Departamento de Informática
Universidad de Almería

Una arquitectura de microservicios para componentes digitales en la Web de las Cosas

Tesis Doctoral

Presentada por

Manel Mena Vicente

Tesis dirigida por

Dr. Luis Iribarne
Catedrático de Universidad
Departamento de Informática
Universidad de Almería

Dr. Javier Criado
Titular de Universidad
Departamento de Informática
Universidad de Almería

Programa de Doctorado en Informática de la UAL

Almería, Mayo, 2023

Escrita por: Manel Mena Vicente
Impresa por: Murex (Almería, Spain)

Mayo 2023

Department of Informatics
University of Almería

A microservice architecture for digital components in the Web of Things

Thesis

Presented by

Manel Mena Vicente

Thesis supervised by

Dr. Luis Iribarne
Full Professor
Department of Informatics
University of Almería

Dr. Javier Criado
Associate Professor
Department of Informatics
University of Almería

Doctoral Program in Informatics, University of Almería

Almería, May, 2023

Written and edited by: Manel Mena
Printed by: Murex Factoría de Color (Almería, Spain)

May 2023

Tesis Doctoral

Una arquitectura de microservicios para componentes digitales en la Web de las Cosas

A microservice architecture for digital components in the Web of Things

(Como requisitos para la Mención Internacional en el título de Doctor, el resumen y las conclusiones de la tesis doctoral han sido redactados tanto en castellano como inglés)

(*As requirements for International Mention in the title of the PhD, the summary and conclusions of the thesis have been written both in Spanish and English*)

Manel Mena Vicente

Este archivo ha sido generado utilizando LaTeX.

Todas las figuras y tablas de este archivo son originales

Una arquitectura de microservicios para componentes digitales en la Web de las Cosas.

Manel Mena Vicente
Departamento de Informática
Grupo de Informática Aplicada (TIC-211)
Universidad de Almería
Almería, Mayo, 2023

http://acg.ual.es

A Luis, Javi y Antonio,
A mi Madre, Padre y Hermana,
A todos aquellos que me importan.

Agradecimientos

Por fin llega mi tesis a su final. Con este documento se cierra un ciclo que empezó hace cinco años y que es fruto de un esfuerzo sostenido durante ese tiempo. Gracias al trabajo duro, a la paciencia y ayuda de muchas personas, he podido llegar a este punto.

Me gustaría, en primer lugar, dar las gracias a Luis Iribarne. Su ayuda, paciencia y consejos han sido fundamentales para que yo pudiera llegar a este punto. Gracias a él he tenido la oportunidad de conocer los *dimes y diretes* del mundo de la investigación. A lo largo de estos años he aprendido mucho de él, y he pasado de considerarlo simplemente un supervisor a un amigo. Gracias por todo.

En segundo lugar, me gustaría agradecer a Javier Criado, mi segundo supervisor, y el que me ha ayudado a mantenerme en el camino correcto. Es probablemente el que más me ha tenido que sufrir a lo largo de estos años, con él he trabajado codo con codo, y cada vez que he tenido algún problema ha sido el primero que siempre ha estado ahí para ayudarme. Gracias por todo.

También me gustaría dedicarle una mención especial a Antonio Corral. Aunque su nombre no aparezca en la portada de esta tesis, su actitud positiva y su ayuda han sido fundamentales para que yo pudiera llegar a este punto. Gracias a Antonio, que fue el primero que me abrió las puertas del grupo de investigación, he podido conocer al resto de personas maravillosas que forman parte del mismo. Gracias por todo.

Gracias también, cómo no, a todo el resto de compañeros del grupo de investigación, en especial a aquellos que han estado en el laboratorio conmigo trabajando día tras día. A aquellos que siguen trabajando con nosotros, como Tesi, Juan Alberto o Fran y a aquellos que ya no están con nosotros como Juanje, Manu o Ruth. Gracias por todo.

Por último, me gustaría dar las gracias a mi familia. A mi madre, que aunque ya no sea capaz de entender lo que ocurre, estoy seguro de que estaría orgullosa de lo que he conseguido, ya que en el comienzo siempre fue ella la que más me alentó; a mi padre, que siempre ha estado ahí para darme consejos y ha sido un espejo en el que mirarme; y a mi hermana, que siempre ha estado presente para darme ánimos. Gracias por todo.

Finalmente, mi agradecimiento a los proyectos que han soportado mi investigación: TIN2017-83964-R ("CoSmart: Estudio de un enfoque holístico para la interoperabilidad y coexistencia de sistemas dinámicos: Implicaciones en los modelos de Smart Cities"), PY20_00809 ("UrbanITA: Un modelo de referencia de servicios IoT abiertos dirigido a estrategias de eficiencia energética en edificios públicos inteligentes") y el proyecto financiado por CEIMAR CEIJ-C01.2 ("WoTport: descubriendo la Web de las Cosas hacia los SmartPorts") en los cuales he tenido el honor de participar. Además, he recibido financiación para la realización del doctorado a través de las ayudas de Formación del Profesorado Universitario (FPU) del gobierno de España FPU17/02010.

Manel Mena Vicente
Departamento de Informática
Grupo de Informática Aplicada
Universidad de Almería
Almería, 2023

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

RESUMEN

En los últimos años, el uso de dispositivos del Internet de las Cosas (Internet of Things, IoT) y de los sistemas ciberfísicos (Cyber-Physical Systems, CPS) ha crecido exponencialmente. De acuerdo a la última previsión realizada por Cisco Systems, el número de dispositivos conectados a Internet será de aproximadamente 500 billones para el año 2030 [Zikria et al., 2021], y no hay indicios de que la tendencia al alza vaya a variar. El gran número de dispositivos y la heterogeneidad de los mismos hace que establecer ecosistemas de este tipo no sea una tarea trivial. En primer lugar, este tipo de dispositivos está gobernado por una ingente cantidad de protocolos, lo cual hace que el desarrollo de software, el uso y la integración de dispositivos sea muy compleja. Derivado de este mismo problema, surge también el aspecto de la interoperabilidad entre dispositivos y plataformas. Para comprender un poco la cantidad de protocolos que existen, es común utilizar la siguiente división [Al-Sarawi et al., 2017]:

— Infraestructura (p.ej., 6LowPAN, IPv4/IPv6, RPL).
— Identificación (p.ej., EPC, uCode, IPv6, URIs).
— Comunicación / Transporte (p.ej., Wifi, Bluetooth, LPWAN).
— Descubrimiento (p.ej., Physical Web, mDNS, DNS-SD).
— Protocolos de datos (p.ej., MQTT, CoAP, AMQP, Websocket, Node).
— Gestión de dispositivos (p.ej., TR-069, OMA-DM).
— Semánticos (p.ej., JSON-LD, Web Thing Model).
— Frameworks Multi-capa (p.ej., Alljoyn, IoTivity, Weave, Homekit).

Para resolver el problema de la interoperabilidad y el de la integración existen ciertas soluciones de dispositivos IoT en entornos de Smart Home, como pueden ser HomeAssistant, OpenHab, Prodea, etc. Este tipo de infraestructuras están diseñadas para manejar un número controlado de dispositivos, y son normalmente aplicaciones de tipo monolítico, por lo que carecen de una buena capacidad de escalabilidad.

Otra problemática que es muy común cuando trabajamos con este tipo de dispositivos es que, debido a que se caracterizan por ser dispositivos en los cuales prima un bajo nivel de consumo energético, suelen estar restringidos en cuanto a su *poder* de computación y *capacidad* de memoria. Esto conlleva que aparezcan cuellos de botella en la comunicación, tanto a la hora de acceder a sus estados (p.ej., leer la temperatura de un sensor, ver el estado de una bombilla), como cuando se quiere actuar sobre ellos (p.ej., encender un dispositivo, cambiar el estado de un actuador, etc.), la comunicación puede ser muy lenta, y en el caso de que el dispositivo reciba muchas peticiones de manera simultánea, puede que no sea capaz de responder a todas ellas.

Además, también existe la necesidad de poder *virtualizar* estos dispositivos para realizar pruebas sobre ellos, sin que esto influya en nuestros procesos de negocio [Shetty, 2017]. Para ofrecer esto último, surge el concepto de componente virtual o gemelo digital (Digital Twin, DT) [Singh et al., 2021]. Los gemelos digitales son representaciones virtuales de elementos físicos, sistemas o dispositivos a lo largo de su ciclo de vida, que pueden usualmente son utilizados para estudiar el comportamiento de sus contrapartidas físicas. El concepto de DT, sin embargo, en muchos casos [Uhlemann et al., 2017, Negri et al., 2019, Lu et al., 2020] se centra en una aproximación monolítica para representar los dispositivos del mundo real y carecen de una aproximación a múltiples niveles que aborden de forma específica cada una de las facetas o capacidades de un dispositivo, tales como la interacción de los clientes con los propios DT, el control de eventos o la interacción por parte de los usuarios por medio de Interfaces de Usuario.

Para abordar todos los aspectos planteados anteriormente, en esta tesis doctoral se propone el concepto de **Digital Dice** (DD). Al igual que los gemelos digitales, los Digital Dice son representaciones virtuales de elementos físicos (p.ej., dispositivos IoT o CPS), que tratan de implicar una serie de mejoras con respecto a ellos. Los Digital Dice proponen una abstracción virtual de dispositivos IoT o sistemas ciberfísicos basada en microservicios, y pretende ser agnóstica a los protocolos utilizados por cada dispositivo cara a los usuarios. En una primera aproximación, se trabajó en la integración de los protocolos de datos (p.ej., MQTT) y semánticos (p.ej., JSON-LD).

Un Digital Dice está constituido por un conjunto de seis componentes, en forma capacidades o facetas. En la Figura 1, que se presenta un poco más adelante, se muestra la arquitectura interna de un Digital Dice, arquitectura que será descrita en detalle a lo largo del presente documento. De manera resumida, estas seis capacidades (en forma de componentes) del Digital Dice son las siguientes:

(a) `Controller`. Este componente se encarga de las funciones necesarias para establecer la comunicación con el usuario. Algunos de los métodos o rutas de esta faceta requerirán autenticación a nivel de aplicación o de usuario, o en ambos casos; es decir, el usuario podrá acceder a esas rutas sólo si tiene permisos, y por una aplicación en concreto. Esta comunicación se realizará a través de una API RESTful [Biehl, 2016] en primera instancia, aunque también se le proporciona al usuario una serie de métodos mediante el protocolo Server-Sent Events (SSE) [Cook, 2014] para que pueda recibir notificaciones en tiempo real.

(b) `Data Handler`. Este componente se encarga de la comunicación directa con la base de datos asociada al dispositivo. Su principal funcionalidad es recuperar los datos que los usuarios quieran consultar, así como guardar en la base de datos las peticiones de los usuarios que supongan el cambio de estado de alguna interacción que establezca el dispositivo. Por otro lado, también realizará un registro de cualquier petición que se realice en el sistema. Es decir, en el caso de dispositivos *actuadores*, este servicio registrará tanto el cambio de estado del actuador propiamente dicho, como quién ha realizado la interacción sobre el mismo. En el caso de los dispositivos *sensores*, se registrará todo cambio de estado producido en el sensor, así como quién lo ha consultado y en qué momento, para su posterior análisis. A parte de lo expuesto, esta faceta también permitirá consultar tanto el estado actual del

dispositivo como un histórico de los cambios de estado asociado al mismo, todo ello soportado por una base de datos documental, MongoDB [Giamas, 2022], que ofrece una gran capacidad de escalabilidad.

(c) `Reflection`. Es un componente responsable de establecer la comunicación con el dispositivo físico ofreciendo un único canal de acceso, y se encarga de "reflejar" cualquier cambio de estado que se produzca tanto en el dispositivo físico como en la base de datos asociada. El componente se encarga de realizar la traducción de los mensajes que se envían y reciban del dispositivo físico a un formato que pueda ser interpretado por el resto de componentes. Por ejemplo, si el dispositivo físico utiliza un protocolo de comunicación como MQTT [Soni and Makwana, 2017], este se encarga de traducir los mensajes de entrada/salida a un formato JSON, usado, como norma, por el resto de los microservicios (que implementan los componentes).

(d) `Event Handler`. Este componente se encarga de gestionar los eventos producidos en el dispositivo físico, previamente registrado por el componente `Reflection`. También es el responsable de escuchar eventos (externos e internos) que puedan afectar al dispositivo gestionado por el microservicio del componente.

(e) `User Interface`. Es un componente compuesto por una representación visual en forma de interfaz gráfica de usuario web (Web Component) del dispositivo IoT o CPS concreto. Por lo tanto, cuando el usuario requiera sólo una visualización gráfica directa del estado del dispositivo, podrá hacer uso de esta capacidad.

(f) `Virtualizer`. Este componente se encarga de la virtualización del dispositivo físico que puede ser usada para realizar pruebas de forma segura sin que esto afecte al dispositivo. Esta representación virtual se crea analizando la información que se encuentre en la base de datos, no siendo necesario que el dispositivo físico esté conectado para poder realizar dichas pruebas.

En su definición, los componentes tipo `Controller`, `Reflection` y `Data Handler` son propiedades obligatorias para cualquier Digital Dice, mientras que `Event Handler`, `User Interface` y `Virtualizer` son opcionales. Su uso o no dependerá de las necesidades del desarrollador o del experto o del propio dispositivo IoT o sistema ciberfísico que se vaya a representar con el Digital Dice.

Como solución tecnológica de la propuesta Digital Dice, se ha estudiado y desarrollado una **arquitectura de microservicios** que implementa cada una de las facetas o componentes del Digital Dice, y que está basada en las recomendaciones de la Web de las Cosas (Web of Things, WoT) del W3C [W3C, 2022a]. Hay que destacar que la Web de las Cosas es un área del Internet de las Cosas relativamente reciente y que está relacionada con la disciplina de la Ingeniería del Software.

La posibilidad de replicar las *facetas* (i.e., capacidades) de un Digital Dice, así como su funcionamiento interno, es lo que proporciona la posibilidad de escalar las peticiones recibidas en los dispositivos IoT. Además, y gracias a que establecer una arquitectura de microservicios nos lleva a un principio de responsabilidad única, tendremos la capacidad de escalar únicamente las facetas que se vean más afectadas en cualquier momento. Por ejemplo, si la faceta (i.e., si el componente) que más peticiones está recibiendo es `Data`

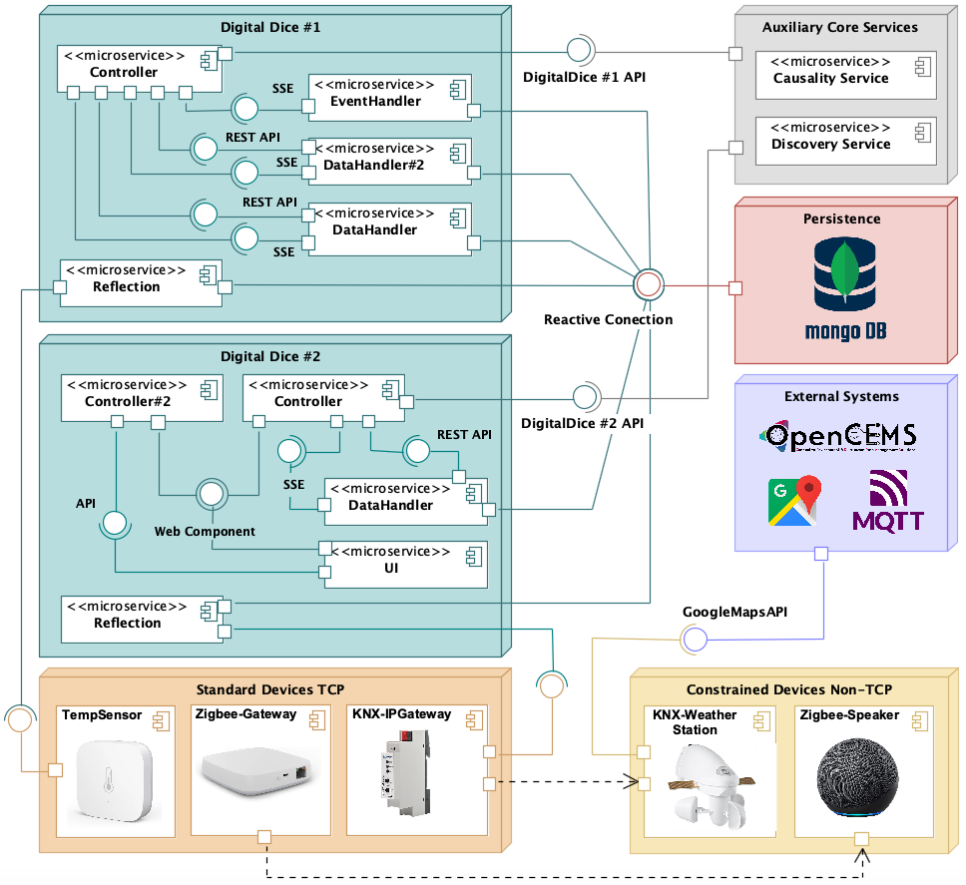


Figura 1: Arquitectura de Digital Dice.

`Handler` porque hay varios usuarios que están intentando acceder a datos históricos del dispositivo, entonces será posible replicar únicamente esta faceta.

Otro de los pilares sobre los que se sustenta el desarrollo de la tesis es la inclusión y el manejo de información contextual derivadas de los dispositivos IoT. Por ejemplo, la posición espacial (geolocalización) de los mismos puede incidir en la disponibilidad del Digital Dice que lo representa en una región en concreto. En el seno de nuestro grupo de investigación[1], se está desarrollando un sistema de descubrimiento/recomendador para dispositivos IoT [Llopis et al., 2022]. Este sistema de descubrimiento, originalmente basado en el modelo de mediación de [Iribarne et al., 2004], permite la búsqueda de dispositivos IoT en función de sus características, como por ejemplo, su posición geográfica, el tipo de dispositivo, o la cantidad de batería, entre otras posibles características. Por lo tanto, la información contextual derivada de los dispositivos IoT puede ser utilizada

[1]Applied Computing Group - `https://acg.ual.es/`

para la búsqueda de los Digital Dice que representan estos dispositivos. Por ejemplo, si un usuario quiere consultar la información de la temperatura, el sistema de descubrimiento nos proporcionará la lista de Digital Dice que representan a los dispositivos IoT de temperatura que se encuentran en la región desde donde se está haciendo la petición sin necesidad de conocer la dirección IP de los mismos.

Los componentes de un Digital Dice han sido diseñados para comunicarse con el exterior siguiendo los estándares, mecanismos o tecnologías establecidos en el marco de la Web de las Cosas (WoT), para así hacerlo compatible con otros sistemas que utilicen este estándar. El concepto WoT es una aproximación del IoT que utiliza la tecnología web para acercar a los desarrolladores a este tipo de sistemas embebidos, añadiendo una capa de abstracción basada en protocolos web de comunicación e interacción.

Para establecer la comunicación con los dispositivos IoT y ciberfísicos hemos desarrollado WoTnectivity, una librería multiprotocolo que ofrece un marco de trabajo diseñado para establecer conexiones con los diferentes protocolos encontrados en el ámbito IoT. Esta librería es capaz de trabajar con diferentes tipos de protocolos, como HTTP [Krishnamurthy and Rexford, 2001], MQTT [Hunkeler et al., 2008], WebSocket [Wang et al., 2013] o KNX [KNXAssoc, 2009], y ofrece a los desarrolladores la posibilidad de establecer la compatibilidad con otros protocolos, simplemente declarando WoTnectivity como una dependencia e implementando la interfaz proporcionada por ella. Esta librería se creó con la idea de combinar diferentes métodos de comunicación en un esfuerzo conjunto para establecer un patrón de uso común para diferentes dispositivos. WoTnectivity ofrece una capa de abstracción, donde se utiliza solo un esquema de comunicación común para operar con diferentes dispositivos, de modo que los desarrolladores puedan establecer comunicación con los dispositivos utilizando este patrón, lo cual reduce la barrera de entrada, pudiendo acelerar el tiempo de desarrollo, y mejorar la eficiencia a la hora de trabajar con este tipo de dispostivos.

Por otro lado, Digital Dice también maneja la posibilidad de manejar eventos [Buchmann and Koldehofe, 2009] mediante lo que, en esta tesis, se ha denominado como modelo de causalidad o "relaciones de causalidad" [Mena et al., 2021c]. Los eventos son divididos en dos niveles de aplicación: Individual o Múltiple.

(a) Individual. Interviene un único Digital Dice. Por ejemplo, un aviso de evento de un sensor de presencia sólo cuando se detecte movimiento dos veces en menos de un intervalo de 5 segundos; para reducir posibles falsos positivos.
(b) Múltiple. Intervienen dos o más Digital Dice. Por ejemplo, disparar un evento de encendido del aire acondicionado cuando se detecte la apertura de la puerta principal de la casa y se encienda la luz de entrada a la casa. Con este tipo de eventos se facilita la gestión de eventos complejos a nivel de múltiples dispositivos IoT. La propuesta tiene en cuenta la entrada de eventos, su procesamiento y la generación de la respuesta de un Digital Dice a través de la declaración de las relaciones de causalidad.

El modelo de causalidad propone un esquema híbrido (centralizado/descentralizado), capaz de gestionar la declaración de eventos en sistemas WoT, mediante un lenguaje específico de dominio (Domain-Specific Language, DSL). Este modelo se basa en la declaración de una serie de relaciones de causalidad en las que unas *causas* (temperatura de la habitación superior a 35 grados, humedad del entorno superior al 85 por ciento)

provocan un *efecto* (encendido del climatizador al 30 por ciento), que, a su vez, puede resultar en una causa de otra relación de causalidad. En el caso de que los dispositivos que intervengan en una relación de causalidad sean Digital Dice, estos dispositivos se encargan de la gestión de la misma ya que el `Event Handler` del Digital Dice cuenta con un motor intérprete de este tipo de relaciones, por lo que su esquema de actuación, en este caso, es descentralizado. Si en una relación de causalidad no intervienen Digital Dice, sólo sistemas o dispositivos WoT, el subsistema de causalidad se encarga de la gestión de la misma, por lo que su esquema de actuación, en este caso, es centralizado.

Por último, se han desarrollado diferentes estrategias y mecanismos para garantizar la alta disponibilidad (High Availability, HA) en los Digital Dice, una propiedad necesaria en redes de IoT y sistemas ciberfísicos, como en modelos de Ciudad Inteligente, Industria 4.0 o Smart Manufacturing. El concepto de Digital Dice ya nació con la idea de ser un sistema altamente escalable, que es uno de los requisitos necesarios para garantizar HA. Además, se han utilizado estrategias de distinta índole, como la replicación de los servicios, la utilización de un sistema de caché, el uso de mallas de servicios, entre otros, que nos permiten definir Digital Dice como un sistema altamente disponible.

Objetivos y contribuciones

En el desarrollo de Digital Dice viene condicionado por la consecución de una serie de objetivos específicos propuestos en este trabajo de investigación:

(1) Experimentar con los distintos protocolos utilizados por los dispositivos IoT, en especial, los protocolos de datos y los semánticos.

(2) Proporcionar una solución que permita abstraer esos dispositivos IoT de los protocolos que utilizan para dar solución a problemas de interoperabilidad.

(3) Establecer un nivel de separación para la gestión y el control de los dispositivos IoT, teniendo en cuenta distintas facetas que caracterizarán a estos y que serán representadas por distintos microservicios.

(4) Establecer un sistema de definición de dispositivos IoT compatible con la WoT, siguiendo la especificación de componentes establecida por el modelo formal de la Thing Description [W3C, 2022b].

(5) Definir una serie de esquemas de datos que permitan el almacenamiento tanto de los datos que generen los dispositivos como de los eventos que se produzcan cuando estos dispositivos son utilizados.

(6) Desarrollar un sistema de orquestación para la arquitectura de microservicios que permita la escalabilidad de servicios cuando sea necesario.

(7) Estudiar la posibilidad de establecer un sistema de control de eventos complejos tanto a nivel de dispositivos IoT de manera individual como colectivamente.

La principal contribución de esta tesis doctoral es el establecimiento de una propuesta para la gestión y el uso de dispositivos IoT o dispositivos ciberfísicos que permitan la interoperabilidad e integración de los mismos en una arquitectura de microservicios, todo ello siguiendo los estándares marcados por la Web of Things del W3C. Esta contribución busca abstraer a los desarrolladores, expertos y usuarios de los dispositivos IoT de los protocolos que estos utilizan para comunicarse entre sí, así como de los protocolos que utilizan para comunicarse con los sistemas que los gestionan.

A continuación, se exponen las aportaciones concretas derivadas de la consecución de los objetivos propuestos en esta tesis y el desarrollo de la misma:

(1) Se ha realizado una revisión de los protocolos de comunicación utilizados por los dispositivos IoT, así como de los protocolos de datos y semánticos que se utilizan para la comunicación entre dispositivos IoT y sistemas de gestión, de lo cual ha derivado WoTnectivity [Mena et al., 2020], una librería multiprotocolo que establece un patrón de comunicación común para distintos protocolos utilizados por dispositivos IoT (p.ej., HTTP, Websocket, KNX, MQTT, etc.).

(2) Se ha desarrollado una aplicación web progresiva (Progressive Web Application, PWA) basada en microservicios que combina la información contextual de un usuario y su dispositivo, con información obtenida de los dispositivos IoT y ciberfísicos para ofrecer una experiencia de usuario personalizada [Mena et al., 2019a].

(3) Se ha realizado una definición de Digital Dice como un sistema software basado en microservicios que permite la gestión de dispositivos IoT y dispositivos ciberfísicos utilizando una API RESTful como lenguaje de comunicación entre los usuarios y los distintos dispositivos [Mena et al., 2019c].

(4) Se ha definido un modelo de datos para establecer un esquema homogéneo común unificado de almacenamiento de los datos generados por los dispositivos IoT y ciberfísicos, así como de las peticiones que se producen cuando estos dispositivos son utilizados [Mena et al., 2021a].

(5) Se ha establecido un proceso de transformación de modelo-a-texto (M2T) para la generación de código fuente de Digital Dice a partir de la especificación Thing Description que define la WoT [Mena et al., 2021b]. Esta especificación facilita la creación de una plantilla de Digital Dice, que puede ser modificada por el desarrollador para adaptarla a sus necesidades.

(6) Se ha definido un lenguaje específico de dominio (DSL) para la definición de las relaciones de causalidad. Este lenguaje permite la definición de eventos complejos a partir de una serie de eventos denominados causas que repercuten sobre la ejecución de ciertas interacciones denominadas efectos [Mena et al., 2021c].

(7) Se ha extendido el concepto Digital Dice para convertirlo en un sistema de alta disponibilidad (High-Availability System, HAS). Para ello, se ha desarrollado un nuevo DSL que permite definir el comportamiento de replicación de cada uno de los microservicios que compone un Digital Dice. Asimismo, se ha definido una serie de

estrategias relacionadas con la comunicación entre microservicios, la comunicación con los dispositivos físicos y la replicación de microservicios para convertir a Digital Dice en un sistema de alta disponibilidad [Mena et al., 2023].

(8) Además, se ha realizado una definición formal del concepto Digital Dice mediante el desarrollo de un metamodelo, que extiende el uso de plantillas Thing Description de la WoT [Mena et al., 2023].

(9) Por último, también se ha realizado una evaluación y validación de la propuesta Digital Dice, mediante pruebas de concepto en distintos escenarios de uso, tanto en entornos reales como en entornos completamente virtuales [Mena et al., 2023]. Para ello, se ha desarrollado el laboratorio de pruebas WoT-Lab[2], además de un directorio de librerías en GitHub con el código fuente y varios escenarios de prueba relacionados con Ciudades Inteligentes. Toda esta información está disponible en abierto en la página web Digital Dice[3] del Grupo ACG/TIC-211 de la UAL.

Justificación

Tras exponer la problemática que se ha abordado en el trabajo de investigación desarrollado en esta tesis doctoral, hay que destacar las siguientes consideraciones. La propuesta desarrollada pretende dar una respuesta a los problemas de **interoperabilidad**, **integración**, **escalabilidad**, y de **usabilidad** que aparecen durante la gestión de sistemas ciberfísicos y dispositivos IoT, tanto reales como virtuales. Para ello, se ha llevado acabo una abstracción de la funcionalidad de los dispositivos IoT mediante la definición de un metamodelo Digital Dice, y una implementación del mismo mediante microservicios, siguiendo los estándares que marca la normativa de la Web de las Cosas (WoT). El uso de arquitecturas de microservicios trae consigo la posibilidad de proponer una orquestación de los mismos en los que se tenga en cuenta requisitos deseables para nuestro sistema, como por ejemplo: (a) una escalabilidad de granularidad fina, que nos permite realizar un mejor aprovechamiento de recursos; (b) un mantenimiento más simple y barato, dado que podemos trabajar mejorando facetas de los Digital Dice de manera individual; y (c) una modularidad alta, pues Digital Dice permite una evolución del sistema de una manera mucho más natural, permitiendo extender nuevas capacidades o ampliar las ya existentes progresivamente.

