Enablement*: Acceso por parte de un *LwM2M Server* a los *Object Instances* y *Resources* expuestos por un *LwM2M Client*. Esta interfaz solo está habilitada después de que se haya completado el proceso de registro. La comunicación solo se produce en *downlink* (de servidor a cliente).
    - *Information Reporting*: Permite que un *LwM2M Server* cree y cancele subscripciones a flujos de actualizaciones en *Resources* existentes en un *LwM2M Client*. Las notificaciones se producen de manera asíncrona originadas en el *LwM2M Client*.

2.1.3. FIWARE

FIWARE es una plataforma promovida por la FIWARE Foundation (compuesta por entidades entre las que destacan Telefónica y Orange). La plataforma está formada por un conjunto de librerías y componentes software open source que buscan proporcionar herramientas reusables para la construcción de *smart applications*. Estas herramientas reciben el nombre de FIWARE *Generic Enablers* (GE) y se agrupan en función de su dominio (e.g. *Data/-Context Management*, IoT, *Security*).

La plataforma FIWARE se basa alrededor de un modelo de datos denominado NGSI v2 [14] que define tres elementos descritos a continuación. Recientemente FIWARE está evolucionando hacia un modelo llamado NGSI-LD [15] cuya principal diferenciación reside en utilizar el formato de serialización JSON-LD para soportar *Linked Data*.

- *Entities*: Entidades físicas o virtuales que tienen información asociada que puede ser actualizada y consultada (e.g. un sensor de temperatura).
- *Attributes*: Propiedades asociadas a las *Entities* que representan su estado (e.g. valor de temperatura de un sensor).
- *Metadata*: Piezas de información que describen los *Attributes* (e.g. precisión del valor de temperatura proporcionado por un sensor).

NGSI v2 define además una API REST sobre HTTP con contenido serializado en formato JSON para consultas, actualizaciones y gestión de observadores de datos de contexto. Al estar basado en REST sobre HTTP se utilizan peticiones HTTP para notificar las actualizaciones a los observadores, es decir, el servidor que implementa NGSI actúa a modo de cliente en la petición de notificación (al contrario que en otros protocolos como por ejemplo CoAP en los que las notificaciones se pueden iniciar desde el servidor).

A continuación se listan alguno de los FIWARE GE más destacables dentro del ecosistema:

- *Orion*: Un *context broker* que proporciona una implementación de la API de NGSI para actuar como punto central de gestión de la información de contexto.
- *Cygnus*: Un componente basado en Apache Flume para construcción de flujos de persistencia de datos de contexto. Esta herramienta extrae los datos de Orion de manera recurrente y los vuelca en un destino de almacenamiento (e.g. HDFS, MySQL, Kafka, STH Comet).
- *Comet*: Un componente de almacenamiento basado en MongoDB para persistencia de datos históricos de contexto (tanto en crudo como agregados).
- *IDAS*: Formado por un conjunto de *IoT Agents* que sirven de pasarela para adaptar la interacción y la recuperación de información de dispositivos IoT en diversos entornos con la interfaz NGSI. Los entornos soportados actualmente son JSON sobre MQTT/HTTP, LwM2M sobre CoAP y UltraLight2.0 sobre MQTT/HTTP. Proporciona librerías para la construcción de agentes personalizados.
- *Keyrock*: Un componente o *Identity Manager* para proporcionar autenticación, autorización y SSO (*Single Sign-On*). Implementa el protocolo OAuth 2, actuando como el *Resource Server* y *Authorization Server* dentro del flujo OAuth 2.

- *AuthZForce*: Un componente que utiliza el estándar XACML para gestión y definición de reglas para control de acceso. Dentro de la arquitectura XACML este componente actúa a modo de PDP (*Policy Decision Point*), el elemento que recibe las peticiones de evaluación de solicitudes de acceso, validando o rechazando en función de las reglas definidas. Se integra con *Keyrock* para formar el stack básico de seguridad en FIWARE.

2.1.4. oneM2M

Dentro de la especificación oneM2M [16] podemos identificar una serie de conceptos fundamentales:

- *Functional Entities*: Componentes funcionales divididos en función de los servicios que prestan o las funciones que llevan a cabo.
  - *Application Entity* (AE): Entidad que implementa la lógica de la aplicación (e.g. una aplicación para monitorización de sensores).
  - *Common Services Entity* (CSE): Entidad que proporciona un grupo de *common service functions*, un conjunto de funcionalidades comunes horizontales a la mayoría de despliegues IoT (e.g. *Discovery* o *Registration*).
  - *Network Services Entity* (NSE): Expone servicios de la red que van más allá del transporte de datos (e.g. servicios de localización, servicios de gestión de pagos).
- *Reference Points*: Localizaciones lógicas dentro del sistema que contienen las interfaces de comunicación de entidades CSE con otras entidades.
  - *Mca*: Comunicación CSE - AE.
  - *Mcc*: Comunicación CSE - CSE.
  - *Mcn*: Comunicación CSE - NSE.
  - *Mcc'*: Comunicación entre CSE situados en nodos IN que residen en dominios de diferentes *Service Provider*.
- *Nodes*: Entidades lógicas que contienen una o más *Functional Entities* y se categorizan en función del dominio en el que se despliegan y su composición interna. El mapeo con respecto a entidades físicas puede variar en función de cada caso. La tabla 2.2 contiene un listado de los distintos tipos de nodos.

- *Resources*: Entidad direccionable dentro de un sistema oneM2M que existe dentro de un CSE y representa cualquier elemento susceptible de ser el receptor de un *Procedure* (operaciones CRUDN). Pueden estar situados en el nivel raíz o existir como hijos de otro *Resource* de mayor nivel.

  Todos los *Resources* tienen un *Resource Type* asociado. Existen tipos para representar a todos los actores involucrados en el sistema, por ejemplo, entidades funcionales (e.g. *CSEBase*, *AE*), un canal de comunicación (e.g. *pollingChannel*) o instancias de datos (e.g. *contentInstance*).

  Cada uno de los verbos CRUDN tiene un significado particular en función del *Resource Type* sobre el que se aplica, por ejemplo, el verbo CREATE no está definido sobre un *CSEBase* dado que este *Resource* representa el recurso raíz de un CSE y debería ser inicializado por una operación externa.

El modelo general del flujo de comunicación de un *Procedure* en oneM2M (basado en el patrón petición-respuesta) puede ser mapeado a varios protocolos de transporte a través de un *Protocol Binding* (de manera similar a como ocurre dentro del estándar W3C WoT). Existen especificaciones para HTTP, CoAP, WebSocket y MQTT, además de guías de interoperabilidad con LwM2M, AllJoyn y la especificación OIC 1.0 del OCF.

Nombre	CSE	AE	Dominio	Ejemplo
Application Service Node (ASN)	1	1+	0+ en *Field*	M2M device
Application Dedicated Node (ADN)	0	1+	0+ en *Field*	Constrained M2M device
Middle Node (MN)	1	0+	0+ en *Field*	M2M gateway
Infrastructure Node (IN)	1	0+	1 en *Infrastructure*	M2M service infrastructure

Cuadro 2.2: Tipos de *Nodes* en oneM2M

2.1.5. Web Thing Model

La especificación [17] propone un modelo de datos y una API REST para la construcción de Things que puedan funcionar de manera interoperable dentro de la Web of Things.

Aunque el documento no es fruto del trabajo del W3C, ha servido como punto de partida para sus grupos dedicados a Web of Things y comparte múltiples similitudes con los resultados obtenidos por éstos.

La especificación está organizada en tres partes:

- Patrones de arquitectura para el despliegue e integración de Web Things y para la resolución de escenarios en los que los dispositivos involucrados no tienen acceso nativo a la Web (i.e. no tienen un stack TCP/IP o soporte para HTTP). Se contemplan tres escenarios: conectividad directa, con un puente gateway o con un puente en la nube.

- Un conjunto de requerimientos que determinan, entre otros, los protocolos que deben ser soportados (HTTP y, de manera recomendada, WebSockets), así como el conjunto de características esperables de una Web Thing (e.g. todas las Web Thing deben tener un recurso accessible en la URL HTTP raíz). Estos requerimientos se dividen por niveles en función del nivel de exigencia (mínimo necesario, recomendación, opcional).

- La especificación de un protocolo REST HTTP, incluyendo los recursos expuestos y la estructura de los mensajes JSON intercambiados. Los recursos definidos por esta API se listan a continuación:

  - *Web Thing*: Recurso raíz que sirve de punto de entrada a la API.
  - *Model*: Representación de la Thing que contiene metadatos (e.g. nombre, descripción) y *Links* expuestos, especialmente los de *Properties* y *Actions*.
  - *Properties*: Variables internas de la Thing.
  - *Actions*: Funciones expuestas por la Thing.
  - *Things*: Things contenidas para las cuales esta Thing sirve de pantalla o gateway.
  - *Subscriptions*: Entidad que representa a los observadores de una *Property* o *Action*. Dado que HTTP no tiene soporte nativo para notificaciones *server-side* se habilitan dos opciones: WebHooks (callbacks HTTP) o WebSockets.

Se definen además tres clases de Web Things en función del nivel de adherencia a la especificación:

- *Web Things* son aquellas que cumplen los requerimientos.

- *Extended Web Things* son aquellas que cumplen los requerimientos y además siguen la especificación del protocolo REST del *Web Thing Model*.
- *Semantic Web Things* son aquellas *Extended Web Things* que contienen anotaciones semánticas haciendo uso de la serialización JSON-LD.

2.1.6. Mozilla Project Things

Project Things es una iniciativa de Mozilla compuesta por un conjunto de servicios y componentes software para la construcción y el despliegue de Things en la Web of Things:

- Una serie de paquetes para Node, Python y Java denominado *Things Framework* para el desarrollo de Things.
- Una imagen de un sistema operativo preparado para ser desplegado en una Raspberry Pi denominado *Things Gateway*. Esta imagen habilita al dispositivo para comportarse como un concentrador y controlador de Things.
- Un conjunto de servicios IoT en la nube denominado *Things Cloud*.

La *Web Thing API* [18] es una especificación que actúa como la base integradora del ecosistema de *Project Things*. En este documento se presentan recomendaciones cuya intención es complementar las recomendaciones producidas por el grupo W3C WoT. El objetivo es reducir el coste de entrada a la WoT para los desarrolladores Web, haciéndola más amigable. Las aportaciones del documento se listan a continuación:

- Un formato para serialización simplificada de una Thing Description en JSON plano (*Web Thing Description*). Es interesante destacar que al no utilizar serialización en JSON-LD se simplifica su interpretación y se pierde la capacidad de integrar la Thing Description dentro del dominio de la red de *Linked Data*.
- Dos *Protocol Bindings* para mapear las interacciones con la Thing a los protocolos de transporte. Un *binding* para un protocolo REST sobre HTTP y otro sobre WebSockets.
- Un conjunto no exhaustivo de tipos predefinidos de Things. Estos tipos determinan las interacciones que deben ser soportadas por la Thing para ajustarse a un tipo concreto, así como restricciones sobre los valores de los campos de la Thing Description.

2.1.7. OGC SensorThings API

La API de SensorThings es un estándar que busca un objetivo muy similar al de la W3C WoT, es decir, la integración de dispositivos IoT en la Web a través de la utilización de tecnologías extendidas (e.g. JSON, HTTP), para reducir los ciclos y costes de desarrollo, mantenimiento y despliegue.

Está dividido en dos partes: la parte 1 [19] se centra en la exposición y gestión de dispositivos IoT sensores, mientras que la parte 2 [20] está orientada a la operación de dispositivos IoT que ofrecen acciones o actuación.

La *Sensing API* se organiza alrededor de los siguientes modelos o entidades:

- *Thing*: Un dispositivo físico o virtual integrado en una red de comunicación (e.g. un sensor de temperatura).
- *Location*: La última localización conocida de la Thing.
- *Historical Location*: Un histórico de tiempos asociados a la localización actual y la última localización de la Thing.
- *Datastream*: Un conjunto de muestras (*Observations*) distribuidas en un espacio de tiempo de un sensor (*Sensor*) para una propiedad (*ObservedProperty*) (e.g. una serie de medidas de temperatura).
- *Sensor*: Dispositivo que se utiliza para obtener una medida aproximada del fenómeno que se desea medir (e.g. un sensor de temperatura de un modelo específico).
- *ObservedProperty*: An ObservedProperty specifies the phenomenon of an Observation (e.g. temperatura).
- *Observation*: Una instancia de medida llevada a cabo por un *Sensor* para obtener un valor estimado para una *ObservedProperty* (e.g. un valor de temperatura).
- *FeatureOfInterest*: La característica destacable del contexto que engloba a un sensor que resulta diferenciadora o de interés para dar sentido a sus medidas (e.g. localización o area geográfica del sensor).

Un servidor SensorThings permite llevar a cabo operaciones CRUD sobre estas entidades a través de una API REST sobre HTTP.

Adicionalmente, y de manera opcional, SensorThings incluye la utilización del protocolo MQTT para habilitar patrones de comunicación *publish-subscribe* (verbo *Notify* que no estaba presente en los verbos CRUD ofrecidos

por la API HTTP). Esta extensión permite la publicación de nuevas *Observation* y la suscripción a actualizaciones sobre cualquiera de las entidades.

2.1.8. Smart Object framework

El framework *Smart Object* (SO) se describe en [21]. Comparativamente, resulta un ejercicio académico en mayor medida que otras especificaciones en esta sección, dado que no emana de consorcios u organizaciones de relevancia en la industria. Aun así, resulta de interés igualmente para tener un mayor contexto por ser un trabajo temprano en la línea temporal de la WoT. De hecho, los autores hacen referencia al hecho de que la WoT es un concepto que proporciona algunas guías de alto nivel, pero que no dispone de diseños concretos.

En este framework, un *nodo* es un dispositivo con un identificador único proporcionado por RFID, así como un conjunto de sensores que forman una unidad atómica. Un SO puede contener un número arbitrario de nodos.

Los SO son capaces de comunicarse entre ellos, y su localización puede evolucionar en el tiempo. Por esta razón el framework proporciona la capacidad de establecer redes ad-hoc. Las redes ad-hoc de SO están basadas en *clustering*, es decir, se selecciona un representante de un grupo de SO que tiene la responsabilidad de actuar como un proxy para las redes externas. Este proceso de *clustering* ocurre en dos niveles: en una red de SO, y en una red de nodos de un único SO. Los algoritmos de selección de proxy se basan en los niveles de batería restantes. Es interesante destacar que los SO no forman redes de manera indiscriminada, ya que necesitan un cierto contexto común para poder llevar a cabo este proceso.

Los autores llegaron a la conclusión de que 6LoWPAN, y otras soluciones basadas en IP para direccionamiento y enrutamiento, no eran adecuadas para este caso de uso. Esto es principalmente porque la mayoría de los nodos presentan recursos computacionales limitados, lo que les impide soportar la pila IP. Para solucionar esta limitación, se desarrolló un protocolo específico llamado *Sequence Chain* (SC). SC presenta un conjunto de propiedades deseables como el soporte de uniones y rupturas dinámicas de redes, un bajo coste computacional y la ausencia de una autoridad central para el mantenimiento y asignación de direcciones.

La denominada *Information Infrastructure* actúa como el gateway central para los SO. Las comunicaciones entre este componente y los SO se basan en dos tipos de eventos: los relacionados con actualizaciones de la estructura de la red, y los eventos del entorno capturados por los sensores de los SO. Los clientes se suscriben a la interfaz *query* para recepción de eventos de red, y a la interfaz *capture* para eventos de sensores.

Finalmente, los autores presentan una implementación experimental de la arquitectura SO. A modo de detalle de la pila de tecnologías, PostgreSQL actúa como el repositorio de datos de la red, y se utilizan SOAP Web Services con serialización en XML para las comunicaciones con los suscriptores de datos de sensores y estructura de la red.

2.2. Web of Things

El campo de la Web of Things se enfoca en trasladar las tecnologías de la Web al dominio del IoT. Es decir, en aplicar patrones y tecnologías de la Web tales como HTTP y REST para facilitar la interoperabilidad y la integración de dispositivos IoT.

En [5, 6, 7] Guinard y Trifa presentan los primeros trabajos claramente clasificados bajo la línea de la WoT. En este conjunto de publicaciones comentan la tendencia creciente de dispositivos embebidos con capacidades de conexión directa a Internet. Sin embargo, estos dispositivos resultan difíciles de controlar e integrar sin utilizar interfaces propietarias.

Se describe la necesidad de considerar a los dispositivos *smart* como ciudadanos de primera clase de la Web. En este caso, la WoT es un *refinamiento* del IoT a través de la integración de dispositivos inteligentes no solo en Internet (la capa de red) si no en la Web (la capa de aplicación).

Para llegar hasta este punto, proponen la aplicación del patrón arquitectural REST para exponer dispositivos en una API. Reconocen el mal encaje de HTTP para escenarios que necesitan comunicación push y mencionan proyectos como Comet, HTML5 SSE y Websockets. Describen dos soluciones para conseguir que los dispositivos alcancen la categoría de *inteligentes*:

- Conexión directa a la Web a través de un servidor Web embebido.
- Utilización de un proxy Web para los dispositivos que no tienen capacidades.

A modo de ejemplo de implementación concreto, se describen dos prototipos basados en estas ideas:

- Una solución basada en *Plogg*, un enchufe inteligente, conectada a un *smart gateway* para la construcción de una interface Web con la que gestionar los aparatos conectados.
- Una red inalámbrica de sensores basada en la plataforma SPOT de Sun. Cada dispositivo SPOT tiene un servidor Web embebido diseñado de acuerdo a la arquitectura REST. En este caso también está

presente un *smart gateway* para minimizar la comunicación directa de los dispositivos con la Web.

Los resultados finales de la evaluación sugieren que la sobrecarga introducida por el protocolo HTTP es aceptable cuando los dispositivos están conectados a una fuente estable de energía y los requerimientos de latencia se encuentran en el orden de cientos de milisegundos. El uso de HTTP introduce muchas ventajas que compensan por la pérdida de rendimiento.

Por otro lado, en uno de los artículos más relevantes y completos en este dominio, Raggett [22] argumenta el futuro crecimiento explosivo en el número de dispositivos conectados a Internet. Esto deriva en una evolución hacia una *Web of Things*, donde las *cosas* (Things) actúan como proxies de objetos físicos y virtuales.

Esto a su vez origina una necesidad de mercados basados en estándares abiertos. En el artículo se propone un framework abierto de WoT, que podría servir como un cimiento sólido para este mercado. El framework en cuestión está estrechamente relacionado con lo que en un futuro cercano serían las especificaciones de la WoT del W3C.

Las tecnologías existentes de la Web, incluyendo protocolos y lenguajes, pueden ser fácilmente aprovechados para construir la Web of Things. Los dispositivos inteligentes presentes en las viviendas, y las plataformas en la nube, podrían evolucionar para proporcionar servicios WoT. Además, los vocabularios actuales podrían utilizarse para describir las interfaces de los servicios, mientras que nuevos vocabularios podrían desarrollarse para cubrir las necesidades detectadas.

De manera adicional, los dispositivos IoT tienden a estar limitados en sus capacidades de computación y pueden no ser actualizables por software. En este caso los dispositivos inteligentes podrían servir como *barreras* actualizables. Otra cuestión importante es el hecho de que los usuarios no siempre van a estar presentes para tareas de autenticación cuando ocurran las interacciones con los dispositivos, así que se necesitará investigar maneras innovadoras de verificar la confianza y gestionar la autorización.

Dado el alto número de dispositivos, la aparición de errores es algo que debe considerarse en los diseños. Los servicios tienen que ser resistentes a fallos y caídas.

Finalmente, se destaca el hecho de que las Things pueden tener más de una representación, es decir, un *avatar*. Pueden existir múltiples representaciones en función del nivel de abstracción. Serán necesarios esfuerzos interdisciplinarios para que las Things se muestren y comporten de una manera más amigable a los usuarios humanos, así como para que incorporen *sentido común*.

En [23] Heuer et al. describen la importancia de entender apropiadamente los mecanismos de interacciones entre Things. En este sentido se identifican dos categorías principales de retos. La primera categoría está basada en que el patrón de arquitectura REST no es especialmente adecuado para modelar interacciones entre Things:

- REST solo soporta interacciones *request-response*. Sin embargo, las Things requieren de comunicaciones iniciadas en el lado del servidor en número notable de situaciones. Además, es difícil asignar el rol jerárquico de cliente y servidor en el contexto de interacciones de Things.
- REST no proporciona un mecanismo para descubrimiento de recursos.
- REST suele requerir documentación externa, y carece de una manera expresiva de describir recursos. *Hypermedia as the Engine of Application State* (HATEOAS) podría servir para mitigar esta problemática.
- REST se basa en interacciones punto a punto. En la WoT, las interacciones tienden a ser punto a multipunto.
- REST carece de un patrón de orquestación para gestionar múltiples flujos de datos interconectados entre Things.

La segunda categoría de retos es la de seguridad:

- Los mecanismos actuales de autenticación están basados sobre pares de usuario y password, así como el intercambio de certificados X.509. Estos patrones no escalan bien al volumen de billones de dispositivos que demanda la WoT.
- Los problemas de autorización en la WoT incluyen *O-to-O* y *O-to-**. La autorización *O-to-O* está actualmente resuelta por la especificación OAuth 2.0. Sin embargo, *O-to-** conlleva el problema de que el propietario debe estar presente cuando el objeto está siendo accedido, esto implica que las reglas deben ser definidas con anterioridad.

En la línea de gateways IoT para extensión de capacidades, se encuentran los resultados presentados en [24]. Los autores proponen un diseño de gateway basado en el framework de interoperabilidad IoT AllJoyn. Se hace una distinción explícita entre gateways *pasivos*, aquellos que requieren interacción manual para añadir o quitar dispositivos IoT del gateway; gateways *semi-automáticos*, aquellos que pueden descubrir automáticamente los dispositivos, pero requieren operación manual para tareas de configuración; y

finalmente, *automáticos*, que pueden descubrir y configurar dispositivos y servicios sin interacción por parte del usuario.

El gateway descrito en el artículo se categoriza como *automático*. Además, soporta 3G/4G, LTE, Wi-Fi, ZigBee, Bluetooth e IrDA. El entorno de pruebas es un despliegue a gran escala en la universidad Sungkyunkwan en Corea del Sur, incluyendo 200 sensores. El despliegue se basa en tres actores: un *servidor* de backend; los *dispositivos IoT* finales, que tienen soporte para redes móviles y WiFi; y el *gateway* que interconecta los dispositivos IoT con el backend.

La necesidad de gateways propietarios, como el comentado anteriormente, es una constante en un número significativo de implementaciones y casos de uso de la WoT. Este es un problema que va más allá y afecta también al IoT en general, originando problemas de escalabilidad. Sin embargo, en la Web se toma una aproximación diferente y más adecuada, en la que el uso de navegadores basados en estándares bien conocidos habilita el acceso ubicuo. Los resultados descritos en [25] son una estrategia para acometer este problema. En este trabajo los autores presentan una aproximación en la que los smartphones pueden utilizar como gateways oportunistas. Proponen la utilización de Bluetooth Low Energy (BLE) para las comunicaciones entre los dispositivos IoT y los smartphones, ya que es una tecnología con soporte extenso que resulta adecuada para su utilización en dispositivos de recursos limitados. Se identifican dos modos concretos de operación:

- El smartphone puede operar como un router IPv6. Esto proporciona el mayor nivel de flexibilidad pero requiere soportar la pila IPv6 en el dispositivo, lo cual puede no resultar la solución más óptima para dispositivos con recursos limitados de computación.

- El smartphone puede operar como un proxy BLE, de manera que se expongan los atributos Bluetooth del dispositivo a la nube.

Esta aproximación de gateways oportunistas tiene una serie de retos. Por un lado, existen problemas de privacidad que emanan del hecho de que smartphones arbitrarios pueden acceder a comunicaciones de dispositivos IoT. Además, se necesitaría incentivar a los usuarios con algún tipo de mecanismo o programa para que presten sus smartphones. Es decir, aunque resulta una idea interesante, es de difícil implementación.

Por otro lado, las conclusiones presentadas en [26] también son de particular interés para la WoT. En esta publicación se analiza el estado actual del IoT para verificar el nivel de integración con la Web, es decir, el nivel de madurez de la WoT. Para ello, los autores construyen un conjunto de datos

de dos millones de dispositivos conectados reales, para intentar identificar patrones y extraer conclusiones.

Aunque la Web está indexada actualmente por motores de búsqueda populares (Google, Bing), el IoT aún se encuentra pendiente de recibir un servicio de relevancia comparable. Además, el acceso a datos IoT de gran escala es todavía un problema complejo para programadores y usuarios finales. Estos problemas impactan negativamente en la visibilidad y la estabilidad del IoT desde un punto de vista global.

Los autores organizan los datos IoT actualmente existentes en tres categorías de alto nivel. En primer lugar, datos almacenados en *servicios IoT en la nube*, que presentan características variadas dependiendo del modelo de negocio (SaaS, PaaS, IaaS) pero comparten la idea de centralizar los recursos de computación y almacenamiento. En segundo lugar, contenidos en *plataformas WoT*. Finalmente, sistemas de *Web Mapping*, basadas en la visualización de datos IoT en un contexto geoespacial (por ejemplo, calidad del aire, actualizaciones de tráfico en tiempo real). Las conclusiones más significativas sobre las características de los datos IoT analizados son las siguientes:

- La mayoría de los sitios Web y Things actualmente existentes pertenecen a la categoría *Web Mapping*.
- Se puede identificar un grado de estacionalidad muy notable en la distribución geoespacial y los patrones de actualización de las Things.
- Los datos tienden a ser estáticos, es decir, no se actualizan con frecuencia. Sin embargo, existe mucha varianza entre diferentes Things.

Los retos futuros que se identifican para la WoT en [26] están alineados con las conclusiones de otros autores. Es decir, el descubrimiento automático de Things y repositorios de datos, y la escalabilidad para asegurar el mantenimiento factible de grandes cantidades de Things.

Con un mayor enfoque en el aspecto software de la WoT, [27] presenta un framework llamado SWoT4CPS. La motivación principal es añadir la capacidad de incluir anotaciones semánticas y razonamiento en Cyber-Physical System (CPS), de manera que se puedan proporcionar soluciones para dos limitaciones clásicas de los CPS. La primera es que los vocabularios específicos de dominio como la ontología W3C Semantic Sensor Network (SSN) no están preparadas para modelar relaciones colaborativas y causales en aplicaciones CPS. Por ejemplo, un evento es la causa de otro efecto que ha sido observado en otra parte del sistema. La segunda es que el conocimiento de dominio presenta un nivel alto de fragmentación, lo que complica la alineación de

términos. Por ejemplo, *Beijing* y *Peking* son equivalentes, y ambos están localizados en *China*. Se pueden identificar cuatro componentes diferenciados dentro del framework SWoT4CPS:

- El *SWoT-O Annotator*, basado en el vocabulario SWoT-O, una extensión a la ontología W3C SSN.
- El *EL Annotator*, que utiliza técnicas de machine learning para enlazar Things anotadas con la ontología SWoT-O con sentido común extraído de DBpedia.
- La *Knowledge Storage API*, implementada sobre Neo4j, una base de datos de grafos que soporta consultas SPARQL.
- Un módulo *Semantic Reasoner* basado en Apache Jena que soporta búsqueda semántica y razonamiento para la detección de anomalías.

Otro trabajo comparable en el dominio del software se puede ver en [28], donde se detalla un el servidor *Web of Virtual Things* (WoVT). Este componente software está diseñado para ser desplegado en la capa edge y dar solución al problema de la interoperabilidad, ofreciendo las siguientes funcionalidades:

- Una interfaz de comunicación basada en REST que soporta un conjunto diverso de protocolos de comunicación. Se define un componente denominado *Transport Handler* que abstrae los protocolos de transporte de capa baja (ZigBee, BLE, IP) entre el servidor WoVT y el dispositivo final. Las peticiones y respuestas se ajustan a un formato ad-hoc similar a HTTP o CoAP que soporta dos verbos (GET y POST).
- Un modelo semántico de Thing para representar entidades de manera que se puedan acomodar diferentes variaciones en formatos de descripción de Things. Concretamente, el módulo *Thing Factory* soporta los formatos de la W3C WoT, ISPO y OCF.
- Un repositorio de scripts que permite a reutilización de bloques comunes de programación de sensores. Los flujos de trabajo se programa utilizando un DSL y se distribuyen a las Things bajo petición.

El servidor WoVT demuestra una vez más los principios de abstracción del protocolo, reutilización de los patrones de la Web y mejoras semánticas que son característicos de la WoT en general y las especificaciones W3C WoT en particular.

De manera similar a WoVT, el diseño del framework WoT descrito en [29] también se asienta sobre REST. Sin embargo, se pone un mayor énfasis en proporcionar herramientas amigables para los desarrolladores. Particularmente, se pueden identificar tres módulos:

- Un modelo de datos en el que las Things se representan en forma de grafos y se exponen como *Web Resources*. Cualquier Thing puede estar compuesta por múltiples nodos, donde cada nodo tiene una URI. Los enlaces entre nodos pueden ser de dos tipos: de *agregación*, cuando un nodo está contenido en otro; y de *referencia*, cuando un nodo apunta a otro.

- Un componente middleware denominado IDN que proporciona soporte al modelo de datos previo. IDN está compuesto por cuatro capas (de alto nivel a bajo nivel): la capa *Virtual Resource* que expone las Things como recursos REST en HTTP; la capa *Information History* para trazabilidad y versionado de los nodos; la capa *Storage Interface* para persistencia; y la capa *Adapter* para integración con recursos externos en la Web, que se encuentran fuera del framework.

- Una colección de librerías para interactuar con IDN, así como componentes de interfaz gráfica para representación visual de los recursos.

La validación del framework de [29] se llevó a cabo sobre una aplicación *mashup* experimental en el contexto del proyecto SmartSantander [30]. Otra implementación de la WoT en un caso de uso real se describe en [31]. Esta referencia define los *Ambient Spaces* (AS) como una representación virtual de un conjunto de Things habilitadas para la Web. Los AS están normalmente presentes siguiendo el mismo patrón en localizaciones que comparten el mismo contexto (por ejemplo, sensores de parking en una plaza). Este concepto se utiliza como el cimiento, junto con un nuevo modelo semántico basado en OWL, para construir un sistema de *Smart Parking Spots* (SPS). La arquitectura de los SPS es razonablemente clásica en el sentido de que se puede identificar un gateway y un servidor Web central; las comunicaciones están basadas en HTTP y se llevan a cabo sobre un canal WiFi.

En [32] se presenta un sistema basado parcialmente en una versión temprana de las especificaciones W3C Web of Things (véase sección 2.2.1). Los autores argumentan que los servicios en la nube utilizados para exponer funcionalidades de Things a los usuarios finales no suelen reaccionar adecuadamente a los cambios en su entorno. Proponen una plataforma para exposición dinámica de Things a través de una API Web localizada en la nube.

Dentro de la arquitectura propuesta, los dispositivos utilizan un gateway WoT para registrarse en un repositorio de Thing Descriptions (TD). Las Things pueden ser descubiertas en ese repositorio por un módulo *Device observer*. Otro módulo denominado *Device exposer* reconoce y expone las funcionalidades de manera automática a través de la interfaz `ExposedThing`, permitiendo que los clientes remotos accedan a las interacciones de las Things. El sistema es capaz de reaccionar en tiempo real a cambios, por ejemplo, Things que se desconectan repentinamente.

Desde el punto de vista del usuario final, el sistema proporciona un módulo *UI Composer* para la construcción de interfaces de usuario dinámicas, basándose en componentes reutilizables (*UI Parts*) que se pueden descargar de un repositorio central. Esto es posible gracias a las APIs estandarizadas que proporciona la W3C WoT, que permiten descubrir dinámicamente las funcionalidades de las Things.

El middleware IoT, es decir, software para la creación de aplicaciones IoT a través de la composición de bloques parametrizables y reutilizables, está estrechamente relacionado con la WoT. De hecho, se puede argumentar que el middleware IoT es uno de los bloques más relevantes para la construcción de la WoT. En este sentido, en [33] se muestra una vista general del estado actual del middleware IoT, proponiendo una clasificación dependiendo en el tipo de arquitectura y describiendo ejemplos existentes. Los tipos de arquitectura identificados son los siguientes:

Service-based Basado en el patrón arquitectural *Service-oriented Architecture* (SoA) y compuesto por tres capas: *física*, de *servicio* y de *aplicación*. La capa de servicio expone un conjunto amplio de servicios (por ejemplo, control de acceso, recolección de datos) que encapsulan la capa física. Esta capa de servicio se despliega normalmente en la nube y requiere un nivel significativo de capacidades de computación, por lo que es incompatible con dispositivos limitados en recursos.

Cloud-based Esta es normalmente la arquitectura de plataformas IoT propietarias. El proveedor expone en este caso una serie de servicios en la nube (por ejemplo, almacenamiento escalable) en la forma de APIs con las que el desarrollador IoT debe integrarse. La infraestructura hardware que está por debajo suele estar normalmente oculta y el proveedor tiene el control ultimo sobre los datos del usuario.

Actor-based En este caso se identifican tres capas concéntricas: sensores y actuadores, gateways, y recursos en la nube. Los recursos de computación están distribuidos por toda la arquitectura, el software suele ser razonablemente ligero y puede desplegarse en todas las capas, lo que

tiene un impacto positivo en la latencia y escalabilidad. Esta categoría tiene una conexión directa con los principios del edge computing (véase sección 2.3).

Resulta también de interés que los autores de [33] mencionan explícitamente una serie de retos futuros que deben ser afrontados por el middleware IoT. Principalmente, mejorar la flexibilidad del middleware para facilitar la composición y personalización de aplicaciones IoT a diferentes contextos, mejorar el descubrimiento automático e incrementar la seguridad de las transacciones y privacidad de los datos de usuario. Precisamente, todos estos retos son parte de los objetivos perseguidos por la WoT, especialmente en el dominio de las especificaciones W3C WoT.

Finalmente, el estudio del diseño y rendimiento de protocolos de capa de aplicación es otro campo relevante en la WoT. Aunque HTTP es indudablemente el protocolo de la Web, y en el que se centran la mayoría de trabajos, se deben considerar además otros protocolos de relevancia en el IoT.

CoAP, un protocolo basado en el diseño de HTTP, pero diseñado específicamente para dispositivos con bajos recursos computacionales, es uno de los principales protocolos de capa de aplicación en el dominio de la WoT. CoAP usa DTLS para la encriptación punto a punto. Existe una problemática en que DTLS tiene que ser utilizado con cuidado en el contexto de redes de sensores debido al impacto notable en el rendimiento. Esto es especialmente así en redes con múltiples saltos cuando DTLS se implementa en los dispositivos finales.

Para acometer los problemas de rendimiento de DTLS, los autores de [34] presentan un gateway llamado Secure Service Proxy (SSP). Se sitúa entre dispositivos finales CoAP y el otro extremo de la sesión DTLS, sirviendo como un proxy DTLS y proporcionando funcionalidad adicional de capa de aplicación. El SSP está diseñado alrededor del concepto de *virtual devices* (VD). Un SSP expone varios VD, donde cada VD contiene un grupo de recursos, y cada recurso representa un par de protocolos de capa de transporte y capa de aplicación (UDP/CoAP, TLS/HTTP). Varias cadenas de adaptadores de capa de aplicación, como por ejemplo sistemas de control de acceso o cachés, pueden ser definidas en cada VD. Los resultados muestran que integrar el SSP en un despliegue IoT puede tener un impacto positivo en la batería y una reducción de carga en los dispositivos de capacidades limitadas.

Continuando dentro del contexto de CoAP, en [35] se describe una arquitectura basada en un proxy CoAP situado entre las aplicaciones web y un conjunto de sensores de recursos limitados. Esto resulta de interés porque aunque no se menciona explícitamente el paradigma WoT, los principios de diseño son notablemente similares. El proxy en cuestión utiliza un mecanis-

mo de caché que mantiene muestras recientes en una base de datos local para reducir la carga de los sensores. Los sensores mandan peticiones CoAP POST con los valores actualizados de las variables siguiendo una distribución de tiempo exponencial. Además, el proxy envía peticiones CoAP GET a los sensores que exponen las variables físicas en el caso de que el valor en caché haya caducado. Este patrón, es decir, la combinación de sondear sensores bajo demanda con métodos GET a la vez que se reciben valores desde los sensores con métodos POST, se denomina *Hybrid Proxy*.

En la línea de estudio de protocolos de capa de aplicación, se presenta una revisión y comparativa en [36]. La revisión considera CoAP, MQTT, XMPP, HTTP (REST), AMQP y Websockets. Este estudio vuelve a incidir sobre la dependencia de CoAP en DTLS para tareas de encriptación, lo que en cierta medida disminuye su utilidad por el coste computacional de DTLS. Además, se resalta también que Websockets es una opción flexible para un número significativo de casos de uso, excepto en aquellas situaciones en las que los recursos computacionales son excesivamente limitados. Una de las conclusiones principales es que el diseño de un gateway capaz de seleccionar el protocolo más adecuado dependiendo del contexto sería una contribución de valor para el IoT. Precisamente, esta es una de las funcionalidades que serían posteriormente incluidas en las especificaciones W3C WoT.

En [37] se toma otra aproximación en el sentido de que los autores se centran más en el estudio del rendimiento de los protocolos y formatos de serialización, en vez de en sus características y casos de uso. Se consideran tres plataformas de desarrollo: Adobe Flash, Microsoft Silverlight y HTML5. Aunque solamente HTML5 es de interés para el caso de la WoT, las conclusiones resultan razonablemente trasladables, y por lo tanto de interés. Para la comparativa de rendimiento se describen dos aplicaciones experimentales. La primera consiste en un gateway situado en la capa edge, que proporciona datos en tiempo real a una aplicación Web utilizando el patrón de mensajes push. En la segunda aplicación, el gateway y la aplicación Web se comunican a través de un broker de mensajes de acuerdo al patrón de mensajes *publish-subscribe*. Las conclusiones más relevantes para la WoT que se extraen del estudio son las siguientes:

- HTTP long-polling y Websockets, ambos con serialización JSON, demuestran buenos resultados de latencia.
- JSON es una solución razonable en todas las plataformas. Sin embargo, Google Protocol Buffers produce mensajes de menor tamaño. Aun así, JSON tiene mejor soporte y menores barreras de entrada.
- La latencia de MQTT es la mejor en todos los casos en los que se pone

a prueba. La tasa de transferencia también es razonablemente buena, aunque AMQP tiene mejores resultados en algunos casos.

2.2.1. Especificaciones W3C Web of Things

En esta sección se presenta una introducción a los conceptos y el estado actual de la W3C Web of Things. Estos conceptos son el cimiento de los resultados incluidos en este trabajo, principalmente del diseño del framework de aplicaciones WoT del capítulo 3.

La recomendación **W3C WoT Architecture** [8] establece las bases de la Web of Things interpretada por el *W3C WoT Working Group*, que incluye a representantes de organizaciones como Fujitsu, Siemens e Intel. Se define la terminología básica y se identifican casos de uso relevantes, así como los requerimientos y principios comunes de un sistema WoT. Además, se describen las piezas fundamentales con las que se pueden construir despliegues WoT. Estas piezas se comentan con más detalle en los siguientes párrafos.

La **Thing** es el elemento principal de la WoT. Una Thing es cualquier entidad física o virtual que tiene presencia en un sistema WoT. Una Thing expone funcionalidades en forma de *interacciones* y se describe de manera completa a través de un documento Thing Description (ambos conceptos se comentan posteriormente). Es importante destacar que una Thing no tiene que tener una relación *uno-a-uno* con elementos físicos. Una Thing puede ser un *avatar* virtual a modo de abstracción de múltiples entidades. Por ejemplo, una Thing `alumbrado` que agrega a todas las Thing `bombilla` de una estancia.

Un **Servient** es otro de los actores principales de un despliegue WoT. Un Servient implementa los *building blocks* (bloques fundamentales) y actúa a modo de cliente y/o servidor WoT. Actuar como cliente WoT implica *consumir* Things, es decir, manipular una instancia de `ConsumedThing` que sirve de representación de una Thing remota. Las instancias de la abstracción `ConsumedThing` se construyen a partir de documentos Thing Description. Por otra parte, actuar como servidor WoT implica *exponer* Things, es decir, crear instancias locales de `ExposedThing` que ofrecen las funcionalidades de la Thing al despliegue, que a su vez se ven representadas en un documento Thing Description.

Todas las posibles funcionalidades de una Thing se representan de acuerdo al conocido como **Interaction Model** (modelo de interacciones). Las interacciones son abstracciones de alto nivel que representan todos los posibles *comportamientos* de las aplicaciones y sistemas IoT. Se definen tres tipos de interacción:

Propiedades Atributos o valores del estado de una Thing que pueden ser leídos y/o escritos. Por ejemplo, la temperatura de un sensor, el estado de una persiana automatizada o la velocidad a la que se mueve un vehículo.

Acciones Procedimientos de duración *significativa* que son invocados sobre la Thing. A diferencia de la lectura de una propiedad, el retorno del resultado de una acción no es un proceso síncrono. Por ejemplo, bajar una persiana automatizada, reiniciar una máquina virtual o lanzar una petición a una API HTTP.

Eventos Ocurrencias observadas por la Thing que son comunicadas a los clientes que tienen una suscripción activa. Por ejemplo, que una aspiradora robot se ha conectado a su punto de carga, que la temperatura ha subido por encima de un nivel o que un servicio HTTP ha empezado a fallar sus *healthcheck*.

Los documentos **Thing Description (TD)** [38] son la descripción *exhaustiva* de una Thing. En una TD se indican las interacciones de una Thing (propiedades, acciones y eventos) y todos los metadatos que son necesarios para su caracterización, por ejemplo, el nombre, el identificador único y el tipo. El concepto de *Linked Data* está integrado como elemento de primer orden en la W3C WoT, es decir, se pueden utilizar vocabularios bien conocidos para la definición de los metadatos en los documentos TD. La adopción de los principios *Linked Data* es relevante porque permite añadir conocimiento de sentido común. Además, bajo cada interacción en una TD se definen los parámetros de acceso particulares de cada protocolo de capa de aplicación. Una misma interacción, que es un concepto de alto nivel, puede estar expuesta a través de múltiples protocolos de bajo nivel.

El proceso de diseño de una Thing según el modelo W3C WoT implica identificar todas las funcionalidades públicas que deben ser expuestas. Posteriormente, las funcionalidades se modelan en la forma de propiedades, acciones o eventos, de manera que se pueda construir su documento Thing Description. La posesión de un documento Thing Description y de las credenciales de autorización (si fuesen necesarias) es suficiente para interactuar de manera plena con la Thing sin necesidad de conocimiento previo o intercambio de información por otro canal.

El modelo de datos de Thing Description puede ser serializado en formato *JSON for Linking Data* (JSON-LD) [39], que permite representar un grafo RDF en un formato similar al bien conocido y popular formato JSON. También existe la opción de utilizar una serialización *JSON simple* para los

casos en los que resulta beneficioso el compromiso de evitar la inclusión de tecnologías semánticas.

En el bloque de código 2.1 se muestra un ejemplo completo de un documento TD serializado en formato JSON simple de acuerdo al modelo de datos de TD *draft* disponible en [40]. Para mayor claridad, se muestra un ejemplo sencillo con una única interacción.

Aunque está definido como un componente opcional, el bloque fundamental **W3C WoT Scripting API** [41] tiene una función instrumental en el ecosistema WoT: permitir que los desarrolladores expongan y consuman Things con mínima inversión de esfuerzo. La Scripting API es la interfaz, o capa de alto nivel, que se asienta sobre el *WoT Runtime* en el que se ejecutan las aplicaciones WoT. Esta API abstrae de la mayoría de los detalles de bajo nivel de la *WoT Interface*, es decir, de la interfaz de red descrita por un documento Thing Description. En los escenarios en los que no está disponible un WoT Runtime que expone la Scripting API, los desarrolladores se ven obligados a proporcionar implementaciones de las comunicaciones de bajo nivel y de la lógica de manipulación de documentos Thing Description, lo cual puede suponer una barrera de entrada inabarcable. En otras palabras, la Scripting API es un mecanismo para facilitar y promover la adopción de la W3C WoT.

El bloque de código 2.2 muestra un ejemplo de aplicación WoT que utiliza la *Scripting API* expuesta por un *WoT Runtime*. Concretamente, es un ejemplo (recortado por brevedad) desarrollado con la implementación experimental [42] de los diseños propuestos en el capítulo 3:

La *nota informativa* **WoT Binding Templates** [43] proporciona una guía para trasladar el modelo de interacciones a comunicaciones de red basadas en un protocolo particular de capa de aplicación o un ecosistema IoT. Una *implementación de Protocol Binding* describe de manera concreta y explícita los mensajes que se intercambian para cumplir los requerimientos de las interacciones de alto nivel. Por ejemplo, la lectura de una propiedad podría representarse como una petición HTTP GET en una implementación HTTP, o como un mensaje publicado en un *topic* predeterminado en una implementación MQTT. Este es uno de los componentes más abiertos de la W3C WoT, es decir, la guía proporcionada por el W3C no es *normativa*. Una implementación de los bloques fundamentales W3C WoT debe tomar múltiples decisiones de diseño para llegar a un conjunto interoperable de implementaciones de Protocol Binding.

Finalmente, para facilitar la organización de conceptos W3C WoT, la figura 2.1 muestra una comparativa simple entre arquitecturas ilustrativas de un despliegue IoT basado en el patrón edge computing y un sistema que adopta la arquitectura WoT.

```
{
  "title": "urn:temperaturething",
  "properties": {
    "temperature": {
      "forms": [
        {
          "href": "ws://192.168.1.166:9393/urn-temperaturething
             ↪ -5b7f17a9-e1ab-1f53-b0a7-ad52b2d44116",
          "contentType": "application/json"
        },
        {
          "href": "http://192.168.1.166:9494/urn-
             ↪ temperaturething-5b7f17a9-e1ab-1f53-b0a7-
             ↪ ad52b2d44116/property/temperature",
          "op": ["readproperty", "writeproperty"],
          "contentType": "application/json"
        },
        {
          "href": "http://192.168.1.166:9494/urn-
             ↪ temperaturething-5b7f17a9-e1ab-1f53-b0a7-
             ↪ ad52b2d44116/property/temperature/subscription",
          "op": ["observeproperty"],
          "contentType": "application/json"
        }
      ],
      "observable": true,
      "type": "number",
      "readOnly": true
    }
  },
  "events": {},
  "security": [{ "scheme": "nosec" }],
  "actions": {},
  "id": "urn:temperaturething",
  "links": [],
  "base": "http://192.168.1.166:9494/urn-temperaturething-5
     ↪ b7f17a9-e1ab-1f53-b0a7-ad52b2d44116"
}
```

Código 2.1: Ejemplo de Thing Description

```
ws_server = WebsocketServer(port=81)
http_server = HTTPServer(port=80)
servient = Servient(catalogue_port=9090)
servient.add_server(ws_server)
servient.add_server(http_server)
wot = await servient.start()

td_doc = {
    "id": "urn:org:fundacionctic:thing:cpumonitor",
    "name": "CPU Monitor Thing",
    "properties": {
        "cpuPercent": {
            "description": "Current CPU usage",
            "type": "number",
            "readOnly": True,
            "observable": True
        },
        "cpuThreshold": {
            "description": "CPU usage alert threshold",
            "type": "number",
            "observable": True
        }
    }
}

exposed_thing = wot.produce(json.dumps(td_doc))
exposed_thing.set_property_read_handler("cpuPercent",
    ↪ cpu_percent_handler)
exposed_thing.expose()
```

Código 2.2: Extracto de ejemplo de aplicación WoT sobre Scripting API

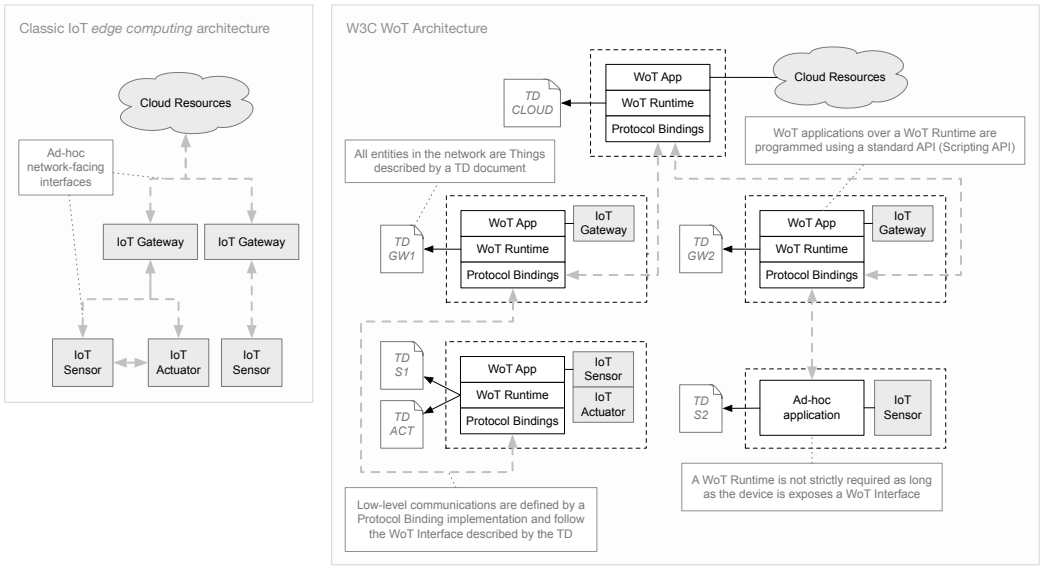


Figura 2.1: Comparativa de arquitecturas IoT y WoT

2.3. Edge computing

El *edge computing* es un patrón de arquitectura aplicable al dominio del IoT. Las bases de esta arquitectura fueron presentadas por primera vez por Flavio Bonomi et al. [2] bajo el nombre de *fog computing*. En esta popular referencia, se describe el fog computing como una plataforma capaz de exponer servicios en cualquier capa entre las fuentes de datos IoT y los centros de datos en el núcleo. Concretamente, se pueden identificar tres servicios de alto nivel: la *red*, la *computación* y el *almacenamiento*. La innovación en este caso emana del hecho de que estos servicios se consideraban hasta el momento nativos a la nube.

La distinción entre edge computing y fog computing tiende a ser difusa. El patrón fog computing puede ser considerado como el ejemplo principal, e incluso una evolución, del patrón edge computing. El fog computing implementa la idea principal del edge computing, es decir, la redistribución de recursos de la nube a localizaciones adyacentes a los dispositivos finales IoT. Sin embargo, no está restringido a esa frontera, y puede existir en cualquiera de las capas superiores que están más cerca de la nube.

Es importante destacar que a lo largo de este documento se usa por norma el término *edge computing* como una forma general de alto nivel que engloba la arquitectura *fog computing*.

El patrón edge computing es una respuesta a un conjunto de retos que

han ido tomando relevancia en el dominio del IoT con el paso del tiempo. El problema principal son los altos volúmenes de datos que se generan en los dispositivos IoT. Esto a su vez deriva en presión sobre la red e incremento de costes, que en muchos casos es inasumible, y un empeoramiento de la latencia en las aplicaciones IoT. Los problemas de latencia pueden llegar hasta el punto en el que aplicaciones en dominios con altas restricciones (e.g. seguridad) resultan inviables.

De manera más general, los despliegues IoT presentan una serie de demandas para las cuales la arquitectura edge computing proporciona beneficios tangibles [3]:

- **Transmisión de datos**: El edge computing puede facilitar la optimización del compromiso entre la *latencia de computación*, mínima en las capas altas de la nube, y la *latencia de transmisión*, mínima en las capas bajas de dispositivos IoT. El ancho de banda y los costes de transmisión también se pueden reducir, junto con el gasto energético y la sobrecarga de transmisión que se deriva de despliegues IoT complejos.

- **Almacenamiento**: Los servidores en capa edge se pueden utilizar para replicar y balancear grandes volúmenes de datos generados por los dispositivos IoT. Estos conjuntos de datos tendrían si no que almacenarse de manera centralizada en la nube.

- **Computación**: El edge computing puede proporcionar una mayor flexibilidad para la distribución de tareas entre capas. La distribución puede llevarse en cabo en función de la prioridad, las políticas de precios y las demandas de capacidad de computación.

Desde el punto de vista de arquitectura, los despliegues IoT basados en el patrón edge computing tienden a estar divididos en tres capas diferenciadas. Estas capas son, en orden de mayor a menor cercanía a los dispositivos IoT finales:

- La capa *front-end* o capa **de sensores**, donde la latencia es mínima y la computación viene proporcionada por los dispositivos finales, que normalmente están muy limitados en recursos.

- La capa *near-end* o capa **edge**, con dispositivos que presentan limitaciones menos significativas en términos de computación y almacenamiento. Gracias a esto, se pueden conseguir niveles aceptables de latencia. Además, estos dispositivos tienden a utilizar canales de comunicación inalámbricos, y son heterogéneos en cuanto a factores de forma y especificaciones.

- La capa *far-end* o capa **nube**, que ofrece recursos de computación prácticamente ilimitados para las tareas más pesadas (e.g. machine learning, procesamiento batch). Sin embargo, esta capa tiene la mayor latencia de todas.

La implementación de edge computing basada en tres capas tiende a ser la más común, y recibe en algunos casos el nombre de *modelo jerárquico*. En este modelo los dispositivos se clasifican en función de sus recursos y latencia, que vienen determinados por su proximidad física a la nube. Por ejemplo, en [44] se presenta un gateway para implementación de edge computing usando en el modelo jerárquico. Existe no obstante otra aproximación para la implementación del patrón edge computing basada en los avances recientes en Software Defined Networking (SDN). Las tecnologías SDN proporcionan capacidades de configuración dinámica de red, que pueden ser explotadas para alcanzar la optimización de latencia y consumo de recursos computacionales que es el objetivo final del edge computing.

Sin embargo, se pueden identificar retos notables que deben ser afrontados para que el patrón edge computing pueda llegar a su mayor potencial. Estos retos incluyen, principalmente, las dificultades que emanan de una heterogeneidad excesiva en los componentes hardware (i.e. diferentes plataformas, lenguajes y protocolos); la optimización de la utilización de recursos; la monitorización del estado del despliegue y la implementación de mecanismos de *self-healing*; y finalmente, las garantías de seguridad y privacidad.

2.3.1. Simulación de aplicaciones IoT

En esta sección se describen referencias relevantes en el dominio de la simulación y emulación de arquitecturas y aplicaciones IoT. De manera directamente relacionada, también se incluyen referencias más cercanas al cloud computing. Las herramientas, que se pueden ver resumidas en la tabla 2.3, se clasifican en función de si son capaces de ejecutar código real, o por el contrario operan con modelos abstractos de alto nivel. Además, también se pone el foco en las capacidades de escalabilidad horizontal, es decir, en la capacidad que tienen las herramientas para afrontar problemas de mayor volumen a través de la inclusión de más nodos hardware. Esto es en contraste con la escalabilidad vertical, que se basa en incrementar los recursos computacionales de un nodo individual.

Los contenedores son un bloque fundamental de una parte significativa de las herramientas de simulación estudiadas. Los contenedores son una tecnología basada en primitivas del kernel de Linux (e.g. *cgroups*) que permiten la ejecución aislada y reproducible de procesos. Aunque existen paralelismos

Herramienta	Detalle	Código real	Escalabilidad horizontal
CloudSim [45]	Simulador de cloud computing con un alto volumen de referencias	No	*No aplica*
ContainerCloudSim [46]	Extensión de CloudSim que habilita la simulación de contenedores	No	*No aplica*
iFogSim [47]	Simulador para arquitecturas fog computing basado en CloudSim	No	*No aplica*
CloudSim Plus [48]	Fork de CloudSim con nuevas funcionalidades y una API mejorada	No	*No aplica*
YAFS [49]	Simulador para arquitecturas edge computing que puede reaccionar dinámicamente a eventos de simulación	No	*No aplica*
Mininet [50]	Emulador de red para la creación de entornos experimentales realistas.	Sí	No
MaxiNet [51]	Extensión de Mininet para la ejecución basada en hosts virtuales Docker sobre múltiples máquinas	Sí	Configuración manual en casos no triviales
Containernet [52]	Fork de Mininet que añade soporte para hosts virtuales Docker	Sí	No
EmuFog [53]	Herramienta para optimización de arquitecturas basadas en edge computing representadas en forma de experimentos MaxiNet	Sí	Mismas limitaciones que MaxiNet
Fogbed [54]	Emulación de arquitecturas basadas en edge computing representadas con Containernet y MaxiNet	Sí	Mismas limitaciones que MaxiNet

Cuadro 2.3: Herramientas de simulación y emulación relacionadas con IoT

entre las bien conocidas máquinas virtuales y los contenedores, la diferencia principal reside en que los contenedores comparten el mismo kernel del sistema operativo, lo que hace que resulten comparativamente más ligeros. Docker [55] es probablemente el producto más popular en el espacio de los contenedores, ya que proporciona implementaciones maduras de todos los bloques fundamentales; por ejemplo, un repositorio de imágenes, un runtime y una herramienta para orquestación (Docker swarm mode). Independientemente de Docker, es importante destacar que existen alternativas tales como el runtime *crun* de Red Hat [56].

Los orquestadores de contenedores son, a su vez, herramientas que permiten la gestión de aplicaciones basadas en contenedores dentro de entornos con múltiples nodos hardware. Un orquestador es capaz de gestionar el ciclo de vida de los contenedores de manera distribuida. Kubernetes [57] es, junto con el previamente mencionado Docker swarm mode, el representante más extendido de las herramientas de orquestación.

Es razonable argumentar que la naturaleza dinámica del edge computing, y sus similitudes y relación directa con el cloud computing, hacen que las tecnologías de contenedores puedan aportar beneficios en tareas de mantenimiento, desarrollo y despliegue [58].

CloudSim [45] es una de las referencias más relevantes en el dominio de la simulación. Esta herramienta software proporciona una API para el modelado de arquitecturas de cloud computing, definiendo entidades tales como centros de datos y máquinas virtuales. Esto permite la representación de una amplia variedad de escenarios, lo que permite a su vez la simulación repetible en tiempo contenido para problemas de tamaño razonablemente grande. ContainerCloudSim [46] aparece como una extensión a CloudSim que añade el concepto de contenedores, permitiendo la simulación de trabajos basados en contenedores, o, dicho de otra manera, la simulación de arquitecturas cloud en términos del modelo Containers as a Service (CaaS).

CloudSim es el cimiento de un conjunto de herramientas de simulación que proporcionan funcionalidad extendida en otros dominios distintos al cloud computing. iFogSim [47], una herramienta para la simulación de aplicaciones IoT basadas en el modelo edge computing, es uno de los ejemplos más relevantes. En iFogSim las aplicaciones IoT se representan en forma de grafos dirigidos basándose en el modelo Distributed Dataflow (DDF) [59]. Los vértices de estos grafos se corresponden con tres tipos distintos de entidades: *sensors*, que generan mensajes de acuerdo a una distribución predeterminada de transmisión; *actuators*, que actúan como los puntos de recepción de los mensajes; y *modules*, una clase que simboliza cualquier algoritmo o paso en una cadena de procesamiento de una aplicación IoT. Las aristas del grafo representan los mensajes intercambiados entre estas entidades, y vienen de-

finidos por sus parámetros de costes de computación y red. Los *application loops* representan a su vez el flujo de mensajes en la aplicación IoT y deben ser inicializados explícitamente para permitir que iFogSim monitorice las interacciones de interés. Finalmente, los usuarios deben definir un conjunto de *fog devices*, indicando sus capacidades de CPU, memoria y consumo de energía; estas entidades serán las responsables de ejecutar los *modules*, cuyo emparejamiento puede ser manual o automatizado.

En [48] se presenta un fork alternativo a CloudSim denominado CloudSim Plus. La motivación principal de este fork es la mejora de los estándares de calidad del código, adoptando para ello buenas prácticas de la ingeniería del software. Esto se traduce en programas más legibles y con menos volumen de código en comparación con CloudSim, lo que repercute positivamente en el mantenimiento y desarrollo. Además, también se incluyen nuevas funcionalidades exclusivas que no están presentes en CloudSim, por ejemplo, la posibilidad de definir estrategias automáticas para el escalado vertical y horizontal de máquinas virtuales.

A diferencia de las referencias previas, YAFS [49] es un simulador para arquitecturas basadas en edge computing que está construido fuera del ecosistema de CloudSim. Sin embargo, es bastante similar en su aproximación al modelado de aplicaciones, ya que las aplicaciones de YAFS se representan también como grafos DDF. Una de sus contribuciones más significativas es la posibilidad de definir algoritmos ad-hoc que se ejecutan en paralelo con la simulación. Esto permite a los usuarios la actualización automática del sistema en respuesta a eventos de simulación, por ejemplo, para el despliegue de nuevas entidades.

Dentro del dominio de las Software-Defined Networks (SDN) se pueden encontrar proyectos estrechamente relacionados con la emulación de escenarios edge computing. Mininet [50] es un proyecto relevante en esta categoría. Mininet está construido sobre primitivas de Linux (e.g. *network namespaces*) y permite la emulación, con comportamiento realista, de redes con cientos de entidades (e.g. hosts virtuales, switches, enlaces) en una sola máquina física. MaxiNet [51] es una herramienta relacionada, diseñada para atajar particularmente esta limitación de Mininet en la que los experimentos deben limitarse a una máquina. Con este fin, MaxiNet se vale del protocolo Generic Routing Encapsulation (GRE) para construir túneles IP-sobre-IP que puedan conectar nodos Mininet. Además, tiene soporte para definir hosts virtuales en forma de contenedores Docker. Sin embargo, en casos complejos suele ser necesario definir de manera manual el posicionamiento de los contenedores, es decir, el emparejamiento entre nodos físicos y hosts virtuales. De manera paralela, también es interesante destacar que el fork Containernet [52] permite añadir soporte Docker a experimentos Mininet sin la necesidad

de MaxiNet.

EmuFog [53] está construido sobre MaxiNet y proporciona funcionalidad para la optimización del diseño de arquitecturas edge computing. En EmuFog los usuarios pueden definir un conjunto de clientes (*device nodes*) y un catálogo de dispositivos edge (*fog nodes*) en términos de CPU, memoria y latencia. A modo de entrada, el usuario proporciona una topología de red en formato BRITE [60], después, se lleva a cabo un proceso de optimización que genera un experimento MaxiNet compuesto por una selección de nodos que cumple las restricciones del usuario. Aunque tiene soporte para contenedores Docker, este soporte está basado en MaxiNet, por lo que comparte las mismas limitaciones (i.e. posicionamiento manual).

De manera similar a EmuFog, Fogbed [54] se vale de Containernet y MaxiNet para ofrecer capacidades de emulación de escenarios edge computing en entornos locales y distribuidos respectivamente. Una de las funcionalidades más interesantes de Fogbed es su capacidad para actualizar de manera dinámica la topología de un experimento en curso.

Estas herramientas han demostrado beneficios tangibles en el proceso de diseño y optimización de despliegues IoT. Aun así, a modo de conclusión, se pueden detectar dos limitaciones claras. La primera es que, aunque las herramientas de simulación permiten alcanzar un tamaño de problema razonablemente alto, requieren que los usuarios modelen sus aplicaciones en términos de un conjunto limitado de parámetros teóricos. Esto es notablemente complicado, y puede derivar en simulaciones que se alejan de la realidad. La segunda limitación es que las herramientas de emulación encontradas presentan incógnitas significativas en lo que a escalabilidad horizontal se refiere. Esto quiere decir que aunque las herramientas de emulación pueden ejecutar código real, presentan complicaciones notables a la hora de emular escenarios de gran volumen.

2.4. Motivación

Las especificaciones W3C WoT representan una oportunidad para la mejora de los problemas de interoperabilidad que son comunes a día de hoy en los despliegues y aplicaciones IoT. La W3C WoT no persigue sustituir alternativas previas, sino diseñar un ecosistema en el que haya cabida para todas estas iniciativas. Además, se pretende conseguir esto a través del reaprovechamiento de tecnologías Web, que resultan uno de los dominios con mayor conocimiento y soporte en el contexto del desarrollo software y la tecnología en general.

Aun así, la W3C WoT se encuentra aún en su infancia. Las especifica-

ciones no describen de manera explícita todos los detalles de diseño de sus bloques fundamentales, lo que genera varias incógnitas y barreras de entrada. Es por esto que es necesario llevar a cabo implementaciones experimentales que permitan sentar las bases para desarrollos de aplicaciones WoT reales. Esto a su vez debería servir de punto de partida para la adopción de la WoT.

Primera motivación

El diseño de un runtime W3C WoT experimental plenamente funcional que incorpore todos los bloques fundamentales propuestos por el W3C WoT Working Group.

La primera motivación presenta retos significativos y requiere la toma de varias decisiones de diseño. De manera particular, se destacan las siguientes contribuciones derivadas:

- Proporcionar una serie de diseños concretos para implementaciones de W3C WoT Protocol Bindings que cubran el conjunto completo de los protocolos de capa de aplicación recomendados por las especificaciones (i.e. MQTT, Websockets, HTTP y CoAP).
- Evaluar de manera exhaustiva el rendimiento del runtime W3C WoT en general, y las implementaciones de Protocol Bindings en particular. La evaluación se debe llevar a cabo en el contexto de un conjunto representativo de escenarios de despliegue.

Por otro lado, la arquitectura edge computing es otra tendencia muy notable en el dominio del IoT. A diferencia de las especificaciones W3C WoT, el patrón edge computing no persigue directamente solucionar problemas de interoperabilidad, sino proporcionar herramientas para permitir la optimización de recursos computacionales, de red y de almacenamiento en aplicaciones IoT.

Es razonable argumentar que ambas piezas (arquitectura edge computing y W3C WoT) deberían ser consideradas en el diseño de aplicaciones IoT modernas. Un estudio exhaustivo del impacto de la W3C WoT en el dominio del IoT requiere la consideración del patrón edge computing.

Segunda motivación

La evaluación del impacto de la arquitectura edge computing en el contexto de las aplicaciones WoT. Concretamente, se persigue diseñar una herramienta de emulación de sistemas IoT/WoT que permita el estudio del rendimiento de propuestas de arquitecturas.

La herramienta en cuestión debería ser suficientemente versátil para representar despliegues IoT basados en arquitectura edge computing. De manera más específica, las motivaciones de esta herramienta de emulación son las siguientes:

- Proporcionar una herramienta que permita que los desarrolladores de software IoT puedan probar sus propuestas de arquitectura utilizando código real de producción. Esto debería resultar más accesible que el diseño de escenarios teóricos utilizando herramientas de simulación más específicas.
- Exponer métricas de rendimiento centradas en la aplicación que resulten de mayor utilidad para los desarrolladores como complemento a las métricas de bajo nivel del sistema (e.g. uso de CPU) que son expuestas por otras herramientas de emulación y simulación.
- Aprovechar las capacidades ofrecidas por herramientas modernas de orquestación de contenedores para proporcionar escalabilidad horizontal a la emulación de arquitecturas IoT. Esta aproximación ofrece un compromiso razonable entre el realismo de utilizar entornos experimentales con hardware realista y el tamaño de experimento alcanzable con herramientas de simulación teórica.
- Integrar el runtime W3C WoT como ciudadano de primer orden dentro de la herramienta. De esta manera, se espera poder avanzar en la validación y adopción de las especificaciones W3C WoT posibilitando que los usuarios puedan representar todas las entidades de la arquitectura bajo prueba en forma de WoT Things.

Capítulo 3

Framework para desarrollo de aplicaciones Web of Things

Este capítulo describe los detalles del diseño y la lógica detrás de la implementación experimental del framework para desarrollo de aplicaciones WoT cuya motivación se presenta en la sección 2.4. El diseño está basado principalmente en los cimientos establecidos por los borradores de las especificaciones W3C WoT [61, 62, 40]. Sin embargo, estas especificaciones no proporcionan una guía detallada de implementación, por lo que existe la necesidad de tomar decisiones de diseño concretas para alcanzar un estado funcional que pueda probarse experimentalmente. El foco se pone precisamente en estas decisiones, mientras que los detalles generales de las especificaciones W3C WoT pueden consultarse con mayor detalle en la sección 2.2.1.

Existe una implementación experimental plenamente funcional que está públicamente disponible en Zenodo [63]. Además, el proyecto **WoTPy** también está publicado en GitHub [42].

Los trabajos en la línea del framework de aplicaciones WoT han dado lugar a una publicación [64] que puede verse adjunta en el apéndice A.

3.1. Arquitectura

La figura 3.1 muestra la arquitectura del framework propuesto, incluyendo todos los componentes relevantes y las relaciones entre ellos. La discusión presentada en los siguientes párrafos se organiza en torno a esta arquitectura.