Por otro lado, la creación del concepto de Digital Dice, hace posible controlar dispositivos IoT que requieran de distintas facetas que los representen, por ejemplo, puede haber dispositivos que se deban controlar a través de servicios web, pero no requieran una representación gráfica de los mismos para explorar visualmente información de asociada a su estado interno. Para facilitar la interoperabilidad entre dispositivos, se establece una solución similar a la que proponen los sistemas de control de eventos, para permitir la comunicación entre distintos Digital Dice y dispositivos WoT en general sin necesidad de que haya una intervención por parte del usuario: las relaciones de causalidad.

[2]WoT-Lab/ACG: `https://acg.ual.es/wot-lab`
[3]Digital Dice, portal web: `https://acg.ual.es/projects/cosmart/digitaldice`

Marco y líneas futuras

El desarrollo de esta tesis ha formado parte de tres proyectos de investigación. En este apartado se describen los proyectos de investigación en los que ha participado el autor de esta tesis y las aportaciones realizadas en cada uno de ellos.

En primer lugar, el proyecto TIN2017-83964-R "*CoSmart: Estudio de un enfoque holístico para la interoperabilidad y coexistencia de sistemas dinámicos: Implicación en modelos de Smart Cities*" financiado por el Ministerio de Economía, Industria y Competitividad (MINECO) del gobierno de España. Este proyecto propone, entre otras cosas, desarrollar un marco común de referencia para la creación de componentes de la WoT, y aplicarlos a sistemas dinámicos en ecosistemas de ciudades inteligentes (Smart Cities). El modelo Digital Dice se ha desarrollado como pieza clave de este marco común de referencia, como componente WoT.

En segundo lugar, está el proyecto P20_00809 "*UrbanITA: Un modelo de referencia de servicios IoT abiertos dirigido a estrategias de eficiencia energética en edificios públicos inteligentes*" del Plan Andaluz de Investigación, Desarrollo e Innovación del Sistema Andaluz del Conocimiento de la Junta de Andalucía. Este proyecto tiene como objetivo fundamental el desarrollo de un modelo de referencia que permita integrar servicios para la gestión energética basado en plataformas software que habiliten la integración e interoperabilidad de tecnologías relevantes como el Internet de las Cosas y de la Web de las Cosas. Digital Dice ha sido una pieza fundamental en el proyecto, al potenciar las capacidades de interoperabilidad ofrecidas por las relaciones de causalidad.

Por último, está el proyecto CEIJ-C01.2 "*WoTport: descubriendo la Web de las Cosas hacia los SmartPorts*" financiado por el Campus de Excelencia Internacional del Mar (CEIMAR). En el proyecto se realiza un estudio para el desarrollo de un servicio de descubrimiento de componentes IoT en aplicaciones del ámbito de los SmartPorts. Digital Dice ha servido de soporte en las fases de prueba del descubrimiento, ofreciendo, además, para el proyecto, una solución basada en la Web de las Cosas (WoT).

Este trabajo ha sido posible también gracias a la ayuda de Formación del Profesorado Universitario (FPU) del Ministerio de Ciencia e Innovación del Gobierno de España FPU17/02010. Esta ayuda corresponde a la convocatoria 2017 y, durante la misma, se llevó a cabo el trabajo de investigación y la participación en los distintos proyectos que han dado lugar a esta tesis. Esta tesis ha sido desarrollada bajo el programa de Doctorado de Informática de la Universidad de Almería (España) con Ref. 8908 del RD99/11.

A partir de los objetivos presentados anteriormente, quedan abiertas varias líneas de investigación. En primer lugar, la posibilidad de extender el modelo de Digital Dice a otros tipos de servicios web externos y no necesariamente dispositivos IoT y dispositivos ciberfísicos. Por otro lado, a nivel de virtualización de dispositivos, la propuesta desarrollada en esta tesis doctoral es una solución supervisada, basada en conocimiento del experto. Puede ser de utilidad considerar soluciones autónomas, centradas en aprendizaje automático para la virtualización de dispositivos. Por último, puede ser interesante estudiar la virtualización de escenarios completos, observando cómo ciertos dispositivos pueden afectar en la virtualización de otros dispositivos, como puede ocurrir cuando trabajamos en un entorno real, donde los dispositivos pueden afectarse entre sí (p.ej., un aire acondicionado puede afectar a un sensor de temperatura dentro en una estancia).

Estructura del documento de tesis

Este documento queda estructurado en cinco capítulos. Al final del mismo, se ofrece un listado de acrónimos para facilitar la lectura, y una completa bibliografía, usada durante el desarrollo de la investigación de la tesis doctoral.

El *Capítulo 1* contiene los conceptos fundamentales de la base de conocimiento necesaria para entender el presente documento de tesis. Este capítulo expone conceptos básicos sobre IoT, entre otros: los dominios de aplicación más importantes o la clasificación en capas de las arquitecturas para el manejo de este tipo de dispositivos; se establece una introducción al estándar de la Web of Things (WoT) donde se incide especialmente en dos de las especificaciones más importantes de la WoT, concretamente la Arquitectura y la Thing Description (TD). Este capítulo también introduce las distintas arquitecturas de software sobre las que se basa la arquitectura propuesta por los Digital Dice, así como las distintas propuestas de middleware para la gestión de dispositivos IoT y dispositivos ciberfísicos. Por último, se introduce el concepto de procesadores de eventos y la importancia que tienen en la gestión de dispositivos IoT y ciberfísicos.

El *Capítulo 2* presenta el concepto de Digital Dice y su relación con la Web de las Cosas (WoT); se describen en detalle los componentes de Digital Dice; se expone la arquitectura de Digital Dice y cómo funciona la comunicación interna entre las piezas que forman parte de esta arquitectura; se presenta el metamodelo que da lugar al lenguaje (DSL) que ayuda a definir las piezas o los microservicios que forman parte de un Digital Dice; se expone el modelo de datos que se utiliza para almacenar la información generada por los dispositivos IoT y dispositivos ciberfísicos que representan nuestro Digital Dice; y por último se define un proceso de transformación de modelo-a-texto (M2T) para establecer la base de código de los distintos componentes de un Digital Dice a partir de la Thing Description que lo representa.

En el *Capítulo 3* se describen las distintas capacidades de la arquitectura de Digital Dice, que representan las contribuciones principales del trabajo de investigación: el modelo de comunicación WoTnectivity, el modelo de relaciones de causalidad, y las estrategias de alta disponibilidad de Digital Dice. Así, en primer lugar, en este capítulo se presentará WoTnectivity, una librería multiprotocolo para la comunicación con dispositivos IoT y dispositivos ciberfísicos. En esta sección se establecerán los conceptos básicos de la librería y su forma de uso; se expondrán los protocolos que actualmente son compatibles con la librería y se definirá cómo implementar otros nuevos protocolos. Seguidamente, se presentará el concepto de relaciones de causalidad y su necesidad para la comunicación entre Things; se establecerán las partes de las relaciones, se presentará el DSL que permite definir estas relaciones de causalidad, y se expondrá un ejemplo de uso de este lenguaje. Por último, en este capítulo, se presentarán los mecanismos y estrategias desarrolladas para facilitar la propiedad de Alta Disponibilidad de sistemas basados en Digital Dice; también se presentarán las pruebas de rendimiento de Digital Dice frente a la conexión directa con dispositivos IoT gobernados por protocolos típicos como KNX, ZigBee o HTTP, y se utilizará un ejemplo base en el ámbito de las Ciudades Inteligentes, para la recolección de basura.

En el *Capítulo 4* se ofrecen detalles de la implementación del modelo Digital Dice, que ha sido realizada mediante microservicios. En este capítulo, se exponen características

relacionadas con los aspectos técnicos empleados para el despliegue de los componentes de Digital Dice, y algunas consideraciones sobre el estudio y desarrollo de los algoritmos implementados mediante microservicios, y validados en una serie de pruebas de concepto realizadas en entornos reales y virtuales.

El *Capítulo 5* contiene las conclusiones del trabajo de investigación, un resumen de las contribuciones y algunas limitaciones que presenta la propuesta que se realiza en esta tesis doctoral. En el capítulo, por tanto, se describirán las principales aportaciones realizadas y su relación con las publicaciones en congresos nacionales e internacionales, así como en revistas de impacto; se expondrán algunas consideraciones de crítica de la propuesta y las líneas que han quedado abiertas en la investigación, que se podrían acometer en trabajos futuros como continuación de este trabajo de tesis.

En general, cada capítulo contiene una pequeña *Introducción* donde se expone el objetivo que se persigue en el mismo y una breve descripción de los contenidos que se van a tratar en cada capítulo. Como complemento, al final del presente documento, se ofrece una lista de acrónimos y una serie de anexos relacionados con implementaciones y documentos de validación de los modelos utilizados por Digital Dice. Por último, se incluye un apartado de la bibliografía utilizada durante el desarrollo del trabajo de investigación en esta tesis doctoral.

SUMMARY

In recent years, the use of devices from the Internet of Things (IoT) and Cyber-Physical Systems (CPS) has grown exponentially. According to the latest forecast by Cisco Systems, the number of devices connected to the Internet will be approximately 500 billion by 20230 [Zikria et al., 2021], and there are no indications that this upward trend will vary. The large number of devices and their heterogeneity make establishing ecosystems of this type a non-trivial task. Firstly, such devices are governed by a huge number of protocols, making software development, device usage, and integration very complex. This problem also gives rise to the issue of interoperability between devices and platforms. To understand the number of protocols that exist, it is common to use the following classification [Al-Sarawi et al., 2017]:

— Infrastructure (e.g., 6LowPAN, IPv4/IPv6, RPL).
— Identification (e.g., EPC, uCode, IPv6, URIs).
— Communication / Transport (e.g., Wi-Fi, Bluetooth, LPWAN).
— Discovery (e.g., Physical Web, mDNS, DNS-SD).
— Data protocols (e.g., MQTT, CoAP, AMQP, Websocket, Node).
— Device management (e.g., TR-069, OMA-DM).
— Semantic (e.g., JSON-LD, Web Thing Model).
— Multi-layer frameworks (e.g., Alljoyn, IoTivity, Weave, Homekit).

To solve the interoperability and integration problems, there are certain IoT device solutions in Smart Home environments, such as HomeAssistant, OpenHab, Prodea, etc. These types of infrastructures are designed to handle a controlled number of devices and are typically monolithic applications, so they lack good scalability capacity.

Another common problem when working with these types of devices is that, due to their low energy consumption, they are often restricted in terms of their computing power and memory capacity. This leads to communication bottlenecks, both when accessing their states (e.g., reading the temperature of a sensor, checking the status of a light bulb) and when trying to act on them (e.g., turning on a device, changing the state of an actuator, etc.). Communication can be very slow, and in the case that the device receives many simultaneous requests, it may not be able to respond to all of them.

In addition, there is also a need to virtualize these devices to perform tests on them without affecting our business processes [Shetty, 2017]. To provide the latter, the concept of a virtual component or digital twin (DT) [Singh et al., 2021] emerged. Digital twins are virtual representations of physical elements, systems, or devices throughout their lifecycle, which can typically be used to study the behavior of their physical counterparts.

However, in many cases [Uhlemann et al., 2017, Negri et al., 2019, Lu et al., 2020], the concept of DT focuses on a monolithic approach to representing real-world devices and lacks a multi-level approach that specifically addresses each of the facets or capabilities of a device, such as customer interaction with the DTs themselves, event control, or user interaction through User Interfaces.

To address all of the above aspects, this doctoral thesis proposes the concept of **Digital Dice** (DD). Like digital twins, Digital Dice are virtual representations of physical elements (e.g., IoT or CPS devices) that seek to provide a number of improvements over them. Digital Dice propose a virtual abstraction of IoT devices or cyber-physical systems based on microservices and aims to be agnostic to the protocols used by each device for user interaction. In a first approach, we worked on the integration of data (e.g., MQTT) and semantic (e.g., JSON-LD) protocols.

A Digital Dice is composed of a set of six components, in the form of capabilities or facets. In Figure 2, which is presented later on, the internal architecture of a Digital Dice is shown, this architecture will be described in detail throughout this document. In summary, the capabilities (in the form of components) of the Digital Dice are as follows:

(a) `Controller`. This component is responsible for the necessary functions to establish communication with the user. Some of the methods or routes of this facet will require application or user-level authentication, or both, meaning that the user can only access those routes if they have permissions, and for a specific application. This communication will be performed through a RESTful API [Biehl, 2016] in the first instance, although the user is also provided with a series of methods through the Server-Sent Events (SSE) protocol [Cook, 2014] so that they can receive real-time notifications.

(b) `Data Handler`. This component is responsible for direct communication with the device's associated database. Its main functionality is to retrieve the data that users want to query, as well as to save in the database the requests from users that involve changing the state of any interaction established by the device. On the other hand, it will also register any requests made in the system. That is, in the case of *actuators*, this service will register both the change in state of the actuator itself, and who has interacted with it. In the case of *sensors*, any change in state produced in the sensor will be registered, as well as who has queried it and when, for subsequent analysis. In addition to the above, this facet will also allow querying both the current state of the device and a history of the state changes associated with it, all supported by a document-based database, MongoDB [Giamas, 2022], which offers great scalability.

(c) `Reflection`. This is a component responsible for establishing communication with the physical device by offering a single access channel, and is responsible for "reflecting" any changes in state that occur both in the physical device and in the associated database. The component is responsible for translating the messages that are sent and received from the physical device to a format that can be interpreted by the other components. For example, if the physical device uses a communication protocol such as MQTT [Soni and Makwana, 2017], this component is responsible

for translating input/output messages to a JSON format, which is commonly used by the rest of the microservices (which implement the components).

(d) `Event Handler`. This component is responsible for managing the events produced in the physical device, previously registered by the `Reflection` component. It is also responsible for listening to events (external and internal) that may affect the device managed by the microservice of the component.

(e) `User Interface`. This component is composed of a visual representation in the form of a web user interface (Web Component) of the specific IoT or CPS device. Therefore, when the user only requires a direct graphical visualization of the device's state, they can make use of this capability.

(f) `Virtualizer`. This component is responsible for virtualizing the physical device and can be used for safe testing without affecting the device. This virtual representation is created by analyzing the information found in the database.

By definition, the `Controller`, `Reflection`, and `Data Handler` components are mandatory properties for any Digital Dice, while `Event Handler`, `User Interface`, and `Virtualizer` are optional. Their use depends on the needs of the developer, expert, or the IoT device or cyber-physical system being represented by the Digital Dice.

As a technological solution for the Digital Dice proposal, a **microservices architecture** has been studied and developed that implements each of the facets or components of the Digital Dice, based on the recommendations of the Web of Things (WoT) of the W3C [W3C, 2022a]. It is worth noting that the Web of Things is a relatively recent area of the Internet of Things and is related to the discipline of Software Engineering.

The ability to replicate the *facets* (i.e., capabilities) of a Digital Dice, as well as its internal operation, is what provides the ability to scale the requests received on IoT devices. Additionally, and thanks to establishing a microservices architecture leads us to a single responsibility principle, we will have the ability to scale only the facets that are most affected at any given time. For example, if the facet (i.e., component) receiving the most requests is the `Data Handler` because several users are trying to access historical data from the device, then it will be possible to replicate only this facet.

Another pillar upon which the development of the thesis is based is the inclusion and management of contextual information derived from IoT devices. For instance, their spatial position (geolocation) may impact the availability of the Digital Dice that represent them in a specific region. Within our research group[4], we are developing a discovery/recommender system for IoT devices [Llopis et al., 2022]. This discovery system, originally based on the mediation model proposed by [Iribarne et al., 2004], allows the search for IoT devices based on their characteristics, such as their geographical position, type, or battery level, among other possible features. Therefore, the contextual information derived from IoT devices can be used to search for the Digital Dice that represent these devices. For instance, if a user wants to retrieve temperature information, the discovery

[4]Applied Computing Group - `https://acg.ual.es/`

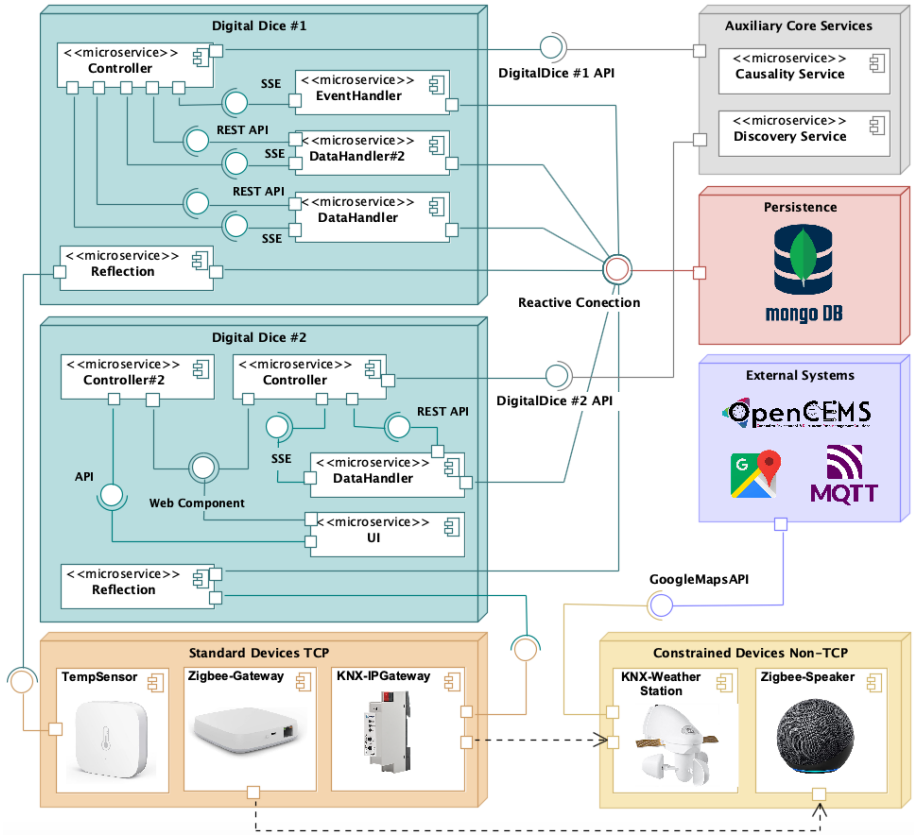


Figura 2: Digital Dice Architecture.

system will provide us with the list of Digital Dice that represent the temperature sensors located in the region where the request is being made, without requiring knowledge of their IP address.

The components of a Digital Dice have been designed to communicate with the outside world following the standards, mechanisms, or technologies established within the framework of the Web of Things (WoT), in order to make it compatible with other systems that use this standard. The WoT concept is an IoT approach that leverages web technology to bring developers closer to embedded systems, adding a layer of abstraction based on web communication and interaction protocols.

To establish communication with IoT and cyber-physical devices, we have developed WoTnectivity, a multiprotocol library that provides a framework designed to establish connections with the different protocols found in the IoT field. This library is capable of working with different types of protocols, such as HTTP [Krishnamurthy and Rexford, 2001], MQTT [Hunkeler et al., 2008], WebSocket [Wang et al., 2013], or KNX [KNXAssoc, 2009], and offers developers the possibility of establishing compatibility with

other protocols by simply declaring WoTnectivity as a dependency and implementing the interface provided by it. This library was created with the idea of combining different communication methods in a joint effort to establish a common usage pattern for different devices. WoTnectivity offers an abstraction layer, where only a common communication scheme is used to operate with different devices, so that developers can communicate with devices using this pattern, which reduces the barrier to entry, accelerates development time, and improves efficiency when working with this type of devices.

On the other hand, Digital Dice also handles the possibility of managing events [Buchmann and Koldehofe, 2009] through what, in this thesis, has been called the causality model or "causality relationships" [Mena et al., 2021c]. The events are divided into two levels of application: Individual or Multiple.

(a) Individual. A single Digital Dice is involved. For example, an event notification from a presence sensor only when movement is detected twice within an interval of less than 5 seconds; to reduce possible false positives.
(b) Multiple. Two or more Digital Dice are involved. For example, triggering an air conditioning switch event when the front door of the house is opened and the entry light is turned on. This type of event facilitates the management of complex events at the level of multiple IoT devices. The proposal takes into account the input of events, their processing, and the generation of the response from a Digital Dice through the declaration of causality relationships.

The causality model proposes a hybrid (centralized/decentralized) scheme capable of managing event declarations in WoT systems, through a domain-specific language (DSL). This model is based on the declaration of a series of causality relationships in which "causes" (room temperature above 35 degrees, environment humidity above 85 percent) trigger an "effect" (air conditioning switch on at 30 percent), which, in turn, can result in a cause of another causality relationship. In the case where Digital Dice devices are involved in a causality relationship, these devices are responsible for managing it, since the Digital Dice's Event Handler has an interpreter engine for this type of relationship, so its operating scheme in this case is decentralized. If a causality relationship does not involve Digital Dice, only WoT systems or devices, the causality subsystem is responsible for managing it, so its operating scheme in this case is centralized.

Finally, different strategies and mechanisms have been developed to ensure high availability (HA) in Digital Dice, a necessary property in IoT networks and cyber-physical systems, such as in Smart City, Industry 4.0, or Smart Manufacturing models. The concept of Digital Dice was already born with the idea of being a highly scalable system, which is one of the necessary requirements to ensure HA. In addition, strategies of different kinds have been used, such as service replication, cache system usage, service meshes usage, among others, that allow us to define Digital Dice as a highly available system.

Objectives and Contributions

The development of Digital Dice is conditioned by the achievement of a series of specific objectives proposed in this research work:

(1) To experiment with the different protocols used by IoT devices, especially data and semantic protocols.

(2) To provide a solution that allows abstracting these IoT devices from the protocols they use to solve interoperability problems.

(3) To establish a level of separation for the management and control of IoT devices, taking into account different facets that will characterize them and that will be represented by different microservices.

(4) To establish a system for defining IoT devices compatible with the WoT, following the component specification established by the formal model of the Thing Description [W3C, 2022b].

(5) To define a series of data schemas that allow the storage of both the data generated by the devices and the events that occur when these devices are used.

(6) To develop an orchestration system for the microservices architecture that allows the scalability of services when necessary.

(7) To study the possibility of establishing a complex event control system both at the level of IoT devices individually and collectively.

The main contribution of this doctoral thesis is the establishment of a proposal for the management and use of IoT devices or cyber-physical devices that allow interoperability and integration of them in a microservices architecture, all of which following the standards set by the W3C's Web of Things. This contribution seeks to abstract developers, experts, and users of IoT devices from the protocols they use to communicate with each other, as well as from the protocols they use to communicate with the systems that manage them.

Below are the specific contributions derived from the achievement of the proposed objectives in this thesis and its development:

(1) A review of the communication protocols used by IoT devices, as well as the data and semantic protocols used for communication between IoT devices and management systems, has been carried out. From this, WoTnectivity [Mena et al., 2020], a multiprotocol library that establishes a common communication pattern for different protocols used by IoT devices (e.g., HTTP, Websocket, KNX, MQTT, etc.), has been derived.

(2) A Progressive Web Application (PWA) based on microservices has been developed, which combines user and device contextual information with information obtained from IoT and cyber-physical devices to offer a personalized user experience [Mena et al., 2019a].

(3) Digital Dice has been defined as a software system based on microservices that allows the management of IoT and cyber-physical devices using a RESTful API as a communication language between users and different devices [Mena et al., 2019c].

(4) A data model has been defined to establish a unified homogeneous schema for the storage of data generated by IoT and cyber-physical devices, as well as for the requests that occur when these devices are used [Mena et al., 2021a].

(5) A model-to-text (M2T) transformation process has been established for generating source code of Digital Dice from the Thing Description specification of the WoT [Mena et al., 2021b]. This specification facilitates the creation of a Digital Dice template that can be modified by the developer to adapt it to their needs.

(6) A Domain-Specific language (DSL) has been defined for the definition of causality relationships. This language allows the definition of complex events from a series of events called causes that impact the execution of certain interactions called effects [Mena et al., 2021c].

(7) The Digital Dice concept has been extended to become a high-availability system (HAS). For this, a new DSL has been developed that allows the definition of the replication behavior of each of the microservices that make up a Digital Dice. Likewise, a series of strategies related to communication between microservices, communication with physical devices, and microservice replication has been defined to make Digital Dice a high-availability system [Mena et al., 2023].

(8) Additionally, a formal definition of the Digital Dice concept has been carried out through the development of a metamodel, which extends the use of Thing Description templates of the WoT [Mena et al., 2023].

(9) Finally, an evaluation and validation of the Digital Dice proposal has been carried out through proof-of-concept testing in different usage scenarios, both in real environments and completely virtual environments [Mena et al., 2023]. To this end, the WoT-Lab test laboratory has been developed[5], as well as a directory of libraries on GitHub with source code and several test scenarios related to Smart Cities. All this information is available openly on the Digital Dice website[6] of the ACG/TIC-211 Group of UAL.

Justification

After presenting the problem addressed in the research work developed in this doctoral thesis, the following considerations should be highlighted. The proposed solution aims to address the issues of **interoperability**, **integration**, **scalability**, and **usability** that arise during the management of cyber-physical systems and IoT devices, both real and virtual. To this end, an abstraction of the functionality of IoT devices has been carried out through the definition of a Digital Dice metamodel and its implementation through microservices, following the standards set by the Web of Things (WoT) regulation. The

[5]WoT-Lab/ACG: `https://acg.ual.es/wot-lab`
[6]Digital Dice, website: `https://acg.ual.es/projects/cosmart/digitaldice`

use of microservice architectures brings with it the possibility of proposing an orchestration of these services, taking into account desirable requirements for our system, such as: (a) fine-grained scalability, which allows us to make better use of resources; (b) simpler and cheaper maintenance, as we can work on improving facets of Digital Dice individually; and (c) high modularity, as Digital Dice allows for a more natural evolution of the system, enabling the gradual extension of new capabilities or the expansion of existing ones.

On the other hand, the creation of the Digital Dice concept makes it possible to control IoT devices that require different facets to represent them. For example, there may be devices that need to be controlled through web services, but do not require a graphical representation to visually explore information associated with their internal state. To facilitate interoperability between devices, a solution similar to that proposed by event control systems is established to allow communication between different Digital Dice and WoT devices in general without the need for user intervention: causality relationships.

Framework and Future Lines

The development of this thesis has been part of three research projects. In this section, we describe the research projects in which the author of this thesis has participated and the contributions made in each of them.

Firstly, the TIN2017-83964-R project "*CoSmart: Study of a holistic approach to the interoperability and coexistence of dynamic systems: Implication in Smart Cities models*" funded by the Ministry of Economy, Industry and Competitiveness (MINECO) of the Spanish government. This project proposes, among other things, the development of a common reference framework for creating components of the WoT, and applying them to dynamic systems in smart city ecosystems (Smart Cities). The Digital Dice model has been developed as a key component of this common reference framework, as a WoT component.

Secondly, there is the P20_00809 project "*UrbanITA: A reference model of open IoT services aimed at energy efficiency strategies in intelligent public buildings*" of the Andalusian Research, Development and Innovation Plan of the Andalusian Knowledge System of the Andalusian Regional Government. The main objective of this project is to develop a reference model that allows the integration of services for energy management based on software platforms that enable the integration and interoperability of relevant technologies such as the Internet of Things and the Web of Things. Digital Dice has been a fundamental piece in the project, by enhancing the interoperability capabilities offered by causality relationships.

Finally, there is the CEIJ-C01.2 project "WoTport: Discovering the Web of Things towards SmartPorts," funded by the International Campus of Excellence in the Sea (CEIMAR). The project conducts a study for the development of an IoT component discovery service in SmartPorts applications. Digital Dice has provided support in the discovery testing phases, and has also provided a Web of Things (WoT) based solution for the project.

This work has also been possible thanks to the help of the University Teaching Training (FPU) program of the Ministry of Science and Innovation of the Government of Spain, FPU17/02010. This help corresponds to the 2017 call, and during it, the research work and participation in the various projects that have led to this thesis were carried out. This thesis has been developed under the Doctoral Program in Computer Science of the University of Almeria (Spain) with Ref. 8908 of RD99/11.

Based on the objectives presented above, several lines of research remain open. Firstly, the possibility of extending the Digital Dice model to other types of external web services, not necessarily IoT devices and cyber-physical devices. On the other hand, in terms of device virtualization, the proposal developed in this doctoral thesis is a supervised solution based on expert knowledge. It may be useful to consider autonomous solutions focused on machine learning for device virtualization. Lastly, it may be interesting to study the virtualization of complete scenarios, observing how certain devices can affect the virtualization of other devices, as can happen when working in a real environment, where devices can affect each other (e.g., an air conditioner may affect a temperature sensor inside a room).

Thesis document structure

This document is structured into five chapters. At the end of the document, a list of acronyms is provided to facilitate reading, as well as a comprehensive bibliography used during the development of the doctoral thesis research.

Chapter 1 contains the fundamental concepts of the knowledge base necessary to understand the present thesis document. This chapter covers basic concepts about IoT, such as the most important application domains or the layer classification of architectures for managing this type of devices. An introduction to the Web of Things (WoT) standard is also provided, with a particular focus on two of the most important WoT specifications, namely, the Architecture and the Thing Description (TD). This chapter also introduces the different software architectures on which the Digital Dice proposed architecture is based, as well as the different middleware proposals for managing IoT and cyber-physical devices. Finally, the concept of event processors is introduced, and their importance in managing IoT and cyber-physical devices is discussed.

Chapter 2 presents the concept of Digital Dice and its relation to the Web of Things (WoT); the components of Digital Dice are described in detail, as well as its architecture and how internal communication between the pieces that form part of this architecture works. The metamodel that gives rise to the language (DSL) that helps to define the pieces or microservices that make up a Digital Dice is presented, as well as the data model used to store the information generated by IoT and cyber-physical devices that represent our Digital Dice. Finally, a model-to-text (M2T) transformation process is defined to establish the code base of the different components of a Digital Dice from the Thing Description that represents it.

In *Chapter 3*, the different capabilities of the Digital Dice architecture are described, representing the main contributions of the research work: the WoTnectivity communication model, the causality relationship model, and the high availability strategies of

Digital Dice. Firstly, WoTnectivity, a multiprotocol library for communication with IoT and cyber-physical devices, is presented. The basic concepts of the library and its usage are established; the protocols currently compatible with the library are presented, and how to implement other new protocols is defined. Secondly, the concept of causality relationships and their need for communication between Things is presented; the parts of the relationships are established, the DSL that allows defining these causality relationships is presented, and an example of using this language is shown. Finally, the mechanisms and strategies developed to facilitate the High Availability property of systems based on Digital Dice are presented in this chapter. The performance tests of Digital Dice are also presented, compared to direct connection with IoT devices governed by typical protocols such as KNX, ZigBee, or HTTP, using a base example in the field of Smart Cities for garbage collection.

In *Chapter 4*, details of the implementation of the Digital Dice model are provided, which was carried out through microservices. This chapter presents characteristics related to the technical aspects used for the deployment of Digital Dice components, and some considerations about the study and development of the algorithms implemented through microservices, validated in a series of proof-of-concept tests carried out in real and virtual environments.

Chapter 5 contains the conclusions of the research work, a summary of the contributions, and some limitations of the proposal presented in this doctoral thesis. In this chapter, the main contributions made will be described along with their relation to publications in national and international conferences, as well as in high-impact journals. Some critical considerations of the proposal and the lines of research that have been left open, which could be undertaken in future works as a continuation of this thesis work, will also be presented.

In general, each chapter contains a small *Introduction* where the objective pursued is exposed, and a brief description of the contents that will be addressed in each chapter. As a complement, at the end of this document, a list of acronyms and a series of annexes related to the implementations and validation documents of the models used by Digital Dice are provided. Finally, a section of the bibliography used during the development of the research work in this doctoral thesis is included.

Capítulo 1

Introducción y Fundamentos

Capítulo 1

Introducción y Fundamentos

Contenido

1.1. Introducción

El objetivo principal de este capítulo es establecer las bases del conocimiento previo, el estado de la situación y los fundamentos relacionados con el ámbito del trabajo de la investigación de esta tesis doctoral.

Este primer capítulo está estructurado de la siguiente manera. La Sección 1.2 realiza una breve introducción al concepto de Internet de las Cosas (Internet of Things, IoT), los dispositivos IoT, sistemas ciberfísicos, los Digital Twins como representaciones virtuales de este tipo de dispositivos y las Ciudades Inteligentes (Smart Cities) como dominio de aplicación del trabajo de investigación realizado. En la Sección 1.3 se presenta el estándar de la Web de las Cosas (Web of Things, WoT), su historia y estructura organizativa, y sus pilares fundamentales; Se hace especial hincapié en la especificación de la arquitectura de la WoT, que establece una forma de integrar los dispositivos IoT con la web, usando medios de comunicación y lenguajes comunes, y en la especificación de Thing Description (TD), la cual establece un esquema que permite definir los metadatos y las interfaces de un dispositivo conectado a la web, indicando cómo se puede interactuar con él y qué datos intercambia. En la Sección 1.4 se hace referencia a los estándares mas comunes en el ámbito de los dispositivos IoT y el motivo de la elección del ámbito WoT. En la Sección 1.5 se presentan las características de las arquitecturas de software que influencian en mayor medida a la solución propuesta en esta tesis. La Sección 1.6 presenta la necesidad de establecer un lenguaje de comunicación como *middleware* entre ecosistemas o redes de dispositivos IoT y dispositivos ciberfísicos. La Sección 1.7 establece las propiedades básicas para disponer un sistema software altamente disponible. Por último, la Sección 1.8 presenta los conceptos básicos del procesamiento de eventos complejos, necesario para la definición de una de las capacidades del modelo Digital Dice propuesto.

1.2. Internet de las Cosas

Internet de las Cosas (Internet of Things, IoT) es un campo de la Informática emergente y desafiante para los investigadores, especialmente en el área de la Ingeniería del Software, por su amplia gama de posibilidades de investigación por explorar, como es en seguridad, interoperabilidad, inteacción, localización de recursos, servicios, evolución, configuración, entre otros muchos aspectos. IoT comprende una red de objetos que están integrados con tecnologías que ayudan a comunicarse e interactuar de manera interna y también con el exterior. En esta sección se aborda el concepto IoT desde una perspectiva pragmática, su arquitectura a nivel de capas y algunos dominios de aplicación más comunes.

La revolución de Internet comenzó conectando computadores. Más tarde se conectaron muchas computadoras entre sí que dio pie a la creación de la World Wide Web. A posteriori, los dispositivos móviles fueron capaces de conectarse a Internet. Finalmente, la idea de conectar los objetos cotidianos a Internet condujo al concepto de Internet de las Cosas [Perera et al., 2015].

El término Internet de las Cosas (IoT) fue acuñado por primera vez Kevin Ashton en 1999 en su artículo titulado "*That 'Internet of Things' Thing*" [Ashton, 1999]. En su trabajo ya se hablaba del potencial de IoT para cambiar el mundo, tal como lo hizo el propio Internet. Más tarde, durante 2001, el centro MIT AutoID Lab presentó su punto de vista sobre IoT [Engels et al., 2001]. Luego, durante 2005, el termino IoT es formalmente reconocido por la Unión Internacional de Telecomunicaciones (International Telecommunication Union, ITU) [Union, 2005].

IoT crea un mundo donde todos los objetos (también llamados objetos inteligentes o *smart objects*) que nos rodean están conectados a Internet y se comunican entre sí sin necesidad de intervención humana. El objetivo final es crear un mundo mejor para los seres humanos, donde los objetos que nos rodean saben lo que nosotros queremos o necesitamos y son capaces de actuar en consecuencia sin instrucciones explícitas.

La investigación actual sobre el IoT principalmente se centra en cómo permitir que los objetos vean, escuchen e interactúen con el mundo físico por sí mismos, y hacerlos capaces de estar conectados para compartir estas observaciones. De manera muy genérica, trasladar la toma de decisiones y el monitoreo de los objetos del lado humano al lado de la máquina, y en la actualidad aplicando investigación puntera intercisciplinar en ingenierías del software y datos e inteligencia artificial.

El concepto del IoT se puede considerar como la conexión en red entre objetos o dispositivos físicos. Una de las definiciones más extendidas sobre el concepto IoT es la de [Madakam et al., 2015]: "*Una red abierta y completa de objetos inteligentes que tienen la capacidad de organizarse automáticamente, compartir información, datos y recursos, reaccionando y actuando en frente a situaciones y cambios en el entorno*".

Gracias a la ayuda de tecnologías de comunicación como son las redes de sensores inalámbricas (Wireless Sensor Networks, WSN) y la identificación mediante radio frecuencia (Radio Frequency Identification, RFID) se produce con más facilidad la compartición de la información. Con la popularización del IoT parece que cada vez más las personas y las cosas pueden estar conectadas en cualquier sitio, a cualquier hora y de múltiples maneras. En el año 2023 ya se cuenta con más de cincuenta billones de dispositivos IoT en el mundo [Hazra et al., 2023], algo que no parece que vaya a parar en los próximos años. La cantidad de dispositivos IoT supera ya con creces el número de personas en el mundo (seis billones y medio). El estudio de arquitecturas y la apertura de nuevos dominios de aplicación de estos dispositivos sigue siendo un nicho de investigación muy interesante. En cuanto a tipos de arquitecturas de ecosistemas IoT, se pueden clasificar dependiendo de las capas que se consideren en su desarrollo [Bandyopadhyay and Sen, 2011]. Básicamente existen cuatro capas distintas a tener en cuenta durante el desarrollo de sistemas IoT, representadas en la Figura 1.1:

(a) Capa de sensorización o capa Edge. Podemos considerar esta capa como la capa mas baja donde inciden las arquitecturas de IoT. Esta capa consiste básicamente en componentes hardware del estilo de redes de sensores, sistemas integrados, etiquetas RFID y lectores u otros tipos de sensores o actuadores. Este tipo de dispositivos son los dispositivos primarios generadores de datos que se despliegan en el terreno. Algunos de estos elementos de hardware proporcionan identificación y almacenamiento de información (por ejemplo, etiquetas RFID), recopilación de

información (por ejemplo, redes de sensores), procesamiento de información (por ejemplo, procesadores perimetrales integrados), comunicación, control y actuación.

(b) **Capa de puertas de enlace y red**. La primera parte del procesamiento de datos nace en esta capa. En esta capa se encarga del enrutamiento, publicación y suscripción de mensajes, recuperación de datos y también permite establecer la comunicación multiplataforma, si es necesario. Esta capa transfiere la información recogida por sensores a la siguiente capa. Se debe apoyar en protocolos universales, escalables, flexibles y estándar para transferencia de datos desde dispositivos heterogéneos (diferentes tipos de nodos sensores). Otras características deseables en esta capa podrían ser: contar con un alto rendimiento o la robustez.

(c) **Capa de servicios de gestión intermedia o middleware**. Esta es considerada una de las capas más críticas. Esta capa opera en modo bidireccional y actúa como una interfaz entre la capa de enlace y de red, y la capa de aplicación en la parte superior. Es responsable de funciones críticas como la gestión de dispositivos y gestión de la información, y también se ocupa de cuestiones como el filtrado de datos, la agregación de datos, análisis semántico, control de accesos, o el descubrimiento de información, entre otros aspectos.

(d) **Capa de aplicación**. Esta capa, situada en la parte superior de la pila, es donde residen las diferentes aplicaciones de las cuales hacen uso los usuarios de IoT. Las aplicaciones pueden pertenecer a diferentes verticales de la industria tales como: logística, fabricación, venta, medio ambiente, seguridad pública, atención médica, alimentos y productos farmacéuticos, etc. Con la creciente madurez de la tecnología ofrecida por los dispositivos IoT o los sistemas ciberfísicos, el campo de las aplicaciones IoT sigue evolucionando a marchas forzadas.

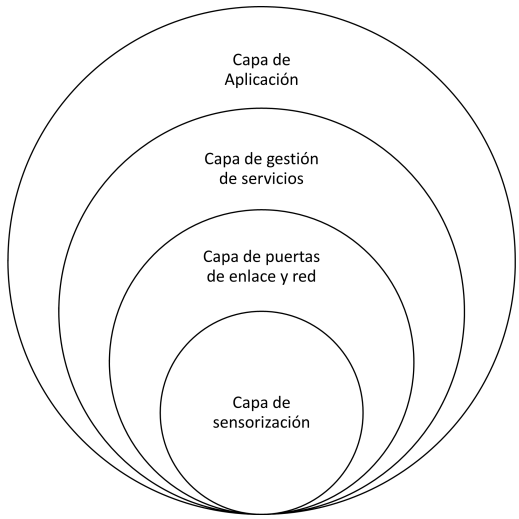


Figura 1.1: Clasificación por capas de una arquitectura IoT.

1.2.1. Sistemas ciberfísicos

Los sistemas ciberfísicos (Cyber-Phisical Systems, CPS) son sistemas de ingeniería que combinan elementos físicos, como máquinas y dispositivos, con componentes cibernéticos, como software y redes de comunicación, para operar de manera coordinada y eficiente [Pivoto et al., 2021]. Estos sistemas son fundamentales en la era digital, ya que permiten la integración y el intercambio de información en tiempo real entre el mundo físico y el digital. Algunos ejemplos de CPS incluyen vehículos autónomos, sistemas de control industrial, redes inteligentes y sistemas de salud.

Los CPS tienen varias características clave que los hacen especialmente relevantes en la actualidad. La primera es la capacidad de comunicarse y compartir datos con otros sistemas y dispositivos, lo que permite una interacción en tiempo real y una toma de decisiones más rápida. La segunda característica es la adaptabilidad, ya que estos sistemas pueden ajustar su comportamiento y operaciones en función de las condiciones cambiantes del entorno. La tercera es la autonomía, lo que significa que los CPS pueden realizar tareas de manera independiente sin intervención humana.

Los dispositivos IoT están estrechamente relacionados con los CPS, ya que ambos involucran la interacción entre el mundo físico y el digital. Sin embargo, los CPS se enfocan más en la coordinación y el control de sistemas complejos y en tiempo real, mientras que los dispositivos IoT se centran en la recopilación y el intercambio de datos a través de sensores y actuadores. A pesar de estas diferencias, los dispositivos IoT pueden ser considerados como componentes clave de los CPS, ya que facilitan la comunicación y el intercambio de información entre sistemas y dispositivos.

1.2.2. Digital Twins

Los Digital Twins, o Gemelos Digitales, son representaciones virtuales y dinámicas de objetos, procesos o sistemas del mundo real. Estas réplicas digitales permiten simular, analizar y predecir el comportamiento de sus contrapartes físicas en un entorno virtual, lo cual facilita la optimización de su rendimiento y la anticipación de posibles problemas.

Los dispositivos IoT y ciberfísicos son elementos fundamentales para el funcionamiento de los Digital Twins, ya que proporcionan la información necesaria para mantener actualizadas las réplicas digitales. Estos dispositivos IoT, que incluyen sensores, actuadores y otros dispositivos conectados, recolectan y transmiten datos en tiempo real sobre el estado, el rendimiento y la ubicación de los objetos o sistemas del mundo real. Esta información es procesada y analizada por el gemelo digital, que ajusta su modelo para reflejar las condiciones actuales y futuras.

La relación entre los Digital Twins, los dispositivos IoT y los sistemas ciberfísicos es bidireccional y simbiótica. Mientras que los dispositivos IoT proporcionan información esencial para la operación y actualización de los Digital Twins, estos últimos permiten optimizar y mejorar el rendimiento de los dispositivos IoT y los sistemas ciberfísicos en su conjunto. Esta sinergia facilita el desarrollo de soluciones más eficientes y efectivas para la industria, la logística, la infraestructura, la energía y otros sectores.

La mayor limitación de los Digital Twins es el hecho de que como meras representaciones virtuales no están diseñados para interactuar con el mundo real. Al amparo de

este problema nace nuestra propuesta de arquitectura de componentes digitales capaz de interactuar con los dispositivos físicos como si de un gestor de dispositivos se tratase, introduciendo el concepto Digital Dice.

El papel desempeñado por Digital Dice no se limita a la virtualización de los dispositivos, si no a la mejora de las capacidades de estos. Por un lado, mediante la utilización de estándares web para facilitar la interacción con estos dispositivos. Y por otro, aliviando el estrés que estos dispositivos pueden llegar a soportar cuando hablamos de escenarios en los que se exige un alto rendimiento.

1.2.3. Ciudades Inteligentes

Uno de los dominios más extendidos hoy día donde se está poniendo en práctica los avances de la investigación en dispositivos IoT y CPS es en las Ciudades Inteligentes, tal vez más conocido por su término en inglés "Smart Cities". Los modelos de ciudad inteligente son enfoques de planificación y gestión urbana moderna que utilizan tecnologías de la información y la comunicación (TIC) para mejorar la calidad de vida de sus habitantes, la eficiencia de los servicios urbanos y la sostenibilidad. Estas ciudades integran múltiples soluciones tecnológicas, como Internet de las Cosas (IoT), sistemas ciberfísicos (CPS), Big Data, análisis de datos e inteligencia artificial, para monitorear y optimizar, en tiempo real, infraestructuras de ciudad, como el tráfico, el consumo de energía, la seguridad y otros aspectos clave de la vida urbana.

En un modelo de ciudad inteligente, la información recopilada por sensores y dispositivos IoT, se utiliza para tomar mejores decisiones con mayor precisión y criterio. Esto permite a las administraciones, autoridades y a los propios ciudadanos gestionar mejor los recursos, reducir el consumo de energía y las emisiones de gases de efecto invernadero, mejorar la movilidad y la conectividad, y aumentar la resiliencia frente a desastres naturales y otros eventos adversos. Además, las ciudades inteligentes buscan fomentar la participación ciudadana y la colaboración entre los sectores público y privado, así como impulsar la innovación y el crecimiento económico.

Los dispositivos IoT cada vez más inciden en un mayor número de dominios o en diferentes campos de aplicación [Ali et al., 2021]. Los modelos de ciudad inteligente es, sin duda, el principal dominio de aplicación de la propuesta desarrollada en este trabajo de investigación, pudiendo llegar a tener un impacto directo en determinados escenarios de ciudad inteligente. Algunos escenarios estos escenarios, en los que se ha estado trabajando experimentalmente a lo largo de esta tesis, son:

- **Smart Homes**. Los hogares y las oficinas modernas utilizan con mayor frecuencia dispositivos IoT en su construcción, con aparatos electrónicos de última generación, como sistemas de climatización, luces, ventiladores, hornos microondas, refrigeradores y/o calefactores integrados con sensores y actuadores que permiten monitorizar y controlar todo tipo de parámetros y establecer alertas para, por un lado, minimizar el coste e incrementar el ahorro de energía, y por otro, mejorar el confort de los usuarios de este tipo de dispositivos.
- **Smart Buildings**. Los edificios inteligentes son estructuras avanzadas que integran tecnologías de automatización, sistemas de gestión energética y comunicación en tiempo real para mejorar la eficiencia, comodidad y seguridad de sus ocupantes.

Estos edificios cuentan con sensores, actuadores y dispositivos conectados a través de redes de comunicación, permitiendo la monitorización y el control centralizado de sistemas como iluminación, climatización, seguridad y gestión de recursos. Además, utilizan algoritmos de aprendizaje automático y análisis de datos para optimizar el consumo de energía y reducir las emisiones de carbono, adaptándose a las necesidades cambiantes de sus ocupantes y del entorno.

— **Smart Parking**. La adaptación de algunos aparcamientos a este tipo de tecnologías permite, por ejemplo, situar sensores en los huecos del aparcamiento para poder detectar si están disponibles o no. Los conductores pueden aparcar su vehículo simplemente mirando en una aplicación que les mostrase los detalles de los huecos qué hay en los parkings más cercanos.

— **Smart Waste Management**. Se trata de sistemas de gestión automatizada de residuos en modelos de ciudad inteligente. En este dominio de aplicación los contenedores y los camiones de recogida de basura quedan dotados por una red de sensores, generándose alertas que permiten saber cuándo un contenedor necesita ser vaciado. En ese momento, se puede añadir ese contenedor a la ruta de un camión de manera automática y sin intervención humana, estableciendo una heurística que lo añade a la ruta de los camiones que menos se tuviesen que desviar de su ruta principal y que siguen teniendo tanto capacidad de combustible, como suficiente hueco para cargar ese contenedor en los mismos.

— **Smart Ports**. Los puertos inteligentes son instalaciones portuarias que incorporan tecnologías avanzadas, automatización y soluciones de comunicación en tiempo real para optimizar las operaciones, mejorar la eficiencia y garantizar la sostenibilidad en el transporte marítimo y logística. Estos puertos utilizan sistemas de información y comunicación, como el Internet de las Cosas (IoT), Inteligencia Artificial (IA) y blockchain, para monitorear y controlar el tráfico marítimo, las operaciones de carga y descarga, el almacenamiento y el transporte terrestre. Además, los Smart Ports promueven la adopción de energías limpias y soluciones ecológicas, como el uso de vehículos eléctricos y la gestión eficiente de residuos.

1.3. Web de las Cosas

Normalmente, en proyectos clásicos de sistemas o dispositivos IoT, los desarrolladores se enfrentan a una serie de situaciones complicadas. Es común que los desarrolladores deban entender multitud de tecnologías heterogéneas que formen parte del dominio donde se está trabajando y que en determinadas ocasiones es necesaria la interconexión de diversos subsistemas y servicios IoT que provienen de distintos vendedores y fabricantes. Esta diversidad se deja notar especialmente en los tipos de protocolos de comunicación, modelos para el intercambio de datos, y requerimientos a nivel de seguridad. En la mayoría de los casos, las aplicaciones IoT se terminan desarrollando dando una solución específica para el dominio específico para el cual ha sido concebido, limitando la posibilidad de ser extendidas, ampliadas o integradas estas aplicaciones con otras nuevas u otras ya existentes. Esta vista tan cerrada a la hora de desarrollar aplicaciones provoca que sean muy difíciles de mantener, reutilizar o extender.

La Web de las Cosas, más conocida por su término en inglés como Web of Things, o simplemente WoT, es un nuevo paradigma del Internet de las Cosas propuesto por la organización W3C [WebThings, 2022] que provee el uso de un conjunto de normas y especificaciones que ayudan a simplificar el desarrollo de aplicaciones IoT.Esta aproximación incrementa la interoperabilidad y la flexibilidad, especialmente para aquellas aplicaciones en las cuales se trabaja con múltiples dominios, permitiendo la reutilización de soluciones y herramientas construidas.

1.3.1. Historia y estructura organizativa de la WoT

La Web of Things (WoT) es un concepto que surge a mediados de la década de 2000 como una extensión del paradigma del Internet de las cosas (IoT), con el objetivo de integrar dispositivos y servicios IoT en la World Wide Web. La idea clave detrás de la WoT es utilizar los principios y tecnologías web, como HTTP, REST y JSON, para simplificar la interoperabilidad y la comunicación entre dispositivos IoT y aplicaciones web. La WoT fue propuesta inicialmente por investigadores como Dominique Guinard y Vlad Trifa, quienes en 2009 publicaron un artículo titulado "Towards the Web of Things: Web Mashups for Embedded Devices" [Guinard and Trifa, 2009] en el Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web.

La WoT aboga por la aplicación de principios y arquitecturas web ya probados y estandarizados en el ecosistema IoT. Esto incluye el uso de protocolos de comunicación basados en la web, como HTTP y WebSockets, formatos de intercambio de datos como XML y JSON, y principios de diseño de aplicaciones, como los de la arquitectura REST. Al adoptar estas tecnologías, la WoT permite que los dispositivos y servicios IoT se integren de manera más sencilla y eficiente en la infraestructura de Internet existente, facilitando el desarrollo y la implementación de aplicaciones IoT a gran escala.

En cuanto a la organización WoT, el World Wide Web Consortium (W3C) ha sido un actor clave en la promoción y estandarización de la WoT. El W3C es una comunidad internacional que desarrolla estándares para garantizar el crecimiento a largo plazo de la web y su interoperabilidad. Dentro del W3C, se creó en 2014 el grupo de trabajo de la Web of Things (WoT Working Group) para desarrollar especificaciones y estándares que faciliten la adopción de la WoT en la industria.

El grupo de trabajo de la WoT (WoT Working Group) está compuesto por miembros de diferentes organizaciones y empresas, así como por expertos en tecnologías de la información y la comunicación. Este grupo se encarga de elaborar especificaciones técnicas y recomendaciones para abordar aspectos clave de la WoT, como la descripción de cosas (Thing Description, TD), los protocolos de enlace (Binding Protocols), la seguridad y la privacidad (privacy and security guidelines), etc. Además, el grupo de trabajo de la WoT colabora con otros grupos dentro del W3C y con organizaciones externas para garantizar la coherencia y la compatibilidad de los estándares desarrollados.

Además del grupo de trabajo de la WoT, también ha establecido grupos de interés (Interest Groups) y grupos de la comunidad (Community Groups) relacionados con la WoT y el IoT en general. Estos grupos tienen como objetivo fomentar la colaboración entre investigadores, desarrolladores y otros interesados en la WoT, así como fomentar la difusión de las especificaciones desarrolladas en el seno del grupo de trabajo.

1.3.2. Principales recomendaciones de la WoT

La Web of Things se subdivide en distintas partes, bloques o especificaciones. En la Figura 1.2 se puede observar las seis principales especificaciones que forman parte de la WoT. Estas especificaciones se dividen entre las que ya son recomendaciones, los documentos en los que aún se está trabajando, y las que se denominan entregables informativos. En el desarrollo de esta tesis, las especificaciones utilizadas han sido la WoT Thing Description y la WoT Architecture, las cuales se descibirán más ampliamente en las siguientes subsecciones. No obstante, es interesante conocer qué incluye cada una de las especificaciones presentadas en la citada figura.

— **WoT Architecture** [W3C, 2023a]. Esta recomendación describe la arquitectura abstracta de la WoT. Esta arquitectura abstracta se basa en una serie de requerimientos que fueron derivados de múltiples y distintos dominios de aplicación, los cuales se incluyen en el documento de la arquitectura. Gracias a este documento, se identificaron los distintos bloques que compone la WoT, y se establece cómo estos bloques se relacionan y trabajan conjuntamente. Define una serie de patrones comunes que se observan en los distintos dominios de aplicación estudiados en el estándar y que ilustran cómo los dispositivos o *things* interaccionan con controladores, otros dispositivos, agentes, y servidores.

— **WoT Thing Description** [W3C, 2022b]. Esta recomendación es el pilar central de la WoT. Resumidamente, la especificación permite establecer un modelo formal con el cual definir las distintas interacciones que una *thing* puede ejecutar. Este modelo formal se representa mediante un archivo JSON que representa una instancia del metamodelo que se establece en el documento de la recomendación.

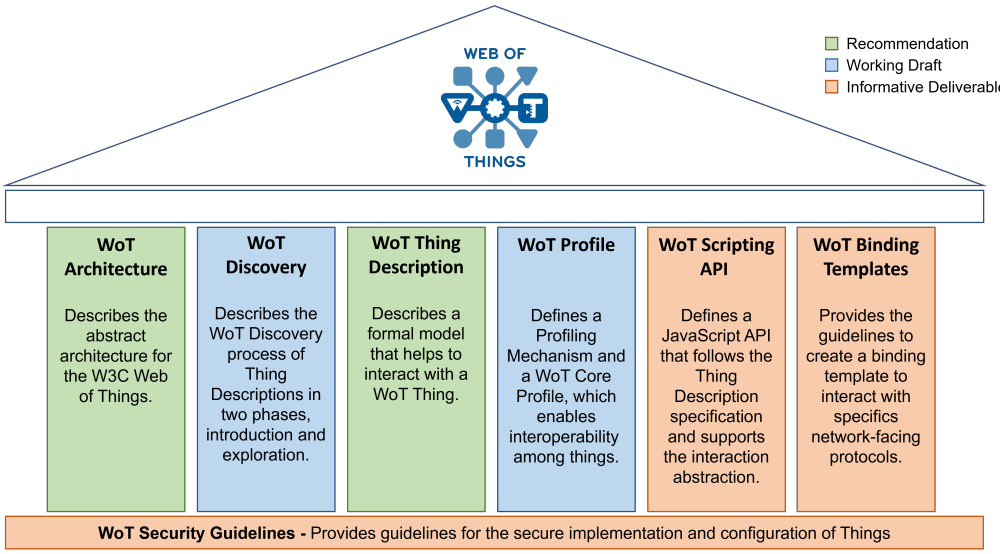


Figura 1.2: Especificaciones de la Web of Things.