El módulo de **Thing Description** (TD) contiene la lógica para validar documentos TD de acuerdo con la especificación del W3C [40]. 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 mó-

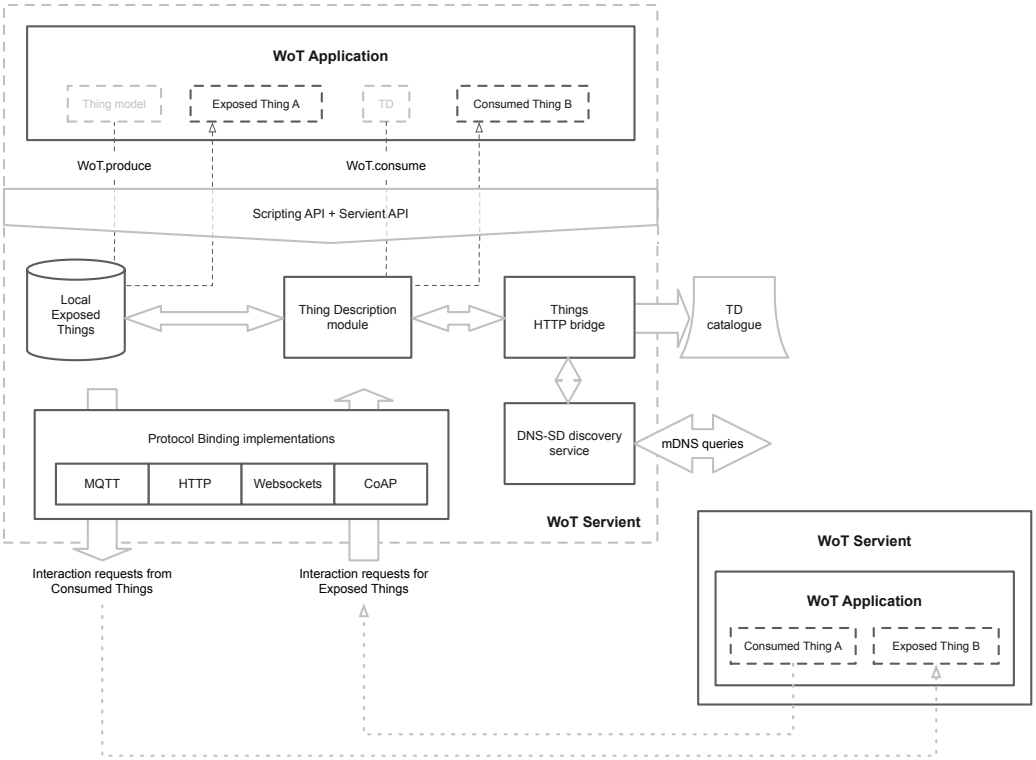


Figura 3.1: Arquitectura del framework WoT

dulo también tiene la capacidad de serializar Things internas a representación textual para su transmisión a través de la red (y viceversa). Se opta por utilizar la serialización *simple* en formato JSON [65] en vez de la serialización JSON-LD [39] debido a la inestabilidad de la especificación en el momento de diseño.

El módulo de **implementaciones de Protocol Bindings** (véase sección 3.2) contiene a su vez cuatro sub módulos que implementan una interfaz común que se expone de manera interna al resto de componentes del framework. El Servient, que es la entidad 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.

En este contexto, se denomina **aplicación WoT** (*WoT Application*) a los programas que gestionan Things y están desplegados sobre un WoT runtime. El runtime es a su vez la capa que contiene al Servient e implementa los detalles de orquestación de Things. La acción de descubrimiento es uno de los primeros pasos en este proceso de gestión (i.e. consumición y exposición) de Things, y para ello se habilitan dos métodos:

- Un método *local* que restringe la búsqueda a las Things registradas en el propio Servient.

- Un método *multicast* basado en las tecnologías DNS Service Discovery (DNS-SD) [66] y Multicast DNS (mDNS) [67]. El protocolo mDNS es una versión descentralizada de DNS en el que los nodos intercambian información acerca de los nombres de dominio usando mensajes multicast. Por su parte DNS-SD especifica cómo utilizar DNS para buscar y registrar servicios en una red local, y se utiliza normalmente en conjunción con mDNS para permitir el descubrimiento dinámico de servicios. Los servicios DNS-SD se identifican por un identificador público de servicio previamente conocido. En este caso, el identificador único es `_wot-servient._tcp.local`.

De manera estrechamente relacionada con el subsistema de descubrimiento, los Servient dependen del módulo **Things HTTP Bridge** para exponer el catálogo interno de Things **Thing Description Catalogue** 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-LD 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. Es decir, el catálogo es el punto de entrada principal al Servient para todos los consumidores externos, excepto en aquellos casos menos comunes en los que la URL de la Thing se conoce explícitamente de manera previa.

Para mayor claridad, en la figura 3.2 se pueden ver las relaciones de alto nivel entre las entidades Thing, las implementaciones de Protocol Binding y el módulo de descubrimiento. De manera más específica, las entidades `ThingModel` y `ThingDescription` son, respectivamente, los objetos que se utilizan para construir `ExposedThing` y `ConsumedThing`. Esta es precisamente la funcionalidad que expone el punto de entrada a la Scripting API, es decir, la clase `WoT`:

- El método `WoT.produce` es el punto de entrada para construcción de *Exposed Things* a partir de *Thing Models* proporcionados por el desarrollador.

- El método `WoT.consume` es el mecanismo para generación de *Consumed Things* a partir de *Thing Descriptions* recuperadas de la red a través de un proceso de descubrimiento, localmente del catálogo interno, o proporcionadas explícitamente por el usuario.

Las Exposed Thing se anuncian públicamente sobre el módulo de *Discovery* utilizando el protocolo Multicast DNS (mDNS). El módulo de descubrimiento es invocado a su vez por las Consumed Thing para la construcción local de las Thing Description. Es destacable que las Exposed Thing y Consumed Thing solo implementan el modelo de interacciones de alto nivel (acciones, propiedades y eventos) las interacciones de bajo nivel en la red son traducidas y gestionadas por el componente de implementaciones de Protocol Binding.

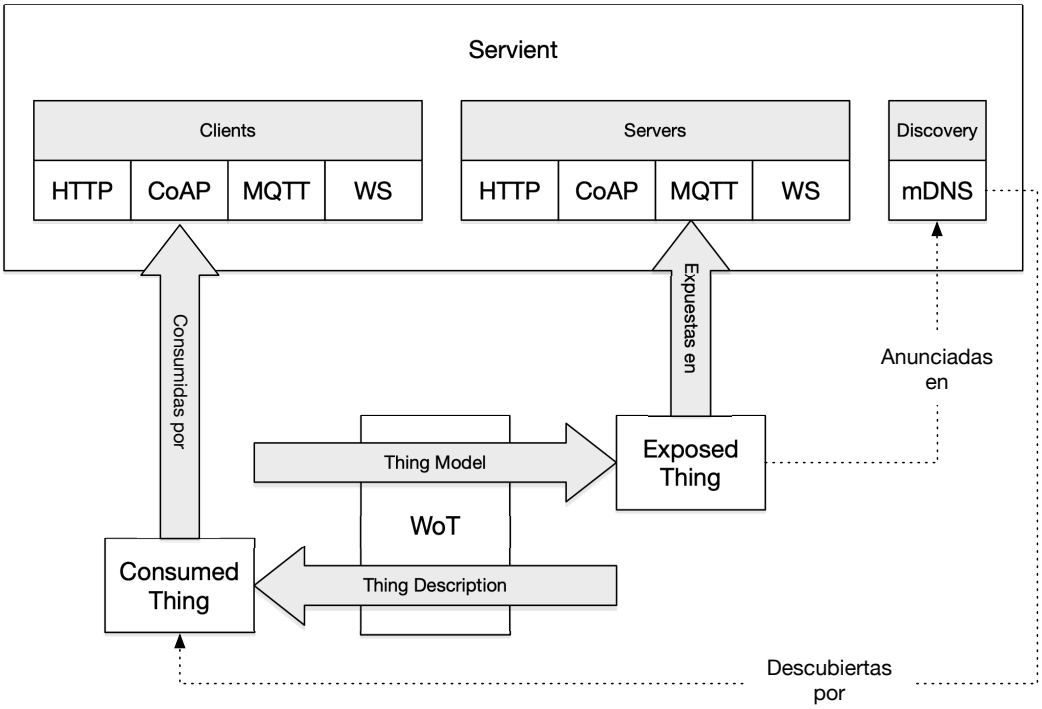


Figura 3.2: Relación entre Things e implementaciones de Protocol Binding

Además, se pueden identificar dos APIs diferenciadas que se exponen

para que las aplicaciones WoT puedan interactuar con las capas inferiores del Servient:

Servient API Esta interfaz no viene definida en las especificaciones y se ha diseñado explícitamente para la implementación experimental del framework. Permite que el programador interaccione con la base de datos local que contiene las Things, así como configurar los clientes y servidores del Servient.

Scripting API Interfaz común definida en las especificaciones [62] que encapsula el funcionamiento interno del WoT runtime. Proporciona un objeto WoT central que sirve como punto de entrada para consumir Things a partir de documentos TD; exponer Things creadas localmente; y descubrir nuevas Things en el entorno. Así mismo define la interfaz de las Consumed y las Exposed Things.

Profundizando en las particularidades del diseño y el desarrollo, se puede destacar que la API del framework 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 [68]. 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. Tornado es un framework de terceras partes mientras que `asyncio` es un módulo oficialmente soportado del núcleo de Python, por lo que el uso de Tornado en Python 2.7 es una opción menos adecuada que `asyncio`, 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 asíncrono. 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 entidades similares a una función denominados *coroutines*. Una *coroutine* se define y comporta igual que una función con una excepción notable: dentro de una *coroutine* se puede utilizar `await` o `yield` para invocar a otra *coroutine* de manera asíncrona. Cuando se invoca un *coroutine* de esta manera, se produce el salto dentro

del loop que se comentaba anteriormente. Cuando la *coroutine* 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 del framework) pero no resultan tan adecuadas para cargas de trabajo computacionalmente costosas (después de todo, solo se dispone de un hilo).

La tabla 3.1 contiene más información al respecto de los paquetes software que se utilizan como cimiento para las implementaciones experimentales de Protocol Binding. Además, está disponible una implementación experimental plenamente funcional de los diseños descritos en este capítulo en los repositorios públicos Zenodo [63] y GitHub [42].

Protocolo	Paquete Python
HTTP	`tornado` [68]
Websockets	`tornado` [68]
CoAP	`aiocoap` [69]
MQTT	`hbmqtt` [70]

Cuadro 3.1: Paquetes base de las implementaciones Protocol Binding

3.2. Implementaciones de Protocol Binding

En esta sección se describe la relación entre las interacciones de alto nivel que se pueden ejecutar sobre una Thing y los mensajes intercambiados entre clientes y servidores WoT dentro del contexto de las implementaciones de Protocol Binding. Existe una implementación particular para cada protocolo, cada una de ellas presenta un diseño particularizado que traduce el modelo de interacciones a los detalles y características del protocolo en cuestión.

Todas las implementaciones específicas de protocolos, como se puede ver en las secciones a continuación, implementan los verbos de interacción especificados en el *draft* del documento de Thing Description [40]. Estos verbos están descritos en detalle en la tabla 3.2.

Verbo	Interacción	Descripción
`readproperty`	Property	Lectura del valor de una propiedad.
`writeproperty`	Property	Actualización del valor de una propiedad.
`observeproperty`	Property	Suscripción a actualizaciones de una propiedad.
`invokeaction`	Action	Invocación de una acción.
`subscribeevent`	Event	Suscripción a emisiones de un evento.
`unsubscribeevent`	Event	Destrucción de una suscripción de evento activa.

Cuadro 3.2: Verbos de interacción implementados en el framework WoT

3.2.1. HTTP

HTTP es un protocolo sin estado basado 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 arquitectural REST [71], por lo que esta es la aproximación tomada en el framework.

HTTP/1.1 no se ajusta bien a los casos de uso que requieren comunicación iniciada desde el servidor, lo cual es un requerimiento básico de un conjunto de los verbos de interacción del modelo W3C WoT. Afortunadamente, la popularidad de HTTP ha dado lugar a un conjunto de patrones y buenas prácticas que intentan cubrir estos puntos débiles. En esta categoría, el patrón *long-polling* es una de las soluciones más extendidas dentro de HTTP/1.1 para mensajes *push*.

La implementación de Protocol Binding para HTTP adopta *long-polling* para solucionar el problema de mensajes del lado del servidor. En la práctica, esto significa que el servidor mantiene la conexión abierta hasta que se genera una respuesta en peticiones de invocación de acción, suscripción de propiedad y suscripción de eventos. Cabe destacar que las peticiones no se pueden mantener indefinidamente abiertas, por lo que se define un timeout global. Aunque *long-polling* es una solución aceptable, no resulta óptima, por lo que el WoT runtime tiende a evitar la implementación HTTP para los verbos de interacción mencionados previamente siempre que no está requerido de manera explícita por el usuario.

Se identifican cinco recursos HTTP diferenciados que se pueden ver a continuación. Los verbos de interacción se mapean a métodos HTTP de acuerdo con las directrices de la arquitectura 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.

La tabla 3.3 muestra el detalle de la estructura de los elementos *form* dentro de la implementación HTTP.

Campo	Descripción
`op`	Verbo de interacción (e.g. `readproperty`) asociado con este elemento *form*.
`href`	URL HTTP en la que el servidor HTTP del WoT runtime expone esta interacción.
`mediaType`	Este campo siempre contiene el tipo MIME de JSON.

Cuadro 3.3: Estructura de elementos *form* en implementación HTTP

En el resto de esta sección se muestran los detalles de bajo nivel de la implementación aplicados al modelo de interacciones WoT. Todos los mensajes se serializan en formato JSON.

Read Property

Estructura del elemento *form*:

```
{
    "op": "readproperty",
    "contentType": "application/json",
    "href": "http://<host>:<port>/<thing_name>/property/<
       ↪ property_name>"
}
```

Código 3.1: Binding HTTP: Elemento *form Read Property*

Formato de la petición HTTP:

```
GET http://<host>:<port>/<thing_name>/property/<property_name>
```

Código 3.2: Binding HTTP: Petición *Read Property*

Formato de la respuesta HTTP:

```
HTTP 200

{
    "value": <property_value>
}
```

Código 3.3: Binding HTTP: Respuesta *Read Property*

Write Property

Estructura del elemento *form*:

```
{
    "op": "writeproperty",
    "contentType": "application/json",
    "href": "http://<host>:<port>/<thing_name>/property/<
       ↪ property_name>"
}
```

Código 3.4: Binding HTTP: Elemento *form Write Property*

Formato de la petición HTTP:

```
PUT http://<host>:<port>/<thing_name>/property/<property_name>

{
    "value": <property_value>
}
```

Código 3.5: Binding HTTP: Petición *Write Property*

Formato de la respuesta HTTP:

```
HTTP 200
```

Código 3.6: Binding HTTP: Respuesta *Write Property*

Invoke Action

Estructura del elemento *form*:

```
{
    "op": "invokeaction",
    "contentType": "application/json",
    "href": "http://<host>:<port>/<thing_name>/action/<
       ↪ action_name>"
}
```

Código 3.7: Binding HTTP: Elemento *form Invoke Action*

Las invocaciones de acción se lanzan utilizando el verbo HTTP POST:

```
POST http://<host>:<port>/<thing_name>/action/<action_name>

{
    "input": <action_argument>
}
```

Código 3.8: Binding HTTP: Petición lanzamiento *Invoke Action*

Un identificador único UUID se asigna en este caso a la invocación en curso. El identificador en cuestión se devuelve en el cuerpo de la respuesta a la petición POST anterior:

```
HTTP 200

{
    "invocation": "/invocation/<uuid>"
}
```

Código 3.9: Binding HTTP: Respuesta con ID de invocación

La URL que viene definida por el identificador único representa el recurso HTTP que describe el estado actual de la invocación:

```
GET http://<host>:<port>/invocation/<uuid>
```

Código 3.10: Binding HTTP: URL de invocación

Una invocación puede encontrarse en tres estados diferenciados:

- Actualmente en curso. El campo `done` es `false`.
- Finalizada con éxito. El campo `done` es `true` y `result` contiene el resultado devuelto.
- Finalizada con error. El campo `done` es `true` y `error` contiene la representación textual del error capturado.

```
HTTP 200

{
    "done": <boolean>,
    "result" <result_value>,
    "error": <error_message>
}
```

Código 3.11: Binding HTTP: Cuerpo de estado de invocación

Observe Property

Estructura del elemento *form*:

```
{
    "op": "observeproperty",
    "contentType": "application/json",
    "href": "http://<host>:<port>/<thing_name>/property/<
       ↪ property_name>/subscription"
}
```

Código 3.12: Binding HTTP: Elemento *form Observe Property*

Las suscripciones se gestionan de manera automática dentro del contexto de la implementación HTTP. Es decir, el usuario no tiene la responsabilidad de destruir la suscripción ni recordar el identificador de una suscripción activa. Una petición al recurso que se puede ver a continuación da lugar a la inicialización de una suscripción, que se cancela una vez que se emite un valor:

```
GET http://<host>:<port>/<thing_name>/property/<property_name>/
   ↪ subscription
```

Código 3.13: Binding HTTP: URL de suscripción *Observe Property*

En este caso, el formato de la respuesta es el mismo que para el verbo de lectura del valor de una propiedad (*read property*):

```
HTTP 200

{
    "value": <property_value>
}
```

Código 3.14: Binding HTTP: Respuesta *Observe Property*

Observe Event

Estructura del elemento *form*:

```
{
    "op": "subscribeevent",
    "contentType": "application/json",
    "href": "http://<host>:<port>/<thing_name>/event/<event_name
       ↪ >/subscription"
}
```

Código 3.15: Binding HTTP: Elemento *form Observe Event*

Formato de la petición HTTP:

```
GET http://<host>:<port>/<thing_name>/event/<event_name>/
   ↪ subscription
```

Código 3.16: Binding HTTP: URL de suscripción *Observe Event*

De manera equivalente a lo que ocurre con la invocación de acciones y la suscripción a propiedades, el servidor HTTP implementa *long-polling* para mantener la conexión abierta hasta que se genera un evento:

```
HTTP 200

{
    "payload": <event_payload>
}
```

Código 3.17: Binding HTTP: Cuerpo de emisión de evento

3.2.2. Websockets

Websockets es un protocolo de comunicación bidireccional soportado en todos los navegadores modernos que 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 [72]. Se identifican cuatro tipos de mensaje que pueden ser intercambiados entre cliente y servidor durante una sesión.

Los mensajes **Request** son enviados por los clientes para iniciar la llamada a un procedimiento, conteniendo 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*.

```
{
    "jsonrpc": "2.0",
    "method": <method_name>,
    "params": <method_params>,
    "id": <message_id>
}
```

Código 3.18: Binding WS: Estructura mensaje *Request*

- `method`: Identificador del método bajo petición.
- `params`: Parámetros de esta petición en concreto.
- `id`: Identificador único del mensaje. El mensaje *response* asociado con esta *request* contendrá el mismo identificador.

Los mensajes **Response** son enviados por el servidor y contienen las respuestas de las llamadas realizadas en peticiones *request*.

```
{
    "jsonrpc": "2.0",
    "result": <request_result>,
    "id": <message_id>
}
```

Código 3.19: Binding WS: Estructura mensaje *Response*

- `result`: Resultado de la petición original.
- `id`: Identificador único del mensaje. Si la *request* original no contenía un identificador, este campo estará vacío.

Los mensajes **Error** son un tipo específico de respuesta que genera el servidor cuando surge un error durante el procesamiento de una *request*. Contienen detalles del error en vez del resultado de la llamada.

```
{
    "jsonrpc": "2.0",
    "error": {
        "code": <error_code>,
        "message": <error_message>,
        "data": <error_data>
    },
    "id": <message_id>
}
```

Código 3.20: Binding WS: Estructura mensaje *Error*

- `error.code`: Código numérico que identifica el error.
- `error.message`: Mensaje descriptivo del error.
- `error.data`: Datos arbitrarios asociados al error.
- `id`: Identificador único del mensaje. Si la *request* original no contenía un identificador, este campo estará vacío.

Los mensajes **Emitted Item** son mensajes que 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).

```
{
    "subscription": <subscription_id>,
    "name": <event_name>,
    "data": <event_payload>
}
```

Código 3.21: Binding WS: Estructura mensaje *Emitted Item*

- `subscription`: Identificador único de la suscripción enlazada a este item.
- `name`: Nombre del evento.
- `data`: Payload asociado al evento emitido.

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.

Debe destacarse que las implementaciones del protocolo Websockets, descrita en esta sección 3.2.2, y el protocolo MQTT, descrita en la sección 3.2.3, utilizan un *pool* de conexiones compartidas con el objetivo de optimizar su rendimiento. La alternativa simple en este caso sería inicializar una nueva conexión en cada interacción (e.g. lectura de propiedad, invocación de acción) lo que conllevaría un uso excesivo y poco óptimo de los recursos de computación de la máquina. La figura 3.3 muestra un diagrama de la lógica que representa el funcionamiento del *pool* compartido de conexiones.

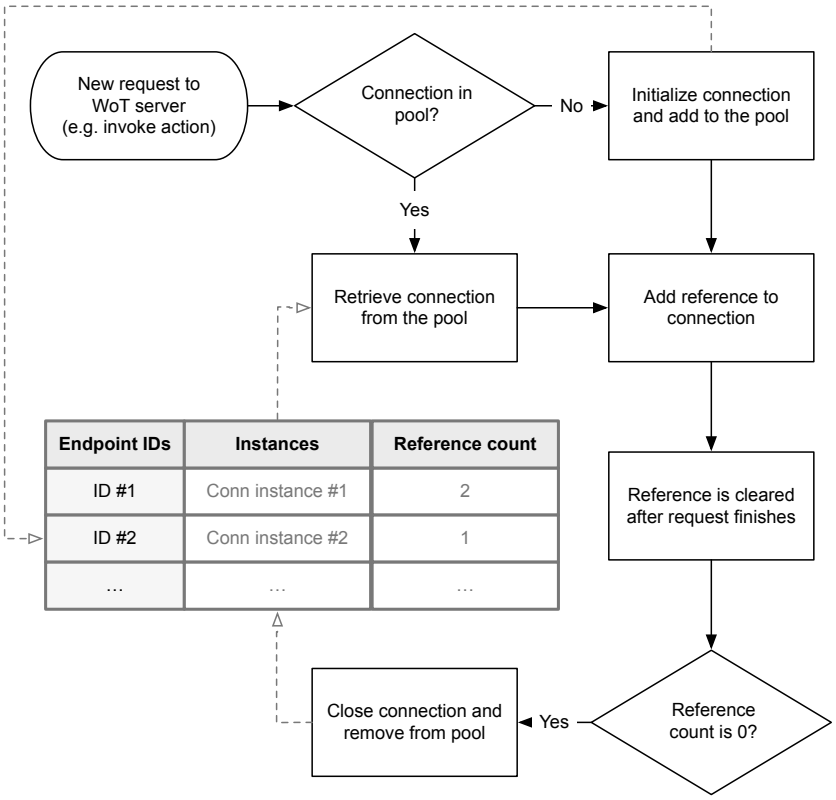


Figura 3.3: *Pool* de conexiones compartidas en protocolos con estado

Por otra parte, la tabla 3.4 muestra el detalle de la estructura de los elementos *form* dentro de la implementación Websockets.

Campo	Descripción
`href`	URL en la que el servidor Websockets escucha por peticiones. Esta URL es la misma para todas las interacciones de una Thing.
`mediaType`	Este campo siempre contiene el tipo MIME de JSON.

Cuadro 3.4: Estructura de elementos *form* en implementación Websockets

En el resto de esta sección se describen los detalles de bajo nivel de la implementación aplicados al modelo de interacciones WoT.

Read Property

Estructura del cuerpo de la petición:

```
{
    "jsonrpc": "2.0",
    "method": "read_property",
    "params": {
        "name": <property_name>
    },
    "id": <request_uuid>
}
```

Código 3.22: Binding WS: Estructura petición *Read Property*

Estructura del cuerpo de la respuesta:

```
{
    "jsonrpc": "2.0",
    "result": <property_value>,
    "id": <request_uuid>
}
```

Código 3.23: Binding WS: Estructura respuesta *Read Property*

Write Property

Estructura del cuerpo de la petición:

```
{
    "jsonrpc": "2.0",
    "method": "write_property",
    "params": {
        "name": <property_name>,
        "value": <property_value>
    },
    "id": <request_uuid>
}
```

Código 3.24: Binding WS: Estructura petición *Write Property*

Estructura del cuerpo de la respuesta:

```
{
    "jsonrpc": "2.0",
    "result": null,
    "id": <request_uuid>
}
```

Código 3.25: Binding WS: Estructura respuesta *Write Property*

El valor de `result` siempre será `null` para indicar que la actualización de la propiedad fue exitosa.

Invoke Action

Estructura del cuerpo de la petición:

```
{
    "jsonrpc": "2.0",
    "method": "invoke_action",
    "params": {
        "name": <action_name>,
        "parameters": <invocation_parameters>
    },
    "id": <request_uuid>
}
```

Código 3.26: Binding WS: Estructura petición *Invoke Action*

Estructura del cuerpo de la respuesta:

```
{
    "jsonrpc": "2.0",
    "result": <action_result>,
    "id": <request_uuid>
}
```

Código 3.27: Binding WS: Estructura respuesta *Invoke Action*

Observe Property

Estructura del cuerpo de la petición:

```
{
    "jsonrpc": "2.0",
    "method": "on_property_change",
    "params": {
        "name": <property_name>
    },
    "id": <request_uuid>
}
```

Código 3.28: Binding WS: Estructura petición *Observe Property*

Estructura del cuerpo de la respuesta:

```
{
    "jsonrpc": "2.0",
    "result": <subscription_id>,
    "id": <request_uuid>
}
```

Código 3.29: Binding WS: Respuesta con suscripción *Observe Property*

El servidor envía un mensaje de la siguiente forma para cada actualización de la propiedad:

```
{
    "subscription": <subscription_id>,
    "name": "propertychange",
    "data": {
        "name": <property_name>,
        "value": <property_value>
    }
}
```

Código 3.30: Binding WS: Mensaje de actualización de propiedad

Observe Event

Estructura del cuerpo de la petición:

```
{
    "jsonrpc": "2.0",
    "method": "on_event",
    "params": {
        "name": <event_name>
    },
    "id": <request_uuid>
}
```

Código 3.31: Binding WS: Estructura petición *Observe Event*

Estructura del cuerpo de la respuesta:

```
{
    "jsonrpc": "2.0",
    "result": <subscription_id>,
    "id": <request_uuid>
}
```

Código 3.32: Binding WS: Respuesta con suscripción *Observe Event*

El servidor envía un mensaje de la siguiente forma para cada evento emitido:

```
{
    "subscription": <subscription_id>,
    "name": <event_name>,
    "data": <event_payload>
}
```

Código 3.33: Binding WS: Mensaje de emisión de evento

Dispose Subscription

La implementación Websockets demanda que el usuario gestione manualmente las suscripciones, esto es en contraste con la implementación HTTP descrita en la sección 3.2.1, que lleva a cabo una gestión automática (aunque notablemente menos óptima). En este caso la estructura del cuerpo de la petición es la siguiente:

```
{
    "jsonrpc": "2.0",
    "method": "dispose",
    "params": {
        "subscription": <subscription_id>
    },
    "id": <request_uuid>
}
```

Código 3.34: Binding WS: Estructura petición *Dispose Subscription*

Estructura del cuerpo de la respuesta:

```
{
    "jsonrpc": "2.0",
    "result": <subscription_id>,
    "id": <request_uuid>
}
```

Código 3.35: Binding WS: Estructura respuesta *Dispose Subscription*

El valor de `result` puede contener `null` en aquellos casos en los que el identificador de la suscripción es desconocido.

3.2.3. MQTT

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 (request-response) 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 diferencia de otras implementaciones de Protocol Binding, que incluyen un servidor auto-contenido que no requiere dependencias externas, la implementación MQTT no proporciona un broker incrustado. Es decir, la implementación MQTT depende en un broker externo. Las razones detrás de esta decisión de diseño son las siguientes:

- Los brokers MQTT son agnósticos al formato de los mensajes publicados por clientes y no requieren programación específica para la aplicación en una mayoría de los casos.
- La probabilidad de disponer de un broker reutilizable en el contexto de un despliegue IoT es significativa, dada la popularidad del protocolo MQTT [1].
- Existen múltiples implementaciones de brokers MQTT de código abierto y calidad contrastada, por ejemplo, Mosquitto [73].

La tabla 3.5 muestra la lista de tópicos de la implementación MQTT. La parte denotada como `<sid>` (*Servient ID*) representa el identificador del WoT runtime obtenido a partir del nombre de la máquina. Este actúa como un espacio de nombres para evitar colisiones dentro del mismo broker.

Tópico	Formato
Property request	`<sid>/property/requests/<thing>/<property>`
Property update	`<sid>/property/updates/<thing>/<property>`
Property write ACK	`<sid>/property/ack/<thing>/<property>`
Action invocation	`<sid>/action/invocation/<thing>/<action>`
Action result	`<sid>/action/result/<thing>/<action>`
Event emission	`<sid>/event/<thing>/<event>`

Cuadro 3.5: Tópicos de la implementación MQTT Protocol Binding

Una característica interesante de esta implementación es que no hay necesidad de gestionar el ciclo de vida de las suscripciones para observar eventos y propiedades. El servidor MQTT mantiene una suscripción global compartida por todos los clientes, que no tienen la responsabilidad de crear y destruir las suscripciones de manera proactiva.

La tabla 3.6 muestra el detalle de la estructura de los elementos *form* dentro de la implementación MQTT.

Campo	Descripción
`op`	Verbo de interacción asociado con este elemento.
`href`	Contiene la URL del broker MQTT unida con el ID único del WoT runtime y el nombre del tópico MQTT. Seguidamente se describe este campo con mayor detalle
`mediaType`	Este campo siempre contiene el tipo MIME de JSON.

Cuadro 3.6: Estructura de elementos *form* en implementación MQTT

A continuación se puede ver un ejemplo ilustrativo del campo `href`, desglosado en todos sus componentes:

```
mqtt://my.mqtt.broker:1883/my-servient/property/requests/
    ↪ benchmark-thing/currenttime
```

Código 3.36: Binding MQTT: Estructura de URL en `href`

- `my.mqtt.broker:1883` es la URL del broker MQTT.
- `my-servient` es el identificador único del WoT runtime (*servient*) que se utiliza como espacio de nombres para evitar colisiones dentro del mismo broker.
- `property/requests/benchmark-thing/currenttime` es el tópico en el que los mensajes son intercambiados para esta interacción y verbo específicos.

En el resto de esta sección se muestran los detalles de bajo nivel de la implementación aplicados al modelo de interacciones WoT.

Read Property

El cliente tiene la opción de publicar un mensaje en el tópico *property request* para forzar al servidor a publicar el valor actual de la propiedad:

```
{
    "action": "read"
}
```

Código 3.37: Binding MQTT: Mensaje para forzar lectura de propiedad

En este caso, el servidor publica el valor de la propiedad en el broker bajo el tópico *property update*:

```
{
    "value": <property_value>,
    "timestamp": <unix_timestamp_ms>
}
```

Código 3.38: Binding MQTT: Mensaje de actualización de propiedad

Write Property

Para actualizar el valor de una propiedad, el cliente puede publicar un mensaje en el tópico *property request* con el siguiente formato:

```
{
    "action": "write",
    "value": <property_value>,
    "ack": <unique_ack_handler>
}
```

Código 3.39: Binding MQTT: Mensaje para escritura de propiedad

El servidor validará la petición de escritura publicando un mensaje en el tópico *property write ACK*. Para identificar de manera unívoca la petición, se utiliza el valor del campo `<unique_ack_handler>`:

```
{
    "ack": <unique_ack_handler>
}
```

Código 3.40: Binding MQTT: Mensaje ACK escritura propiedad

Invoke Action

Las acciones pueden ser invocadas a través de la publicación de un mensaje en el tópico de *action invocation*:

```
{
    "id": <unique_invocation_handler>,
    "input": <action_argument>
}
```

Código 3.41: Binding MQTT: Mensaje de invocación de acción

El resultado de la invocación se publicará en el tópico *action result* una vez que haya finalizado:

```
{
    "id": <unique_invocation_handler>,
    "timestamp": <unix_timestamp_ms>,
    "result" <result_value>,
    "error": <error_message>
}
```

Código 3.42: Binding MQTT: Mensaje de resultado de invocación

Este tipo de interacciones es uno de los escenarios más adecuados para MQTT. El patrón de invocación de acciones encaja especialmente bien dentro del patrón publish-subscribe de MQTT.

Observe Property

Como se menciona en la descripción del verbo *read property*, todas las actualizaciones de propiedad se publican automáticamente en el tópico *property update* sin ninguna intervención activa por parte del cliente. El cliente sólamente debe suscribirse al tópico para recibir los mensajes:

```
{
    "value": <property_value>,
    "timestamp": <unix_timestamp_ms>
}
```

Código 3.43: Binding MQTT: Mensaje de actualización de propiedad

Una vez más, MQTT proporciona mecanismos que se adaptan de manera directa a la observación de eventos sin intervención del cliente.

Observe Event

De manera similar a las interacciones anteriores, todas las emisiones de eventos se publican automáticamente en el tópico *event emission*:

```
{
    "name": <event_name>,
    "data": <event_payload>,
    "timestamp": <unix_timestamp_ms>
}
```

Código 3.44: Binding MQTT: Mensaje de emisión de evento

3.2.4. CoAP

CoAP [74] es un protocolo de petición y respuesta especialmente indicado para dispositivos con capacidades limitadas (por ejemplo, microcontroladores). Presenta múltiples similitudes con HTTP, de hecho, 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 normalmente sobre UDP en vez de TCP, aunque existen alternativas para transporte sobre TCP [75]. 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 [76]. 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, que es la implementación de Protocol Binding con la que se pueden establecer paralelismos más claros:

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.

La tabla 3.7 muestra el detalle de la estructura de los elementos *form* dentro de la implementación CoAP.

Campo	Descripción
`op`	Verbo de interacción asociado a este elemento.
`href`	La URL del servidor CoAP en el que se expone esta interacción.
`mediaType`	Este campo siempre contiene el tipo MIME de JSON.

Cuadro 3.7: Estructura de elementos *form* en implementación CoAP

En el resto de esta sección se muestran los detalles de bajo nivel de la implementación aplicados al modelo de interacciones WoT.

Read Property

Estructura del cuerpo de la petición:

```
GET coap://<host>:<port>/property?thing=<thing_name>&name=<
    ↪ property_name>
```

Código 3.45: Binding CoAP: Petición *Read Property*

Estructura del cuerpo de la respuesta:

```
CoAP 2.05 Content

{
    "value": <property_value>
}
```

Código 3.46: Binding CoAP: Respuesta *Read Property*

Write Property

Estructura del cuerpo de la petición:

```
PUT coap://<host>:<port>/property?thing=<thing_name>&name=<
    ↪ property_name>

{
    "value": <property_value>
}
```

Código 3.47: Binding CoAP: Petición *Write Property*

Estructura del cuerpo de la respuesta:

```
CoAP 2.04 Changed
```

Código 3.48: Binding CoAP: Respuesta *Write Property*

Invoke Action

El proceso de invocación de un acción se inicia a través de una petición POST:

```
POST coap://<host>:<port>/action?thing=<thing_name>&name=<
    ↪ action_name>

{
    "input": <action_argument>
}
```

Código 3.49: Binding CoAP: Petición *Invoke Action*

En este caso, el servidor CoAP responderá con el identificador único asignado a esta invocación en particular:

```
CoAP 2.01 Created

{
    "id": <invocation_id>
}
```

Código 3.50: Binding CoAP: Respuesta con ID de invocación

El cliente puede comprobar en cualquier momento el estado de la invocación registrándose como un *observer* del recurso e indicando el identificador de la invocación en el cuerpo:

```
GET coap://<host>:<port>/action?thing=<thing_name>&name=<
    ↪ action_name>

{
    "id": <invocation_id>
}
```

Código 3.51: Binding CoAP: Registro de *observer* de invocación

Los mensajes de estado de la invocación que se envían del lado del servidor se ajustan al siguiente formato:

```
CoAP 2.05 Content

{
    "done": <boolean>,
    "id": <invocation_id>,
    "result" <result_value>,
    "error": <error_message>
}
```

Código 3.52: Binding CoAP: Respuesta de estado de invocación

Observe Property

La interfaz de este verbo de interacción es equivalente al caso de *read property* con la única diferencia de que el cliente debe registrarse como un *observer*, de acuerdo a la especificación del RFC [76], para empezar a recibir las actualizaciones por parte del servidor.

Observe Event

El cliente puede crear una suscripción a un evento registrándose como *observer* del recurso que representa el evento:

```
GET coap://<host>:<port>/event?thing=<thing_name>&name=<
    ↪ event_name>
```

Código 3.53: Binding CoAP: Petición *Observe Event*

Todas las respuestas iniciadas desde el lado del servidor contendrán la emisión más reciente del evento en cuestión:

```
CoAP 2.05 Content

{
    "name": <event_name>,
    "data": <event_payload>,
    "time": <timestamp_ms>
}
```

Código 3.54: Binding CoAP: Respuesta con emisión de evento

Capítulo 4

Emulador de aplicaciones Web of Things sobre arquitecturas basadas en edge computing

Este capítulo describe los detalles del diseño y la lógica detrás del emulador de despliegues Web of Things basados en arquitectura edge computing. Una herramienta de estas características puede permitir la validación de arquitecturas WoT con una baja inversión, antes de comprometerse a un despliegue real en campo o una prueba con dispositivos reales. Además, la integración de un orquestador de contenedores moderno mejora significativamente la portabilidad y escalabilidad horizontal de los experimentos. La motivación se describe con mayor detalle en la sección 2.4.

Existe una implementación experimental plenamente funcional que está públicamente disponible en Zenodo [77]. Además, el proyecto **WoTemu** también está publicado en GitHub [78].

Los trabajos en la línea del emulador de aplicaciones WoT basadas en edge computing han dado lugar a una publicación [79] que puede verse adjunta en el apéndice B.

4.1. Arquitectura

La orquestación de contenedores es uno de los bloques fundamentales del diseño del emulador propuesto y su implementación experimental. Tal y como se describe en la sección 2.3, las herramientas existentes dentro del estado de la técnica relacionadas con la simulación y emulación de despliegues IoT presentan incógnitas significativas en lo que a escalabilidad horizontal se refiere. La integración del orquestador de contenedores *Docker Swarm mode* permi-

te crecer de manera horizontal (añadiendo más nodos al clúster) de manera sencilla, habilitando la emulación de experimentos de tamaño significativo. Además, ofrece funcionalidades que se alinean bien con las necesidades del emulador, por ejemplo, la limitación de recursos computacionales y los drivers de red distribuidos (driver *overlay*).

Kubernetes [57] es probablemente la herramienta de orquestación de contenedores más extendida para despliegue y mantenimiento de servicios en producción. Por esta razón Kubernetes también se tuvo en consideración durante el diseño. Sin embargo, una revisión inicial de la documentación reveló que adaptar alguna de las implementaciones disponibles de *Container Network Interface* (CNI) de Kubernetes era comparativamente más costoso. En otras palabras, el driver *overlay* de Swarm se ajusta particularmente mejor al caso de uso de las redes del emulador que la alternativa equivalente en Kubernetes. Las razones para la selección de Swarm como cimiento del emulador se describen con mayor detalle a continuación:

- Docker está compuesto por un conjunto de servicios y utilidades que son razonablemente sencillas de instalar y que están disponibles en todas las plataformas relevantes. Swarm es el orquestador de contenedores incluido por defecto en Docker, por lo que requiere relativamente poca configuración. Es posible llegar a disponer de un clúster Swarm funcional con una baja inversión de tiempo.

- Las herramientas de orquestación de contenedores se caracterizan por sus capacidades de escalabilidad horizontal, que son un requerimiento básico para habilitar la emulación de experimentos con un alto número de entidades. Esta característica está precisamente ausente en las herramientas de emulación dentro del estado de la técnica.

- Emular dispositivos hardware con capacidades limitadas es simple gracias a la funcionalidad de imposición de límites arbitrarios en contenedores. De esta manera, se pueden representar aproximadamente las capacidades de placas de desarrollo IoT y *Single-Board Computers*.

- El ciclo de vida de los experimentos se simplifica significativamente gracias a los contenedores. Los recursos ocupados por los experimentos son efímeros y pueden eliminarse del sistema con sencillez. Además, se favorece la reproducibilidad y la velocidad de iteración.

- El driver de red distribuido *overlay* que está incluido por defecto en Swarm tiene un buen encaje para la representación de redes aisladas y

conexiones entre entidades dentro de un escenario basado en arquitectura edge computing. La sección 4.1.2 profundiza en la manera en la que el emulador aprovecha las capacidades del driver *overlay*.

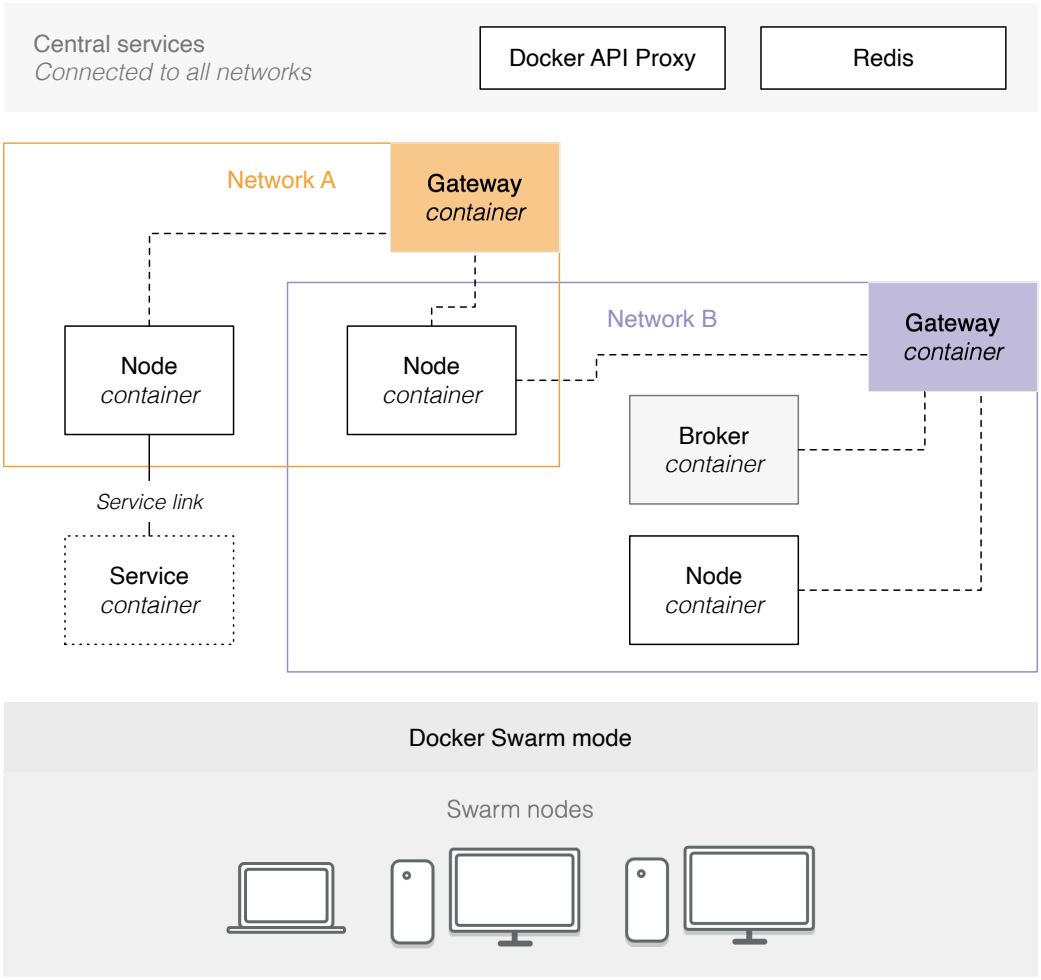


Figura 4.1: Vista de alto nivel de la arquitectura del emulador

El emulador está intrínsecamente enlazado con conceptos propios de Docker Swarm. A continuación se incluyen definiciones de estos conceptos para mayor claridad:

Swarm stack Un *stack* es un conjunto de servicios heterogéneos que suelen tener relación entre sí y que se despliegan y gestionan como una única unidad lógica de alto nivel.

Swarm task Una *task* (o tarea) está enlazada con un contenedor particular en relación uno a uno. Es decir, la *task* es la unidad atómica de ejecución dentro del clúster. Si una *task* falla for cualquier razón, el orquestador intenta volver a crearla para mantener el estado deseado del *stack*. Los conceptos de *task* y contenedor son intercambiables en la mayoría de los casos dentro del contexto del emulador.

Swarm service Un *service* (o servicio) es una plantilla para la generación de *tasks* que describe atributos tales como los puertos, los volúmenes y la imagen base. Un *service* puede tener múltiples réplicas, donde cada una es una *task* individual. Por defecto, todas las peticiones dirigidas a un *service* se balancean entre sus réplicas.

Swarm network Una *network* (o red) interconecta diferentes *services*, cuyas *tasks* pueden estar desplegadas en distintos nodos físicos del clúster gracias a las capacidades del driver *overlay*. Los *services* que están en una misma *network* pueden utilizar las capacidades de descubrimiento de servicios de Swarm para comunicarse internamente sin necesidad de acceder a través de las interfaces públicas (de hecho, la mayoría de los servicios de un stack no suelen estar expuestos a la red pública).

La figura 4.1 muestra una vista general de alto nivel del modelo de arquitectura del emulador, incluyendo los distintos tipos de contenedores, que se explican con mayor detalle en los siguientes párrafos. Estos cuatro tipos o *clases* de contenedores son los bloques con los que se construye una topología para la ejecución de experimentos:

Node Los contenedores de tipo *node* representan programas que deben ser sometidos a prueba durante la ejecución del experimento para su análisis posterior. Todos los *node* ejecutan una aplicación de usuario, que tiene la forma de un módulo Python que expone una función asíncrona (`async def`) que recibe tres argumentos: el punto de entrada `WoT` para la consumición y exposición de Things, la configuración y la instancia del loop de eventos `asyncio`. Esta relación uno a uno con programas de usuario hace que, por simplicidad, los contenedores *node* reciban a veces el nombre de aplicaciones. Dado que la adopción de las especificaciones W3C WoT como un ciudadano de primer orden es uno de los principales objetivos de diseño del emulador, el argumento `WoT` mencionado anteriormente es una instancia decorada del punto de entrada proporcionado por WoTPy [64], que es una implementación experimental de las especificaciones W3C WoT (véase capítulo 3). Sin embargo,

es importante destacar que el usuario no está forzado a definir su aplicación en términos del modelo WoT, cualquier programa que siga el modelo asíncrono de `asyncio` [80] puede ser ejecutado dentro de un *node*. Es decir, una aplicación puede actuar como un dispositivo IoT emulado, un programa de control para un dispositivo IoT real, un servicio de procesamiento, una interfaz para una base de datos, o cualquier otro elemento de interés para el experimento. Finalmente, el emulador incluye un conjunto de aplicaciones de serie (*built-in apps*) que pueden servir como *mocks* y utilidades configurables con fines de prueba.

Gateway Existe exactamente un contenedor de clase *gateway* para cada una de las *swarm network* en la topología. Estos contenedores se crean de manera automática y se inyectan transparentemente en medio de las comunicaciones entre contenedores *node* para emular condiciones reales de red (por ejemplo, límites de ancho de banda). La sección 4.1.2 contiene más información acerca de la emulación de red.

Service Este tipo de contenedores representan cualquier servicio software que pudiese ser necesario en el contexto de una aplicación. Por ejemplo, una base de datos, una cola de mensajes o un sistema de autenticación. Se pueden identificar dos diferencias principales con los otros tipos de contenedores. La primera diferencia es que no se ejecutan procesos de monitorización en segundo plano (descritos en la sección 4.3). La segunda es que los servicios existen en sus propias redes independientes, de manera que un *node* debe declarar un enlace explícito (*service link*) para comunicarse con un *service*. Los *service* proporcionan utilidad a la topología, pero no son las entidades bajo prueba que se persigue analizar.

Broker La implementación de Protocol Binding MQTT del framework WoT no puede operar de manera autocontenida, es decir, necesita forzosamente la presencia de un broker MQTT externo. Es por esta razón que los broker MQTT reciben una consideración especial dentro de los experimentos de emulación, lo que implica la inclusión de procesos de monitorización en segundo plano (a diferencia de los contenedores *service*). La implementación de los *broker* se basa en el popular y contrastado proyecto de código abierto Eclipse Mosquitto [73].

Además, todos los experimentos de emulación dependen de dos **servicios centrales** que están conectados a todas las redes dentro de la topología:

Docker API Proxy Varios de los procesos de inicialización de contenedores y procesamiento de los resultados del experimento dependen del

estado del clúster Swarm. Puesto que no existen entidades centrales encargadas de la gestión del experimento, cada contenedor debe poder tener acceso al estado de manera independiente para su propia autoconfiguración. El estado se expone a través de un instancia de la imagen `docker-socket-proxy` [81], que actúa como proxy para el acceso a la Docker API a través de las redes *overlay* internas. La Docker API es la interfaz de programación para gestión de todas las entidades en un entorno Docker, y permite, por ejemplo, descubrir todas las *task* creadas bajo un *service* y actualizar dinámicamente las restricciones de cuota de CPU y memoria de un contenedor.

Redis Redis [82] es un servicio de almacenamiento clave-valor de código abierto que reside en memoria (aunque tiene capacidades de persistencia a disco). Redis se caracteriza por su alto rendimiento, por lo que su utilización, en contraste con una base de datos relacional clásica, resulta en latencias menores para operaciones de lectura y escritura a cambio de una mayor huella de memoria. Dentro de los experimentos, se utiliza para almacenar datos históricos y medidas de métricas que son necesarias para caracterizar y analizar el comportamiento de la topología. Resulta especialmente indicado para el caso del emulador porque asegura un impacto de rendimiento mínimo en todos los contenedores. La sección 4.1.1 contiene más información acerca de los datos que se almacenan en Redis durante un experimento.

Dentro de la terminología del emulador, una configuración concreta de las entidades que se describen anteriormente recibe el nombre de *topología*. Un *experimento* es una instancia o ejecución particular de una topología. Las topologías se definen en Python utilizando la API del emulador, para después ser convertidas en archivos Compose usando el CLI del emulador. Una vez que se dispone de un fichero Compose, la topología puede desplegarse en un clúster Swarm utilizando el CLI de Docker de manera regular. Este flujo de trabajo se muestra de manera ilustrativa en la figura 4.2.

Finalmente, es importante destacar que una de las principales ventajas del diseño propuesto es que los usuarios no están limitados en la utilización de dispositivos *virtuales* o emulados. El emulador puede ejecutar **código real en tiempo real** en la forma de un stack de servicios basados en contenedores. En la práctica, esto significa que cualquier aplicación puede usarse dentro de un experimento con mínimas modificaciones, lo que habilita a los usuarios a comunicarse con la mayoría de los dispositivos IoT reales existentes a día de hoy. En otras palabras, cualquier dispositivo IoT que exponga una interfaz basada en un protocolo bien conocido se puede integrar dentro de un experimento.

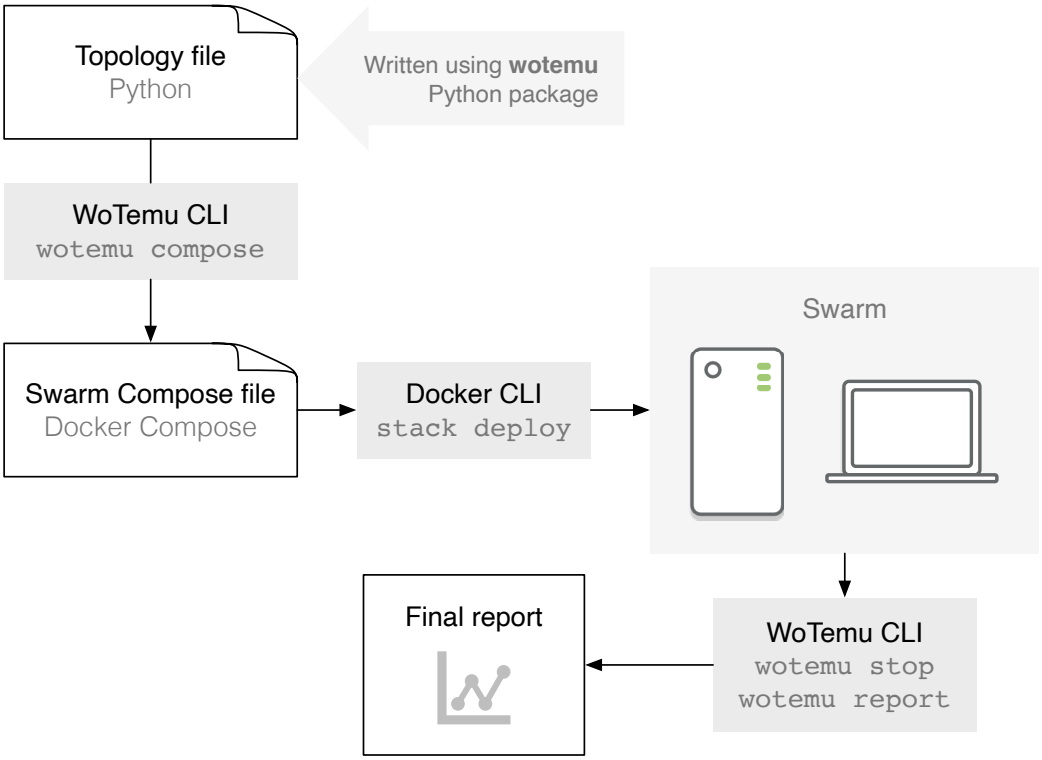


Figura 4.2: Flujo de trabajo del emulador

4.1.1. Módulo de monitorización

Uno de los principales objetivos del emulador es proporcionar visibilidad del comportamiento y rendimiento de los experimentos del usuario. El *módulo de monitorización* tiene precisamente esta responsabilidad, es decir, establecer los bloques fundamentales que permiten la captura, almacenamiento y posterior procesamiento de los datos que representan el comportamiento de la topología.

Se pueden identificar cuatro procesos de monitorización independientes que se inyectan en todos los contenedores del experimento de manera transparente. Esto ocurre en todos los experimentos durante la fase de inicialización, es decir, la aplicación proporcionada por el usuario es ejecutada por el punto de entrada del emulador junto con un conjunto de hilos paralelos.

Centralizar el almacenamiento en Redis supone una mayor carga de memoria en comparación con una aproximación más clásica basada en una base de datos relacional (o incluso una opción NoSQL). Sin embargo, Redis asegura una latencia mínima en las interacciones, lo que a su vez implica un

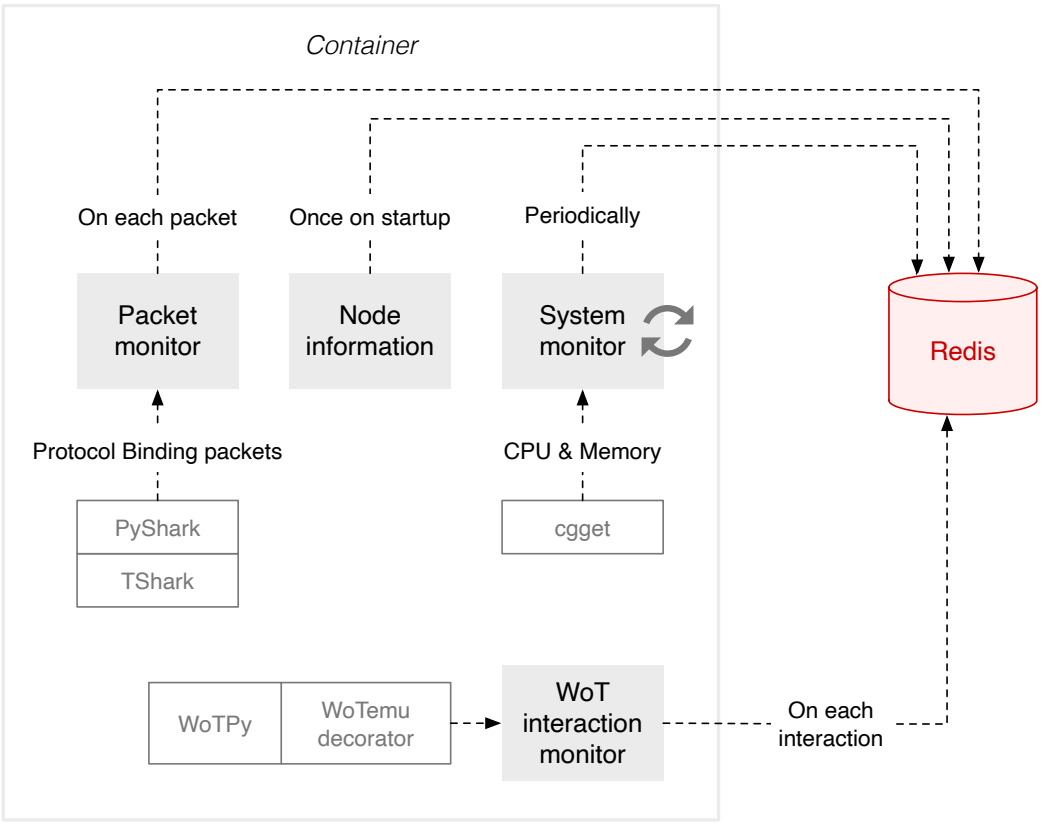


Figura 4.3: Módulo de monitorización

mínimo impacto en el rendimiento de la aplicación y unos resultados finales más representativos del comportamiento real. En otras palabras, atendiendo al caso de uso del emulador, es preferible pagar el coste de huella de memoria que el de latencia y carga de procesamiento.

En la figura 4.3 se puede ver un diagrama que muestra las interacciones de alto nivel de los procesos de monitorización, que se describen a continuación con mayor detalle.

Monitor de paquetes La red es uno de los limitantes principales en un despliegue IoT. Es común encontrar escenarios en los que la cobertura es escasa, y los dispositivos se comunican a través de conexiones con tarificación por volumen. Esto tiene implicaciones importantes a la hora de diseñar un despliegue, optimizar la transferencia de datos y detectar *fugas* inesperadas es un objetivo relevante en la mayoría de los casos. El monitor de paquetes está basado en el analizador de paquetes TShark [83], que es la utilidad de línea de comandos de la popular herramienta

Wireshark. El monitor ejecuta un proceso en segundo plano que está configurado para capturar el tráfico que emana de las implementaciones de Protocol Binding. Concretamente, se utiliza el paquete PyShark [84] a modo de interfaz programática con TShark. Todos los paquetes capturados se almacenan en el servicio central Redis, sin llevar a cabo agregaciones previas, con la intención de habilitar un análisis profundo del tráfico de red en una fase posterior del experimento.

Información del nodo Todos los contenedores presentan unas características y atributos de entorno que son necesarios para proporcionar contexto adecuado a la hora de generar el informe final. Este contexto permite que el módulo de informes (sección 4.1.4) categorice y agrupe apropiadamente los contenedores. Con este objetivo, se ejecuta un proceso efímero en todos los contenedores durante su inicialización para la captura de los metadatos necesarios. Ejemplos de los metadatos capturados son el modelo de CPU, información extendida sobre las interfaces de red, el identificador único de la *swarm task* y los límites (*constraints*) de recursos que han sido impuestos por el usuario.

Monitor de sistema Este monitor se encarga de capturar muestras del consumo de recursos del sistema con una frecuencia elevada. Para poder obtener muestras representativas de cada contenedor de manera independiente, las métricas se leen a través de la utilidad de línea de comandos `cgget`, que permite la lectura de parámetros de grupos de control en Linux (*control groups*).

Monitor de interacciones WoT Habilitar al usuario para diseñar y optimizar un despliegue en base a las especificaciones W3C WoT es uno de los principales objetivos que persigue el diseño experimental del emulador. Por esta razón, se incluye el monitor de interacciones Web of Things, que se encarga de registrar todas *interacciones* llevadas a cabo por las Things de la topología (por ejemplo, *action invocations*). El diseño se apoya sobre el hecho de que las aplicaciones de usuario hacen uso de un punto de entrada WoT (interfaz `WoT` del framework descrito en el capítulo 3) para consumir y exponer Things. Dentro del emulador, el punto de entrada inyectado en la aplicación de usuario está decorado para capturar y registrar de manera transparente todas las interacciones. De esta manera se libera al usuario de la responsabilidad de la monitorización.

Finalmente, y con el objetivo de proporcionar mayor flexibilidad al usuario a la hora de diseñar los experimentos, el emulador proporciona una API

para la escritura de **métricas de aplicación** ad-hoc. Estas también son conocidas como *métricas de usuario*. Con este mecanismo el usuario puede aprovechar las herramientas integradas en el emulador para persistir métricas que tienen un significado particular en su experimento. El módulo de informes (sección 4.1.4) es capaz posteriormente de procesar de manera dinámica las métricas de aplicación.

4.1.2. Módulo de enrutamiento

La presencia de redes de comunicación inestables y con anchos de banda variables es uno de los limitantes más significativos que se presentan a la hora de diseñar un despliegue IoT. Afrontar una arquitectura IoT asumiendo la presencia de una capa de red efectivamente transparente, como podría ocurrir en un entorno en la nube, es un riesgo alto que pone en peligro la viabilidad del diseño. Por ejemplo, el firmware de una Thing que utiliza MQTT como protocolo de capa de aplicación, y que demuestra comportamiento aceptable en laboratorio, podría verse invalidado en campo por no haber considerado el coste de las constantes conexiones contra un broker MQTT causadas por la inestabilidad de la red móvil disponible.

El módulo de enrutamiento del emulador habilita a los usuarios a emular el comportamiento de redes realista de manera transparente para la aplicación. Esto implica la definición bajo demanda de límites de ancho de banda, latencia y modelos de pérdida. El módulo se construye sobre dos bloques fundamentales de la gestión de red en Linux: el paquete *iproute2* [85] que contiene utilidades para gestión del stack TCP/IP y la herramienta de filtrado de paquetes *iptables* [86]. Dentro de *iproute2* se encuentra la utilidad de control de tráfico denominada *tc*, que se utiliza en el contexto del emulador para imponer restricciones arbitrarias en la calidad de las conexiones de red entre contenedores. Concretamente, se utilizan los módulos *Network Emulator* (NetEm) y *Token Bucket Filter* (TBF).

Todos los contenedores de tipo *node* y *broker* se configuran automáticamente en su inicialización con las reglas de enrutamiento necesarias para habilitar la emulación de red. A continuación se detallan dichas reglas, que se pueden ver representadas en el diagrama 4.4:

- Dentro de la tabla `mangle` de iptables se definen un conjunto de reglas que añaden una *kernel mark* a paquetes de las implementaciones de Protocol Binding del WoT runtime. Las reglas se añaden a la cadena `OUTPUT` y están diseñadas para ajustarse a los puertos de origen o destino de los protocolos de capa de aplicación: HTTP, Websockets,

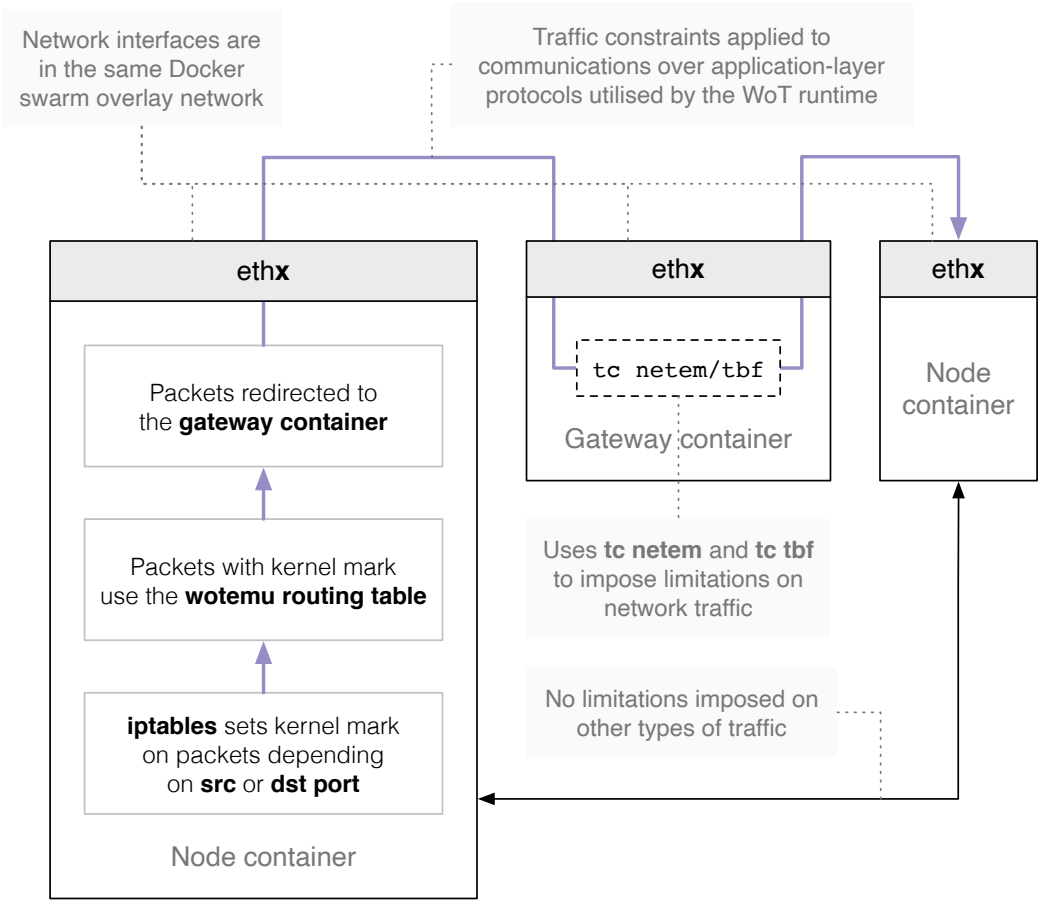


Figura 4.4: Módulo de enrutamiento

CoAP y MQTT. Este proceso es capaz de ajustarse a puertos configurados de manera dinámica, es decir, los puertos no están codificados de antemano.

- Existe una regla en la base de datos de enrutamiento que hace que todos los paquetes con la *kernel mark* mencionada anteriormente utilicen la tabla de enrutamiento específica del emulador.
- La tabla de enrutamiento del emulador (tabla `wotemu`) contiene entradas para forzar a todos los paquetes (o datagramas) de las implementaciones de Protocol Binding a pasar a través del contenedor *gateway* particular de la red *overlay* de Docker sobre la que se transportan las comunicaciones.

Los contenedores *gateway* ejecutan un conjunto de procesos *tc* que se

encargan de ajustar el tráfico que pasa a través del contenedor para emular los niveles deseados de latencia, ancho de banda y probabilidades de pérdida. Existe un único contenedor *gateway* para cada red de la topología, donde se agrega todo el tráfico de los distintos contenedores de la red en cuestión. Las redes se implementan con el driver *overlay* de Docker, que proporciona los mecanismos para habilitar comunicación transparente entre contenedores presentes en diferentes nodos físicos del clúster. Esta aproximación, basada en la agregación de tráfico en un cuello de botella *gateway*, permite llevar a cabo una emulación más realista en la que múltiples clientes pueden competir por los recursos de un único canal de comunicación; por ejemplo, una conexión de red móvil compartida por un conjunto de dispositivos sensores desplegados en campo.

Los principios de diseño escalable del emulador demandan que cada contenedor individual debe ser capaz de configurar su propio stack de red TCP/IP sin la intervención de un componente central. Para esto, los contenedores deben tener acceso a un servicio proxy de la API de Docker para recuperar información sobre la configuración de red del swarm. Esto presenta un compromiso en el sentido de que se disminuye el nivel de aislamiento entre contenedores. Algunos ejemplos de información centralizada requerida por los contenedores son: el identificador de *task* del contenedor actual, el identificador de *task* de todos los contenedores *gateway* en la topología y todas las réplicas (contenedores) existentes de un *swarm service*.

La principal desventaja de esta pérdida de aislamiento es que aumentan los riesgos de seguridad. Sin embargo, es razonable argumentar que los experimentos del emulador están diseñados para ejecutarse de manera efímera en entornos experimentales privados. Es decir, a diferencia de un conjunto de servicios *comunes*, los experimentos no deberían estar activos de manera continuada en el tiempo ni expuestos en Internet. Esto disminuye notablemente el impacto de la pérdida de seguridad.

4.1.3. Módulo de benchmark

Establecer un mecanismo para representar niveles de rendimiento objetivos, y aplicables de manera horizontal a cualquier nodo que pudiese estar presente en el clúster, es un reto significativo. Esto es necesario en el contexto del emulador para que un nodo pueda operar consistentemente de acuerdo al perfil de rendimiento que se establece en la topología. Los limites de rendimiento son el bloque fundamental que permite emular el comportamiento de una aplicación en un entorno limitado, sin necesidad de ejecutarlo en una plataforma hardware real. Si se necesitase disponer de hardware real para emular placas de desarrollo IoT, el emulador perdería su sentido y utilidad,

ya que el coste y complejidad del experimento crecería aproximadamente hasta el punto de un despliegue real.

Se pueden establecer límites de memoria para contenedores en términos de tamaño (bytes), así como límites de uso de CPU en términos de cuota en el *Completely Fair Scheduler* (CFS), que es un planificador de procesos integrado en el kernel de Linux. El problema principal es que estos mecanismos nativos que expone Docker para limitación del rendimiento de la CPU no son adecuados para el caso de uso del emulador.

Los límites de memoria mantienen su significado, de manera aproximada, cuando se aplican en diferentes máquinas en el clúster, independientemente del tamaño total de la memoria instalada. La frecuencia y la latencia de la memoria física toman un papel relevante, pero es razonable argumentar que la capacidad de memoria es el cuello de botella principal en la mayoría de las aplicaciones IoT. Sin embargo, este razonamiento no se puede aplicar para el caso de las cuotas de CPU en el planificador CFS. La misma cuota puede representar niveles notablemente distintos de capacidad computacional en diferentes modelos de CPU. Es decir, hay una gran diferencia entre la capacidad ofrecida por el 100 % de uso de un núcleo en un procesador x86 moderno y un procesador ARM de bajo consumo diseñado para operar con alimentación por batería.

El diseño del emulador propone una solución a esta problemática, es decir, la portabilidad de límites de CPU, a través de la utilización de puntuaciones sintéticas de CPU proporcionadas por una herramienta de benchmark. La herramienta seleccionada en este caso es Sysbench [87], un candidato destacable comúnmente utilizado en el contexto de las bases de datos [88] con estabilidad contrastada y aceptación en la comunidad. Es relevante destacar que los benchmarks sintéticos no resultan altamente precisos a la hora de caracterizar el rendimiento de un componente, si se persigue alta precisión, es necesario diseñar cargas de trabajo ad-hoc que permitan modelar los casos de uso reales. Sin embargo, en este caso las puntuaciones sintéticas proporcionadas por Sysbench representan un compromiso razonable entre alcance, fiabilidad e impacto en el rendimiento del emulador en su conjunto.

El mecanismo utilizado por el módulo de *benchmark* para ajustar apropiadamente las cuotas de CPU se describe con detalle en la siguiente lista enumerada. Para mayor claridad, la figura 4.5 representa este mismo proceso en forma de diagrama.

1. El usuario puede, de manera opcional, definir una variable de entorno en el servicio que contenga la puntuación objetivo de rendimiento de CPU para todos los contenedores que emanan de ese servicio en concreto. Si la variable está definida, el punto de entrada del emulador lanzará el

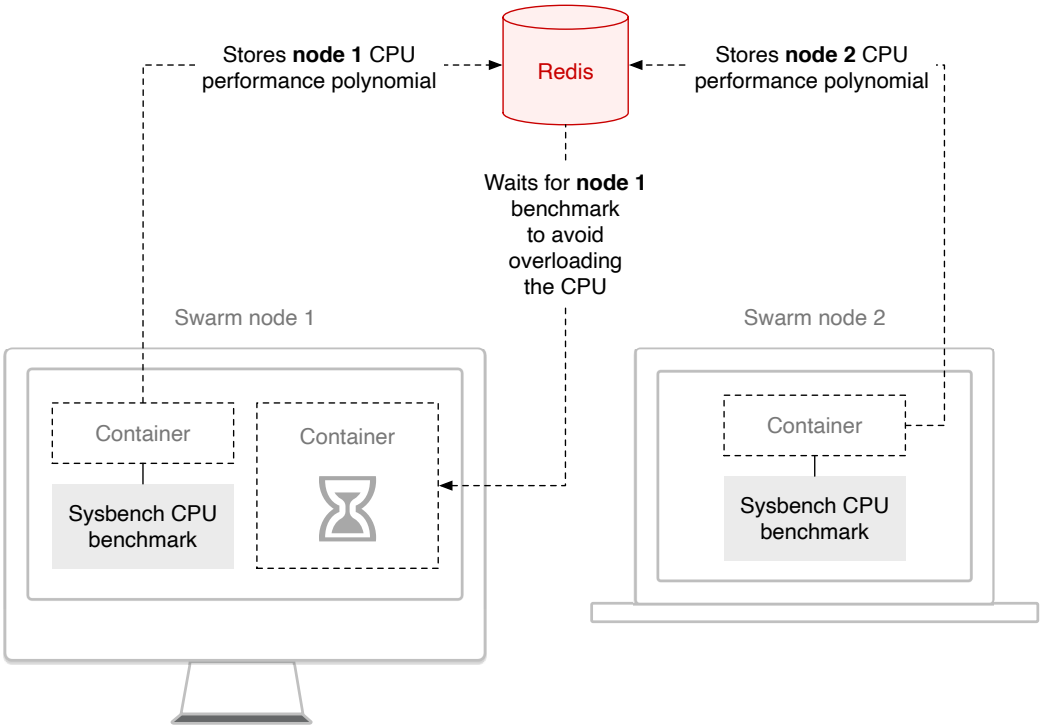


Figura 4.5: Diagrama del proceso de benchmark de CPU

proceso para actualizar la cuota de CPU en todos los contenedores de manera que se ajuste a la puntuación objetivo.

2. El primer contenedor en un nodo físico (*swarm node*) que necesita actualizar la cuota de CPU crea una nueva clave en el servicio Redis y después pasa a ejecutar una serie de benchmarks de CPU. La clave en cuestión utiliza un prefijo a modo de espacio de nombres que incluye el ID del nodo físico, que naturalmente es el mismo para todos los contenedores dentro de él. Es decir, todos los contenedores dentro del nodo físico que llegan a este punto del proceso en un punto más avanzado en el tiempo sabrán que el proceso de benchmark se encuentra actualmente en ejecución, por lo que no es necesario volver a iniciarlo. Si esto no fuese así, múltiples procesos de benchmark existirían de manera paralela, lo que tendría un impacto negativo en el rendimiento de la CPU y produciría resultados de rendimiento poco representativos.

3. El proceso de benchmark de CPU consiste en la ejecución consecutiva de varias instancias de benchmark CPU de Sysbench. Los procesos

Sysbench se ejecutan dentro de contenedores efímeros con limitaciones crecientes de cuota de CPU; estas limitaciones crecen hasta el 100 % de uso de un solo núcleo. El objetivo final es obtener los coeficientes de un polinomio ajustado que pueda caracterizar el rendimiento de un núcleo en la CPU actual. Es decir, los resultados individuales de cada instancia de benchmark Sysbench sirven como muestras para el ajuste de la curva, donde el eje horizontal de la curva es el porcentaje de uso de un núcleo y el eje vertical es la puntuación de CPU Sysbench.

4. Mientras tanto, el resto de los contenedores en el nodo físico simplemente esperan por la generación de los coeficientes del polinomio representativo de un núcleo de CPU. Para ello, observan una clave en Redis a la espera de que se actualice.

5. Cuando el proceso de benchmark de CPU termina, los coeficientes se actualizan en Redis. Después, los contenedores resuelven el polinomio de CPU para su puntuación objetivo de CPU. Con esto se obtiene la cuota de CPU objetivo, que posteriormente se aplica a los contenedores a través de la API de Docker.

4.1.4. Módulo de informes

El módulo de informes tiene la responsabilidad de transformar las métricas capturadas, que se almacenan en el servicio central Redis, en representaciones adecuadas para ser procesadas e interpretadas por usuarios. Concretamente, se ofrece soporte para dos tipos de representación:

- Una representación en formato Web, que resulta más amigable para usuarios finales y es la manera recomendada de obtener una primera visión del comportamiento del experimento. En la implementación experimental, el informe se apoya sobre los componentes Web expuestos por Plotly [89], una librería de código abierto para la construcción de gráficos y diagramas. Se puede ver un ejemplo de informe en una página pública desplegada en GitHub[1]. En la figura 4.6 se muestra una captura de pantalla de dicho informe de ejemplo.

- Una representación semiestructurada serializada en formato JSON, que resulta más adecuada para su procesamiento posterior por una máquina, en caso de que el usuario necesite implementar algún tipo de agregación o flujo de procesamiento específico que no esté contemplado dentro del informe Web.

[1] `https://agmangas.github.io/demo-wotemu-report/`

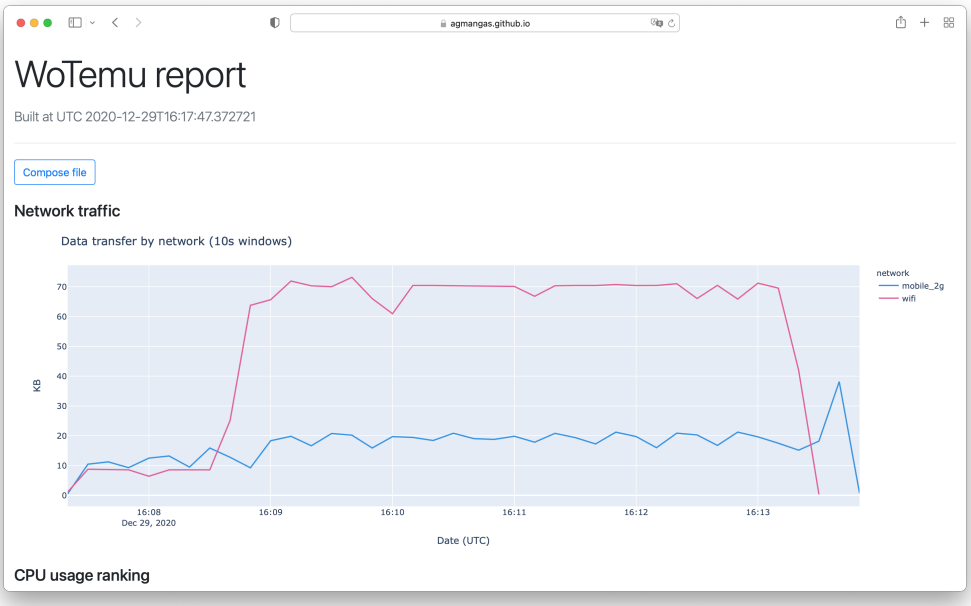


Figura 4.6: Captura de un informe del emulador en formato Web

A continuación se presenta una lista de las vistas más relevantes incluidas en un informe del emulador en formato Web:

Service traffic Mapas de calor que describen el volumen del flujo de datos que fluye *desde* y *hacia* los servicios. Es decir, los datos están agrupados por contenedor (*task*) y servicio. Es especialmente útil para detectar los servicios que tienen la mayor carga desde un punto de vista de utilización de la red, lo que puede permitir ajustar el dimensionamiento. Por ejemplo, podría ser que una API HTTP desplegada en el *edge* estuviese recibiendo un volumen excesivo de peticiones de los dispositivos finales desplegados en campo; un vistazo al mapa de calor revelaría la necesidad de balancear la carga entre varias instancias de la API.

Network traffic Series temporales de la evolución del volumen del tráfico para cada una de las redes definidas en el experimento. Esta vista sirve principalmente para detectar las redes que se encuentran al límite de sus recursos. Es decir, observar que el tráfico de una red permanece de manera constante en el límite superior del ancho de banda es indicativo de que se deberían aumentar los recursos, por ejemplo, creciendo de una red 3G a 4G.

Resource usage rankings Distribución del uso de recursos computacionales de CPU y memoria para todos los contenedores. El uso de CPU se categoriza por modelo para acomodar los casos en los que el clúster (*swarm*) está formado por máquinas con componentes heterogéneos. Proporciona información acerca de la varianza del uso de recursos, así como de los contenedores que generan una mayor carga.

Timeline of tasks La línea temporal que incluye información visual acerca de los momentos de creación y parada de todos los contenedores, incluyendo los que terminaron con error. Esto permite detectar de un vistazo situaciones en las que los contenedores se detienen y vuelven a ser creados de manera constante por el orquestador, o que terminan de manera temprana e inesperada.

Task resource usage Serie temporal de la evolución detallada de uso de CPU y memoria para un contenedor en concreto. Esta vista incluye la referencia de los límites de recursos (*resource constraints*) en caso de que estén definidos.

Task data transfer Series temporales de volumen de tráfico de red agrupado por interfaz de red (interna al contenedor) o el protocolo de capa superior. Permite identificar la presión que una implementación de Protocol Binding específica está ejerciendo sobre los recursos de red. El identificador del protocolo es el que reporta la herramienta de monitorización de red (TShark), que no siempre es capaz de detectar apropiadamente el protocolo de capa de aplicación; por ejemplo, en la duración de un flujo Websockets es posible que parte de los paquetes se categoricen solamente como TCP.

Task interactions Esta categoría de componentes visuales proporciona información desde el punto de vista de la WoT, es decir, en términos de WoT Things y verbos de interacción. Concretamente, se muestran las distribuciones de latencia de invocación de acciones y lectura/escritura de propiedades, la serie temporal de ocurrencia de verbos de evento y los totales de interacciones categorizadas en función de su estado (éxito o fallo).

```
{
    "data": [
        {
            "created_at": 1622234898.6148794,
            "latency_ptz": 8.837477445602417,
            "time_arrival": 1622234889.6014142,
            "time_capture": 1622234889.3947783
        },
        ...
    ],
    "key": "ptz_latency",
    "task": "camera.1.g4qsogt64xpbjm4y2oefuzjzb"
}
```

Código 4.1: Ejemplo array `app_metrics`

La implementación experimental del módulo de informes está construida sobre el popular paquete Pandas [90], que proporciona capacidades avanzadas para agregación y filtrado de datos. Por esta razón, la versión del informe de salida del emulador con formato JSON está basado parcialmente en la serialización de un DataFrame Pandas, donde el parámetro `orient` se define con el valor `split`. Esto quiere decir que el documento JSON que representa el DataFrame contiene tres campos: un array de nombres de columna `columns`, un array de datos representados como filas `data`, y un array de índices de las filas `index` que por defecto es una serie incremental de enteros desde 0.

De manera paralela a la lista descriptiva previa enfocada en las vistas incluidas en el informe Web, en los párrafos siguientes se puede ver un listado de los campos incluidos en los informes serializados en formato JSON. Se describen sus contenidos y estructura, y además se muestran ejemplos para mayor claridad. Para más información, se puede revisar el informe publicado en [91], cuyos contenidos se describen con más detalle en la sección 5.2.

Array `app_metrics`. Una lista de *métricas de aplicación* ad-hoc que han sido definidas dentro del código en el espacio del usuario. Los elementos del array son objetos que contienen el identificador de la métrica, el identificador del contenedor de origen y un array de muestras pertenecientes a esta métrica específica. Es decir, existe un documento para cada combinación de métrica y contenedor de origen. En el bloque de código 4.1 se muestra un ejemplo representativo de elemento del array que hace referencia a una métrica ad-hoc con el nombre `ptz_latency`.

```
{
    "columns": ["dst_service", "len", "src_task"],
    "data": [
        ["detector", 13458686, "camera.1.
            ↪ g4qsogt64xpbjm4y2oefuzjzb"],
        ...
    ],
    "index": [0, ...]
}
```

Código 4.2: Ejemplo objeto `service_traffic.inbound`

Objeto `service_traffic.inbound`. Un DataFrame que contiene el volumen total de tráfico de red (bytes) agrupado por contenedor de origen y servicio de destino. Esto permite básicamente analizar el tráfico que entra en un servicio con el objetivo de facilitar la detección de desequilibrios de carga en el despliegue. Es una vista agregada derivada de los datos proporcionados por las claves `tasks.<task>.packet` descritas posteriormente. El bloque de código 4.2 muestra un ejemplo representativo.

Objeto `service_traffic.outbound`. Son equivalentes al objeto anterior (`inbound`) con la diferencia de que describen el tráfico que *sale* de los servicios, es decir, en función de servicios de origen y contenedores de destino.

Objeto `snapshot`. Un DataFrame que expone el estado de todos los contenedores para todos los servicios en el momento de parada del experimento, es decir, en la ejecución del comando `wotemu stop`. El estado incluye campos tales como la fecha de creación, el identificador del nodo del swarm en el que se ejecuta el contenedor y un *flag* para indicar el estado de error. Esto implica interactuar con la API de Docker para filtrar los servicios del stack, las tareas de los servicios y recuperar de manera centralizada la cola de los logs de cada contenedor. Los datos de *snapshot* son necesarios para, por ejemplo, construir la línea temporal de contenedores dentro del experimento. El bloque de código 4.3 muestra un ejemplo representativo en el que se ha truncado el tamaño de los logs por claridad.

```
{
    "columns": ["desired_state", "is_running", "is_error", "task
       ↪ ", "task_id", "node_id", "service_id", "created_at", "
       ↪ updated_at", "container_id", "logs", "stopped_at"],
    "data": [
        [
            "running",
            true,
            false,
            "broker.1.sc51l5uolachb9icl9aaxquhn",
            "sc51l5uolachb9icl9aaxquhn",
            "niow49d7d45roqdgt3nl61v2s",
            "0g35qp9b723tf58ql96zeg4i9",
            "2021-05-28T20:44:28.324Z",
            "2021-05-28T20:44:35.060Z",
            "64244b4e83c1",
            "2021-05-28 21:03:09 broker.1.
               ↪ sc51l5uolachb9icl9aaxquhn wotemu.utils[1]
               ↪ DEBUG /usr/bin/cgget -v -r memory.
               ↪ usage_in_bytes /: b'/:\\n86720512\\n\\n...",
            "2021-05-28T21:05:28.100Z"
        ],
        ...
    ],
    "index": [0, ...]
}
```

Código 4.3: Ejemplo objeto `snapshot`

Objeto `tasks.<task>.packet`. El DataFrame que contiene la serie temporal detallada de paquetes de red que fueron enviados y recibidos por las implementaciones de Protocol Binding expuestas en el objeto `WoT` inyectado por el emulador a la aplicación. La información base la proporciona TShark [83], un analizador de red de calidad y relevancia contrastada. En el ejemplo del bloque de código 4.4 se puede ver que se incluyen los identificadores del protocolo de capa más alta, del protocolo de transporte, los puertos origen y destino, el tamaño del paquete en bytes y los nombres de contenedor, tarea y red dentro del contexto del experimento.

Es importante destacar que las buenas prácticas indican que las comunicaciones entre contenedores en el stack deben llevarse a cabo utilizando los nombres de servicio internos; la alternativa implicaría utilizar un servicio de nombres externo (complejidad añadida) o apoyarse en direcciones fijas predefinidas (poco escalable y sostenible). Según el diseño de referencia de Docker swarm, un servicio (*swarm service*) encapsula un conjunto elástico de contenedores (*swarm tasks*). Esto se refleja en que los nombres de servicio resuelven a una IP virtual (VIP) en vez de a una dirección IP de un contenedor particular. Las comunicaciones a esas VIP se distribuyen en *Layer 4* (transporte) de manera balanceada entre los contenedores del servicio. Esto implica que no siempre es posible trazar los contenedores origen y destino de cada uno de los paquetes, ya que existen paquetes para los que el kernel reporta una VIP de servicio como dirección IP origen o destino. Sin embargo, si que se puede conocer de manera consistente el tráfico entre servicios, dado que el emulador encapsula todos los nodos en una capa que reporta automáticamente su servicio y su VIP al inicio del experimento para permitir la traducción posterior.

Objeto `tasks.<task>.info`. Un objeto generado por la capa decoradora del emulador de manera automática durante el proceso de inicialización de todos los contenedores. Contiene información relevante del contenedor que es necesaria para la construcción de varias vistas del informe. Por ejemplo, el modelo de CPU es necesario para poder agrupar las distribuciones de uso de CPU, las VIP del servicio permiten la agrupación a la hora de calcular el volumen de tráfico en servicios, los límites de recursos computacionales se utilizan para proporcionar una referencia a las series temporales que se capturan en medidas absolutas, y la representación de las redes en notación *Classless Inter-Domain Routing* (CIDR) permite categorizar apropiadamente los paquetes. El bloque de código 4.5 contiene un ejemplo representativo de este tipo de objetos.

Objeto `tasks.<task>.interaction`. Un DataFrame que contiene la serie temporal de interacciones WoT (e.g. invocación de acciones, lectura de propiedades) para todas las Things expuestas y consumidas por el contene-

dor en cuestión. Esto es posible gracias a que el objeto `WoT` proporcionado por el emulador está decorado de manera transparente para capturar y analizar todas las interacciones, respetando la interfaz del framework WoT que expone las funcionalidades de gestión de Things. En el bloque de código 4.6 se puede ver el objeto generado por la interacción de suscripción a un evento llamado `jpgVideoFrame`.

Objeto `tasks.<task>.system`. En este DataFrame se exponen las series temporales de utilización de recursos computacionales (CPU y memoria), lo que permite detectar los contenedores que están excesivamente sobrecargados, o que muestran un comportamiento inestable (por ejemplo, con fugas de memoria). El bloque de código 4.7 muestra un ejemplo de la manera en la que se representan los datos de sistema.

```
{
    "columns": ["len", "src", "dst", "proto", "transport", "time
        ↪ ", "srcport", "dstport", "src_task", "src_service", "
        ↪ dst_task", "dst_service", "network"],
    "data": [
        [
            2862,
            "10.0.210.6",
            "10.0.210.12",
            "tcp",
            "tcp",
            1622234822.7052321,
            80.0,
            60440.0,
            "camera.1.oc3kft1s7yzj0kip1piu98j25",
            "camera",
            "detector.1.reg10w741bkmo62w7i7thyllo",
            "detector",
            "field_loc1"
        ],
        ...
    ],
    "index": [
        ["2021-05-28T20:47:02.705Z", "eth0"],
        ...
    ]
}
```

Código 4.4: Ejemplo objeto `tasks.<task>.packet`

```
{
    "boot_time": 1528875907.0,
    "constraints": {"cpu_percent": 26.59, "mem_limit_mb":
       ↪ 256.0},
    "container_id": "1b565d12f153",
    "cpu_count": 8,
    "cpu_model": "Intel(R) Core(TM) i7-6770HQ @ 2.60GHz",
    "env": {"REDIS_URL": "redis://redis", ...},
    "hostname": "camera.1.oc3kft1s7yzj0kip1piu98j25",
    "mem_total": 33615335424,
    "net": {
        "eth0": [
            {
                "address": "10.0.210.6",
                "broadcast": "10.0.210.255",
                "family": 2,
                "netmask": "255.255.255.0",
                "ptp": null
            }
        ]
    },
    "networks_cidr": {"field_loc1": ["10.0.210.0/24"]},
    "process": {"1": {"name": "wotemu", "username": "root"}},
    "python_version": "3.8.5",
    "service_vips": {"field_loc1": "10.0.210.5"},
    "task_id": "oc3kft1s7yzj0kip1piu98j25",
    "time": 1622234799.6127222,
    "uname": {"processor": "x86_64", ...}
}
```

Código 4.5: Ejemplo objeto `tasks.<task>.info`

```
{
    "columns": ["event", "time", "host", "class", "subscription
        ↪ ", "item", "latency", "error", "result"],
    "data": [
        [
            "on_subscribe",
            1622234801.2105045,
            "camera.1.oc3kft1s7yzj0kip1piu98j25",
            "ExposedThing",
            "2cf2350a8bab4a309ab82ebfe28a9a0b",
            null,
            null,
            null,
            null
        ],
        ...
    ],
    "index": [
        [
            "2021-05-28T20:46:41.210Z",
            "urn:org:fundacionctic:thing:wotemu:camera",
            "jpgVideoFrame",
            "subscribeevent"
        ],
        ...
    ]
}
```

Código 4.6: Ejemplo objeto `tasks.<task>.interaction`

```
{
    "columns": ["time", "cpu_percent", "cpu_percent_constraint",
        ↪  "mem_mb", "mem_percent"],
    "data": [
        [1622234805.013772, 25.5, 96.0, 136.44, 53.3],
        ...
    ],
    "index": [
        "2021-05-28T20:46:45.013Z",
        ...
    ]
}
```

Código 4.7: Ejemplo objeto `tasks.<task>.system`

Capítulo 5

Resultados experimentales

En este capítulo se describen los experimentos que se han llevado a cabo para validar formalmente las contribuciones de los diseños presentados en los capítulos anteriores 3 y 4:

- La **sección 5.1** describe un experimento para evaluación del rendimiento de la implementación experimental de los bloques fundamentales W3C WoT. Se analiza de manera exhaustiva el rendimiento de las implementaciones de Protocol Binding en el contexto del modelo de interacciones WoT.

- La **sección 5.2** presenta un caso de uso realista en el que se aprovechan las funcionalidades novedosas del emulador para análisis y optimización de la arquitectura de una aplicación diseñada sobre el modelo WoT.

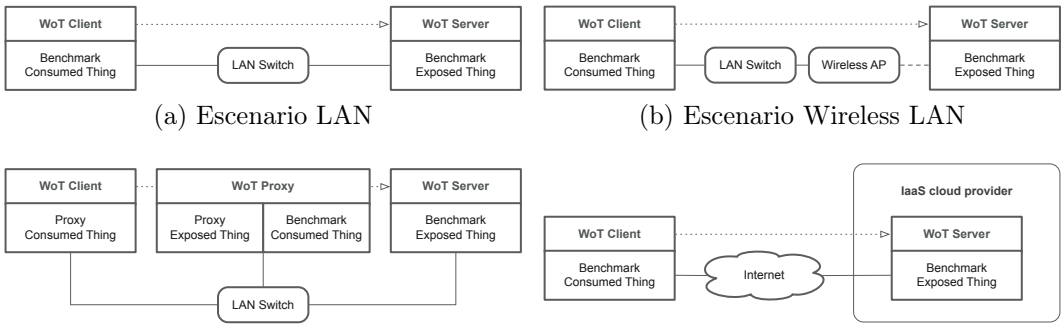
5.1. Evaluación de rendimiento del framework WoT

5.1.1. Definición del experimento

En este experimento de evaluación de rendimiento se definen dos entidades diferenciadas. Concretamente, escenarios (*scenario*) y configuraciones (*configuration*):

- Un **escenario** representa la arquitectura de alto nivel de los actores en un experimento. Es decir, la topología, las conexiones entre nodos y los medios físicos de conexión.

- Una **configuración** representa una instancia concreta de un escenario. Es decir, define los detalles concretos que están sujetos a configuración respecto a los actores de un escenario. Las configuraciones son los entornos experimentales que se ejecutan finalmente, y como tal vienen indicados en los resultados (véase la tabla 5.2 para un resumen de las figuras y tablas de resultados).



(a) Escenario LAN (b) Escenario Wireless LAN

(c) Escenario Proxy (d) Escenario Cloud

Figura 5.1: Escenarios de evaluación del framework

Se consideran **cuatro escenarios** dentro del experimento, que han sido diseñados para cubrir los escenarios descritos en la especificación de arquitectura WoT del W3C [61]. La figura 5.1 presenta los diagramas que representan estos cuatro escenarios, descritos con más detalle en la lista a continuación.

Configuración	Escenario
Constrained	Wireless LAN (figura 5.1b)
Cloud	Cloud (figura 5.1d)
Proxy MQTT	Proxy (figura 5.1c)
Proxy HTTP	Proxy (figura 5.1c)
Single-Board Computer	LAN (figura 5.1a)
Unconstrained	LAN (figura 5.1a)

Cuadro 5.1: Relación entre *escenarios* y *configuraciones*

LAN Este es el escenario más simple posible dentro de los límites razonables. El *WoT Client* está directamente conectado por cable a un *WoT Server* a través de un switch en una red de área local. Es apropiado en aquellos casos en los que se quiere reducir el impacto del medio de comunicación, es decir, de los posibles cuellos de botella que emanan de una red inalámbrica o una conexión pública a Internet.

Wireless LAN Este escenario es similar al escenario *LAN*, con la única diferencia de que la conectividad desde el switch de red hasta el *WoT Server* es sobre un medio inalámbrico, concretamente WiFi (802.11n). Esto representa una situación más realista, ya que la mayoría de dispositivos comúnmente localizados en la capa de sensores operan con protocolos inalámbricos.

Proxy El escenario *Proxy* está diseñado para representar una situación en la que un nodo proporciona soporte únicamente para un protocolo concreto, ya sea por limitaciones de rendimiento, funcionalidad del software u optimización de recursos. Por delante de ese nodo se encuentra un gateway con capacidades más avanzadas que es capaz de exponer Things *extendidas*, es decir, que son versiones *proxy* de la Thing original y soportan el conjunto completo de protocolos.

Cloud Los escenarios anteriores están enfocados en un entorno local, que podría representar un despliegue doméstico o industrial. Sin embargo, los servicios de computación en la nube son un componente altamente relevante en despliegues IoT. Por esta razón, se incluye un escenario para representar esta situación de manera simplificada. El *WoT Server* se despliega en una instancia de un proveedor de servicios *Infrastructure as a Service* (IaaS) lo que a su vez implica que la conectividad con el *WoT Client* ocurre sobre Internet. La presencia de Internet inyecta un nivel significativo de incógnita dentro del escenario experimental, por lo que la conexión se monitoriza de manera paralela y separada para verificar que no aparecen pérdidas de rendimiento inesperadas por razones que no tienen relación con el experimento en sí.

Por otro lado, se definen **seis configuraciones** que emanan de los escenarios previamente comentados. Las relaciones entre configuraciones y escenarios se pueden ver en la tabla 5.1. Estas configuraciones se han diseñado para ser lo suficientemente representativas de situaciones reales, manteniendo un volumen razonable de resultados en el experimento que faciliten su discusión y comprensión.

Constrained Una configuración diseñada para representar el despliegue de aplicaciones WoT en dispositivos de bajo coste con capacidades computacionales limitadas, lo que resulta una ocurrencia común en el dominio del IoT y la WoT. Este tipo de dispositivos se encuentran normalmente en las capas bajas de sensores o *near-end*, y no resultan adecuados como nodos centrales en situaciones que requieren el manejo de grandes volúmenes de datos. Podría argumentarse que con el paso del tiempo

se acumulan las mejoras de rendimiento y eficiencia en hardware, por lo que se podrían relajar las restricciones de rendimiento en la capa de sensores; sin embargo, la tendencia apunta a que esto no va a ser así en un futuro cercano. La implementación se lleva a cabo sobre una placa de desarrollo IoT Onion Omega 2+, que dispone de un microprocesador RISC a 580 MHz y 128 MB de memoria física. Además, se han añadido 128 MB de memoria swap adicional en un disco externo USB 2.0 para permitir la ejecución de cargas razonablemente complejas que requieren un mayor tamaño de memoria.

Single-Board Computer (SBC) Esta configuración persigue representar el rendimiento aproximado de un gateway de coste medio-bajo, con un rendimiento comparable a los que se encuentran comúnmente en entornos domésticos (e incluso industriales en algunos casos). El *WoT Server* se implementa sobre un SBC con un procesador ARM quad-core de 1.2 GHz y 1 GB de memoria física, concretamente, la popular Raspberry Pi 3 Model B.

Unconstrained Esta configuración representa una situación ideal en la que no hay limitaciones de recursos, es decir, que los cuellos de botella tienen su origen mayoritario en el diseño del framework WoT, en vez de en los recursos computacionales o de red. El *WoT Server* se implementa sobre un nodo con una CPU Intel x86 de 4 cores y 8 hilos, con frecuencia base de 2.6 GHz y frecuencia turbo de 3.7 GHz, además de 32 GB de memoria física y almacenamiento SSD.

Cloud Esta configuración representa el rendimiento de una aplicación WoT cuando se despliega sobre un proveedor IaaS (*Infrastructure as a Service*) en la nube. En este caso el *WoT Server* se lanza en una instancia con 1 hilo de CPU y 1 GB de memoria. Se ha prestado especial atención a la selección de un proveedor que no aplica *throttling* de CPU, práctica que consiste en disminuir el rendimiento de la CPU en cargas continuadas de trabajo con el objetivo de optimizar la distribución del uso entre distintos clientes. Dado que la implementación de protocol binding de MQTT necesita un broker independiente, se ha desplegado este servicio en una instancia en la nube separada con las mismas especificaciones. Ambas instancias se encuentran el mismo centro de datos, lo que permite descartar razonablemente el impacto de la latencia en sus comunicaciones.

Proxy MQTT En esta configuración, el *WoT Server* es el mismo dispositivo hardware que en la configuración *SBC* (Raspberry Pi 3B), además,

solo está habilitado el protocolo MQTT. Se utiliza MQTT por su popularidad en el contexto del IoT y su relativamente pequeña huella computacional y de red. El *WoT Proxy* se implementa a su vez sobre el mismo servidor de la configuración *Unconstrained* con todos los protocolos del runtime WoT (HTTP, MQTT, Websockets y CoAP).

Proxy HTTP Esta configuración es similar al caso de *Proxy MQTT*. La única diferencia es que el *WoT Server* está configurado para utilizar el binding HTTP en vez de MQTT. La razón detrás de esta variación, que puede parecer inicialmente redundante, es que HTTP es el protocolo que sirve como cimiento de la Web, lo que le asigna una relevancia especial y justifica su evaluación por separado.

En el bloque de código 5.1 se muestra el documento Thing Description que representa la *Benchmark Thing* referenciada en el diagrama de escenarios de la figura 5.1. Esta Thing expone cuatro interacciones, descritas posteriormente, que sirven para obtener una medida del rendimiento en cada una de las configuraciones a través de su comunicación con el cliente WoT. El documento está serializado en formato JSON.

Propiedad `currentTime` Una propiedad que retorna el timestamp UNIX actual reportado por el sistema.

Acción `measureRoundTrip` Esta acción persigue simular un procedimiento general. Para ello, cuando se recibe una invocación, se entra en un bucle que ejecuta operaciones en cada iteración cuyo único objetivo es ocupar espacio en la CPU. El tiempo de duración de este bucle viene determinado por una distribución normal, por lo que la acción tiene dos parámetros de entrada: un parámetro `mu` (μ) y un parámetro `sigma` (σ) que representan la media y la desviación estándar de la distribución respectivamente. La acción devuelve un objeto que contiene los timestamp del momento de llegada y de retorno de la invocación.

Acción `startEventBurst` Genera un conjunto de emisiones del evento descrito a continuación `burstEvent`. La secuencia de eventos se modela como un proceso Poisson, de manera que el intervalo entre eventos consecutivos es aleatorio a la vez que se mantiene una tasa media predeterminada. La acción tiene dos parámetros de entrada: un parámetro `lambd` (λ), que representa la tasa deseada de eventos por segundo, y un parámetro `total`, que representa el número total de eventos que se producirán en esta secuencia. Devuelve un resultado vacío cuando la secuencia termina.

```
{
  "actions": {
    "measureRoundTrip": {
      "idempotent": false,
      "input": {
        "type": "object"
      },
      "output": {
        "type": "object"
      },
      "safe": true
    },
    "startEventBurst": {
      "idempotent": false,
      "input": {
        "type": "object"
      },
      "safe": true
    }
  },
  "events": {
    "burstEvent": {
      "data": {
        "type": "object"
      }
    }
  },
  "id": "urn:org:fundacionctic:thing:benchmark",
  "name": "Benchmark Thing",
  "properties": {
    "currentTime": {
      "readOnly": true,
      "type": "integer"
    }
  }
}
```

Código 5.1: Thing Description de *Benchmark Thing*

Evento `burstEvent` Este es el evento que se emite en el transcurso de las invocaciones a `startEventBurst`. El cuerpo de cada evento emitido es un objeto formado por los siguientes componentes: el identificador único de la secuencia actual, el índice de esta emisión en particular dentro de la secuencia, el timestamp del momento inicial de la secuencia, y un valor binario para indicar que es el último evento de la secuencia.

5.1.2. Métricas de rendimiento

En esta sección se describen las métricas consideradas en el experimento de evaluación de rendimiento del framework cuyos resultados se presentan y discuten en la sección 5.1.3.

Las métricas hacen referencia a timestamps T (momentos en el tiempo) en formato UNIX con precisión en milisegundos. Todos los relojes se sincronizan usando el mismo conjunto de servidores NTP con el objetivo de reducir los errores derivados de las desviaciones entre cada uno de los nodos que participan en el experimento.

La **latencia de invocación de acciones** (*action invocation latency*) se define como:

$$L_a = (T_{res} - T_{req}) - (T_{ret} - T_{arr}) \tag{5.1}$$

T_{res} (***response***) Timestamp en el que el resultado de la invocación llega al cliente.

T_{req} (***request***) Timestamp en el que el cliente envía la petición.

T_{ret} (***return***) Timestamp en el que el proceso de la acción finaliza y devuelve el resultado en el servidor.

T_{arr} (***arrival***) Timestamp en el que el servidor inicia la ejecución del proceso de acción.

El resultado de $T_{ret} - T_{arr}$ representa el tiempo de ejecución del proceso de acción. Esto se resta de la latencia que percibe el cliente $T_{res} - T_{req}$ para considerar únicamente las latencias que resultan de la implementación de protocol binding y los mecanismos internos del framework.

La **latencia de eventos** (*event latency*) se define como:

$$L_e = T_{rec} - T_{emi} \tag{5.2}$$

T_{rec} (***received***) Timestamp en el que el evento llega al cliente.

T_{emi} **(*emission*)** Timestamp el el que el servidor emite el evento.

La **latencia de lectura de propiedades** (*property read latency*) se define como:

$$L_p = T_{res} - T_{req} \tag{5.3}$$

T_{res} **(*response*)** Timestamp en el que el cliente recibe la respuesta.

T_{req} **(*request*)** Timestamp en el que el cliente envía la petición.

El **volumen de transferencia de datos** (*data transfer volume*) se define como la cantidad total de datos transmitidos por el cliente a través de la interfaz de red tanto en dirección *upstream* como *downstream*. El filtro de captura de datos se basa en el puerto de la comunicación a nivel de capa de transporte. Concretamente, se consideran ambos protocolos de capa de transporte (UDP y TCP) ya que CoAP se transporta sobre UDP, mientras que el resto de protocolos se transmiten sobre TCP. El filtro se define dinámicamente en base al puerto que está definido en el documento TD.

La **tasa de error de invocación de acciones** (*action invocation error*) se define como:

$$E_a = 1 - (N_{ok}/N_{total}) \tag{5.4}$$

N_{ok} Número de invocaciones que terminan con éxito.

N_{total} Número total de invocaciones.

La **tasa de pérdida de eventos** (*event loss*) se define como:

$$E_e = 1 - (N_{rec}/N_{total}) \tag{5.5}$$

N_{rec} Número de eventos recibidos en el lado del cliente.

N_{total} Número total de eventos emitidos por el servidor.

La **tasa de error de lectura de propiedades** (*property read error*) se define como:

$$E_p = 1 - (N_{ok}/N_{total}) \tag{5.6}$$

N_{ok} Número de peticiones con éxito.

N_{total} Número total de peticiones de lectura.

Como se puede ver, no se consideran métricas específicas para los verbos *property write* y *property observe*. La razón detrás de esta decisión de diseño es que están representados por *property read* y *observe event* respectivamente. Por un lado, la implementación de escritura de propiedades es muy similar a la de lectura. Por otro lado, existen muchos paralelismos entre las implementaciones de observación de propiedades y eventos, por ejemplo, se utiliza la misma implementación de cola de eventos asíncronos dentro del framework. De aquí se deriva que el rendimiento de *property write* y *property observe* debe resultar suficientemente similar y comparable a *property read* y *observe event*.

5.1.3. Resultados y discusión

Esta sección presenta los resultados obtenidos de la ejecución de las configuraciones experimentales descritas en la sección 5.1.1. Se incluyen resultados relacionados con tres métricas de alto nivel (véase sección 5.1.2 para más detalles sobre las métricas):

- La **latencia** proporciona una medida objetiva de la *responsividad* de las aplicaciones, y es uno de los factores con mayor impacto en la experiencia de usuario. Una latencia alta tiene un impacto negativo muy notable en todo el sistema, pudiendo llegar a causar una cascada de *timeouts*. Se persigue mantener una latencia mínima en todos los casos.
- El **volumen de transferencia de datos** da una indicación de la carga de la red, y es especialmente importante en despliegues en el dominio del IoT, que suelen apoyarse sobre redes con tarificación por volumen. Las decisiones de diseño detrás de protocolos distintos pueden dar lugar a huellas de volumen de transferencia muy diferentes para la misma interacción de alto nivel.
- Idealmente, se busca evitar la aparición de cualquier tipo de **error**, ya sea originado por la conexión de red, la aplicación o la infraestructura. Sin embargo, las condiciones de un despliegue real hacen que esto sea una tarea muy compleja. En este experimento se busca identificar la manera en la que las distintas cargas de trabajo impactan sobre las tasas de error de las implementaciones de protocol binding.

La ejecución se ha llevado a cabo sobre la implementación experimental del framework publicada en Zenodo [63] (versión `0.11.0`). Alternativamente,

el historial completo del código fuente, incluyendo los últimos desarrollos y actualizaciones, está disponible en GitHub [42].

Verbo	Tasa de error	Latencia	Transferencia
Property Read	Tabla 5.4	Figura 5.4	Figura 5.5
Action Invoke	Tabla 5.3	Figura 5.2	Figura 5.3
Event Observe	Tabla 5.5	Figura 5.6	Figura 5.7

Cuadro 5.2: Resumen de resultados de evaluación del framework

En la tabla 5.2 se puede ver un resumen de las tablas y figuras que presentan los resultados de latencia, volumen de transferencia y tasas de error. Las tablas de errores solo muestran entradas para aquellas implementaciones de protocol binding que presentan valores distintos a cero. Por otro lado, las figuras de latencia solo muestran valores correspondientes a interacciones con éxito (las interacciones que generan errores se descartan en este caso).

A modo de aclaración, todas las figuras de latencia referenciadas en la tabla 5.2 se representan utilizando un tipo de gráfico denominado *letter-value plot* [92], que es razonablemente similar al bien conocido boxplot. La razón que lleva a evitar boxplot en este caso es que las figuras de latencia aparecen con cajas muy estrechas, bigotes muy largos y un número alto de *outliers*, es decir, un aspecto que no ayuda a su comprensión y añade bastante ruido. Los letter-value plots encajan mejor en casos de uso con conjuntos de datos grandes que se benefician de la representación de la distribución de las muestras en los bigotes. Concretamente, tienen las siguientes características:

- La caja situada en la mitad (la más ancha) es equivalente a la caja de un boxplot. Es decir, los límites se establecen en los cuartiles primero y tercero, mientras que la línea horizontal que la atraviesa representa la mediana.
- Las cajas sucesivas, por encima y por debajo de la caja intermedia, representan los cuantiles inferior y superior de orden exponencialmente creciente (8, 16, 32, ...). El ancho de las cajas decrece de manera lineal.
- El número de cajas (k) se determina utilizando la regla conocida como *Tukey*: $k = \lfloor \log_2 n \rfloor - 3$, donde n es el número de muestras.
- Todas las muestras que caen fuera de las cajas se consideran *outliers* y se representan con un diamante.

Además, es importante destacar que en este experimento se demuestran de manera implícita los beneficios de desarrollo de los bloques fundamentales

de las especificaciones W3C WoT: arquitectura WoT, Scripting API, Protocol Bindings y Thing Description. El tamaño del código fuente del experimento completo se encuentra en el orden de cientos de líneas de código, donde la mayoría es lógica de negocio ad-hoc en vez de código de configuración WoT. Alcanzar una funcionalidad comparable sin los bloques fundamentales W3C WoT implicaría una inversión de recursos de diseño y desarrollo significativamente mayor.

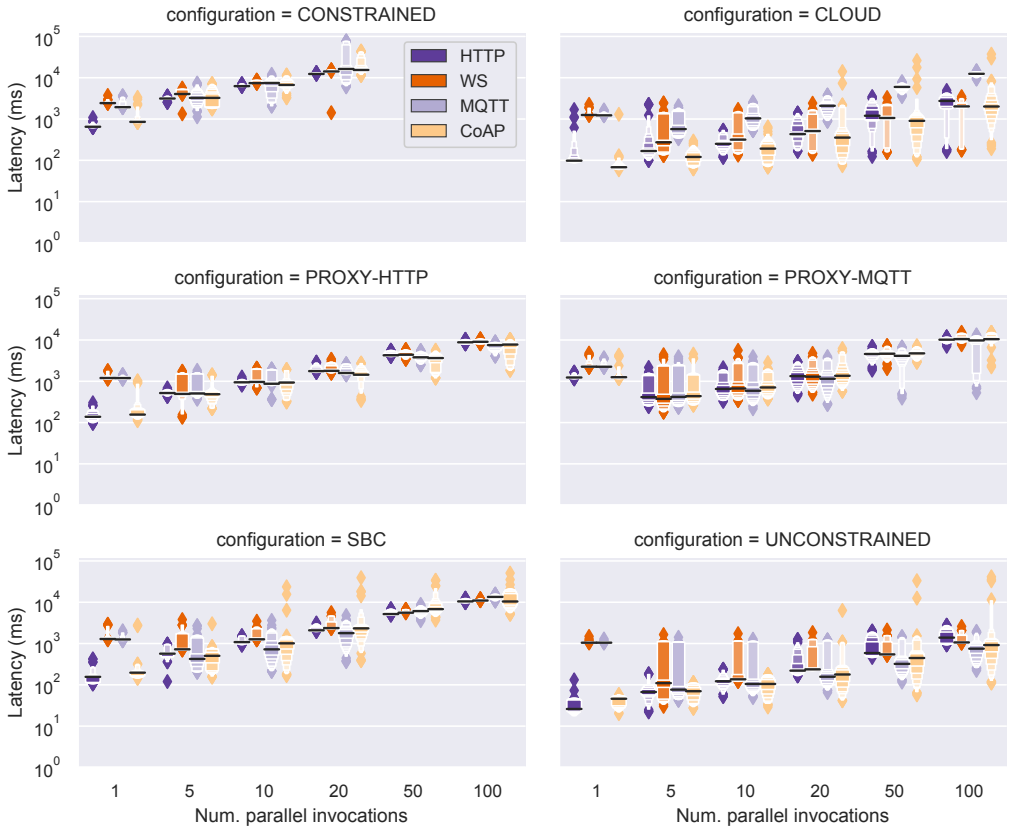


Figura 5.2: Latencia de invocación de acciones

Los **resultados del verbo de acción** se muestran en la figura 5.2 de latencia de invocación de acción, la figura 5.3 de volumen de transferencia de datos y la tabla 5.3 de tasas de error de invocación. En total se incluyen datos de 1000 invocaciones de la acción `measureRoundTrip`. Las invocaciones se generan en paralelo en ráfagas de tamaño creciente de 1, 5, 10, 20, 50 y 100. La razón de esta estrategia es que las acciones son procedimientos de larga duración, por lo que se ha estimado interesante caracterizar el

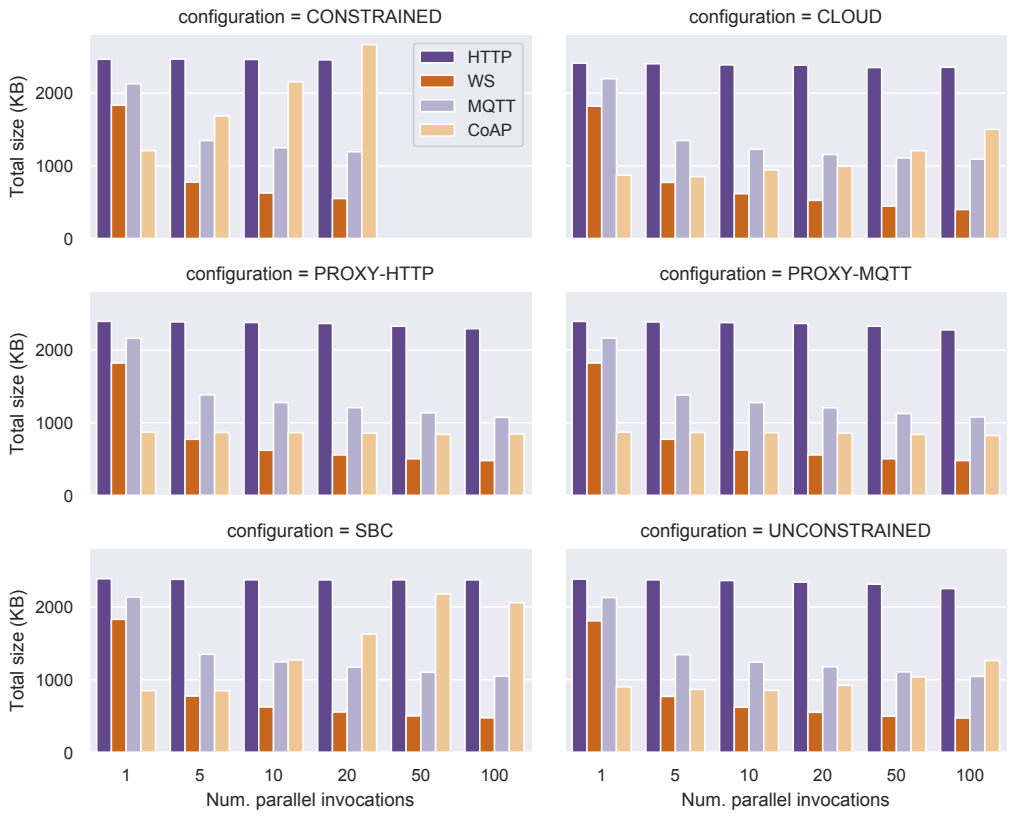


Figura 5.3: Transferencia de datos en invocación de acciones

	Num. invocaciones paralelas					
	1	5	10	20	50	100
CONSTRAINED						
CoAP	0.001	0.018	0.029	0.126	-	-
PROXY-HTTP						
CoAP	0.000	0.000	0.000	0.000	0.001	0.000

Cuadro 5.3: Tasa de error para invocación de acciones

comportamiento de las implementaciones de Protocol Binding usando ráfagas de invocaciones paralelas en vez de flujos constantes de invocaciones a una frecuencia predeterminada. El parámetro `mu` (μ) es 0.0 y `sigma` (σ) es 1.0 para todas las invocaciones. Estos parámetros resultan en un tiempo de

ejecución del proceso de la acción en el orden de 10^2 ms. El resultado es un compromiso razonable para un tiempo extendido de ejecución, que es un orden de magnitud mayor que el de las interacciones de lectura de propiedades, Además, la varianza es suficientemente baja como para que la mayoría de las invocaciones en una ráfaga finalicen al mismo tiempo, lo que añade más carga al Protocol Binding.

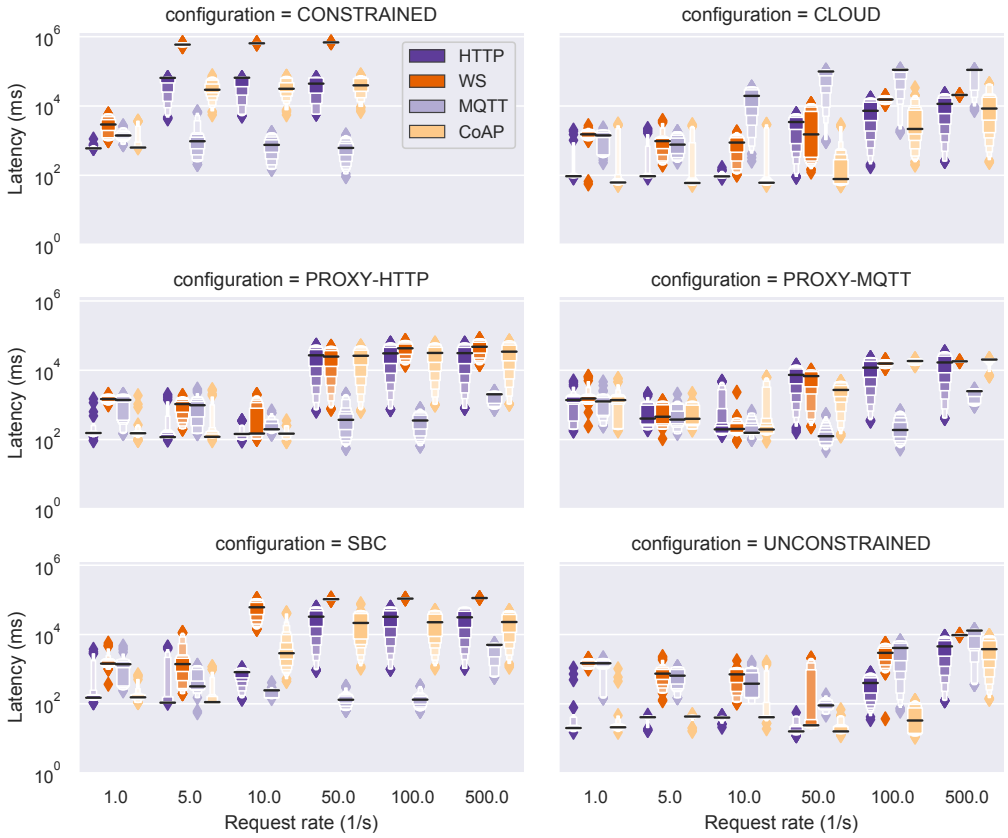


Figura 5.4: Latencia de lectura de propiedades

Los **resultados del verbo de propiedad** se muestran en la figura 5.4 de latencia de lectura de propiedad, la figura 5.5 de volumen de transferencia de datos y la tabla 5.4 de tasas de error de lectura de propiedades. Los resultados corresponden a 1000 peticiones de lectura de la propiedad `currentTime`. Las peticiones se generan en el cliente con tasas crecientes de 1, 5, 10, 50, 100 y 500 peticiones/s.

Los **resultados del verbo de evento** se muestran en la figura 5.6 de latencia de evento, la figura 5.7 de volumen de transferencia de datos y la

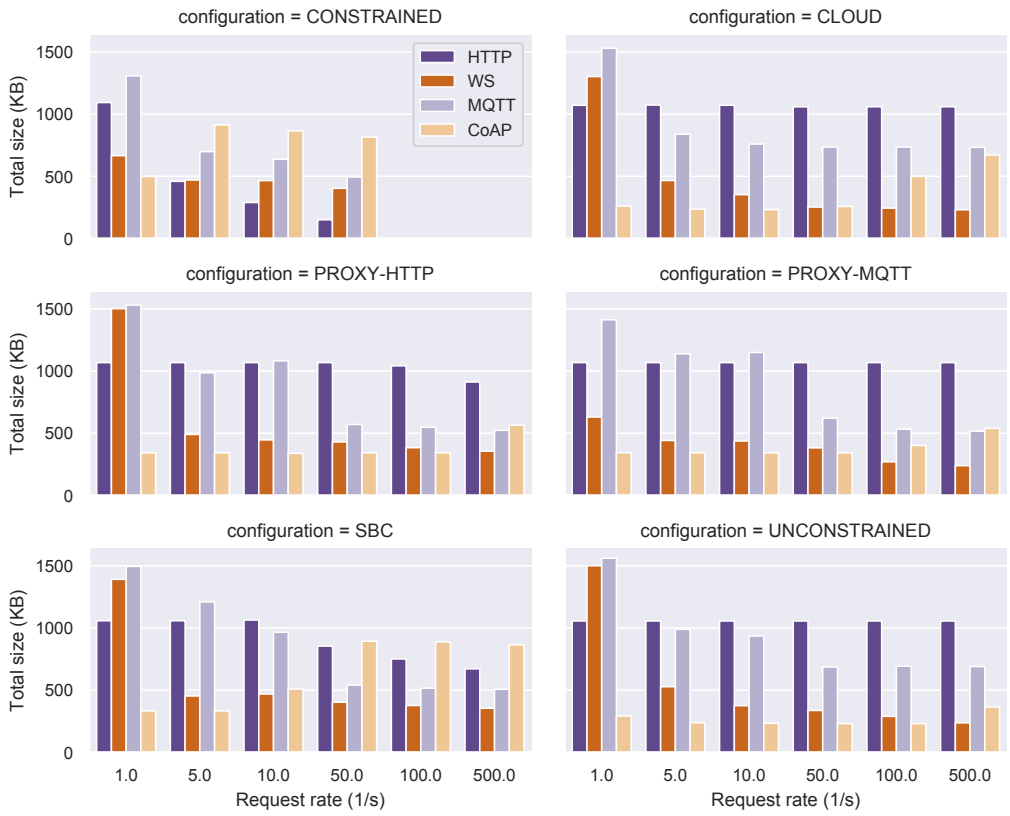


Figura 5.5: Transferencia de datos en lectura de propiedades

tabla 5.5 de tasas de perdida de eventos. Los resultados incluyen seis series diferenciadas de emisiones del evento `burstEvent`. Cada serie se inicializa con una invocación de la acción `startEventBurst`, donde el parámetro `total` se fija a 1000 para todas las invocaciones y el parámetro `lambd` (λ) (tasa de eventos objetivo en el servidor) crece por la serie de 1, 5, 10, 50, 100 y 500 eventos/s.

En el resto de este capítulo se discuten los resultados enfocados en cada implementación de Protocol Binding en particular (véase resumen de referencias de figuras en tabla 5.2).

HTTP

Como se comenta en la sección 3.2.1, la implementación de Protocol Binding de HTTP se basa en el patrón *long-polling* para los verbos de interacción que requieren comunicación iniciada desde el servidor. Esta aproximación no

	Tasa de peticiones (1/s)					
	1.0	5.0	10.0	50.0	100.0	500.0
CONSTRAINED						
HTTP	0.000	0.570	0.730	0.860	-	-
CoAP	0.077	0.647	0.794	0.887	-	-
CLOUD						
CoAP	0.000	0.000	0.000	0.000	0.000	0.002
PROXY-HTTP						
HTTP	0.000	0.000	0.000	0.000	0.024	0.146
SBC						
HTTP	0.000	0.000	0.000	0.200	0.300	0.370
CoAP	0.000	0.000	0.001	0.397	0.470	0.562

Cuadro 5.4: Tasa de error para lectura de propiedades

parece ser adecuada para los casos en los que hay una tasa alta de mensajes. La tabla 5.5 muestra pérdidas para todas las condiciones, especialmente para las tasas más altas de 100 y 500 eventos/s y los escenarios donde la latencia de red tiene mayor presencia (por ejemplo, el escenario *cloud*). El Protocol Binding HTTP está diseñado en torno al principio de la ausencia de estado, lo que tiene un impacto negativo en el rendimiento de los mensajes *push*. Concretamente, una vez que se ha enviado una respuesta que contiene un evento al cliente, el servidor del Protocol Binding HTTP elimina la suscripción internamente, lo que implica que todos los eventos que ocurren entre ese momento y la siguiente petición GET *long-polling* se pierden. Aunque el cliente envía esa petición GET en cuanto recibe el evento, existe un intervalo de pérdida inevitable.

Por otra parte, se puede identificar una degradación significativa de rendimiento en situaciones con tasas muy altas de peticiones por segundo en hardware de capacidades limitadas. Esto se muestra en la tabla 5.4 en los resultados de error de las configuraciones *SBC* y *constrained*. La mayoría de los errores HTTP se deben a los *timeouts* en las peticiones, lo que sugiere que el Protocol Binding HTTP es computacionalmente pesada. Si fuese aceptable para la aplicación, podrían mejorarse las tasas de error de lectura de propiedades incrementando los *timeout* por encima de los valores por defecto.

La implementación HTTP de Protocol Binding se adapta mejor a situaciones con una mayoría de interacciones según los verbos *property read*,

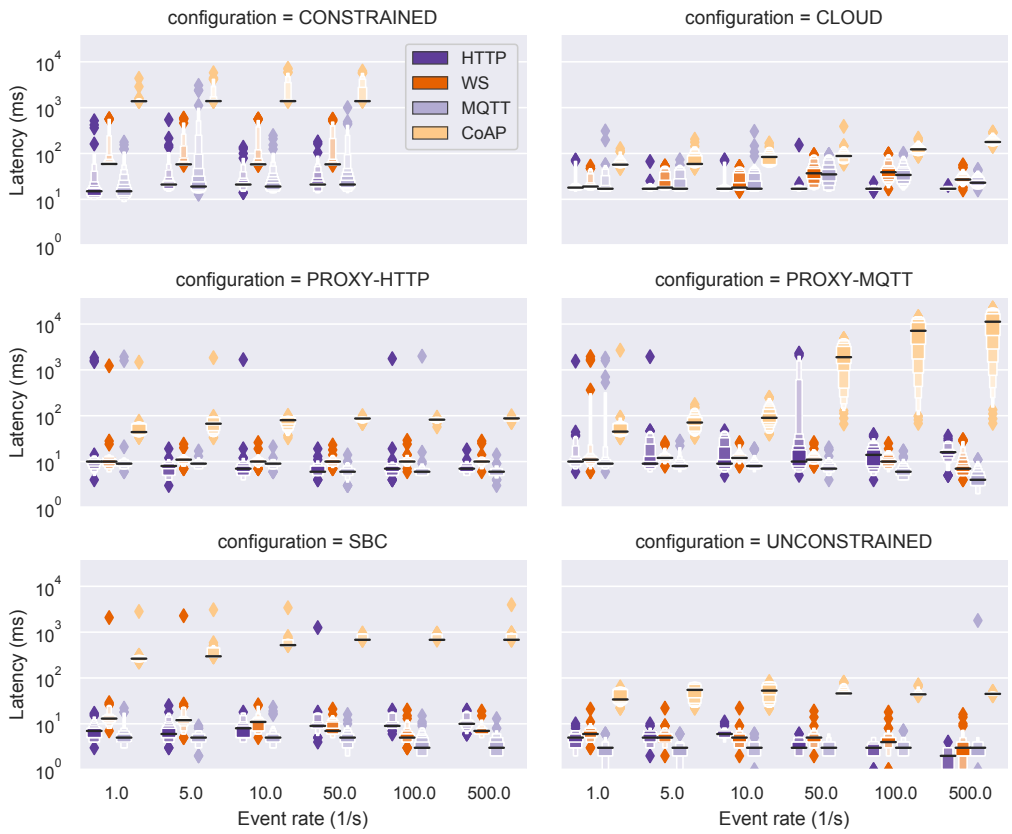


Figura 5.6: Latencia de recepción de eventos

property write y, en menor medida, *action invocation*. Estos verbos de interacción se encuentran más cercanos al modelo petición-respuesta de HTTP. Los resultados muestran rendimiento aceptable excepto para tasas muy altas en hardware limitado (figura 5.4). Sin embargo, deben tenerse en cuenta que los volúmenes de transferencia de datos son comparablemente superiores a otras implementaciones de Protocol Binding (figura 5.7).

El punto fuerte más destacable del Protocol Binding HTTP no es su rendimiento objetivo, que es aceptable en algunos casos y mejorable en otros, si no el hecho de que HTTP es el protocolo cimiento de la Web. HTTP es uno de los protocolos mejor conocidos, con presencia en una gran mayoría de los contextos, desde microcontroladores hasta servidores en la nube. Esta ubicuidad hace de HTTP una opción necesaria e interesante a considerar incluso en presencia de implementaciones de Protocol Binding con mejor rendimiento.

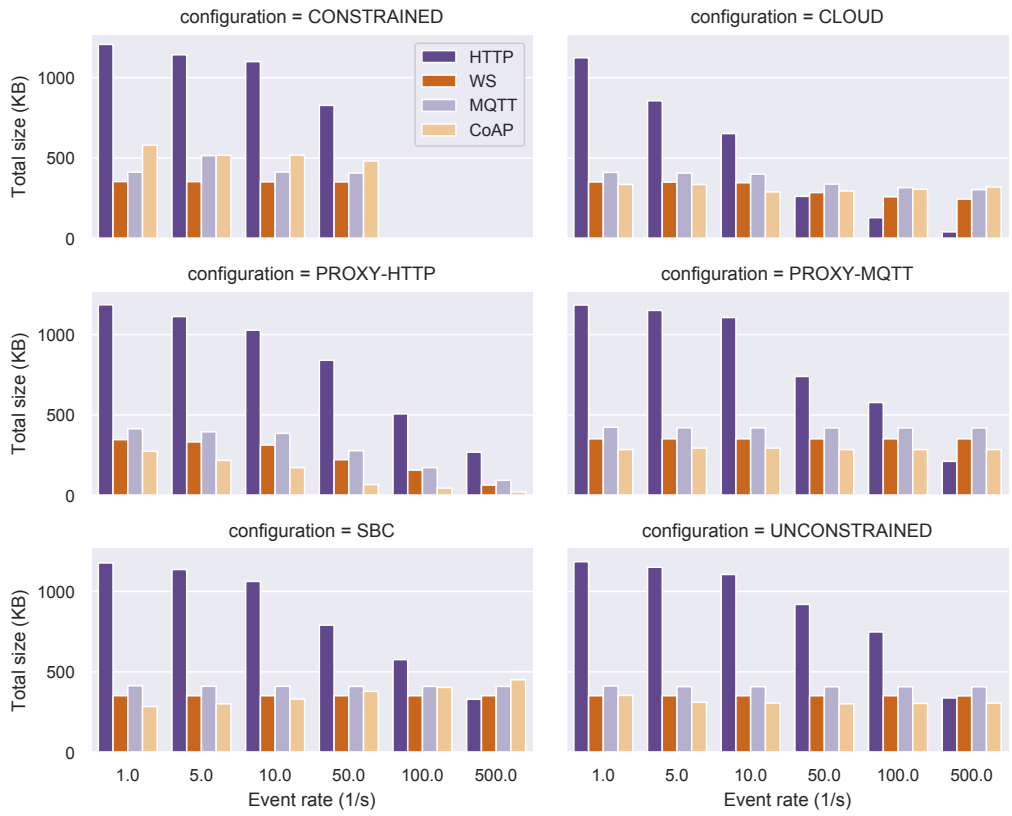


Figura 5.7: Transferencia de datos en recepción de eventos

Websockets

El protocolo Websockets también muestra un rendimiento mejorable en entornos con recursos computacionales limitados, que deriva de su alto coste computacional. Aunque no se observan errores en las peticiones de lectura de propiedad, si que se pueden ver latencias muy altas en el orden de 10^5 ms para tasas ≥ 5 en la configuración *constrained* (figura 5.4). Por otra parte, es uno de los Protocol Binding con un volumen más bajo de transferencia de datos, especialmente cuando se tienen en cuenta los resultados de error (puede haber implementaciones con volúmenes menores pero más errores).

Se puede observar que la arquitectura del escenario *proxy* tiene un impacto positivo en el rendimiento de Websockets y HTTP, a la vez que permite que dispositivos con recursos restringidos se limiten a implementaciones de Protocol Binding más ligeras (por ejemplo, MQTT). Las latencias de Websockets (figura 5.4) y los errores de HTTP (tabla 5.4) son menores en las configura-

	Tasa de eventos (1/s)					
	1.0	5.0	10.0	50.0	100.0	500.0
CONSTRAINED						
HTTP	0.011	0.062	0.099	0.322	-	-
CLOUD						
HTTP	0.070	0.291	0.461	0.786	0.896	0.969
PROXY-HTTP						
HTTP	0.013	0.074	0.145	0.301	0.580	0.779
WS	0.015	0.055	0.112	0.372	0.557	0.823
MQTT	0.023	0.060	0.082	0.340	0.592	0.781
CoAP	0.037	0.269	0.430	0.797	0.881	0.972
PROXY-MQTT						
HTTP	0.014	0.042	0.079	0.385	0.521	0.826
SBC						
HTTP	0.012	0.046	0.108	0.337	0.518	0.725
UNCONSTRAINED						
HTTP	0.003	0.032	0.070	0.227	0.372	0.718

Cuadro 5.5: Tasa de error para recepción de eventos

ciones del escenario *proxy* en comparación con la configuración *SBC*.

Las figuras 5.3 y 5.5 muestran que los volúmenes de transferencia de datos en Websockets disminuyen a medida que las condiciones son más exigentes, en comparación con el mismo número de invocaciones o peticiones bajo condiciones menos exigentes. La razón detrás de este comportamiento es la *reutilización de conexiones abiertas* dentro del cliente descrita en la sección 3.2.2. Las tasas más bajas derivan en que el cliente abre y cierra la conexión para cada petición o invocación porque una instancia termina antes de que se lance la siguiente. Esto mismo no se observa de manera igualmente clara en la figura 5.7 por el hecho de que la ráfaga completa de eventos se recibe siempre por una misma conexión independientemente de la tasa.

En las figuras 5.2 y 5.4 se puede ver que las latencias de Websockets son menores para condiciones más exigentes en algunos casos. Por ejemplo, en la figura 5.2 se puede ver que la mediana de la latencia de *action invocation* en la configuración *unconstrained* es $\sim 10^3$ ms para bloques de 1 invocación

paralela mientras que bloques de 5 y 10 invocaciones paralelas producen resultados de $\sim 10^2$ ms. Esto es otra consecuencia positiva de la optimización de reutilización de conexiones.

En conclusión, como podía ser esperable dadas sus características, la implementación de Protocol Binding Websockets es adecuada para verbos de interacción que dependen de comunicaciones iniciadas desde el servidor (*action invocation*, *property observe* y *event subscribe*) debido a los niveles reducidos de latencia y volúmenes de transferencia de datos (figuras 5.2, 5.3, 5.6 y 5.7). Por el contrario, no resulta aconsejable para escenarios donde se necesita mantener una tasa alta y sostenida de peticiones (*property read* y *property write*), especialmente para dispositivos con recursos limitados de computación (figura 5.4).

MQTT

La calidad de la conexión entre el broker y el cliente MQTT tiene un impacto notable en la latencia de acciones y propiedades, como se puede ver en la configuración *cloud* en las figuras 5.2 y 5.4. En este caso el cliente tiene que establecer múltiples conexiones a través de Internet con la instancia en la nube sobre la que está desplegada el broker. Por otra parte, el impacto en la latencia de eventos (figura 5.6) no es tan significativo porque la conexión solo tiene que establecerse una única vez.

Es importante destacar que los efectos positivos que derivan de la reutilización de conexiones, comentados en la sección de resultados de Websockets, también aplican en este caso de MQTT. Concretamente, volúmenes de transferencia de datos y latencias comparativamente más bajas para condiciones más exigentes, como se puede observar en las figuras 5.2, 5.3, 5.4 y 5.5.

La implementación MQTT es una opción versátil con buen rendimiento en todos los verbos de interacción. No tiene ninguna desventaja destacable siempre y cuando el broker se encuentre localizado cercano al cliente para minimizar la latencia de conexión. Los volúmenes de transferencia de datos son mayores que las implementaciones Websockets y CoAP en algunos casos (figuras 5.3 y 5.5). Si esto fuese algo crítico, se podrían configurar niveles menos restrictivos de Quality of Service (QoS) para disminuir el uso de datos.

CoAP

CoAP produce los volúmenes de datos más bajos del experimento para la mayoría de los casos en condiciones poco exigentes. Sin embargo, los volúmenes de transferencia en CoAP tienden a crecer notablemente en condiciones exigentes por culpa de la pérdida de datagramas UDP, que a su vez resulta en

retransmisiones de mensajes y duplicados. Por ejemplo, esto puede verse en el alto volumen de datos de la configuración *SBC* en la figura 5.5 para tasas ≥ 50, donde los tasas de error son $\geq 0{,}39$ (tabla 5.4). Además, en comparación con otros Protocol Binding, en CoAP se puede observar una latencia menor en los verbos *property read* y *action invocation* en comunicaciones a través de Internet (configuración *cloud*).

En la configuración *proxy-mqtt* se puede observar una alta variabilidad de latencia de eventos para CoAP en función de la tasa de eventos objetivo. En la figura 5.6 se muestra que esta variabilidad es de varios órdenes de magnitud. La explicación es que el Protocol Binding CoAP no puede mantener el ritmo con el superior rendimiento del Protocol Binding MQTT, es decir, en MQTT se pueden emitir eventos a una tasa superior que en CoAP. El servidor CoAP introduce un nivel notable de latencia, y esto, combinado con el hecho de que los eventos se emiten de manera secuencial, implica un retardo en todos los eventos recibidos por el cliente *proxy* MQTT (que a su vez son vueltos a emitir por el servidor *proxy* CoAP). El efecto de este retardo se acumula para las tasas de evento más altas y origina las latencias observadas.

Al igual que ocurre en el Protocol Binding HTTP, se pueden observar problemas de rendimiento en CoAP para altas tasas de interacciones *property read* (tabla 5.4). Los logs indican que CoAP tiene problemas a la hora de gestionar retransmisiones en condiciones de alta carga. Es importante destacar que CoAP presenta un límite superior de tasa de mensajes que depende de la configuración del servidor [74]. Cuando el límite se sobrepasa, los mensajes empiezan a ser incorrectamente interpretados como duplicados por culpa de la reutilización de *Message ID*.