— **WoT Discovery** [W3C, 2023b]. En el momento de la redacción de esta tesis, esta especificación está en fase de borrador, pero dentro de poco formará parte de la recomendación como tal. La Thing Description facilita la interoperabilidad entre distintas plataformas IoT y dominios de aplicación. Para poder utilizar una *thing* a través de la Thing Description, es necesario saber cómo se obtiene (identificación y localización). En este documento se establece la forma de ejecutar el proceso de descubrimiento, que permite la identificación y localización de *things*.
— **WoT Profile** [W3C, 2023c]. Al igual que la anterior, este documento es aún un borrador. Esta especificación proporciona un mecanismo de generación de perfiles, y un sistema denominado WoT Core Profile, el cual intenta establecer interoperabilidad directa entre dispositivos y/o *things* a través de esos perfiles.
— **WoT Scripting API** [W3C, 2023d]. Este entregable informativo es considerado un bloque opcional dentro de la WoT, y presenta una interfaz de programación (API) para operar las distintas *things* descritas mediante Thing Descriptions de manera automática. Este documento presenta una implementación básica del estándar y proporciona un *runtime* adecuado para el manejo de WoT Things.
— **WoT Binding Templates** [W3C, 2023f]. Este documento permite que una Thing Description sea adaptada a un protocolo o tipo de dato especifico de una tecnología diferente a las contempladas en primera instancia por la WoT (HTTP, COAP, MQTT, etc.). El documento describe un vocabulario inicial que permite la extensión de la Thing Description. A lo largo del ciclo de vida del estándar se espera que se añadan protocolos adicionales y nuevas estructuras de datos que permitan seguir extendiendo las WoT Binding Templates.
— **WoT Security and Privacy Guidelines** [W3C, 2023e]. Este documento proporciona una guía no informativa sobre los aspectos de seguridad y privacidad dentro de un sistema WoT. El documento define una serie de requerimientos de seguridad completamente opcionales dentro de un sistema WoT utilizando un modelo de posibles amenazas.

La creación del estándar de la Web of Things por parte de la W3C, así como la definición de los distintos pilares que lo sustentan, persigue explorar el futuro de la web física [Sneps-Sneppe and Namiot, 2016]. Su objetivo fundamental es la creación de ecosistemas abiertos escalables y flexibles de sistemas ciberfísicos y/o dispositivos IoT utilizando tecnologías web como capa de aplicación.

La solución que se presenta en el trabajo de investigación de esta tesis doctoral hace uso de una parte reducida de las especificaciones definidas por la WoT, con especial interés en el documento de Thing Description (TD) como modelo de definición y documentación de la funcionalidad de un Digital Dice. El uso de plantillas Thing Description proporciona a los desarrolladores la posibilidad de describir, de manera formal, esta funcionalidad, y a su vez facilita la compatibilidad directa del dispositivo o el sistema que se esté describiendo con la TD, con otros que utilicen este estándar. Por otro lado, elevar el nivel de abstracción mediante una plantilla TD para describir la funcionalidad del sistema proporciona una mejor legibilidad para los desarrolladores que trabajen con una solución basada en Digital Dice. Al final, las plantillas Thing Description de la WoT son, para los dispositivos IoT o ciberfísicos, unas abstracciones virtual de los mismos, de la misma manera que OpenAPI lo es para una API RESTful.

1.3.3. Arquitectura de la WoT

La especificación de la arquitectura WoT [W3C, 2023a] explica, de manera general, cómo los pilares que la sustentan (comentados anteriormente) se interrelacionan entre sí. Por otro lado, la especificación cubre aspectos arquitectónicos no normativos y condiciones para desplegar sistemas WoT. Esto se describe mediante una serie de escenarios de implementación mostrados en la normativa. No obstante, estos escenarios no buscan definir la normalización de las implementaciones específicas concretas, esto queda a mano de los distintos desarrolladores. Esta especificación sirve como paraguas para el resto de especificaciones de W3C WoT y define los conceptos básicos, como la terminología y la arquitectura abstracta subyacente de la Web of Things del W3C. En resumen, el propósito de esta especificación es proporcionar:

(a) Un conjunto de casos de uso que conducen a la arquitectura WoT del W3C.

(b) Un conjunto de requisitos para las implementaciones de la WoT.

(c) Una definición de la arquitectura abstracta.

(d) Una visión general de de bloques de construcción de WoT y su interacción.

(e) Una guía informativa sobre cómo mapear la arquitectura abstracta a posibles implementaciones concretas.

(f) Ejemplos informativos de posibles escenarios de implementación.

(g) Ejemplos de implementaciones de WoT.

(h) Una discusión, a un alto nivel, de aspectos de seguridad y privacidad a tener en cuenta al implementar un sistema basado en la arquitectura WoT.

Esta subsección se centra en explicar la terminología básica para entender cómo funciona la propuesta presentada en esta tesis doctoral. Se expondrá un resumen general de la normativa de la arquitectura y los patrones de comunicación con controladores, otros dispositivos, agentes o servicios.

Uno de los conceptos más importantes por definir es el de una **Thing** en el contexto de la Web de las Cosas.

Definición 1.1 (Thing) *Una **Thing**, es una abstracción de una entidad física o virtual (por ejemplo, una bombilla o un edificio) cuyos metadatos e interfaces están descritos mediante una plantilla WoT Thing Description.*

Los consumidores de Things deben poder analizar y procesar el formato de representación de la Thing Description (TD). El formato se puede procesar a través de bibliotecas JSON clásicas o un procesador JSON-LD [Longley et al., 2014], ya que el modelo de información subyacente está basado en gráficos y su serialización es compatible con JSON-LD 1.1 [Sporny et al., 2020]. El uso de un procesador JSON-LD para procesar una TD también permite el procesamiento semántico, incluida la transformación a tuplas

RDF, la inferencia semántica y la realización de tareas basadas en términos ontológicos, lo que haría que los consumidores se comporten de manera más autónoma. Una TD es específica a una estancia particular (es decir, no tipos de Things) y muestra la representación textual (Web) de las capacidades de una Thing predeterminada.

Para ser considerado una Thing, al menos una representación de la TD debe estar disponible. La WoT Thing Description [W3C, 2022b] es un formato estándar que debe ser comprensible por usuarios y máquinas que permite a los consumidores descubrir e interpretar la funcionalidad de una Thing y adaptarse a diferentes implementaciones (por ejemplo, diferentes protocolos) al interactuar con una Thing, lo que permite la interoperabilidad entre diferentes plataformas relacionadas con el ecosistema IoT.

Una Thing también puede ser la abstracción de una entidad virtual. Una entidad virtual puede estar compuesta de una o más Things (por ejemplo, un edificio con sensores y actuadores). Una opción para las composiciones podría ser proporcionar una TD única y consolidada que contenga el superconjunto de capacidades de la entidad virtual. Aunque, en los casos en que la composición es bastante compleja, su TD puede vincularse a sub-Things jerárquicas dentro de la composición. El TD principal podría actuar como punto de entrada tan solo contener metadatos y capacidades generales. Mientras que las sub-Things podrían agrupar ciertos aspectos que tuviesen sentido.

La vinculación no sólo se aplica a las Things cuando se habla de jerarquías, sino que también se pueden aplicar a las relaciones entre las Things y otros recursos externos e internos. Los tipos de relación de enlace (o Links) muestran cómo las Thing se relacionan, por ejemplo, un interruptor que enciende un ventilador o una puerta monitoreada por un sensor de movimiento. Otros recursos que podrían estar en una relación de enlace son los manuales de uso, catálogos de repuestos, archivos CAD, una interfaz de usuario gráfica o cualquier otro documento en la Web. En general, la vinculación web entre las Things hace que la WoT sea navegable, tanto para humanos como para máquinas. Esto se puede facilitar aún más proporcionando directorios de Things que administran un catálogo de Things disponibles, generalmente almacenando en caché la representación TD. En resumen, las WoT Thing Description pueden vincularse a otras Things y otros recursos en la Web para formar una Web de Cosas.

Las Things deben estar alojadas en componentes que tengan acceso a red, con software que permita realizar la interacción a través de una interfaz orientada a la red. Un ejemplo de esto es un servidor HTTP que se ejecuta en un dispositivo integrado con sensores y actuadores. Sin embargo, la WoT no obliga a conocer dónde deban estar alojadas las Things; podrían estar alojadas directamente en un dispositivo IoT, en el Edge, o en la nube. Un desafío de implementación podría ser un escenario en el que no se puede acceder a las redes locales desde Internet. Para remediar esta situación, WoT permite mediación (patrón usada en esta tesis) entre las *things* y los consumidores.

Definición 1.2 (Intermediario) *Un* ***Intermediario*** *de la WoT, también conocido en la literatura como ORB,* mediador *o* trader*, puede actuar como proxies para las Things; Este intermediario debe tener una TD muy similar a la Thing original, pero que apunta a la interfaz WoT proporcionada por el intermediario. Los intermediarios también pueden extender las Things existentes con capacidades adicionales o componer una nueva Thing a partir de múltiples Thing disponibles, formando así una entidad virtual.*

En la Figura 1.3 se muestra el comportamiento de un patrón de mediación, que es parecido al de un ORB (Object Request Broker). Para los consumidores, los intermediarios se parecen a las Things normales, ya que poseen TD y proporcionan una interfaz WoT, y por lo tanto pueden ser indistinguibles de las Things en una arquitectura de sistema en capas como la Web. Un identificador en la TD debe permitir la correlación de múltiples TD que representan la misma Thing original o la entidad física única.

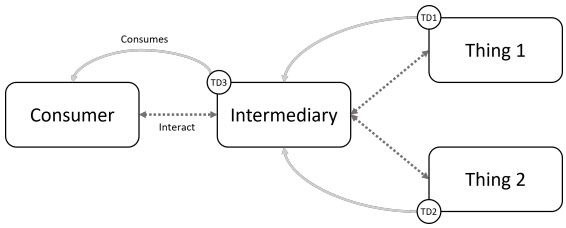


Figura 1.3: Intermediario de Things

En soluciones basadas en la Web de las Cosas (WoT), como la que se presenta en la propuesta de esta tesis, un intermediario se denomina también como Servient software o simplemente **Servient**.

Definición 1.3 (Servient) *Un **Servient** es un software que implementa la norma de mediación de la WoT. Un Servient puede exponer y/o consumir Things.*

Los Servients pueden admitir múltiples enlaces de protocolo para permitir la interacción con diferentes plataformas de IoT. A parte de actuar como un Servient software de la WoT, la solución propuesta tiene la capacidad de actuar como un sistema muy parecido a los Digital Twins [Singh et al., 2021], aunque este tipo de propuestas suelen estar mayormente dirigidas a lo que seria la virtualización completa de los dispositivos mientras que nuestra solución aúna dos campos, virtualización y gestión.

1.3.4. WoT Thing Description

Una Thing Description (TD) describe los metadatos e interfaces de *things* en formato JSON, siendo una *thing* una abstracción de una entidad virtual o física que proporcione interacciones que participen en escenarios relacionados con el campo de la WoT. TD proporciona un conjunto de interacciones basadas en un pequeño vocabulario que hace posible integrar dispositivos diversos y permite a distintas aplicaciones interoperar con ellos. Por otro lado, el estándar proporciona un JSON-LD que permite validar de manera automática la taxonomía y el contenido de una TD.

En el listado de la Figura 1.4 se puede observar un ejemplo de una TD. En este ejemplo se declara una simple bombilla de luz que cuenta con una propiedad denominada `status` que devolverá verdadero o falso dependiendo si la luz está encendida o apagada. Por otro lado, la TD cuanta con una acción denominada `switch` que permite el encendido o apagado de la bombilla.

```
{
"@context": "https://www.w3.org/2019/wot/td/v1",
"id": "acg:lab:light",
"title": "ACG Lab Light",
"properties": {
 "status": {
  "type": "object",
  "properties": {
   "value": {
    "type": "boolean"
  }},
  "forms": [{
   "href": "https://example.com/acg:lab:light/property/status/",
   "contentType": "application/json"},
  {
   "href": "https://example.com/acg:lab:light/property/status/sse",
   "subprotocol": "sse",
    "response": {
     "contentType": "text/event-stream" }}]
}},
"actions": {
 "switch": {
  "input": {
  "type": "object",
   "properties": {
    "value": {
     "type": "boolean"
   }}},
   "required": [ "value" ],
    "forms": [{
     "href": "https://example.com/acg:lab:light/action/switch/",
     "contentType": "application/json"
  }]}
}
}
```

Figura 1.4: Plantilla Thing Description de un Digital Dice Light.

En la Figura 1.5, se observa una representación en forma de diagrama UML del esquema que nos permite definir distintas TD [W3C, 2022b]. En la citada figura se puede observar cómo la clase principal Thing está formada a partir de un conjunto de acciones, propiedades y eventos que esta puede gestionar. Las acciones, propiedades y eventos de una Thing Description son subclases de una denominada en la WoT como `InteractionAffordance`. Esta `InteractionAffordance` representa los metadatos de una Thing que muestra las posibles elecciones con las que cuenta un consumidor a la hora de interactuar con la Thing. Como se observa en la Figura 1.5, estas `InteractionAffordance` están formadas por uno o más `Form` o maneras de acceder a los datos. A su vez, y dependiendo también del tipo de `InteractionAffordance` que estemos intentando invocar, contaremos con un `DataSchema` en el cual podemos definir qué tipo de respuesta va a devolver cada una de esas `InteractionAffordance` o qué tipo de datos van a ser necesarios para poder interactuar de una manera adecuada.

Otro aspecto a destacar de la TD es el campo denominado `Link`. Un `Link` puede verse como una declaración de relación entre nuestra Thing y el objetivo al que apunta el `Link`. En el contexto de la solución propuesta en esta tesis doctoral, utilizamos los `Link` para definir relaciones entre una o más `Things`.

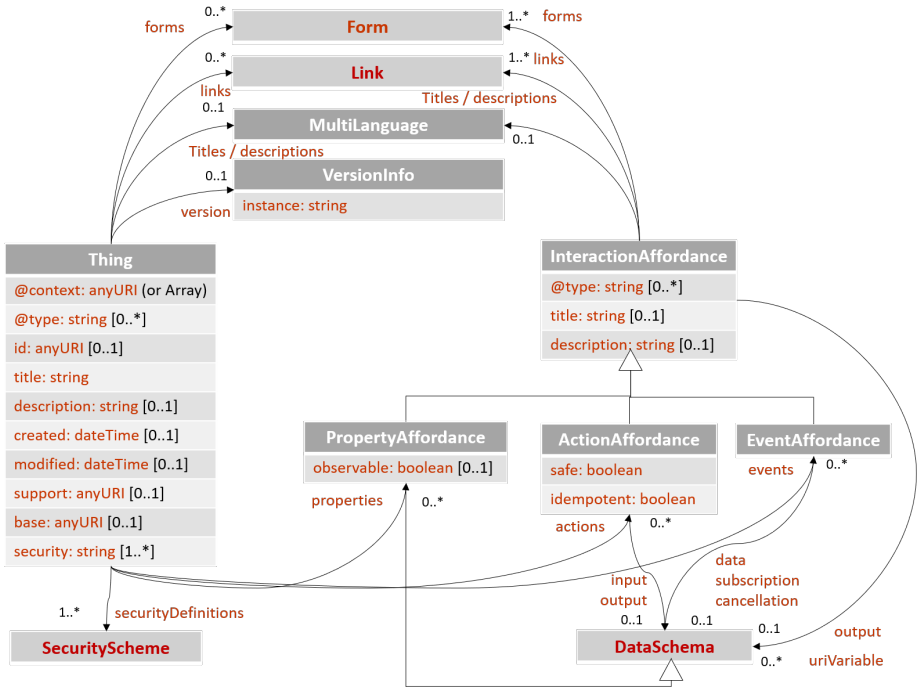


Figura 1.5: Modelo de la Thing Description.

Por último, una TD permite definir a nivel de interacción o de Thing, cierta información semántica con la propiedad `@type`. En la propuesta que se realiza en esta tesis, esta propiedad se usa para varios propósitos, por ejemplo, para definir si una Thing es virtual o no, o si alguna interacción utiliza algún tipo de tecnología no contemplada por la TD. Este campo semántico permite establecer cierto comportamiento no estándar, el cual podrá ser interpretado por la propuesta para extender la compatibilidad de esta a tecnologías fuera del alcance de los modelos de definición establecidos por la TD.

Al final, el propósito que busca la WoT con la TD, es muy similar a lo que OpenAPI es para las API REST: un lenguaje de definición de operaciones. De hecho, la WoT adopta el método API REST como el modelo estándar de definición de las rutas de cada una de las interacciones. Si hacemos una pequeña comparación entre la figura y el ejemplo que hemos mostrado en esta sección, podemos observar que no todos los campos definidos por el lenguaje son requeridos a la hora de definir una Thing. En la especificación [W3C, 2022b], se puede observar qué campos son obligatorios y cuáles son opcionales. Este lenguaje establece una gran cantidad de campos opcionales porque trata de ser compatible con una amplia casuística, pero en la mayoría de los casos no todas las propiedades serán necesarias para representar los metadatos de una Thing.

Los conceptos WoT y en particular TD son bastante más complejos que el simple resumen realizado en esta subsección. Se ha centrado en `InteractionAffordance`, `Link` y `DataSchema` para facilitar la explicación de la propuesta más adelante.

1.4. Estándares de la industria relacionados

En esta sección se describen brevemente los principales estándares o normas relacionados con el ámbito de IoT, sus orígenes y las principales características. Son muchos los que hay en la literatura, pero se ha centrado el estudio en tres de ellos, por su relevancia con la propuesta desarrollada en la presente tesis doctoral.

En general, todos los estándares, incluyendo la WoT, buscan la interoperabilidad de los dispositivos IoT, pero desde distintos enfoques. La elección de la Web de las Cosas (WoT) viene dada por las ventajas que tiene frente a estos estándares. En lugar de limitarse a la interoperabilidad de dispositivos IoT, la WoT utiliza estándares web abiertos como HTTP y JSON para permitir la interconexión y la interoperabilidad de cualquier dispositivo o servicio a través de la web, lo cual facilita el manejo de los dispositivos, dado que estas tecnologías son de conocimiento común para la comunidad de desarrolladores. Esto significa que los dispositivos y los servicios pueden comunicarse entre sí de manera más fluida y segura, incluso si pertenecen a diferentes ecosistemas tecnológicos. Además, la WoT proporciona una arquitectura más escalable y adaptable que se adapta fácilmente a las necesidades de diferentes aplicaciones y escenarios. Por ejemplo, se puede utilizar para controlar sistemas de domótica en el hogar, monitorear el rendimiento de la maquinaria industrial o para optimizar la logística en el transporte.

1.4.1. La norma ISO/IEC 30141

El primer estándar de obligado nombramiento es la norma ISO/IEC 30141 sobre Internet de las Cosas [ISO/IEC, 2018]. Esta norma fue desarrollada por la Organización Internacional de Normalización (ISO) y la Comisión Electrotécnica Internacional (IEC) para establecer un marco de referencia que permitiera a los sistemas de IoT interconectarse e interoperar de manera más eficiente y segura. El estándar ISO/IEC 30141 se basa en tres pilares fundamentales: la identificación de los dispositivos y los servicios, la comunicación entre los dispositivos y la seguridad. El estándar define una arquitectura de referencia para la interoperabilidad de sistemas IoT que abarca seis capas: aplicación, gestión de datos, gestión de dispositivos, conectividad, infraestructura y entorno. Además, establece recomendaciones para la gestión de identidades, el intercambio de datos y la privacidad de la información. Entre las limitaciones de este estándar destacan el hecho de que se enfoca principalmente en la comunicación entre dispositivos y no aborda temas como la interoperabilidad de los datos y la integración con otras tecnologías emergentes como la inteligencia artificial y el aprendizaje automático.

1.4.2. La norma IoT-A

En segundo lugar está la norma IoT-A [Slange, 2013]. Este estándar fue desarrollado por la Unión Europea en colaboración con expertos en tecnología de todo el mundo para establecer una arquitectura de referencia que permita la interoperabilidad y la integración de los sistemas de IoT de manera eficiente y segura. La arquitectura de referencia del IoT-A se divide en cuatro capas: la capa de sensores y actuadores, la capa de red, la capa de servicios y la capa de aplicación. Cada una de estas capas

contiene diferentes componentes que trabajan juntos para proporcionar una plataforma integral y escalable para los sistemas de IoT. Entre los principales objetivos del IoT-A se encuentran la interoperabilidad, la escalabilidad, la seguridad y la sostenibilidad. Para lograr estos objetivos, el estándar define un conjunto de principios, buenas prácticas y recomendaciones para la implementación de sistemas de IoT. El mayor problema de este estándar es que se enfoca principalmente en la interoperabilidad técnica y no considera suficientemente aspectos como la gobernanza, la ética y la privacidad de los datos.

1.4.3. La Serie Y de ITU-T

Por último está la Serie Y de ITU-T [ITU-T, 2012], concretamente la serie Y.2060 sobre la visión general de Internet de las Cosas, desarrollado por la Unión Internacional de Telecomunicaciones (ITU) para establecer recomendaciones técnicas para la implementación de sistemas de telecomunicaciones y tecnologías de la información, incluyendo IoT. La serie de recomendaciones de la ITU-T para IoT establece un marco para la implementación de sistemas de IoT, que incluye una arquitectura de referencia, protocolos de comunicación, requisitos de seguridad, y otros aspectos técnicos relevantes. El estándar ITU-T también proporciona pautas para la gestión de la información y los servicios de IoT, así como para la interoperabilidad y la compatibilidad de los sistemas. Una de las ventajas del estándar ITU-T es que es ampliamente aceptado y utilizado por los proveedores de servicios de telecomunicaciones y tecnologías de la información en todo el mundo. Esto significa que los sistemas que se implementan de acuerdo con las recomendaciones de la ITU-T tienen una alta probabilidad de ser compatibles con otros sistemas existentes siempre y cuando hablemos de aspectos técnicos. Aquí se encuentra su principal limitación, sólo se centra en aspectos técnicos y no aborda temas como la seguridad de la información, la privacidad de datos o la ética en el uso de la tecnología.

1.5. Arquitecturas Software

Una arquitectura software es una representación de alto nivel que describe la estructura de un sistema software y sus componentes. En este contexto, un sistema software es un conjunto de componentes software que interactúan entre sí para lograr un objetivo común. En la literatura, se han definido diferentes tipos de arquitecturas software, como por ejemplo, arquitecturas orientadas a servicios, arquitecturas basadas en componentes, arquitecturas orientadas a recursos, etc. En esta sección vamos a definir las arquitecturas software que han influido en la solución propuesta durante en el desarrollo de esta tesis.

1.5.1. Arquitecturas orientadas a servicios

Una arquitectura orientada a servicios (Service-Oriented Architecture, SOA) [Barry and Gannon, 2003] es un estilo de arquitectura software que define una aplicación como un conjunto de servicios. Estos servicios son independientes, y se comunican entre sí a través de un protocolo de comunicación (HTTP, MQTT, COAP, etc.). Los servicios pueden ser desarrollados por diferentes equipos, en diferentes lenguajes de programación

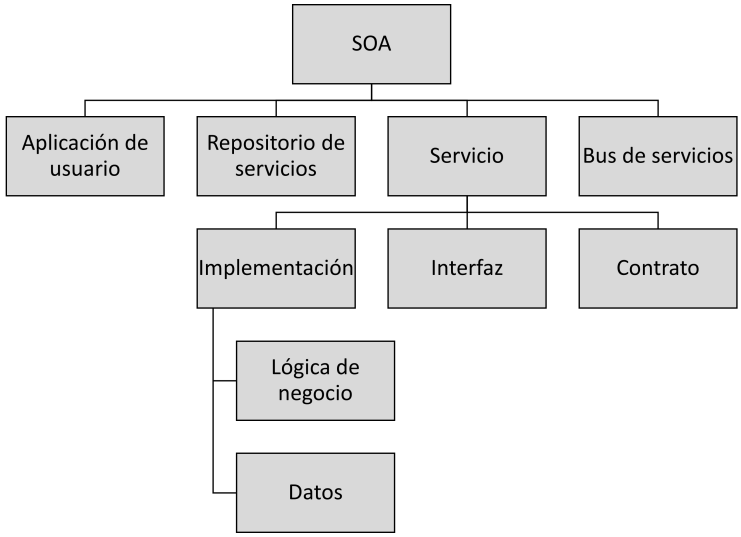


Figura 1.6: Diagrama de bloques de una arquitectura SOA.

y pueden ser desplegados en diferentes máquinas. Esta elasticidad y flexibilidad es una de las principales ventajas de las arquitecturas orientadas a servicios.

Un servicio es un componente software que tiene una funcionalidad concreta a la que podremos acceder de manera remota y actúa de manera independiente. Podemos pensar en estos servicios como una caja negra que nos ofrece un comportamiento específico a través de una interfaz. El concepto de orientación a servicios se basa en la idea de que una aplicación puede ser construida a partir de servicios que se comunican entre sí. En la Figura 1.6 podemos observar los distintos elementos que pueden formar parte de una arquitectura orientada a servicios.

Este tipo de arquitectura se caracteriza por ser altamente escalable, muy flexible y fácil de mantener. El estilo arquitectónico SOA suele presentar una serie de características de diseño que lo definen, y que debemos tener en cuenta a la hora de desarrollar soluciones basadas en este tipo de arquitecturas. Estas características son las siguientes:

— **Acoplamiento bajo**. El acoplamiento es el grado de dependencia entre los componentes de un sistema. Cuanto menor sea el acoplamiento, más fácil será modificar el sistema sin afectar a los demás componentes. En este sentido, los servicios deben ser lo más independientes posible.
— **Reusabilidad de los servicios**. Los servicios deben ser lo más reutilizables posible. De esta forma, si un servicio se utiliza en varias aplicaciones, no será necesario desarrollar el servicio de nuevo para cada aplicación.
— **Autonomía de los servicios**. Una característica muy deseable en los servicios es que sean lo más autónomos posible. De esta forma, si un servicio falla, no afectará al resto de servicios.
— **Servicios sin estado**. Los servicios deben ser *stateless*, es decir, no deben guardar ningún estado. Este hecho determina que los servicios no deben guardar ningún

tipo de información en memoria, ya que si el servicio falla, se perderá toda la información. En este sentido, los servicios deben ser lo más *stateless* posible.

— **Composición de servicios**. Los servicios deben tener una interfaz bien definida, es decir, deben tener una descripción de cómo se comunican con el resto de servicios. Esta descripción debe ser lo más precisa posible, para que los servicios tengan la capacidad de coordinarse normalmente mediante un bus de servicios. A su vez, el tener una interfaz bien definida ayuda a descomponer problemas complejos en problemas más simples. Esta característica ayuda en el descubrimiento de nuevos procesos de negoció a partir de servicios ya existentes.

— **Interoperabilidad entre servicios**. Los servicios que desarrollemos deben usar en la mayor medida de lo posible estándares abiertos. Algunos de estos estándares podrán ser de transmisión de datos (HTTP, MQTT, COAP, etc.), de descripción de servicios (WSDL, UDDI, REST, etc.) o de descripción de datos (XML, JSON, AVRO, etc.). De esta forma, los servicios podrán ser reutilizados en diferentes aplicaciones sin importar la tecnología que se use para desarrollarlos.

Aunque las arquitecturas orientadas a servicios son muy flexibles y escalables, también presentan algunas desventajas. Entre las mas importantes podemos destacar que: (a) las arquitecturas orientadas a servicios pueden ser muy complejas de desarrollar y mantener; (b) estas arquitecturas son muy dependientes de los estándares que utilicen los servicios; y (c) son arquitecturas muy difíciles de depurar y realizar pruebas en un sistema que se compone de muchos servicios, dado que no solo tenemos que tener en cuenta los servicios de manera independiente, sino que debemos tener cuidado con los problemas derivados de la integración de servicios [Almonaies et al., 2010].

1.5.2. Arquitecturas de microservicios

Las arquitecturas de microservicios son una evolución de las arquitecturas orientadas a servicios. En este tipo de arquitecturas, los servicios se dividen en microservicios: otros servicios lo suficientemente pequeños como para que cada uno de ellos pueda ser desarrollado por un equipo de desarrollo independiente, lo cual facilita la escalabilidad y la flexibilidad de la arquitectura, incluso más que en las arquitecturas orientadas a servicios tradicionales [Nadareishvili et al., 2016].