Los resultados indican que el Protocol Binding CoAP es adecuado cuando es prioritario ahorrar en transferencia de datos, siempre y cuando sea en condiciones poco exigentes (figuras 5.7 y 5.5). Esto es esperable si se tiene en cuenta que CoAP se transporta sobre UDP, en contraste con los otros tres protocolos que se transportan sobre TCP. Los verbos de interacción basados en comunicación iniciada desde el servidor (*property observe* y *event subscribe*) también presentan una latencia razonable, aunque es mayor que en otros protocolos. Por otra parte, los errores en los verbos *action invocation* y *property read/write* son significativos en hardware con recursos limitados en situaciones con altas tasas de petición (tablas 5.3 y 5.4).

5.2. Validación del emulador WoT

5.2.1. Definición del experimento

Esta sección establece las bases del experimento diseñado para validar la contribución del emulador de aplicaciones WoT basadas en arquitectura edge computing. Principalmente se demuestra como el emulador, llamado WoTemu en su implementación experimental, puede utilizarse para analizar el comportamiento de propuestas de arquitecturas de sistemas WoT.

El experimento se denomina **vigilancia inteligente** (*intelligent surveillance*). En este despliegue existen un conjunto de cámaras de vídeo *inteligentes* con recursos computacionales limitados que son capaces ejecutar bloques de pre-procesamiento. Estas cámaras deben monitorizar el entorno en busca de personas, ajustando sus posiciones en función de los resultados de detección. Las capturas y eventos se almacenan centralmente para permitir que los usuarios humanos puedan consultar los históricos.

La implementación completa de este experimento, incluyendo las aplicaciones de los contenedores *node* y las definiciones de las topologías, está contenida en el código fuente del emulador disponible en [77].

La justificación que sostiene que esta propuesta puede actuar como experimento de validación se basa en el hecho de que otras publicaciones en el dominio [47, 93] presentan versiones comparables de la misma aplicación. Además, se puede argumentar que la configuración del experimento es significativamente más compleja que en las versiones de otras referencias, incluyendo lógica de negocio no trivial, que podría de hecho servir para un despliegue real con ajustes menores.

Utilizar un experimento comparable a otras publicaciones permite resaltar las contribuciones y características distintivas del emulador propuesto. Principalmente, que el diseño del emulador permite iterar de manera rápida sobre distintas versiones de la topología. De esta manera se pueden comparar las ventajas de diferentes aproximaciones a través de pasos incrementales o alternativas. Gracias a las capacidades ofrecidas por la orquestación de contenedores, resulta razonablemente sencillo limpiar los recursos asociados al experimento, modificar la base de código y volver a ejecutar la topología. Por otra parte, la utilización de código real facilita la caracterización del rendimiento, sin necesidad de llegar aproximadamente a un conjunto de parámetros para alimentar a modelos de aplicación abstractos en los que suele resultar complejo considerar todos los detalles de una aplicación moderna.

La capacidad de iteración rápida de topologías se aprovecha concretamente en esta sección a través de las definición de dos escenarios diferenciados. Ambos escenarios están diseñados para cubrir las necesidades del caso de uso

de *vigilancia inteligente*:

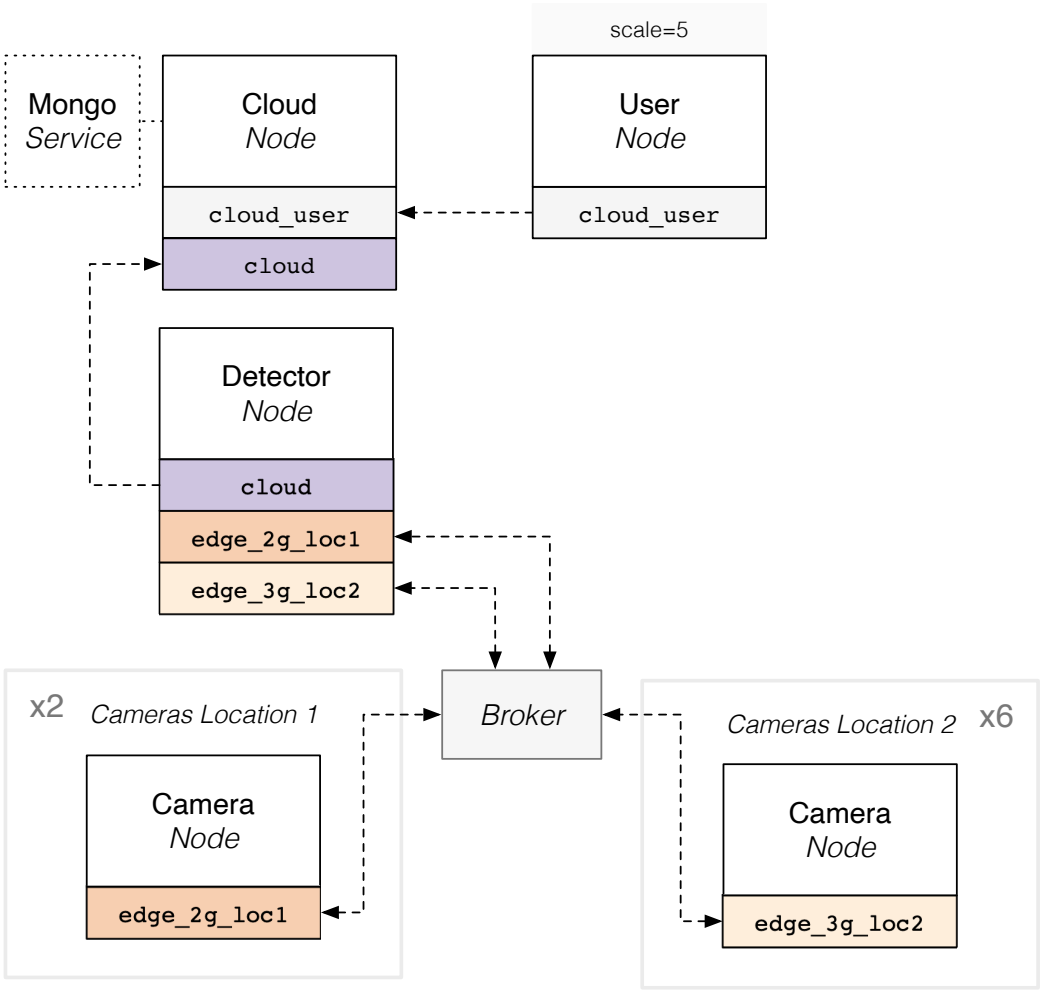


Figura 5.8: Arquitectura escenario *cloud* en el experimento de vigilancia

- Un escenario *cloud* basado en una aproximación de arquitectura plenamente centralizada en la que existe un conjunto de servicios en la nube que son accedidos a través de Internet por todos los dispositivos de capa de sensores. La arquitectura del escenario *cloud* se muestra en la figura 5.8.
- Un escenario *edge* que se presenta como alternativa al escenario anterior. La arquitectura está basada en el modelo edge computing, aprovechando las posibilidades de optimización de latencia y transferencia

de datos a través del despliegue de infraestructura hardware y servicios software en capas intermedias. La arquitectura del escenario *edge* se muestra en la figura 5.9.

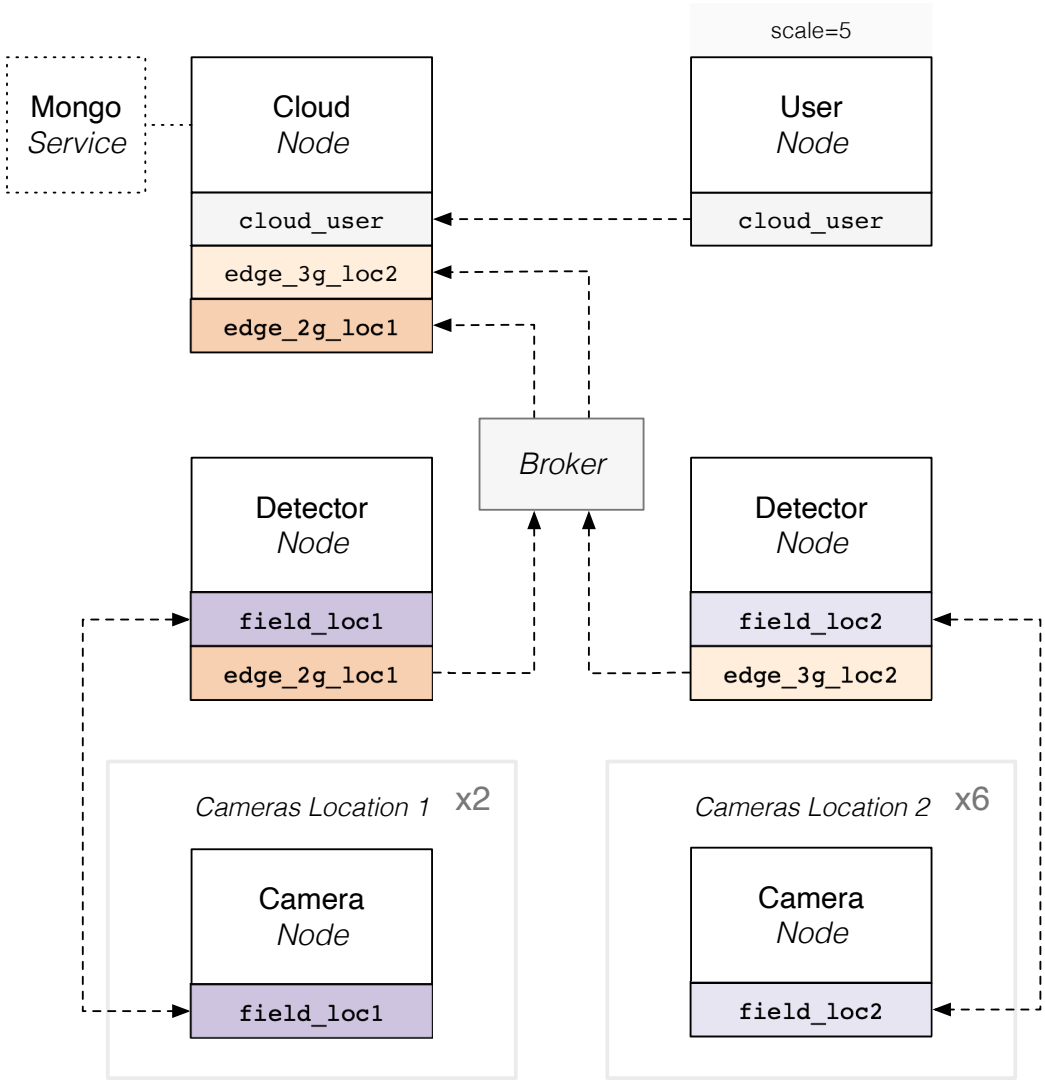


Figura 5.9: Arquitectura escenario *edge* en el experimento de vigilancia

Se pueden identificar **cuatro tipos de aplicaciones** (contenedores de tipo *node* dentro de la terminología del emulador) que son los bloques fundamentales con los que se construyen las topologías de los escenarios *cloud* y *edge*:

Camera Esta aplicación tiene el objetivo de emular una cámara de vídeo con capacidades de procesamiento simples. La implementación se basa en el popular framework de visión por computador OpenCV [94]. La aplicación lee un vídeo local codificado en formato H.264 en un bucle infinito, llevando a cabo un proceso de detección de movimiento en el proceso. Este proceso es razonablemente simple: primero se diferencia un conjunto de frames de vídeo recientes y consecutivos, seguidamente se obtienen las medias de los resultados de diferenciación, finalmente se calcula la media global y se aplica un valor de corte para determinar si hay movimiento. Cuando se detecta movimiento, el frame de vídeo en cuestión se convierte a formato JPG, se codifica en Base64 y se emite en forma del evento WoT `jpgVideoFrame`.

Detector Esta aplicación se suscribe a los eventos `jpgVideoFrame` de un conjunto de *cameras* para detección de caras en los frames de vídeo recibidos. La implementación se basa en el paquete Python face-recognition [95], que a su vez está basado en las primitivas de deep learning expuestas por el *toolkit* de machine learning dlib [96]. Un *detector* expone dos propiedades WoT. La primera propiedad `latestDetections` contiene (en memoria) el frame de vídeo y resultado de detección asociado más recientes de cada *camera*. La segunda propiedad `cameras` contiene la configuración de la aplicación, incluyendo todas las URL de las *camera*. Cada vez que el nodo *detector* encuentra una cara en un frame de vídeo, se invoca la acción de ajuste PTZ `controlPTZ` del nodo *camera* correspondiente. La aplicación *detector* está diseñada cuidadosamente para optimizar el uso de recursos computacionales. Los frames de vídeo se almacenan en una cola en memoria y se procesan de manera asíncrona: el orden de procesamiento de los frames lo determina su prioridad, que es una función de la antiguedad del frame en segundos y el número de frames que han sido procesados previamente con éxito para la *camera* correspondiente. Es decir, se da prioridad a los frames más recientes de los nodos *camera* que se han visto privados previamente de recursos.

Cloud La aplicación *cloud* es una instancia de la aplicación *built-in* denominada *historian*. Utiliza un servicio MongoDB para actuar como un almacén persistente de datos y agregador, exponiendo una API HTTP para clientes externos que necesitan leer los datos de detección desde diferentes localizaciones. De manera general, un *historian* lee y almacena de manera periódica las propiedades de Things arbitrarias definidas como argumento de configuración.

User La aplicación *user* es una instancia de la aplicación *built-in* denomi-

nada *caller*, que invoca de manera continua un conjunto arbitrario de acciones de una Thing. Estos nodos representan los clientes dentro de la topología, y para ello consultan con regularidad los frames de vídeo y resultados de detección de caras. El nodo *cloud* actúa a modo de punto de entrada para los clientes, exponiendo los datos de manera agregada a través de su API HTTP. Las invocaciones siguen un proceso de Poisson para intentar modelar el comportamiento de un grupo de usuarios reales. Aunque la tasa de petición λ (1/s) es un argumento, existen límites superiores efectivos de este valor que vienen impuestos por el rendimiento de un núcleo de la CPU. Por esta razón, y para generar un volumen signicativo de peticiones, se crean 5 réplicas del nodo *user* donde el valor de λ es de 5 peticiones por segundo, lo que genera aproximadamente 25 peticiones por segundo de manera global.

Aplicación	Límite de memoria	Puntuación CPU
Camera	256 MB	200
Detector (solo en escenario *edge*)	1 GB	600

Cuadro 5.6: Límites de recursos computacionales en nodos del experimento

De manera general, el hardware presente en la capa de sensores tiende a tener más limitaciones de recursos computacionales que los dispositivos en capas intermedias edge, que a su vez tienden a ser menos capaces que la infraestructura hardware en las capas de nube. Esto suele ser la consecuencia de varios factores, por ejemplo, las restricciones de alimentación y los altos números de dispositivos típicamente presentes en la capa de sensores. Para poder incorporar estas diferencias en las topologías, la tabla 5.6 muestra los límites de CPU y memoria que se imponen en los nodos de la topología para los dos escenarios. El valor 600 de puntuación de CPU representa los resultados obtenidos con Sysbench en un procesador ARM de la popular *Single-Board Computer* Raspberry Pi Model 3B, que es uno de elementos más populares que se suele encontrar en la capa *edge*. El valor de puntuación de los nodos *camera* está definido a 200 para emular unas restricciones significativamente más severas y es el mismo para los dos escenarios. A modo de comparación, se destaca que la puntuación de un hilo de CPU en un procesador *mobile* Intel Core i7 de 4ª generación es de aproximadamente 1000.

Como se describe en la sección 4.1.2, la integración de características realistas de red es una necesidad prioritaria para garantizar la utilidad de los resultados de emulación. Los detalles de configuración de las redes presentes en las topologías (figuras 5.8 y 5.9) se describen en la tabla 5.7. Las redes

Red	Perfil de red	Latencia	Jitter	Ancho de banda
`edge_2g_loc1`	`GPRS`	700 ms	100 ms	50 kbit
`edge_3g_loc2`	`REGULAR_3G`	300 ms	150 ms	1500 kbit
`field_loc1`	`WIFI`	25 ms	5 ms	50 mbit
`field_loc2`	`WIFI`	25 ms	5 ms	50 mbit
`cloud_user`	`CABLE`	5 ms	5 ms	100 mbit
`cloud`	`CABLE`	5 ms	5 ms	100 mbit

Cuadro 5.7: Características de las redes del experimento

`edge_2g_loc1` y `edge_3g_loc2` representan una conexión de red móvil desde el despliegue en campo a Internet, que suele ser una ocurrencia común en despliegues IoT. Por otro lado, las redes `cloud` y `cloud_user` no tienen límites efectivos, ya que representan respectivamente la infraestructura de red que interconecta instancias dentro de la nube y la conexión de usuarios con los servidores en la nube (que se supone de fibra o similar).

5.2.2. Métricas de rendimiento

Las métricas de rendimiento representadas por los resultados mostrados en la siguiente sección 5.2.3 son mayoritariamente las que proporciona por defecto el módulo de informes del emulador. La sección relativa al módulo de informes 4.1.4 contiene detalles al respecto de estas métricas por defecto. Adicionalmente, se han definido dos *métricas de aplicación* ad-hoc que son necesarias para caracterizar el experimento de *vigilancia inteligente*.

La primera métrica de aplicación es la **latencia de detección** (*detection latency*), que representa el retardo desde que se captura un frame de vídeo hasta que se termina su procesamiento. Esta métrica corresponde a los nodos *detector* y está dividida a su vez en dos componentes para mayor claridad en el análisis. Todos los elementos T a continuación hacen referencia a timestamps UNIX con precisión en milisegundos:

El componente de **latencia de encolado**, descrito como *latency from camera capture to arrival* en las figuras, se define como:

$$L_{queue} = T_{queue} - T_{capture} \tag{5.7}$$

El componente de **latencia de procesamiento**, descrito como *latency from arrival to detection* en las figuras, se define como:

$$L_{process} = T_{process} - T_{queue} \tag{5.8}$$

$T_{capture}$ Timestamp en el que se captura un frame de vídeo en un nodo *camera*.

T_{queue} Timestamp en el que el frame de vídeo entra en la cola de procesamiento del nodo *detector*.

$T_{process}$ Timestamp en el que el proceso de detección de caras termina de ejecutarse sobre el frame de vídeo.

La segunda métrica de aplicación es la **latencia PTZ** (*PTZ latency*). PTZ es un acrónimo que significa *Pan-Tilt-Zoom* y hace referencia a las propiedades de rotación y zoom de las cámaras. Esta métrica corresponde a los nodos *camera* y representa el retardo en la actuación de la cámara para ajustarse a los eventos detectados en el entorno:

$$L_{PTZ} = T_{PTZ} - T_{capture} \tag{5.9}$$

$T_{capture}$ Timestamp en el que se captura un frame de vídeo en un nodo *camera* (igual que en el caso de la latencia de detección).

T_{PTZ} Timestamp en el que se recibe la invocación de ajuste PTZ para el frame de vídeo en cuestión.

No todos los frames de vídeo resultan en un ajuste PTZ, concretamente, se descartan todos los frames en los que no se detectan caras y los que son excesivamente antiguos. Además, el ajuste PTZ está protegido por un *lock* de exclusión mutua para evitar conflictos en las peticiones de ajuste, lo que supone otro punto de corte que puede descartar frames. Por otra parte, es interesante resaltar que la métrica de latencia de detección está plenamente contenida dentro de esta métrica de latencia PTZ.

5.2.3. Resultados y discusión

En esta sección se presentan las figuras que representan los resultados del experimento de validación del emulador WoT. Para ello, primero se comentan los resultados del escenario *cloud*, para después comentar los resultados del escenario *edge* como contraste de ambas aproximaciones arquitecturales. La tabla 5.8 contiene una referencia de todas las figuras de resultados para ambos escenarios, y puede ser de utilidad a modo de índice para comparar métricas en ambos casos.

El conjunto de datos completo (con un tamaño de 6.3 GB) que representa los resultados del experimento de vigilancia se encuentra disponible

Resultado del experimento	Escenario *cloud*	Escenario *edge*
Volumen de datos de servicios	Figura 5.10	Figura 5.18
Vista general de recursos computacionales	Figura 5.11	Figura 5.19
Latencia PTZ	Figura 5.12	Figura 5.20
Latencia de detección	Figura 5.13	Figura 5.21
Detalle de nodo *camera*	Figura 5.15	Figura 5.22
Detalle de nodo *detector*	Figura 5.16	Figura 5.23
Detalle de nodo *cloud*	Figura 5.17	Figura 5.24

Cuadro 5.8: Resumen de resultados de validación del emulador

públicamente en Zenodo [91]. Estos resultados han obtenido de la ejecución de los experimentos en un clúster *swarm* de dos nodos con las siguientes características:

- Intel Core i7-6770HQ CPU @ 2.60GHz (4 cores, 8 threads).
- 32GB DDR4 2133 MHz (2x16GB en configuración *dual channel*).
- 480GB SATA3 SSD.

En los gráficos que aparecen a continuación se puede observar que los nombres de los contenedores (*swarm tasks*) muestran un sufijo alfanumérico arbitrario separado por puntos. Este sufijo es parte de un identificador que es asignado por Docker swarm. La primera parte es el índice de la réplica, mientras que la segunda parte es un identificador único. Por ejemplo, en la figura 5.10 aparece una *swarm task* llamada `cloud.1.601`, la cual hace referencia al primer contenedor (réplica) del servicio `cloud` con un identificador único que empieza por `601`; el identificador único se acorta en las figuras por motivos de espacio.

Escenario *cloud*

En la figura 5.10 se puede ver el mapa de calor del tráfico de los servicios. Por claridad, solo se muestran los servicios con un volumen de datos significativo. La mayoría del tráfico de salida (*outbound*) se genera en los nodos *user* que envían peticiones de lectura al nodo *cloud* de manera recurrente y en el nodo *detector* que transmite de manera continua los datos agregados de las cámaras al nodo *cloud*. En cuanto al tráfico de entrada (*inbound*), el nodo *detector* vuelve a mostrar los resultados del flujo combinado de todos los nodos *camera*, que pasa al nodo *cloud* para salir hacia los clientes *user* de la topología.

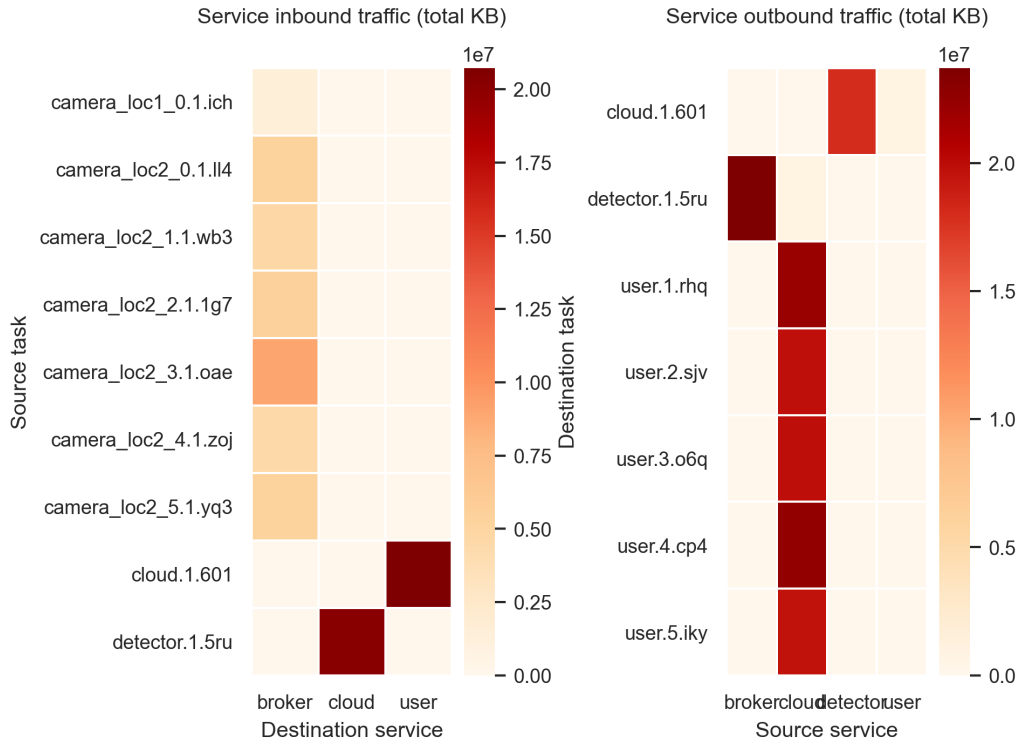


Figura 5.10: Volumen de transferencia de datos (*cloud*)

La figura 5.11 muestra las distribuciones de utilización de CPU y memoria de los contenedores a lo largo de la vida del experimento. Las distribuciones se dividen en dos categorías: los gráficos *constrained* muestran los contenedores que tienen límites computacionales impuestos, mientras que los gráficos *unconstrained* muestran los contenedores que tienen acceso a todos los recursos de la máquina (4 cores, 32 GB). Es necesario disponer de detalles acerca del hardware sobre el que se ejecuta el experimento para dar contexto a los datos de los servicios que están marcados como *unconstrained*. Por otro lado, los datos de servicios *constrained* proporcionan información valiosa acerca de la adecuación de los recursos computacionales asignados, independientemente de la infraestructura hardware que está por debajo. De esta figura se extrae que las cámaras operan cerca de los límites superiores de recursos computacionales descritos en la tabla 5.6, lo cual puede intepretarse como que las cámaras operan dentro de límites estrechos y que podrían beneficiarse de un incremento de recursos para mayor estabilidad.

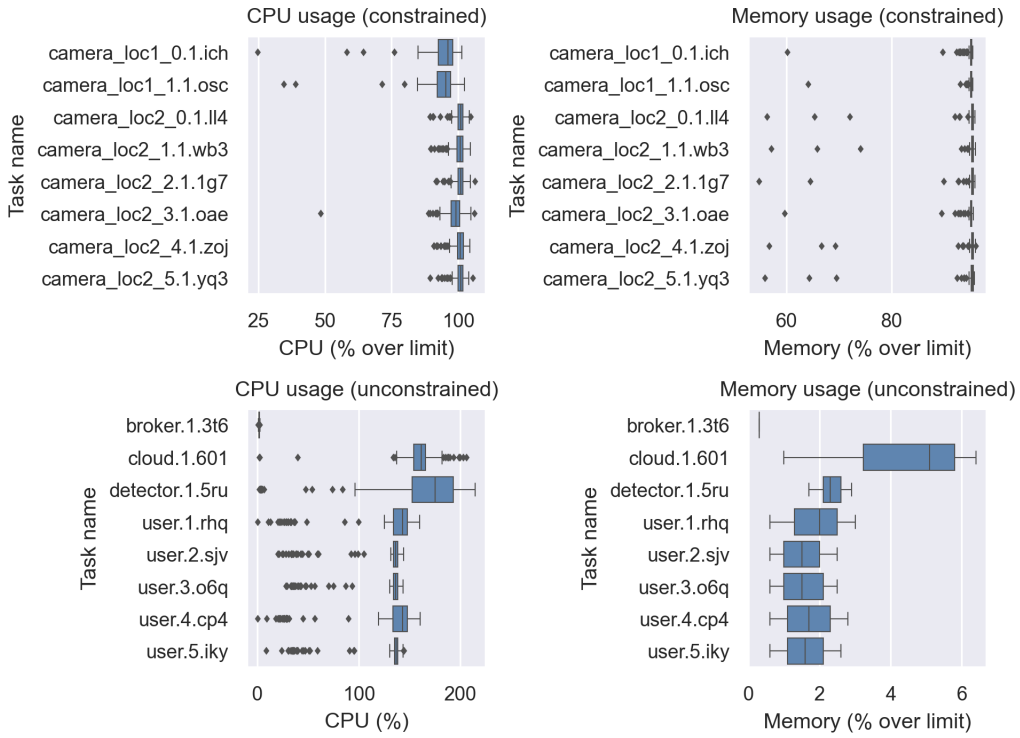


Figura 5.11: Vista general de recursos computacionales (*cloud*)

Las métricas de latencia PTZ y latencia de detección del escenario *cloud* se pueden ver en las figuras 5.12 y 5.13 respectivamente. La distribución de la latencia PTZ se encuentra en el orden de cientos de segundos, lo cual es totalmente inaceptable. Esto se debe principal a la congestión de red causada por todos los nodos *camera* accediendo a la red de manera concurrente para publicar los eventos de frame de vídeo `jpgVideoFrame`. Las redes móviles (véase tabla 5.7) presentan un bajo ancho de banda y una alta latencia y jitter, es decir, una calidad de conexión bastante baja que deriva en un número excesivo de retransmisiones TCP; esto es especialmente notable en la red `edge_2g_loc1`.

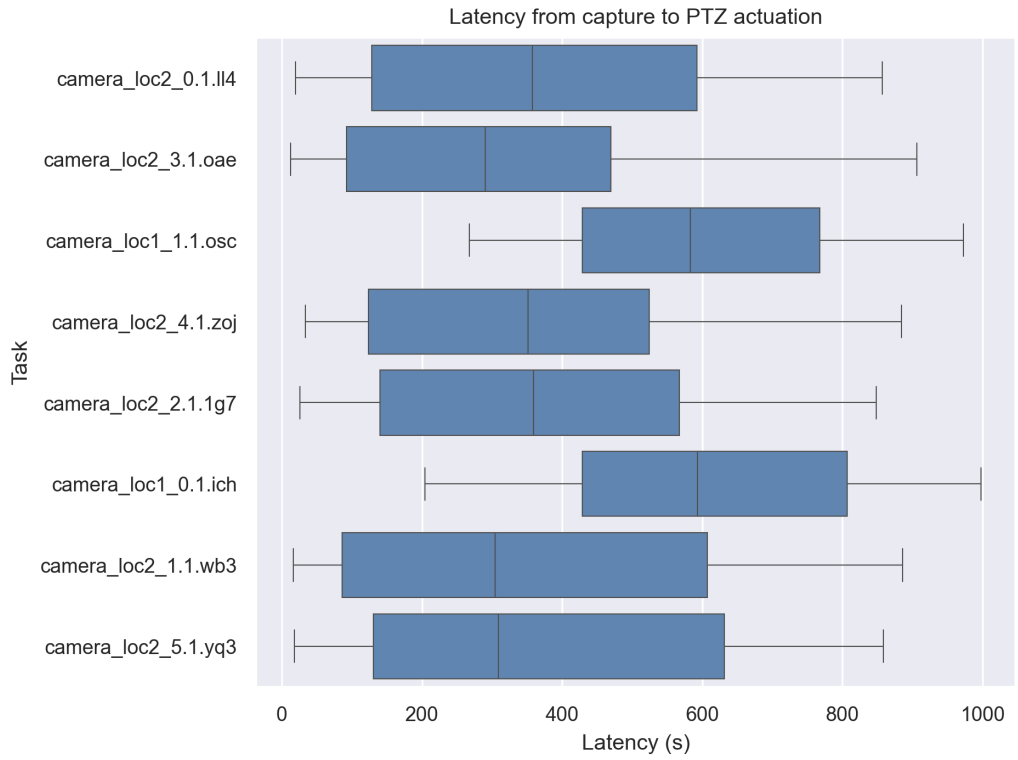


Figura 5.12: Latencia PTZ (*cloud*)

La figura 5.14 muestra la evolución temporal de la media de la *latencia de encolado* individualizada para cada nodo *camera*. Esto es complementario a la figura 5.13, que muestra la distribución agregada para todas las cámaras en formato boxplot. La latencia de los frames de vídeo crece de manera descontrolada hasta el punto en el que los nodos *detector* empiezan a rechazar los frames debido al excesivo retardo. Este efecto es mucho más pronunciado en el primer conjunto de nodos *camera* que están conectados a una red con perfil 2G.

Los eventos de frame de vídeo `jpgVideoFrame` se transportan sobre la implementación de Protocol Binding de MQTT. En MQTT, los mensajes se almacenan en colas internas en el broker y se entregan de manera secuencial al nodo *detector*: este comportamiento también tiene también una influencia negativa en la latencia porque los retardos se acumulan en dichas colas.

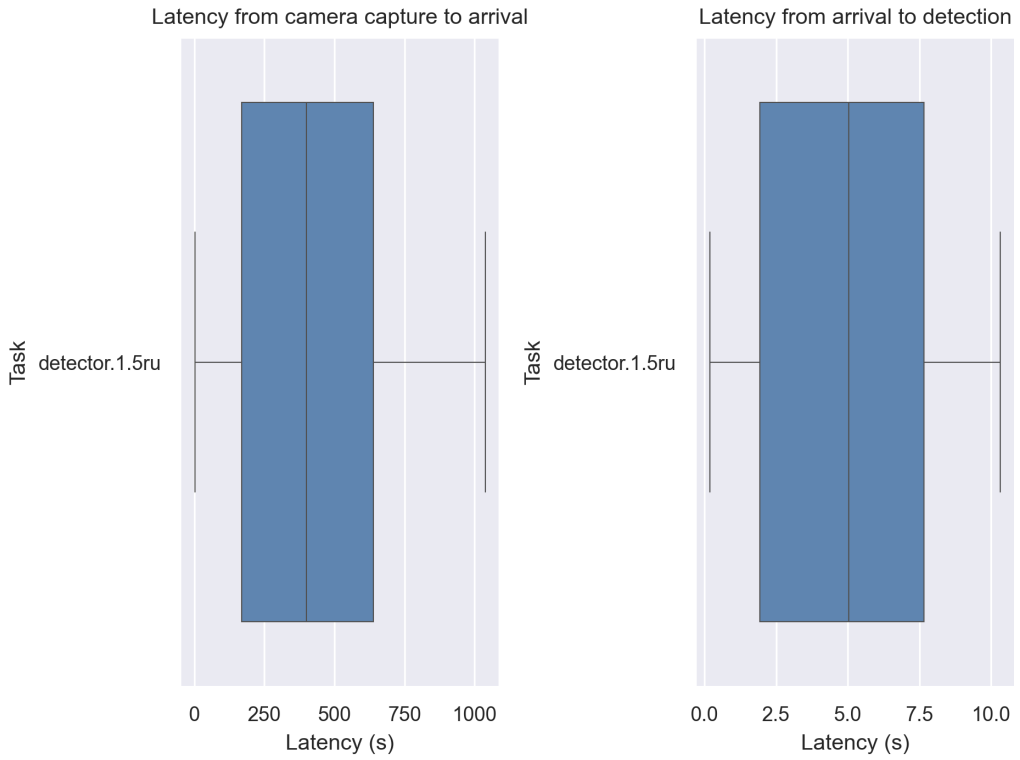


Figura 5.13: Latencia de detección (*cloud*)

Aunque la figura 5.11 proporciona una vista agregada de la distribución del uso de recursos computacionales, también resulta interesante tener una fotografía de bajo nivel para cada contenedor individual. Estos detalles se presentan en las figuras 5.15, 5.16 y 5.17.

De manera particular, la figura 5.15 corresponde a una muestra representativa de un nodo *camera* (las figuras equivalentes para otros nodos *camera* son similares). Se puede confirmar lo que se comentaba anteriormente: la cámara opera muy cerca de los límites de CPU y memoria a lo largo de toda la vida del experimento. Además, todo el tráfico generado corresponde al protocolo MQTT a una tasa aproximada de 1-2 KBps. Es importante destacar que puesto que MQTT se transporta sobre TCP, un subconjunto de los paquetes capturados por TShark (la herramienta de monitorización de red) se categorizan simplemente en capa 4 como TCP en vez de MQTT, aunque pertenezcan al Protocol Binding MQTT.

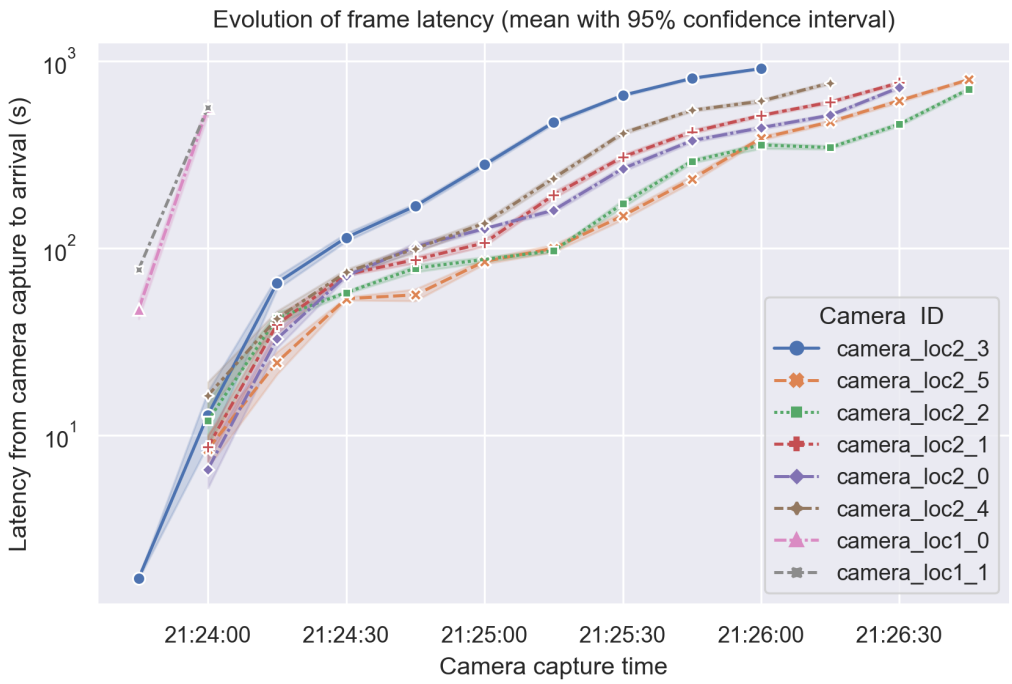


Figura 5.14: Evolución temporal de latencia de frames (*cloud*)

El detalle de los nodos *detector* (figura 5.16) y *cloud* (figura 5.17) muestra tasas de red aproximadas cercanas a los 10 KBps en ambos casos. La mayoría del tráfico es TCP porque corresponde a la transmisión de imágenes JPG codificadas en Base64 generadas por las interacciones `jpgVideoFrame`. El uso de memoria crece de manera continuada a lo largo del experimento, probablemente debido a que el stack nunca llega a un estado de *convergencia*, es decir, los eventos de imagen se acumulan en cuellos de botella y no son procesados ni descartados a una tasa suficiente.

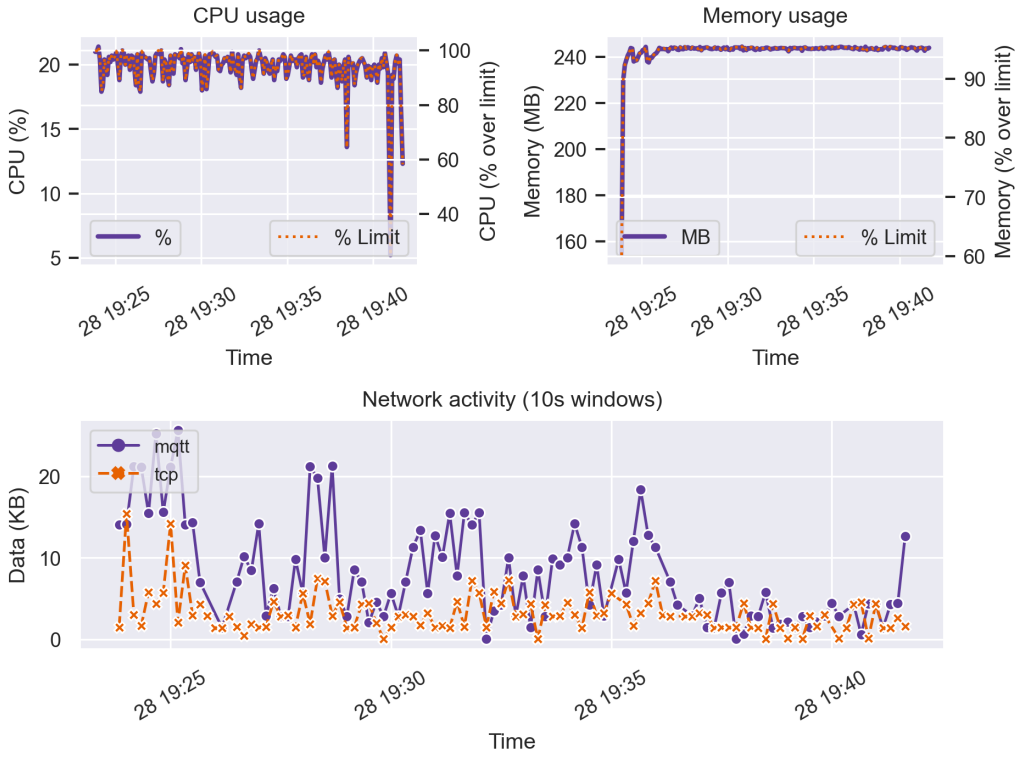


Figura 5.15: Evolución de recursos de un nodo *camera* (*cloud*)

Figura 5.16: Recursos computacionales del nodo *detector* (*cloud*)

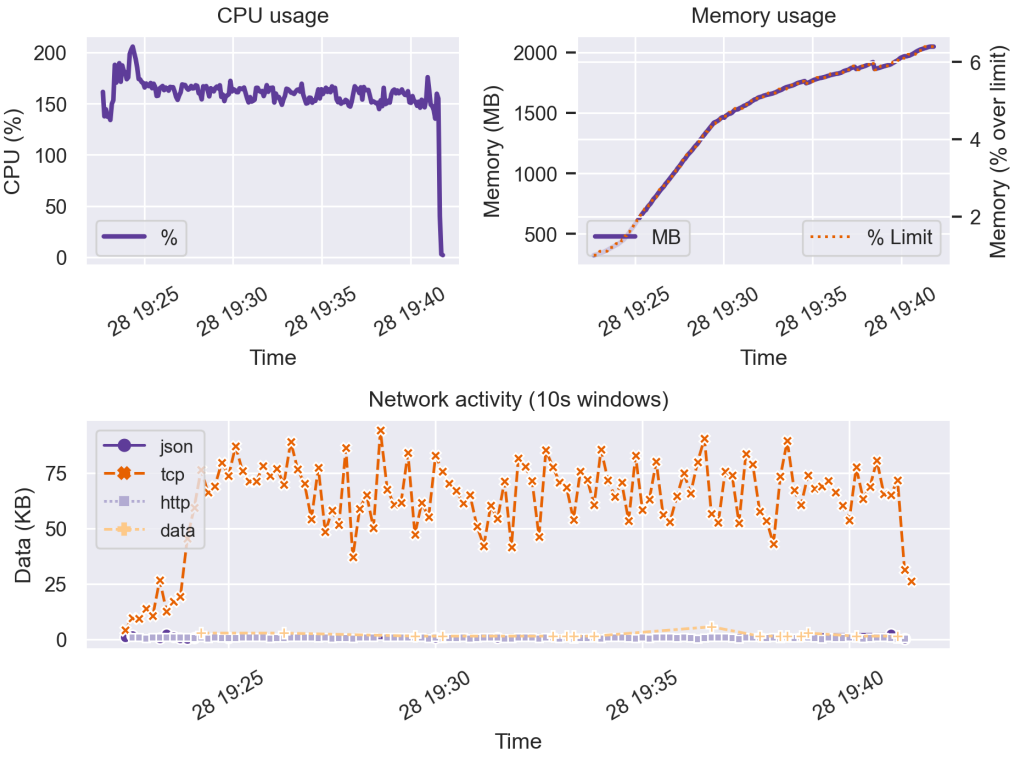


Figura 5.17: Recursos computacionales del nodo *cloud* (*cloud*)

Escenario *edge*

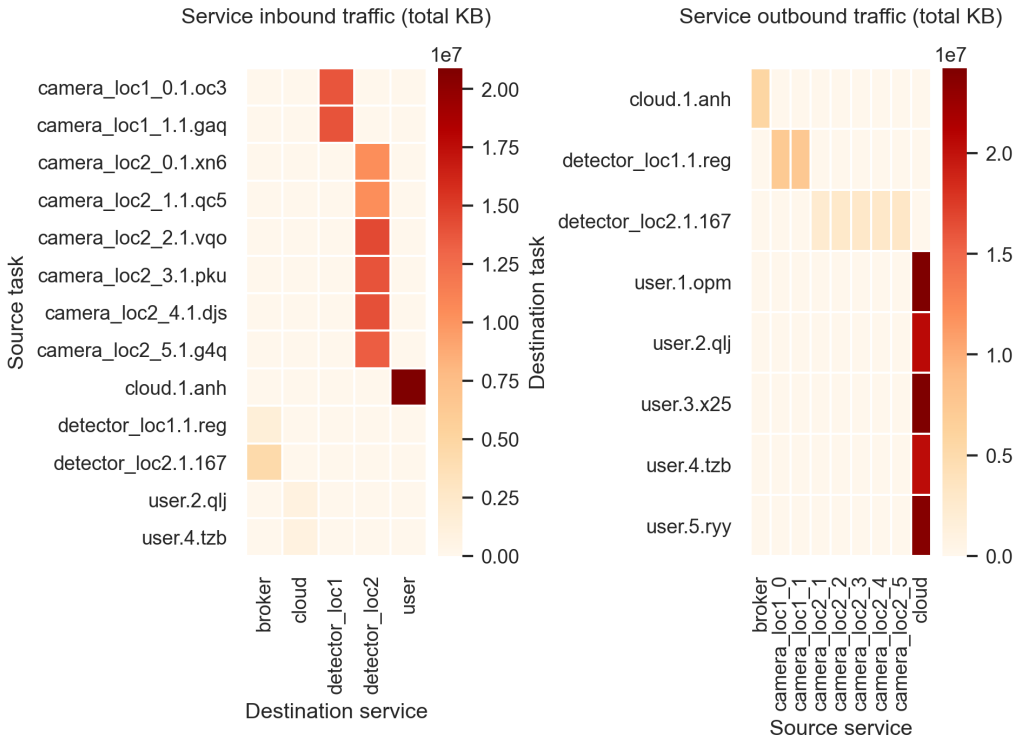


Figura 5.18: Volumen de transferencia de datos (*edge*)

Los resultados mostrados en la sección anterior demuestran que una aproximación plenamente centralizada no resulta viable en este caso. La adopción del patrón edge computing puede suponer una vía de mejora de los resultados para llegar hasta niveles aceptables. De manera específica, la topología del escenario *edge* (figura 5.9) presenta las siguientes diferencias destacables con la topología del escenario *cloud* (figura 5.8):

- Cada conjunto de nodos *camera* dispone de un nodo *detector* localizado en su despliegue local para optimizar latencia y volumen de transferencia de datos. Es decir, los nodos *camera* no están directamente conectados a la WAN y residen en la misma LAN que su *detector*. En el caso del escenario *cloud* existía un solo *detector* centralizado y todos los nodos *camera* se comunicaban con él a través de las redes `edge` que representan una conexión a Internet.

- El nodo *cloud* lee dos nodos *detector* a través de un broker MQTT sobre conexiones de red móvil. En el caso *cloud* la conexión entre *cloud* y *detector* se representaba con las características de una conexión por cable.

- Mientras que en el escenario *cloud* los nodos *detector* no tenían límites de recursos, si que se definen límites en el escenario *edge* (véase tabla 5.6). Esto es así para representar que los nodos *detector* se implementan sobre una plataforma *Single-Board Computer*.

- Las redes `field_loc1` y `field_loc2` emulan una red inalámbrica local entre el nodo *detector* y los conjuntos de nodos *camera*, y solo están presentes en el escenario *edge*.

El mapa de calor de tráfico (figura 5.18) una vez más muestra que en sentido *outbound* la mayoría del tráfico ocurre desde el nodo *cloud* a los nodos cliente *user*. Esto es así porque las respuestas del nodo *cloud* contienen imágenes de las cámaras en formato JPG codificadas en Base64. En sentido *inbound*, la presencia de dos nodos *detector* independientes permite balancear el tráfico de las cámaras, sin sobrecargar de manera excesiva la conexión móvil y el broker MQTT como ocurría en el escenario *cloud*.

En la figura 5.19, que muestra la distribución de uso de recursos computacionales, los nodos *detector* se encuentran al límite del uso de CPU: la distribución muestra que el uso está cerca del 100 % prácticamente durante todo el experimento. La memoria sin embargo presenta una dependencia clara en el número de cámaras cliente. El *detector* de la primera localización solo está conectado a dos nodos *camera*, así que la memoria se situa alrededor del 50 %. El *detector* de la segunda localización, que gestiona seis nodos *camera*, se acerca notablemente a los límites superiores.

La aproximación edge computing resulta adecuada para el caso de uso de *vigilancia inteligente*, como se demuestra a través de las métricas de latencia PTZ (figura 5.20) y latencia de detección (figura 5.21). La mediana de latencia PTZ es inferior a 8 segundos para todos los nodos *camera*, lo que resulta un nivel de rendimiento razonable si se tienen en cuenta los limitados recursos computacionales asignados a los nodos *camera* y *detector*. La instalación de dos *detector* independientes en el *edge* de los despliegues de cámaras permite alcanzar medianas de menos de 1 segundo en la métrica de *latencia de encolado* en ambos casos. Por otro lado, la *latencia de procesamiento* no tiene diferencias notables con respecto al escenario *cloud* porque es una métrica local a los nodos *detector*.

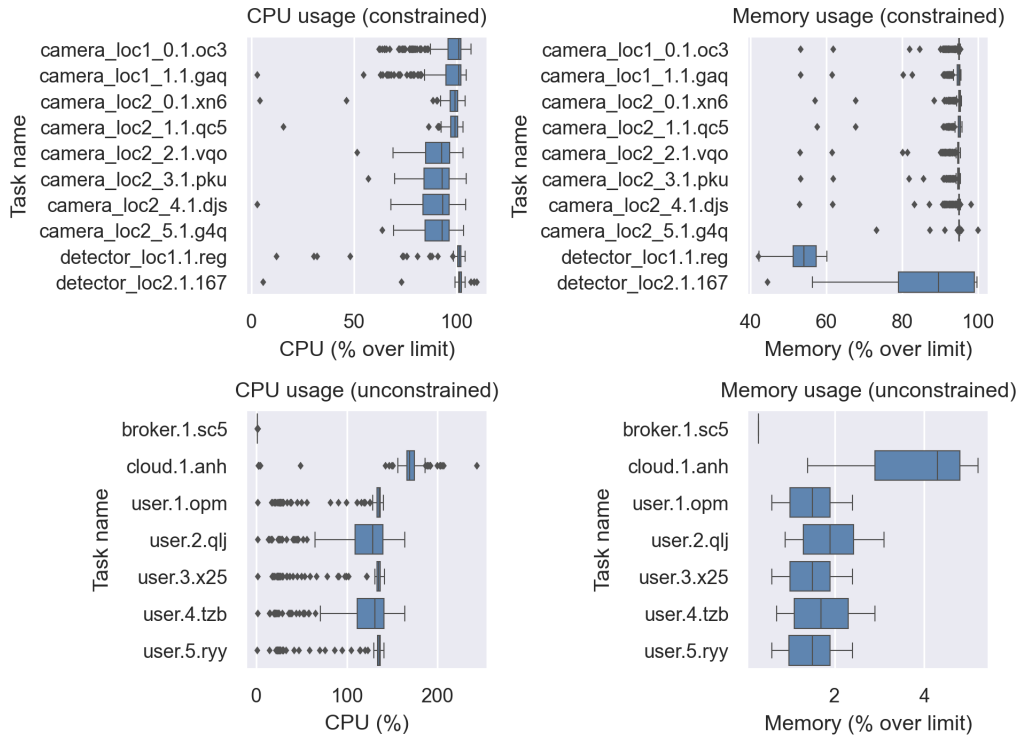


Figura 5.19: Vista general de recursos computacionales (*edge*)

De manera similar a lo que ocurre en el escenario *cloud*, la figura 5.22 proporciona más visibilidad al hecho de que las limitaciones de memoria impuestas en los nodos *camera* son estrictas aunque viables. Respetar estas limitaciones es necesario para mantener los costes dentro de un límite razonable. Es decir, puesto que se necesita un elevado número de cámaras en la capa de sensores, el coste de una cámara individual tiene un impacto notable en el coste global del sistema. En contraste con el escenario *cloud*, se puede observar que el tráfico procede de la implementación de Protocol Binding HTTP en vez de MQTT. Como detalle, se destaca que hay un conjunto de paquetes clasificados como *json* porque ese es el formato de serialización del cuerpo de las peticiones HTTP. Además, el volumen de transferencia de datos es superior al escenario *cloud* porque HTTP es un protocolo con una huella de tráfico significativamente mayor a MQTT.

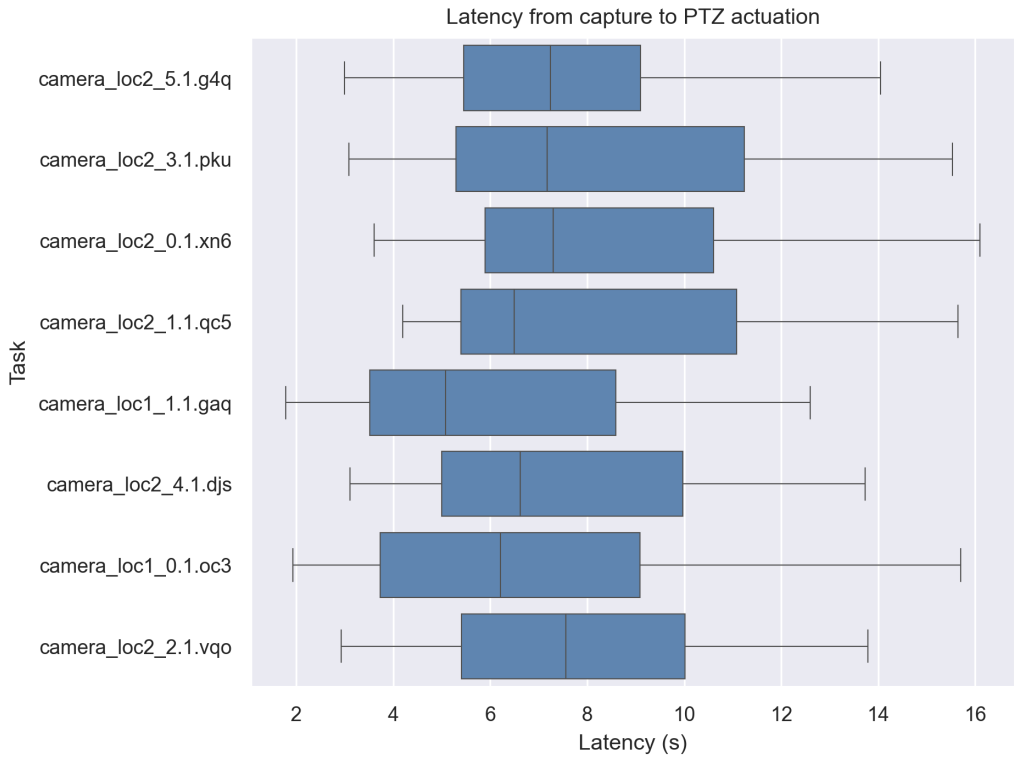


Figura 5.20: Latencia PTZ (*edge*)

El cuello de botella de los nodos *detector* está en el procesador, como se puede ver en la figura 5.23 (aunque la figura 5.23 se refiere al *detector* de la localización 2, la gráfica de CPU del *detector* en localización 1 es muy similar). Una inspección de los logs de este nodo *detector* revela que un porcentaje significativo de todos los frames de vídeo que llegan a él se descartan porque el buffer interno está lleno, es decir, el *detector* no es capaz de mantener la frecuencia de llegada de los frames de vídeo. Esto indica que los nodos *detector* se beneficiarían notablemente de una CPU con más capacidad. Aun así, esto no es un problema crítico porque la aplicación *detector* es capaz de operar en estas condiciones: los frames contenidos en el buffer que son excesivamente antiguos se descartan para permitir el procesamiento de eventos recientes, además se utiliza la lógica de prioridad descrita en la sección 5.2.1.

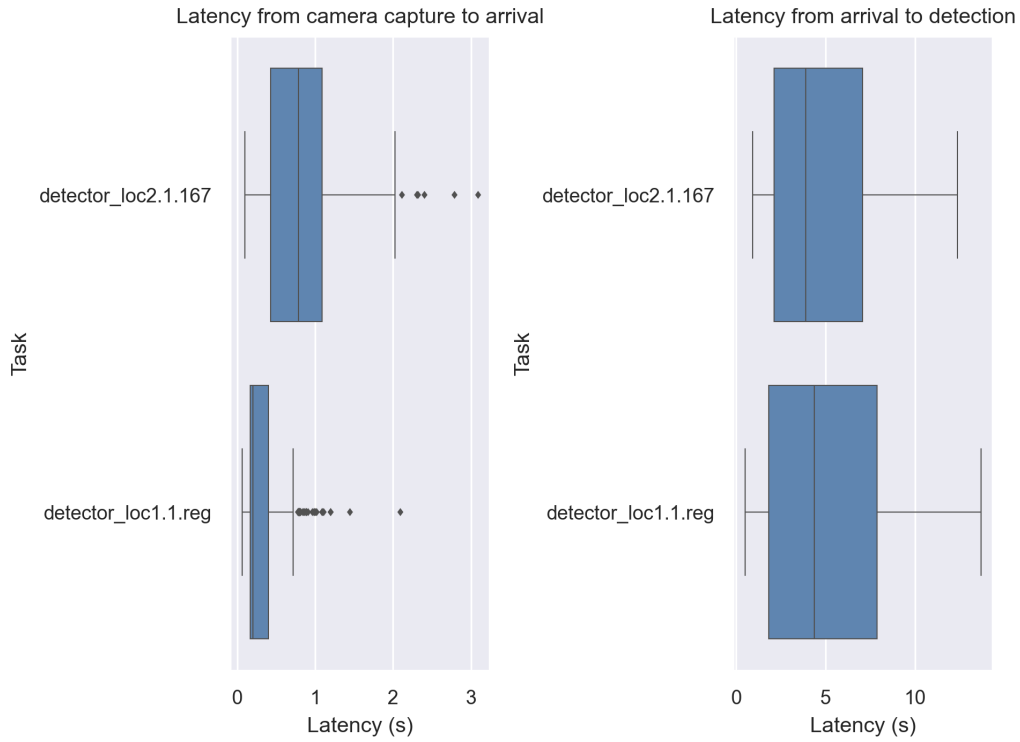


Figura 5.21: Latencia de detección (*edge*)

El comportamiento del nodo *cloud* mostrado en la figura 5.24 es efectivamente equivalente al caso del escenario *cloud* (figura 5.17). Aunque en el escenario *edge* el nodo *cloud* se conecta directamente al broker MQTT para interactuar con los nodos *detector*, esto no tiene un impacto notable en el volumen de tráfico de red. La mayor carga en el nodo *cloud* procede de la presión ejercida en la CPU por el volumen de interacciones HTTP originadas en los nodos *user*.

La figura 5.25 contiene más información acerca del comportamiento observado dentro del nodo *cloud*. En esta gráfica se puede ver que la lectura de las propiedades `cameras` y `latestDetections` de los nodos *detector* presenta una latencia en el orden de pocas decenas de segundos. Esta latencia es alta, aunque razonable si se tienen en cuenta los límites de recursos computacionales y de red impuestos en las capas bajas, especialmente los límites severos derivados de la conexión 2G. En otras palabras, la latencia de 20-30 segundos en la disponibilidad de frames de vídeo procesados es un compromiso viable frente al coste de inversión de recursos.

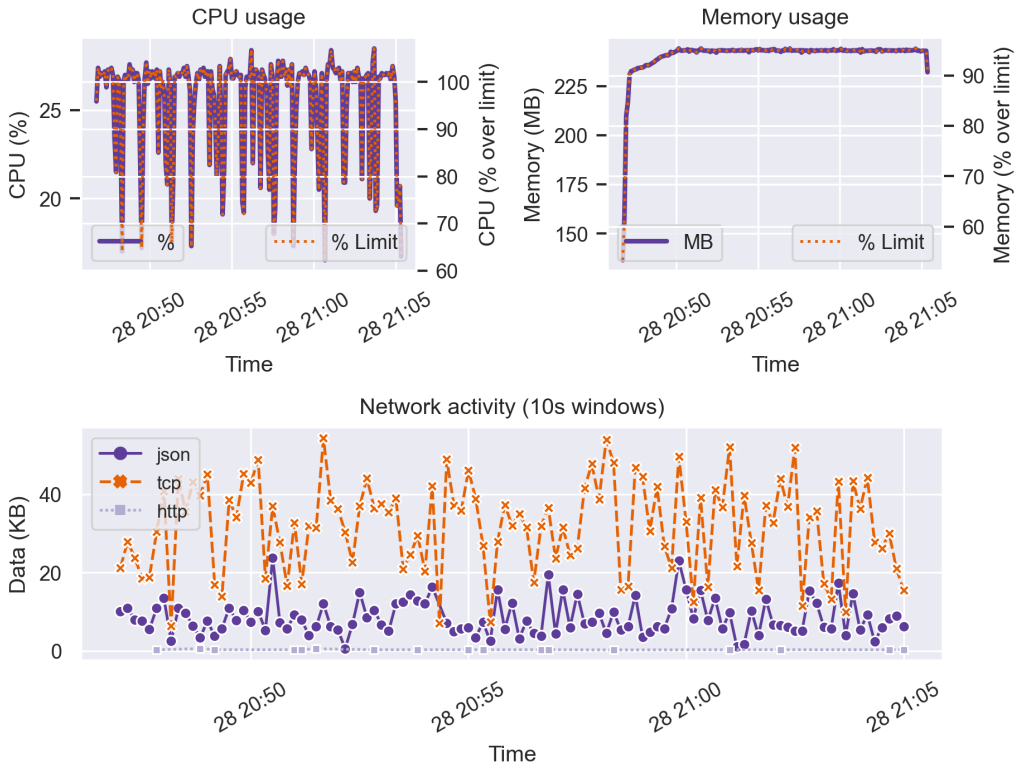


Figura 5.22: Recursos computacionales en un nodo *camera* (*edge*)

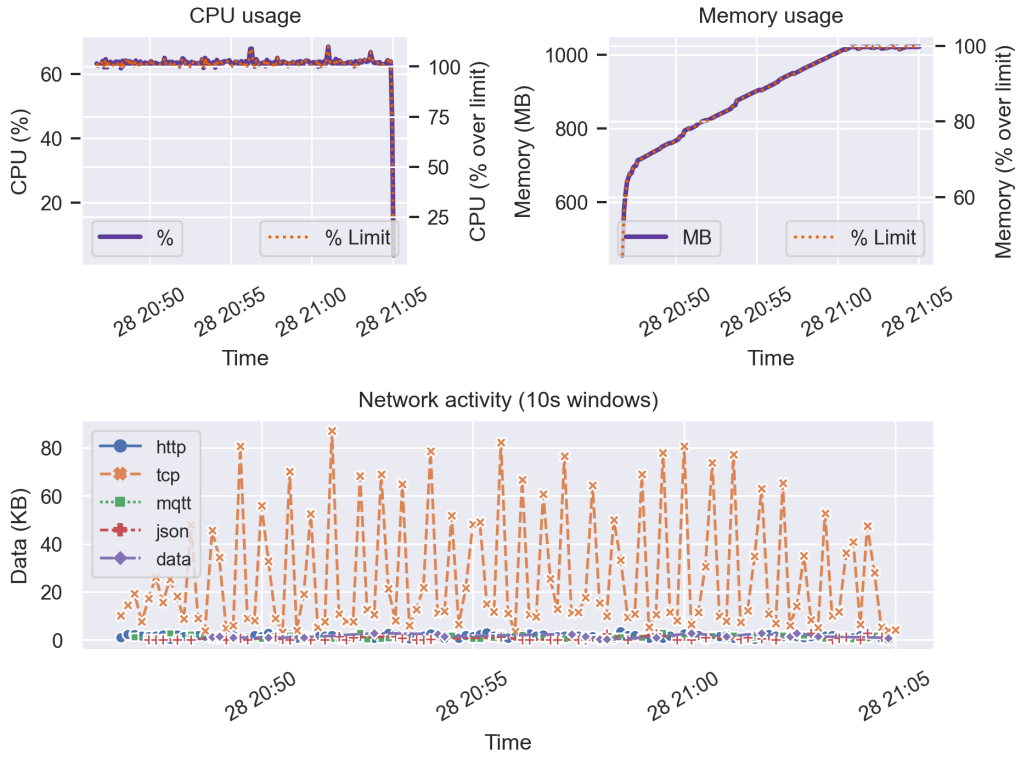


Figura 5.23: Recursos computacionales del nodo *detector* en localización 2 (*edge*)

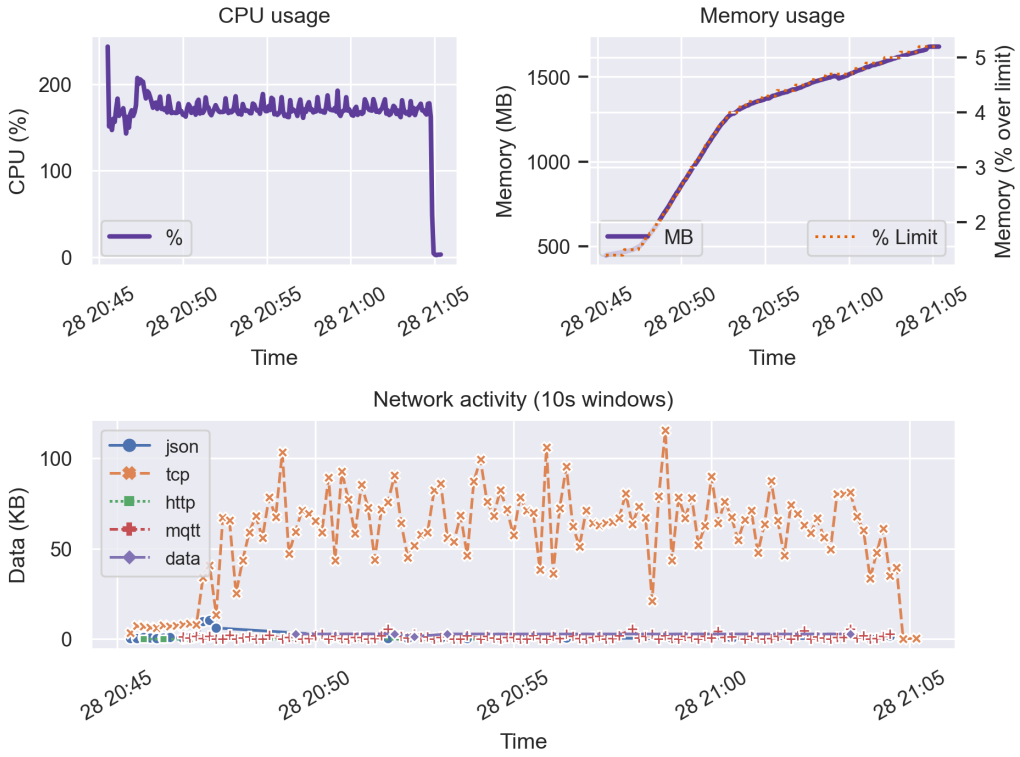


Figura 5.24: Recursos computacionales del nodo *cloud* (*edge*)

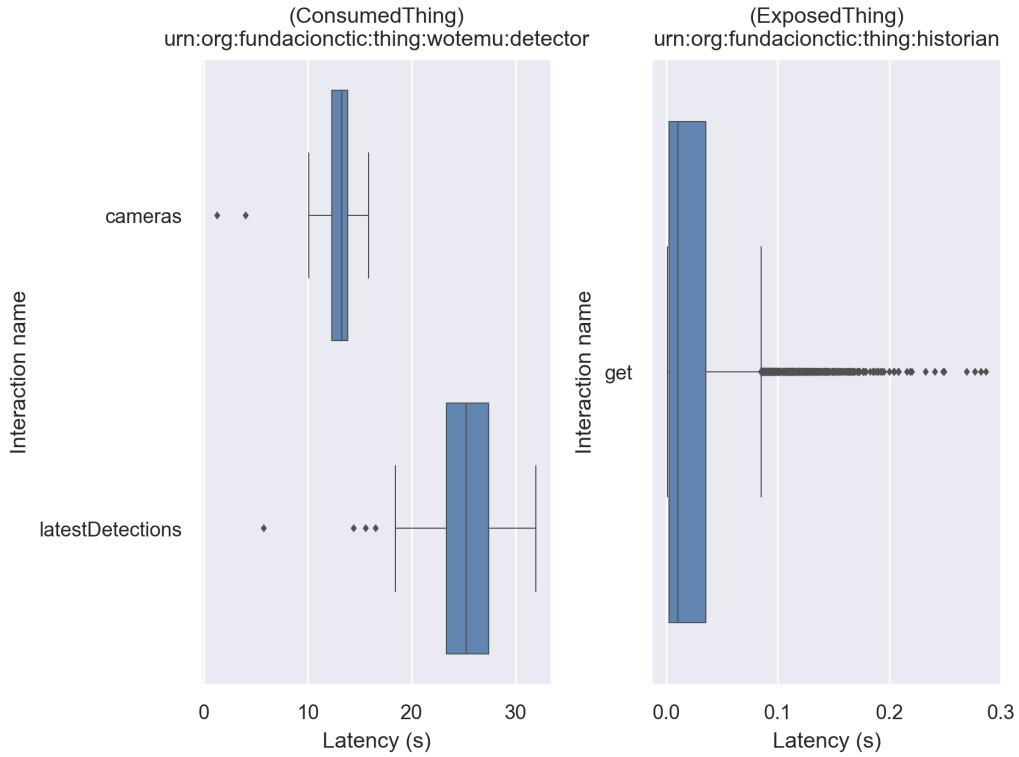


Figura 5.25: Latencias de interacciones WoT en nodo *cloud* (*edge*)

Capítulo 6

Conclusiones

Las aportaciones principales de la tesis doctoral desarrollada son las siguientes:

- Un diseño de un **framework basado en los bloques fundamentales W3C WoT**. Este framework proporciona un WoT Runtime plenamente funcional con soporte para todos los verbos de interacción y todos los protocolos de capa de aplicación recomendados en las especificaciones. Hasta donde llega nuestro conocimiento, este es el primer framework de estas características en el momento de su publicación.

- Un diseño de un **emulador de aplicaciones WoT basadas en el patrón de arquitectura edge computing**. El emulador propone una solución al problema de validación de arquitecturas IoT a través del aprovechamiento de la repetibilidad y escalabilidad horizontal de Docker Swarm mode, el orquestador de contenedores incluido por defecto en el ecosistema Docker. Se proporciona visibilidad del comportamiento de los sistemas WoT en múltiples niveles, incluyendo las capas bajas de red, la capa de aplicación y a nivel de infraestructura hardware. Obtener esta visibilidad es normalmente una tarea compleja y que además requiere invertir esfuerzos en preparar un despliegue real.

Estos diseños han dado lugar a las respectivas implementaciones experimentales y a publicaciones de impacto. Sus referencias se indican en la tabla 6.1. Además los dos componentes han sido validados y evaluados a su vez con sendos experimentos:

- Se ha desarrollado un conjunto de aplicaciones WoT utilizando el framework. Después, se han combinado en varias configuraciones representativas de escenarios reales. Las configuraciones incluyen la ejecución

Resultado	Publicación de impacto	Implementación experimental
Framework	[64] *WOTPY: A framework for web of things applications* (apéndice A)	[42] `WoTPy`
Emulador	[79] *WoTemu: An emulation framework for edge computing architectures based on the Web of Things* (apéndice B)	[78] `WoTemu`

Cuadro 6.1: Resumen de resultados obtenidos

en plataformas hardware reales procedentes de todas las capas del modelo edge computing. La ejecución de las configuraciones ha dado lugar a una serie de resultados de rendimiento que caracterizan los verbos de interacción en cada una de las implementaciones de Protocol Binding. Los resultados permiten además obtener conclusiones acerca de los protocolos más adecuados para cada escenario. Los protocolos nativos de la Web (HTTP y Websockets) demuestran un buen rendimiento, aunque podrían considerarse excesivamente pesados para dispositivos de capacidades limitadas. Estos dos protocolos se complementan mutuamente, ya que Websockets es más adecuado para verbos de interacción basados en comunicaciones iniciadas desde el servidor, lo cual resulta una debilidad clara de HTTP. Por otra parte, MQTT es una buena opción de propósito general, que tiene la desventaja de necesitar un broker externo, mientras que CoAP es la mejor opción para garantizar bajos niveles de transferencia de datos.

- En el caso del emulador, se ha diseñado una arquitectura para dar solución a un caso de uso real: un sistema de videovigilancia con capacidades de detección y seguimiento de personas. Se incluye el uso de modelos de inteligencia artificial, restricciones realistas de red y consideraciones de optimización del coste de la plataformas hardware. Concretamente, se proponen dos versiones de la arquitectura, una versión basada en un modelo centralizado simple y otra versión que aprovecha los beneficios de la arquitectura *edge computing*. El emulador (WoTemu) ha sido instrumental en la detección de cuellos de botella a través del análisis de métricas de rendimiento representativas, y ha permitido la optimización de recursos computacionales y de red.

De los experimentos con el emulador se extrae además una confirmación de la importancia de este tipo de herramientas en el proceso de diseño IoT o WoT. Estos sistemas se caracterizan por un número elevado de dispositi-

vos heterogéneos interconectados, lo que hace que sea notablemente costoso comprometerse a un despliegue real para validación y pruebas. La emulación basada en código real supone un compromiso razonable que reduce significativamente los costes a cambio de una disminución en la precisión de los resultados (en comparación con la ejecución en hardware y redes reales). Esta pérdida de precisión no suele resultar relevante para la detección de cuellos de botella y problemas significativos en el diseño, lo cual es una de las prioridades en las fases tempranas. Una de las ventajas más notables de este emulador propuesto es su capacidad para ejecutar experimentos de gran volumen a través de las capacidades de escalabilidad horizontal de tecnologías de contenedores modernas; esta capacidad es única en el contexto del estado de la técnica.

En un plano más subjetivo, las experiencias extraídas de los experimentos indican que la Web of Things es una aproximación que verdaderamente tiene la capacidad de reducir el impacto del problema de interoperabilidad en el IoT. El modelo de interacciones y Things resulta razonablemente intuitivo, permitiendo el desarrollo rápido y minimizando el tiempo hasta el despliegue. Los desarrolladores pueden concentrarse en la lógica de negocio que distingue a sus aplicaciones en vez de invertir esfuerzos en diseñar los componentes de bajo nivel que determinan el sistema. Además, es un hecho indiscutible que la Web es uno de los dominios tecnológicos mejor documentados y más accesibles actualmente. Es común que especialistas en campos diversos, tales como el diseño de sistemas embebidos, aplicaciones móviles o sistemas de control industrial, tengan un cierto nivel de conocimiento de los componentes que forman la Web: principalmente el protocolo HTTP y el ubicuo patrón arquitectural REST.

La comunidad se encuentra actualmente en el proceso de desarrollar implementaciones de los bloques fundamentales WoT para un conjunto amplio de plataformas y lenguajes. Para más información, la reciente *review* del dominio WoT presentada por Sciullo *et al.* [97] incluye una vista exhaustiva de todas las iniciativas, entre las que se encuentran nuestros trabajos. Se incluyen frameworks para dispositivos con capacidades computacionales más altas, así como librerías más limitadas en funcionalidad que tienen como objetivo el uso en microcontroladores y plataformas de bajas capacidades.

Esta tendencia apunta a que en un futuro cercano se establecerán implementaciones maduras de código abierto que reducirán los costes de entrada. Consideramos que la presencia de estos componentes software de calidad contrastada es un requisito básico para asegurar la relevancia continuada de la W3C WoT. Es probable que los integradores de sistemas IoT descarten la opción W3C WoT si se encuentran con complejidades de desarrollo, especialmente en comparación con otros ecosistemas IoT con planteamientos

más sencillos (muchas veces con el compromiso de la presencia de tecnologías propietarias).

El principio de *flexibilidad* que guía la W3C WoT es otra de las características que pueden garantizar su estabilidad a futuro. Otros ecosistemas IoT buscan sustituir en mayor o menor medida las alternativas existentes, a veces por motivos comerciales. Sin embargo, la arquitectura W3C WoT está planteada para dar cabida a despliegues y tecnologías existentes sin necesidad de llevar a cabo grandes inversiones. Las tecnologías que sirven de cimiento de la WoT son libres y están implementadas en prácticamente todas las plataformas.

Los retos derivados de la interoperabilidad pueden aparentar ser de menor impacto que los detalles funcionales y tecnológicos de los despliegues IoT. Sin embargo, esto es una percepción engañosa, ya que los efectos de los problemas de interoperabilidad se manifiestan en las fases avanzadas de los proyectos, una vez que la inversión de recursos ya es significativa y resulta difícil corregir el rumbo. En otras palabras, la interoperabilidad puede parecer un reto sospechosamente simple a primera vista, lo que puede aumentar el riesgo efectivo. Es por esto que los trabajos de investigación en la línea de la interoperabilidad, concretamente de la WoT, son relevantes a día de hoy y seguirán siéndolo en los próximos años. Dentro de este dominio, nuestros trabajos han intentando aportar visibilidad, así como validar la arquitectura WoT con implementaciones y experimentos en el contexto del mundo real.

Finalmente, en relación a las líneas de trabajo futuro, identificamos dos oportunidades principales:

1. La actualización *continua* de las implementaciones experimentales de los bloques fundamentales W3C WoT para exploración de las nuevas versiones de las especificaciones. Estas nuevas versiones introducen cambios significativos que van más allá de las modificaciones menores de API, lo que hace que sea interesante llevar a cabo pruebas en casos de uso de mundo real.

2. El diseño de nuevos experimentos en el contexto del emulador de aplicaciones WoT. Con estos experimentos se perseguirá comprobar activamente los límites efectivos de volumen de experimento, es decir, los límites de escalabilidad de las herramientas de orquestación de contenedores que sirven de cimiento. Por otra parte, también es de interés la validación de arquitecturas reales y el estudio de resultados de los trabajos actuales en el dominio. A continuación se muestran algunos ejemplos concretos:

a) El sistema de gestión de *smart grids* en [98] que persigue la optimización de la latencia de medidas de *smart meters* a través de la aplicación del patrón edge computing.

b) La comparación con los resultados obtenidos por el modelo de optimización de asignación de recursos en arquitecturas edge computing descrito en [99]. Concretamente, los autores modelan una aplicación basada en detección facial que es razonablemente comparable a la presentada en la sección 5.2.

c) La implementación de un sistema de delegación de procesos de drones a dispositivos en capas edge para minimizar el tiempo de misión propuesto en [100].

Apéndice A

Publicación framework WoTPy

Este apéndice incluye una publicación de impacto derivada de los trabajos descritos en la sección 3. El artículo a continuación se presenta en su versión *accepted manuscript*, que no incluye los trabajos de formato final del editor. La referencia completa de este artículo se muestra a continuación:

Referencia de la publicación derivada del framework WoT

García Mangas, A., & Suárez Alonso, F. J. (2019). WOTPY: A framework for web of things applications. Computer Communications, 147, 235–251. `https://doi.org/10.1016/j.comcom.2019.09.004`

WOTPY: A framework for Web of Things applications

Andrés García Mangas[a,*], Francisco José Suárez Alonso[b]

[a] *Technological Center for Information and Communication (CTIC), W3C Spain Office host, Ada Byron 39, 33203, Gijón, Spain*
[b] *Department of Computer Science, University of Oviedo, Spain*

Abstract

The interoperability problems that originate from the heterogeneity in protocols and platforms is one of the main challenges currently faced by the Internet of Things (IoT). The Web of Things (WoT) is an architectural solution to this issue based on leveraging the Web as a means to ensure interoperability. The World Wide Web Consortium (W3C) is currently behind one of the most relevant WoT initiatives—a group of building blocks to serve as a possible foundation for the WoT. This work describes an experimental framework based on the W3C WoT, including a set of concrete and original protocol binding implementations (HTTP, Websockets, MQTT and CoAP). One of the main novelties is that all protocol binding implementations have support for all interaction verbs from the WoT interaction model. The framework is especially adequate to build WoT applications for devices on all layers of the fog computing model; this multi-layer integration is achieved by leveraging the W3C WoT architecture and interaction model. A functional implementation in Python is also described, including low-level designs and implementation details for the binding templates. The behavior of the framework and the protocol bindings is studied by implementing a benchmark application under multiple conditions and hardware platforms. Finally, recommendations are extracted from the obtained results for the most adequate protocols for each scenario and interaction verb.

Keywords: Web of Things, Internet of Things, Fog computing

1. Introduction

Multiple platforms and protocols for interconnection of IoT devices can be found in the current IoT landscape [1]. The lack of a common standard and clear interoperability guidelines can result in increased maintenance and development costs. It could be expected that the scalability problems of the IoT will become more severe if the current growth rate is maintained in the near future. The WoT is a recent approach in the IoT domain that is focused on solving the IoT interoperability problem by using the Web as an integration layer. The Web, as one of the pillars of the Internet, offers a reasonable chance of reaching an industry-wide consensus on the path to realizing IoT interoperability [2].

IoT devices can be interconnected using a wide array of protocol choices on each layer. These protocols contribute different messaging paradigms and strengths—there are usually multiple valid options for each scenario. For example, a group of nodes may use a network layer protocol focused on constrained devices such as 6LoWPAN to optimize battery usage, or opt for the more commonplace IPv4 for its widespread support. There is another level of integration that comes from using an IoT middleware solution to build applications and compose isolated IoT deployments into complex systems [3]. Some of these solutions focus on describing standard architectures and interfaces (like the ones discussed in 2.1); while others take a more commercial approach in the form of cloud services (PaaS or SaaS) based on proprietary or open source technologies. In the WoT the specifics of protocols and middleware are encapsulated and exposed using mature Web standards and architectural patterns that are well known by the community (e.g. HTTP, Websockets, JSON, REST).

There are multiple views of the IoT [1] that identify three layers (or subdivisions of these): a bottom layer that includes IoT sensors and actuators (commonly implemented on constrained hardware); a middle layer that includes the hardware and software services for networking and interconnection; and a top layer that implements the business logic. The concept of fog computing [4] is derived from an alternative model where layers are characterized depending on their proximity to the core (i.e. cloud servers).

In the fog computing model (also known as edge computing), low power sensors are located at the bottom, while the cloud layer at the top hosts a limited number of devices without practical limitations to computational and storage capabilities. In the middle sits the fog layer, representing a compromise between the low latency that exists at the sensors layer, and the raw computational power and resources available at the cloud layer.

*Corresponding author
Email addresses: agmangas@gmail.com (Andrés García Mangas), fjsuarez@uniovi.es (Francisco José Suárez Alonso)

Costs associated with processing huge volumes of data in the cloud and the necessity for minimal latency lead to the current relevance of the fog computing paradigm [5]. Fog computing brings multiple benefits to common IoT demands such as optimization of latency compromises, storage distribution and replication, and flexibility when assigning computing resources. Currently relevant fog computing challenges include abstracting from heterogeneous capabilities in devices and addressing security concerns.

The W3C WoT, as one of the most relevant initiatives in the WoT field, is able to translate proven solutions from the Web that could be applied in the context of the fog computing model to reduce the impact of issues related to interoperability.

The main contributions of this paper are:

- Concrete designs for a group of W3C WoT protocol binding implementations that map a set of application layer protocols (HTTP, Websockets, MQTT, CoAP) to the WoT interaction model. These bindings have support for all interaction verbs on all protocols, and are included in a framework based on the architectural designs of the W3C WoT [6]. A functional implementation of the framework for the Python platform is also described.
- A comprehensive performance evaluation of multiple deployment scenarios obtained from the experimental implementation of the framework. We provide an analysis of the best usage scenarios for each protocol binding implementation in the context of the WoT interaction model based on the experiment results. Moreover, the obtained performance data could be used in the future to simulate and optimize IoT application deployments based on the fog computing model and the W3C WoT architecture.

The rest of the article is organized as follows. Section 2 provides a review of relevant IoT interoperability solutions, focusing on those located in the WoT domain. It also discusses recent research efforts on IoT gateways for the fog layer. Section 3 presents the framework architecture and the low level details for each of the protocol binding templates. Section 4 describes the experiments used to evaluate the framework performance, including scenarios and metrics; while section 5 shows and discusses the results obtained from said experiments. Finally, section 6 presents the conclussions for the previous analysis of the results.

2. Background

There are multiple references to the REST architecture in the following sections. REST is an architectural pattern for building distributed systems with a well defined set of constraints. It is not tied to any particular protocol nor representation format [7], although it is commonly applied in the Web to build HTTP APIs using JSON/XML. It should be noted that on a closer inspection some entities claiming to use REST do not fully meet the requirements (e.g. not being strictly hypertext-driven or displaying some level of coupling between client and server regarding the hierarchy of resources). Regardless, both in the literature and in industry they are commonly categorized as REST. We therefore take the same approach for simplicity.

2.1. Interoperability solutions

This section describes tools and architectural solutions to the IoT interoperability problem that are currently relevant in the IoT domain. A clear distinction is made between the solutions that take a classic (non WoT) approach and the ones based on the Web (i.e. WoT). We focus on describing the W3C WoT building blocks as they serve as the foundation of our work.

Table 1 provides an overview of the interoperability solutions referenced in this section.

2.1.1. IoT interoperability

The Open Connectivity Foundation (OCF) specification [8] is based on a REST architecture implemented over CoAP with UDP or TCP transports and CBOR serialization. An interesting feature of the OCF is that it provides explicit descriptions for various resource and device types from different IoT verticals (e.g. smart home).

LwM2M [9] is also built around CoAP and REST. Messages are transported over UDP or SMS (no TCP). LwM2M clients expose object instances containing a set of resources that are consumed by LwM2M servers. In practice, this means that all LwM2M clients arguably act as servers.

OneM2M [10] identifies three types of *functional entities* that act as the core logical components of any IoT deployment—ad hoc applications, common horizontal services and network connectivity. The high-level REST architecture of OneM2M can be mapped to a wide set of application layer protocols (MQTT, Websockets, HTTP, CoAP) and IoT platforms (e.g. LwM2M, AllJoyn) using protocol binding specifications and interoperability guides. Parallelisms can be found with the W3C WoT protocol bindings discussed in section 2.1.2.

A distinct approach centered around battery powered constrained devices can be found in the Smart Object (SO) framework [11]. SOs consist of atomic groups of sensors identified with RFID tags, and establish ad-hoc networks between them by relying on clustering techniques based on battery usage forecasts. This framework opts for SOAP-based XML Web Services instead of the usual REST APIs.

The IMPReSS Systems Development Platform (SDP) [12] puts the focus on providing horizontal software components to accelerate the development of IoT applications. IMPReSS distinguishes itself from other solutions by including a reusable User Interface (UI) components layer. The middle layer consists of a set of middleware APIs for

IoT interoperability solution	WoT	Highlights
[8] OCF		CoAP over TCP/UDP. REST. Explicitly defines devices and properties for multiple verticals.
[9] LwM2M		CoAP over UDP/SMS. REST. Comprehensive set of interfaces for communications between LwM2M clients and servers.
[10] oneM2M		REST. Multiple application-layer protocols and interoperability guides.
[11] Smart Object (SO) framework		Framework based around constrained nodes identified using RFID (SOs). SOs build ad-hoc networks based on algorithms for optimal battery usage.
[12] IMPReSS SDP		Middleware and UI components to accelerate development of IoT applications on multiple verticals. XMPP. HTTP REST. MQTT.
[13] Guinard, Trifa, Wilde	•	One of the first to use HTTP REST to integrate IoT devices into the Web and set the foundations for the WoT.
[6, 14, 15, 16] W3C Web of Things	•	Four building blocks (WoT architecture, Scripting API, protocol bindings and Thing Description) that describe a comprehensive WoT specification.
[17] Web Thing model	•	Requirements for an IoT device to be considered Web-enabled. HTTP REST.
[18] Mozilla Web Thing API	•	Developer-centric specification to expose IoT devices in the Web. HTTP REST. Websockets.
[19] Paganelli, Turchi, Giuli	•	Things are represented as graphs. Nodes in the graph represent the *interactions* and are modeled as HTTP REST resources.
[20, 21] OGC SensorThings	•	HTTP REST API for sensors and actuators. MQTT extension.
[22, 23] FIWARE	•	Ecosystem of reusable components to build IoT applications. Data model based on JSON-LD.
[24] WoTDL2API	•	Automatic generation of HTTP REST APIs to integrate devices in the WoT.

Table 1: Summary of IoT interoperability solutions

common IoT services (e.g. storage). The bottom layer includes a series of adapters for concrete IoT devices or platforms, thus enabling support for multiple messaging models.

2.1.2. WoT interoperability

One of the first works on the WoT was presented by Guinard, Trifa and Wilde [13]. It focused on a REST-based architecture to integrate IoT devices into the Web, bringing the nodes from the network layer to the application layer. They argue that HTTP, a protocol traditionally considered too heavy on computation resources in the context of constrained devices, is an adequate choice for IoT deployments.

The W3C WoT is an effort by the W3C to produce a series of standard WoT building blocks [6] (currently a work in progress, although close to completion). In the W3C WoT a Thing is any entity (virtual or physical) that may be represented by a Thing Description (TD) [14]. The TD contains all information required to represent and access the functionalities exposed by the Thing to the network and is commonly serialized in JSON-LD format, although there is also a simplified JSON representation. These functionalities are modeled following the interaction model which identifies three types of interactions:

- *Properties* represent atomic attributes or values of a Thing (e.g. temperature, current speed). Execution times of read or write requests for properties should be reasonably short.
- *Actions* represent procedures of unknown duration that may be invoked on a Thing (e.g. restarting a virtual machine, draining a water tank).
- *Events* are occurrences observed by the Thing within the context of its environment that are communicated to subscribers following a push messaging pattern (e.g. a proximity sensor has detected something, an autonomous vacuum cleaner has arrived at the dock).

High level operations from the interaction model are translated to low level exchanges in an specific protocol using protocol binding implementations [15] (e.g. property reads can be translated to HTTP GET requests on an HTTP-JSON binding). There may be multiple implementations to interact with any given IoT platform.

The WoT servient is one of the main entities found in the W3C WoT ecosystem, acting as both client and server for Things. A WoT servient is based on a WoT runtime layer that implements the interaction model. This layer enables the developer to interact with remote Things (*consumed Things*) by interpreting their TD documents, and to create and manage local Things (*exposed Things*). The aforementioned protocol bindings implementations are located in a layer under the WoT runtime, serving as a

proxy between the network and both consumed and exposed Things. The final building block of the W3C WoT is the Scripting API [16], which is the API used by developers to interact with the WoT runtime and build WoT applications.

The node-wot package [25] provides a reference Node.js implementation of a W3C WoT runtime and Scripting API with support for the full set of protocol bindings (i.e. HTTP, Websockets, CoAP and MQTT). A subset of its features are also supported in the browser (i.e. only HTTP and Websockets in client mode).

For further clarity, figure 1 shows a high-level view of the W3C WoT architecture compared with the three-layer fog computing IoT model. The IoT side shows a simplified view with two distinct deployments connected to the cloud with one fog level. The WoT side shows how the W3C WoT building blocks would be integrated in this case.

An innovative application of the W3C WoT can be found in [26]. The authors identify the issue of services that expose end devices in the Web which are not reacting in an adequate fashion to environmental changes. They propose a novel architecture based on the W3C WoT specifications where the system adapts in real time to these changes. User interfaces can be built with normalized parts obtained from a UI parts store and also adapt automatically to the current Thing status.

The Web Thing Model [17] document describes a data model and HTTP REST protocol for the integration of IoT devices in the Web. This document identifies the *Web Thing* as a Web-enabled entity with a minimal set of requirements. The Web Thing API [18] presents a simplified Thing data model with default JSON serialization (i.e. Web Thing Description) along with protocol templates for HTTP and Websockets. Both show parallelisms with the W3C WoT (e.g. the presence of the Action and Property interaction entities). The Web Thing Model was submitted as a possible base for the work of the W3C WoT WG, while the Web Thing API takes a more developer-centric approach with a focus on deployment on current browsers.

Another WoT interoperability solution can be found in [19], where the authors propose a framework based on the prevalent REST architecture with a focus on designing tools to accelerate development. The framework represents Things (e.g. sensors) as graphs. Any given Thing may be composed of multiple nodes, where each node has a URI. A middleware layer exposes the actual REST resources, provides persistence and interoperates with other data sources.

The OGC SensorThings API could also arguably be classified under the WoT domain due to its focus on Web technologies (i.e. HTTP REST, JSON). It is divided in two parts: the first one [20] deals with the exposition and management of IoT sensors; while the second [21] is focused on IoT actuators. An optional extension is also provided to use MQTT instead of HTTP to support the pub-sub pattern.

FIWARE differs from the other interoperability solutions in that it is more a heterogeneous platform than a concrete specification. It provides a set of reusable open source software components called Generic Enablers (GE). Most GEs revolve around the NGSI v2 [22] data model, although the specification seems to be closed and current efforts are directed at developing NGSI-LD [23], which adds semantic capabilities by leveraging JSON-LD. It should be noted that although the NGSI v2 API is based on the HTTP REST and JSON pattern, there are *IoT Agents* to adapt to other protocols and specifications (e.g. JSON over MQTT, LwM2M).

Other works have addressed the costs of integrating existing IoT devices into the WoT. The WoTDL2API [24] tool is designed to build documents compliant with the OpenAPI specification for Things that have been described with WoTDL (an OWL-based ontology). These OpenAPI documents are later deployed as functional HTTP REST APIs using software toolchains.

Recent research efforts in the WoT field have also focused on the semantic interoperability plane to reduce the fragmentation of ontologies and simplify the automatic interpretation of IoT data. To that end, the authors in [27] present a methodology (KE4WoT) to extract the most representative topics from a set of IoT verticals in an automated fashion.

2.2. Fog computing

IoT gateways are one of the main components of the fog layer. These devices can be considered as support nodes, with capabilities beyond the majority of constrained devices [5, 1]. Gateways usually take the role of bridges between different networks and protocols, data stores or centralized computation nodes.

There is a significant degree of fragmentation in commercial gateways. These gateway solutions support different protocols, languages and platforms, unlike the approach used in the Web, where a well-defined set of standards provide interoperability and ubiquitous access. The authors in [28] propose using smartphones as opportunistic gateways using BLE as the transport protocol to mitigate this issue.

A Web of Virtual Things (WoVT) server located in the fog layer is proposed in [29] to solve the interoperability challenge. The WoVT is able to parse W3C WoT, IPSO and OCF thing description formats for increased compatibility with existing deployments. The authors also focus on optimizing development time—end devices may be programmed using a domain-specific language (DSL) that allows the reuse of common software blocks kept in a repository.

FOGG [30] leverages the fog layer to integrate Information Centric Networking (ICN) with the Internet. Content is the first-class citizen in ICN networks, in contrast with IP networks where hosts take the most prominent position. This gateway is able to bridge the gap between IP and ICN by translating ICN protocols, adapting naming patterns and implementing proper security mechanisms.

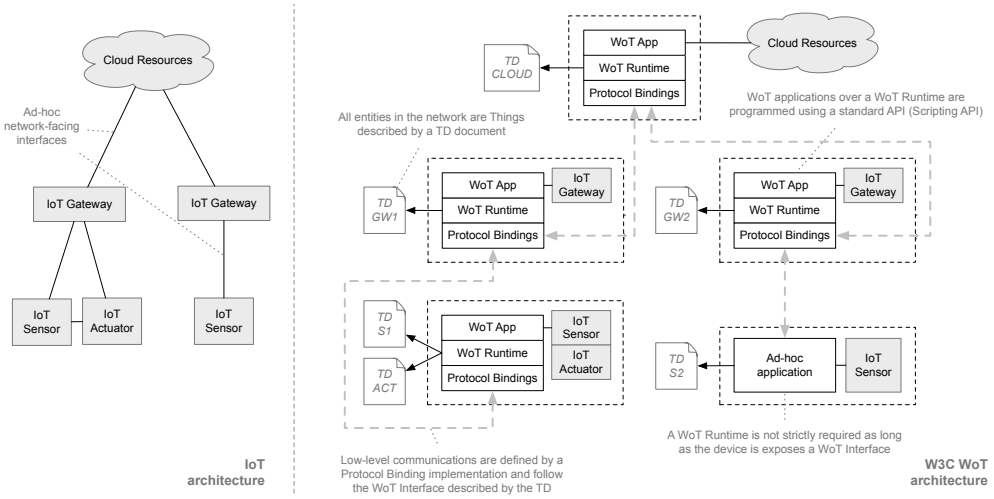


Figure 1: Simplified view of IoT and W3C WoT architectures

Commercial industrial IoT gateway solutions usually have a high cost. In [31] a multi-MCU Modbus TCP gateway is introduced as an affordable solution that overcomes the limitations of using a single MCU. The resulting product has higher performance and lower energy consumption than comparable Modbus gateways with lower costs. The DIIG gateway [32] is another solution focused on the industrial domain, featuring support for S7 comm (a propietary protocol by Siemens) and Modbus TCP. The differentiating feature of DIIG is that it opts to communicate with IoT end devices by using shared memory instead of a common protocol.

The authors in [33] make the distinction between *passive* gateways, which require manual input to add or remove IoT devices from the gateway; *semi-automatic* gateways, which are able to automatically discover and establish connections to new devices, but require manual operation to configure services; and *fully-automatic* gateways which discover and configure devices and services without any explicit input from the user. They present a fully-automatic IoT gateway based on AllJoyn (a discontinued IoT interoperability specification managed by the OCF) with support for multiple network protocols (e.g. 3G/4G, ZigBee, WiFi) that uses CoAP in the application layer.

CoAP features prominently in other works such as [34], which evaluates the impact of multithreading on a gateway that acts a proxy for a set of CoAP-only constrained devices. This gateway keeps a cache for each device and takes a hybrid-proxy approach that combines both push and pull communications between the gateway and the sensors. The results show that blocking probability and latency depend heavily on the number of threads, while cache misses are barely affected.

As a protocol transported over UDP, CoAP uses DTLS for end-to-end encryption. The issue of DTLS being too heavy on computation resources when deployed on constrained devices is addressed in [35]. The authors present a gateway known as SSP (Secure Service Proxy) that sits between CoAP end devices and the other end of a DTLS session, serving as a DTLS proxy. The results show that using the SSP can optimise battery usage and reduce load on the constrained devices.

Other works have leveraged the principles of fog computing and the WoT to take a more application-focused approach. A representative example can be seen in [36], where an innovative system based on HTTP is proposed to integrate a set of parking spots into the WoT.

2.3. Motivation

We have observed that most works on gateways for the fog computing layer are based on one or two application layer protocols. The majority opt for CoAP or HTTP using the REST architecture, which provides a solid foundation but has shortcomings in some scenarios. Some IoT interoperability solutions can be applied in the context of IoT gateways to address this issue, enabling support for a wide array of protocols and messaging models. However, these are normally based around more complex architectures and require a significant investment in terms of time and effort.

Our work leverages the W3C WoT to build a self-contained framework that can be used to support multiple

protocols and integrate fog gateways into the WoT. This approach has the advantage of allowing the same set of skills to be applied on the other layers of the fog computing stack—the framework can also be used to build WoT applications for devices on the cloud or sensors layer, enabling interoperability with reduced resource investment.

To the best of our knowledge, this is the first experimental implementation of a W3C WoT runtime with complete support for all interaction verbs in all application layer protocols referenced by the specifications [15]. The W3C WoT is a very recent and exciting prospect that represents one of the best chances for industry-wide IoT interoperability; thus, we believe that the framework described in this work adds value to the current IoT landscape. The following list describes the motivation in more detail:

- The latest version of the reference Node.JS implementation [25] has some issues at the time of writing. Protocol support appears to be inconsistent and some protocol features are not fully utilized. For example, property read and write verbs are not supported in the MQTT binding; the CoAP binding does not leverage CoAP *Observe* on property observe requests; and the implementation of the Websockets client seems to be empty. These issues are explicitly addressed by our work.

- The design and implementation of concrete bindings to adapt protocols with different capabilities and paradigms to the W3C WoT interaction model is a task that requires careful validation (e.g. a naive approach in the HTTP binding for interaction verbs based on server-initiated communications would probably result in unacceptable performance). This is an issue that is currently not fully solved and is a first step for the adoption of the WoT as an IoT interoperability solution. Our framework proposes original concrete designs that leverage the strengths of each protocol, while at the same time taking into account their weaknesses to ensure full support of all interaction verbs. It should be noted that TD documents are able to describe any possible interface, leaving the design process to the developer; however, a set of predefined bindings such as these may reduce the barriers of entry. Furthermore, although the W3C WoT recommendations provide some guidance, there are no specifics on the actual templates for each protocol. These original bindings are described in section 3.1 and validated with the experiments in section 4. They are our main contribution and may serve as a possible foundation for future works in the WoT domain.

- Python and MicroPython (a port of the Python runtime adapted for constrained microcontrollers) are popular languages for IoT projects [37, 38]. The existence of an alternative implementation in Python complements the reference Node.JS implementation and may result in an increased adoption rate, thus reducing the impact of the IoT interoperability problem. One of the main challenges lies in the fact that the Scripting API specification [16] has been designed for Javascript and its asynchronous nature, which means that the API does not always translate easily to Python. Implementation challenges are described in section 3.2.

3. Framework design

This section describes the architecture and design of the WoT runtime introduced in this paper. The design is based on the reference architecture of the WoT servient provided by the W3C WoT specifications [6]. The main difference resides in the protocol bindings layer—all four protocol binding implementations have been designed from the ground up. Other minor differences from the reference implementation include a custom discovery module based on DNS-SD and an ad-hoc *Servient API* to manage the lifecycle of the servient.

The Protocol Binding implementations module contains four submodules that implement a common interface, encapsulating the low level details of each of their protocols. The servient observes the local set of exposed Things and invokes the binding implementations to regenerate the URIs and connection parameters for the given interaction and protocol when changes are detected (e.g. adding a new property to a Thing). The entities that contain the URIs and parameters are referred to as *Forms* and are part of the TD model. All binding implementations operate in the same way from a high level point of view—on the network-facing side they expose and consume Thing interactions that follow the specific patterns of their protocols, while on the servient-facing side they interact with the Things using the Scripting API.

A WoT application (i.e. programs that are deployed over servients and implement the application logic) that is running on a servient can discover other Things using two methods: a *local* method that restricts the search to the locally registered Things; or a *multicast* method based on zero-configuration technologies, namely DNS Service Discovery (DNS-SD) [39] and Multicast DNS (mDNS) [40]. The mDNS protocol is a decentralized version of DNS where nodes exchange information about domain names by using multicast messages in a link-local context. DNS-SD defines how to use DNS to browse and register services in a network, and is commonly used alongside mDNS to provide a decentralized service discovery solution. Services in DNS-SD are identified by a well known public *service type* and a human readable *service name* that is different for each service instance in the link. The service type in this case is defined as `_wot-servient._tcp.local` (transport is TCP due to the fact that the service registered for discovery is an HTTP server), while the service name is built from the hostname of each servient. A hostname can

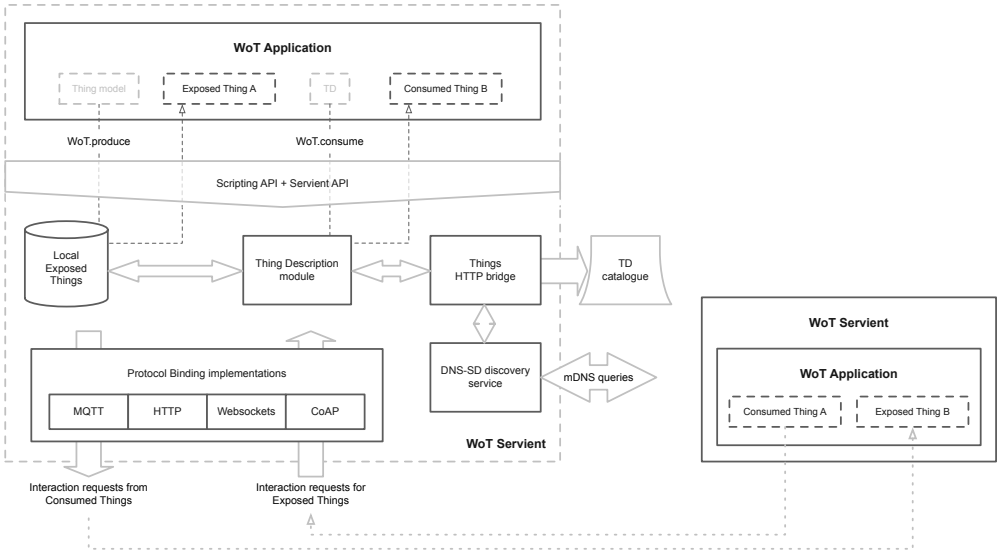


Figure 2: Framework architecture

be explicitly set, although a servient will fall back on automatic hostname (or public IP) detection when none is defined.

The Thing Description module contains logic to validate TD documents according to the specification—all external TDs are validated before building a new consumed Thing to ensure they fit the expected data model. This module also has the capability of serializing internal exposed Thing representations to TDs and vice versa.

Servients rely on the Things HTTP bridge module to expose the internal catalogue of Things for other servients (this is actually the service registered by the DNS-SD module). The root level of the TD catalogue provides a map that lists all Things contained in the servient alongside their specific URIs. The full TD of each Thing in JSON format can be retrieved from those URIs.

WoT applications interact with the lower levels of the servient stack using two APIs:

- The Servient API: a custom interface (i.e. not defined in the specifications) that allows the programmer to interact with the local database of exposed Things; inject instances of protocol binding implementations (clients and servers may be configured separately, although all implementations are active by default); configure the Things HTTP bridge and DNS-SD modules; and manage the lifecycle of the servient.
- The Scripting API: a programming interface defined by the W3C specifications [16] that encapsulates the internal operation of the WoT runtime. It provides a `WoT` object that serves as the entrypoint to *consume* Things from TD documents; *expose* Things from *Thing models* (i.e. either a TD document serialized in JSON format, a consumed Thing, or a *Thing fragment*, which is an object specific to the implementation that models the TD hierarchy); and discover new Things in the environment. It also defines the interfaces of exposed and consumed Things.

The framework is designed to run on an implementation of an asynchronous event loop. An event loop is an alternative to traditional multithreading strategies when implementing software with a significant load of I/O operations. Event loops run on the main thread periodically checking for pending I/O operations—these operations are asynchronous and non-blocking in nature. Most event loop implementations leverage low level Linux APIs (e.g. `epoll`) to be notified when a file descriptor has been updated (e.g. a network socket has received new data). This approach has been found to be especially adequate on scenarios with high network I/O load [41]. Request throughput is usually higher compared to multithreading solutions, as long as requests are not computationally heavy.

3.1. Protocol templates

This section describes each protocol binding implementation, that is, the specification of how high level inter-

Verb	Type	Description
Read	Property	Read the property value.
Write	Property	Update the property value.
Observe	Property	Subscribe for notifications of changes in the property value.
Invoke	Action	Invoke (i.e. run the procedure associated with) the action.
Subscribe	Event	Subscribe for notifications of emissions of the event.
Unsubscribe	Event	Unsubscribe from an active subscription.

Table 2: Interaction verbs

action verbs are mapped to low level protocol messages. Interaction verbs are the set of operations that can be applied to any given interaction [15]. A list of interaction verbs is presented in table 2.

3.1.1. HTTP

HTTP is a stateless protocol based on the request-response model that serves as the foundation of the Web. As such, it is a first class citizen in most WoT environments. As seen in section 2 the majority of HTTP API implementations in the IoT domain are currently based on the REST architecture. Thus, we take this approach for the HTTP binding. It should be noted that this binding borrows heavily from REST, but it would be more appropriate to categorize it as an HTTP API rather than a REST API. This is because it does not strictly meet the HATEOAS (Hypermedia as the Engine of Application State) constraint that requires the API to be navigable in a hypermedia-driven fashion. It could be argued that the TD serves for this purpose, as it provides the full hierarchy of resource URIs in a single root entry point, avoiding namespace coupling between client and server.

HTTP REST does have some shortcomings when applied in the context of the WoT, one of the most notable being the absence of a well-supported solution for server-initiated messaging [42]. A widespread alternative to native push messages is the long-polling pattern, where the client repeatedly polls the server for data, with the server keeping the connection open until there is data to send. One of the main advantages of long polling over simple polling is that it avoids the costs of opening and closing the HTTP connection for emtpy responses. Relevant alternatives to long polling for push messaging in HTTP are chunked transfer encoding in HTTP 1.1 (i.e. HTTP streaming) and the native streaming capabilities of HTTP 2. We opt for long polling because we focus on being developer-friendly and using well known patterns for easier integration, (other bindings can provide native push messaging if the need arises).

We identify five different resources described in the following list. Interaction verbs are mapped to HTTP methods following the standard semantic as required by the REST *uniform interface* constraint.

Property Represents the value of a property. GET requests are used to read the property, while updates use the PUT method.

Property subscription Represents an active subscription to notifications on property updates. The binding implementation uses the long-polling pattern—HTTP binding servers subscribe to the requested exposed Thing property (GET) using the Scripting API and wait for the first emission, sending the updated value to the client and disposing of the subscription immediately afterwards.

Action Represents action procedures that can be invoked (created) using POST requests. A successful request returns without waiting for the procedure and results in the creation of an *Action invocation* resource whose URI is returned as part of the response (following the HATEOAS principle).

Action invocation Represents instances of action procedures that are still in progress or are already completed. The status and result, if any, of an action invocation can be retrieved using a GET request. The binding implementation follows the long-polling pattern in this case. Active action invocations are kept in memory, with completed invocations being cleaned automatically after a configurable amount of time has passed since the last time the action result was actually retrieved by a client.

Event subscription Represents a subscription to notifications of event emissions. The actual implementation is similar to that of *property subscriptions*—event emissions are delivered using the long-polling pattern after a GET on the resource. Event payloads are contained in the response body.

3.1.2. Websockets

Websockets is a bidirectional communication protocol supported by all modern browsers that is especially suitable for communications initiated from the server side. It can be used alongside HTTP to satisfactorily cover the majority of scenarios in the Web. Sessions start with a handshake over HTTP that uses the Upgrade header to signal a desired transition to Websockets.

This binding is organized around a remote procedure call (RPC) model based on the JSON-RPC 2.0 specification [43]. The following list describes the four different types of messages that can be exchanged between client and server during a session:

Request Message sent by clients to initiate a procedure call that contains a method name and the call parameters, if any. There is a different procedure type (identified by the aforementioned method name) for each interaction verb except for *unsubscribe*, which is covered by the method *dispose* and serves to unsubscribe from property updates or event emissions subscriptions.

Response Message sent by servers that contains the result of the procedure call of a previous request.

Error This is a subtype of response that is returned by the server when an error arises while processing a request. It contains details about the error (i.e. code, message and optional data) instead of the result of a call.

Emitted item Message that notifies the client about new emissions in a currently active subscription. It contains the subscription ID, type of notification (e.g. property update) and emission payload. This type of message is not mentioned in the JSON-RPC specification and was included to cover the scenario of *streaming* responses (i.e. responses delivered over multiple messages with an unknown total length, which is the case of subscription notifications).

All request messages are identified by a unique ID that is referenced in the corresponding response or error messages. This is due to the fact that responses may arrive at the client in any order independently of the original order of requests. It is actually one of most interesting features of this binding, allowing for a simplified approach to long-running action invocations, where results are delivered as soon as the invocation is completed without the need for polling or other suboptimal patterns.

The subscription process to *observe* properties and *subscribe* to events is different from the request-response pattern of property *read* or *write*, and action *invoke*. An initial request is sent with the name of the property or event that must be observed. The server subscribes to the interaction using the Scripting API and sets a unique ID for the subscription (SID). The SID is then returned as part of the response. An emitted item message will be sent to the client for each property update or event occurrence. The client can identify the emitted item messages by looking at the SID, and may unsubscribe by sending a *dispose* call passing the SID as parameter—the server will dispose of the active subscription upon receiving this call.

3.1.3. MQTT

MQTT is a lightweight protocol transported over TCP, based on the publish-subscribe messaging pattern. The central entity in MQTT is the broker, a service that acts as a message router and buffer. Clients connect to the broker to publish messages or subscribe to topics; a client that is subscribed to a topic receives the messages published by other clients in said topic. Three different quality of service (QoS) levels can be requested by a client for message delivery, offering a compromise between speed/resources and reliability.

Unlike other protocol binding implementations, which include a functional server without external dependencies, the MQTT server binding implementation does not provide a built-in broker, depending on an external broker instead. The rationale behind this decision is based on three factors:

- MQTT brokers are agnostic to the format of the messages published by clients and do not require programming specific to the application in most cases.
- The probability of having a pre-existing broker that can be reused in an IoT deployment is significant given the popularity of MQTT [1].
- There are multiple implementations of MQTT brokers that are readily available, high quality and open source (e.g. Eclipse Mosquitto, RabbitMQ).

A list of topics used by the MQTT binding is presented in table 3. The `sid` part represents the *servient ID* obtained from the hostname, acting as a namespace to avoid collisions with other servients in a shared broker.

Property read and write verbs are mapped to messages published in the *property request* topic—the message payload contains the verb and the updated value in case of a write. Clients can listen for an optional write confirmation by subscribing to the *property write ACK* topic (a write operation can be identified by providing a unique ID). All property updates are in turn published to the *property update* topic, regardless of the origin of the update. Read operations cause the binding to publish the current value in the *property update* topic, even if the value has not changed from the last time.

An interesting feature of this binding is that there is no need to manage subscription lifecycles to observe property updates. The MQTT server maintains a single active subscription to each property and clients are not required to actively create or dispose of them.

Event subscribe and unsubscribe verbs are designed in a similar fashion to the property observe verb. All events have their own topic where the server publishes all emissions—all clients share the same subscription whose lifecycle is tied to that of the MQTT server; the same happens with properties.

Action invoke is implemented using two distinct topics. Clients first publish a message to the *action invocation* topic containing the input parameters and a unique ID. Results are then published to the *action result* topic when the invocation is ready (clients should subscribe to this topic before actually publishing the invocation message). Each result message is identified with the ID provided by the invocation request.

Topic	Pattern
Property request	`<sid>/property/requests/<thing>/<property>`
Property update	`<sid>/property/updates/<thing>/<property>`
Property write ACK	`<sid>/property/ack/<thing>/<property>`
Action invocation	`<sid>/action/invocation/<thing>/<action>`
Action result	`<sid>/action/result/<thing>/<action>`
Event emission	`<sid>/event/<thing>/<event>`

Table 3: MQTT binding topics

The MQTT binding is especially suitable when observing changes in interactions (i.e. property updates, events) due to its simplicity—the client needs only to subscribe to a topic for each interaction. On the other hand, implementations of verbs closer to the request-response model (e.g. property write) are more cumbersome when compared to protocols such as HTTP.

3.1.4. CoAP

CoAP is a request-response protocol suitable for constrained devices that is commonly found in IoT deployments. It bears multiple similarities to HTTP and REST by design—CoAP servers expose resources identified by URIs that may be manipulated by clients using a subset of HTTP methods (GET, POST, PUT, DELETE) with predefined semantics.

In contrast with the other discussed protocols, CoAP is transported over UDP to avoid the increased resource requirements of a TCP network stack. The characteristic unreliability of UDP is compensated with an optional built-in reliability mechanism based on a 16-bit message ID—reliability may be achieved by using `CON` (confirmable) messages that require the other side to acknowledge reception by responding with either a `RST` (reset) or `ACK` (acknowledgement) message containing the original message ID. Messages may also be transmitted without reliability (i.e. no ack required) using the `NON` (non-confirmable) type.

The datagram-oriented nature of UDP facilitates a solution to the server-initiated messaging shortcomings of HTTP in the form of CoAP Observe. Clients may set the *Observe* option to `0` in the request to initiate the observe process—servers will then send notifications on resource updates using *Token* to identify the subscription and *Observe* to contain a sequence number.

Resources in this binding are similar in their design to the HTTP binding, this is in accordance with one of the main principles of CoAP (i.e. architectural similarity between HTTP and CoAP). One of the most notable differences is that by leveraging CoAP Observe the mapping between high level interactions and modeled resources is more direct (i.e. there is a single CoAP resource for each interaction type). The following list describes the CoAP binding resources:

Property Property reads are mapped to GET requests, while property writes use PUT requests where the encoded value is contained in the request payload. The implementation of the observe verb is based on CoAP Observe as it is a straightforward solution. There is little difference between reads and observations—initiating an observation means that the client will get `CON` read responses on each property update without an actual GET request first.

Action The action interaction resource design is fairly simple compared to the HTTP binding. A new action invocation can be created with a POST request, which returns the unique ID of the invocation as determined by the server. Clients may then observe the invocation and be notified when it completes by sending a GET request stating the invocation ID in the payload.

Event CoAP Observe is also used to implement the event subscribe and unsubscribe verbs. Clients can create a new subscription to the event by observing the event resource with a GET request. The binding then subscribes to the event using the Scripting API and sends a `CON` message back to the client containing the data for each new event occurrence. The unsubscribe verb is implemented with explicit subscription cancellation (i.e. GET with the *Observe* option set to `1`).

3.2. Implementation details

This section describes the most relevant decisions and challenges faced during the implementation phase of the previously described framework. These challenges are in addition to the design challenges that have been addressed in section 3.1. Most of the development effort focused on the protocol binding modules—it is arguably the most complex component and our main contribution.

A functional beta implementation of the framework has been published in the Python Package Index (PyPI) [44]. All evaluation scenarios and benchmarks described in section 4 are implemented on Python 3.7 using this package. It has limited support for a subset of features in Python 2.7 and full support for Python 3.6+.

The HTTP and Websockets protocol binding implementations are based on the Tornado framework [45], while the CoAP and MQTT bindings are based on aiocoap [46]

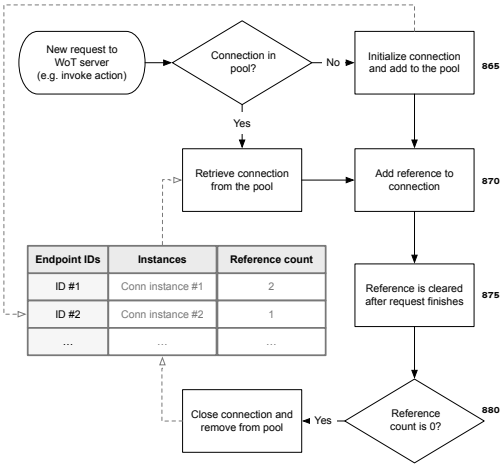


Figure 3: Shared connection pools in stateful client bindings

and HBMQTT [47] respectively. Multicast DNS discovery is implemented on python-zeroconf [48].

One of the first challenges was designing the public API (i.e. Scripting API) of the WoT runtime to be as similar as possible to the one defined in the specification—this contributes towards making developer knowledge portable between runtimes. The main issue is that the specification is heavily based around Javascript Promises and callbacks. Python has an asynchronous I/O module in the form of *asyncio* that introduces the concept of *coroutine* (comparable to an asynchronous function in Javascript) and is only available in Python 3.4 or later. Python 2.7 will reach end-of-life by 2020; however, providing limited support for Python 2.7 will probably help increase adoption in those cases where developers are restricted by a legacy codebase or hardware. We use the Tornado framework to provide an I/O event loop implementation and coroutines API for Python 2.7. In Python 3, Tornado acts as a wrapper around the built-in *asyncio* loop—it does not polute the API for those developers on Python 3 or force them to learn another framework. With this approach, all asynchronous Javascript functions can be translated as *coroutines*, which is respectful with the original signatures.

Results obtained in a first iteration of the benchmarks showed substantially worse performance in the Websockets and MQTT bindings than in the case of HTTP or CoAP, especially in the data transfer volume plane. This was the consequence of a naive first implementation where Websockets and MQTT clients initialized a new connection on each request, regardless of the fact that these connections were to the same servers and coincided in time (e.g. reading a property while invoking an action). A shared connection pool was added to both clients to solve this issue—each instance now maintains a set of reusable open connections in its internal state. The public client interface exposed to the servient by the client implementation had to remain stable, so the connection pool is automatically managed without intervention from the developer. The connection pool logic is described in figure 3.

The shared connection pool implementation entailed various challenges. For example, the automatic handling of connection errors and reconnections to avoid error propagation between requests using the same connection; or the creation of a centralized synchronized queue to distribute incoming messages from the server among the pending requests. After this update, results showed significant improvements in latency and data transfer overheads for scenarios with multiple parallel messages (see section 5).

Multiple edge cases appeared during the benchmarks in the testing phase before the results were deemed stable. These were the result of unexpected interactions between different protocols in extreme cases. Reproducing these issues, implementing patches and writing new test cases proved to be another significant challenge. Some examples of these included:

- HTTP long-polling requests that hanged indefinitely because of unexpected message losses on the MQTT side as a result of using an unrealiable QoS level.
- Errors originated by high request rates that caused the CoAP binding to enter an unstable state due to race conditions in the consumption of internal message buffers.
- Unexpectedly high event losses in the MQTT binding on scenarios with high event rates. This was the result of unstable connections to the MQTT broker. The underlying issue was that the task responsible of sending the MQTT *Keep Alive* messages was being starved of resources.

4. Performance evaluation

This section describes the experiments that were carried out to assess the performance of the framework. The main objectives are to validate our protocol binding designs to discard any possible design flaws; and to identify the best protocol binding depending on the type of scenario. To that end, a set of WoT applications running over the framework implementation (see section 3.2) were tested in different scenarios. These WoT applications and scenarios are described in sections 4.1 and 4.2 respectively. We also plan to use the performance data for future simulation and optimization of IoT application deployments based on the fog computing model and the W3C WoT architecture.

It should be noted that the benefits of the W3C WoT building blocks (i.e. WoT architecture, Scripting API, protocol bindings and the TD) are demonstrated implicitly in

this case. The code footprint for the entire experiment is in the order of hundreds of lines of code; the majority of it being business logic instead of WoT boilerplate. Furthermore, achieving the same functionality without the W3C WoT building blocks and the framework would have required significantly higher investment of development resources. Source code for the experiments is available in [44].

4.1. Experimental design

Three distinct WoT applications were implemented using the framework:

- A *WoT server* application to create and expose a *benchmark Thing*. The interactions of this Thing are designed to characterize all three interaction patterns in a configurable fashion.
- A *WoT client* application to consume the interactions of the benchmark Thing exposed by the WoT server. This application implements the logic to calculate the performance metrics described in 4.3.
- A *WoT proxy* application that takes a TD document URL and exposes a Thing that acts as a mirror of the Thing consumed from said TD.

These applications are combined in different scenarios and configurations (see 4.2) in order to assess the performance of each protocol binding implementation and the framework as a whole.

The following list describes the benchmark Thing interactions exposed by the WoT server:

`currentTime` Property that provides the current timestamp in Unix format as an integer. Implemented using a custom handler that retrieves the system time.

`measureRoundTrip` Action that enters a dummy loop for a random amount of time. Returns an object that contains the Unix timestamps of the request arrival time and the return time of this invocation. This action aims to represent a generic procedure without any particular computation characteristics. To this end, we used a folded normal distribution (i.e. distribution of the absolute value of a normal distribution variable) to model the loop time. This action takes two parameters: a `mu` (μ) parameter, the mean of the distribution; and a `sigma` (σ) parameter, the standard deviation of the distribution.

`startEventBurst` Action that generates a series of emissions of the `burstEvent`. The sequence of events is modeled as a Poisson process so that the interval between consecutive events is random while maintaining a pre-determined average rate. This action takes two parameters: a `lambd` (λ) parameter for the exponential distribution that determines the time between emissions (i.e. target average rate of events per second); and a `total` parameter that represents the total number of events in the burst. It returns an empty result when all events have been emitted.

`burstEvent` Event that is emitted in bursts after an invocation of the `startEventBurst` action. The payload of each event emission is an object that contains the following: the unique identifier of the current burst; the index of this emission within the burst sequence; the Unix timestamp of the beginning of the burst and of this emission; and a flag that indicates if this is the last emission.

4.2. Scenarios

In this context a **scenario** describes the architecture of the deployment for the benchmark (components, physical connections and networks). A **configuration** is a concrete implementation of one of the scenarios (i.e. it defines the specific hardware of each component and protocols). Some scenarios have various configurations.

Diagrams describing each one of the scenarios are presented in figure 4—these align with the deployment scenarios described in the WoT architecture document [6]. The figure references the three types of WoT servients (one for each WoT application) mentioned in 4.1.

The following list describes the configurations considered for the performance evaluation phase, including the rationale for including each one. The WoT client on all configurations is a machine with 4 cores at 2.5 GHz, 16 GB of memory and SSD storage.

Unconstrained Instance of the LAN scenario (figure 4a) where the WoT server is an unconstrained x86 server with 4 cores (8 threads) at 2.6 GHz, 32 GB of memory and SSD storage. This configuration shows the performance on ideal conditions with no resource limitations.

SBC Instance of the LAN scenario (figure 4a) where the WoT server is a single-board computer (SBC) with an ARM quad-core 1.2 GHz CPU and 1 GB of memory (a Raspberry Pi 3 Model B). This aims to represent the approximate performance of a gateway of limited cost that may be present in a domestic home or industrial environment.

Constrained Instance of the wireless LAN scenario (figure 4b) with a WoT server that features a 580 MHz RISC microprocessor and 128 MB of physical memory (an Onion Omega 2+ computer module). The device is configured with an extra 128 MB of swap memory located in a USB 2.0 storage drive. This configuration represents performance on a low cost device with limited resources and wireless connecitivity (i.e. WiFi). These types of devices are usually located in the sensors layer (i.e. not used as edge gateways in the fog layer). They are not suitable for scenarios where high throughput is required, as

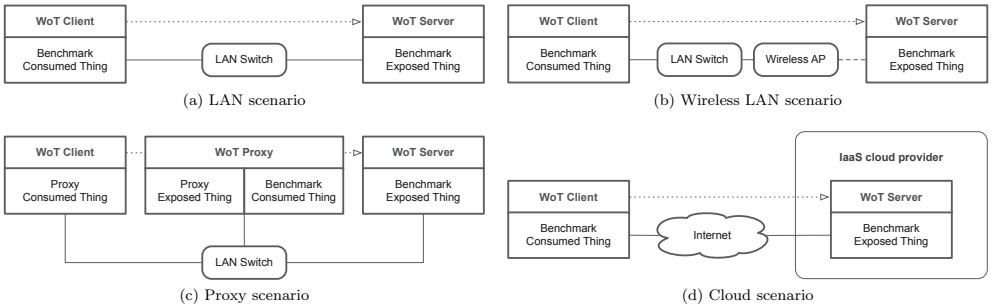


Figure 4: Performance evaluation scenarios

they would run out of memory or block due to the computational load. Hence the set of testing conditions is more limited when compared to the other configurations.

MQTT proxy Instance of the proxy scenario (figure 4c). The WoT server is the same device as in the SBC configuration (i.e. a Raspberry Pi) and is configured only with the MQTT protocol binding. The WoT proxy is deployed on the server from the unconstrained configuration with the entire set of protocol bindings. This represents an scenario where only a single protocol is supported on a gateway in an attempt to optimize resources, while other nodes with more capable hardware expose Things that act as proxies and are *augmented* versions of the Things exposed by the gateway.

HTTP proxy This case is similar to the MQTT proxy configuration, with the main difference that the WoT server is configured to use the HTTP binding instead of MQTT. HTTP is included here due to its widespread use, support and high relevance in the Web (it is probably the best-known protocol of our set).

Cloud Instance of the cloud scenario (figure 4d) where the WoT server is deployed in a cloud instance with 1 core and 1 GB of memory. It should be noted that the IaaS provider does not use CPU throttling in this instance (a practice used by some providers for low cost instances). Represents performance when the WoT servient is deployed on a container in the cloud infrastructure of a prominent IaaS provider. In this case the MQTT broker required by the MQTT binding is deployed in another publicly exposed and separate cloud instance with the same specifications (both instances are located in the same data center, therefore having minimal latency when communicating with each other).

4.3. Metrics

This section describes the performance metrics that are later represented in the results in section 5. Please note that timestamps in the following expressions are in UNIX time format with millisecond precision. Clocks in both the client and server are synchronized using the NTP protocol with the same NTP pool.

Action invocation latency is defined as:

$$L_a = (T_{res} - T_{req}) - (T_{ret} - T_{arr}) \tag{1}$$

where T_{res} (*response*) is the timestamp when the action result arrived at the client; T_{req} (*request*) is the timestamp when the client sent the request; T_{ret} (*return*) is the timestamp when the action handler finished and returned in the server; and T_{arr} (*arrival*) is the timestamp when the server started the execution of the action handler. The difference between T_{ret} and T_{arr} represents the actual execution time of the action procedure, and is subtracted from the latency perceived from the client side to consider only the latencies that result from the protocol binding and the action invocation pipeline in the framework.

Event latency is defined as:

$$L_e = T_{rec} - T_{emi} \tag{2}$$

where T_{rec} (*received*) is the timestamp when the event arrived at the client; and T_{emi} (*emission*) is the timestamp when the server emitted the event.

Property read latency is defined as:

$$L_p = T_{res} - T_{req} \tag{3}$$

where T_{res} (*response*) is the timestamp when the client received the response; and T_{req} (*request*) is the timestamp when the client generated the request.

Data transfer volume (for actions, events and properties) is defined as the total amount of data transmitted by the client through the network interface in both the upstream and downstream. Data is captured and filtered by port at the transport layer (UDP for CoAP and TCP for

the other protocols) using the TShark tool from the Wireshark network analyzer. The port is dynamically retrieved from the TD document of the benchmark Thing.

Action invocation error is defined as:

$$E_a = 1 - (N_{ok}/N_{total}) \quad (4)$$

where N_{ok} is the number of successful invocations; and N_{total} is the total number of invocations.

Event loss is defined as:

$$E_e = 1 - (N_{rec}/N_{total}) \quad (5)$$

where N_{rec} is the number of event occurrences received in the client; and N_{total} is the total number of events emitted by the server.

Property read error is defined as:

$$E_p = 1 - (N_{ok}/N_{total}) \quad (6)$$

where N_{ok} is the number of successful requests; and N_{total} is the total number of requests.

For clarity in the results (i.e. to reduce the number of tables and figures), we do not consider metrics for the property write or observe verbs. These are reasonably characterized by property read and event subscribe/unsubscribe respectively. Processes for property write are mostly equal to property read on all bindings, while the designs for the observation of property updates and event emissions share various similarities.

5. Results and discussion

Results are presented in three groups in the following paragraphs (see 4.1 for a detailed description of the benchmark Thing interactions):

Figures 5, 6 and table 4 present the action invocation latency, data transfer volume and action invocation error for a total of 1000 invocations of `measureRoundTrip`. Invocations are sent in parallel in groups of 1, 5, 10, 20, 50 and 100—actions are long-running procedures, so we opt for characterizing the behavior of the bindings when dealing with multiple parallel invocations instead of a constant rate of requests (as is the case with the property benchmarks). Parameter `mu` (μ) is 0.0 and `sigma` (σ) is 1.0 for all invocations. These parameters cause the execution time to be in the order of 10^2 ms, which is a reasonable compromise for an extended action execution time (one order of magnitude higher than property requests), while the variance is low enough to have most invocations in a batch finish close to each other, adding more stress to the binding.

Figures 7, 8 and table 5 present the event latency, data transfer volume and event loss for six series of emissions of the `burstEvent` event. Each series is initiated with an invocation of `startEventBurst` where `total` is 1000 for all invocations and `lambd` (λ) (target event rate in the server) is 1, 5, 10, 50, 100 and 500 events/s.

	Num. parallel invocations					
	1	5	10	20	50	100
	CONSTRAINED					
CoAP	0.001	0.018	0.029	0.126	-	-
	PROXY-HTTP					
CoAP	0.000	0.000	0.000	0.000	0.001	0.000

Table 4: Action invocation error (1000 total invocations)

	Event rate (1/s)					
	1.0	5.0	10.0	50.0	100.0	500.0
	CONSTRAINED					
HTTP	0.011	0.062	0.099	0.322	-	-
	CLOUD					
HTTP	0.070	0.291	0.461	0.786	0.896	0.969
	PROXY-HTTP					
HTTP	0.013	0.074	0.145	0.301	0.580	0.779
WS	0.015	0.055	0.112	0.372	0.557	0.823
MQTT	0.023	0.060	0.082	0.340	0.592	0.781
CoAP	0.037	0.269	0.430	0.797	0.881	0.972
	PROXY-MQTT					
HTTP	0.014	0.042	0.079	0.385	0.521	0.826
	SBC					
HTTP	0.012	0.046	0.108	0.337	0.518	0.725
	UNCONSTRAINED					
HTTP	0.003	0.032	0.070	0.227	0.372	0.718

Table 5: Event loss (1000 total emissions)

Figures 9, 10 and table 6 present the property read latency, data transfer volume and property read error for a total of 1000 read requests of `currentTime`. Requests are originated in the client with rates of 1, 5, 10, 50, 100 and 500 requests/s.

All data volume and latency figures contain six subfigures, one for each configuration defined in 4.2.

Latency data sets in figures 5, 7 and 9 are represented using letter-value plots [49] instead of the more commonplace boxplot. Using boxplots results in narrow boxes, long tails and many outliers, which are difficult to interpret. Letter-value plots are more adequate for large data sets and convey more information on the distribution of samples on the tails.

Letter-value plots have the following characteristics:

- The box located in the middle (i.e. the widest) is equivalent to a boxplot box—limits are drawn on the

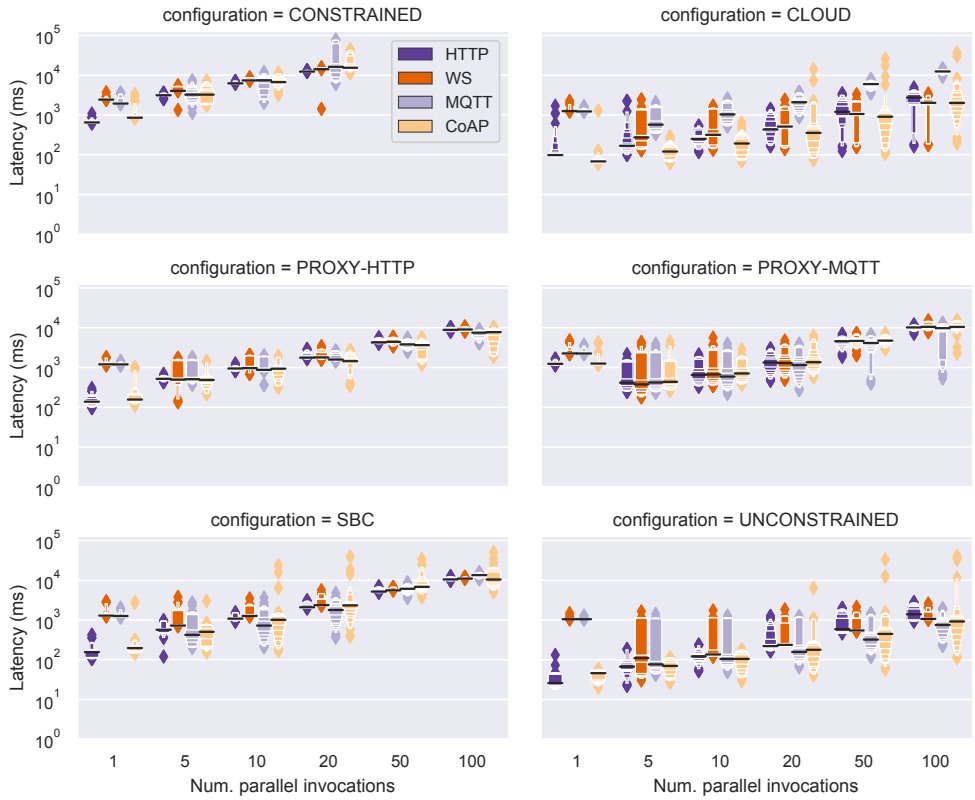


Figure 5: Action invocation latency (1000 total invocations)

lower and upper 4-quantile, while the band inside the box represents the median.

- Successive boxes are drawn on the lower and upper 8-quantile, 16-quantile, 32-quantile, etc. The width of the boxes decreases linearly.
- The number of boxes (k) is determined using the *Tukey* rule: $k = \lfloor \log_2 n \rfloor - 3$, where n is the number of samples. All samples that fall outside the upper or lower boxes are represented as outliers (diamond shape).

It should be noted that the latency figures only show values for successful invocations, event notifications and requests respectively (i.e. errors are omitted).

For conciseness, tables 4, 5 and 6 only show rows where at least one value is different from zero in the series for the given protocol and configuration.

The remainder of the section is focused on the most significant results for each protocol binding, referencing the figures and tables when appropriate. In this discussion we introduce the concepts of *demanding* and *undemanding* conditions. The former refers to high rates (for events and properties) and number of parallel invocations, especially in the context of constrained hardware resources; while the latter refers to sets of parameters that result in relatively low loads for both hardware resources and the network.

5.1. HTTP

The long-polling approach of the HTTP binding does not appear to be suitable for high rates of push messages. Table 5 shows some degree of losses for all conditions, especially for the highest rates (i.e. 100 and 500 events/s) and the scenarios with higher network latency (i.e. cloud). The HTTP binding server unsubscribes internally (using the Scripting API) after the response containing the event

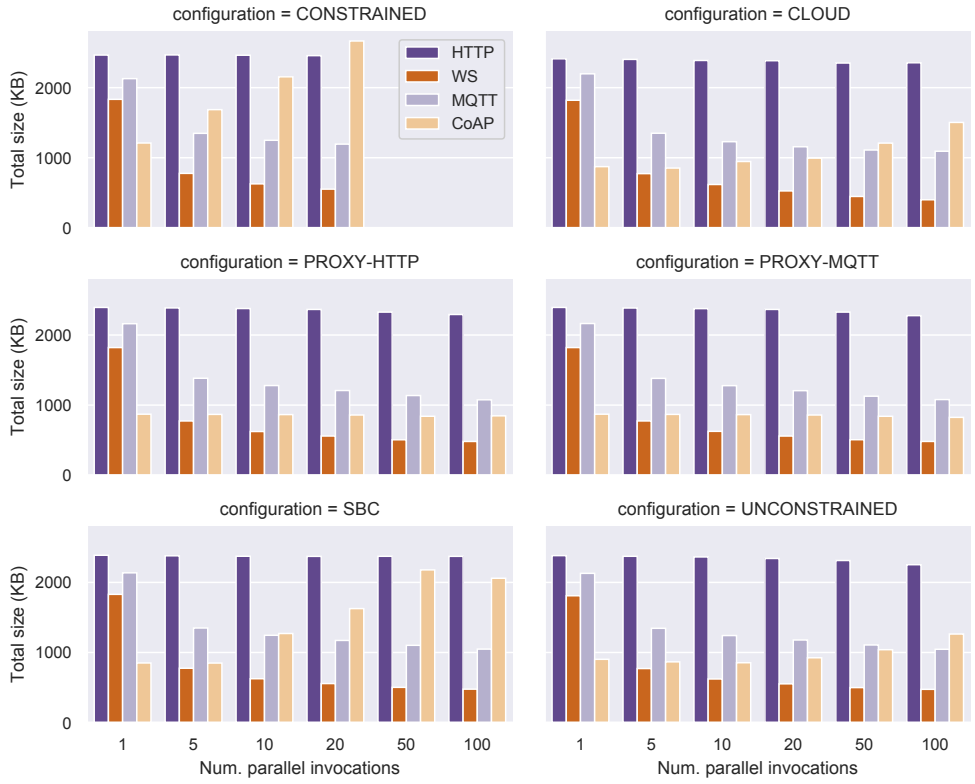


Figure 6: Action invocation data transfer volumes (1000 total invocations)

is sent to the client, hence all the events that occur between that moment and the next long-polling GET request are lost. The client sends another GET request as soon as it receives the event. The HTTP binding subscribes and unsubscribes on each cycle to avoid being stateful in accordance with design principles.

There is a significant degradation in performance when faced with high rates of requests per second on less capable hardware (see error results for the *SBC* and *constrained* configurations in table 6). The majority of HTTP errors are due to request timeouts, which suggests that the HTTP binding is computationally heavy—error rates could be improved by increasing the timeout beyond the recommended defaults if acceptable for the application.