En las arquitecturas orientadas a servicios pueden contar con servicios monolíticos, es decir, servicios que contienen toda o mucha de la funcionalidad de la aplicación. En est sentido, si se deseara modificar la funcionalidad de un servicio, se tendría que hacer para todo el servicio completo, siendo esto un gran problema. En las arquitecturas de microservicios, los servicios se dividen en microservicios, de esta forma, al modificar la funcionalidad de un servicio, solo se realizaría sobre el microservicio que contiene la funcionalidad que queremos modificar. La idea es subdividir en la mayor medida de lo posible el servicio en pequeñas unidades funcionales y autónomas.

Alrededor de este tipo de arquitecturas, en la industria ha surgido una serie de soluciones orientadas a la creación, gestión y monitorización de microservicios. Entre estas soluciones hay que destacar por influencia en el desarrollo de la investigación de la tesis doctoral: Docker, Kubenetes e Istio.

Docker Se trata de una plataforma de código abierto que permite a los desarrolladores y administradores de sistemas automatizar el proceso de creación, implementación y ejecución de aplicaciones dentro de contenedores [Nickoloff and Kuenzli, 2019]. Un contenedor es una unidad de software que incluye todo lo necesario para ejecutar una aplicación, incluyendo el código, las dependencias, las bibliotecas y el entorno de ejecución. La funcionalidad principal de Docker se basa en la tecnología de contenedores, que es una alternativa más liviana y eficiente a las máquinas virtuales tradicionales. Los contenedores permiten aislar las aplicaciones y sus dependencias, lo que facilita la ejecución en diferentes entornos sin problemas de compatibilidad. Por otro lado, Docker ofrece la posibilidad de crear entornos consistentes y aislados, mejora la eficiencia en el uso de recursos y promueve la colaboración y reutilización de componentes de software.

Kubernetes Kubernetes [Burns et al., 2022], también conocido como K8s, es un sistema de código abierto para la automatización, escalabilidad y gestión de aplicaciones en contenedores, como los creados con Docker. Fue diseñado por Google y se basa en su experiencia con la gestión de aplicaciones a gran escala en contenedores. Hoy en día, Kubernetes es mantenido por la Cloud Native Computing Foundation (CNCF). La mayor fortaleza de Kubernetes con respecto a Docker es que permite realizar automatismos en lo referente al escalado de contenedores de manera automática. Las principales funcionalidades de Kubernetes son la capacidad de encargarse de la gestión del ciclo de vida de los contenedores, incluyendo la creación, el escalado, la actualización y la eliminación. La capacidad de distribuir automáticamente el tráfico entre los pods de un servicio para garantizar una distribución uniforme de la carga y la alta disponibilidad. La detección de fallos en los contenedores para reiniciarlos y remplazarlos automáticamente y de esa manera mantener la aplicación funcionando sin interrupciones. Y facilitar la implementación de nuevas versiones de aplicaciones lo cual permite realizar actualizaciones progresivas con control de versiones y rollback si es necesario.

Istio Istio [Larsson et al., 2020] es una malla de servicios (*service mesh*) de código abierto desarrollada por Google, IBM y Lyft que se ejecuta en plataformas de orquestación de contenedores como Kubernetes. Las mallas de servicios ofrecen una infraestructura dedicada que gestiona la comunicación entre los microservicios que componen una aplicación. A su vez, proporciona una serie de funciones clave como el descubrimiento de servicios, balanceo de carga, encriptación, autenticación y autorización, monitoreo y trazabilidad de las solicitudes sin necesidad de implementar estas características en cada microservicio individualmente.

1.5.3. Arquitecturas orientadas a funciones

Las arquitecturas orientadas a funciones, también conocidas como arquitecturas sin servidor (serverless) o Functions as a Service (FaaS) [Jonas et al., 2019], se centran en el despliegue y la ejecución de funciones individuales y autocontenidas en respuesta a eventos específicos. Esto permite a los desarrolladores enfocarse en la lógica de negocio, mientras que la administración y el mantenimiento de la infraestructura subyacente recaen en los proveedores de servicios en la nube.

Una característica clave de las arquitecturas FaaS es la escalabilidad automática, que ajusta dinámicamente los recursos en función de la demanda. Cuando se desencadena un evento, el sistema crea automáticamente una instancia de la función correspondiente y la ejecuta. Si la demanda aumenta, se crean múltiples instancias para manejar las solicitudes concurrentes, lo que permite a las aplicaciones escalar fácilmente.

En el modelo FaaS, los desarrolladores pagan solo por el tiempo de ejecución de sus funciones, en lugar de pagar por recursos reservados como en los modelos tradicionales de infraestructura como servicio (IaaS) o plataforma como servicio (PaaS). Esta característica hace que las arquitecturas orientadas a funciones sean especialmente atractivas para aplicaciones con patrones de carga variables o impredecibles.

Las arquitecturas *serverless* promueven un enfoque modular y desacoplado en el diseño de aplicaciones. Las funciones en un entorno FaaS son unidades de código independientes que se ejecutan en respuesta a eventos específicos, lo que facilita la reutilización de código, la separación de responsabilidades y la implementación de actualizaciones.

Uno de los desafíos en las arquitecturas FaaS es el tiempo de arranque en frío, que se refiere al tiempo necesario para crear una instancia de una función y ponerla en marcha. Este tiempo de arranque puede afectar la latencia de la aplicación, especialmente si las funciones no se ejecutan con frecuencia. Por otro lado, establecer sistemas de monitoreo y gestión de las funciones es muy complejo, ya que no existe un estándar que permita realizar estas tareas en las funciones de manera homogénea [Jacobs et al., 2022].

1.5.4. Arquitecturas orientadas a recursos

Al igual que las arquitecturas orientadas a microservicios, las arquitecturas orientas a recursos (ROA) son una evolución de las arquitecturas orientadas a servicios. En ROA los servicios se dividen en recursos, que son servicios mucho más pequeños que los servicios de las SOA. Este tipo de arquitecturas casa muy bien con las arquitecturas orientadas a microservicios, ya que podemos considerar que cada uno de los recursos sea un microservicio.

Uno sus mayores exponentes son las arquitecturas de tipo REST. En REST, los recursos se identifican mediante una URI, de forma que cada recurso tiene una URI única. Las operaciones que se pueden realizar sobre los recursos se definen mediante los verbos HTTP, de forma que cada recurso tiene un conjunto de operaciones asociadas. Por ejemplo, si tenemos un recurso que representa a un usuario, podemos realizar las siguientes operaciones sobre el recurso:

— **GET**. Esta operación permite obtener los usuarios.
— **POST**. Esta operación permite crear un usuario.
— **PUT**. Esta operación permite modificar un usuario.
— **DELETE**. Esta operación permite eliminar un usuario.

Al tener un esquema tan simple y conciso, este tipo de arquitecturas se pueden definir de manera muy sencilla utilizando lenguajes de descripción de recursos, como por ejemplo OpenAPI [Swagger, 2022]. Estos lenguajes permiten definir los recursos de una API de forma muy sencilla, de manera que se pueda generar la documentación de la API de manera automática e incluso las implementaciones de las mismas.

Digital Dice, como solución que se presenta en este trabajo de tesis doctoral, está basado en este tipo de arquitecturas. Para definir la funcionalidad y los distintos recursos que maneja el Digital Dice se ha utilizado la Thing Description de la WoT. Como se verá más adelante, Digital Dice es una solución capaz de interactuar con los usuarios siguiendo dos patrones principales de interacción: *stateless* y *stateful*. En el patrón *stateless*, el usuario puede interactuar con el Digital Dice para realizar una petición mediante una API RESTful y recibir la respuesta directamente. En el patrón *stateful*, el usuario se subscribe a un recurso y recibe una notificación cuando este recurso cambia.

1.6. Middleware en IoT

Desarrollar un ecosistema de dispositivos ciberfísicos (CPS), dispositivos IoT o una aplicación distribuida con cierta complejidad utilizando redes de sensores inalámbricos se vuelve complicado si no establecemos un middleware capaz de combinar este tipo de dispositivos [Al-Jaroodi and Mohamed, 2012]. La naturaleza compleja de los CPSs plantea una serie de retos relacionados con la heterogeneidad, la seguridad, la comunicación, e incluso la implementación de estos tipos de dispositivos.

Añadir middlewares en este tipo de ecosistemas nos permite promover la posible inclusión de servicios de valor añadido en nuestros despliegues. Estos tipos de servicios de valor añadido podrían ser, por ejemplo, servicios relacionados con el análisis del comportamiento de los dispositivos, el establecimiento de patrones de comportamiento entre dispositivos, o la inclusión de servicios de notificación de errores en nuestra aplicación, por nombrar algunos. Estos middleware pueden estar agrupados en función de sus enfoques de diseño (basados en eventos, orientados a servicios, basados en VM, basados en agentes, orientados a bases de datos, etc.) [Razzaque et al., 2016]. Aún así, en algunos casos, las soluciones pueden estar en más de una categoría.

1.6.1. Middlewares basados en eventos

Los middlewares basados en eventos engloba una categoría de software que facilita la comunicación y coordinación entre componentes y aplicaciones en un sistema distribuido mediante el uso de eventos. Estos middlewares son especialmente útiles en escenarios donde la naturaleza de las interacciones entre componentes es dinámica, compleja y requiere una alta capacidad de adaptación.

Los middleware basados en eventos, como Hermes [Zhang et al., 2020], tienden a usar un patrón *publición/suscripción* donde los eventos son propagados desde los productores a los consumidores. En algunos casos, estos eventos pueden incluir metadatos relacionados con el evento en particular producido. Este tipo particular de middleware basado en eventos es Message-Oriented-Middleware (MOM). Los Message-Oriented Middleware, son una clase de software de infraestructura que facilita la comunicación entre diferentes aplicaciones y componentes en un sistema distribuido. Los MOMs son una parte fundamental en la arquitectura de muchos sistemas empresariales modernos, ya que permiten la interacción entre aplicaciones y servicios independientes de una manera escalable, confiable y eficiente.

En esencia, los MOMs funcionan como intermediarios en la comunicación entre distintas aplicaciones, permitiendo el intercambio de mensajes de manera asíncrona y desacoplada. Esto significa que las aplicaciones no necesitan conocer detalles específicos sobre la ubicación, el lenguaje de programación o el sistema operativo de las otras partes involucradas en la comunicación. Los MOMs utilizan patrones de mensajería como publicación-suscripción, cola de mensajes y solicitud-respuesta para gestionar y transmitir los mensajes entre diferentes componentes. Gracias a estas características, los sistemas basados en MOMs son altamente escalables y pueden manejar fácilmente picos en la carga de trabajo y aumentar la resiliencia en caso de fallos o errores en el sistema.

Un ejemplo de un MOM es la solución establecida en [Kuzminykh et al., 2019]. Esta solución fue diseñada para facilitar el desarrollo de aplicaciones sobre redes de sensores inalámbricos, permitiendo a los sensores agrupar datos para reducir el número de transmisiones de mensajes y el consumo de energía de dichos sensores. El problema con esta solución es que no soporta topología de red dinámica, lo que la hace menos adecuada para entornos que cambian rápidamente.

Los MOM se basan en los *brokers* de cola de mensajes (Message Queue, MQ) para ofrecer soporte a la comunicación *publicación/suscripción* intentando homogeneizar los diferentes flujos de mensajes entre dichas entidades. Mosquitto [Light, 2017], WebSphere MQ [Aranha et al., 2013] o RabbitMQ [Roy, 2017] son *brokers* que permiten la comunicación *publicación/suscripción* a través de tópicos (*topics*). Estos tópicos son una manera que tienen los MOM para filtrar los mensajes y que pueden estar divididos en varios niveles. Por ejemplo, puede existir un tópico `ual/citic/laboratorio/temperatura`, al que un consumidor se puede suscribir. También lo puede hacer a cualquier nivel; Así, si quisiera recibir todos los avisos de un laboratorio, no solo de los de temperatura, se podría subscribir al tópico `ual/citic/laboratorio`.

1.6.2. Middleware orientado a servicios

Los middleware orientados a servicios (Service-Oriented Middleware, SOM) son un conjunto de tecnologías y herramientas que facilitan la creación, integración y gestión de aplicaciones y servicios basados en arquitecturas orientadas a servicios SOA [Perrey and Lycett, 2003]. SOM proporciona un marco de trabajo y abstracciones comunes que facilitan el desarrollo, despliegue y mantenimiento de aplicaciones SOA, al tiempo que promueve la interoperabilidad y la reutilización de componentes de software.

SOM se basa en una serie de componentes clave para llevar a cabo sus funciones. Entre ellos se encuentran los sistemas de mensajería asíncrona, los brokers de servicios, los repositorios de metadatos y los motores de orquestación y coreografía. Los sistemas de mensajería asíncrona permiten la comunicación entre servicios a través de mensajes, facilitando la desacoplación y mejorando la escalabilidad y la resiliencia del sistema. Los brokers de servicios actúan como intermediarios, ayudando a descubrir, enlazar y coordinar las interacciones entre los servicios. Los repositorios de metadatos almacenan información sobre los servicios disponibles, sus interfaces, políticas de seguridad y otros detalles relevantes. Por último, los motores de orquestación y coreografía gestionan el flujo de trabajo y la coordinación entre múltiples servicios, asegurando que se ejecuten en el orden y de la manera adecuada.

Uno de los mayores problemas que aparece a la hora de establecer comunicación con sistemas ciberfísicos o dispositivos IoT es el gran número de protocolos de comunicación diferentes que pueden ser utilizados. La Thing Description de la WoT soporta protocolos basados en comunicaciones de tipo IP, más específicamente COAP, HTTP y MQTT, gracias a la extensibilidad de las Binding Templates de la WoT [W3C, 2023f], que permite establecer cabeceras con datos necesarios específicos a un protocolo en particular.

En esta tesis se ha usado la categoría SOM. Una de sus ventajas es la posibilidad de implementar nuevos protocolos distintos a la de la WoT. En la propuesta, uno de estos protocolos usados ha sido KNX [Ruta et al., 2017]. KNX es un protocolo de comunicación relacionado con sistemas Smart Homes y Smart Buildings. KNX tiene la particularidad de que las interacciones se basan en operaciones de escritura y lectura sobre direcciones de grupo, que se establecen en un BUS, donde se conectan los diferentes dispositivos. Este protocolo no permite su declaración a través de una Thing Description, ya que no se basa en comunicaciones de tipo IP, pero gracias a la inclusión de lo que se conoce como un gateway IP y una solución de software que permite la comunicación directa con este gateway, WoTnectivity [Mena et al., 2020], se puede llegar a establecer la comunicación con dispositivos de este tipo. Al igual que la capacidad de trabajar con KNX, es posible la inclusión de otros protocolos bien conocidos en la gestión de dispositivos IoT o sistemas ciberfísicos como Zigbee, Modbus, Z-wave, etc.

La solución que se presenta en esta tesis tiene como objetivo ser una solución holística cuando se trata de ofrecer la capacidad de trabajar con protocolos de comunicación comunes con dispositivos IoT. La adopción de los principios establecidos en la WoT, hacen de nuestro middleware una solución general para la gestión de sistemas ciberfísicos y dispositivos IoT. Además, el uso de la WoT como capa de definición, hace que nuestra solución sea compatible con otras plataformas que utilizan estos estándares, como WoT Store [Sciullo et al., 2019], o WoTpy [Mangas and Alonso, 2019].

1.7. Sistemas de alta disponibilidad

Los sistemas de alta disponibilidad (High-Availability Systems, HAS) [Piedad and Hawkins, 2001] son esenciales en ámbitos donde la demanda de servicios ininterrumpidos es requerida, siendo fundamental en numerosas industrias y aplicaciones, tales como centros de datos, servicios de telecomunicaciones, aplicaciones financieras, sistemas de control industrial y plataformas de comercio electrónico, entre otros. La necesidad de mantener un servicio constante y fiable en estas áreas es crucial para garantizar la satisfacción del cliente y el buen funcionamiento de las operaciones comerciales.

Un aspecto importante en el diseño de los sistemas de alta disponibilidad es la capacidad de identificar y gestionar los fallos de manera efectiva y eficiente. Para lograr esto, estos sistemas suelen contar con mecanismos de monitoreo y diagnóstico que permiten detectar anomalías o fallos en los componentes y tomar medidas correctivas antes de que el problema se convierta en una interrupción del servicio.

Además, los sistemas de alta disponibilidad también pueden incluir redundancia en sus componentes, de manera que si un componente falla, otro puede asumir su función. Esta redundancia puede ser activa o pasiva, dependiendo de si el componente redundante

se encuentra en funcionamiento de forma constante o solo se activa en caso de un fallo en el componente principal.

Otra característica de los sistemas de alta disponibilidad es la capacidad de realizar actualizaciones y mantenimiento sin interrupciones en el servicio. Esto se logra mediante el uso de técnicas como la actualización en caliente (hot swapping) y la migración en vivo (live migration) de aplicaciones o servicios, lo que permite realizar cambios en el sistema sin que los usuarios se vean afectados.

La eficiencia energética y la escalabilidad también son aspectos fundamentales en los sistemas de alta disponibilidad. La optimización del consumo de energía y la capacidad de adaptarse rápidamente a cambios en la demanda, mediante la incorporación o eliminación de recursos, son factores clave para garantizar la sostenibilidad y rentabilidad de estos sistemas a largo plazo..

1.7.1. Tipos y principios básicos

Los sistemas de alta disponibilidad se pueden clasificar en dos grandes grupos: sistemas de alta disponibilidad activos y sistemas de alta disponibilidad pasivos. Los sistemas de alta disponibilidad **activos** son aquellos que tienen la capacidad de detectar un fallo y de reaccionar al mismo para ofrecer un servicio continuo. Los sistemas de alta disponibilidad **pasivos** son aquellos que no tienen la capacidad de detectar un fallo, pero que ofrecen un servicio continuo a pesar de que se produzcan estos fallos.

Las arquitecturas basadas en microservicios son especialmente interesantes cuando se trata de ofrecer sistemas de alta disponibilidad. Estas arquitecturas están caracterizadas por el desacoplamiento de los diferentes microservicios que componen el sistema, lo que permite que cada uno de ellos pueda ser reemplazado por otro sin que el sistema deje de funcionar. Además, cada uno de de los microservicios puede ser escalado, aumentar o disminuir el número de réplicas, de forma independiente, lo que permite que el sistema pueda ser escalado de forma horizontal.

Estas características hacen que las arquitecturas basadas en microservicios sean especialmente interesantes cuando se trata de ofrecer sistemas de alta disponibilidad. Los tres principios básicos para dotar a un sistema de alta disponibilidad son:

(a) Eliminar los *puntos catastróficos de fallo*. Estos puntos son aquellos que cuando ocurren causan un fallo total en nuestro sistema y hacen que este deje de funcionar. Una manera de limitar estos puntos de fallos es el uso de sistemas redundantes para que un fallo catastrófico no conlleve la pérdida total del sistema.

(b) El hecho de incluir redundancia en los microservicios puede incurrir en la inclusión de nuevos puntos de fallo, estos puntos de fallo se denominan *puntos de cruce (crossover point)*. En estos puntos se ha de determinar que réplica se hará cargo de una petición concreta. Al mismo tiempo, se debe saber cómo responder a que alguna de las réplicas falle.

(c) Para responder a posibles fallos en los puntos de cruce, se debe incluir un mecanismo de detección de fallos en las réplicas, y un mecanismo de recuperación en caso de que se produzca un fallo en los microservicios.

Las arquitecturas de microservicios se adhieren intrínsecamente a estos principios. En la mayoría de los casos, los microservicios se despliegan de forma redundante, lo cual

permite eliminar esos *puntos catastróficos de fallo*. La monitorización de los microservicios corre a cargo de la malla de servicios (*service mesh*) bajo la cual se puede realizar un seguimiento exhaustivo de los microservicios y del tráfico entre y sobre los mismos. Por otro lado, estas mallas de servicios permiten la inclusión de patrones operativos como el *circuit breaker* [Pacheco, 2018], permiten definir en tiempo real el comportamiento del sistema cuando se producen fallos en alguno de los microservicios, aportando robustez en los *puntos de cruce*.

Digital Dice busca ofrecer un sistema de alta disponibilidad para el manejo de los dispositivos IoT o Ciberfísicos. Para ello, se aplican estrategias relacionadas con la comunicación de los dispositivos físicos, la comunicación de los distintos microservicios y la forma de replicar los microservicios. Estas estrategias serán detalladas en el *Capítulo 3*, donde se explica en detalle cómo adaptar un Digital Dice y convertirlo en en un sistema de alta disponibilidad, entre otras cosas.

1.8. Procesamiento de eventos

Para comprender el funcionamiento del sistema propuesto en este trabajo, el cual permite la interacción automática y transparente entre distintos Digital Dice, es necesario comprender los principios fundamentales de los sistemas de procesamiento de eventos complejos (Complex Event Processing, CEP). Esto incluye una conceptualización precisa de los eventos, para lo cual se ha tomado como referencia la definición presentada por Etzion y Niblett [Etzion and Niblett, 2011].

Definición 1.4 (Evento) *Un **evento** es una ocurrencia dentro de un sistema o dominio particular; es algo que ha ocurrido o se contempla como ocurrido en ese dominio. La palabra **evento** también se usa para significar una entidad de programación que representa tal ocurrencia en un sistema de cómputo.*

En el contexto de los sistemas CEP, se utiliza la segunda definición de evento, es decir, una representación computacional de una ocurrencia. Esta definición es esencial debido a la característica principal de los CEP, que es la capacidad de reaccionar ante la ocurrencia de eventos específicos en el sistema. Para ello, es necesaria una representación computacional de los eventos para poder tratarlos.

En CEP, se pueden distinguir dos tipos de eventos: los eventos simples y los eventos complejos. Los eventos simples son generados por productores de eventos, como sensores (por ejemplo, medición de temperatura), sistemas (por ejemplo, sistemas de detección de sobrecarga de tráfico de red), o procesos de negocio (por ejemplo, reserva de habitaciones en un hotel). Estos eventos simples son enviados a un motor CEP para su procesamiento. Los eventos complejos, por otro lado, son generados a través de reglas específicas en el motor CEP. Estas reglas expresan patrones de eventos específicos y permiten reaccionar ante ellos [Chen et al., 2014]. Por ejemplo, si un sensor de temperatura detecta que la diferencia de temperatura en una habitación es mayor de 3 °C con respecto a lo que marca un sistema de aire acondicionado en el intervalo de una hora, se dispara un evento aviso de funcionamiento erróneo del aire, este último evento se considera un evento complejo. Los eventos complejos pueden ser enviados directamente a los consumidores de eventos,

los cuales realizan acciones en consecuencia. Estos consumidores pueden ser sistemas de persistencia, actuadores o cualquier otro sistema que lance procesos de negocio. Además, estos eventos complejos pueden ser utilizados nuevamente por otras reglas del motor CEP si requieren la presencia de ese evento para ser ejecutadas en sucesivas acciones.

1.8.1. Ejecución de las reglas en un CEP

Las reglas mencionadas anteriormente para generar eventos complejos son especificadas mediante lenguajes de procesamiento de eventos (Event Processing Language, EPL), los cuales tienen una estructura común en la mayoría de los sistemas de CEP [Moreno et al., 2018]. Esta estructura se compone de tres fases:

(a) Fase de selección: En esta fase se determinan los eventos, tanto simples como complejos, necesarios para que se ejecute una regla en particular. Por ejemplo, en el caso de un sensor de temperatura, se seleccionarían sólo los eventos relacionados con la medición de temperatura. Se establecen conjuntos de dependencias para cada regla, los cuales incluyen todos los eventos necesarios para su ejecución.

(b) Fase de coincidencia: Se evalúan los eventos seleccionados en la fase anterior para determinar si cumplen con los requisitos establecidos para la ejecución de la regla. Por ejemplo, en el caso del sensor de temperatura, la regla sólo se ejecutaría si la temperatura es superior a 3 °C con respecto a la del aire acondicionado. En escenarios más complejos, se pueden utilizar operadores lógicos para combinar varios eventos y evaluar si cumplen con los requisitos para la ejecución de la regla.

(c) Fase de derivación: Se especifican los datos o acciones, extraídos de los atributos de los eventos seleccionados en la fase de selección, que se ejecutarán en forma de un nuevo evento complejo después de que se ejecute la regla. En el ejemplo anterior, se enviará el evento aviso de funcionamiento erróneo del aire, se podrían incluir atributos como el identificador del sensor que produjo la alerta para que el consumidor sepa dónde ocurrió la misma.

En resumen, los sistemas de procesamiento de eventos complejos son programas que se basan en reglas para deducir conclusiones al detectar patrones específicos en eventos. Estos sistemas se caracterizan por tener propiedades como la aciclicidad y el orden entre las reglas debido a su naturaleza no determinista y confluente [Burgueño et al., 2018]. Los motores de procesamiento de eventos y los lenguajes como ESPER [Espertech, 2022] son herramientas que cumplen con estas propiedades. En comparación con un enfoque clásico, los sistemas CEP persisten las reglas en lugar de los datos, y los datos fluyen a través de estas reglas para generar resultados. Este enfoque es más adecuado para aplicaciones en tiempo real, ya que los eventos desencadenan las acciones en lugar de depender de una consulta en particular.

1.8.2. Server-Sent Events y su contexto en los CEP

Los Server-Sent Events (SSE) [de la Torre et al., 2019] son un estándar basado en la tecnología web que permite a los servidores enviar actualizaciones en tiempo real a los clientes, como navegadores web o aplicaciones móviles, a través de una conexión HTTP unidireccional. Esta tecnología es especialmente útil en aplicaciones que requieren flujos

de datos en tiempo real y actualizaciones frecuentes desde el servidor, como notificaciones, tableros de control en vivo y aplicaciones de monitoreo.

A diferencia de otras tecnologías de comunicación en tiempo real, como WebSockets [Wang et al., 2013], en las que se establece una conexión bidireccional entre el cliente y el servidor, los eventos SSE se centran exclusivamente en la transmisión de datos, simplificando el proceso de implementación y reduciendo la carga en el servidor, al no ser necesario mantener una conexión bidireccional activa en todo momento.

Los eventos SSE utilizan el formato MIME "`text/event-stream`" para transmitir eventos y datos en un formato fácil de entender y procesar. El cliente simplemente se suscribe a un flujo de eventos desde el servidor, y este último envía actualizaciones a medida que ocurren nuevos eventos. Cada evento puede contener información en formato de texto o datos estructurados en formatos como JSON, lo que facilita su procesamiento e integración en aplicaciones web y móviles.

En este contexto, los sistemas de eventos complejos (CEP) son una tecnología complementaria que puede ayudar a procesar y analizar eventos en tiempo real en aplicaciones que utilizan Server-Sent Events. Los sistemas de eventos complejos permiten identificar patrones y correlaciones entre eventos, lo que facilita la toma de decisiones y la detección de problemas o tendencias emergentes en aplicaciones en tiempo real.

Al combinar SSE con CEP, los desarrolladores pueden construir aplicaciones altamente escalables y eficientes que procesan y responden a eventos en tiempo real. La capacidad de los sistemas CEP para analizar y procesar eventos permite a las aplicaciones que utilizan SSE tomar decisiones más informadas y ofrecer un mejor servicio a sus usuarios. En conjunto, estas tecnologías representan una solución poderosa y flexible para aplicaciones que requieren procesamiento de eventos en tiempo real y comunicación eficiente entre servidores y clientes.

Como se verá mas adelante, Digital Dice cuenta con su propio sistema de control de eventos para proporcionar un sistema de interacción entre dispositivos sin intervención humana, denominado como modelo de Causalidad.

Capítulo 2

Digital Dice

Capítulo 2

Digital Dice

Contenido

2.1. Introducción

En este capítulo se presenta Digital Dice (DD), una infraestructura para la gestión de dispositivos IoT y sistemas ciberfísicos. El término "Dice" (dado en inglés) simboliza las capacidades o facetas que un dispositivo virtual en DD puede tener. El modelo Digital Dice está compuesto por seis facetas diferentes: (a) *Controller*, (b) *Data Handler*, (c) *Reflection*, (d) *Event Handler*, (e) *User Interface (UI)* y (f) *Virtualizer*. Cada una de estas facetas ha sido implementada como microservicios. La ventaja de usar microservicios es que se pueden escalar de forma dinámica e individual.