The **HTTP** binding is better used in scenarios with a majority of property reads, property writes and, to a lesser extent, action invocations. These interaction verbs are closer to the HTTP request-response model, and results show acceptable performance except for higher request rates on less capable hardware (figure 9). An increased data transfer volume when compared to other bindings (figure 8) and high loss values for events generated at high rates (table 5) are its two most important drawbacks. It should be noted that the ubiquity and support of HTTP in most contexts, especially in the Web, make this binding a protocol to consider even in the face of alternatives with better performance.

5.2. Websockets

This binding also shows the degraded behavior that results from being computationally heavy in constrained environments. Although there are no request errors, it presents very high latencies in the order of 10^5 ms in the *constrained* configuration for rates ≥ 5 (see figure 9). It is, however, one of the bindings with the lowest data transfer footprint, especially when considering error results (i.e.

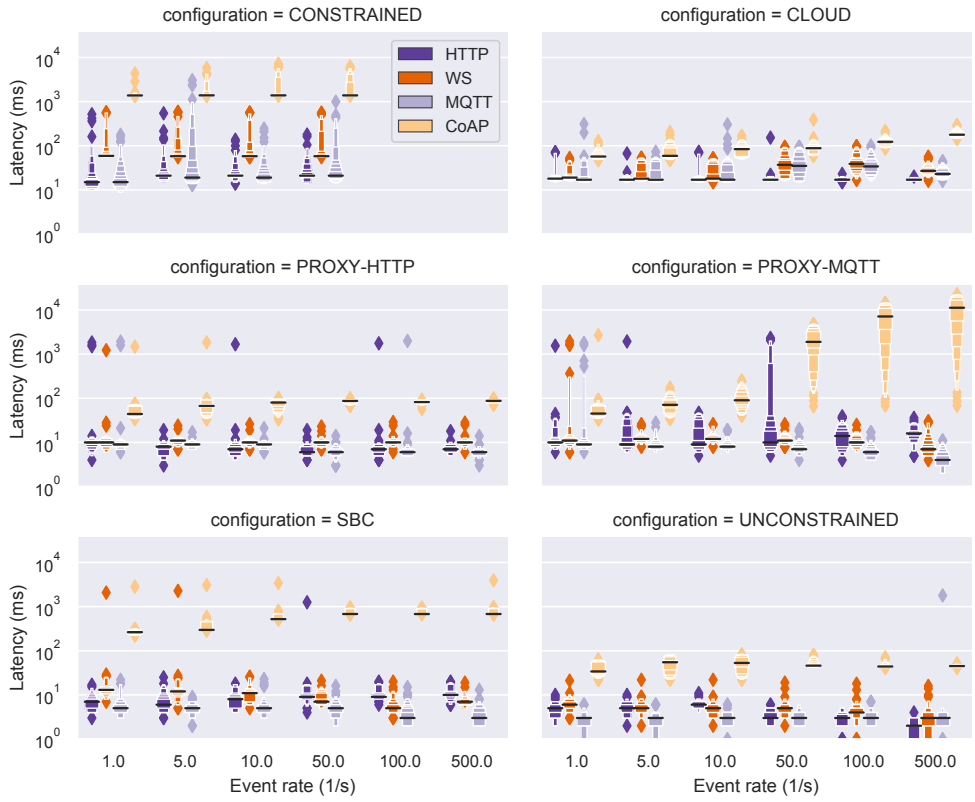


Figure 7: Event latency (1000 total emissions)

there may be bindings with lower data transfer volumes but more errors).

We can observe that the proxy pattern helps in improving Websockets and HTTP performance while allowing the constrained device to restrict itself to the lighter bindings—Websocket latencies (see figure 9) and HTTP errors (see table 6) are lower in the proxy scenario configurations when compared to the *SBC* configuration.

Figures 6 and 10 show that the Websockets data transfer volumes decrease as conditions get more demanding when compared with the same total number of invocations or requests under undemanding conditions. This is explained by the optimizations included to reuse previously opened connections in the client—lower rates cause the binding client to open and close the connection on each request or invocation (i.e. the invocation completes before the next one is initiated, hence the connection cannot be reused). The same is not clearly observed in figure 8 due to the fact that a single connection is used to receive the complete stream of events in every case (i.e. the binding only needs to open one connection to receive the stream of events, independently of the event rate).

Another effect of the connection reuse optimization is shown in figures 5 and 9 where latencies for Websockets are actually lower for more demanding conditions in some instances (e.g. in figure 5 the median of the action invocation latency in the *unconstrained* configuration is $\sim 10^3$ ms for batches of 1 parallel invocation while it is $\sim 10^2$ ms for 5 and 10 parallel invocations).

The **Websockets** binding is a good fit for interaction verbs that benefit from server-initiated messaging (i.e. action invocation, property observe, event subscribe) due to reduced latency and data transfer volumes (figures 5, 6, 7, 8). It is not suitable in scenarios where a high sustained request throughput (property read, property write) is required, especially for constrained devices (figure 9).

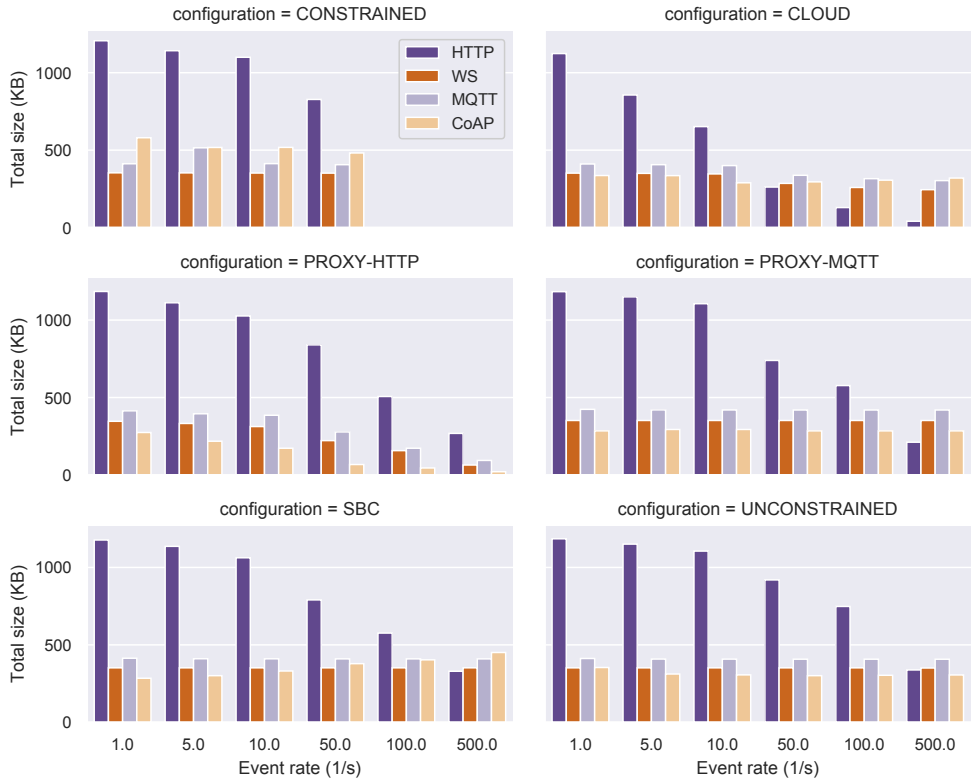


Figure 8: Event data transfer volumes (1000 total emissions)

5.3. MQTT

The quality of the connection between the broker and the MQTT client has notable impact on action and property latency as seen in the *cloud* configuration (figures 5 and 9). In this case the client has to establish additional connections over the Internet with the cloud instance that contains the broker (located in the same network as the servient containing the WoT server application). The impact on event latency is not significant (figure 7) as the connection is only established once.

The positive performance effects of reusing connections (discussed in 5.2) also apply to the MQTT binding (i.e. lower data volumes and latencies for more demanding conditions as seen in figures 5, 6, 9 and 10).

The **MQTT** binding appears to be a versatile option with good performance on all interaction verbs. It does not have any notable drawbacks as long as the MQTT broker is located close to the client (i.e. low latency connection). Data volumes are higher than the Websockets and CoAP bindings in some cases (figures 6 and 10)—less reliable QoS levels could be configured if needed to decrease data usage.

5.4. CoAP

CoAP consistently generates the lowest data volumes for most instances of undemanding conditions. On the other hand, CoAP transfer volumes tend to increase dramatically in demanding conditions due to UDP datagram loss and the overhead that results from message retransmissions and duplicates (e.g. see the *SBC* configuration in figure 10 where error ratios for request rates ≥ 50 requests/s are ≥ 0.39 as seen in table 6). It also displays lower latency on property reads and action invocations when communicating over the Internet (*cloud* configuration) when compared with the other bindings.

There is a difference of one order of magnitude in CoAP event latency between high and low target event rates in

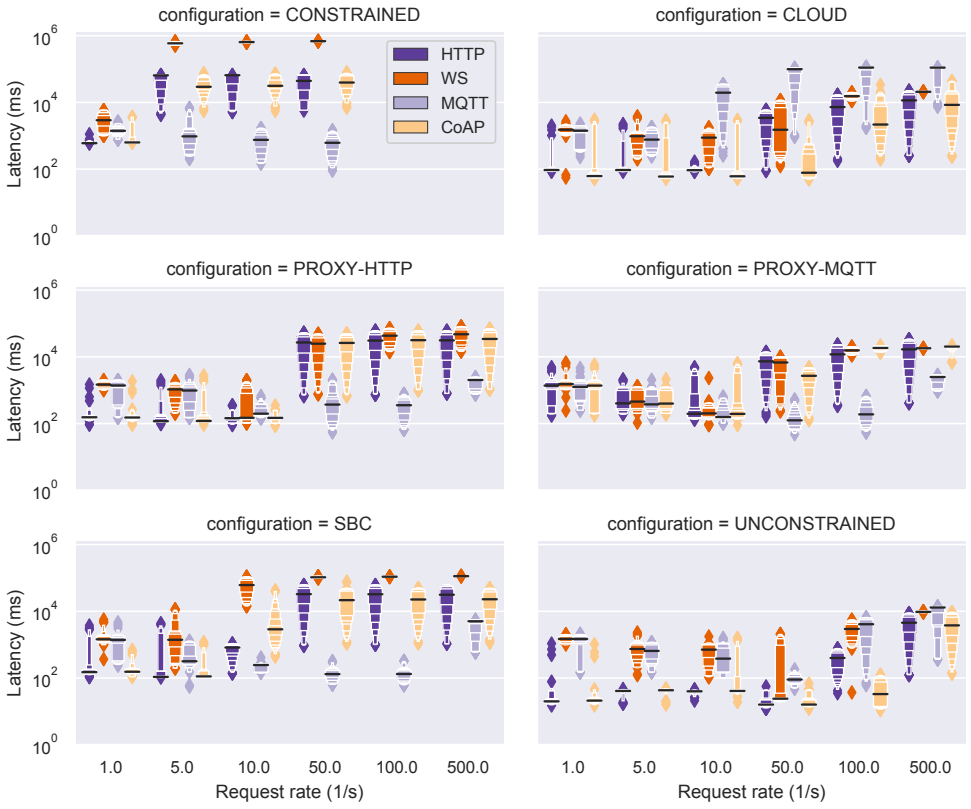


Figure 9: Property read latency (1000 total requests)

the *proxy-mqtt* configuration (figure 7). This can be explained by the CoAP binding in the WoT proxy struggling to keep up with the higher performance of the MQTT binding in the WoT server (i.e. the MQTT binding is able to emit events at a higher rate than the CoAP binding). The event pipeline of the CoAP binding server introduces a significant amount of latency, and this, combined with the fact that events are emitted sequentially, means that all events received by the proxy MQTT client and re-emitted by the proxy CoAP server get delayed. The effect of this delay accumulates for higher target event rates and is the main cause of the latencies.

Similar performance problems to those observed on HTTP for high rates of requests per second also appear on the CoAP binding (table 6). A closer inspection of the logs reveals that CoAP has problems when dealing with retransmissions under load. It should be noted that CoAP has an upper limit for the message rate depending on the configuration parameters before messages start being incorrectly interpreted as duplicates due to reused message IDs [50].

The **CoAP** binding is especially adequate in undemanding conditions when low data transfer volumes are a priority, as could be expected due to its UDP transport (figures 6, 8, 10). It also shows good performance for demanding conditions on capable hardware. Interaction verbs based on subscriptions (property observe and event subscribe) present a higher latency than their counterparts, but have reasonable performance nonetheless (figure 7). Action invocation and property read/write errors are significant on constrained hardware when this binding is exposed to higher throughputs (tables 4 and 6).

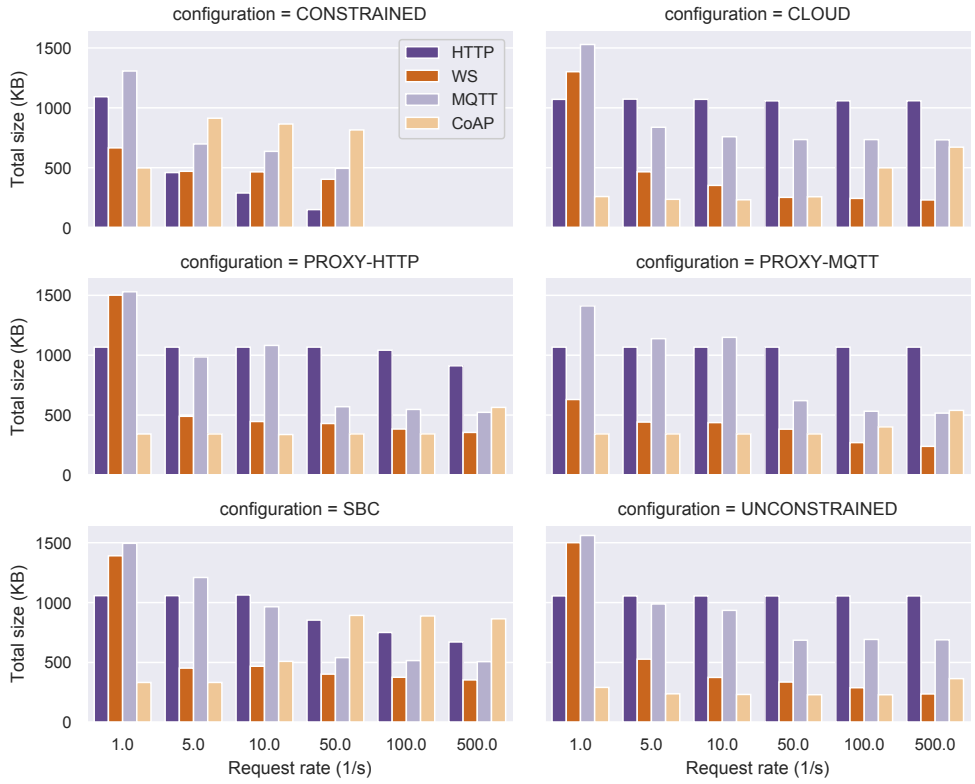


Figure 10: Property read data transfer volumes (1000 total requests)

6. Conclusions and future work

The WoT paradigm can be applied to ensure the interoperability of IoT devices. In this study, an experimental design and implementation in Python of a framework based on the W3C WoT building blocks was described. This framework provides a full-featured WoT runtime with support for all interaction verbs on all protocol binding implementations—the first to do so, to the best of our knowledge.

A set of WoT applications were built on top of the framework and combined into multiple common configurations. Devices from all layers of the fog computing stack can be integrated into the Web using the framework, as demonstrated by the multiple hardware platforms included in the experiment configurations. Performance results for the framework protocol bindings were obtained and analyzed to identify the most adequate protocols for each scenario and type of interaction. Protocols native to the browser (i.e. HTTP and Websockets) show good performance, although they could be considered heavy for use on constrained devices. These two protocols complement each other, as Websockets is more adequate for interaction verbs that benefit from server-initiated messaging. This is a weakness of HTTP. On the other hand, MQTT is a good all-around option (with the main drawback of requiring a broker), while CoAP works best to ensure low data transfer usages.

As a final note, we would like to highlight the decrease in development costs due to the adoption of the W3C WoT building blocks and the framework under discussion—a complete proxy application supporting all protocols and interaction types was programmed in under 200 lines of code (including boilerplate specific for the experiments), while the server application had an even smaller footprint.

Future work will focus on researching improved discovery mechanisms for the automatic generation of WoT

	Request rate (1/s)					
	1.0	5.0	10.0	50.0	100.0	500.0
CONSTRAINED						
HTTP	0.000	0.570	0.730	0.860	-	-
CoAP	0.077	0.647	0.794	0.887	-	-
CLOUD						
CoAP	0.000	0.000	0.000	0.000	0.000	0.002
PROXY-HTTP						
HTTP	0.000	0.000	0.000	0.000	0.024	0.146
SBC						
HTTP	0.000	0.000	0.000	0.200	0.300	0.370
CoAP	0.000	0.000	0.001	0.397	0.470	0.562

Table 6: Property read error (1000 total requests)

avatars—virtual representations of Things that provide a view with some added value, for example an aggregation of multiple Things that is automatically generated upon discovery. Additional protocols, such as AMQP, XMPP, OPC, will also be considered to extend the protocol binding layer.

Furthermore, the performance data obtained in this work will serve as the basis for a tool to simulate and optimize IoT application deployments based on the fog computing model and the W3C WoT architecture. This would build on the foundation laid by works such as [51].

7. Acknowledgements

This research has been partially funded by the Spanish National Plan of Research, Development and Innovation under the project OCAS (RTI2018-094849-B-I00); the European Regional Development Fund; and the Principality of Asturias Plan of Science, Technology and Innovation under the project WORMS (IDI/2018/000101).

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications, IEEE Communications Surveys Tutorials 17 (4) (Fourthquarter 2015) 2347–2376. doi:10.1109/COMST.2015.2444095.

[2] D. Raggett, The Web of Things: Challenges and Opportunities, Computer 48 (5) (2015) 26–32. doi:10.1109/MC.2015.149.

[3] J. Mineraud, O. Mazhelis, X. Su, S. Tarkoma, A gap analysis of Internet-of-Things platforms, Computer Communications 89-90 (2016) 5–16. doi:10.1016/j.comcom.2016.03.015.

[4] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog Computing and Its Role in the Internet of Things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16. doi:10.1145/2342509.2342513.

[5] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, X. Yang, A Survey on the Edge Computing for the Internet of Things, IEEE Access 6 (2018) 6900–6919. doi:10.1109/ACCESS.2017.2778504.

[6] M. Kovatsch, K. Kajimoto, R. Matsukura, M. Lagally, T. Kawaguchi, Web of Things (WoT) Architecture, Tech. Rep. W3C Editor's Draft 25 February 2019, W3C (Feb. 2019).
URL https://w3c.github.io/wot-architecture/

[7] R. T. Fielding, REST: Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, Irvine (2000).
URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[8] Open Connectivity Foundation, OCF Core Specification, Tech. Rep. Version 2.0.1, Open Connectivity Foundation (Feb. 2019).
URL https://openconnectivity.org/specs/OCF_Core_Specification_v2.0.1.pdf

[9] Open Mobile Alliance, Lightweight Machine to Machine Technical Specification, Tech. Rep. Approved Version 1.0.2, Open Mobile Alliance (Feb. 2018).
URL http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/OMA-TS-LightweightM2M-V1_0_2-20180209-A.pdf

[10] oneM2M, oneM2M Functional Architecture, Tech. Rep. TS 118 101 V2.10.0 (2016-10), ETSI (Oct. 2016).
URL http://www.etsi.org/deliver/etsi_ts/118100_118199/118101/02.10.00_60/ts_118101v021000p.pdf

[11] T. Sánchez López, D. C. Ranasinghe, M. Harrison, D. McFarlane, Adding sense to the Internet of Things, Personal and Ubiquitous Computing 16 (3) (2012) 291–308. doi:10.1007/s00779-011-0399-8.

[12] C. Kamienski, M. Jentsch, M. Eisenhauer, J. Kiljander, E. Ferrera, P. Rosengren, J. Thestrup, E. Souto, W. S. Andrade, D. Sadok, Application development for the Internet of Things: A context-aware mixed criticality systems development platform, Computer Communications 104 (2017) 1–16. doi:10.1016/j.comcom.2016.09.014.

[13] D. Guinard, V. Trifa, E. Wilde, A resource oriented architecture for the Web of Things, in: 2010 Internet of Things (IOT), 2010, pp. 1–8. doi:10.1109/IOT.2010.5678452.

[14] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, Web of Things (WoT) Thing Description, Tech. Rep. W3C Editor's Draft 24 February 2019, W3C (Feb. 2019).
URL https://w3c.github.io/wot-thing-description/

[15] M. Koster, Web of Things (WoT) Protocol Binding Templates, Tech. Rep. W3C Editor's Draft 21 January 2019, W3C (Jan. 2019).
URL https://w3c.github.io/wot-binding-templates/

[16] Z. Kis, K. Nimura, D. Peintner, J. Hund, Web of Things (WoT) Scripting API, Tech. Rep. W3C Editor's Draft 18 February 2019, W3C (Feb. 2019).
URL https://w3c.github.io/wot-scripting-api/

[17] D. Guinard, V. Trifa, D. Carrera, Web Thing Model, W3C Member Submission (Apr. 2017).
URL http://model.webofthings.io/

[18] B. Francis, Web Thing API, Unofficial Draft, Mozilla Corporation (Jan. 2019).
URL https://iot.mozilla.org/wot/

[19] F. Paganelli, S. Turchi, D. Giuli, A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City, IEEE Systems Journal 10 (4) (2016) 1412–1423. doi:10.1109/JSYST.2014.2354835.

[20] S. Liang, C.-Y. Huang, T. Khalafbeigi, OGC SensorThings API Part 1: Sensing, OGC Implementation Standard 15-078r6, Open Geospatial Consortium (Jul. 2016).
URL http://docs.opengeospatial.org/is/15-078r6/15-078r6.html

[21] S. Liang, T. Khalafbeigi, OGC SensorThings API Part 2 – Tasking Core, OGC Implementation Standard 17-079r1, Open Geospatial Consortium (Jan. 2019).
URL http://docs.opengeospatial.org/is/17-079r1/

17-079r1.html
[22] J. M. Cantera Fonseca, F. Galán Márquez, T. Jacobs, FIWARE-NGSI v2 Specification, Tech. Rep. v2.0, FIWARE Foundation (2018).
URL http://fiware.github.io/specifications/ngsiv2/stable/
[23] ETSI ISG CIM, Context Information Management (CIM) NGSI-LD API, Tech. Rep. GS CIM 009 V1.1.1 (2019-01), ETSI (Jan. 2019).
URL https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.01.01_60/gs_CIM009v010101p.pdf
[24] M. Noura, S. Heil, M. Gaedke, Webifying Heterogenous Internet of Things Devices, in: M. Bakaev, F. Frasincar, I.-Y. Ko (Eds.), Web Engineering, Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 509–513. doi:10.1007/978-3-030-19274-7_36.
[25] Eclipse Thingweb, Node-wot (0.6.1) (Apr. 2019).
URL https://github.com/eclipse/thingweb.node-wot
[26] T. Sakamoto, H. Ito, K. Nimura, Dynamically Exposing and Controlling Physical Devices by Expanding Web of Things Scheme, in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 2, 2017, pp. 329–335. doi:10.1109/COMPSAC.2017.114.
[27] M. Noura, A. Gyrard, S. Heil, M. Gaedke, Automatic Knowledge Extraction to build Semantic Web of Things Applications, IEEE Internet of Things Journal (2019) 1–1doi:10.1109/JIOT.2019.2918327.
[28] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, P. Dutta, The Internet of Things Has a Gateway Problem, in: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile '15, ACM, New York, NY, USA, 2015, pp. 27–32. doi:10.1145/2699343.2699344.
[29] B. Negash, T. Westerlund, H. Tenhunen, Towards an interoperable Internet of Things through a web of virtual things at the Fog layer, Future Generation Computer Systems 91 (2019) 96–107. doi:10.1016/j.future.2018.07.053.
[30] S. S. Adhatarao, M. Arumaithurai, X. Fu, FOGG: A Fog Computing Based Gateway to Integrate Sensor Networks to Internet, in: 2017 29th International Teletraffic Congress (ITC 29), Vol. 2, 2017, pp. 42–47. doi:10.23919/ITC.2017.8065709.
[31] C. H. Chen, M. Y. Lin, C. C. Liu, Edge Computing Gateway of the Industrial Internet of Things Using Multiple Collaborative Microcontrollers, IEEE Network 32 (1) (2018) 24–32. doi:10.1109/MNET.2018.1700146.
[32] M. Hemmatpour, M. Ghazivakili, B. Montrucchio, M. Rebaudengo, DIIG: A Distributed Industrial IoT Gateway, in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, 2017, pp. 755–759. doi:10.1109/COMPSAC.2017.110.
[33] B. Kang, D. Kim, H. Choo, Internet of Everything: A Large-Scale Autonomic IoT Gateway, IEEE Transactions on Multi-Scale Computing Systems 3 (3) (2017) 206–214. doi:10.1109/TMSCS.2017.2705683.
[34] F. Banaie, J. Misic, V. B. Misic, M. H. Yaghmaee, S. A. Hosseini, Performance Analysis of Multithreaded IoT Gateway, IEEE Internet of Things Journal (2018) 1–1doi:10.1109/JIOT.2018.2879467.
[35] F. Van den Abeele, I. Moerman, P. Demeester, J. Hoebeke, Secure Service Proxy: A CoAP(s) Intermediary for a Securer and Smarter Web of Things, Sensors 17 (7) (2017) 1609. doi:10.3390/s17071609.
[36] S. S. Mathew, Y. Atif, Q. Z. Sheng, Z. Maamar, Building sustainable parking lots with the Web of Things, Personal and Ubiquitous Computing 18 (4) (2014) 895–907. doi:10.1007/s00779-013-0694-7.
[37] D. Mazzei, G. Baldi, G. Montelisciani, G. Fantoni, A full stack for quick prototyping of IoT solutions, Annals of Telecommunications 73 (7) (2018) 439–449. doi:10.1007/s12243-018-0644-5.
[38] O. Debauche, M. E. Moulat, S. Mahmoudi, S. Boukraa, P. Manneback, F. Lebeau, Web Monitoring of Bee Health for Researchers and Beekeepers Based on the Internet of Things, Procedia Computer Science 130 (2018) 991–998. doi:10.1016/j.procs.2018.04.103.
[39] S. Cheshire, M. Krochmal, DNS-Based Service Discovery, RFC 6763, RFC Editor, http://www.rfc-editor.org/rfc/rfc6763.txt (Feb. 2013).
URL http://www.rfc-editor.org/rfc/rfc6763.txt
[40] S. Cheshire, M. Krochmal, Multicast DNS, RFC 6762, RFC Editor, http://www.rfc-editor.org/rfc/rfc6762.txt (Feb. 2013).
URL http://www.rfc-editor.org/rfc/rfc6762.txt
[41] K. Lei, Y. Ma, Z. Tan, Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js, in: 2014 IEEE 17th International Conference on Computational Science and Engineering, 2014, pp. 661–668. doi:10.1109/CSE.2014.142.
[42] J. Heuer, J. Hund, O. Pfaff, Toward the Web of Things: Applying Web Technologies to the Physical World, Computer 48 (5) (2015) 34–42. doi:10.1109/MC.2015.152.
[43] JSON-RPC Working Group, JSON-RPC 2.0 Specification, Tech. Rep. Version 2.0 (Jan. 2013).
URL https://www.jsonrpc.org/specification
[44] A. Garcia Mangas, WoTPy: A framework for Web of Things applications in Python (Apr. 2019). doi:10.5281/zenodo.2605071.
URL https://doi.org/10.5281/zenodo.2605071
[45] Facebook, Tornado Web Server (Sep. 2018).
URL http://www.tornadoweb.org/
[46] C. Amsüss, M. Wasilak, Aiocoap: Python CoAP Library, Energy Harvesting Solutions (2013).
URL http://github.com/chrysn/aiocoap/
[47] N. Jouanin, HBMQTT (Nov. 2018).
URL https://github.com/beerfactory/hbmqtt
[48] P. Scott-Murphy, W. McBrine, J. Stasiak, Python-zeroconf (Sep. 2018).
URL https://github.com/jstasiak/python-zeroconf
[49] H. Hofmann, K. Kafadar, H. Wickham, Letter-value plots: Boxplots for large data, Tech. rep., had.co.nz (2011).
[50] Z. Shelby, K. Hartke, C. Bormann, The Constrained Application Protocol (CoAP), Tech. Rep. RFC7252, RFC Editor (Jun. 2014). doi:10.17487/rfc7252.
URL https://www.rfc-editor.org/info/rfc7252
[51] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, R. Buyya, iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments, Software: Practice and Experience 47 (9) (2017) 1275–1296. doi:10.1002/spe.2509.

Apéndice B

Publicación emulador WoTemu

Este apéndice incluye una publicación de impacto derivada de los trabajos descritos en la sección 4. El artículo a continuación se presenta en su versión *accepted manuscript*, que no incluye los trabajos de formato final del editor. La referencia completa de este artículo se muestra a continuación:

Referencia de la publicación derivada del emulador WoT

García Mangas, A., Suárez Alonso, F. J., García Martínez, D. F., & Díez Díaz, F. (2022). WoTemu: An emulation framework for edge computing architectures based on the Web of Things. Computer Networks, 209, 108868. `https://doi.org/10.1016/j.comnet.2022.108868`

WoTemu: An Emulation Framework for Edge Computing Architectures based on the Web of Things

Andrés García Mangas[a,*], Francisco José Suárez Alonso[b], Daniel Fernando García Martínez[b], Fidel Díez Díaz[a]

[a] *Technological Center for Information and Communication (CTIC), W3C Spain Office host, Ada Byron 39, 33203, Gijón, Spain*
[b] *Department of Computer Science, University of Oviedo, Spain*

Abstract

The edge computing model is an approach to Internet of Things (IoT) architectures based on the redistribution of services and infrastructure from centralized clouds to locations closer to IoT devices. The Web of Things (WoT) is another important IoT trend, currently led by the W3C, which aims at solving the IoT interoperability problem by adopting proven technologies and patterns from the Web. The design and validation of IoT deployments based on these paradigms is a complex task that involves multiple services, heterogeneous hardware and diverse communication technologies. Testing such projects in real world conditions usually requires a significant investment of resources. There are simulation tools that can assist in this process with much lower barriers of entry, however, they involve the designer making modelling assumptions that are not always representative of the real systems. This work presents an emulation tool for IoT projects based on the edge computing model that is able to seamlessly scale horizontally by leveraging container orchestration (Docker swarm mode). Furthermore, the W3C WoT model is included as a first-class citizen, enabling the designer to model all actors in the system as Things. The tool can run the real production code with minimal modifications and provides meaningful insights into the behaviour of the proposed architecture. This knowledge serves to rapidly iterate the optimization process, simplifying design issues and the detection of bottlenecks before committing to a real deployment in the field. A real-world scenario is also emulated in order to demonstrate its capabilities and validate its contribution.

Keywords: Web of Things, Internet of Things, edge computing, container orchestration

1. Introduction

The concept of fog computing was first introduced by Flavio Bonomi *et al* [1] as a platform which exposes services that had previously been considered native to the cloud (i.e. networking, computing, storage) in any layer between the IoT data sources and the core data centers. The distinction between *edge computing* and *fog computing* is sometimes vague. Fog computing can be considered as the prime example and even an evolution of the more general edge computing concept [2]. It implements the core idea of edge computing, that is, the redistribution of resources from the cloud to locations adjacent to the IoT end devices. However, it is not constrained to the edge and can exist in the upper layers that are closer to the cloud. In this work, we refer to *edge computing* as a more general term, which includes *fog computing*.

Adoption of the edge computing architectural model can provide solutions to transmission, computation and storage requirements [3] that the cloud computing model is unable to handle. Examples include adapting to computational loads that evolve due to the geographic mobility of end devices [4], and ensuring consistently low latency [5].

The rapid growth of the IoT has led to a high degree of heterogeneity. Many different systems and platforms can be found, providing different implementations for similar challenges [6]. This can cause interoperability problems and data silos: isolated datasets that may become costly barriers for integration into other systems or applications. The Web of Things is a reasonably recent paradigm [7] which aims to meet the IoT interoperability challenge by adopting proven patterns and technologies from the Web.

In one of the earliest works on the WoT [8], Guinard *et al* proposed the integration of physical Things into the Web by exposing an HTTP API based on the REST architectural model. This work is one of the foundations of the W3C WoT, which is currently the leading implementation of the WoT paradigm. The W3C WoT reference architecture [9] revolves around the concept of the Thing—any virtual or physical entity that is described by a Thing Description (TD) [10]. A TD is a machine-readable document commonly serialized in JSON-LD [11] that contains Thing metadata. All functionalities exposed by the Thing are modelled in the TD as one of three types of interac-

*Corresponding author
Email addresses: andres.garcia@fundacionctic.org (Andrés García Mangas), fjsuarez@uniovi.es (Francisco José Suárez Alonso), dfgarcia@uniovi.es (Daniel Fernando García Martínez), fidel.diez@fundacionctic.org (Fidel Díez Díaz)

tions: properties, actions and events, as defined by the WoT Interaction model. High-level interactions are then mapped to specific platforms by WoT Binding Templates [12], e.g. a template could map property write operations to HTTP POST requests. Finally, the Scripting API specification [13] is an optional but important building block that defines the public programming interface exposed by compatible WoT runtimes. This API enables developers to quickly expose, consume and discover Things while abstracting from the implementation details.

Containers are a ubiquitous presence in the cloud computing space. This technology leverages Linux kernel primitives (e.g. *cgroups*) to enable isolated, portable execution of processes. Docker [14] is the most relevant container product, providing an entire stack of components including container runtimes and image repositories. It should be noted, however, that there are alternatives to individual pieces in the Docker ecosystem, for example Red Hat *crun* [15] is an alternative container runtime to *runc*. Moreover, container orchestration tools, such as Kubernetes [16] or Docker Swarm mode, enable the management of container-based applications across multiple hosts. Given the dynamic nature of edge computing and the similarities with cloud computing, container technologies can be of benefit for deployment, development and maintenance tasks [17].

1.1. Background

This section describes a set of relevant simulation and emulation tools in the context of cloud computing and the Internet of Things. Table 1 contains a summary of references to the tools. The tools have been categorized depending on their ability to run real code and scale horizontally. More specifically, horizontal scalability is the ability of a system to increase its performance and capacity by adding more nodes to the system. Vertical scaling is based on upgrading the components of each node (i.e. CPU, GPU, memory, storage). Horizontal scalability tends to be a more optimal strategy in this context, as vertical scaling has significant cost barriers in the majority of cases. Furthermore, container orchestration tools contribute significantly to achieving seamless horizontal scalability, which is one of the main rationales for using container orchestration as the foundation of WoTemu.

CloudSim [18] is a widely referenced software framework that allows for the modelling of systems based on cloud computing entities, e.g. datacenters, virtual machines, using a programmatic interface. These systems can be used to represent a wide variety of cloud computing scenarios and simulated in a repeatable fashion. ContainerCloudSim [19] is a built-in module that augments CloudSim to include the concept of application containers, enabling researchers to define strategies for container allocation on virtual machines and model cloud architectures in terms of Containers as a Service (CaaS).

CloudSim is the cornerstone for a range of simulation tools that provide extended functionality for other domains. The iFogSim project [20] is one of the most relevant examples in this group. IoT applications in iFogSim are represented as directed graphs based on the Distributed Dataflow (DDF) model [28]. Vertices in these graphs correspond to three distinct types of entities: *sensors*, which generate messages following a predefined transmission distribution (e.g. deterministic, normal); *actuators*, which act as the receiving end of messages; and *modules*, a class meant to symbolize any algorithm or stage in an IoT processing pipeline (e.g. a classification algorithm). Messages exchanged by the previous entities are modelled as edges, and are parameterized by their computation and network costs. *Application loops* represent the flow of messages in the IoT application graph and must be explicitly initialized to indicate the graph paths to monitor. *Modules* are then allocated on *fog devices*, which are described in terms of CPU resources, available memory and power usage. Users have the option of setting the *module* placement or using an automated strategy.

The authors in [21] introduce CloudSim Plus as an alternative fork of CloudSim with the aim of improving code quality, adopting software engineering best practices and including exclusive features. These efforts have provided several advantages, such as a simplified programming interface that requires less code for comparable CloudSim examples; an updated hierarchy of classes that improves extensibility and facilitates the implementation of ad hoc algorithms; and the ability to define vertical and horizontal autoscaling strategies for virtual machines.

Unlike the previous references, YAFS [22] is an edge computing simulator built from the ground up, outside of the CloudSim ecosystem. It is, however, fairly similar to iFogSim in its modelling approach—applications are represented as DDF graphs, and nodes are characterized in terms of computation, memory and cost. One of its most significant contributions is its ability to define ad-hoc algorithms that run in parallel with the simulation. This enables the designer to dynamically update the system in response to simulation events, for example to deploy new actuators.

Mininet [23] is a relevant project in the Software-Defined Networks (SDN) field that is closely related to edge computing emulation. It leverages features native to Linux, such as network namespaces, to build a framework that is capable of emulating networks with hundreds of entities (e.g. virtual hosts, switches and links) in a single host. One of its main limitations—scaling to multiple hosts—is addressed by MaxiNet [24]. To this end, MaxiNet uses an ad-hoc cluster manager and the Generic Routing Encapsulation (GRE) protocol to build IP-in-IP tunnels that interconnect Mininet workers. In addition, it has support for Docker containers. However, complex scenarios tend to require manual container placement, as the automated placement algorithm is rather limited. In a related note, Docker support can also be enabled for Mininet experiments thanks to the contribution of the Containernet [25] fork.

Building upon MaxiNet, EmuFog [26] provides capabil-

Tool	Highlights	Real code	Horizontal scalability
CloudSim [18]	Widely extended cloud computing simulator.	No	*Not applicable*
ContainerCloudSim [19]	CloudSim extension that enables simulation of containers.	No	*Not applicable*
iFogSim [20]	CloudSim-based IoT fog computing simulator.	No	*Not applicable*
CloudSim Plus [21]	CloudSim fork with additional features and a focus on software engineering best practices.	No	*Not applicable*
YAFS [22]	Edge computing simulator able to dynamically react to events during the simulation.	No	*Not applicable*
Mininet [23]	Network emulator for the creation of network testbeds consisting of virtual hosts, switches and links.	Yes	No
MaxiNet [24]	Enables Mininet to run across multiple hosts and adds Docker-based virtual hosts.	Yes	Manual container placement in non-trivial cases.
Containernet [25]	Mininet fork that adds Docker-based virtual hosts.	Yes	No
EmuFog [26]	Tool to optimize edge computing architectures (in the form of MaxiNet experiments) based on a series of constraints.	Yes	Same limitations as MaxiNet
Fogbed [27]	Emulation of edge computing topologies based on Containernet and MaxiNet.	Yes	Same limitations as MaxiNet

Table 1: Summary of related simulation and emulation tools.

ities for efficient design of edge computing topologies. It enables users to define a set of end clients (*device nodes*) and a catalog of edge computing devices (*fog nodes*) in terms of CPU, memory and latency cost. As imput this tool uses a network topology in BRITE [29] format. It then performs an optimization process to arrive at a MaxiNet experiment comprised of a *fog node* selection from the previous catalogue that fulfills the user constraints. Although it offers support for the execution of Docker containers, it is based on MaxiNet, and thus shares its limitations.

In a similar fashion to EmuFog, Fogbed [27] leverages Containernet and MaxiNet to offer both local (i.e. Mininet) and distributed (i.e. MaxiNet) emulations of Docker-based edge computing network topologies. An interesting feature of Fogbed is its ability to dynamically update the topology of a running experiment, for example to add a new host.

All of the software utilities described above have proved to be of great help in the design process of IoT applications. For instance, the authors in [30] present an optimized remote pain monitoring IoT architecture based on the simulation results provided by iFogSim.

To date, research on design tools for edge and cloud computing architectures has mostly focused on a more theoretical and high-level view. Users are required to settle on a set of parameters and models to characterize their scenarios as accurately as possible. This entails significant difficulties and compromises during the design process. It is reasonable to argue that lowering the barrier of entry may drive the adoption of design tools in this domain. This would in turn serve to optimize resources and avoid costly deployment issues that could have been detected in an earlier stage.

To the best of our knowledge, all of the studied tools, which aim at emulating real code, struggle to some degree with horizontal scaling. Furthermore, they abstract from the full scope of application monitoring, leaving significant time-consuming responsibilities, such as identifying the flow of traffic between services, to the user.

1.2. Motivation

Considering these limitations, this work proposes an application-centric emulation framework based on container orchestration to help in the design of IoT/WoT systems that follow the edge computing architectural model. The motivation is detailed in the following list.

- To bridge the gap between developers, who may be more inclined to test IoT architectures using real production code, and the field of theoretical simulation tools.
- To provide, in addition to high-level host measurements, performance metrics that are focused on the application itself and therefore more meaningful to

developers. For instance, network traffic for each application component broken down by protocol would be of great use.

- To lever the capabilities of a modern container orchestration utility, such as Docker swarm mode, for emulation of IoT architectures with seamless horizontal scalability. This approach offers a sensible compromise between the realism of hardware testbeds and the experiment scale that is achievable with theoretical simulators.
- To make the WoT paradigm a first-class citizen in the emulation scenarios by enabling users to represent all actors in the system as WoT Things.

2. Design

This section presents the architecture of experiments in WoTemu and explains the rationale behind the design choices. The main types of entities in WoTemu are identified and defined, and extended subsections for the most relevant modules in the framework are also included.

A fully functional implementation of WoTemu is publicly available in both Zenodo [31] and GitHub[1] under the MIT licence. The minimum requirements of this implementation are Python 3.6, Docker Compose 1.27.0 and Docker Engine 20.10.0.

WoTemu is based on Docker swarm mode, a container orchestration tool included in the Docker ecosystem. It should be noted that Kubernetes, which is arguably the most popular orchestration tool, was also considered. However, the following reasons led to the adoption of containers in general and swarm mode in particular as the foundation of WoTemu.

- Docker is a widely extended containerization platform that is easy to install and is readily available in multiple platforms. Swarm mode is the default built-in orchestration tool in Docker. It is also easy to configure, taking a few minutes at most to setup a cluster with multiple machines.
- Orchestration tools are characterized by straightforward horizontal scalability, which is a basic requirement to enable the emulation of experiments with a high number of entities, but which is lacking in tools in the current state of the art.
- Emulation of constrained platforms is reasonably simple using containers. Resources such as CPU and memory can be configured on a container-by-container basis.
- The life cycle of experiments is simplified by using containers. Existing resources (e.g. volumes, networks) can be simply removed from the hosts without leaving traces.
- The default *overlay* network stack implementation included in swarm mode is a great fit to represent isolated networks and connections between entities in an edge computing scenario.

Therefore, WoTemu is intrinsically linked to Docker swarm mode concepts. Throughout this paper there are references to swarm *services*, *tasks* and *networks*; these concepts are clarified below.

- A swarm *task* is linked to a specific container and is the atomic execution unit in a swarm (i.e. there is a container for each *task*). If a task fails for any reason (i.e. the container exits with a non-zero code) the swarm manager will attempt to reload it to maintain the desired swarm state.
- A swarm *service* is a template for *tasks*, describing, for example, the base image or the connected *networks*. A *service* may have multiple replicas, each one being a different *task*. Requests going into a *service* are balanced between all the replicas.
- A swarm *network* interconnects different *services*. *Services* in the same *network* are able to communicate with each other. It is important to note that a *network* is not limited to *services* in a single physical swarm node—Docker provides the overlay network driver to enable distributed networks.

Figure 1 shows a general view of the WoTemu architectural model, including the different types of containers, which are described in more detail in the following paragraphs.

Node Node containers, also referenced to as *applications*, represent programs that must be monitored during the execution of an experiment to be analysed later. From a development standpoint, the application executed by a node takes the form of a Python file exposing an asynchronous function, which takes the WoT runtime entrypoint, configuration and event loop instance as arguments. WoTemu is tailored for WoT applications, that is, programs that use the WoT runtime provided in WoTPy [32] to perform their operations in terms of exposing and consuming Things. However, WoTemu has no restrictions on what is executed in the application function, except that programs must follow the asynchronous I/O programming model of asyncio. Therefore, applications can act as emulated IoT devices, control programs for real IoT devices, processing pipelines, or any other element required in the scenario. Furthermore, WoTemu includes a series of *built-in apps*.

[1] https://github.com/agmangas/wotemu

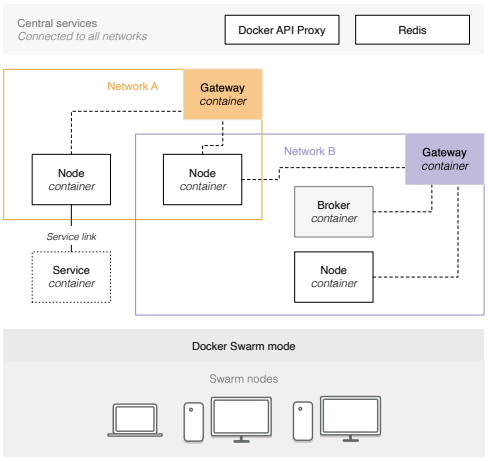


Figure 1: Architectural model in WoTemu.

These are configurable applications which can serve as mocks and placeholders for testing purposes.

Gateway There is one gateway container for each network in the experiment. These containers are created automatically and injected transparently into the communications of node containers, acting as middlemen to emulate real world network conditions. See Section 2.1 for further information.

Service This type of container represents any service or dependency that may be required by an application. Possible examples include databases, message queues or identity and access management. There are two main differences from other types of containers. The first is that no background monitoring processes run in service containers. Secondly, that services exist in their own isolated networks, and node containers must declare an explicit *service link* in order to communicate with them.

Broker Unlike the other Protocol Binding implementations that can operate in a self-contained fashion, the MQTT implementation requires an external MQTT broker component. Therefore, broker containers are treated as first-class citizens in WoTemu experiments, including the usage of background monitoring processes. The implementation is based on the widely-used, open-source Eclipse Mosquitto [33].

There are also two central services in all WoTemu experiments which are connected to all the networks in the topology:

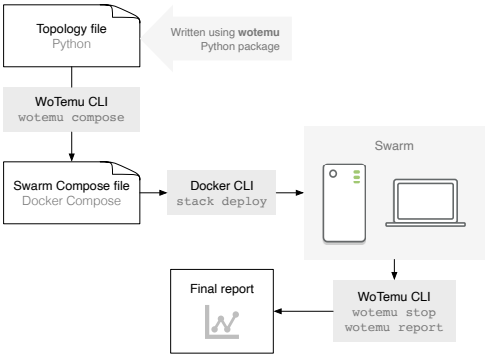


Figure 2: Typical workflow for an experiment in WoTemu.

Docker API proxy This container is an instance of the *tecnativa/docker-socket-proxy* [34] image. It serves as a gateway for other containers in the experiment to access the Docker API of the swarm through the internal overlay networks. The Docker API is a program interface to manage all entities in a Docker environment. This elevated access is required to provide containers with introspection capabilities so they can self-configure independently from the others. Section 2.1 contains further discussion on this design decision.

Redis This is an open-source, in-memory key-value data store. It is used to save historical data and measurements necessary to characterize and analyse the behaviour of the experiment. Using Redis [35] as opposed to a relational database results in lower latencies for read/write operations at the cost of an increased memory footprint. It is especially indicated for this case, as it ensures that the background monitor processes running on all containers have a small performance impact. Section 2.2 contains more information on the data that is persisted in the Redis central service.

One configuration of the entities describe above is known as a *topology*. An *experiment* is an execution of a topology. Topologies are defined in Python using the programmatic interface of WoTemu which are then converted into Docker Compose files with the WoTemu Command Line Interface (CLI). This workflow is shown in Figure 2.

One of the main advantages of the WoTemu design is that users are not limited to built-in emulated IoT devices. WoTemu runs real code in real-time. In practice, any Python application can be integrated, which enables users to communicate with the majority of real IoT devices. For example, an application could be designed to communicate over MQTT with a real wireless sensor board, or even

through a serial port in one of the swarm nodes. However, this would require defining placement constraints so the application is always deployed in the swarm node that the board is connected to.

2.1. Routing module

Real life networks, especially those used in the context of IoT deployments, must deal with packet loss, bandwidth and data transfer limitations. These issues must be factored into architecture designs to ensure an adequate implementation. The routing module in WoTemu enables users to emulate the behaviour of real networks transparently. To this end, WoTemu leverages the iproute2 [36] Linux package and the iptables [37] packet filtering tool. The traffic control utility of iproute2 (tc), and particularly the Network Emulator (NetEm) and Token Bucket Filter (TBF) modules, provide an interface to impose arbitrary restrictions on the quality of network connections.

All *node* and *broker* containers are automatically configured on startup by the WoTemu entrypoint with the appropriate routing rules to enable network emulation. These rules are detailed below.

- There is a set of iptables rules that match the ports of the application-layer protocols on the WoT runtime Protocol Binding implementations (i.e. HTTP, Websockets, CoAP and MQTT). These rules are located in the *mangle* table and appended to the OUTPUT chain to add an internal kernel *mark* to the matching packets.
- There is a rule in the routing policy database that causes all packets with the kernel *mark* stamped by iptables to use the WoTemu routing table.
- The WoTemu routing table contains entries to force all matching Protocol Binding packets or datagrams to go through the gateway container of the current Docker overlay network.

Gateway containers run a set of TC processes that shape the traffic passing through the container to emulate the latency, bandwidth and loss conditions of any given network. There is a single gateway container for each network where all traffic from the different containers in the network is aggregated. This leads to a more realistic emulation of the bandwidth of a network, as opposed to imposing bandwidth constraints in each container.

Figure 3 shows a routing configuration where a subset of network communications are redirected through a gateway container that shapes traffic according to predefined rules.

The approach here is to ensure that each individual node is able to configure its own network stack without the intervention of a proactive centralized component. This fulfills one of the main objectives of WoTemu by increasing scalability, at the cost of decreased container isolation. The loss in isolation is because containers have access to

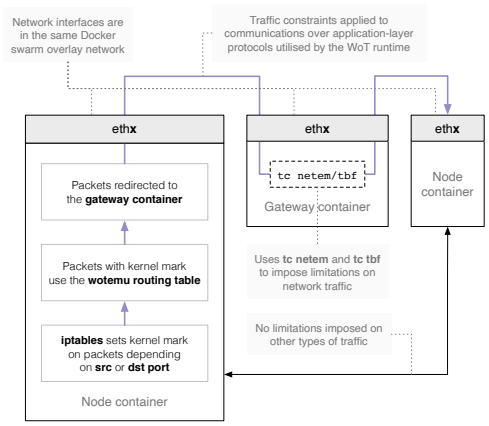


Figure 3: Traffic shaping strategy in WoTemu.

the Docker API proxy service in order to retrieve information on the Docker swarm stack configuration. Examples of centralized information required by containers include:

- the swarm task identifier of the current container.
- WoTemu network identifiers currently attached to the container.
- the swarm task identifier of the gateway container for all WoTemu networks.
- all the currently existing container replicas of any given swarm service.

The main downside of the loss in isolation is that it leads to decreased security. It could be argued, however, that WoTemu experiments are not meant to be publicly exposed in a production environment and are ephemereal in nature. This position significantly eases concerns about security.

2.2. Monitoring module

The monitoring module provides visibility to the performance and operation of the experiment, capturing the relevant metrics during its execution. Captured data points are stored in the central Redis service to be used during the construction of the final report.

There are four distinct monitor processes that are injected into containers during the experiment. This is a transparent operation from the user's point of view, that is, the user-provided application is automatically augmented by the WoTemu entrypoint with the monitor processes. Figure 4 shows an overview of the monitor processes, which are described in more detail in the following paragraphs.

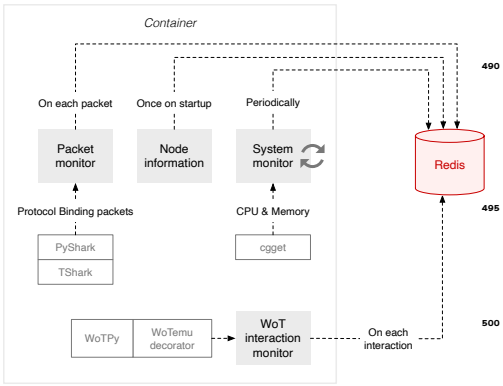


Figure 4: Monitor processes transparently included in containers.

Packet monitor Based on the TShark [38] network sniffer, this monitoring module runs a background process configured with a set of filters that match the ports of the Protocol Binding implementations. The PyShark [39] Python package is used to interface with TShark. All matching packets are pre-processed and stored in Redis to extract meaningful information related to the network footprint of the application.

Node information Runs only once on container startup and collects metadata about the current container that is necessary to identify the application during the generation of the final report. Examples of metadata captured by this process include the CPU model, extended information about network interfaces, the virtual IP addresses for the current swarm service and a snapshot of the environment variables.

System monitor Periodically retrieves the memory and CPU resource usage of the application. These metrics are read by *cgget*, a command line utility to read parameters of Linux control groups (cgroups). This is necessary to ensure that the measurements reflect the resources used by the current container instead of the system as a whole.

WoT interaction monitor The user-provided application function receives an argument that is a decorated version of the WoTPy entrypoint modified to capture and log all WoT interactions. These data points enable a higher-level, WoT-focused analysis of the behaviour of the experiment that cannot be obtained from low-level metrics such as CPU usage.

Furthermore, users can explicitly write ad-hoc *application metrics*. These are namespaced by the hostname of the container (*task*), and an arbitrary metric key provided by the user. They are then stored in Redis alongside the other data points generated by the monitor processes.

2.3. Benchmarking module

Memory and CPU constraints can be easily configured in containers. This can be leveraged to approximately emulate the behaviour of any given application in a constrained environment without the need of actually running it in a real platform. Memory limits are expressed as size, while CPU limits are expressed as a quota on the CPU Completely Fair Scheduler (CFS).

Memory limits retain their significance when applied to different host machines in the swarm cluster, regardless of the total amount of installed memory; for example, a limit of 100MB means the same on all machines. However, the same does not hold for limits on the CPU CFS quota. The same quota can represent wildly different levels of computational power on different CPU models. There is a significant difference between full usage of a single core in a low power ARM processor and in a core in a modern desktop CPU.

WoTemu proposes a solution to the portability of processor constraints by using sysbench [40] synthetic CPU benchmark scores to represent the desired level of computational power in the container. The rationale behind using Sysbench is that it is a versatile, actively maintained, widely available and proven performance benchmark tool with multiple modules. Its popularity, especially in the context of database benchmarking [41], is reflected in the 3.9k stars and 771 forks in GitHub at the time of writing this paper.

The following list describes the strategy used in all swarm nodes to update the CPU quotas of the containers that require CPU constraints. Figure 5 shows a diagram with a simplified view of this process.

1. The WoTemu entrypoint first checks the existence of an environment variable in the container indicating the target CPU performance score. If the variable is defined, the entrypoint uses the WoTemu CLI to update the CPU quota to match the desired computational power.
2. The first container in the swarm node that needs to update the CPU quota creates a new key in Redis and then runs a series of CPU benchmarks. This key is namespaced by the swarm node ID and is the same for all containers in the swarm node. Thus, all containers in the swarm node that reach at this step later are aware that a CPU benchmark is already in process.
3. The CPU benchmarking process consists in running multiple sysbench CPU benchmarks consecutively. Sysbench processes are executed inside ephemereal containers that are constrained with increasing levels of CPU quotas (up to 100% usage of a single core). The aim here is to obtain the coefficients of a fitted

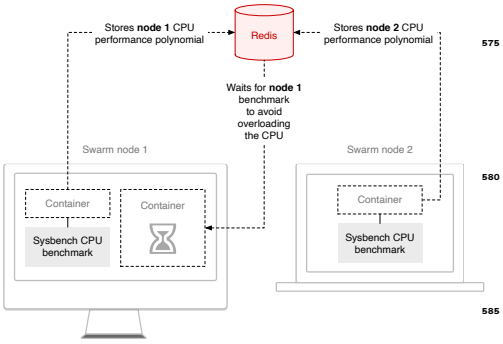


Figure 5: Processor benchmarking stage in WoTemu.

polynomial to characterize the performance of a core in the current CPU.

4. The rest of containers in the swarm node are simply waiting for the CPU performance coefficients to be set in Redis. This ensures that the CPU is not overloaded with multiple parallel benchmarks.
5. When the process ends, all containers update their own CPU quotas by solving the CPU performance polynomial for the target performance score. The target score is initially defined in the swarm service as an environment variable.

Although it would be preferable in terms of computational resources to reuse the performance polynomial for multiple experiments, WoTemu places more importance on being stateless and leaving minimal traces on swarm nodes after an experiment.

2.4. Reporting module

As discussed in Section 2.2, many performance metrics and data items are collected and stored in Redis during an experiment. This is, however, an intermediary state, as the format in which data is kept in Redis cannot be read by human users.

The reporting module transforms captured metrics into user-friendly representations. WoTemu supports two distinct report formats:

- Reports can be rendered as static webpages. To this end, the reporting module relies heavily on different charts provided by the Plotly Python Open Source Graphing Library [42]. An example of a WoTemu static webpage report is available online for demonstration purposes[2].
- The user may need to apply additional processing or use the output of WoTemu as an input for another tool or script. For these cases, WoTemu supports rendering the report as a machine-readable JSON [43] document.

The following list provides a description of views and charts included in a WoTemu report.

Service traffic Total volume of network traffic grouped by *task* and *service* for data coming into the *service* and data coming out of the *service*. It is especially useful to detect at a glance the *services* with the highest activity from a network traffic standpoint.

Network traffic Timeseries of the evolution of network traffic grouped by *network*.

Resource usage rankings Distribution of the memory and CPU usage samples captured during the experiment for all *tasks*. This provides insight into the variance of the resource consumption of *tasks*.

Timeline of tasks Timeline of the creation and stop of all the *tasks* in the experiment. This is a convenient way of reviewing the frequency and occurrence of *task* failures.

Task resource usage Timeseries of the evolution of memory and CPU usage for a given *task*. Two different baselines are used in this case for comparison: the host machine (i.e. swarm node) and any resource constraints imposed on the container.

Task data transfer Timeseries of the volume of network traffic in a *task* grouped by network interface or protocol. Protocol names are identified by the TShark capturing process in the packet monitor.

Task interactions A WoT-focused view that provides insight into the performance of WoT applications deployed in node containers. This includes the distribution of latencies for interaction *request* verbs, the total count of successful and failed interaction verbs and the timeline of event verb occurrences.

For further clarity, the following list describes the structure of a WoTemu output report serialized in JSON format. Each element of the list represents a key in the JSON document; key names containing dots represent nested objects. In addition, as the reporting module makes heavy use of Pandas [44], some of the values in the document are JSON-serialized Pandas DataFrames (DF):

`app_metrics` An array of ad-hoc *application metrics* as defined by the user. The array items are objects that contain the metric key, the task where the metric was generated and an array of data points.

[2] `https://agmangas.github.io/demo-wotemu-report/`

`service_traffic.inbound` DF that contains the total volume of network traffic grouped by *source task* and *destination service*. This allows the traffic coming into any given *service* to be analysed.

`service_traffic.outbound` DF that contains the total volume of network traffic grouped by *source service* and *destination task*. This allows the traffic coming out of any given *service* to be analysed.

`snapshot` DF that contains the status of all swarm *tasks* for all *services* at the time the emulation process is stopped. This includes the service ID, the most recent log entries, the date of creation and the container ID.

`tasks.<task>.info` An object with miscellaneous information about the given *task*, including network interfaces, memory limits, virtual IP addresses for each network and environment variables.

`tasks.<task>.interaction` DF that contains the time series of WoT interactions for both consumed and exposed WoT Things for the given *task*. This includes the payload of WoT events and the latency of consumed WoT actions.

`tasks.<task>.packet` DF that contains the detailed time series of network packets sent or received by the WoT runtime protocol binding implementations for the given *task*. This includes the packet size, the swarm network name and the protocol.

`tasks.<task>.system` DF that contains the time series of system utilization metrics (i.e. CPU and memory) for the given *task*.

A sample of a WoTemu output report in JSON format is available online [45] for further details.

3. Experimental results

3.1. Introduction

This section presents a real-world edge computing scenario and demonstrates how WoTemu topologies can be utilized to analyze the behaviour of IoT architecture proposals. The scenario in question is an approximation of the *Intelligent Surveillance* application discussed in iFogSim [20] [46]. The rationale for selecting this particular scenario was the following:

- It highlights the differences between WoTemu and comparable simulation tools.
- It is complex enough and serves to demonstrate the flexibility of WoTemu topologies. The experiment integrates different services and techniques such as deep learning algorithms and NoSQL data stores.

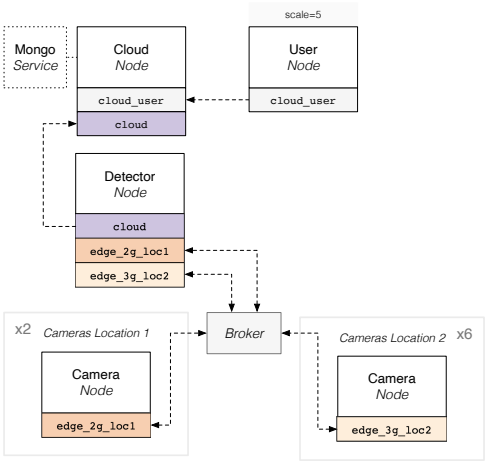


Figure 6: Surveillance application topology (*cloud* version).

The *Intelligent Surveillance* scenario is approximately emulated by a set of smart cameras with motion detection capabilities. In the event of motion detection, video streams from the cameras are forwarded to intelligent object detection modules. The output of the object detection modules is then used to update the Pan Tilt Zoom (PTZ) camera configuration. Finally, there is an interface for the users to check the activity of the system.

A large part of the value of WoTemu lies in the insight obtained from quickly iterating with intermediate versions of the topology. To illustrate this idea, Section 3.2 presents an initial architecture proposal that evolves to a more optimal version based on edge computing in Section 3.3.

The source code of the framework implementation and the experiment described in this section is publicly available [31] and the dataset containing the full experimental results is also published online [45]. All experiments were run on a swarm with 2 nodes with the following specifications:

- Intel Core i7-6770HQ CPU @ 2.60GHz (4 cores, 8 threads),
- 32GB of DDR4 2133 MHz (2x16GB in dual channel configuration).
- 480GB SATA3 SSD.

3.2. Cloud topology

A topology intended to fulfil the requirements of the *Intelligent Surveillance* scenario is shown in Figure 6. The diagram includes the different nodes, the connections between them, the networks attached to each node and the

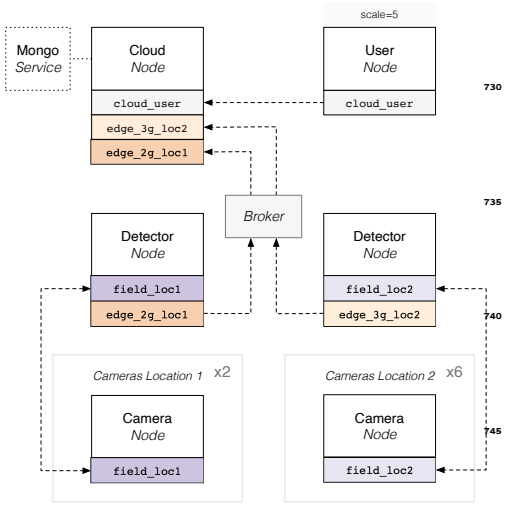


Figure 7: Surveillance application topology (*edge* version).

typical flow of communications. This topology is based on a simple cloud computing approach where the sensor layer is directly connected to the cloud layer.

There are two sets of *cameras* that represent two geographical locations where surveillance cameras are deployed. All *cameras* are connected to a *detector* in the cloud through a MQTT broker. The *detector*, in turn, is monitored by a *cloud* node in the same LAN that maintains a history of the *detector* state. The *cloud* node is the only stateful element in the topology. Finally, a set of *user* nodes communicate with the cloud services.

The four types of node applications used in this topology are further described below:

Camera These nodes are instances of the *camera* built-in app, an application based on OpenCV [47] that aims at emulating a simple video camera. To this end, the application reads a test H.264-encoded local video file [48] in an infinite loop, performing naive motion detection in the process. Movement in the video is detected when the motion score goes over a threshold. The motion score is the mean of the differential of the most recent frames. Whenever motion is detected, the corresponding frame is converted to JPG encoded in Base64 and emitted as an event interaction.

Detector A *detector* subscribes to the JPG frame events of a set of cameras and then processes all received frames to perform face recognition. This process is based on the face-recognition Python package [49], which is publicly available in the Python Package Index. The face-recognition package in turn leverages the deep learning capabilities of dlib [50]. Two read-only properties are exposed by a *detector*. First, the *latestDetections* property contains the most recent in-memory video frame and detection result for each camera. Second, the *cameras* property contains the configuration, including the *camera* URLs. The PTZ adjustment action of a *camera* is invoked by the *detector* on each positive detection (i.e. a human face was detected in the video stream) as long as the previous invocation for said *camera* has already completed. The *detector* application is carefully designed to optimize resource usage: video frames are stored in a buffer queue in memory and processed asynchronously. The processing order of the frames is determined by their priority, which is a function of the age of the frame (in seconds) and the number of frames that have already been processed for that *camera*.

Cloud There is a single *cloud* node in the topology. The *cloud* node is an instance of the *historian* built-in app. Connected to a MongoDB service, it acts as an aggregator, data store and HTTP API for clients to retrieve the detection data from the different locations. A *historian* reads the properties of arbitrary WoT Things passed as arguments, writing the collected samples in MongoDB. In the particular instance of the *cloud* topology, the *cloud* node reads a single *detector* node that is located in the same LAN.

User The user nodes, which check the video frames and face detection results, represent the clients of the architecture. The *cloud* node serves as an entry point for these clients, exposing data in an aggregated fashion through HTTP. *User* nodes are instances of the *caller* built-in app, which continuously invokes the actions of an arbitrary WoT Thing passed as argument. The invocations follow a Poisson process in an attempt to model the behaviour of a group of real users. The request rate λ (1/s) of this process can be passed as an argument. However, it cannot grow indefinitely, as there are limitations imposed by using a single core. To overcome this limitation, the *user* service can be easily scaled up to generate a higher load by changing the number of replicas. In this particular experiment, both λ and the number of replicas are set to 5, which generates a significant volume of requests (approximately 25 requests per second).

It is important to highlight that although this particular experiment only uses built-in apps, that is, applications that are already included in WoTemu for convenience, any Python program that conforms to the asyncio

asynchronous I/O programming model can be integrated into a WoTemu topology.

Table 2 details all the networks, including available bandwidth, latency and jitter. Networks `edge_2g_loc1` and `edge_2g_loc2` represent a cellular network backhaul, which is a common occurrence in IoT sensor deployments. Networks `cloud` and `cloud_user` are effectively unconstrained, with the former representing the LAN interconnecting cloud resources, and the latter the WAN connection between users and cloud resources.

Generally, devices in the sensors layer tend to be more constrained in terms of resources (i.e. memory, computation, storage) than devices in the edge layer, which, in turn, tend to be less capable than devices in the cloud layer. This is a consequence of multiple factors, such as the energy contraints that are usually present in the sensors layer, and the number of devices in each layer. To incorporate these differences into the topology, Table 3 describes the memory and CPU resource constraints imposed on nodes. To represent a processor with very low capabilities, similar to those that could be found in a low-cost camera, the target CPU score of the *camera* nodes is set to 200. For comparison, the value for one thread in a fourth generation mobile Intel Core i7 is approximately 1000.

Please note that Tables 2 and 3 include additional elements, which are explained in Section 3.3.

Two ad-hoc application metrics (see Section 2.2) are defined for the *Intelligent Surveillance* scenario. These ad-hoc metrics are in addition to the performance metrics that are automatically captured by the monitor processes.

Detection latency Detection latency originates in the *detector* nodes and is divided into two metrics. The first one is the time in seconds from the moment a video frame is captured in a *camera* to the moment the frame is enqueued in the *detector*. The second one is the time from the moment the frame enters the queue to the moment the face detection process for that frame is completed.

PTZ latency This application metric originates in the *camera* nodes and contains the time between the timestamp in which a video frame is captured, and the timestamp when the PTZ adjustment invocation for said video frame is received. Note that a video frame must be processed by a *detector* before the PTZ adjustment action is invoked, and that not all video frames result in a PTZ adjustment action invocation. Moreover, the previously described detection latency is fully contained within the PTZ latency.

In the following figures, task names contain an apparently arbitrary alphanumeric suffix, separated by a dot. This is part of an alphanumeric ID that is automatically assigned by Docker swarm mode to uniquely identify each task. The number before said alphanumeric ID is the *replica number* (indexed from 1). Only the most representative and interesting figures are shown here for clarity. Section 2.4 gives a more detailed view of all the reports produced by a WoTemu experiment.

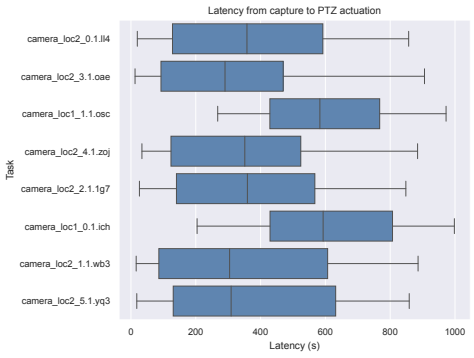


Figure 8: PTZ latency (*cloud* version).

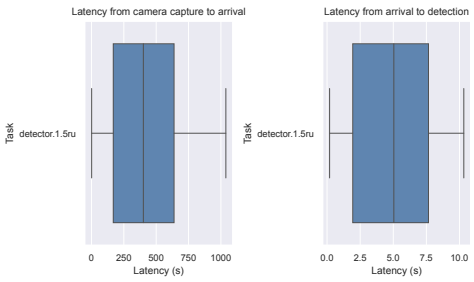


Figure 9: Detection latency (*cloud* version).