Digital Dice abstrae a los usuarios y desarrolladores de aplicaciones de este tipo de dispositivos de las tecnologías que estos utilizan para comunicarse con el mundo exterior. Para dotar a Digital Dice de esta capacidad, se ofrece a los usuarios la posibilidad de acceder a los dispositivos mediante una interfaz de programación de aplicaciones (API), más concretamente una API RESTful [Biehl, 2016]; de esta manera los usuarios podrán realizar peticiones para recuperar algún dato o ejecutar alguna funcionalidad de los dispositivos. Por otro lado, para consultar los datos de los dispositivos en tiempo real, los usuarios podrán suscribirse a cualquier propiedad o evento que Digital Dice maneje. Para realizar esta suscripción, se ofrecen al usuario los datos mediante Server-Sent Events (SSE) [de la Torre et al., 2019]. SSE es una tecnología que permite a cualquier usuario recibir notificaciones *push* de manera automática del lado del servidor mediante una conexión HTTP. La utilización del protocolo SSE viene justificada por el hecho de que tanto las API RESTful como los SSE son tecnologías que se encuentran soportadas por la mayoría de los navegadores web, por lo que cualquier usuario podrá acceder a Digital Dice desde cualquier dispositivo que disponga de un navegador web, y a su vez ambas se basan en la realización de peticiones HTTP, lo cual permite al desarrollador que solo tenga que valerse de un cliente HTTP para realizar las peticiones a Digital Dice.

Este capítulo esta subdividido en una serie de secciones que nos permiten conocer el funcionamiento de Digital Dice. En la Sección 2.2 se establece la relación que existe entre la Web of Things y los Digital Dice. En la Sección 2.3 se presentan los distintos microservicios que forman parte de un Digital Dice. En la Sección 2.4 se establece la arquitectura de un Digital Dice, en la cual se describirán las distintas capas que componen un Digital Dice, así como de los servicios auxiliares que permiten el funcionamiento de la solución propuesta. En la Sección 2.5 se presenta un lenguaje específico de dominio (Domain-Specific language, DSL) [Kosar et al., 2016] para la definición de las partes de un Digital Dice. En esta misma sección se establece el modelo de datos que Digital Dice utiliza para registrar las interacciones por parte de los usuarios de un Digital Dice, así como los datos generados por los dispositivos que un dispositivo Digital Dice están representando. Por último, en la Sección 2.6 se presenta un proceso de transformación de modelo a texto (Model-to-Text, M2T) [Rose et al., 2012] para establecer la base de código de los distintos componentes de un Digital Dice a partir de la plantilla Thing Description que lo representa.

2.2. WoT en el contexto de Digital Dice

En las etapas iniciales del desarrollo de la propuesta basada en Digital Dice, la Web of Things (WoT) se encontraba en una fase muy temprana de desarrollo. En aquel momento, la WoT estaba en fase de desarrollo, conformando sus pilares, documentos y las recomendaciones correspondientes. En la actualidad, la WoT se encuentra en una fase de maduración, y se ha convertido en un estándar de facto para la Web of Things. En esta sección, nos centramos en cómo Digital Dice hace uso de la WoT, más concretamente de la Thing Description para exponer la funcionalidad de un Digital Dice a los usuarios, y se ve cómo se usan los campos semánticos de la Thing Description para, en algunos casos, establecer diferentes comportamientos de los Digital Dice.

2.2.1. Digital Dice como servient software de la WoT

Digital Dice es lo que la Thing Description define como un *servient*, es decir, un software que hace uso de los pilares de la WoT, para definir su funcionalidad y metadatos. Digital Dice cumple con los requisitos de un *servient*, ya que es capaz de exponer la funcionalidad de un dispositivo a través de una interfaz de programación de aplicaciones (Application Programming Interface, API), y además es capaz de interactuar con otros *servients*. En el contexto de la WoT, los *servient* pueden ser de tres tipos: *servients* **Consumidores** que consumen la funcionalidad de otros *servients* o dispositivos a través de sus APIs; *servients* **Productores** que exponen la funcionalidad a través de sus propias APIs; y los *servients* **Intermediarios**, que tienen una doble función de consumir un dispositivo o *servient*, y de exponerlo a través de su propia API, actuando como un intermediario entre los usuarios y el dispositivo representado.

Digital Dice es un *servient* de tipo **intermediario**, ya que es capaz de exponer la funcionalidad de un dispositivo a través de su API, y además es capaz de consumir la funcionalidad de otros *servients* o dispositivos a través de sus APIs. Existe un patrón funcional de Digital Dice que permite pasar de **intermediario** a **productor**, y sucede cuando un Digital Dice necesita usar su faceta de dispositivo virtualizado (virtualización), adoptando un papel más parecido al de un Digital Twin. En la Figura 2.1, se puede observar la relación entre los *servients* y los dispositivos que estos representan dependiendo del tipo de *servient* que se trate.

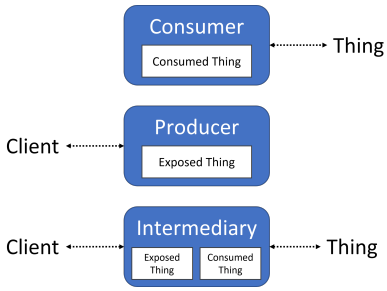


Figura 2.1: Posibles configuraciones de un *servient*.

2.2.2. Digital Dice y la Thing Description

Como se ha visto, Digital Dice hace uso de un documento Thing Description para exponer la funcionalidad de un dispositivo. En el listado de la Figura 2.2 se puede observar un ejemplo de la Thing Description de un Digital Dice de una luz con regulador de intensidad (*dimmer*). Teniendo en cuenta esa Thing Description, se van a analizar todos sus campos para ver qué funcionalidad se está exponiendo a los usuarios y qué particularidades se encuentran en su definición.

Como se puede observar en el listado de la Figura 2.2, la Thing Description de un Digital Dice es perfectamente válida, ya que cumple con todos los requisitos que se establecen en la especificación de la WoT. En la Thing Description (TD) se puede

```
{
"@context": "https://www.w3.org/2019/wot/td/v1",
"id": "acg:lab:dimmer1",
"title": "ACG Lab Dimmer 1",
"@type": ["KNX"],
"base": "https://acg.ual.es/things",
"properties": {
  "luminosity-dimmer1": {
    "type": "object",
    "properties": {
      "brightness": {"type": "number", "minimum": 0,"maximum": 100}
    },
    "required": ["brightness"],
    "forms": [
      {
        "op": ["readproperty", "writeproperty"],
        "href": "#/acg:lab:suitcase-dd/property/luminosity-dimmer1/",
        "contentType": "application/json"
      },
      {
        "op": ["readproperty"],
        "href": "#/acg:lab:suitcase-dd/property/luminosity-dimmer1/sse",
        "subprotocol": "sse",
        "response": {"contentType": "text/event-stream"}
      }
    ]
  }
},
"actions": {
  "switch-dimmer1": {
    "input": {
      "type": "object",
      "properties": {
        "brightness": {
          "type": "number", "minimum": 0, "maximum": 100}
        }},
    "required": ["brightness"],
    "forms": [
      {
        "href": "#/acg:lab:suitcase-dd/action/switch-dimmer1/",
        "contentType": "application/json"
      }]},
  }
}
```

Figura 2.2: Un documento Thing Description de un Digital Dice.

observar que el dispositivo que representa es un regulador de intensidad, comúnmente conocido como *dimmer*. Este *dimmer* cuenta con una propiedad, la luminosidad del dispositivo (`luminosity-dimmer1`) en la línea 8 y una acción *switch* (`switch-dimmer1`) en la línea 30, la cual nos permite cambiar esa luminosidad. Si se analiza la propiedad con más detenimiento, se ve que establece el parámetro `brightness` para mostrar la luminosidad, y este parámetro es un número entre 0 y 100 (línea 11), lo que nos indica que el *dimmer* puede tener 100 niveles de luminosidad. Además, la propiedad cuenta con dos formas de acceso indicados en el campo `op`, una forma de lectura/escritura (`readproperty`/`writeproperty`) y otra de escritura (`writeproperty`), y ambas formas de acceso cuentan con un contenido de respuesta diferente establecido en el `contentType`. La primera forma de acceso será simplemente estableciendo una llamada HTTP a la URL indicada en el campo `href`, que como es una URL parcial habrá que calcularla a partir del campo base de la TD. Al tener las dos variantes `readproperty` y `writeproperty`, se pueden hacer las llamadas con el verbo GET para recuperar el dato de la luminosidad, o con un POST con el campo requerido en `brightness` en el cuerpo de la petición si se quiere modificar la luminosidad con el número establecido por ese campo.

La segunda forma de acceso es a través de Server-Sent Events (SSE), donde a partir de la URL indicada en el campo `href` se establecerá una conexión SSE con el dispositivo, y a partir de ese momento el dispositivo enviará los datos de la luminosidad cada vez que se produzca un cambio en el valor de la luminosidad. La acción que se muestra en esta TD es la acción *switch*, que nos permite cambiar la luminosidad del dispositivo. Para ello, se establece un campo `input` que indica que la acción requiere un parámetro de entrada. En este caso, el parámetro de entrada es el campo `brightness`, que es un número entre 0 y 100. La forma de acceso a esta acción es a través de una llamada de tipo HTTP POST a la URL indicada en el campo `href`. En el cuerpo de la petición se establecerá el campo `brightness` con el valor de la luminosidad que se quiere establecer.

2.2.3. Información semántica para la configuración de DD

En el ejemplo del listado de la Figura 2.2, se observa un campo denominado `@type`, que permite introducir información semántica para establecer algunos parámetros de configuración. Este campo puede aparecer tanto a nivel de Thing como a nivel de interacción (`action`, `property` o `event`). En el ejemplo, se está indicado a nivel de Thing, y que se usa un dispositivo KNX (línea 5). Esto último es interesante debido a que, tal y como se verá en la Sección 2.6, se usará este valor de la variable del campo `@type` para generar una serie de aplicaciones que corresponden a los microservicios del Digital Dice, necesarios para manejar este dispositivo; por ejemplo, en este caso, este valor se usará para saber que hay que importar la librería *wotnectivity-knx* en el microservicio `Reflection` del Digital Dice del dispositivo en cuestión.

En la Tabla 2.1, se muestran los distintos valores `@type` que utiliza DD, así como sus significados a nivel de Thing y/o a nivel de interacción. El vocabulario controlado por DD en la variable semántica `@type` permite ir añadiendo progresivamente información semántica, por lo que en el futuro puede que aparezcan nuevos valores para esta variable de utilidad.

@type	**Thing**	**Interacción**
KNX	La Thing se maneja mediante el protocolo de comunicación KNX.	La interacción utiliza el protocolo KNX.
WS	La Thing se maneja mediante el protocolo de comunicación WS, relativo a Web Sockets.	Indica que la interacción utiliza el protocolo WS.
MQTT	La Thing se maneja mediante el protocolo de comunicación MQTT.	La interacción particular utiliza el protocolo MQTT.
virtual	El DD es completamente virtual, es decir, todas sus interacciones son virtuales.	La interacción es virtual.
ui	Tiene una interfaz de usuario que permite interactuar con todas las interacciones del dispositivo implementado. La interfaz estará presente en el campo `link`.	La interacción tiene una interfaz de usuario para trabajar con ella. La interfaz estará presente en el campo `link`.

Tabla 2.1: Valores de `@type` para Digital Dice.

2.3. Microservicios de un componente digital

Digital Dice se basa en una arquitectura de microservicios, donde cada uno de ellos se encarga de un cometido concreto dentro del Digital Dice. De esa manera se consigue una mayor escalabilidad y mantenibilidad del sistema, así como una mayor modularidad. En las siguientes subsecciones se describe la funcionalidad de cada microservicio.

2.3.1. Controller

Este componente o faceta se ocupa de las funciones necesarias para establecer la comunicación con el usuario. Los métodos generados para la comunicación con este sistema seguiran los principios REST. Estos principios incluyen:

(a) La utilización de un protocolo sin estado, como HTTP, en el que cada petición contiene toda la información necesaria para ser procesada.

(b) La adopción de un conjunto estandarizado de métodos bien definidos, como GET, POST, PUT y DELETE.

(c) La identificación de recursos mediante URI (Uniform Resource Identifiers), proporcionando una estructura jerárquica y organizada.

(d) El intercambio de representaciones de recursos en formatos estandarizados como JSON, permitiendo la interoperabilidad entre sistemas.

(f) La incorporación de enlaces y relaciones entre recursos, favoreciendo la navegación y descubrimiento de la API de manera intuitiva.

Algunos de los métodos o rutas de esta faceta requerirán autenticación a nivel de aplicación, a nivel de usuario o a ambos, es decir que el usuario sólo pueda acceder a esas rutas si tiene permisos y por una aplicación en concreto. Esta comunicación se realizará

a través de una API RESTful [Biehl, 2016] en primera instancia, aunque también se le proporciona al usuario una serie de métodos mediante Server-Sent Events (SSE) [Cook, 2014] para que pueda recibir notificaciones en tiempo real. El `Controller` redirige las peticiones del usuario al resto de facetas que forman parte del Digital Dice, haciendo de puerta de entrada al sistema.

2.3.2. Data Handler

El componente `Data Handler` se encarga de la comunicación directa con la base de datos. Su principal funcionalidad es recuperar los datos que los usuarios quieran consultar, así como guardar en la base de datos las peticiones de los usuarios que supongan el cambio de estado de alguna interacción que establezca el dispositivo. Por otro lado, también realizará un registro de cualquier petición que se realice al sistema. Es decir, en el caso de dispositivos actuadores, este servicio registrará tanto el cambio de estado del actuador propiamente dicho, como quién ha realizado la interacción sobre el mismo. En el caso de sensores, registrará todo cambio de estado en el sensor, así como quién lo ha consultado y en qué momento para su posterior análisis. A parte de lo expuesto, esta faceta permitirá consultar tanto el estado actual del dispositivo como un histórico de los cambios de estado del mismo.

Este componente establece la comunicación con la base de datos utilizando patrones reactivos. Los patrones reactivos hacen referencia a un enfoque de diseño y programación que busca mejorar la eficiencia, escalabilidad y robustez de los sistemas de gestión de datos. Este enfoque se centra en la propagación de cambios en los datos de manera asincrónica y no bloqueante, permitiendo a las aplicaciones responder de forma más ágil a eventos y cambios en el estado de los datos. Para ello, se utilizan conceptos como la programación reactiva, en la que los datos se modelan como flujos de eventos y se emplean operadores de manipulación para transformar y combinar dichos flujos, y el uso de bases de datos reactivas que soportan consultas y actualizaciones en tiempo real.

Además, los patrones reactivos también promueven la adaptabilidad y resiliencia ante fallos y fluctuaciones en la carga de trabajo. Para lograr esto, se implementan técnicas de escalado horizontal y replicación de datos, permitiendo a las bases de datos distribuir la carga entre múltiples nodos y garantizar la disponibilidad y consistencia en caso de fallos. También se enfatiza el uso de sistemas de colas y mecanismos de retroceso para manejar la presión en el sistema y evitar la sobrecarga de recursos.

Gracias a el uso de estos patrones reactivos se da soporte a SSE, ya que a el usuario se le va a ofrecer una prolongación de los eventos que manda la base de datos al `Data Handler` a través de una tecnología conocida y estándar.

2.3.3. Reflection

`Reflection` es el responsable de establecer la comunicación mediante un único canal con el dispositivo físico, y será el que se encargue de por un lado "reflejar" cualquier cambio de estado que se produzca en el dispositivo físico en la base de datos y por otro lado, "reflejar" cualquier cambio de estado que se produzca en la base de datos en el dispositivo físico. Este microservicio se encarga de realizar la traducción de los mensajes

que se envían y reciben del dispositivo físico a un formato que pueda ser interpretado por el resto de microservicios. Por ejemplo, si el dispositivo físico utiliza un protocolo de comunicación como MQTT [Soni and Makwana, 2017], este microservicio se encarga de traducir los mensajes que se envían y reciben a un formato JSON, que es el que utiliza el resto de microservicios. El hecho de que este componente de microservicio sea el único que se conecte con el dispositivo físico, permite que el resto de microservicios no tengan que conocer el protocolo de comunicación que utiliza el dispositivo físico, lo que facilita la escalabilidad del sistema. De la misma manera, el ser el único punto de comunicación con el dispositivo tiene ciertas connotaciones a nivel de seguridad y rendimiento, como por ejemplo que si el dispositivo físico se encuentra en una red privada, el resto de microservicios no tendrán que conocer la IP del dispositivo físico, o por ejemplo que las peticiones de escritura de los usuarios serán necesariamente y en última estancia atendidas por este microservicios, pudiendo de esa manera establecer distintos comportamientos en función de la situación.

2.3.4. Event Handler

El componente `Event Handler` se encarga de gestionar los eventos producidos en el dispositivo físico, previamente registrados por `Reflection`. También es el responsable de escuchar eventos (externos e internos) que puedan afectar al dispositivo gestionado por el microservicio. Este microservicio es la base de las relaciones de causalidad (ver Capítulo 3), ya que se encarga de manejar aquellas relaciones que estén declaradas en el servicio de causalidad que afecten al dispositivo representado, ejerciendo la función de gestor de eventos complejos dentro de cada uno de nuestros Digital Dice.

Una de las fucionalidades que añade este servicio, es la capacidad que tiene `Event Handler` de retroalimentarse de los eventos complejos generados por las relaciones de causalidad que en principio no tienen por qué estar declarados como eventos en el documento de la Thing Description.

2.3.5. User Interface

El componente *User Interface* (UI) está compuesto por representaciones visuales en forma de componentes web de interfaz de usuario (Web Component) del dispositivo IoT en concreto, o de interacciones individuales del dispositivo representado. Este componente de microservicio permite establecer una interfaz de usuario para controlar el dispositivo representado. Las interfaces generadas hacen uso de la API RESTful proporcionada por el microservicio `Controller` para establecer la comunicación con el resto de microservicios. Este componente puede ofrecer interfaces tanto a nivel de toda la Thing como a nivel de interacciones particulares, declaradas en la plantilla Thing Description. Para establecer las interfaces de usuario que tendrá que generar este componente se hace uso de la variable semantica `@type` de la Thing Description, donde se establece el valor `ui`, para indicar con ello que la Thing o la interacción individual cuenta con una interfaz gráfica de usuario para explorar su estado.

2.3.6. Virtualizer

`Virtualizer` se encarga de la virtualización del dispositivo físico. El microservicio de virtualización desempeña el papel de un sustituto para el dispositivo físico, ofreciendo a los microservicios asociados una interfaz de programación de aplicaciones (API) que replica fielmente la empleada al interactuar con el dispositivo real. Esta interfaz posibilita que los microservicios realicen operaciones de lectura y escritura en la representación virtual del dispositivo, facilitando la ejecución de pruebas de extremo a extremo en sistemas IoT. Además, al utilizar una representación virtual del dispositivo físico, los desarrolladores de sistemas IoT pueden diseñar escenarios de prueba intrincados y diversos sin preocuparse por la disponibilidad o el estado actual del dispositivo físico en sí. De esta manera, se logra una mayor flexibilidad en el proceso de desarrollo y prueba, permitiendo la optimización y mejora de la calidad del sistema IoT en su conjunto.

2.3.7. Configuración de microservicios

Digital Dice puede definirse con varias configuraciones de microservicios, dependiendo de las necesidades de cada dispositivo. No obstante, existen microservicios que son comunes a todos los Digital Dice: `Controller` (`C`), `Data Handler` (`DH`) y `Reflection` (`R`). El resto de microservicios son opcionales y dependerán si el dispositivo cuenta con eventos, y se usará su componente `Event Handler` (`EH`), si se necesita una interfaz gráfica de usuario para explorar sus capacidades, usando su componente `User Interface` (`UI`), o bien si se desea trabajar con un dispositivo virtual, para lo cual se usará su componente `Virtualizer` (`V`). Por tanto, Digital Dice puede desplegarse con diferentes configuraciones de servicios, y puede ser escalado de manera sencilla añadiendo o quitando replicas de los microservicios de manera independiente. Esta escalabilidad es una de las principales ventajas de Digital Dice y a la vez una de las estrategias que nos permite dotar de la propiedad de alta disponibilidad (HA) a Digital Dice.

2.4. Arquitectura de Digital Dice

Como se ha comentado anteriormente, Digital Dice se basa en una arquitectura de microservicios, pero también se inspira en ideas establecidas, por ejemplo, de las arquitecturas orientadas a recursos. En esta sección, se explica la arquitectura de Digital Dice en detalle. Anteriormente se han visto los distintos microservicios de los que están compuestos los Digital Dice, pero no se ha explicado cómo se comunican entre ellos. En esta sección se va a explicar cómo se comunican los distintos microservicios entre sí, y cómo se comunican con el dispositivo físico. Además, se va a ver qué otros servicios y aplicaciones se utilizan para el correcto funcionamiento de Digital Dice.

La Figura 2.3 representa la arquitectura de Digital Dice con todos los diferentes artefactos que forman parte de la misma. Se muestra cómo las diferentes partes interactúan entre sí, y como Digital Dice en su conjunto ofrece una API para la comunicación con usuarios u otros dispositivos. Como se puede ver en la Figura 2.3, Digital Dice puede tener diferentes configuraciones a nivel de microservicios, así como diferentes números de replicas dentro de esos microservicios. En este caso se observa que el `Digital Dice #1`

tiene un `Event Handler`, y que el microservicio `Data Handler` está replicado. Mientras que `Digital Dice #2`, tiene un microservicio `UI`, y el `Controller` tiene una réplica. El motivo de la replica de cada una de las configuraciones en este ejemplo podría ser porque `Digital Dice #1` esté recibiendo internamente muchas peticiones que tienen que ver con la base de datos de manera interna y `Digital Dice #2` esta recibiendo muchas peticiones a su API por parte de los usuarios, y ha sido necario replicar el comportamiento de `Data Handler` y el `Controller` respectivamente.

En la figura, también se puede observar lo que se ha representado como dos capas la de dispositivos no compatibles con el protocolo TCP (Constrained Devices Non-TCP) y aquellos que si lo son (Standard Devices TCP). Se observa como la `Reflection` del `Digital Dice #1` esta conectada a la interfaz de un sensor de temperatura (TempSensor) dado que este es compatible con protocolos que trabajan sobre TCP (HTTP, WebSocket, MQTT, etc.). En el caso del `Digital Dice #2`, sin embargo, la `Reflection` se conecta al dispositivo KNX-Weather Station a través de una puerta de enlace KNX que sí es compatible con TCP (KNX-IP Gateway).

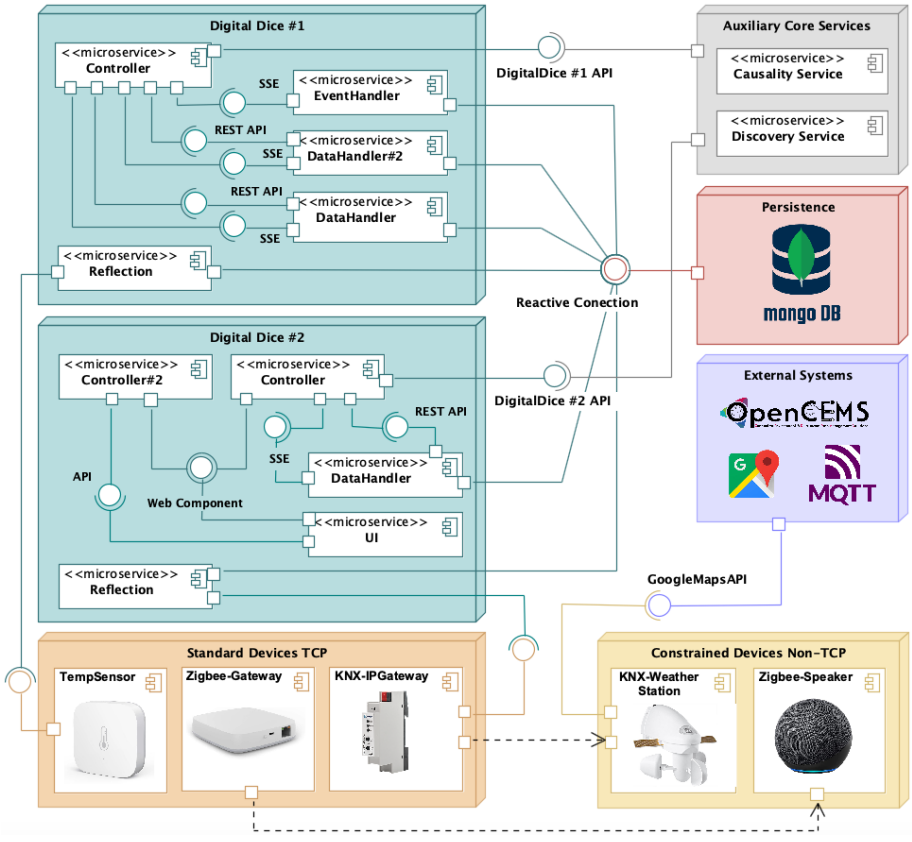


Figura 2.3: Arquitectura de Digital Dice.

La arquitectura también cuenta con una librería de servicios "External Services" que el Digital Dice puede requirir para su correcto funcionamiento. Algunos ejemplos podrían ser, plataformas de datos abiertos (p.ej., OpenCEMS [Mansour et al., 2021]) para el análisis de las interacciones registradas por nuestros Digital Dice, APIs externas (p.ej., Google Directions API) para recabar información de direcciones geográficas, o servicios auxiliares (p.ej., Mosquitto MQTT server).

La arquitectura de Digital Dice, además de los componentes específicos para cada instancia, también requiere una serie de componentes comunes que garantizan su correcto funcionamiento y eficiencia, como componentes para permitir la persistencia de datos, la interacción entre dispositivos, el descubrimiento, o la orquestación de peticiones. A continuación se describen brevemente cada uno de ellos.

2.4.1. Persistencia de datos en Digital Dice

En primer lugar, es fundamental contar con un sistema de almacenamiento y gestión de datos adecuado que respalde las operaciones de la plataforma. En este caso, se opta por utilizar MongoDB como base de datos, ya que ofrece ventajas significativas en comparación con otras soluciones. MongoDB es una base de datos NoSQL orientada a documentos que se adapta perfectamente a las necesidades de Digital Dice, debido a su capacidad para manejar datos no estructurados de manera eficiente. Esto es esencial en el contexto de los sistemas IoT, donde la diversidad y la naturaleza dinámica de los datos pueden representar un desafío para las bases de datos tradicionales. Además, MongoDB facilita la indexación y consulta de datos, lo que resulta en un rendimiento óptimo para las operaciones de lectura y escritura en la plataforma. Por otro lado, MongoDB brinda la posibilidad de configurar fácilmente réplicas y implementar estrategias de escalado horizontal. Esto permite a Digital Dice gestionar de manera efectiva el crecimiento en la demanda y garantizar la disponibilidad y resistencia del sistema ante posibles fallos o eventos adversos o no esperados.

2.4.2. Interacción entre Dispositivos

El segundo componente necesario para el correcto funcionamiento de Digital Dice es el llamado servicio de causalidad [Mena et al., 2021c]. Este servicio permite establecer un mecanismo para definir interacciones entre diferentes dispositivos. Para ello, este servicio cuenta con una API RESTful que permite registrar y consultar lo que se denomina relaciones de causalidad. Esta relación se forma por las entidades causa y efecto. Las relaciones de causalidad ayudan a definir cómo diferentes dispositivos interactúan sin intervención humana, a través de un esquema de definición común que se explicará en el Capítulo 3. Siempre que el Digital Dice cuente con un componente `Event Handler`, este debe consultar el servicio de causalidad para recuperar las relaciones de causalidad que debe manejar. A partir de ese momento, el peso del control de esas relaciones recae en el Event Handler, que será el que tenga que realizar la función del gestor de eventos como vimos en la Sección 2.3.4.

2.4.3. Servicio de descubrimiento

Un componente importante para Digital Dice es el servicio de descubrimiento. Este servicio permite consultar las diferentes Thing Descriptions que están disponibles en la red. Este servicio aparece en las últimas modificaciones incluidas en las recomendaciones WoT. Conforme a la mencionada recomendación, se ha creado un laboratorio de experimentación que hace uso del servicio de descubrimiento, que se puede encontrar en el sitio WoT-Lab[1], en el que los usuarios pueden interactuar con los diferentes dispositivos físicos y virtuales que conforman el entorno de experimentación.