The PTZ latency and detection latency application metrics for the *cloud* scenario are shown in Figures 8 and 9 respectively. The PTZ latency distribution is in the order of hundreds of seconds, which is totally inadequate. This is mostly due to network congestion caused by all *cameras* accessing the edge network concurrently to publish the video frame events. The low bandwidth, high packet delay and jitter (see Table 2) exhibited by the networks degrade the quality of the connection and result in issues such as excessive TCP retransmissions.

Latency issues are exacerbated by the message queueing feature of the MQTT binding. That is, messages are enqueued in the broker and delivered sequentially to the *detector*, which causes delays to accumulate.

Figure 10 shows how the latencies of video frames from the *cameras* grow to the point where all frames are rejected

Network name	Constraint profile	Latency	Jitter	Bandwidth
`edge_2g_loc1`	`GPRS`	700 ms	100 ms	50 kbit
`edge_3g_loc2`	`REGULAR_3G`	300 ms	150 ms	1500 kbit
`field_loc1`	`WIFI`	25 ms	5 ms	50 mbit
`field_loc2`	`WIFI`	25 ms	5 ms	50 mbit
`cloud_user`	`CABLE`	5 ms	5 ms	100 mbit
`cloud`	`CABLE`	5 ms	5 ms	100 mbit

Table 2: Details of the networks in the application topologies.

Node application	Memory	CPU score
Camera	256 MB	200
Detector (only in *edge* version)	1 GB	600

Table 3: Node resource limits in the application topologies.

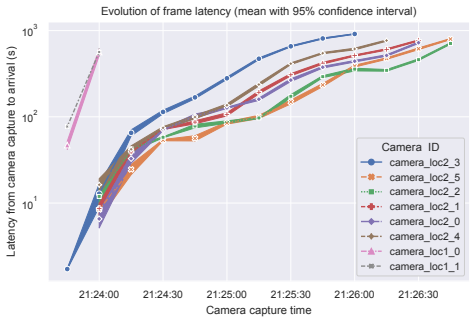


Figure 10: Evolution of frame latency in the *detector* (*cloud* version).

by their respective *detectors* due to an excessive delay. This effect is significantly more pronounced for the first set of *cameras* due to the 2G network constraints imposed on the edge network `edge_2g_loc1`.

3.3. Edge topology

The results discussed in Section 3.2 prove that an arguably naive cloud computing approach is unfeasible for the requirements of this instance of the *Intelligent Surveillance* scenario. In this case, adopting the edge computing paradigm is a viable solution, as discussed below.

Figure 7 shows the *edge* version of the topology, an alternative proposal that leverages the edge computing paradigm. The following list describes in more detail the differences with respect to the *cloud* version:

- Unlike the *cloud* version, a *detector* is locally placed at the edge of each set of *cameras* in an attempt to optimize latency and data transfer volume. That is, *cameras* are not directly connected to the WAN and are in the same LAN as their *detector*.
- The *cloud* node reads two *detector* nodes through a MQTT broker over mobile connections, as opposed to being connected to a single *detector* in the same LAN.
- Resource limits are imposed on *detector* nodes (see Table 3). A benchmark score of 600 is an approximation of the results reported by the ARM processor in a Raspberry Pi Model 3B, which is a popular single-board computer that may be commonly found in the edge layer. The *camera* constraints are the same in the *edge* and *cloud* versions.
- Networks `field_loc1` and `field_loc2` only appear in the *edge* version, acting as a local WLAN connection between the edge *detector* and the *cameras* (see Table 2).
- There are no differences in the behaviour and configuration of the node applications (i.e. *camera*, *detector*, *cloud* and *user*). Therefore, the detection latency and PTZ latency application metrics have the same interpretation as before. See Section 3.2 for a detailed description of applications and metrics.

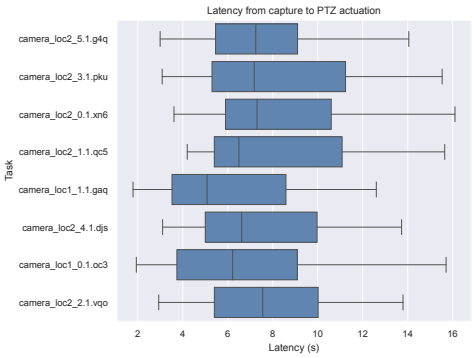


Figure 11: PTZ latency (*edge* version).

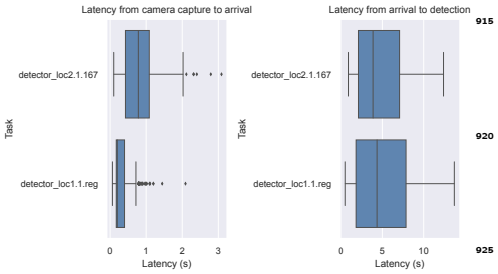


Figure 12: Detection latency (*edge* version).

The edge computing paradigm proves to be a good fit for the *Intelligent Surveillance* scenario, as demonstrated by the observed PTZ latency (Figure 11) and detection latency (Figure 12). Median PTZ latency is under 8 seconds for all *cameras*, which is a reasonable performance, especially when considering the limited resources of the *camera* and *detector* nodes. Placing the *detector* nodes at the edge of the *camera* locations contributes to achieving a median latency from video frame capture to arrival in under 1 second. On the other hand, as expected, latency from arrival to detection shows no differences with respect to the *cloud* version of the topology, this is a purely local process.

Figure 13 shows a heatmap of the network traffic between tasks and services in both inbound and outbound directions. Section 2 shows details of the differences between Docker swarm tasks and services. A service is a template from which one or more tasks are created, and each task is a container. In this case, for example, the *user* service includes all five *user* tasks. A subset of only

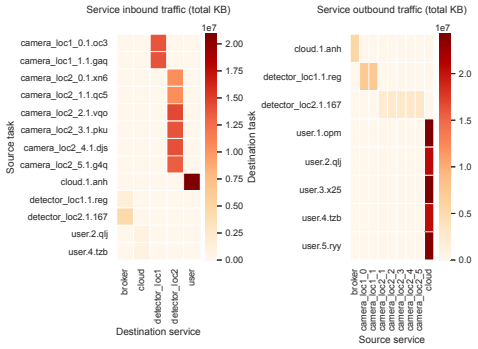


Figure 13: Traffic for the most significant services (*edge* version).

the most significant services and tasks are presented for clarity.

As could be expected, a majority of the network traffic occurs from *cameras* to *detectors*, and bidirectionally between *users* and *cloud*, while traffic from and to the MQTT broker is markedly lower.

Figure 14 shows the distribution of CPU and memory usage samples for all containers. The X axis of the memory subplot shows the percentage of use over the container memory limit rather than the absolute memory. This is because the former provides more information about how close the container is to the point where it runs out of memory. The memory limit of unconstrained containers (*broker*, *user* and *cloud*) is the size of the entire host memory pool (32GB), whereas constrained containers (*detector* and *camera*) have a predefined hard upper limit (see Table 3). The interpretation of the X axis of the CPU subplot varies depending on whether the container is constrained or unconstrained. Unconstrained containers are allowed to use more than one core, and as such can go over 100%—a value of 100% represents full usage of one core. On the other hand, constrained containers are limited to a specific quota in one core, thus, 100% represents full usage of the allocated CPU quota (see Table 3).

Memory constraints imposed on *camera* nodes are adequate for this topology, although these nodes are close to the limit and would benefit from a larger memory pool. This conclusion also applies to the *detector* node in the second location, unlike the *detector* in the first location, which has the capacity to scale up to handle more *cameras*. The *cloud* node, although unconstrained, shows a reasonably small memory footprint (the median memory consumption is approximately 1.3GB, that is, 4% of a total of 32GB). The *detector* nodes are CPU-bound. Furthermore, the logs indicate that a significant percentage of the video frames are dropped due to the internal buffer queue being full (i.e. the *detector* is not able to keep up with the

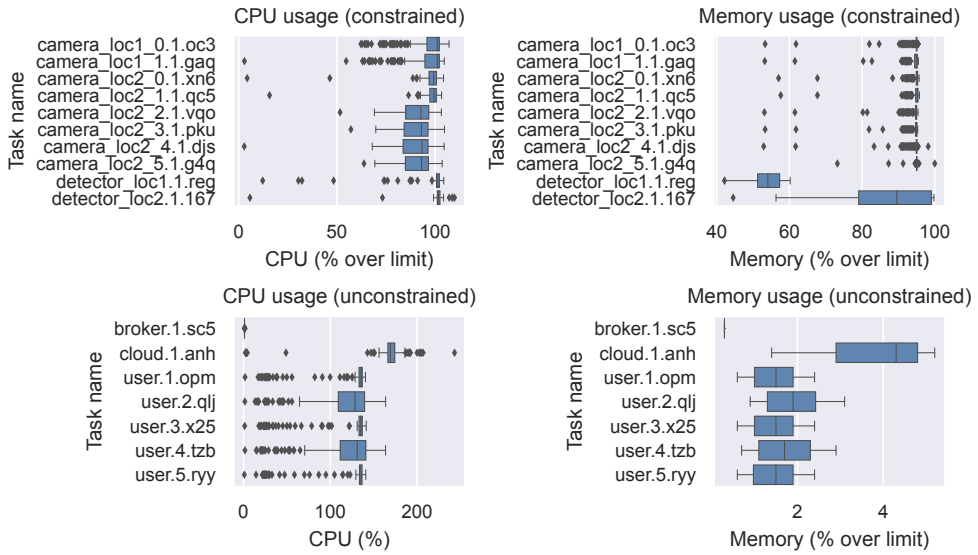


Figure 14: CPU and memory usage distributions (*edge* version).

rate at which frames arrive). Therefore, *detectors* would require a processor with a higher capacity to achieve optimal performance. The *cloud* node is also CPU-bound, as could be expected from a HTTP server handling a significant volume of requests.

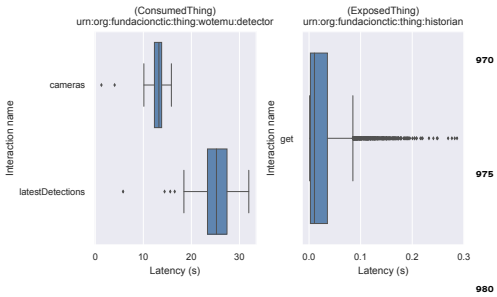


Figure 15: Interaction latencies for task *cloud* (*edge* version).

It can be seen in Figure 15 that *detectors* struggle to serve read requests for their properties (see Section 3.1 for a detailed description of the properties). This is a result of the combination of multiple factors, including the size of the video frames contained in the *latestDetections* property, the performance of the cellular (i.e. 2G and 3G) networks that interconnect the *detectors* with the *cloud*, latencies inherent to the MQTT binding and the resource constraints imposed on *detectors*. The experiment shows that the *edge* topology would be adequate for cases where a reasonably high average latency for video frames with face detection data is acceptable for clients. Otherwise, more resources (i.e. network, computation) would be required by *detectors*.

Finally, Figure 16 shows a representative example of performance metrics in a *camera* node. All the other *cameras* have similar behaviour. These nodes operate close to the limit, as could be expected, due to the severe resource constraints imposed on them. Respecting these constraints is necessary to keep costs to a reasonable limit. It is important to note that, although the majority of the network traffic in the *camera* nodes is transported over the HTTP binding, TShark reports most of the packets as *json* (the serialization format of the body in HTTP requests) or *tcp* (the underlying HTTP transport protocol) rather than *http*.

4. Conclusions

Emulation tools are of great importance for IoT architectures. These systems are characterized by a large

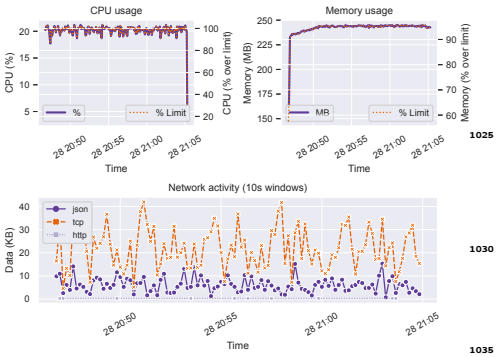


Figure 16: Performance of a *camera* node (*edge* version).

number of interconnected nodes with multiple communication flows, especially in the context of the edge computing paradigm. Therefore, validating the performance of an architecture design is a complex but important issue. Adoption of an emulation tool such as WoTemu leads to a decrease in deployment and testing costs. Unlike other tools, WoTemu scalably runs real application code with minimal modifications. It enables users to test real applications and avoid costly errors by quickly detecting flaws in proposed designs before committing to a deployment in the real world.

WoTemu leverages the repeatability and horizontal scalability capabilities of Docker swarm mode, a modern container orchestration tool, to provide a solution for this validation issue. Furthermore, WoTemu gives detailed insight into the behaviour of IoT applications in multiple domains, including the application layer, network layer and hardware infrastructure. Finally, the W3C WoT specifications, which are a proven solution for the IoT interoperability problem, are considered as first class citizens to ensure a future-proof approach.

A complex real world scenario was used to validate a full-featured experimental implementation. In this scenario, WoTemu was instrumental in detecting bottlenecks, obtaining meaningful application performance metrics, optimizing hardware resources and discarding problematic designs.

Future work will focus on optimizing resource usage for very large experiments, that is, experiments with hundreds of containers or of considerable duration (in the order of several hours or even days). This is likely to entail the evaluation and adoption of Redis cluster in the data storage layer, and also updates to the reporting module to avoid processing large amounts of data in a naive non-distributed way. Another interesting addition would be to include a recommendations module to provide optimization suggestions for the topology, such as automatically detecting bottlenecks in network links.

5. Acknowledgements

This research has been partially funded by the Spanish National Plan of Research, Development and Innovation under the project OCAS (RTI2018-094849-B-I00); and the Spanish Ministry of Science and Innovation through the Centre for the Development of Industrial Technology (CDTI) under the project CEL.IA (CER-20211022).

[1] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog Computing and Its Role in the Internet of Things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16. doi:10.1145/2342509.2342513.

[2] M. De Donno, K. Tange, N. Dragoni, Foundations and Evolution of Modern Computing Paradigms: Cloud, IoT, Edge, and Fog, IEEE Access 7 (2019) 150936–150948. doi:10.1109/ACCESS.2019.2947652.

[3] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, X. Yang, A Survey on the Edge Computing for the Internet of Things, IEEE Access 6 (2018) 6900–6919. doi:10.1109/ACCESS.2017.2778504.

[4] C. Aguzzi, L. Gigli, L. Sciullo, A. Trotta, M. Di Felice, From Cloud to Edge: Seamless Software Migration at the Era of the Web of Things, IEEE Access 8 (2020) 228118–228135. doi:10.1109/ACCESS.2020.3045632.

[5] P. O'Donovan, C. Gallagher, K. Bruton, D. T. O'Sullivan, A fog computing industrial cyber-physical system for embedded low-latency machine learning Industry 4.0 applications, Manufacturing Letters 15 (2018) 139–142. doi:10.1016/j.mfglet.2018.01.005.

[6] J. Mineraud, O. Mazhelis, X. Su, S. Tarkoma, A gap analysis of Internet-of-Things platforms, Computer Communications 89–90 (2016) 5–16. doi:10.1016/j.comcom.2016.03.015.

[7] D. Raggett, The Web of Things: Challenges and Opportunities, Computer 48 (5) (2015) 26–32. doi:10.1109/MC.2015.149.

[8] D. Guinard, V. Trifa, E. Wilde, A resource oriented architecture for the Web of Things, in: 2010 Internet of Things (IOT), 2010, pp. 1–8. doi:10.1109/IOT.2010.5678452.

[9] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, K. Kajimoto, Web of Things (WoT) Architecture, W3C recommendation, W3C (Apr. 2020).
URL https://www.w3.org/TR/2020/REC-wot-architecture-20200409/

[10] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, M. Kovatsch, Web of Things (WoT) Thing Description, W3C recommendation, W3C (Apr. 2020).
URL https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/

[11] D. Longley, P.-A. Champin, G. Kellogg, JSON-LD 1.1, W3C recommendation, W3C (Jul. 2020).

[12] M. Koster, E. Korkan, Web of Things (WoT) Binding Templates, Tech. rep., W3C (Jan. 2020).
URL https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/

[13] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, K. Nimura, Web of Things (WoT) Scripting API, W3C note, W3C (Nov. 2020).
URL https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/

[14] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux J. 2014 (239).

[15] The crun development team, Crun, Red Hat Inc. (Jan. 2022).
URL https://github.com/containers/crun

[16] The Kubernetes Authors, Kubernetes (K8s) (May 2021).
URL https://kubernetes.io/

[17] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, O. Hong Hoe, Evaluation of Docker as Edge computing platform, in: 2015 IEEE Conference on Open Systems (ICOS), 2015, pp. 130–135. doi:10.1109/ICOS.2015.7377291.

[18] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1) (2011) 23–50. doi:10.1002/spe.995.

[19] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, R. Buyya, ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers, Software: Practice and Experience 47 (4) (2017) 505–521. doi:10.1002/spe.2422.

[20] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, R. Buyya, iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments, Software: Practice and Experience 47 (9) (2017) 1275–1296. doi:10.1002/spe.2509.

[21] M. C. S. Filho, R. L. Oliveira, C. C. Monteiro, P. R. M. Inácio, M. M. Freire, CloudSim Plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness, in: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2017, pp. 400–406. doi:10.23919/INM.2017.7987304.

[22] I. Lera, C. Guerrero, C. Juiz, YAFS: A Simulator for IoT Scenarios in Fog Computing, IEEE Access 7 (2019) 91745–91758. doi:10.1109/ACCESS.2019.2927895.

[23] B. Lantz, B. Heller, N. McKeown, A network in a laptop: Rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, p. 19. doi:10.1145/1868447.1868466.

[24] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, H. Karl, MaxiNet: Distributed emulation of software-defined networks, in: Proceedings of the 2014 IFIP Networking Conference (Networking 2014), 2014, pp. 1–9. doi:10.1109/IFIPNetworking.2014.6857078.

[25] M. Peuster, H. Karl, S. van Rossem, MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments, in: 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE, Palo Alto, CA, 2016, pp. 148–153. doi:10.1109/NFV-SDN.2016.7919490.

[26] R. Mayer, L. Graser, H. Gupta, E. Saurez, U. Ramachandran, EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures, in: 2017 IEEE Fog World Congress (FWC), 2017, pp. 1–6. doi:10.1109/FWC.2017.8368525.

[27] A. Coutinho, F. Greve, C. Prazeres, J. Cardoso, Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing, in: 2018 IEEE International Conference on Communications (ICC), 2018, pp. 1–7. doi:10.1109/ICC.2018.8423003.

[28] N. K. Giang, M. Blackstock, R. Lea, V. C. Leung, Developing IoT applications in the Fog: A Distributed Dataflow approach, in: 2015 5th International Conference on the Internet of Things (IOT), IEEE, Seoul, South Korea, 2015, pp. 155–162. doi:10.1109/IOT.2015.7356560.

[29] A. Medina, A. Lakhina, I. Matta, J. Byers, BRITE: An approach to universal topology generation, in: MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001, pp. 346–353.

[30] S. R. Hassan, I. Ahmad, S. Ahmad, A. Alfaify, M. Shafiq, Remote Pain Monitoring Using Fog Computing for e-Healthcare: An Efficient Architecture, Sensors 20 (22) (2020) 6574. doi:10.3390/s20226574.

[31] A. García Mangas, WoTemu: Emulator for edge computing applications in Python, Zenodo (May 2021). doi:10.5281/ZENODO.4769358.

[32] A. García Mangas, F. J. Suárez Alonso, WOTPY: A framework for web of things applications, Computer Communications 147 (2019) 235–251. doi:10.1016/j.comcom.2019.09.004.

[33] R. A. Light, Mosquitto: Server and client implementation of the MQTT protocol, The Journal of Open Source Software 2 (13) (2017) 265. doi:10.21105/joss.00265.

[34] J. Llopis, J. Marques, Docker Socket Proxy, Tecnativa (Jan. 2021).
URL https://github.com/Tecnativa/docker-socket-proxy

[35] Redis Labs, Redis (May 2021).
URL https://redis.io/

[36] A. Kuznetsov, S. Hemminger, Iproute2 (Apr. 2021).
URL https://wiki.linuxfoundation.org/networking/iproute2

[37] R. Russell, Netfilter Core Team, Iptables (Jan. 2021).
URL https://www.netfilter.org/projects/iptables/index.html

[38] Wireshark Foundation, TShark (Apr. 2021).
URL https://www.wireshark.org/

[39] D. Green, Pyshark (Feb. 2021).
URL https://github.com/KimiNewt/pyshark

[40] A. Kopytov, Sysbench (Apr. 2020).
URL https://github.com/akopytov/sysbench

[41] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and Linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, Philadelphia, PA, USA, 2015, pp. 171–172. doi:10.1109/ISPASS.2015.7095802.

[42] Plotly, Plotly Python Open Source Graphing Library (Jan. 2021).
URL https://plotly.com/

[43] T. Bray, The JavaScript object notation (JSON) data interchange format, STD 90, RFC Editor / RFC Editor (Dec. 2017).

[44] J. Reback, W. McKinney, jbrockmendel, J. V. den Bossche, T. Augspurger, P. Cloud, S. Hawkins, gfyoung, Sinhrks, M. Roeschke, A. Klein, T. Petersen, J. Tratner, C. She, W. Ayd, S. Naveh, patrick, M. Garcia, J. Schendel, A. Hayden, D. Saxton, V. Jancauskas, M. Gorelli, R. Shadrach, A. McMaster, P. Battiston, S. Seabold, K. Dong, chris-b1, h-vetinari, Pandas-dev/pandas: Pandas 1.2.4, Zenodo (Apr. 2021). doi:10.5281/zenodo.4681666.

[45] A. García Mangas, WoTemu experimental results (Jun. 2021). doi:10.5281/ZENODO.4782152.

[46] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe, Mobile fog: A programming model for large-scale applications on the internet of things, in: Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing, MCC '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 15–20. doi:10.1145/2491266.2491270.

[47] G. Bradski, The OpenCV library, Dr. Dobb's Journal of Software Tools.

[48] W. Lun, C. Contaoi, Intel IoT Developer Kit Sample Videos, Intel (Nov. 2018).
URL https://github.com/intel-iot-devkit/sample-videos

[49] A. Geitgey, Face Recognition (Feb. 2020).
URL https://github.com/ageitgey/face_recognition

[50] D. E. King, Dlib-ml: A machine learning toolkit, Journal of Machine Learning Research 10 (2009) 1755–1758.

Apéndice C

API framework WoTPy

Este apéndice contiene la documentación de la API de la implementación experimental del framework WoT cuyo diseño se describe en la sección 3. El framework está publicado con el nombre `wotpy` en el repositorio de paquetes *Python Package Index* (PyPI)[1].

La documentación ha sido generada de manera automática con la herramienta Sphinx[2] apoyándose en los comentarios incrustados en el código fuente.

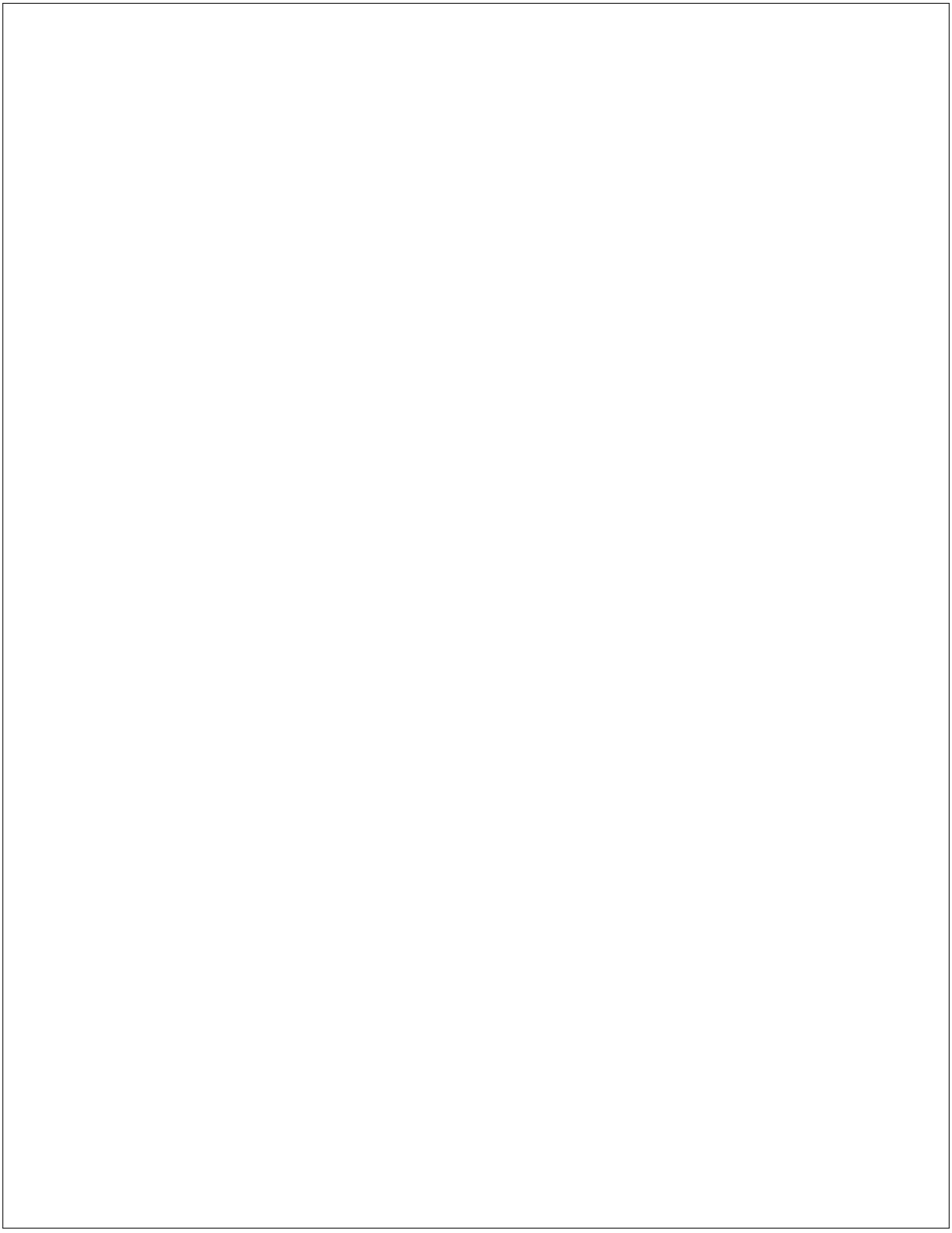
[1] `https://pypi.org/project/wotpy`
[2] `https://www.sphinx-doc.org`

WoTPy Documentation

Release 0.11.0

Andres Garcia Mangas

Mar 25, 2022



CONTENTS

ii

CHAPTER

ONE

TABLE OF CONTENTS

1.1 API Reference

wotpy.wot	Dictionaries and classes defined under the W3C WoT Scripting API specification.
wotpy.utils	Utility functions and classes.
wotpy.protocols	Implementations of the supported Protocol Binding templates.
wotpy.codecs	Classes to serialize and deserialize messages.

1.1.1 wotpy.wot

Dictionaries and classes defined under the W3C WoT Scripting API specification.

wotpy.wot.consumed	ConsumedThing and related entities.
wotpy.wot.dictionaries	Objects defined in the Scripting API specification represented as classes that are basically dict-wrappers.
wotpy.wot.discovery	Thing discovery services.
wotpy.wot.exposed	ExposedThing and related entities.
wotpy.wot.constants	Constants related to objects in the Thing hierarchy.
wotpy.wot.enums	Classes that contain various enumerations.
wotpy.wot.events	Classes that represent events that are emitted by Things.
wotpy.wot.form	Class that represents the form entities exposed by interactions.
wotpy.wot.interaction	Classes that represent all interaction patterns.
wotpy.wot.servient	Class that represents a WoT servient.
wotpy.wot.td	Classes that represent the JSON and JSON-LD serialization formats of a Thing Description document.
wotpy.wot.thing	Class that represents a Thing.
wotpy.wot.validation	Schemas following the JSON Schema specification used to validate the shape of Thing Description documents.
wotpy.wot.wot	Class that serves as the WoT entrypoint.

wotpy.wot.consumed

ConsumedThing and related entities.

wotpy.wot.consumed.interaction_map	Classes that represent Interaction instances accessed on a ConsumedThing.
wotpy.wot.consumed.thing	Class that represents a Thing consumed by a servient.

wotpy.wot.consumed.interaction_map

Classes that represent Interaction instances accessed on a ConsumedThing.

Classes

ConsumedThingAction(consumed_thing, name)	The ThingAction interface implementation for ConsumedThing objects.
ConsumedThingActionDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the ThingAction interface for each action in a given ConsumedThing.
ConsumedThingEvent(consumed_thing, name)	The ThingEvent interface implementation for ConsumedThing objects.
ConsumedThingEventDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the ThingEvent interface for each event in a given ConsumedThing.
ConsumedThingInteractionDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the Interaction interface for each interaction in a given ConsumedThing.
ConsumedThingProperty(consumed_thing, name)	The ThingProperty interface implementation for ConsumedThing objects.
ConsumedThingPropertyDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the ThingProperty interface for each property in a given ConsumedThing.

class wotpy.wot.consumed.interaction_map.**ConsumedThingAction**(*consumed_thing*, *name*)

Bases: object

The ThingAction interface implementation for ConsumedThing objects.

invoke(**args*, ***kwargs*)

The invoke() method when invoked, starts the Action interaction with the input value provided by the inputValue argument.

class wotpy.wot.consumed.interaction_map.**ConsumedThingActionDict**(**args*, ***kwargs*)

Bases: wotpy.wot.consumed.interaction_map.ConsumedThingInteractionDict

A dictionary that provides lazy access to the objects that implement the ThingAction interface for each action in a given ConsumedThing.

interaction_dict

Returns an interactions dict by name. The dict values are the raw dict interactions as contained in a TD document.

thing_interaction_class

Returns the class that implements the Interaction interface for this type of interaction.

class wotpy.wot.consumed.interaction_map.**ConsumedThingEvent**(*consumed_thing*, *name*)

Bases: object

The ThingEvent interface implementation for ConsumedThing objects.

subscribe(**args*, ***kwargs*)
Subscribe to an stream of emissions of this event.

class wotpy.wot.consumed.interaction_map.**ConsumedThingEventDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.consumed.interaction_map.ConsumedThingInteractionDict*

A dictionary that provides lazy access to the objects that implement the ThingEvent interface for each event in a given ConsumedThing.

interaction_dict
Returns an interactions dict by name. The dict values are the raw dict interactions as contained in a TD document.

thing_interaction_class
Returns the class that implements the Interaction interface for this type of interaction.

class wotpy.wot.consumed.interaction_map.**ConsumedThingInteractionDict**(**args*, ***kwargs*)

Bases: collections.UserDict

A dictionary that provides lazy access to the objects that implement the Interaction interface for each interaction in a given ConsumedThing.

interaction_dict
Returns an interactions dict by name. The dict values are the raw dict interactions as contained in a TD document.

thing_interaction_class
Returns the class that implements the Interaction interface for this type of interaction.

class wotpy.wot.consumed.interaction_map.**ConsumedThingProperty**(*consumed_thing*, *name*)

Bases: object

The ThingProperty interface implementation for ConsumedThing objects.

read(*timeout=None*, *client_kwargs=None*)
The read() method will fetch the value of the Property. A coroutine that yields the value or raises an error.

subscribe(**args*, ***kwargs*)
Subscribe to an stream of events emitted when the property value changes.

write(*value*, *timeout=None*, *client_kwargs=None*)
The write() method will attempt to set the value of the Property specified in the value argument whose type SHOULD match the one specified by the type property. A coroutine that yields on success or raises an error.

class wotpy.wot.consumed.interaction_map.**ConsumedThingPropertyDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.consumed.interaction_map.ConsumedThingInteractionDict*

A dictionary that provides lazy access to the objects that implement the ThingProperty interface for each property in a given ConsumedThing.

interaction_dict
Returns an interactions dict by name. The dict values are the raw dict interactions as contained in a TD document.

thing_interaction_class
Returns the class that implements the Interaction interface for this type of interaction.

wotpy.wot.consumed.thing

Class that represents a Thing consumed by a servient.

Classes

ConsumedThing(servient, td)	An entity that serves to interact with a Thing.

class wotpy.wot.consumed.thing.**ConsumedThing**(*servient*, *td*)
Bases: object

An entity that serves to interact with a Thing. An application uses this class when it acts as a *client* of the Thing.

actions
Returns a dictionary of ThingAction items.

events
Returns a dictionary of ThingEvent items.

invoke_action(*name*, *input_value=None*, *timeout=None*, *client_kwargs=None*)
Takes the Action name from the name argument and the list of parameters, then requests from the underlying platform and the Protocol Bindings to invoke the Action on the remote Thing and return the result. Returns a Future that resolves with the return value or rejects with an Error.

links
Represents a dictionary of WebLink items.

on_event(*name*, *client_kwargs=None*)
Returns an Observable for the Event specified in the name argument, allowing subscribing to and unsubscribing from notifications.

on_property_change(*name*, *client_kwargs=None*)
Returns an Observable for the Property specified in the name argument, allowing subscribing to and unsubscribing from notifications.

on_td_change()
Returns an Observable, allowing subscribing to and unsubscribing from notifications to the Thing Description.

properties
Returns a dictionary of ThingProperty items.

read_property(*name*, *timeout=None*, *client_kwargs=None*)
Takes the Property name as the name argument, then requests from the underlying platform and the Protocol Bindings to retrieve the Property on the remote Thing and return the result. Returns a Future that resolves with the Property value or rejects with an Error.

servient
Returns the Servient that contains this Consumed Thing.

subscribe(**args*, ***kwargs*)
Subscribes to changes on the TD of this thing.

td
Returns the ThingDescription instance that represents the TD that this Consumed Thing is based on.

write_property(*name*, *value*, *timeout=None*, *client_kwargs=None*)
Takes the Property name as the name argument and the new value as the value argument, then requests from the underlying platform and the Protocol Bindings to update the Property on the remote Thing and return the result. Returns a Future that resolves on success or rejects with an Error.

wotpy.wot.dictionaries

Objects defined in the Scripting API specification represented as classes that are basically dict-wrappers.

wotpy.wot.dictionaries.base	Base class for WoT dictionaries.
wotpy.wot.dictionaries.filter	Wrapper class for dictionaries to represent Thing filters.
wotpy.wot.dictionaries.interaction	Wrapper classes for dictionaries for interaction initialization that are defined in the Scripting API.
wotpy.wot.dictionaries.link	Wrapper classes for link dictionaries defined in the Scripting API.
wotpy.wot.dictionaries.schema	Wrapper classes for data schema dictionaries defined in the Scripting API.
wotpy.wot.dictionaries.security	Wrapper classes for security dictionaries defined in the Scripting API.
wotpy.wot.dictionaries.thing	Wrapper class for dictionaries to represent Things.
wotpy.wot.dictionaries.version	Wrapper classes for versioning dictionaries defined in the Scripting API.

wotpy.wot.dictionaries.base

Base class for WoT dictionaries.

Classes

WotBaseDict(*args, **kwargs)	Base class for all WoT data types represented as dictionaries in the Scripting API specification.

class wotpy.wot.dictionaries.base.**WotBaseDict**(**args*, ***kwargs*)
Bases: object

Base class for all WoT data types represented as dictionaries in the Scripting API specification.

class Meta
Bases: object

defaults = {}

fields = {}

required = {}

to_dict()
Returns the pure dict (JSON-serializable) representation of this WoT dictionary.

wotpy.wot.dictionaries.filter

Wrapper class for dictionaries to represent Thing filters.

Classes

ThingFilterDict(*args, **kwargs)	The ThingFilter dictionary that represents the constraints for discovering Things as key-value pairs.

class wotpy.wot.dictionaries.filter.**ThingFilterDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

The ThingFilter dictionary that represents the constraints for discovering Things as key-value pairs.

class Meta

Bases: object

defaults = {'method': 'any'}

fields = {'fragment', 'method', 'query', 'url'}

wotpy.wot.dictionaries.interaction

Wrapper classes for dictionaries for interaction initialization that are defined in the Scripting API.

Classes

ActionFragmentDict(*args, **kwargs)	A dictionary wrapper class that contains data to initialize an Action.
EventFragmentDict(*args, **kwargs)	A dictionary wrapper class that contains data to initialize an Event.
InteractionFragmentDict(*args, **kwargs)	Base class for the three types of Interaction patterns (Properties, Actions and Events).
PropertyFragmentDict(*args, **kwargs)	A dictionary wrapper class that contains data to initialize a Property.

class wotpy.wot.dictionaries.interaction.**ActionFragmentDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.interaction.InteractionFragmentDict*

A dictionary wrapper class that contains data to initialize an Action.

class Meta

Bases: object

defaults = {'idempotent': False, 'safe': False}

fields = {'description', 'forms', 'idempotent', 'input', 'output', 'safe', 'scopes'

input

Used to define the input data schema of the action.

output

Used to define the output data schema of the action.

class wotpy.wot.dictionaries.interaction.**EventFragmentDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.interaction.InteractionFragmentDict*

A dictionary wrapper class that contains data to initialize an Event.

class Meta
Bases: object

fields = {'cancellation', 'data', 'description', 'forms', 'scopes', 'security', 'su

cancellation
Defines any data that needs to be passed to cancel a subscription, e.g., a specific message to remove a Webhook.

data
Defines the data schema of the Event instance messages pushed by the Thing.

subscription
Defines data that needs to be passed upon subscription, e.g., filters or message format for setting up Webhooks.

class wotpy.wot.dictionaries.interaction.**InteractionFragmentDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

Base class for the three types of Interaction patterns (Properties, Actions and Events).

class Meta
Bases: object

fields = {'description', 'forms', 'scopes', 'security', 'title', 'uriVariables'}

forms
Indicates one or more endpoints from which an interaction pattern is accessible.

security
Set of security configurations, provided as an array, that must all be satisfied for access to resources at or below the current level, if not overridden at a lower level.

uri_variables
Define URI template variables as collection based on DataSchema declarations.

class wotpy.wot.dictionaries.interaction.**PropertyFragmentDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.interaction.InteractionFragmentDict*

A dictionary wrapper class that contains data to initialize a Property.

class Meta
Bases: object

fields = {'description', 'forms', 'observable', 'scopes', 'security', 'title', 'uri

data_schema
The DataSchema that represents the schema of this property.

to_dict()
Returns the pure dict (JSON-serializable) representation of this WoT dictionary.

writable
Returns True if this Property is writable.

wotpy.wot.dictionaries.link

Wrapper classes for link dictionaries defined in the Scripting API.

Classes

FormDict(*args, **kwargs)	Communication metadata indicating where a service can be accessed by a client application.
LinkDict(*args, **kwargs)	A Web link, as specified by IETF RFC 8288.

class wotpy.wot.dictionaries.link.**FormDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.link.LinkDict*

Communication metadata indicating where a service can be accessed by a client application. An interaction might have more than one form.

class Meta

Bases: object

defaults = {'contentType': 'application/json'}

fields = {'anchor', 'contentType', 'href', 'op', 'rel', 'scopes', 'security', 'subp

required = {'href'}

resolve_uri(*base=None*)

Resolves and returns the Link URI. When the href does not contain a full URL the base URI is joined with said href.

security

Set of security configurations, provided as an array, that must all be satisfied for access to resources at or below the current level, if not overridden at a lower level

class wotpy.wot.dictionaries.link.**LinkDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

A Web link, as specified by IETF RFC 8288.

class Meta

Bases: object

fields = {'anchor', 'href', 'rel', 'type'}

required = {'href'}

wotpy.wot.dictionaries.schema

Wrapper classes for data schema dictionaries defined in the Scripting API.

Classes

ArraySchemaDict(*args, **kwargs)	Properties to describe an array type.
BooleanSchemaDict(*args, **kwargs)	Properties to describe a boolean type.

Continued on next page

Table 11 – continued from previous page

DataSchemaDict(*args, **kwargs)	Represents the common properties of a value type definition.
IntegerSchema(*args, **kwargs)	Properties to describe an integer type.
NumberSchemaDict(*args, **kwargs)	Properties to describe a numeric type.
ObjectSchemaDict(*args, **kwargs)	Properties to describe an object type.
StringSchemaDict(*args, **kwargs)	Properties to describe a string type.

class wotpy.wot.dictionaries.schema.**ArraySchemaDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.schema.DataSchemaDict*

Properties to describe an array type.

class Meta
Bases: object

defaults = {'readOnly': False, 'writeOnly': False}

fields = {'const', 'description', 'enum', 'items', 'maxItems', 'minItems', 'readOnl

items
Used to define the characteristics of an array.

type
The type property represents the value type enumerated in DataType.

class wotpy.wot.dictionaries.schema.**BooleanSchemaDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.schema.DataSchemaDict*

Properties to describe a boolean type.

type
The type property represents the value type enumerated in DataType.

class wotpy.wot.dictionaries.schema.**DataSchemaDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

Represents the common properties of a value type definition.

class Meta
Bases: object

defaults = {'readOnly': False, 'writeOnly': False}

fields = {'const', 'description', 'enum', 'readOnly', 'title', 'type', 'unit', 'wri

classmethod build(**args*, ***kwargs*)
Builds an instance of the appropriate subclass for the given ValueType.

class wotpy.wot.dictionaries.schema.**IntegerSchema**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.schema.NumberSchemaDict*

Properties to describe an integer type.

type
The type property represents the value type enumerated in DataType.

class wotpy.wot.dictionaries.schema.**NumberSchemaDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.schema.DataSchemaDict*

Properties to describe a numeric type.

class Meta
Bases: object

defaults = {'readOnly': False, 'writeOnly': False}

fields = {'const', 'description', 'enum', 'maximum', 'minimum', 'readOnly', 'title'

type
The type property represents the value type (a member of DataType).

class wotpy.wot.dictionaries.schema.**ObjectSchemaDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.schema.DataSchemaDict*

Properties to describe an object type.

class Meta
Bases: object

defaults = {'readOnly': False, 'writeOnly': False}

fields = {'const', 'description', 'enum', 'properties', 'readOnly', 'required', 'ti

properties
Data schema nested definitions.

type
The type property represents the value type enumerated in DataType.

class wotpy.wot.dictionaries.schema.**StringSchemaDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.schema.DataSchemaDict*

Properties to describe a string type.

type
The type property represents the value type enumerated in DataType.

wotpy.wot.dictionaries.security

Wrapper classes for security dictionaries defined in the Scripting API.

Classes

APIKeySecuritySchemeDict(*args, **kwargs)	API key authentication security configuration.
BasicSecuritySchemeDict(*args, **kwargs)	Basic authentication security configuration using an unencrypted username and password.
BearerSecuritySchemeDict(*args, **kwargs)	Bearer token authentication security configuration.
CertSecuritySchemeDict(*args, **kwargs)	Certificate-base asymmetric key security configuration.
DigestSecuritySchemeDict(*args, **kwargs)	Digest authentication security configuration.
NoSecuritySchemeDict(*args, **kwargs)	A security configuration indicating there is no authentication or other mechanism required to access the resource.
OAuth2SecuritySchemeDict(*args, **kwargs)	OAuth2 authentication security configuration.
PSKSecuritySchemeDict(*args, **kwargs)	Pre-shared key authentication security configuration.
PoPSecuritySchemeDict(*args, **kwargs)	Proof-of-possession token authentication security configuration.
PublicSecuritySchemeDict(*args, **kwargs)	Raw public key asymmetric key security configuration.
SecuritySchemeDict(*args, **kwargs)	Contains security related configuration.

class wotpy.wot.dictionaries.security.**APIKeySecuritySchemeDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

API key authentication security configuration. This is for the case where the access token is opaque and is not using a standard token format.

class Meta

Bases: object

defaults = {'in': 'query'}

fields = {'description', 'in', 'name', 'proxy', 'scheme'}

required = {'scheme'}

scheme

The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**BasicSecuritySchemeDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Basic authentication security configuration using an unencrypted username and password.

class Meta

Bases: object

defaults = {'in': 'header'}

fields = {'description', 'in', 'name', 'proxy', 'scheme'}

required = {'scheme'}

scheme

The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**BearerSecuritySchemeDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Bearer token authentication security configuration. This scheme is intended for situations where bearer tokens are used independently of OAuth2. If the oauth2 scheme is specified it is not generally necessary to specify this scheme as well as it is implied.

class Meta

Bases: object

defaults = {'alg': 'ES256', 'format': 'jwt', 'in': 'header'}

fields = {'alg', 'authorization', 'description', 'format', 'in', 'name', 'proxy', '

required = {'scheme'}

scheme

The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**CertSecuritySchemeDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Certificate-base asymmetric key security configuration.

class Meta

Bases: object

fields = {'description', 'identity', 'proxy', 'scheme'}

required = {'scheme'}

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**DigestSecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Digest authentication security configuration. This scheme is similar to basic authentication but with added features to avoid man-in-the-middle attacks.

class Meta
Bases: object

defaults = {'in': 'header', 'qop': 'auth'}

fields = {'description', 'in', 'name', 'proxy', 'qop', 'scheme'}

required = {'scheme'}

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**NoSecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

A security configuration indicating there is no authentication or other mechanism required to access the resource.

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**OAuth2SecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

OAuth2 authentication security configuration. For the implicit flow the authorization and scopes are required. For the password and client flows both token and scopes are required. For the code flow authorization, token, and scopes are required.

class Meta
Bases: object

defaults = {'flow': 'implicit'}

fields = {'authorization', 'description', 'flow', 'proxy', 'refresh', 'scheme', 'sc

required = {'scheme'}

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**PSKSecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Pre-shared key authentication security configuration.

class Meta
Bases: object

fields = {'description', 'identity', 'proxy', 'scheme'}

required = {'scheme'}

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**PoPSecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Proof-of-possession token authentication security configuration.

class Meta
Bases: object

defaults = {'alg': 'ES256', 'format': 'jwt', 'in': 'header'}

fields = {'alg', 'authorization', 'description', 'format', 'in', 'name', 'proxy', '

required = {'scheme'}

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**PublicSecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.security.SecuritySchemeDict*

Raw public key asymmetric key security configuration.

class Meta
Bases: object

fields = {'description', 'identity', 'proxy', 'scheme'}

required = {'scheme'}

scheme
The scheme property represents the identification of the security scheme to be used for the Thing.

class wotpy.wot.dictionaries.security.**SecuritySchemeDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

Contains security related configuration.

class Meta
Bases: object

fields = {'description', 'proxy', 'scheme'}

required = {'scheme'}

classmethod build(**args*, ***kwargs*)
Builds an instance of the appropriate subclass for the given SecurityScheme.

wotpy.wot.dictionaries.thing

Wrapper class for dictionaries to represent Things.

Classes

ThingFragment(*args, **kwargs)	ThingFragment is a wrapper around a dictionary that contains properties representing semantic metadata and interactions (Properties, Actions and Events).

class wotpy.wot.dictionaries.thing.**ThingFragment**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

ThingFragment is a wrapper around a dictionary that contains properties representing semantic metadata and interactions (Properties, Actions and Events). It is used for initializing an internal representation of a Thing Description, and it is also used in ThingFilter.

class Meta
Bases: object

fields = {'actions', 'base', 'created', 'description', 'events', 'id', 'lastModifie

fields_dict = ['properties', 'actions', 'events']

fields_instance = ['version']

fields_list = ['links', 'security']

fields_readonly = ['id']

fields_str = ['name', 'description', 'support', 'created', 'lastModified', 'base']

required = {'id'}

actions
The actions optional attribute represents a dict with keys that correspond to Action names and values of type ActionFragment.

events
The events optional attribute represents a dictionary with keys that correspond to Event names and values of type EventFragment.

links
The links optional attribute represents an array of Link objects.

name
The name of the Thing. This property returns the ID if the name is undefined.

properties
The properties optional attribute represents a dict with keys that correspond to Property names and values of type PropertyFragment.

security
Set of security configurations, provided as an array, that must all be satisfied for access to resources at or below the current level, if not overridden at a lower level. A default nosec security scheme will be provided if none are defined.

version
Provides version information.

wotpy.wot.dictionaries.version

Wrapper classes for versioning dictionaries defined in the Scripting API.

Classes

VersioningDict(*args, **kwargs)	Carries version information about the TD instance.

class wotpy.wot.dictionaries.version.**VersioningDict**(**args*, ***kwargs*)
Bases: *wotpy.wot.dictionaries.base.WotBaseDict*

Carries version information about the TD instance. If required, additional version information such as firmware and hardware version (term definitions outside of the TD namespace) can be extended here.

class Meta
Bases: object

fields = {'instance'}

required = {'instance'}

wotpy.wot.discovery

Thing discovery services.

wotpy.wot.discovery.dnssd	DNS Service Discovery (based on Multicast DNS) Thing discovery service.

wotpy.wot.discovery.dnssd

DNS Service Discovery (based on Multicast DNS) Thing discovery service.

wotpy.wot.discovery.dnssd.service	Service discovery based on Multicast DNS and DNS-SD (Bonjour, Avahi).

wotpy.wot.discovery.dnssd.service

Service discovery based on Multicast DNS and DNS-SD (Bonjour, Avahi).

Functions

build_servient_service_info(servient[, ...])	Takes a Servient and optional IP address and builds the zeroconf ServiceInfo that describes the WoT Servient service.

Classes

DNSSDDiscoveryService([address])	Manages a DNS Service Discovery service (based on Multicast DNS) that is run on a separate thread (on a loop executor) to discover link-local WoT Servients and expose its own.

class wotpy.wot.discovery.dnssd.service.**DNSSDDiscoveryService**(*address=None*)
Bases: object

Manages a DNS Service Discovery service (based on Multicast DNS) that is run on a separate thread (on a loop executor) to discover link-local WoT Servients and expose its own.

WOT_SERVICE_TYPE = '_wot-servient._tcp.local.'

find(*min_results=None*, *timeout=5*)
Browses the link to discover WoT Servient services using mDNS. Returns a list of (ip_address, port). If min_results is defined it will stop as soon as that number of results are found.

is_running
Returns True if the mDNS service is currently running.

register(*servient*, *instance_name=None*)
Takes a Servient and registers the TD catalogue service for discovery by other hosts in the same link.

start()
Starts the DNS-SD thread on a loop executor.

stop()
Signals the DNS-SD thread to stop and waits for the executor future to yield.

unregister(*servient*, *instance_name=None*)
Takes a Servient and unregisters the TD catalogue service.

wotpy.wot.discovery.dnssd.service.**build_servient_service_info**(*servient*, *address=None*, *instance_name=None*)
Takes a Servient and optional IP address and builds the zeroconf ServiceInfo that describes the WoT Servient service.

wotpy.wot.exposed

ExposedThing and related entities.

wotpy.wot.exposed.interaction_map	Classes that represent Interaction instances accessed on a ExposedThing.
wotpy.wot.exposed.thing	Classes that represent Things exposed by a servient.
wotpy.wot.exposed.thing_set	Class that represents a group or set of ExposedThing instances that exist in the same context.

wotpy.wot.exposed.interaction_map

Classes that represent Interaction instances accessed on a ExposedThing.

Classes

ExposedThingAction(exposed_thing, name)	The ThingAction interface implementation for ExposedThing objects.

Continued on next page

Table 20 – continued from previous page

ExposedThingActionDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the ThingAction interface for each action in a given ExposedThing.
ExposedThingEvent(exposed_thing, name)	The ThingEvent interface implementation for ExposedThing objects.
ExposedThingEventDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the ThingEvent interface for each event in a given ExposedThing.
ExposedThingInteractionDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the Interaction interface for each interaction in a given ExposedThing.
ExposedThingProperty(exposed_thing, name)	The ThingProperty interface implementation for ExposedThing objects.
ExposedThingPropertyDict(*args, **kwargs)	A dictionary that provides lazy access to the objects that implement the ThingProperty interface for each property in a given ExposedThing.

class wotpy.wot.exposed.interaction_map.**ExposedThingAction**(*exposed_thing*, *name*)

Bases: object

The ThingAction interface implementation for ExposedThing objects.

invoke(**args*)

The run() method when invoked, starts the Action interaction with the input value provided by the inputValue argument.

class wotpy.wot.exposed.interaction_map.**ExposedThingActionDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.exposed.interaction_map.ExposedThingInteractionDict*

A dictionary that provides lazy access to the objects that implement the ThingAction interface for each action in a given ExposedThing.

interaction_dict

Returns the InteractionPattern objects dict by name.

thing_interaction_class

Returns the class that implements the Interaction interface for this type of interaction.

class wotpy.wot.exposed.interaction_map.**ExposedThingEvent**(*exposed_thing*, *name*)

Bases: object

The ThingEvent interface implementation for ExposedThing objects.

emit(*payload*)

Emits an event that carries data specified by the payload argument.

subscribe(**args*, ***kwargs*)

Subscribe to an stream of emissions of this event.

class wotpy.wot.exposed.interaction_map.**ExposedThingEventDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.exposed.interaction_map.ExposedThingInteractionDict*

A dictionary that provides lazy access to the objects that implement the ThingEvent interface for each event in a given ExposedThing.

interaction_dict

Returns the InteractionPattern objects dict by name.

thing_interaction_class
Returns the class that implements the Interaction interface for this type of interaction.

class wotpy.wot.exposed.interaction_map.**ExposedThingInteractionDict**(**args*, ***kwargs*)

Bases: collections.UserDict

A dictionary that provides lazy access to the objects that implement the Interaction interface for each interaction in a given ExposedThing.

interaction_dict
Returns the InteractionPattern objects dict by name.

thing_interaction_class
Returns the class that implements the Interaction interface for this type of interaction.

class wotpy.wot.exposed.interaction_map.**ExposedThingProperty**(*exposed_thing*, *name*)

Bases: object

The ThingProperty interface implementation for ExposedThing objects.

read()
The get() method will fetch the value of the Property. A coroutine that yields the value or raises an error.

subscribe(**args*, ***kwargs*)
Subscribe to an stream of events emitted when the property value changes.

write(*value*)
The set() method will attempt to set the value of the Property specified in the value argument whose type SHOULD match the one specified by the type property. A coroutine that yields on success or raises an error.

class wotpy.wot.exposed.interaction_map.**ExposedThingPropertyDict**(**args*, ***kwargs*)

Bases: *wotpy.wot.exposed.interaction_map.ExposedThingInteractionDict*

A dictionary that provides lazy access to the objects that implement the ThingProperty interface for each property in a given ExposedThing.

interaction_dict
Returns the InteractionPattern objects dict by name.

thing_interaction_class
Returns the class that implements the Interaction interface for this type of interaction.

wotpy.wot.exposed.thing

Classes that represent Things exposed by a servient.

Classes

ExposedThing(servient, thing)	An entity that serves to define the behavior of a Thing.

class wotpy.wot.exposed.thing.**ExposedThing**(*servient*, *thing*)

Bases: object

An entity that serves to define the behavior of a Thing. An application uses this class when it acts as the Thing 'server'.

class HandlerKeys
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of handler keys.

INVOKE_ACTION = 'invoke_action'

OBSERVE = 'observe'

RETRIEVE_PROPERTY = 'retrieve_property'

UPDATE_PROPERTY = 'update_property'

class InteractionStateKeys
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of interaction state keys.

PROPERTY_VALUES = 'property_values'

actions
Returns a dictionary of ThingAction items.

add_action(*name*, *action_init*, *action_handler=None*)
Adds an Action to the Thing object as defined by the action argument of type ThingActionInit and updates th,e Thing Description.

add_event(*name*, *event_init*)
Adds an event to the Thing object as defined by the event argument of type ThingEventInit and updates the Thing Description.

add_property(*name*, *property_init*, *value=None*)
Adds a Property defined by the argument and updates the Thing Description. Takes an instance of ThingPropertyInit as argument.

destroy()
Stop serving external requests for the Thing and destroy the object. Note that eventual unregistering should be done before invoking this method.

emit_event(*event_name*, *payload*)
Emits an the event initialized with the event name specified by the event_name argument and data specified by the payload argument.

events
Returns a dictionary of ThingEvent items.

expose()
Start serving external requests for the Thing, so that WoT interactions using Properties, Actions and Events will be possible.

id
Returns the ID of the Thing.

invoke_action(*name*, *input_value=None*)
Invokes an Action with the given parameters and yields with the invocation result.

on_event(*name*)
Returns an Observable for the Event specified in the name argument, allowing subscribing to and unsubscribing from notifications.

on_property_change(*name*)
Returns an Observable for the Property specified in the name argument, allowing subscribing to and unsubscribing from notifications.

on_td_change()
Returns an Observable, allowing subscribing to and unsubscribing from notifications to the Thing Description.

properties
Returns a dictionary of ThingProperty items.

read_property(*name*)
Takes the Property name as the name argument, then requests from the underlying platform and the Protocol Bindings to retrieve the Property on the remote Thing and return the result. Returns a Future that resolves with the Property value or rejects with an Error.

remove_action(*name*)
Removes the Action specified by the name argument, updates the Thing Description and returns the object.

remove_event(*name*)
Removes the event specified by the name argument, updates the Thing Description and returns the object.

remove_property(*name*)
Removes the Property specified by the name argument, updates the Thing Description and returns the object.

servient
Servient that contains this ExposedThing.

set_action_handler(*name*, *action_handler*)
Takes name as string argument and action_handler as argument of type ActionHandler. Sets the handler function for the specified Action matched by name. Throws on error. Returns a reference to the same object for supporting chaining.

set_property_read_handler(*name*, *read_handler*)
Takes name as string argument and read_handler as argument of type PropertyReadHandler. Sets the handler function for reading the specified Property matched by name. Throws on error. Returns a reference to the same object for supporting chaining.

set_property_write_handler(*name*, *write_handler*)
Takes name as string argument and write_handler as argument of type PropertyWriteHandler. Sets the handler function for writing the specified Property matched by name. Throws on error. Returns a reference to the same object for supporting chaining.

subscribe(**args*, ***kwargs*)
Subscribes to changes on the TD of this thing.

thing
Returns the object that represents the Thing beneath this ExposedThing.

write_property(*name*, *value*)
Takes the Property name as the name argument and the new value as the value argument, then requests from the underlying platform and the Protocol Bindings to update the Property on the remote Thing and return the result. Returns a Future that resolves on success or rejects with an Error.

wotpy.wot.exposed.thing_set

Class that represents a group or set of ExposedThing instances that exist in the same context.

Classes

ExposedThingSet()	Represents a group of ExposedThing objects.

class wotpy.wot.exposed.thing_set.**ExposedThingSet**
Bases: object

Represents a group of ExposedThing objects. A group cannot contain two ExposedThing with the same Thing ID.

add(*exposed_thing*)
Add a new ExposedThing to this set.

contains(*exposed_thing*)
Returns True if this group contains the given ExposedThing.

exposed_things
A generator that yields all the ExposedThing contained in this group.

find_by_interaction(*interaction*)
Finds the ExposedThing whose Thing contains the given Interaction.

find_by_thing_id(*thing_id*)
Finds an existing ExposedThing by Thing ID. The ID argument may be the original Thing ID or the URL-safe name (which is also unique and based on the ID).

remove(*thing_id*)
Removes an existing ExposedThing by ID. The thing_id argument may be the original Thing ID or the URL-safe name.

wotpy.wot.constants

Constants related to objects in the Thing hierarchy.

wotpy.wot.constants.**WOT_COMMON_CONTEXT_URL = 'https://w3c.github.io/wot/w3c-wot-common-cont**
W3C WoT common semantic context.

wotpy.wot.constants.**WOT_TD_CONTEXT_URL = 'https://w3c.github.io/wot/w3c-wot-td-context.jso**
W3C WoT TD semantic context.

wotpy.wot.enums

Classes that contain various enumerations.

Classes

DataType	Defines the types that values can take.
DefaultThingEvent	Enumeration for the default events that are supported on all ExposedThings.
DiscoveryMethod	Enumeration of discovery types.
InteractionTypes	Enumeration of interaction types.
SecuritySchemeType	Defines the supported security schemes.
TDChangeMethod	This attribute tells what operation has been applied to the TD: addition, removal or change.

Continued on next page

Table 23 – continued from previous page

TDChangeType	Represents the change type, whether has it been applied on properties, Actions or Events.

class wotpy.wot.enums.**DataType**
Bases: *wotpy.utils.enums.EnumListMixin*

Defines the types that values can take.

ARRAY = 'array'

BOOLEAN = 'boolean'

INTEGER = 'integer'

NULL = 'null'

NUMBER = 'number'

OBJECT = 'object'

STRING = 'string'

class wotpy.wot.enums.**DefaultThingEvent**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration for the default events that are supported on all ExposedThings.

ACTION_INVOCATION = 'actioninvocation'

DESCRIPTION_CHANGE = 'descriptionchange'

PROPERTY_CHANGE = 'propertychange'

class wotpy.wot.enums.**DiscoveryMethod**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of discovery types.

ANY = 'any'

DIRECTORY = 'directory'

LOCAL = 'local'

MULTICAST = 'multicast'

class wotpy.wot.enums.**InteractionTypes**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of interaction types.

ACTION = 'Action'

EVENT = 'Event'

PROPERTY = 'Property'

class wotpy.wot.enums.**SecuritySchemeType**
Bases: *wotpy.utils.enums.EnumListMixin*

Defines the supported security schemes.

APIKEY = 'apikey'

BASIC = 'basic'

BEARER = 'bearer'

CERT = 'cert'

DIGEST = 'digest'

NOSEC = 'nosec'

OAUTH2 = 'oauth2'

POP = 'pop'

PSK = 'psk'

PUBLIC = 'public'

class wotpy.wot.enums.**TDChangeMethod**
Bases: *wotpy.utils.enums.EnumListMixin*

This attribute tells what operation has been applied to the TD: addition, removal or change.

ADD = 'add'

CHANGE = 'change'

REMOVE = 'remove'

class wotpy.wot.enums.**TDChangeType**
Bases: *wotpy.utils.enums.EnumListMixin*

Represents the change type, whether has it been applied on properties, Actions or Events.

ACTION = 'action'

EVENT = 'event'

PROPERTY = 'property'

wotpy.wot.events

Classes that represent events that are emitted by Things.

Classes

ActionInvocationEmittedEvent(init)	Event triggered to indicate an action invocation.
ActionInvocationEventInit(action_name, ...)	Represents the data contained in an action invocation event.
EmittedEvent(init, name)	Base event class.
PropertyChangeEmittedEvent(init)	Event triggered to indicate a property change.
PropertyChangeEventInit(name, value)	Represents the data contained in a property update event.
ThingDescriptionChangeEmittedEvent(init)	Event triggered to indicate a thing description change.
ThingDescriptionChangeEventInit(..., [, data, ...])	Represents the data contained in a thing description update event.

class wotpy.wot.events.**ActionInvocationEmittedEvent**(*init*)
Bases: *wotpy.wot.events.EmittedEvent*

Event triggered to indicate an action invocation. Should be initialized with a ActionInvocationEventInit instance.

class wotpy.wot.events.**ActionInvocationEventInit**(*action_name*, *return_value*)
Bases: object

Represents the data contained in an action invocation event.

Parameters

- **action_name** (*str*) – Name of the property.
- **return_value** – Result returned by the action invocation.

class wotpy.wot.events.**EmittedEvent**(*init*, *name*)
Bases: object

Base event class. Represents a generic event defined in a TD.

data
Data property.

class wotpy.wot.events.**PropertyChangeEmittedEvent**(*init*)
Bases: *wotpy.wot.events.EmittedEvent*

Event triggered to indicate a property change. Should be initialized with a PropertyChangeEventInit instance.

class wotpy.wot.events.**PropertyChangeEventInit**(*name*, *value*)
Bases: object

Represents the data contained in a property update event.

Parameters

- **name** (*str*) – Name of the property.
- **value** – Value of the property.

class wotpy.wot.events.**ThingDescriptionChangeEmittedEvent**(*init*)
Bases: *wotpy.wot.events.EmittedEvent*

Event triggered to indicate a thing description change. Should be initialized with a ThingDescriptionChangeEventInit instance.

class wotpy.wot.events.**ThingDescriptionChangeEventInit**(*td_change_type*, *method*, *name*, *data=None*, *description=None*)
Bases: object

Represents the data contained in a thing description update event.

Parameters

- **td_change_type** (*str*) – An item of enumeration *TDChangeType*.
- **method** (*str*) – An item of enumeration *TDChangeMethod*.
- **name** (*str*) – Name of the Interaction.
- **data** – An instance of ThingPropertyInit, ThingActionInit or ThingEventInit (or None if the change did not add a new interaction).
- **description** (*dict*) – A dict that represents a TD serialized to JSON-LD.

wotpy.wot.form

Class that represents the form entities exposed by interactions.

Classes

Form(interaction, protocol[, form_dict])	Communication metadata where a service can be accessed by a client application.

class wotpy.wot.form.**Form**(*interaction*, *protocol*, *form_dict=None*, ***kwargs*)
Bases: object

Communication metadata where a service can be accessed by a client application.

form_dict
The Form dictionary of this Form.

id
Returns the ID of this Form. The ID is a hash that is based on the Form attributes. No two Forms with the same ID may exist within the same Interaction. The ID of a Form could change during its lifetime if some attributes are updated.

interaction
Interaction that contains this Form.

protocol
Form protocol.

wotpy.wot.interaction

Classes that represent all interaction patterns.

Classes

Action(thing, name[, init_dict])	Actions offer functions of the Thing.
Event(thing, name[, init_dict])	The Event Interaction Pattern describes event sources that asynchronously push messages.
InteractionPattern(thing, name[, init_dict])	A functionality exposed by Thing that is defined by the TD Interaction Model.
Property(thing, name[, init_dict])	Properties expose internal state of a Thing that can be directly accessed (get) and optionally manipulated (set).

class wotpy.wot.interaction.**Action**(*thing*, *name*, *init_dict=None*, ***kwargs*)
Bases: *wotpy.wot.interaction.InteractionPattern*

Actions offer functions of the Thing. These functions may manipulate the internal state of a Thing in a way that is not possible through setting Properties.

init_class
Returns the init dict class for this type of interaction.

interaction_type
Interaction type.

class wotpy.wot.interaction.**Event**(*thing*, *name*, *init_dict=None*, ***kwargs*)
Bases: *wotpy.wot.interaction.InteractionPattern*

The Event Interaction Pattern describes event sources that asynchronously push messages. Here not state, but state transitions (events) are communicated (e.g., clicked).

init_class
Returns the init dict class for this type of interaction.

interaction_type
Interaction type.

class wotpy.wot.interaction.**InteractionPattern**(*thing*, *name*, *init_dict=None*, ***kwargs*)
Bases: object

A functionality exposed by Thing that is defined by the TD Interaction Model.

add_form(*form*)
Add a new Form.

clean_forms()
Removes all the Forms from this Interaction.

forms
Sequence of forms linked to this interaction.

init_class
Returns the init dict class for this type of interaction.

interaction_fragment
The InteractionFragment dictionary of this interaction.

name
Interaction name. No two Interactions with the same name may exist in a Thing.

remove_form(*form*)
Remove an existing Form.

thing
Thing that contains this Interaction.

url_name
URL-safe version of the name.

class wotpy.wot.interaction.**Property**(*thing*, *name*, *init_dict=None*, ***kwargs*)
Bases: *wotpy.wot.interaction.InteractionPattern*

Properties expose internal state of a Thing that can be directly accessed (get) and optionally manipulated (set).

init_class
Returns the init dict class for this type of interaction.

interaction_type
Interaction type.

wotpy.wot.servient

Class that represents a WoT servient.

Classes

Servient([hostname, catalogue_port, ...])	An entity that is both a WoT client and server at the same time.
TDCatalogueHandler(application, request, ...)	Handler that returns the entire catalogue of Things contained in this servient.

Continued on next page

Table 27 – continued from previous page

TDHandler(application, request, **kwargs)	Handler that returns the TD document of a given Thing.

Exceptions

ServientStateException	Exception raised when the user modifies the Servient while the Servient is in an inappropriate state.

class wotpy.wot.servient.**Servient**(*hostname=None*, *catalogue_port=9090*, *clients=None*, *clients_config=None*, *dnssd_enabled=False*, *dnssd_instance_name=None*)

Bases: 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.

add_client(*client*)
Adds a new Protocol Binding client to this servient.

add_exposed_thing(*exposed_thing*)
Adds an ExposedThing to this Servient. ExposedThings are disabled by default.

add_server(*server*)
Adds a new Protocol Binding server to this servient.

catalogue_port
Returns the current port of the HTTP Thing Description catalogue service.

clients
Returns the dict of Protocol Binding clients attached to this servient.

disable_exposed_thing(*thing_id*)
Disables the ExposedThing with the given ID. This is, the servers will not listen for requests for this thing.

disable_td_catalogue()
Disables the servient TD catalogue.

dnssd
Returns the DNS-SD instance linked to this Servient (if enabled and started).

dnssd_instance_name
Returns the user-given DNS-SD service instance name.

enable_exposed_thing(*thing_id*)
Enables the ExposedThing with the given ID. This is, the servers will listen for requests for this thing.

enabled_exposed_things
Returns an iterator for the enabled ExposedThings contained in this Servient.

exposed_thing_set
Returns the ExposedThingSet instance that contains the ExposedThings of this servient.

exposed_things
Returns an iterator for the ExposedThings contained in this Servient.

get_exposed_thing(*thing_id*)
Finds and returns an ExposedThing contained in this servient by Thing ID. Raises ValueError if the ExposedThing is not present.

get_thing_base_url(*exposed_thing*)
Return the base URL for the given ExposedThing for one of the currently active servers.

hostname
Hostname attached to this servient.

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

refresh_forms()
Cleans and regenerates Forms for all the ExposedThings and servers contained in this servient.

remove_client(*protocol*)
Removes the Protocol Binding client with the given protocol from this servient.

remove_exposed_thing(*thing_id*)
Disables and removes an ExposedThing from this Servient.

remove_server(*protocol*)
Removes the Protocol Binding server with the given protocol from this servient.

select_client(*td*, *name*)
Returns the Protocol Binding client instance to communicate with the given Interaction.

servers
Returns the dict of Protocol Binding servers attached to this servient.

shutdown()
Stops the server configured under this servient.

start()
Starts the servers and returns an instance of the WoT object.

exception wotpy.wot.servient.**ServientStateException**
Bases: Exception

Exception raised when the user modifies the Servient while the Servient is in an inappropriate state.

class wotpy.wot.servient.**TDCatalogueHandler**(*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

Handler that returns the entire catalogue of Things contained in this servient. May return TDs in expanded format or URL pointers to the individual TDs.

get()

initialize(*servient*)
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...
```

(continues on next page)

(continued from previous page)

```
app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

class wotpy.wot.servient.**TDHandler**(*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

Handler that returns the TD document of a given Thing.

get(*thing_url_name*)

initialize(*servient*)
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

wotpy.wot.td

Classes that represent the JSON and JSON-LD serialization formats of a Thing Description document.