El servicio de descubrimiento contiene un `core` que administra el descubrimiento y procesa las operaciones. Las operaciones del `core` se basan en UDDI, ODP y la propuesta preliminar de descubrimiento de la WoT [W3C, 2023b]. El servicio de descubrimiento sigue la propuesta de ODP, con tres conjuntos de operaciones: (a) operaciones de búsqueda para localizar dispositivos en el directorio o en la red; (b) operaciones de registro para realizar acciones sobre dispositivos; y (c) operaciones de administración para gestionar la configuración del sistema. Cada operación del `core` puede ser ejecutada únicamente por la API RESTful, que actúa como una capa de seguridad entre los agentes externos y el Núcleo. El servicio de descubrimiento aborda los desafíos asociados con la gestión y el descubrimiento de dispositivos IoT en entornos de gran escala. Al combinar las ventajas de las especificaciones UDDI, ODP y W3C WoT Discovery, se logra un sistema eficiente y flexible que facilita la integración y la interoperabilidad entre dispositivos y agentes externos. Además, la implementación de una API RESTful como intermediario entre los agentes y el `core` del sistema garantiza la seguridad y la robustez de las operaciones, protegiendo la integridad y la confidencialidad de los datos y dispositivos involucrados.

2.4.4. Orquestación de las peticiones

Por otro lado, también es importante disponer de un modelo de orquestación de peticiones en la arquitectura. En la Figura 2.4 se muestran dos tipos de peticiones diferentes a un Digital Dice; Por un lado, la invocación de una acción `X`, y por otro, la petición de una propiedad `Y`. En la primera petición, el usuario solicita al componente `Controller` que invoque la acción `X` (1a), este entonces solicita la acción al componente `Data Handler` (2a), el cual se encarga de registrar la petición en la base de datos (3a). Esta petición será observada por el component `Reflection` (4a) del dispositivo que ejecuta la acción `X` sobre el dispositivo (5a). En la segunda petición, el usuario pide al componente `Controller` el valor de la propiedad `Y` (1b), el cual redirige la petición al componente `Data Handler` (2b) que consulta la base de datos sobre dicha propiedad (3b). Finalmente, la base de datos recupera el último valor de la propiedad, devolviéndola a `Data Handler` (4b) y a su vez éste como respuesta al usuario (5b). La elección de estas dos operaciones no es casual. La primera es una operación de escritura, mientras que la segunda es una operación de lectura.

Es importante tener en cuenta que solo las operaciones de escritura requieren comunicación con el dispositivo. En cambio, las operaciones de lectura usan la base de datos como un tipo de caché para los datos solicitados al dispositivo IoT. Cuando se

[1] ACG WoT-Lab - `https://acg.ual.es/wot-lab`

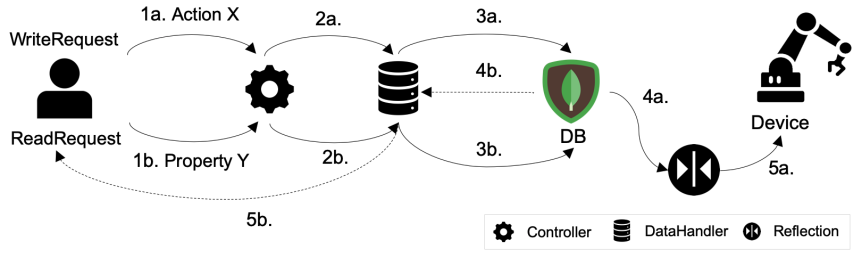


Figura 2.4: Flujo de las peticiones de lectura y escritura en Digital Dice.

habla de la Thing Description, se ve que las posibles interacciones de un dispositivo IoT según este esquema son acciones, propiedades y eventos. La invocación de una acción siempre será una operación de escritura, mientras que escuchar un evento siempre será una operación de lectura, y en el caso de las propiedades, estas pueden ser tanto de lectura como de escritura. Reducir el número de peticiones ejecutadas por el dispositivo IoT es muy importante, especialmente cuando se habla de entornos industriales. En estos entornos, muchos dispositivos o sistemas ciberfísicos utilizan protocolos de tipo bus, como Modbus o Profibus [Gabor et al., 2018], que tienen limitaciones en el número de agentes que pueden realizar operaciones.

2.5. Modelado de Digital Dice

En esta sección se describen los distintos modelos y metamodelos necesarios para el funcionamiento de Digital Dice. Se han desarrollado dos metamodelos, uno para definir un Digital Dice, donde se establecer la configuración de los microservicios del dispositivo virtual, y otro para definir las relaciones de causalidad, que ofrece la capacidad de poder establecer relaciones entre los diferentes dispositivos representados por los Digital Dice. A nivel de modelos, se ha utilizado un modelo de datos para guardar las interacciones que se ejecutan en los dispositivos representados, este modelo se denomina el modelo de interacción. En las siguientes subsecciones de este capítulo, se estudian estos lenguajes y modelos con mas detalle.

2.5.1. Metamodelo de definición DD

El primero de ellos es metamodelo de definición de Digital Dice que permite establecer todos los parámetros de configuración de un Digital Dice. La Figura 2.5 representa el metamodelo para la definición de un Digital Dice, en el que se describen los microservicios que pueden formar parte de él, cada uno de los cuales tiene diferentes `Environments` donde se declaran las variables de entorno y la configuración. Además, para la comunicación interna y externa, cada microservicio tiene una serie de `Resources` definidos por `href`, sus `DataSchemas` de entrada y salida, y el `opType` que indica si esa `Resource` es una operación de lectura o de escritura. Por otro lado, las clases de color verde de-

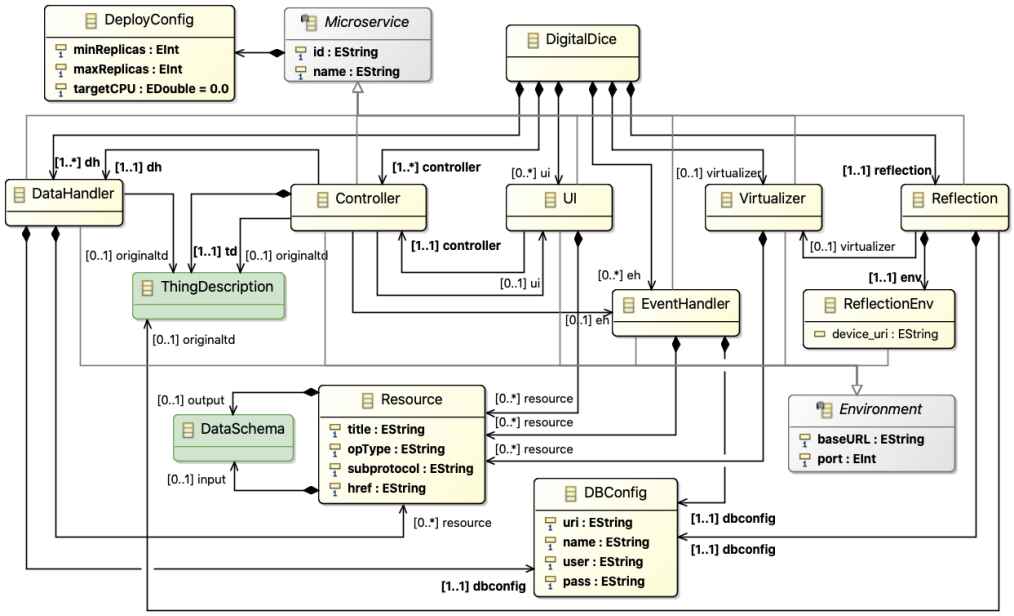


Figura 2.5: Metamodelo de definición de Digital Dice.

notan aquellas ya establecidas en la Thing Description de la WoT, y reutilizadas por nuestro metamodelo de definición. La clase `DataSchemas` de la WoT, es una clase que nos permite guardar prácticamente cualquier tipo de objeto, esto es necesario, dado que se debe adecuar al dispositivo que se está representando, el cual puede llegar a ser muy heterogéneo. Por último, también se cuenta con las `DeployConfig`, que establecen, los parámetros necesarios para el escalado y el uso de CPU objetivo de los microservicios. El metamodelo desarrollado, presentado en la Figura 2.5, representa un lenguaje específico de dominio (Domain-Specific Language, DSL) para definir un Digital Dice. Las instancias del metamodelo se realizan mediante documentos en JSON.

2.5.2. Metamodelo de relaciones de causalidad

El segundo metamodelo desarrollado en este trabajo ha sido para definir las relaciones de causalidad. El metamodelo, que se muestra en la Figura 2.6, es mucho más simple que el anterior, y establece un lenguaje para definir la comunicación entre distintos dispositivos. A modo introductorio, se puede ver cómo estas relaciones de causalidad se basan en declarar efectos (`Effect`) que se van a producir a partir de ciertas causas (`Cause`). Como se observa en la figura, una relación de causalidad puede estar compuesta por varios efectos que serán ejecutados cuando se cumplan las causas establecidas, estas causas apuntan mediante la relación `Link` a la Thing Description que son el origen de estas causas. Este lenguaje, junto con los sistemas que hacen posible su interpretación van a ser estudiados en el Capítulo 3.

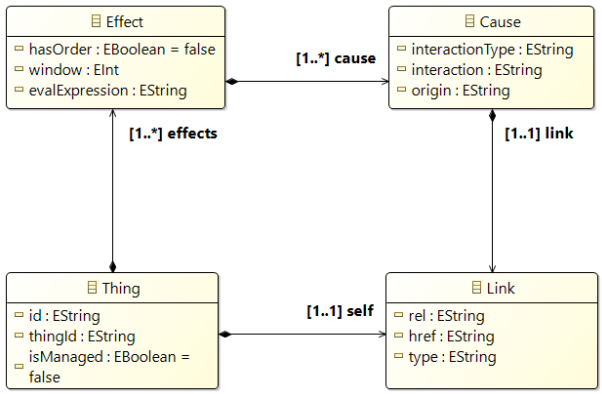


Figura 2.6: Metamodelo de relaciones de causalidad.

2.5.3. Modelo de datos de interacción

Por último, se establece el modelo de datos con el que se guardan las distintas interacciones registradas por el Digital Dice. Este modelo de datos se encuentra definido en la Figura 2.7, y se compone de una serie de propiedades, y los datos generados o enviados en la interacción concreta que esté recibiendo el Digital Dice. En la Tabla 2.2, se describen las propiedades que componen el modelo de datos.

En el listado de la Figura 2.8 se observa una instancia ejemplo del modelo de interacción, para una interacción concreta. Se trata de una interacción de tipo `property`, mas concretamente `containerDetails`, cuyo *source* es un dispositivo virtual. Si se estudia la Thing Description del Digital Dice que está representando este dispositivo, cuyo fragmento de documento se muestra en el listado de la Figura 2.9, se puede observar que la propiedad `containerDetails` ha sido definida en la Thing Description, y que tiene

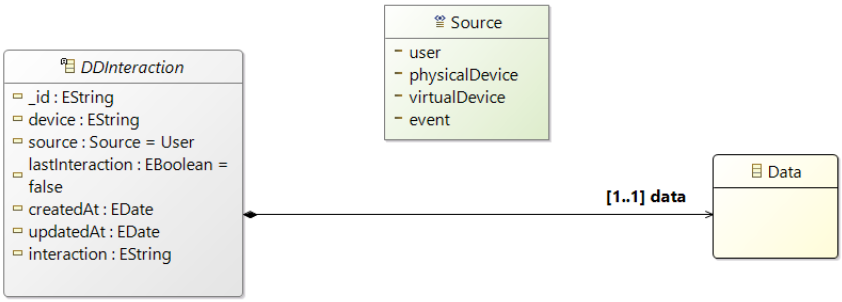


Figura 2.7: Modelo de datos de interacción.

asociada un esquema de datos `data schema` que luego es usada dentro de la instancia en la variable `data` (líneas 6 a la 16). Al ser el `source` del dato un dispositivo virtual quiere decir que ha sido el componente `Reflection` el que ha guardado esa interacción en la base de datos, posiblemente porque ha habido algún cambio en alguno de los valores dentro de `Data`. De esta forma, `Reflection` busca cambios de estado en las propiedades a la espera de algún cambio, y si lo hay da lugar a una ejecución con el dispositivo físico o virtual. En la propiedad `lastInteraction` se almacena el estado de la última interacción, si ha sido la última o no. En el caso del ejemplo, esta propiedad tiene el valor `true`, significando que ha sido entonces la última interacción registrada de esa propiedad. Por defecto, el valor de esta propiedad se define como falsa en el modelo.

Propiedad	**Tipo de dato**	**Descripción**
`_id`	String	`Id` que identifica la interacción.
`device`	String	`Id` que identifica la Thing de la cual procede o se ejecuta la interacción. Esta propiedad debe de coincidir con la `Id` de la Thing Description que se está representando.
`source`	String	Indica el origen de la interacción. Posibles valores: `physicalDevice`, `virtualDevice`, `user`, `event`.
`interaction`	String	Indica la interacción concreta que se esta invocando. {`property/action/event`}.{`interactionName`}
`lastInteraction`	Boolean	Indica si esta es la última llamada que se ha realizado a la interacción.
`createdAt`	Date	Indica la fecha de creación.
`updatedAt`	Date	Indica la fecha de modificación.
`data`	Object	Posibilita el guardar el dato generado en la interacción, o el cuerpo de la petición.

Tabla 2.2: Propiedades del modelo de datos de interacción.

```
1  {
2   "_id" : "62eba8910367a43e31d2e7e5",
3   "device" : "acg:lab:virtual-containers:37523",
4   "source" : "virtualDevice",
5   "interaction" : "property.containerDetails",
6   "data" : {
7     "location" : {
8       "lat" : "36.837410",
9       "lng" : "-2.308610"
10    },
11    "totalCapacity" : 300,
12    "capacity" : 13,
13    "temperature" : 20,
14    "garbageClass" : "organic",
15    "serialNumber" : 37523,
16    "address" : "Paseo Del Toyo 144, Retamar, Almeria, Spain"
17  },
18  "lastInteraction" : true,
19  "createdAt" : ISODate("2022-08-04T11:08:01.677Z"),
20  "updatedAt" : ISODate("2022-08-13T19:36:32.449Z"),
21 }
```

Figura 2.8: Instancia del modelo de interacción Digital Dice.

```
"containerDetails": {
  "type": "object",
  "properties": {
    "location": {
      "type": "object",
      "properties": {
        "lat": {
          "type": "number"
        },
        "lng": {
          "type": "number"
        }
      }
    },
    "totalCapacity": {
      "type": "number",
      "min": 0
    },
    "capacity": {
      "type": "number",
      "min": 0
    },
    "garbageClass": {
      "type": "string"
    },
    "temperature": {
      "type": "number"
    },
    "serialNumber": {
      "type": "number",
      "min": 0
    },
    "address": {
      "type": "string"
    }
}
```

Figura 2.9: Esquema de datos de `containerDetails`.

2.6. Especificación de un componente Digital

Digital Dice usa la Thing Description de la WoT de dos maneras distintas, como se muestra en la Figura 2.10. La primera, como una *Thing* propiamente de la WoT, para que otros sistemas compatibles con este paradigma puedan hacer uso de ella. Este método requiere una intervención manual por parte del desarrollador, ya que tiene que crear el Digital Dice siguiendo los parámetros establecidos por su definición. La segunda forma es consumir las Things y exponerlas a través la propia Thing Description. Esto añade varias ventajas, como representar una forma común de acceder a ellas a través de una API bien formada, o mejorar el rendimiento a través del uso de microservicios y limitando el número de conexiones que afectan a la Thing representada. En este último caso, la creación del Digital Dice puede ser parcialmente automatizado, aplicando técnicas de transformación modelo-a-texto (Model-to-Text, M2T). Como se muestra en la Figura 2.10, la Thing necesita tener una TD asociada con ella, o en su defecto se tendría que generar manualmente. Como parte del ecosistema de la propuesta de Digital Dice, se ha

desarrollado la herramienta **TD2DD Transformer**[2]. Esta herramienta convierte una TD a los diferentes microservicios expuestos por el Digital Dice que pueden representarla, y ofrece la infraestructura de software necesaria para que el sistema pueda funcionar.

Para ese propósito, el transformador convierte una Thing Description en un Digital Dice siguiendo una serie de pasos diseñados para mejorar su fiabilidad. Además, el proceso intenta evitar en la medida de lo posible la intervención del usuario, solo requiriendo su participación cuando elija la TD que quiere convertir y descargue las partes generadas. La Figura 2.11 muestra el flujo de trabajo que sigue el proceso de transformación para generar el Digital Dice y su infraestructura. El flujo de trabajo ejecuta cinco pasos diferentes: validación, verificación, definición, generación y empaquetado.

La Tabla 2.3 muestra la localización de las operaciones de cada paso en el código fuente. En ella se muestra cómo cada paso tiene una serie de métodos que definen su funcionalidad. La validación se gestiona por la clase `validator.service.ts`, mientras que los otros pasos se gestionan en la clase `dd-builder.service.ts`. Las siguientes subsecciones describen cada paso de este flujo de trabajo.

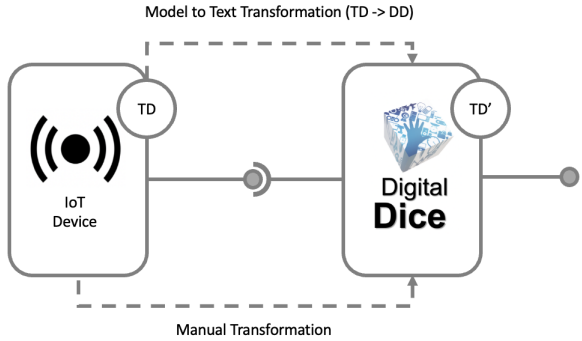


Figura 2.10: Diferentes usos de una Thing Description en Digital Dice.

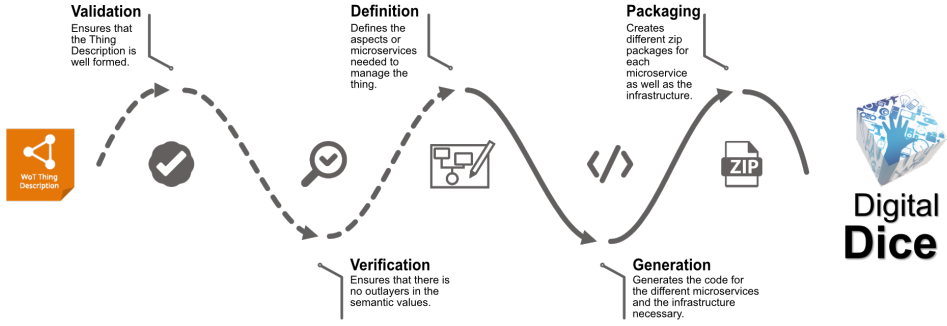


Figura 2.11: Flujo de trabajo de TD2DD Transformer.

[2]TD2DD Transformer - `https://github.com/acgtic211/td2dd-transformer`

Paso	Archivo	Operaciones
Validation	`validator.service.ts`	`validate(data):boolean`
Verification	`dd-builder.service.ts`	`verify(td):string[]` `librariesNeed()`
Definition	`dd-builder.service.ts`	`servicesNeeded(td)`

Generation	`dd-builder.service.ts`	`private buildController()`
		`private buildDataHandler()`
		`private buildEventHandler()`
		`private buildUi()`
		`private buildReflection()`

Packaging	`dd-builder.service.ts`	`zipController()`
		`zipDataHandler()`
		`zipEventHandler()`
		`zipUi()`
		`zipReflection()`
		`zipInfrastructure()`

Tabla 2.3: Correspondencia de métodos para cada paso del transformador.

2.6.1. Validación y Verificación

El primer paso se ejecuta al introducir la Thing Description en el transformador. El software tiene un cuadro de texto donde se introduce la Thing Description que se quiere transformar en un Digital Dice. Una vez que el usuario introduce la TD, el sistema comprueba si dicha descripción cumple con las directivas definidas en el esquema JSON de la Thing Description (disponible en el portal web Digital Dice[3]). Si no, se mostrará una marca de atención en la parte superior de la caja y el botón de transformación no estará disponible para el usuario, para evitar iniciar el proceso de conversión. El listado de la Figura 2.12 muestra el código fuente que cumple con el proceso de validación mencionado anteriormente. Entre las líneas 2–6, el método comprueba si la TD de entrada está bien formada. Cuando el documento JSON es correcto, se comprueba el esquema de la TD (líneas 7–12). Una de las ventajas de validar el código con el esquema TD oficial es que el código generado a partir de la transformación no puede ser incorrecto si el modelo de origen es correcto y se asume que las reglas de transformación son correctas.

```
1   validate(data):boolean {
2       try{
3           var jsonData = JSON.parse(data);
4       }catch(e){
5           return false;
6       }
7       var ajv = Ajv({ allErrors: true });
8       apply(ajv);
9       var valid = ajv.validate(this.schema, jsonData);
10      if (!valid) console.log(ajv.errors);
11      return valid;
12  }
```

Figura 2.12: Código de validación.

[3]Portal Digital Dice / Esquema JSON-TD: `https://acg.ual.es/projects/cosmart/digitaldice`

Una vez se comprueba que la Thing Description cumple con los parámetros establecidos por el esquema, el paso de verificación comienza. Este paso inicia el procesamiento de la TD, comprobando los valores semánticos de la TD, más específicamente los parámetros `@type`. Este proceso comprueba si los valores son compatibles con los que maneja el Digital Dice. Estos parámetros alteran el DD generado en los pasos de definición y generación. Es importante destacar que los valores de `@type` pueden ser declarados en dos niveles diferentes dentro de la TD, en el nivel de la Thing y en el nivel de la interacción. Los valores de `@type` que pueden alterar el comportamiento en la construcción del Digital Dice se explican a continuación.

Primero, la propiedad `ui`. Cuando esta propiedad se encuentra al nivel de la Thing, se necesita una interfaz de usuario global con todas las interacciones del dispositivo. Si este parámetro se encuentra en el nivel de la interacción, el sistema necesita exponer una interfaz de usuario para esa interacción en particular. Un Digital Dice puede tener múltiples `InteractionAffordances` con `ui` expuesto, pero no es obligatorio exponer una interfaz global salvo que en el `@type` a nivel de Thing se tenga el parámetro `ui`.

Otro valor que puede alterar el comportamiento del Digital Dice es `@type` con el valor `virtual`. A nivel de la Thing, este valor indica que todas las `InteractionAffordances` tienen que ser virtualizadas dentro del Digital Dice. En este caso, cuando un usuario interactúa con el Digital Dice, interactúa con un dispositivo completamente virtual y no con el físico. Si se encuentra este valor de propiedad a nivel de interacción, indica que la interacción particular tiene que ser virtualizada.

Digital Dice incluye, en algunos casos, otros parámetros contextuales para `@type` para aclarar tecnologías no totalmente compatibles con la WoT, como por ejemplo KNX. Los dispositivos KNX no pueden ser representados por una Thing Description, ya que no usan tecnologías web por sí mismos. Por lo tanto, para declarar una Thing Description de un dispositivo KNX, se necesita un software middleware adecuado [Ngu et al., 2016] y un dispositivo conocido como KNX IP gateway. Además, esta Thing Description usa parámetros particulares para poder trabajar con la KNX IP Gateway (Direcciones de grupo y DataTypes KNX) derivados de esta propia tecnología de conexión. Para definir este tipo de dispositivos, se usa el valor KNX para `@type` del dispositivo, este valor indica a un Digital Dice que se trata de un dispositivo KNX (línea 5 del listado de la Figura 2.2). Esta información será tenida en cuenta en el proceso de transformación para importar las librerías necesarias.

2.6.2. Definición

El paso de definición es la etapa principal del proceso de transformación en su totalidad, ya que define los diferentes microservicios donde se lleva a cabo la transformación modelo a texto. Para ello, primero, la herramienta de transformación, después de comprobar que la Thing Description está bien formada a través de los dos primeros pasos (Verificación y Validación), descubre automáticamente qué microservicios se necesitarán para representar una *Thing* en particular.

La Figura 2.13 muestra los microservicios generados por la conversión de la TD a DD, y cómo ciertos valores de la Thing Description generan diferentes configuraciones para el Digital Dice. Diferentes propiedades, acciones o eventos generarán diferentes

configuraciones de microservicios. Las interacciones siempre generarán, al menos, los tres microservicios diferentes, `Controller`, `Reflection` y `Data Handler`. Un microservicio `Event Handler` se generará cuando la TD contenga eventos.

Tal y como se ha mencionado en la subsección anterior, se generará un microservicio `Virtualizer` o un microservicio `User Interface` cuando el parámetro semántico `@type` tenga los valores *virtual* o `ui` respectivamente. En el listado de la Figura 2.14 se puede ver este hecho, ya que las líneas 3 y 4 son dos métodos que analizan el `@type` de la Thing Description para determinar las librerías de comunicación que el proyecto necesitará, así como comprobar si se van a necesitar los microservicios `ui` o `virtualizer`. En la línea 5, se muestran los microservicios base del DD (`Controller`, `Data Handler` y `Reflection`) junto con la infraestructura auxiliar necesaria para soportar el funcionamiento del Digital Dice que solo necesita ser desplegado una vez, independientemente del número de DDs que tenga nuestro sistema. Además, entre las líneas 6–17, se definen todos los microservicios necesarios de acuerdo con la Figura 2.13.

2.6.3. Generación

Una vez definidos los microservicios, el sistema generará los diferentes archivos necesarios para crear dichos microservicios, así como la infraestructura para soportar los Digital Dice. En esta etapa es donde se lleva a cabo la mayor parte de la transformación M2T, definiendo las clases, métodos y archivos necesarios para crear una plantilla de cada microservicio que ayuda al usuario final a establecer el Digital Dice que permita controlar su dispositivo. Sin embargo, antes de eso, se debe generar la infraestructura de software que soporta Digital Dice.

La infraestructura para manejar el ecosistema de Digital Dice se basa en dos arte-

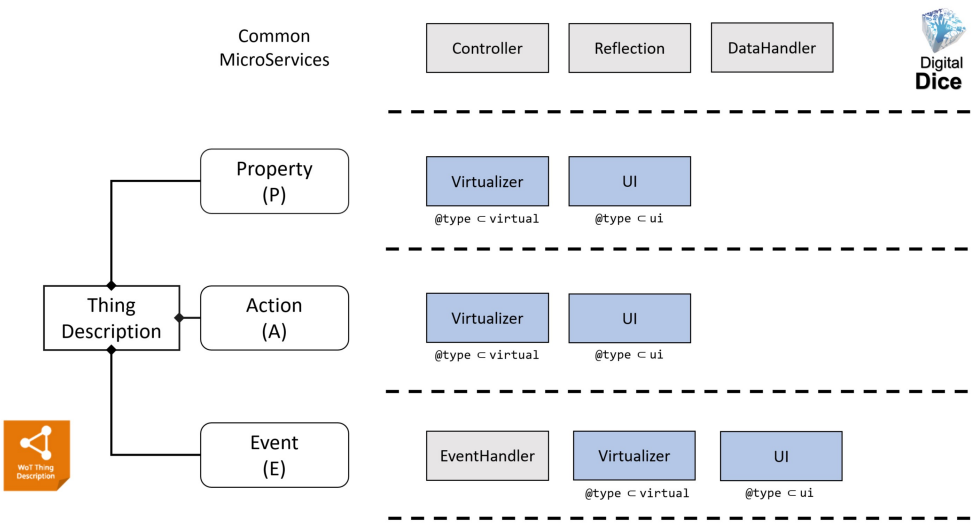


Figura 2.13: Definición de microservicios a partir de una TD.

factos necesarios para que DD funcione correctamente. El primero es una base de datos MongoDB [Chodorow, 2019] que se define mediante un archivo de imagen de docker [Nickoloff and Kuenzli, 2019]. Y el segundo es el archivo de configuración de Kubernetes [Burns et al., 2022] para establecer una configuración básica de los microservicios que el Digital Dice particular, que se está generando, se pueda levantar de manera automática. Estos archivos de configuración proporcionan diferentes maneras de escalar y de reducir los microservicios de forma automática cuando sea necesario. Además, estos archivos de configuración tienen un código base común, pudiendo ser modificado dependiendo de los microservicios descubiertos en el paso de definición.