Classes

ThingDescription(doc)	Class that represents a Thing Description document.

class wotpy.wot.td.**ThingDescription**(*doc*)
Bases: object

Class that represents a Thing Description document. Contains logic to validate and transform a Thing to a serialized TD and vice versa.

build_thing()
Builds a new Thing object from the serialized Thing Description.

classmethod from_thing(*thing*)
Builds an instance of a JSON-serialized Thing Description from a Thing object.

get_action_forms(*name*)
Returns a list of FormDict for the action that matches the given name.

get_event_forms(*name*)
Returns a list of FormDict for the event that matches the given name.

get_forms(*name*)
Returns a list of FormDict for the interaction that matches the given name.

get_property_forms(*name*)
Returns a list of FormDict for the property that matches the given name.

to_dict()
Returns the JSON Thing Description as a dict.

to_str()
Returns the JSON Thing Description as a string.

to_thing_fragment()
Returns a ThingFragment dictionary built from this TD.

classmethod validate(*doc*)
Validates the given Thing Description document against its schema. Raises ValidationError if validation fails.

wotpy.wot.thing

Class that represents a Thing.

Classes

Thing([thing_fragment])	An abstraction of a physical or virtual entity whose metadata and interfaces are described by a WoT Thing Description.

class wotpy.wot.thing.**Thing**(*thing_fragment=None*, ***kwargs*)
Bases: object

An abstraction of a physical or virtual entity whose metadata and interfaces are described by a WoT Thing Description.

THING_FRAGMENT_WRITABLE_FIELDS = {'base', 'created', 'description', 'lastModified', 'l

actions
Actions interactions.

add_interaction(*interaction*)
Add a new Interaction.

events
Events interactions.

find_interaction(*name*)
Finds an existing Interaction by name. The name argument may be the original name or the URL-safe version.

id
Thing ID.

interactions
Sequence of interactions linked to this thing.

name
Thing name.

properties
Properties interactions.

remove_interaction(*name*)
Removes an existing Interaction by name. The name argument may be the original name or the URL-safe version.

thing_fragment
The ThingFragment dictionary of this Thing.

url_name
Returns the URL-safe name of this Thing. The URL name of a Thing is always unique and stable as long as the ID is unique.

uuid
Thing UUID in hex string format (e.g. a5220c5f-6bcb-4675-9c67-a2b1adc280b7). This value is deterministic and derived from the Thing ID. It may be of use when URL-unsafe chars are not acceptable.

wotpy.wot.validation

Schemas following the JSON Schema specification used to validate the shape of Thing Description documents.

Functions

interaction_schema_for_type(interaction_type)	Returns the JSON schema that describes an interaction for the given interaction type.
is_valid_safe_name(val)	Returns True if the given value is a safe machine-readable name.
is_valid_uri(val)	Returns True if the given value is a valid URI.

Exceptions

InvalidDescription	Exception raised when a document for an object in the TD hierarchy has an invalid format.

exception wotpy.wot.validation.**InvalidDescription**
Bases: Exception

Exception raised when a document for an object in the TD hierarchy has an invalid format.

wotpy.wot.validation.**interaction_schema_for_type**(*interaction_type*)
Returns the JSON schema that describes an interaction for the given interaction type.

wotpy.wot.validation.**is_valid_safe_name**(*val*)
Returns True if the given value is a safe machine-readable name.

wotpy.wot.validation.**is_valid_uri**(*val*)
Returns True if the given value is a valid URI.

wotpy.wot.wot

Class that serves as the WoT entrypoint.

Classes

WoT(servient)	The WoT object is the API entry point and it is exposed by an implementation of the WoT Runtime.

class wotpy.wot.wot.**WoT**(*servient*)

Bases: object

The WoT object is the API entry point and it is exposed by an implementation of the WoT Runtime. The WoT object does not expose properties, only methods for discovering, consuming and exposing a Thing.

consume(*td_str*)
Accepts a thing description string argument and returns a ConsumedThing object instantiated based on that description.

consume_from_url(*url*, *timeout_secs=None*)
Return a Future that resolves to a ConsumedThing created from the thing description retrieved from the given URL.

discover(*thing_filter*, *dnssd_find_kwargs=None*)
Starts the discovery process that will provide ThingDescriptions that match the optional argument filter of type ThingFilter.

classmethod fetch(*url*, *timeout_secs=None*)
Accepts an url argument and returns a Future that resolves with a Thing Description string.

produce(*model*)
Accepts a model argument of type ThingModel and returns an ExposedThing object, locally created based on the provided initialization parameters.

produce_from_url(*url*, *timeout_secs=None*)
Return a Future that resolves to an ExposedThing created from the thing description retrieved from the given URL.

register(*directory*, *thing*)
Generate the Thing Description as td, given the Properties, Actions and Events defined for this ExposedThing object. Then make a request to register td to the given WoT Thing Directory.

unregister(*directory*, *thing*)
Makes a request to unregister the thing from the given WoT Thing Directory.

1.1.2 wotpy.utils

Utility functions and classes.

wotpy.utils.enums	Utilities related to enumerations.
wotpy.utils.utils	Some utility functions for the WoT data type wrappers.

wotpy.utils.enums

Utilities related to enumerations.

Classes

EnumListMixin	Mixin that provides methods to list enumerated values.

class wotpy.utils.enums.**EnumListMixin**
Bases: object

Mixin that provides methods to list enumerated values.

classmethod list()
Returns a list of enumerated values.

wotpy.utils.utils

Some utility functions for the WoT data type wrappers.

Functions

get_main_ipv4_address()	Returns the main IPv4 address of the current machine in a portable fashion.
handle_observer_finalization(observer)	Builds a decorator that yields the wrapped coroutine and calls on_completed or on_error on the observer when the coroutine ends or raises an error.
merge_args_kwargs_dict(args, kwargs)	Takes a tuple of args and dict of kwargs.
to_camel(val)	Takes a string and transforms it to camelCase.
to_json_obj(obj)	Recursive function that attempts to convert any given object to a JSON-serializable object.
to_snake(val)	Takes a string and transforms it to snake_case.

wotpy.utils.utils.**get_main_ipv4_address**()
Returns the main IPv4 address of the current machine in a portable fashion. Attribution to the answer provided by Jamieson Becker on: https://stackoverflow.com/a/28950776

wotpy.utils.utils.**handle_observer_finalization**(*observer*)
Builds a decorator that yields the wrapped coroutine and calls on_completed or on_error on the observer when the coroutine ends or raises an error.

wotpy.utils.utils.**merge_args_kwargs_dict**(*args*, *kwargs*)
Takes a tuple of args and dict of kwargs. Returns a dict that is the result of merging the first item of args (if that item is a dict) and the kwargs dict.

wotpy.utils.utils.**to_camel**(*val*)
Takes a string and transforms it to camelCase.

wotpy.utils.utils.**to_json_obj**(*obj*)
Recursive function that attempts to convert any given object to a JSON-serializable object.

wotpy.utils.utils.**to_snake**(*val*)
Takes a string and transforms it to snake_case.

1.1.3 wotpy.protocols

Implementations of the supported Protocol Binding templates.

wotpy.protocols.coap	CoAP Protocol Binding implementation.
wotpy.protocols.http	HTTP Protocol Binding implementation.
wotpy.protocols.mqtt	MQTT Protocol Binding implementation.
wotpy.protocols.ws	WebSockets Protocol Binding implementation.
wotpy.protocols.client	Class that represents the abstract client interface.
wotpy.protocols.enums	Enumeration classes related to the various protocol servers.
wotpy.protocols.exceptions	Exceptions raised by the protocol binding implementations.
wotpy.protocols.server	Class that represents the abstract server interface.
wotpy.protocols.utils	Utility functions used by client and server implementations.

wotpy.protocols.coap

CoAP Protocol Binding implementation.

wotpy.protocols.coap.resources	CoAP resources that implement each of the Interaction verbs.
wotpy.protocols.coap.client	Classes that contain the client logic for the CoAP protocol.
wotpy.protocols.coap.enums	Enumeration classes related to the CoAP server.
wotpy.protocols.coap.server	Class that implements the CoAP server.

wotpy.protocols.coap.resources

CoAP resources that implement each of the Interaction verbs.

wotpy.protocols.coap.resources.action	CoAP resources to deal with Action interactions.
wotpy.protocols.coap.resources.event	CoAP resources to deal with Event interactions.
wotpy.protocols.coap.resources.property	CoAP resources to deal with Property interactions.
wotpy.protocols.coap.resources.utils	Utility functions for CoAP resources.

wotpy.protocols.coap.resources.action

CoAP resources to deal with Action interactions.

Functions

get_thing_action(server, request)	Takes a CoAP request and returns the Thing Action identified by the request arguments.

Classes

ActionResource(server[, clear_ms])	CoAP resource to invoke Actions and observe those invocations.

class wotpy.protocols.coap.resources.action.**ActionResource**(*server*, *clear_ms=None*)

Bases: aiocoap.resource.ObservableResource

CoAP resource to invoke Actions and observe those invocations.

DEFAULT_CLEAR_MS = 300000

add_observation(*request*, *server_observation*)
Method that decides whether to add a new observer. Observers are added for GET requests (checks for invocation status) but not for POST requests (action invocations).

render_get(*request*)
Handler to check the status of an ongoing invocation.

render_post(*request*)
Handler for action invocations.

wotpy.protocols.coap.resources.action.**get_thing_action**(*server*, *request*)
Takes a CoAP request and returns the Thing Action identified by the request arguments.

wotpy.protocols.coap.resources.event

CoAP resources to deal with Event interactions.

Functions

get_thing_event(server, request)	Takes a CoAP request and returns the Thing Event identified by the request arguments.

Classes

EventResource(server)	CoAP resource to observe Event emissions.

class wotpy.protocols.coap.resources.event.**EventResource**(*server*)
Bases: aiocoap.resource.ObservableResource

CoAP resource to observe Event emissions.

add_observation(*request*, *server_observation*)
Method that decides whether to add a new observer. A new observer is added for each GET request.

render_get(*request*)
Returns a CoAP response with the last observed event emission.

wotpy.protocols.coap.resources.event.**get_thing_event**(*server*, *request*)
Takes a CoAP request and returns the Thing Event identified by the request arguments.

wotpy.protocols.coap.resources.property

CoAP resources to deal with Property interactions.

Functions

get_thing_property(server, request)	Takes a CoAP request and returns the Thing Property identified by the request arguments.

Classes

PropertyResource(server)	CoAP resource that implements the Property read, write and observe verbs.

class wotpy.protocols.coap.resources.property.**PropertyResource**(*server*)
Bases: aiocoap.resource.Resource

CoAP resource that implements the Property read, write and observe verbs.

add_observation(*request*, *server_observation*)
Method that decides whether to add a new observer. A new observer is added for each GET request.

render_get(*request*)
Returns a CoAP response with the current property value.

render_put(*request*)
Updates the property with the value retrieved from the CoAP request payload.

wotpy.protocols.coap.resources.property.**get_thing_property**(*server*, *request*)
Takes a CoAP request and returns the Thing Property identified by the request arguments.

wotpy.protocols.coap.resources.utils

Utility functions for CoAP resources.

Functions

parse_request_opt_query(request)	Takes a CoAP Request and returns a dict containing the parsed URI query parameters.

wotpy.protocols.coap.resources.utils.**parse_request_opt_query**(*request*)
Takes a CoAP Request and returns a dict containing the parsed URI query parameters.

wotpy.protocols.coap.client

Classes that contain the client logic for the CoAP protocol.

Classes

CoAPClient()	Implementation of the protocol client interface for the CoAP protocol.

class wotpy.protocols.coap.client.**CoAPClient**

Bases: *wotpy.protocols.client.BaseProtocolClient*

Implementation of the protocol client interface for the CoAP protocol.

invoke_action(*td*, *name*, *input_value*, *timeout=None*)
Invokes an Action on a remote Thing.

is_supported_interaction(*td*, *name*)
Returns True if the any of the Forms for the Interaction with the given name is supported in this Protocol Binding client.

on_event(*td*, *name*)
Subscribes to an event on a remote Thing. Returns an Observable.

on_property_change(*td*, *name*)
Subscribes to property changes on a remote Thing. Returns an Observable

on_td_change(*url*)
Subscribes to Thing Description changes on a remote Thing. Returns an Observable.

protocol
Protocol of this client instance. A member of the Protocols enum.

read_property(*td*, *name*, *timeout=None*)
Reads the value of a Property on a remote Thing.

write_property(*td*, *name*, *value*, *timeout=None*)
Updates the value of a Property on a remote Thing.

wotpy.protocols.coap.enums

Enumeration classes related to the CoAP server.

Classes

CoAPSchemes	Enumeration of CoAP schemes.

class wotpy.protocols.coap.enums.**CoAPSchemes**

Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of CoAP schemes.

COAP = 'coap'

COAPS = 'coaps'

wotpy.protocols.coap.server

Class that implements the CoAP server.

Classes

CoAPServer([port, ssl_context, action_clear_ms])	CoAP binding server implementation.

class wotpy.protocols.coap.server.**CoAPServer**(*port=5683*, *ssl_context=None*, *action_clear_ms=None*)

Bases: *wotpy.protocols.server.BaseProtocolServer*

CoAP binding server implementation.

DEFAULT_PORT = 5683

action_clear_ms
Returns the timeout (ms) before completed actions are removed from the server.

build_base_url(*hostname*, *thing*)
Returns the base URL for the given Thing in the context of this server.

build_forms(*hostname*, *interaction*)
Builds and returns a list with all Form that are linked to this server for the given Interaction.

is_secure
Returns True if this server is configured to use SSL encryption.

protocol
Protocol of this server instance. A member of the Protocols enum.

scheme
Returns the URL scheme for this server.

start()
Starts the CoAP server.

stop()
Stops the CoAP server.

wotpy.protocols.http

HTTP Protocol Binding implementation.

wotpy.protocols.http.handlers	HTTP request handlers that implement each of the Interaction verbs.
wotpy.protocols.http.client	Classes that contain the client logic for the HTTP protocol.
wotpy.protocols.http.enums	Enumeration classes related to the HTTP server.
wotpy.protocols.http.server	Class that implements the HTTP server.

wotpy.protocols.http.handlers

HTTP request handlers that implement each of the Interaction verbs.

wotpy.protocols.http.handlers.action	Request handler for Action interactions.
wotpy.protocols.http.handlers.event	Request handler for Event interactions.

Continued on next page

Table 51 – continued from previous page

wotpy.protocols.http.handlers.property	Request handler for Property interactions.
wotpy.protocols.http.handlers.utils	Request handler for Property interactions.

wotpy.protocols.http.handlers.action

Request handler for Action interactions.

Classes

ActionInvokeHandler(application, request, ...)	Handler for Action invocation requests.
PendingInvocationHandler(application, ...)	Handler to check the status of pending action invocations.

class wotpy.protocols.http.handlers.action.**ActionInvokeHandler**(*application*, *request*, ***kwargs*)

Bases: tornado.web.RequestHandler

Handler for Action invocation requests.

initialize(*http_server*)

Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

post(*thing_name*, *name*)

Invokes the action and returns the invocation result.

class wotpy.protocols.http.handlers.action.**PendingInvocationHandler**(*application*, *request*, ***kwargs*)

Bases: tornado.web.RequestHandler

Handler to check the status of pending action invocations.

get(*invocation_id*)

Checks and returns the status of the Future that represents an action invocation.

initialize(*http_server*)

Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

wotpy.protocols.http.handlers.event

Request handler for Event interactions.

Classes

EventObserverHandler(application, request, ...)	Handler for Event subscription requests.

class wotpy.protocols.http.handlers.event.**EventObserverHandler**(*application*, *request*, ***kwargs*)

Bases: tornado.web.RequestHandler

Handler for Event subscription requests.

get(*thing_name*, *name*)
Subscribes to the given Event and waits for the next emission (HTTP long-polling pattern). Returns the event emission payload and destroys the subscription afterwards.

initialize(*http_server*)
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

on_finish()
Destroys the subscription to the observable when the request finishes.

wotpy.protocols.http.handlers.property

Request handler for Property interactions.

Classes

PropertyObserverHandler(application, ...)	Handler for Property subscription requests.
PropertyReadWriteHandler(application, ...)	Handler for Property get/set requests.

class wotpy.protocols.http.handlers.property.**PropertyObserverHandler**(*application*, *request*, ***kwargs*)

Bases: tornado.web.RequestHandler

Handler for Property subscription requests.

get(*thing_name*, *name*)
Subscribes to Property updates and waits for the next event (HTTP long-polling pattern). Returns the updated value and destroys the subscription.

initialize(*http_server*)
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

on_finish()
Destroys the subscription to the observable when the request finishes.

class wotpy.protocols.http.handlers.property.**PropertyReadWriteHandler**(*application*, *request*, ***kwargs*)

Bases: tornado.web.RequestHandler

Handler for Property get/set requests.

get(*thing_name*, *name*)
Reads and returns the Property value.

initialize(*http_server*)
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

put (*thing_name*, *name*)
Updates the Property value.

wotpy.protocols.http.handlers.utils

Request handler for Property interactions.

Functions

get_argument(req_handler, name[, default])	Returns an argument extracted from the request.
get_exposed_thing(server, thing_name)	Utility function to retrieve an ExposedThing from the HTTPServer or raise an HTTPError.

wotpy.protocols.http.handlers.utils.**get_argument** (*req_handler*, *name*, *default=None*)
Returns an argument extracted from the request. Interprets the body as JSON if the Content-Type is application/json. Reverts to the default Tornado get_argument otherwise.

wotpy.protocols.http.handlers.utils.**get_exposed_thing** (*server*, *thing_name*)
Utility function to retrieve an ExposedThing from the HTTPServer or raise an HTTPError.

wotpy.protocols.http.client

Classes that contain the client logic for the HTTP protocol.

Classes

HTTPClient([connect_timeout, request_timeout])	Implementation of the protocol client interface for the HTTP protocol.

class wotpy.protocols.http.client.**HTTPClient** (*connect_timeout=60*, *request_timeout=60*)
Bases: *wotpy.protocols.client.BaseProtocolClient*

Implementation of the protocol client interface for the HTTP protocol.

DEFAULT_CON_TIMEOUT = 60

DEFAULT_REQ_TIMEOUT = 60

JSON_HEADERS = {'Content-Type': 'application/json'}

connect_timeout
Returns the default connection timeout for all HTTP requests.

invoke_action(*td*, *name*, *input_value*, *timeout=None*)
Invokes an Action on a remote Thing. Returns a Future.

is_supported_interaction(*td*, *name*)
Returns True if the any of the Forms for the Interaction with the given name is supported in this Protocol Binding client.

on_event(*td*, *name*)
Subscribes to an event on a remote Thing. Returns an Observable.

on_property_change(*td*, *name*)
Subscribes to property changes on a remote Thing. Returns an Observable

on_td_change(*url*)
Subscribes to Thing Description changes on a remote Thing. Returns an Observable.

classmethod pick_http_href(*td*, *forms*, *op=None*)
Picks the most appropriate HTTP form href from the given list of forms.

protocol
Protocol of this client instance. A member of the Protocols enum.

read_property(*td*, *name*, *timeout=None*)
Reads the value of a Property on a remote Thing. Returns a Future.

request_timeout
Returns the default request timeout for all HTTP requests.

write_property(*td*, *name*, *value*, *timeout=None*)
Updates the value of a Property on a remote Thing. Returns a Future.

wotpy.protocols.http.enums

Enumeration classes related to the HTTP server.

Classes

HTTPSchemes	Enumeration of HTTP schemes.

class wotpy.protocols.http.enums.**HTTPSchemes**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of HTTP schemes.

HTTP = 'http'

HTTPS = 'https'

wotpy.protocols.http.server

Class that implements the HTTP server.

Classes

HTTPServer([port, ssl_context, action_ttl_secs])	HTTP binding server implementation.

class wotpy.protocols.http.server.**HTTPServer**(*port=80*, *ssl_context=None*, *action_ttl_secs=300*)

Bases: *wotpy.protocols.server.BaseProtocolServer*

HTTP binding server implementation.

DEFAULT_PORT = 80

action_ttl
Returns the Action invocations Time-To-Live (seconds).

app
Tornado application.

build_base_url(*hostname*, *thing*)
Returns the base URL for the given Thing in the context of this server.

build_forms(*hostname*, *interaction*)
Builds and returns a list with all Form that are linked to this server for the given Interaction.

invocation_check_times
Dict that contains the timestamp of the last time an invocation was checked by a client..

is_secure
Returns True if this server is configured to use SSL encryption.

pending_actions
Dict of pending action invocations represented as Futures.

protocol
Protocol of this server instance. A member of the Protocols enum.

scheme
Returns the URL scheme for this server.

start()
Starts the HTTP server.

stop()
Stops the HTTP server.

wotpy.protocols.mqtt

MQTT Protocol Binding implementation.

wotpy.protocols.mqtt.handlers	Entities that handle the MQTT operations needed to support each of the Interaction verbs.
wotpy.protocols.mqtt.client	Classes that contain the client logic for the MQTT protocol.
wotpy.protocols.mqtt.enums	Enumeration classes related to the MQTT protocol binding.
wotpy.protocols.mqtt.runner	Base class for MQTT handlers.
wotpy.protocols.mqtt.server	Class that implements the MQTT server (broker).

wotpy.protocols.mqtt.handlers

Entities that handle the MQTT operations needed to support each of the Interaction verbs.

wotpy.protocols.mqtt.handlers.action	MQTT handler for Action invocations.
wotpy.protocols.mqtt.handlers.base	Base class for all MQTT handlers.
wotpy.protocols.mqtt.handlers.event	MQTT handler for Event subscriptions.
wotpy.protocols.mqtt.handlers.ping	MQTT handler for PING requests published on the MQTT broker.
wotpy.protocols.mqtt.handlers.property	MQTT handler for Property reads, writes and subscriptions to value updates.
wotpy.protocols.mqtt.handlers.subs	Class that subscribes to all the Interactions of one kind for all the ExposedThings contained by a Protocol Binding server.

wotpy.protocols.mqtt.handlers.action

MQTT handler for Action invocations.

Classes

ActionMQTTHandler(mqtt_server[, qos])	MQTT handler for Action invocations.

class wotpy.protocols.mqtt.handlers.action.**ActionMQTTHandler**(*mqtt_server*, *qos=2*)

Bases: *wotpy.protocols.mqtt.handlers.base.BaseMQTTHandler*

MQTT handler for Action invocations.

KEY_INPUT = 'input'

KEY_INVOCATION_ID = 'id'

build_action_result_topic(*thing*, *action*)
Returns the MQTT topic for Action invocation results.

handle_message(*msg*)
Listens to all Property request topics and responds to read and write requests.

classmethod to_result_topic(*invocation_topic*)
Takes an Action invocation MQTT topic and returns the related result topic.

topic_wildcard_invocation
Wildcard topic to subscribe to all Action invocations.

topics
List of topics that this MQTT handler wants to subscribe to.

wotpy.protocols.mqtt.handlers.base

Base class for all MQTT handlers.

Classes

BaseMQTTHandler(mqtt_server)	Base class for all MQTT handlers.

class wotpy.protocols.mqtt.handlers.base.**BaseMQTTHandler**(*mqtt_server*)

Bases: object

Base class for all MQTT handlers.

handle_message(*msg*)
Called each time the runner receives a message for one of the handler topics.

init()
Initializes the MQTT handler. Called when the MQTT runner starts.

mqtt_server
MQTT server that contains this handler.

queue
Asynchronous queue where the handler leaves messages that should be published later by the runner.

servient_id
Servient ID that is used to avoid topic collisions øwhen multiple Servients are connected to the same broker.

teardown()
Destroys the MQTT handler. Called when the MQTT runner stops.

topics
List of topics that this MQTT handler wants to subscribe to.

wotpy.protocols.mqtt.handlers.event

MQTT handler for Event subscriptions.

Classes

EventMQTTHandler(mqtt_server[, qos, callback_ms])	MQTT handler for Event subscriptions.

class wotpy.protocols.mqtt.handlers.event.**EventMQTTHandler**(*mqtt_server*, *qos=0*, *callback_ms=None*)

Bases: *wotpy.protocols.mqtt.handlers.base.BaseMQTTHandler*

MQTT handler for Event subscriptions.

DEFAULT_CALLBACK_MS = 2000

DEFAULT_JITTER = 0.2

build_event_topic(*thing*, *event*)
Returns the MQTT topic for Event emissions.

init()
Initializes the MQTT handler. Called when the MQTT runner starts.

teardown()
Destroys the MQTT handler. Called when the MQTT runner stops.

wotpy.protocols.mqtt.handlers.ping

MQTT handler for PING requests published on the MQTT broker.

Classes

PingMQTTHandler(mqtt_server[, qos])	MQTT handler for PING requests published on the MQTT broker.

class wotpy.protocols.mqtt.handlers.ping.**PingMQTTHandler**(*mqtt_server*, *qos=1*)

Bases: *wotpy.protocols.mqtt.handlers.base.BaseMQTTHandler*

MQTT handler for PING requests published on the MQTT broker.

handle_message(*msg*)
Publishes a message in the PONG topic with the same payload as the one received in the PING topic.

topic_ping
Ping topic.

topic_pong
Pong topic.

topics
List of topics that this MQTT handler wants to subscribe to.

wotpy.protocols.mqtt.handlers.property

MQTT handler for Property reads, writes and subscriptions to value updates.

Classes

PropertyMQTTHandler(mqtt_server[, ...])	MQTT handler for Property reads, writes and subscriptions to value updates.

class wotpy.protocols.mqtt.handlers.property.**PropertyMQTTHandler**(*mqtt_server*, *qos_observe=0*, *qos_rw=2*, *callback_ms=None*)

Bases: *wotpy.protocols.mqtt.handlers.base.BaseMQTTHandler*

MQTT handler for Property reads, writes and subscriptions to value updates.

ACTION_READ = 'read'

ACTION_WRITE = 'write'

DEFAULT_CALLBACK_MS = 2000

DEFAULT_JITTER = 0.2

KEY_ACK = 'ack'

KEY_ACTION = 'action'

KEY_VALUE = 'value'

build_property_updates_topic(*thing*, *prop*)
Returns the MQTT topic for Property updates.

handle_message(*msg*)
Listens to all Property request topics and responds to read and write requests.

init()
Initializes the MQTT handler. Called when the MQTT runner starts.

publish_write_ack(*msg*)
Takes a Property write request message and publishes the related write ACK message.

teardown()
Destroys the MQTT handler. Called when the MQTT runner stops.

classmethod to_write_ack_topic(*requests_topic*)
Takes a Property requests topic and returns the related write ACK topic.

topic_wildcard_requests
Wildcard topic to subscribe to all Property requests.

topics
List of topics that this MQTT handler wants to subscribe to.

wotpy.protocols.mqtt.handlers.subs

Class that subscribes to all the Interactions of one kind for all the ExposedThings contained by a Protocol Binding server.

Classes

InteractionsSubscriber(interaction_type, ...)	Class that subscribes to all the Interactions of one kind for all the ExposedThings contained by a Protocol Binding server.

class wotpy.protocols.mqtt.handlers.subs.**InteractionsSubscriber**(*interaction_type*, *server*, *on_next_builder*)

Bases: object

Class that subscribes to all the Interactions of one kind for all the ExposedThings contained by a Protocol Binding server.

dispose()
Disposes of all the currently active subscriptions.

refresh()
Refresh all subscriptions for the entire set of ExposedThings.

wotpy.protocols.mqtt.client

Classes that contain the client logic for the MQTT protocol.

Classes

MQTTClient([deliver_timeout_secs, ...])	Implementation of the protocol client interface for the MQTT protocol.

class wotpy.protocols.mqtt.client.**MQTTClient**(*deliver_timeout_secs=1*, *msg_wait_timeout_secs=5*, *msg_ttl_secs=15*, *timeout_default=None*, *hbmqtt_config=None*, *stop_loop_timeout_secs=60*)

Bases: *wotpy.protocols.client.BaseProtocolClient*

Implementation of the protocol client interface for the MQTT protocol.

DEFAULT_CLIENT_CONFIG = {'keep_alive': 90}

DEFAULT_DELIVER_TIMEOUT_SECS = 1

DEFAULT_MSG_TTL_SECS = 15

DEFAULT_MSG_WAIT_TIMEOUT_SECS = 5

DEFAULT_STOP_LOOP_TIMEOUT_SECS = 60

DELIVER_TERMINATE_LOOP_SLEEP_SECS = 0.1

SLEEP_SECS_DELIVER_ERR = 1.0

invoke_action(*td*, *name*, *input_value*, *timeout=None*, *qos_publish=2*, *qos_subscribe=1*)
Invokes an Action on a remote Thing. Returns a Future.

is_supported_interaction(*td*, *name*)
Returns True if the any of the Forms for the Interaction with the given name is supported in this Protocol Binding client.

on_event(*td*, *name*, *qos=0*)
Subscribes to an event on a remote Thing. Returns an Observable.

on_property_change(*td*, *name*, *qos=0*)
Subscribes to property changes on a remote Thing. Returns an Observable

on_td_change(*url*)
Subscribes to Thing Description changes on a remote Thing. Returns an Observable.

protocol
Protocol of this client instance. A member of the Protocols enum.

read_property(*td*, *name*, *timeout=None*, *qos_publish=1*, *qos_subscribe=1*)
Reads the value of a Property on a remote Thing. Returns a Future.

write_property(*td*, *name*, *value*, *timeout=None*, *qos_publish=2*, *qos_subscribe=1*, *wait_ack=True*)
Updates the value of a Property on a remote Thing. Due to the MQTT binding design this coroutine yields as soon as the write message has been published and will not wait for a custom write handler that yields to another coroutine Returns a Future.

wotpy.protocols.mqtt.enums

Enumeration classes related to the MQTT protocol binding.

Classes

MQTTCodesACK	Enumeration of MQTT ACK codes.
MQTTCommandCodes	Enumeration of MQTT packet types.
MQTTQoSLevels	Enumeration of MQTT Quality of Service levels.
MQTTSchemes	Enumeration of MQTT schemes.
MQTTVocabularyKeys	Enumeration of terms that form the MQTT vocabulary that may appear in TD Form elements.

class wotpy.protocols.mqtt.enums.**MQTTCodesACK**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of MQTT ACK codes.

CON_OK = 0

SUB_ERROR = 128

class wotpy.protocols.mqtt.enums.**MQTTCommandCodes**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of MQTT packet types.

PUBLISH = 3

SUBSCRIBE = 8

UNSUBSCRIBE = 10

class wotpy.protocols.mqtt.enums.**MQTTQoSLevels**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of MQTT Quality of Service levels.

AT_LEAST_ONCE = 1

EXACTLY_ONCE = 2

FIRE_FORGET = 0

class wotpy.protocols.mqtt.enums.**MQTTSchemes**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of MQTT schemes.

MQTT = 'mqtt'

class wotpy.protocols.mqtt.enums.**MQTTVocabularyKeys**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of terms that form the MQTT vocabulary that may appear in TD Form elements.

COMMAND_CODE = 'mqtt:commandCode'

OPTIONS = 'mqtt:options'

OPTION_NAME = 'mqtt:optionName'

OPTION_NAME_DUP = 'mqtt:dup'

OPTION_NAME_QOS = 'mqtt:qos'

OPTION_NAME_RETAIN = 'mqtt:retain'

OPTION_VALUE = 'mqtt:optionValue'

wotpy.protocols.mqtt.runner

Base class for MQTT handlers.

Classes

MQTTHandlerRunner(broker_url, mqtt_handler)	Class that wraps an MQTT handler.

class wotpy.protocols.mqtt.runner.**MQTTHandlerRunner**(*broker_url*, *mqtt_handler*, *messages_buffer_size=500*, *timeout_loops=0.1*, *sleep_error_reconnect=2.0*, *hbmqtt_config=None*)

Bases: object

Class that wraps an MQTT handler. It handles connections to the MQTT broker, delivers messages, and runs the handler in a loop.

DEFAULT_CLIENT_CONFIG = {'keep_alive': 90}

DEFAULT_MSGS_BUF_SIZE = 500

DEFAULT_SLEEP_ERR_RECONN = 2.0

DEFAULT_TIMEOUT_LOOPS_SECS = 0.1

connect(*force_reconnect=False*)
Connects to the MQTT broker.

disconnect()
Disconnects from the MQTT broker.

start()
Starts listening for published messages.

stop()
Stops listening for published messages.

wotpy.protocols.mqtt.server

Class that implements the MQTT server (broker).

Classes

MQTTServer(broker_url[, ...])	MQTT binding server implementation.

class wotpy.protocols.mqtt.server.**MQTTServer**(*broker_url*, *property_callback_ms=None*, *event_callback_ms=None*, *servient_id=None*)

Bases: *wotpy.protocols.server.BaseProtocolServer*

MQTT binding server implementation.

DEFAULT_SERVIENT_ID = 'wotpy'

build_base_url(*hostname*, *thing*)
Returns the base URL for the given Thing in the context of this server.

build_forms(*hostname*, *interaction*)
Builds and returns a list with all Forms that are linked to this server for the given Interaction.

protocol
Protocol of this server instance. A member of the Protocols enum.

servient_id
Servient ID that is used to avoid topic collisions øwhen multiple Servients are connected to the same broker.

start()
Starts the MQTT broker and all the MQTT clients that handle the WoT clients requests.

stop()
Stops the MQTT broker and the MQTT clients.

wotpy.protocols.ws

WebSockets Protocol Binding implementation.

wotpy.protocols.ws.client	Classes that contain the client logic for the Websocket protocol.
wotpy.protocols.ws.enums	Enumeration classes related to the WebSockets server.
wotpy.protocols.ws.handler	Class that handles incoming WebSockets messages.
wotpy.protocols.ws.messages	Classes that represent JSON-RPC messages exchanged over WebSockets.
wotpy.protocols.ws.schemas	Schemas following the JSON Schema specification used to validate the shape of WebSockets messages.
wotpy.protocols.ws.server	Class that implements the WebSockets server.

wotpy.protocols.ws.client

Classes that contain the client logic for the Websocket protocol.

Classes

WebsocketClient([receive_timeout_secs, ...])	Implementation of the protocol client interface for the Websocket protocol.

class wotpy.protocols.ws.client.**WebsocketClient**(*receive_timeout_secs=1.0*, *ping_interval=2000*)
Bases: *wotpy.protocols.client.BaseProtocolClient*

Implementation of the protocol client interface for the Websocket protocol.

RECEIVE_LOOP_TERMINATE_SLEEP_SECS = 0.1

SLEEP_AFTER_ERR_SECS = 1.0

invoke_action(*td*, *name*, *input_value*, *timeout=None*)
Invokes an Action on a remote Thing. Returns a Future.

is_supported_interaction(*td*, *name*)
Returns True if the any of the Forms for the Interaction with the given name is supported in this Protocol Binding client.

on_event(*td*, *name*)
Subscribes to an event on a remote Thing. Returns an Observable.

on_property_change(*td*, *name*)
Subscribes to property changes on a remote Thing. Returns an Observable.

on_td_change(*url*)
Subscribes to Thing Description changes on a remote Thing. Returns an Observable.

protocol
Protocol of this client instance. A member of the Protocols enum.

read_property(*td*, *name*, *timeout=None*)
Reads the value of a Property on a remote Thing. Returns a Future.

write_property(*td*, *name*, *value*, *timeout=None*)
Updates the value of a Property on a remote Thing. Returns a Future.

wotpy.protocols.ws.enums

Enumeration classes related to the WebSockets server.

Classes

WebsocketErrors	Enumeration of JSON RPC error codes.
WebsocketMethods	Enumeration of available websocket message actions.
WebsocketSchemes	Enumeration of Websocket schemes.

class wotpy.protocols.ws.enums.**WebsocketErrors**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of JSON RPC error codes.

INTERNAL_ERROR = -32603

INVALID_METHOD_PARAMS = -32602

INVALID_REQUEST = -32600

METHOD_NOT_FOUND = -32601

PARSE_ERROR = -32700

SUBSCRIPTION_ERROR = -32000

class wotpy.protocols.ws.enums.**WebsocketMethods**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of available websocket message actions.

DISPOSE = 'dispose'

INVOKE_ACTION = 'invoke_action'

ON_EVENT = 'on_event'

ON_PROPERTY_CHANGE = 'on_property_change'

ON_TD_CHANGE = 'on_td_change'

READ_PROPERTY = 'read_property'

WRITE_PROPERTY = 'write_property'

class wotpy.protocols.ws.enums.**WebsocketSchemes**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of Websocket schemes.

WS = 'ws'

WSS = 'wss'

wotpy.protocols.ws.handler

Class that handles incoming WebSockets messages.

Classes

WebsocketHandler(*args, **kwargs)	Tornado handler for Websocket messages.

class wotpy.protocols.ws.handler.**WebsocketHandler**(**args*, ***kwargs*)
Bases: tornado.websocket.WebSocketHandler

Tornado handler for Websocket messages. This class processes all incoming WebSocket messages and translates them to actions executed on ExposedThing objects.

POLICY_VIOLATION_CODE = 1008

POLICY_VIOLATION_REASON = 'Not found'

check_origin(*origin*)
Should return True to accept the request or False to reject it. The origin argument is the value of the Origin HTTP header, the url responsible for initiating this request

exposed_thing
Exposed thing property. Retrieves the ExposedThing from the parent server.

on_close()
Called when the WebSockets connection is closed.

on_message(*message*)
Called each time the server receives a WebSockets message. All messages that do not conform to the protocol are discarded.

open(*name*)
Called when the WebSockets connection is opened.

wotpy.protocols.ws.messages

Classes that represent JSON-RPC messages exchanged over WebSockets.

Functions

parse_ws_message(raw_msg)	Takes a raw WebSockets message and attempts to parse it to create a message instance.

Classes

WebsocketMessageEmittedItem(subscription_id, ...)	Represents a Websockets message for an item emitted by an active subscription.
WebsocketMessageError(message[, code, data, ...])	Represents a WoT Websockets JSON-RPC error message.
WebsocketMessageRequest(method, params[, msg_id])	Represents a message received on a websocket that contains a JSON-RPC WoT action request.
WebsocketMessageResponse(result[, msg_id])	Represents a WoT Websockets JSON-RPC response message.

Exceptions

WebsocketMessageException	Exception raised when a WS message appears to be invalid.

class wotpy.protocols.ws.messages.**WebsocketMessageEmittedItem**(*subscription_id*, *name*, *data*)

Bases: object

Represents a Websockets message for an item emitted by an active subscription.

classmethod from_raw(*raw_msg*)
Builds a new WebsocketMessageEmittedItem instance from a raw socket message. Raises WebsocketMessageException if the message is invalid.

to_dict()
Returns this message as a dict.

to_json()
Returns this message as a JSON string.

class wotpy.protocols.ws.messages.**WebsocketMessageError**(*message*, *code=-32603*, *data=None*, *msg_id=None*)

Bases: object

Represents a WoT Websockets JSON-RPC error message.

classmethod from_raw(*raw_msg*)
Builds a new WebsocketMessageError instance from a raw socket message. Raises WebsocketMessageException if the message is invalid.

id
ID property.

to_dict()
Returns this message as a dict.

to_json()
Returns this message as a JSON string.

exception wotpy.protocols.ws.messages.**WebsocketMessageException**
Bases: Exception

Exception raised when a WS message appears to be invalid.

class wotpy.protocols.ws.messages.**WebsocketMessageRequest**(*method*, *params*, *msg_id=None*)
Bases: object

Represents a message received on a websocket that contains a JSON-RPC WoT action request.

classmethod from_raw(*raw_msg*)
Builds a new WebsocketMessageRequest instance from a raw socket message. Raises WebsocketMessageException if the message is invalid.

id
ID property.

to_dict()
Returns this message as a dict.

to_json()
Returns this message as a JSON string.

class wotpy.protocols.ws.messages.**WebsocketMessageResponse**(*result*, *msg_id=None*)
Bases: object

Represents a WoT Websockets JSON-RPC response message.

classmethod from_raw(*raw_msg*)
Builds a new WebsocketMessageResponse instance from a raw socket message. Raises WebsocketMessageException if the message is invalid.

id
ID property.

to_dict()
Returns this message as a dict.

to_json()
Returns this message as a JSON string.

wotpy.protocols.ws.messages.**parse_ws_message**(*raw_msg*)
Takes a raw WebSockets message and attempts to parse it to create a message instance.

wotpy.protocols.ws.schemas

Schemas following the JSON Schema specification used to validate the shape of WebSockets messages.

wotpy.protocols.ws.server

Class that implements the WebSockets server.

Classes

WebsocketServer([port, ssl_context])	WebSockets binding server implementation.

class wotpy.protocols.ws.server.**WebsocketServer**(*port=81*, *ssl_context=None*)
Bases: *wotpy.protocols.server.BaseProtocolServer*

WebSockets binding server implementation. Builds a Tornado application that uses the WebsocketHandler handler to process WebSockets messages.

DEFAULT_PORT = 81

app
Tornado application property.

build_base_url(*hostname*, *thing*)
Returns the base URL for the given Thing in the context of this server.

build_forms(*hostname*, *interaction*)
Builds and returns a list with all Form that are linked to this server for the given Interaction.

is_secure
Returns True if this server is configured to use SSL encryption.

protocol
Protocol of this server instance. A member of the Protocols enum.

scheme
Returns the URL scheme for this server.

start()
Starts the WebSockets server.

stop()
Stops the WebSockets server.

wotpy.protocols.client

Class that represents the abstract client interface.

Classes

BaseProtocolClient	Base protocol client class.

class wotpy.protocols.client.**BaseProtocolClient**
Bases: object

Base protocol client class. This is the interface that must be implemented by all client classes.

invoke_action(*td*, *name*, *input_value*, *timeout=None*)
Invokes an Action on a remote Thing. Returns a Future.

is_supported_interaction(*td*, *name*)
Returns True if the any of the Forms for the Interaction with the given name is supported in this Protocol Binding client.

on_event(*td*, *name*)
Subscribes to an event on a remote Thing. Returns an Observable.

on_property_change(*td*, *name*)
Subscribes to property changes on a remote Thing. Returns an Observable

on_td_change(*url*)
Subscribes to Thing Description changes on a remote Thing. Returns an Observable.

protocol
Protocol of this client instance. A member of the Protocols enum.

read_property(*td*, *name*, *timeout=None*)
Reads the value of a Property on a remote Thing. Returns a Future.

write_property(*td*, *name*, *value*, *timeout=None*)
Updates the value of a Property on a remote Thing. Returns a Future.

wotpy.protocols.enums

Enumeration classes related to the various protocol servers.

Classes

InteractionVerbs	Interactions have one or more defined interaction verbs for each interaction pattern.
Protocols	Enumeration of protocol types.

class wotpy.protocols.enums.**InteractionVerbs**
Bases: *wotpy.utils.enums.EnumListMixin*

Interactions have one or more defined interaction verbs for each interaction pattern. Form Relations allow an interaction to have separate protocol mechanisms to support different interaction verbs.

INVOKE_ACTION = 'invokeaction'

OBSERVE_PROPERTY = 'observeproperty'

READ_PROPERTY = 'readproperty'

SUBSCRIBE_EVENT = 'subscribeevent'

UNSUBSCRIBE_EVENT = 'unsubscribeevent'

WRITE_PROPERTY = 'writeproperty'

class wotpy.protocols.enums.**Protocols**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of protocol types.

COAP = 'COAP'

HTTP = 'HTTP'

MQTT = 'MQTT'

WEBSOCKETS = 'WEBSOCKETS'

wotpy.protocols.exceptions

Exceptions raised by the protocol binding implementations.

Exceptions

ClientRequestTimeout(*args, **kwargs)	Exception raised when a protocol client request reaches the timeout.
FormNotFoundException(*args, **kwargs)	Exception raised when a form for a given protocol binding could not be found in a Thing Description.
ProtocolClientException(*args, **kwargs)	Base Exceptions raised by clients of the protocol binding implementations.

exception wotpy.protocols.exceptions.**ClientRequestTimeout**(**args*, ***kwargs*)
Bases: *wotpy.protocols.exceptions.ProtocolClientException*

Exception raised when a protocol client request reaches the timeout.

DEFAULT_MSG = 'Timeout in protocol client request'

exception wotpy.protocols.exceptions.**FormNotFoundException**(**args*, ***kwargs*)
Bases: *wotpy.protocols.exceptions.ProtocolClientException*

Exception raised when a form for a given protocol binding could not be found in a Thing Description.

DEFAULT_MSG = 'Protocol Form not found in TD'

exception wotpy.protocols.exceptions.**ProtocolClientException**(**args*, ***kwargs*)
Bases: Exception

Base Exceptions raised by clients of the protocol binding implementations.

DEFAULT_MSG = 'Protocol client error'

wotpy.protocols.server

Class that represents the abstract server interface.

Classes

BaseProtocolServer(port)	Base protocol server class.

class wotpy.protocols.server.**BaseProtocolServer**(*port*)
Bases: object

Base protocol server class. This is the interface that must be implemented by all server classes.

add_codec(*codec*)
Adds a BaseCodec to this server.

add_exposed_thing(*exposed_thing*)
Adds the given ExposedThing to this server.

build_base_url(*hostname*, *thing*)
Returns the base URL for the given Thing in the context of this server.

build_forms(*hostname*, *interaction*)
Builds and returns a list with all Form that are linked to this server for the given Interaction.

codec_for_media_type(*media_type*)
Returns a BaseCodec to serialize or deserialize content for the given media type.

exposed_thing_set
Returns the ExposedThingSet instance that contains the ExposedThings of this server.

exposed_things
Returns an iterator for all the ExposedThings contained in this server.

get_exposed_thing(*name*)
Finds and returns an ExposedThing contained in this server by name. Raises ValueError if the ExposedThing is not present.

port
Port property.

protocol
Server protocol.

remove_exposed_thing(*thing_id*)
Removes the given ExposedThing from this server.

start()
Coroutine that starts the server.

stop()
Coroutine that stops the server. Some requests could be still in progress and would be served after the server has stopped.

wotpy.protocols.utils

Utility functions used by client and server implementations.

Functions

is_scheme_form(form, base, scheme)	Returns True if the scheme of the URI for the given Form matches the scheme argument.
pick_form(td, forms, schemes[, op])	Picks the Form that will be used to connect to the remote Thing.

wotpy.protocols.utils.**is_scheme_form**(*form*, *base*, *scheme*)
Returns True if the scheme of the URI for the given Form matches the scheme argument.

wotpy.protocols.utils.**pick_form**(*td*, *forms*, *schemes*, *op=None*)
Picks the Form that will be used to connect to the remote Thing.

1.1.4 wotpy.codecs

Classes to serialize and deserialize messages.

wotpy.codecs.base	Class that represents the codec interface.
wotpy.codecs.enums	Enumeration classes related to codecs.
wotpy.codecs.json_codec	Class that implements the JSON codec.
wotpy.codecs.text	Class that implements the text codec.

wotpy.codecs.base

Class that represents the codec interface.

Classes

BaseCodec	Base codec abstract class.

class wotpy.codecs.base.**BaseCodec**
Bases: object

Base codec abstract class. All codecs must implement this interface.

media_types
Property getter for the supported media types of this codec.

to_bytes(*value*)
Takes a Python object and encodes it to an UTF8 bytes string to be included in a response.

to_value(*value*)
Takes an encoded value from a request that may be an UTF8 bytes or unicode string and decodes it to a Python object.

wotpy.codecs.enums

Enumeration classes related to codecs.

Classes

MediaTypes	Enumeration of media types.

class wotpy.codecs.enums.**MediaTypes**
Bases: *wotpy.utils.enums.EnumListMixin*

Enumeration of media types.

JSON = 'application/json'

TEXT = 'text/plain'

wotpy.codecs.json_codec

Class that implements the JSON codec.

Classes

JsonCodec	JSON codec class.

class wotpy.codecs.json_codec.**JsonCodec**
Bases: *wotpy.codecs.base.BaseCodec*

JSON codec class.

media_types
Returns the JSON media types.

to_bytes(*value*)
Takes an object and serializes it to an UTF8 bytes JSON string.

to_value(*value*)
Takes an encoded value from a request that may be an UTF8 bytes or unicode JSON string and deserializes it to a Python object.

wotpy.codecs.text

Class that implements the text codec.

Classes

TextCodec	Text codec class.

class wotpy.codecs.text.**TextCodec**
Bases: *wotpy.codecs.base.BaseCodec*

Text codec class.

media_types
Returns the text media types.

to_bytes(*value*)
Takes an unicode string and encodes it to an UTF8 bytes string.

to_value(*value*)
Takes an encoded value from a request that may be a UTF8 bytes or unicode string and decodes it to an unicode string.

Referencias

[1] A. Al-Fuqaha y col. «Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications». En: *IEEE Communications Surveys Tutorials* 17.4 (Fourthquarter 2015), págs. 2347-2376. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[2] Flavio Bonomi y col. «Fog Computing and Its Role in the Internet of Things». En: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. New York, NY, USA: ACM, 2012, págs. 13-16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513.

[3] W. Yu y col. «A Survey on the Edge Computing for the Internet of Things». En: *IEEE Access* 6 (2018), págs. 6900-6919. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2778504.

[4] Peter Newman. *The Internet of Things 2020*. Inf. téc. Business Insider, mar. de 2020. URL: https://store.businessinsider.com/products/the-internet-of-things-report.

[5] D. Guinard, V. Trifa y E. Wilde. «A Resource Oriented Architecture for the Web of Things». En: *2010 Internet of Things (IOT)*. Nov. de 2010, págs. 1-8. DOI: 10.1109/IOT.2010.5678452.

[6] D. Guinard y col. «Towards Physical Mashups in the Web of Things». En: *2009 Sixth International Conference on Networked Sensing Systems (INSS)*. Jun. de 2009, págs. 1-4. DOI: 10.1109/INSS.2009.5409925.

[7] Dominique Guinard. «A Web of Things Application Architecture – Integrating the Real-World into the Web». Tesis doct. Zurich, Switzerland: ETH Zurich, ago. de 2011.

[8] Matthias Kovatsch y col. *Web of Things (WoT) Architecture*. W3C Recommendation. W3C, abr. de 2020. URL: https://www.w3.org/TR/wot-architecture10/ (visitado 17-05-2022).

[9] Open Connectivity Foundation. *OCF Core Specification*. Inf. téc. Version 2.0.1. Open Connectivity Foundation, feb. de 2019. URL: https://openconnectivity.org/specs/OCF_Core_Specification_v2.0.1.pdf (visitado 13-02-2019).

[10] *IoTivity*. URL: https://iotivity.org/ (visitado 24-04-2018).

[11] Open Connectivity Foundation. *OCF Device Specification*. Inf. téc. Version 2.0.1. Open Connectivity Foundation, feb. de 2019. URL: https://openconnectivity.org/specs/OCF_Device_Specification-2.0.1.pdf (visitado 13-02-2019).

[12] *OCF - AllJoyn*. URL: https://openconnectivity.org/developer/reference-implementation/alljoyn (visitado 24-04-2018).

[13] Open Mobile Alliance. *Lightweight Machine to Machine Technical Specification*. Inf. téc. Approved Version 1.0.2. Open Mobile Alliance, feb. de 2018. URL: http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/OMA-TS-LightweightM2M-V1_0_2-20180209-A.pdf (visitado 25-04-2018).

[14] Jose Manuel Cantera Fonseca, Fermín Galán Márquez y Tobias Jacobs. *FIWARE-NGSI v2 Specification*. Inf. téc. v2.0. FIWARE Foundation, 2018. URL: http://fiware.github.io/specifications/ngsiv2/stable/ (visitado 11-02-2019).

[15] ETSI ISG CIM. *Context Information Management (CIM) NGSI-LD API*. Inf. téc. GS CIM 009 V1.1.1 (2019-01). ETSI, ene. de 2019. URL: https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.01.01_60/gs_CIM009v010101p.pdf (visitado 11-02-2019).

[16] oneM2M. *oneM2M Functional Architecture*. Inf. téc. TS 118 101 V2.10.0 (2016-10). ETSI, oct. de 2016. URL: http://www.etsi.org/deliver/etsi_ts/118100_118199/118101/02.10.00_60/ts_118101v021000p.pdf (visitado 11-02-2019).

[17] Dominique Guinard, Vlad Trifa y David Carrera. *Web Thing Model*. W3C Member Submission. Abr. de 2017. URL: http://model.webofthings.io/ (visitado 02-05-2018).

[18] Ben Francis. *Web Thing API*. Unofficial Draft. Mozilla Corporation, ene. de 2019. URL: https://iot.mozilla.org/wot/ (visitado 12-02-2019).

[19] Steve Liang, Chih-Yuan Huang y Tania Khalafbeigi. *OGC SensorThings API Part 1: Sensing*. OGC Implementation Standard 15-078r6. Open Geospatial Consortium, jul. de 2016. URL: http://docs.opengeospatial.org/is/15-078r6/15-078r6.html (visitado 21-02-2019).

[20] Steve Liang y Tania Khalafbeigi. *OGC SensorThings API Part 2 – Tasking Core*. OGC Implementation Standard 17-079r1. Open Geospatial Consortium, ene. de 2019. URL: http://docs.opengeospatial.org/is/17-079r1/17-079r1.html (visitado 21-02-2019).

[21] Tomás Sánchez López y col. «Adding Sense to the Internet of Things». En: *Personal and Ubiquitous Computing* 16.3 (mar. de 2012), págs. 291-308. ISSN: 1617-4917. DOI: 10.1007/s00779-011-0399-8.

[22] D. Raggett. «The Web of Things: Challenges and Opportunities». En: *Computer* 48.5 (mayo de 2015), págs. 26-32. ISSN: 0018-9162. DOI: 10.1109/MC.2015.149.

[23] J. Heuer, J. Hund y O. Pfaff. «Toward the Web of Things: Applying Web Technologies to the Physical World». En: *Computer* 48.5 (mayo de 2015), págs. 34-42. ISSN: 0018-9162. DOI: 10.1109/MC.2015.152.

[24] Byungseok Kang, Daecheon Kim y Hyunseung Choo. «Internet of Everything: A Large-Scale Autonomic IoT Gateway». En: *IEEE Transactions on Multi-Scale Computing Systems* 3.3 (jul. de 2017), págs. 206-214. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2017.2705683.

[25] Thomas Zachariah y col. «The Internet of Things Has a Gateway Problem». En: *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. HotMobile '15. New York, NY, USA: ACM, 2015, págs. 27-32. ISBN: 978-1-4503-3391-7. DOI: 10.1145/2699343.2699344.

[26] Ali Shemshadi y col. «Searching for the Internet of Things: Where It Is and What It Looks Like». En: *Personal and Ubiquitous Computing* 21.6 (dic. de 2017), págs. 1097-1112. ISSN: 1617-4917. DOI: 10.1007/s00779-017-1034-0.

[27] Zhenyu Wu y col. «Towards a Semantic Web of Things: A Hybrid Semantic Annotation, Extraction, and Reasoning Framework for Cyber-Physical System». En: *Sensors* 17.2 (feb. de 2017), pág. 403. DOI: 10.3390/s17020403.

[28] Behailu Negash, Tomi Westerlund y Hannu Tenhunen. «Towards an Interoperable Internet of Things through a Web of Virtual Things at the Fog Layer». En: *Future Generation Computer Systems* 91 (feb. de 2019), págs. 96-107. ISSN: 0167-739X. DOI: 10.1016/j.future.2018.07.053.

[29] F. Paganelli, S. Turchi y D. Giuli. «A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City». En: *IEEE Systems Journal* 10.4 (dic. de 2016), págs. 1412-1423. ISSN: 1932-8184. DOI: 10.1109/JSYST.2014.2354835.

[30] José M. Hernández-Muñoz y Luis Muñoz. «The SmartSantander Project». En: *The Future Internet*. Ed. por Alex Galis y Anastasius Gavras. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, págs. 361-362. ISBN: 978-3-642-38082-2.

[31] Sujith Samuel Mathew y col. «Building Sustainable Parking Lots with the Web of Things». En: *Personal and Ubiquitous Computing* 18.4 (abr. de 2014), págs. 895-907. ISSN: 1617-4909, 1617-4917. DOI: 10.1007/s00779-013-0694-7.

[32] T. Sakamoto, H. Ito y K. Nimura. «Dynamically Exposing and Controlling Physical Devices by Expanding Web of Things Scheme». En: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. Jul. de 2017, págs. 329-335. DOI: 10.1109/COMPSAC.2017.114.

[33] A. H. Ngu y col. «IoT Middleware: A Survey on Issues and Enabling Technologies». En: *IEEE Internet of Things Journal* 4.1 (feb. de 2017), págs. 1-20. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2615180.

[34] Floris Van den Abeele y col. «Secure Service Proxy: A CoAP(s) Intermediary for a Securer and Smarter Web of Things». En: *Sensors* 17.7 (jul. de 2017), pág. 1609. DOI: 10.3390/s17071609.

[35] F. Banaie y col. «Performance Analysis of Multithreaded IoT Gateway». En: *IEEE Internet of Things Journal* (2018), págs. 1-1. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2879467.

[36] Vasileios Karagiannis y col. «A Survey on Application Layer Protocols for the Internet of Things». En: *Transaction on IoT and Cloud Computing* 3.1 (2015), págs. 11-17.

[37] Z. B. Babovic, J. Protic y V. Milutinovic. «Web Performance Evaluation for Internet of Things Applications». En: *IEEE Access* 4 (2016), págs. 6974-6992. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2615181.

[38] Sebastian Kaebisch y col. *Web of Things (WoT) Thing Description*. W3C Recommendation. W3C, abr. de 2020. URL: https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/ (visitado 17-05-2022).

[39] Dave Longley, Pierre-Antoine Champin y Gregg Kellogg. *JSON-LD 1.1*. W3C Recommendation. W3C, jul. de 2020.

[40] Sebastian Käbisch y Takuki Kamiya. *Web of Things (WoT) Thing Description*. W3C Working Draft. W3C, oct. de 2018. URL: https://www.w3.org/TR/2018/WD-wot-thing-description-20181021/.

[41] Zoltan Kis y col. *Web of Things (WoT) Scripting API*. W3C Working Group Note. W3C, nov. de 2020. URL: https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/ (visitado 17-05-2022).

[42] Andrés García Mangas. *WoTPy: Experimental Implementation of a W3C WoT Runtime*. CTIC Centro Tecnológico. Jun. de 2021. URL: https://github.com/agmangas/wot-py.

[43] Michael Koster y Ege Korkan. *Web of Things (WoT) Binding Templates*. W3C Working Group Note. W3C, ene. de 2020. URL: https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/ (visitado 17-05-2022).

[44] S. S. Adhatarao, M. Arumaithurai y X. Fu. «FOGG: A Fog Computing Based Gateway to Integrate Sensor Networks to Internet». En: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 2. Sep. de 2017, págs. 42-47. DOI: 10.23919/ITC.2017.8065709.

[45] Rodrigo N. Calheiros y col. «CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms». En: *Software: Practice and Experience* 41.1 (2011), págs. 23-50. ISSN: 1097-024X. DOI: 10.1002/spe.995.

[46] Sareh Fotuhi Piraghaj y col. «ContainerCloudSim: An Environment for Modeling and Simulation of Containers in Cloud Data Centers». En: *Software: Practice and Experience* 47.4 (2017), págs. 505-521. ISSN: 1097-024X. DOI: 10.1002/spe.2422.

[47] Harshit Gupta y col. «iFogSim: A Toolkit for Modeling and Simulation of Resource Management Techniques in the Internet of Things, Edge and Fog Computing Environments». En: *Software: Practice and Experience* 47.9 (2017), págs. 1275-1296. ISSN: 1097-024X. DOI: 10.1002/spe.2509.

[48] Manoel C. Silva Filho y col. «CloudSim Plus: A Cloud Computing Simulation Framework Pursuing Software Engineering Principles for Improved Modularity, Extensibility and Correctness». En: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Mayo de 2017, págs. 400-406. DOI: 10.23919/INM.2017.7987304.

[49] Isaac Lera, Carlos Guerrero y Carlos Juiz. «YAFS: A Simulator for IoT Scenarios in Fog Computing». En: *IEEE Access* 7 (2019), págs. 91745-91758. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927895.

[50] Bob Lantz, Brandon Heller y Nick McKeown. «A Network in a Laptop: Rapid Prototyping for Software-Defined Networks». En: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Ene. de 2010, pág. 19. DOI: 10.1145/1868447.1868466.

[51] Philip Wette y col. «MaxiNet: Distributed Emulation of Software-Defined Networks». En: *Proceedings of the 2014 IFIP Networking Conference (Networking 2014)*. 2014, págs. 1-9. DOI: 10.1109/IFIPNetworking.2014.6857078.

[52] Manuel Peuster, Holger Karl y Steven van Rossem. «MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments». En: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Palo Alto, CA: IEEE, nov. de 2016, págs. 148-153. ISBN: 978-1-5090-0933-6. DOI: 10.1109/NFV-SDN.2016.7919490.

[53] Ruben Mayer y col. «EmuFog: Extensible and Scalable Emulation of Large-Scale Fog Computing Infrastructures». En: *2017 IEEE Fog World Congress (FWC)*. Oct. de 2017, págs. 1-6. DOI: 10.1109/FWC.2017.8368525.

[54] Antonio Coutinho y col. «Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing». En: *2018 IEEE International Conference on Communications (ICC)*. Mayo de 2018, págs. 1-7. DOI: 10.1109/ICC.2018.8423003.

[55] Dirk Merkel. «Docker: Lightweight Linux Containers for Consistent Development and Deployment». En: *Linux J.* 2014.239 (mar. de 2014). ISSN: 1075-3583.

[56] The crun development team. *Crun*. Red Hat Inc. Ene. de 2022. URL: https://github.com/containers/crun.

[57] The Kubernetes Authors. *Kubernetes (K8s)*. Mayo de 2021. URL: https://kubernetes.io/.

[58] Bukhary Ikhwan Ismail y col. «Evaluation of Docker as Edge Computing Platform». En: *2015 IEEE Conference on Open Systems (ICOS)*. Ago. de 2015, págs. 130-135. DOI: 10.1109/ICOS.2015.7377291.

[59] Nam Ky Giang y col. «Developing IoT Applications in the Fog: A Distributed Dataflow Approach». En: *2015 5th International Conference on the Internet of Things (IOT)*. Seoul, South Korea: IEEE, oct. de 2015, págs. 155-162. ISBN: 978-1-4673-8056-0 978-1-4673-8058-4. DOI: 10.1109/IOT.2015.7356560.

[60] Alberto Medina y col. «BRITE: An Approach to Universal Topology Generation». En: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2001, págs. 346-353.

[61] Kazuo Kajimoto, Matthias Kovatsch y Uday Davuluru. *Web of Things (WoT) Architecture*. W3C First Public Working Draft. W3C, sep. de 2017. URL: https://www.w3.org/TR/2017/WD-wot-architecture-20170914/.

[62] Zoltan Kis y col. *Web of Things (WoT) Scripting API*. W3C Working Draft. W3C, nov. de 2018. URL: https://www.w3.org/TR/2018/WD-wot-scripting-api-20181129/.

[63] Andres Garcia Mangas. *WoTPy: A Framework for Web of Things Applications in Python*. Zenodo. Abr. de 2019. DOI: 10.5281/ZENODO.2605071.

[64] Andrés García Mangas y Francisco José Suárez Alonso. «WOTPY: A Framework for Web of Things Applications». En: *Computer Communications* 147 (nov. de 2019), págs. 235-251. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2019.09.004.

[65] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor / RFC Editor, dic. de 2017. URL: https://www.rfc-editor.org/std/std90.txt.

[66] S. Cheshire y M. Krochmal. *DNS-Based Service Discovery*. RFC 6763. RFC Editor, feb. de 2013. URL: http://www.rfc-editor.org/rfc/rfc6763.txt.

[67] S. Cheshire y M. Krochmal. *Multicast DNS*. RFC 6762. RFC Editor, feb. de 2013. URL: http://www.rfc-editor.org/rfc/rfc6762.txt.

[68] Facebook. *Tornado Web Server*. Sep. de 2018. URL: http://www.tornadoweb.org/.

[69] Christian Amsüss y Maciej Wasilak. *Aiocoap: Python CoAP Library*. Energy Harvesting Solutions. 2013. URL: http://github.com/chrysn/aiocoap/.

[70] Nicolas Jouanin. *HBMQTT*. Nov. de 2018. URL: https://github.com/beerfactory/hbmqtt.

[71] Roy Thomas Fielding. «REST: Architectural Styles and the Design of Network-based Software Architectures». Tesis doct. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[72] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. Inf. téc. Version 2.0. Ene. de 2013. URL: https://www.jsonrpc.org/specification (visitado 02-04-2019).

[73] Roger A Light. «Mosquitto: Server and Client Implementation of the MQTT Protocol». En: *The Journal of Open Source Software* 2.13 (mayo de 2017), pág. 265. ISSN: 2475-9066. DOI: 10.21105/joss.00265.

[74] Z. Shelby, K. Hartke y C. Bormann. *The Constrained Application Protocol (CoAP)*. Inf. téc. RFC7252. RFC Editor, jun. de 2014. DOI: 10.17487/rfc7252.

[75] C. Bormann y col. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*. Inf. téc. RFC8323. RFC Editor, feb. de 2018. DOI: 10.17487/RFC8323.

[76] K. Hartke. *Observing Resources in the Constrained Application Protocol (CoAP)*. Inf. téc. RFC7641. RFC Editor, sep. de 2015. DOI: 10.17487/RFC7641.

[77] Andrés García Mangas. *WoTemu: Emulator for Edge Computing Applications in Python*. Zenodo. Mayo de 2021. DOI: 10.5281/ZENODO.4769358.

[78] Andrés García Mangas. *WoTemu: Emulator for Edge Computing Applications in Python*. CTIC Centro Tecnológico. Mayo de 2021. URL: https://github.com/agmangas/wotemu.

[79] Andrés García Mangas y col. «WoTemu: An Emulation Framework for Edge Computing Architectures Based on the Web of Things». En: *Computer Networks* 209 (2022), pág. 108868. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2022.108868.

[80] Python Core Team. *Asyncio: Asynchronous I/O*. Python Software Foundation. 2022. URL: https://docs.python.org/3/library/asyncio.html.

[81] Jairo Llopis y João Marques. *Docker Socket Proxy*. Tecnativa. Ene. de 2021. URL: https://github.com/Tecnativa/docker-socket-proxy.

[82] Redis Labs. *Redis*. Mayo de 2021. URL: https://redis.io/.

[83] Wireshark Foundation. *TShark*. Abr. de 2021. URL: https://www.wireshark.org/.

[84] Dor Green. *Pyshark*. Feb. de 2021. URL: https://github.com/KimiNewt/pyshark.

[85] Alexey Kuznetsov y Stephen Hemminger. *Iproute2*. Abr. de 2021. URL: https://wiki.linuxfoundation.org/networking/iproute2.

[86] Rusty Russell y Netfilter Core Team. *Iptables*. Ene. de 2021. URL: https://www.netfilter.org/projects/iptables/index.html.

[87] Alexey Kopytov. *Sysbench*. Abr. de 2020. URL: https://github.com/akopytov/sysbench.

[88] Wes Felter y col. «An Updated Performance Comparison of Virtual Machines and Linux Containers». En: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Philadelphia, PA, USA: IEEE, mar. de 2015, págs. 171-172. ISBN: 978-1-4799-1957-4. DOI: 10.1109/ISPASS.2015.7095802.

[89] Plotly. *Plotly Python Open Source Graphing Library*. Ene. de 2021. URL: https://plotly.com/.

[90] Jeff Reback y col. *Pandas-Dev/Pandas: Pandas 1.2.4*. Zenodo. Abr. de 2021. DOI: 10.5281/zenodo.4681666.

[91] Andrés García Mangas. *WoTemu Experimental Results*. Jun. de 2021. DOI: 10.5281/ZENODO.4782152.

[92] Heike Hofmann, Karen Kafadar y Hadley Wickham. *Letter-Value Plots: Boxplots for Large Data*. Inf. téc. had.co.nz, 2011.

[93] Kirak Hong y col. «Mobile Fog: A Programming Model for Large-Scale Applications on the Internet of Things». En: *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*. MCC '13. New York, NY, USA: Association for Computing Machinery, 2013, págs. 15-20. ISBN: 978-1-4503-2180-8. DOI: 10.1145/2491266.2491270.

[94] G. Bradski. «The OpenCV Library». En: *Dr. Dobb's Journal of Software Tools* (2000).

[95] Adam Geitgey. *Face Recognition*. Feb. de 2020. URL: https://github.com/ageitgey/face_recognition.

[96] Davis E. King. «Dlib-Ml: A Machine Learning Toolkit». En: *Journal of Machine Learning Research* 10 (2009), págs. 1755-1758.

[97] Luca Sciullo y col. «A Survey on the Web of Things». En: *IEEE Access* 10 (2022), págs. 47570-47596. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3171575.

[98] Alex F R Trajano y col. «Leveraging Mobile Edge Computing on Smart Grids Using LTE Cellular Networks». En: *2019 IEEE Symposium on Computers and Communications (ISCC)*. Barcelona, Spain: IEEE, jun. de 2019, págs. 1-7. ISBN: 978-1-72812-999-0. DOI: 10.1109/ISCC47284.2019.8969784.

[99] Zhengzhe Xiang y col. «Robust and Cost-effective Resource Allocation for Complex IoT Applications in Edge-Cloud Collaboration». En: *Mobile Networks and Applications* (mayo de 2022). ISSN: 1383-469X, 1572-8153. DOI: 10.1007/s11036-022-01977-9.

[100] Theodoros Kasidakis y col. «Reducing the Mission Time of Drone Applications through Location-Aware Edge Computing». En: *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*. Melbourne, Australia: IEEE, mayo de 2021, págs. 45-52. ISBN: 978-1-66540-291-0. DOI: 10.1109/ICFEC51620.2021.00014.