Una vez definida la infraestructura, el proceso de generación construye los diferentes microservicios de los Digital Dice. El componente `Controller` genera una ruta y un método para cada `InteractionAffordance`, dos en el caso de que la interacción particular tenga un comportamiento SSE, o incluso más si la interacción tiene más de un `Form` para interactuar con el dispositivo. El componente `Controller` tiene otras características, como exponer la TD para manejar el Digital Dice particular o la orquestación de la solicitud enviada por el usuario a los otros servicios, actuando como puerta de enlace para el Digital Dice particular. La comunicación con el resto de los componentes de microservicio se gestiona siguiendo patrones reactivos [Malhotra, 2019]. Esto permite mejorar el rendimiento del sistema utilizando una cantidad mínima de recursos. El componente `Reflection` solicita las `PropertyAffordances` de un dispositivo por segundo y guarda los datos recuperados en la base de datos, si ha cambiado desde la última vez que se solicitó. Este componente de microservicio realiza *polling*, o llamadas consecutivas, cada segundo, si el dispositivo físico no tiene ningún protocolo de datos en tiempo real implementado. El componente `Event Handler` propaga los eventos declarados en el dispositivo al usuario cuando es necesario. En cuanto al componente `Virtualizer`, este se encarga de generar un método para cada `InteractionAffordance` declarada con el atributo `@type` como *virtual*, lo mismo con el atributo `UI`, pero con el valor *ui*.

Al generar los microservicios, se genera el código fuente por lo que el usuario de la

```
1    servicesNeeded(td) {
2        this.parsedTd = JSON.parse(td);
3        var types = this.verify(this.parsedTd);
4        this.librariesNeed();
5        var services = { services: ["infrastructure", "controller","data
             handler" y "reflection"] };
6        types.forEach(element => {
7          services.services.push(element);
8        });
9        if (this.parsedTd.events) {
10         services.services.push("eventHandler");
11       }
12       if (this.parsedTd.actions || this.parsedTd.properties) {
13         services.services.push("dataHandler");
14       }
15
16       return services;
17   }
```

Figura 2.14: Código de definición.

herramienta de transformación deberá revisarlos, cumplimentarlos y compilarlos cuando los considere oportunos. Cada faceta del Digital Dice es un proyecto individual, y cada proyecto tiene todos los archivos de código fuente necesarios, así como los archivos de dependencia (`package.json`), los `.env` necesarios para establecer configuración de puertos y otros parámetros de configuración interna de cada uno de los microservicios y un archivo `app.js` que implementa los métodos necesarios en cada uno de los microservicios. Si el usuario desea modificar el comportamiento o implementar algo diferente para el Digital Dice particular, debe modificar el código fuente generado. En algunos casos, el usuario tiene que intervenir para implementar parte de los métodos ya que el transformador es una transformación M2T parcial [Burgueño et al., 2019].

2.6.4. Empaquetado

El último paso en el proceso de transformación es el empaquetado de cada microservicio o faceta. Para ello, cada proyecto creado en el paso de generación se estructura y comprime en un archivo zip. Después de eso, el usuario puede descargarlo y modificarlo. La Figura 2.15 muestra la interfaz de usuario web del transformador TD2DD, en la que aparece un ejemplo de Thing Description de un sensor de temperatura, y todos los paquetes generados con los microservicios que necesita el Digital Dice.

En la Figura 2.16, se pueden observar los archivos resultantes de descomprimir cada uno de los empaquetados generados en la Figura 2.15. En la descompresión se generan cuatro carpetas con la siguiente información. En primer lugar, la carpeta `Controller` donde se encuentra el código fuente del microservicio, las dependencias del mismo y un `dockerfile` para construir la imagen que se despliega mediante la configuración de Kubernetes. En segundo lugar, la carpeta del `Data Handler`, que contiene, al igual que la de *Controller*, el código fuente del microservicio, dependencias y el `dockerfile`. En tercer lugar, la carpeta de infraestructura, donde se encuentran un archivo `docker-compose` con la base de datos Mongodb, y dos archivos de configuración de Kubernetes donde se desplegaran las imágenes anteriormente mencionadas. Y por último, la carpeta del microservicio `Reflection`, que contiene código fuente, dependencias y el `dockerfile`. Es importante comentar que el código fuente de los microservicios generados debe ser supervisado, dado que como ya comentamos, este es un proceso semi-automático, ya que no se pueden tener en cuenta determinados parámetros ajenos a la Thing Description, como las personalizaciones relacionadas con los sistemas donde se produzca el despliegue, o consideraciones de la lógica de negocio propia de los microservicios.

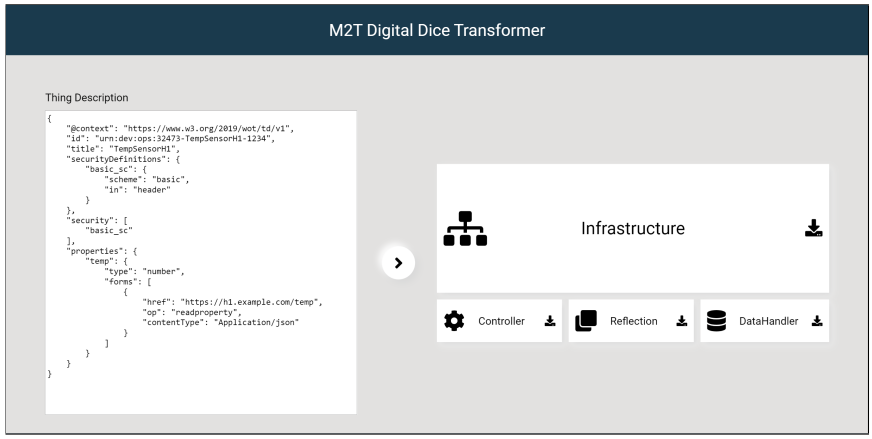


Figura 2.15: Interfaz web de la herramienta TD2DD.

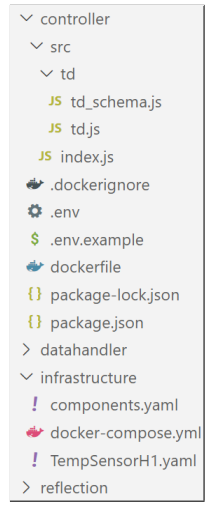


Figura 2.16: Listado de archivos que genera la herramienta TD2DD.

Capítulo 3

Capacidades de la Arquitectura Digital Dice

Capítulo 3

Capacidades de la Arquitectura Digital Dice

Contenido

3.1. Introducción

En este capítulo, se abordan las principales contribuciones del trabajo de investigación de esta tesis doctoral. Estas contribuciones constituyen los pilares fundamentales del desarrollo del componente Digital, siendo estas: la librería de comunicación WoTnectivity, el modelo de relaciones de causalidad y las estrategias de alta disponibilidad.

En la Sección 3.2 se introduce WoTnectivity, una librería multiprotocolo diseñada para facilitar la comunicación con dispositivos IoT y ciberfísicos. Se detallan los principios subyacentes y el funcionamiento de la librería, así como los protocolos compatibles, y las directrices para incorporar nuevos protocolos en el futuro.

Seguidamente, en la Sección 3.3, se aborda el concepto de *relaciones de causalidad*, esencial para la comunicación efectiva entre Things. Se definen los componentes que conforman estas relaciones y se presenta el Lenguaje Especifico de Dominio (Domain-Specific Language, DSL) utilizado para definir dichas relaciones de causalidad. Además, se proporciona un ejemplo práctico del uso de este lenguaje.

Finalmente, en la Sección 3.4, se exponen los mecanismos y estrategias para garantizar la *alta disponibilidad* de los sistemas basados en Digital Dice. Se presentan pruebas de rendimiento comparativas entre Digital Dice y la conexión directa con dispositivos IoT, gobernados por protocolos IoT conocidos, como KNX, ZigBee o HTTP. Se ilustra un ejemplo en el ámbito de las Smart Cities para la gestión de la recolección de residuos, y demostrar la eficacia y versatilidad de Digital Dice en contextos reales.

3.2. WoTnectivity

WoTnectivity es una librería multiprotocolo que ofrece un marco de trabajo diseñado para establecer conexiones con los diferentes protocolos encontrados en el mundo IoT. Se trata de una librería capaz de trabajar con diferentes protocolos de comunicación, como HTTP [Krishnamurthy and Rexford, 2001], MQTT [Hunkeler et al., 2008], WebSocket [Wang et al., 2013] o KNX [KNXAssoc, 2009]. Además, el desarrollador puede establecer la compatibilidad con otros protocolos declarando WoTnectivity como una dependencia e implementando la interfaz que este proporciona.

Esta infraestructura se creó con la idea de combinar diferentes métodos de comunicación, en un esfuerzo conjunto para establecer un patrón de uso común para diferentes dispositivos. WoTnectivity ofrece una capa de abstracción, donde se utiliza sólo un esquema de comunicación común para operar con diferentes dispositivos, lo cual facilita la forma de acceso y, asimismo, acelera el tiempo de desarrollo y mejora la eficiencia de los desarrolladores que utilizan WoTnectivity. Además, la capacidad de WoTnectivity para ser extendido y trabajar con protocolos adicionales mediante la implementación de la interfaz proporcionada, hace que la librería sea altamente adaptable y escalable para las necesidades futuras. Esto es especialmente importante en ecosistemas de continuo cambio, como en los sistemas de dispositivos IoT y ciberfísicos.

3.2.1. Fundamentos de WoTnectivity

La necesidad de desarrollar esta librería surge al tratar de hacer que los Digital Dice sean compatibles con diferentes tecnologías. Anteriormente, era necesario aprender múltiples librerías de conexión diferentes, lo que resultaba insostenible y requería revisiones constantes de la estrategia que se utilizaría para conectar con los dispositivos. Esta problemática se acentúa cuando se busca utilizar diferentes tecnologías en el futuro.

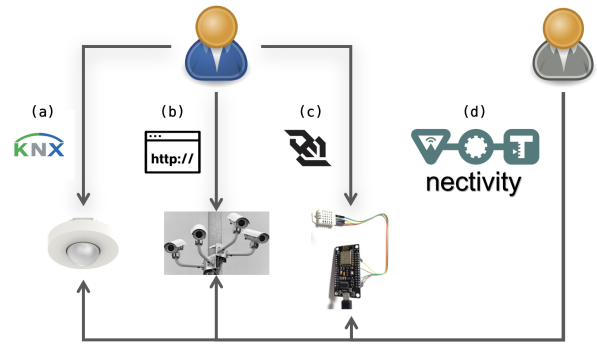
Además, en el desarrollo del modelo parcial de transformación entre Thing Descriptions y Digital Dice, se encontró que el uso de diferentes librerías dificultaba la realización del proceso con la suficiente profundidad, ya que las plantillas de los microservicios generados por la herramienta de conversión eran insuficientes e incapaces de manejar diferentes protocolos, lo que hacía que el proceso no fuese lo suficientemente exhaustivo. Además, las diferentes librerías presentan formas de uso muy diferentes entre sí. Es por eso que se identificó la necesidad de desarrollar un artefacto que permitiera establecer un patrón de comunicación común para resolver estos problemas.

En la Figura 3.1 se ilustra el uso de la librería WoTnectivity como patrón de comunicación, frente a otras librerías de comunicación para conectar con diferentes tipos de dispositivos. En esta figura se observa cómo se realizan peticiones en diferentes tecnologías de comunicación, como dispositivos KNX (a), peticiones HTTP (b) y peticiones mediante WebSockets (c), frente al uso de WoTnectivity en cualquiera de las tres tecnologías (d). WoTnectivity es una librería multiprotocolo capaz de trabajar con múltiples tecnologías implementadas como MQTT, Web Sockets, KNX y HTTP, con una estructura optimizada y un comportamiento similar, independientemente de la tecnología utilizada. La librería ha sido desarrollada tanto en Java como en Node.js [Cantelon et al., 2014], pero la idea detrás de la misma puede ser traducida a otros lenguajes de programación.

A la hora de decidir cómo empaquetar WoTnectivity se estudiaron dos enfoques diferentes para desarrollar la idea de una librería multiprotocolo. El primer enfoque consistía en crear un proyecto donde residir todos los protocolos de conexión considerados. Sin embargo, este enfoque agregaba cierta complejidad y tamaño a la librería. El segundo enfoque consistió en crear proyectos separados para cada uno de los protocolos de conexión implementados. Cada uno de ellos debían cumplir con la interfaz principal del proyecto WoTnectivity, con todos los subprotocolos implementados. Este último enfoque tiene la ventaja de ser más fácil de mantener y más eficiente, en términos de memoria.

La interfaz definida en la librería de comunicación debía de ser lo suficientemente abierta, ya que no todas las tecnologías funcionan con los mismos protocolos de transferencia de datos y suelen tener múltiples patrones de conexión diferentes. Después de estudiar las distintas tecnologías soportadas por la librería, se decidió clasificarlas en dos categorías distinguidas por su tipo de conexión: *persistentes* y *no persistentes*. Las conexiones persistentes dejan una conexión abierta con el dispositivo en cuestión, no sólo para recibir una respuesta aislada, sino también para recibir los mensajes vinculados continuamente. Para este tipo de conexión, se requiere la implementación de una acción `listener`, clase que implementa un comportamiento cuando se reciben nuevos mensajes, y que sirve para controlar el flujo de datos de esa conexión. El segundo tipo de conexión se establece cuando el usuario de la librería simplemente desea interactuar con un dispositivo para obtener una respuesta.

Estos diferentes comportamientos de uso se deben tener en consideración cuando se desee agregar nuevos protocolos de comunicación a la librería, ya que la solución debe ser compatible con ambos tipos de conexión. La librería WoTnectivity ofrece una solución a la complejidad y la heterogeneidad de las tecnologías utilizadas en la Web de las Cosas (WoT), permitiendo establecer un patrón de comunicación común para diferentes dispositivos. La implementación de una interfaz abierta y la clasificación de los diferentes comportamientos de conexión hace posible que WoTnectivity pueda ser extendida con facilidad para agregar nuevas tecnologías de comunicación.



(a)

```
this.remote = new InetSocketAddress(address, 3671);
try (KNXNetworkLink knxLink = KNXNetworkLinkIP.newTunnelingLink(null,this.remote, false,
TPSettings.TP1);
        ProcessCommunicator pc = new ProcessCommunicatorImpl(knxLink)) {
        GroupAddress groupAddress = new GroupAddress(group);
        final Datapoint dp = new StateDP(groupAddress, "my datapoint
        "+groupAddress.toString());
        dp.setDPT(0, datatype);
        String result = pc.read(dp).toString();
        pc.close();
        knxLink.close();
        return result;
}
catch (KNXException | InterruptedException e) {
        throw new Exception("Error accessing KNX datapoint: " + e.getMessage());
}
```

(b)

```
var request = HttpRequest.newBuilder()
.POST(BodyPublishers.ofString(payload))
.uri(URI.create(address))
.headers(headersArrayList.toArray(headersArray))
.build();
return client.sendAsync(request, BodyHandlers.ofString())
.thenApply(HttpResponse::body)
.exceptionally(e -> "Error: " + e.getMessage());
```

(c)

```
WebSocket.Listener listener = new WsListener()
String address = "wss://device3.com";
String payload = "test";
this.wsConnection = this.client.newWebSocketBuilder().buildAsync(URI.create(address),
this.listener).join();
return this.wsConnection.sendText(payload, true);
```

(d)

```
IRequester.sendRequest(address, configuration, payload).get();
```

Figura 3.1: Uso de WoTnectivity frente a otras librerías.

3.2.2. Protocolos en WoTnectivity

Para el desarrollo de la librería WoTnectivity se han construido dos versiones, una en Java y otra en Node.js. Los ejemplos de WoTnectivity que sedescriben en esta sección en adelante, son de la versión Java. La versión Node.js sigue los mismos principios. La implementación se puede consultar en los repositorios oficiales de npm[1].

La librería de WoTnectivity utiliza la interfaz `IRequester` que se muestra en la Figura 3.2. Esta interfaz permite crear una forma común de implementar diferentes protocolos y a la vez ofrece una forma de poder ampliarlos. Cada protocolo agregado a la librería debe implementar esta interfaz, cuyo único método es `sendRequest`. Como se observa en la Figura 3.3, la interfaz en Node.js es la misma, sin tener en cuenta los tipos de datos, dado que el método y los parámetros son los mismos. Este método requiere tres parámetros: **address**, que define la dirección a la que se debe enviar la solicitud; **configuration**, que permite la definición de diferentes parámetros de configuración de la solicitud; y **payload**, que es el cuerpo de la solicitud y debe ser lo más genérico posible para que diferentes protocolos puedan manejar diferentes tipos de cargas útiles.

```
1 public interface IRequester{
2   CompletableFuture<?> sendRequest(String address,
3   HashMap<String, Object> configuration,
4   Object payload);
5 }
```

Figura 3.2: `IRequester.java`.

```
1 interface IRequester{
2   sendRequest(address: String, configuration: {}, payload: {});
3 }
```

Figura 3.3: `IRequester.ts`.

Es importante tener en cuenta que la librería solo funciona con versiones de Java 11 o superiores, debido al uso de *CompletableFuture* [Davis and Davis, 2019] como tipo de retorno del método. *CompletableFuture* es un tipo introducido en Java 8 que permite trabajar con solicitudes persistentes no bloqueantes, lo que Java denomina programación reactiva [Malhotra, 2019], esencial para que la librería pueda funcionar correctamente. El establecimiento de dos librerías distintas para Node.js y Java viene motivado por nuestra intención de desligar WoTnectivity del concepto de Digital Dice y que pueda ser utilizado por cualquier otro proyecto que lo necesite.

A continuación, se definen un par de ejemplos sobre cómo añadir nuevos protocolos a WoTnectivity. La elección de estos dos ejemplos permite entender mejor cómo implementar protocolos de los tipos anteriormente definidos al comienzo de esta sección, *persistentes* y *no persistentes*. Cualquier otra tecnología que se desee implementar se puede desarrollar con un patrón muy similar a alguno de estos ejemplos.

[1]WoTnectivity-KNX npm –https://www.npmjs.com/package/wotnectivity-knx

3.2.2.1. Implementando el protocolo HTTP

Como se ha comentado anteriormente, se han definido dos protocolos en dos categorías separadas: *persistentes* y *no persistentes*. Para los protocolos *persistentes*, se debe de declarar en el constructor una acción `listener` específica para ese protocolo, con el comportamiento deseado cuando se reciban peticiones en la librería. Si no se pasa un `listener` en el constructor, se define un `listener` predeterminado en la librería, cuyo comportamiento es simplemente el de mostrar los mensajes recibidos en la consola.

El primer protocolo que usaremos como ejemplo es HTTP. La Figura 3.4 muestra una parte de la implementación de `HttpReq`. En la línea 1 del códigose declara el nombre de la clase, así como la interfaz que esta clase va a implementar con `Implements IRequester`. El constructor predeterminado en las líneas 3-5 inicializa el cliente `HttpClient`. En este caso, sólo hay un constructor porque la conexión HTTP no requiere una conexión permanente, por lo tanto HTTP pertenece a lo que se ha denominado protocolos *no persistentes*. Sólo se realiza una solicitud y se espera una respuesta con el recurso necesario, por lo que no se necesita un `listener` para administrar este tipo de conexión. Las líneas 6-29 definen un método (`sendPostRequest`) que crea una solicitud POST a la **dirección** dada con un **cuerpo** de solicitud específico y los **encabezados** esperados que vienen como un mapa de clave-valor en el parámetro. Las líneas 31-48 definen la función requerida por la interfaz mostrada en la Figura 3.2 (`sendRequest`). Esta función recuperará del parámetro de configuración el *tipo de método* de la solicitud HTTP. Luego, enviará esa solicitud a la función particular correspondiente a ese *tipo de método*, donde enviará la **dirección**, los **encabezados** (recuperados del parámetro de **configuración**) y el **cuerpo**, si es necesario para esa función.

La librería Java así como su documentación se encuentra disponible en un repositorio público WoTnectivity de GitHub[2].

3.2.2.2. Implementando el protocolo KNX

El tratamiento de KNX es algo poco distinto a lo visto con la implementación de HTTP. KNX es un protocolo *persistente*, al existir un tipo de peticiones (`subscribe`) en las cuales se establece un canal de comunicación entre el cliente y el servidor, que se mantendrá abierto hasta que el cliente lo cierre. Esto conlleva la necesidad de implementar una acción `listener` para establecer el comportamiento que tendrá cuando reciba un mensaje del servidor. Por defecto la librería KNX ya implementa un `listener` genérico que lo único que hace es imprimir por pantalla el mensaje recibido. No obstante, el usuario de la librería puede instanciar su propio `listener` y pasarlo como parámetro al constructor de la clase `KnxReq`, que es la clase que implementa la interfaz `IRequester` y que se encarga de gestionar las peticiones al protocolo KNX. La implementación completa de la librería WoTnectivity-KNX se puede consultar en el repositorio oficial de GitHub[3].

El listado de la Figura 3.5, muestra la implementación parcial de la clase `KnxReq` que implementa la interfaz `IRequester` y que se encarga de gestionar las peticiones al protocolo KNX. La interfaz dispone del método `sendRequest` (línea 23) que recibe tres

[2]WoTnectivity – `https://github.com/acgtic211/wotnectivity`
[3]WoTnectivity-KNX – `https://github.com/acgtic211/wotnectivity-knx`

parámetros: `address`, `configuration` y `payload`. En el caso de KNX, el parámetro `address` es la dirección IP del servidor KNX; `configuration` es un objeto de tipo `HashMap` con la dirección del grupo al que se quiere suscribir `group`, el tipo de dato que se espera recibir `dataType` y el tipo de *request* que se espera ejecutar (`requestType`); El parámetro `payload` es una cadena de texto con el valor de la petición que se va a escribir en la dirección de grupo, sólo si es de tipo `write`, ya que para este tipo de peticiones se manda un valor, y para el resto (`read` y `subscribe`) no, pues son de lectura.

```
1 public class HttpReq implements IRequester {
2     private HttpClient client;
3     public HttpReq() {
4         this.client = HttpClient.newHttpClient();
5     }
6     private CompletableFuture<String>
7     sendPostRequest(String address,
8     HashMap<String, Object> headers, String payload){
9         List<String> headersArrayList =
10        new ArrayList<>();
11
12        headers.forEach((k,v)->
13        {headersArrayList.add(k);
14        headersArrayList.add(v.toString());});
15
16        String[] headersArray =
17        new String[headersArrayList.size()];
18
19        var request = HttpRequest.newBuilder()
20            .POST(BodyPublishers.ofString(payload))
21            .uri(URI.create(address))
22            .headers(headersArrayList.toArray(headersArray))
23            .build();
24
25        return client.sendAsync(request,
26        BodyHandlers.ofString())
27        .thenApply(HttpResponse::body)
28        .exceptionally(e -> "Error: " + e.getMessage());
29    }
30    ...
31    @Override
32    public CompletableFuture<?> sendRequest(String address,
33    HashMap<String, Object> configuration, Object payload)
34    {
35        String methodType =
36        configuration.get("MethodType").toString();
37
38        CompletableFuture<String> output;
39
40        if (methodType.equals("POST")){
41            output = this.sendPostRequest(address,
42            configuration.get("headers"),
43            payload.toString());
44        }else if (methodType.equals("GET")){
45        ...
46        }
47        return output;
48    }
49 }
```

Figura 3.4: Código de implementación para peticiones HTTP. `HttpReq.java`.

```java
public class KnxReq implements IRequester {
    InetSocketAddress remote;
    ProcessListener pl;

    public KnxReq() {
        this.pl = new KnxProcessListener();
    }
    public KnxReq(ProcessListener pl) {
        this.pl = pl;
    }
    public void subscribeToKNXBuffer(String address, String group, String
         datatype) throws Exception {
        final InetSocketAddress remote = new InetSocketAddress(address, 3671);
        try (KNXNetworkLink knxLink = KNXNetworkLinkIP.newTunnelingLink(
                null, remote, false, TPSettings.TP1);
                ProcessCommunicator pc = new ProcessCommunicatorImpl(knxLink)) {
            pc.addProcessListener(pl);
            while (knxLink.isOpen())
                Thread.sleep(1000);
        } catch (final KNXException | InterruptedException | RuntimeException e)
             {
            System.err.println(e);
        }
    }
    public CompletableFuture<?> sendRequest(String address, HashMap<String,
         Object> configuration, Object payload) {

        String requestType = configuration.get("requestType").toString();
        configuration.remove("requestType");
        CompletableFuture<?> output = new CompletableFuture<String>();
        if (requestType.equals("read")) {
            output = CompletableFuture.supplyAsync(() -> {
                try {
                    return readStatus(address, configuration.get("group").
                        toString(),
                            configuration.get("dataType").toString());
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return "Error in request";
            });
        } else if (requestType.equals("write")) {
            output = CompletableFuture.runAsync(() -> {
                try {
                    writeStatus(address, configuration.get("group").toString(),
                            configuration.get("datatype").toString(), payload.
                                toString());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        } else if (requestType.equals("subscribe")) {
            output = CompletableFuture.runAsync(() -> {
                try {
                    subscribeToKNXBuffer(addr, configuration.get("group").
                        toString(), configuration.get("datatype").toString());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        } else {
            output = null;
        }
        return output;
    }
}
```

Figura 3.5: Código de implementación para peticiones KNX. `KnxReq.java`.

En la Figura 3.5, se puede observar que el método `sendRequest` (líneas 23–59), hace lo mismo que hacía WoTnectivity-Http en la Figura 3.4, básicamente discernir entre las distintas operaciones y lanzar el método correspondiente. En la Figura 3.5, se muestra el método `subscribeToKNXBuffer` (líneas 11–22) que es el encargado de gestionar las peticiones de tipo `subscribe`. Este método se encarga de crear un objeto de tipo `InetSocketAddress` que contiene la dirección IP del servidor KNX y el puerto por defecto (3671), y a continuación crea un objeto de tipo `KNXNetworkLink` que es el encargado de la gestión de la conexión con el servidor KNX. Una vez creados estos objetos, se crea un objeto de tipo `ProcessCommunicator` que es el encargado de ejecutar las peticiones al servidor KNX. A continuación se añade un `listener` al objeto `ProcessCommunicator` creado anteriormente, y se mantiene la conexión abierta hasta que el usuario la cierre. El método `subscribeToKNXBuffer` se ejecuta de forma asíncrona, es decir, se ejecuta en un hilo diferente al hilo principal de la aplicación, y por tanto no bloquea la ejecución del resto de la aplicación.

3.2.3. Escenario Ejemplo

En esta sección se muestra un ejemplo particular de uso de la librería. El escenario consiste en cambiar el estado de una persiana en un dispositivo KNX controlado por una puerta de enlace KNX-IP y recuperar la temperatura de un sensor con un servidor de API gestionado en un microcontrolador ESP-8266. En la Figura 3.6 se ilustra el comportamiento del escenario. Si la temperatura supera los 30°C o es inferior a los 10°C, la persiana debe cerrarse. Entre 10°C y 20°C la persiana debe estar semiabierta y entre 20°C y 30°C debe estar abierta.

Los dos dispositivos IoT que se van a utilizar y sus propiedades son una persiana y un sensor de temperatura. La *persiana* es tratada como un dispositivo actuador que acepta un valor decimal entre 0 y 1, siendo 0 totalmente cerrado y 1 totalmente abierto. La dirección de grupo del dispositivo es 1/0/1. La dirección de grupo de KNX representa un punto final virtual relacionado con uno o más dispositivos para intercambiar información.

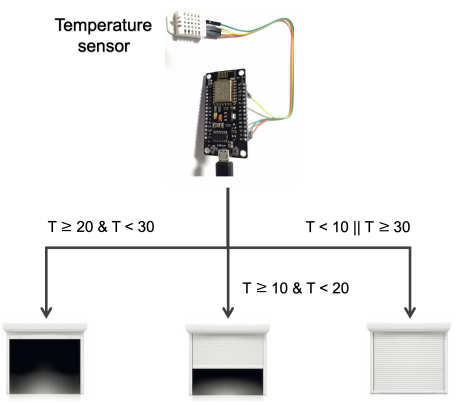


Figura 3.6: Escenario de ejemplo.

La dirección `ip` de la puerta de enlace KNX que lo gestiona es 10.0.0.4. Por otro lado, el *sensor de temperatura* tiene una dirección para recuperar la temperatura del sensor en particular en `http://localhost:17345/temp`. Será necesario establecer una petición GET a esa dirección con la autorización para recibir la respuesta en texto plano.

Una vez definidos los dispositivos IoT utilizados en el ejemplo, el siguiente paso consiste en agregar las librerías necesarias como dependencias de la aplicación *wotnectivity-http* y *wotnectivity-knx*. Los siguientes pasos son necesarios para utilizar las librerías en el escenario explicado anteriormente:

(a) En primer lugar, instanciar la clase `HttpReq` y la clase `KnxReq` de nuestra librería. Ambas clases implementan la interfaz `IRequester`:

```
KnxReq knxreq = new KnxReq();
HttpReq httpreq = new HttpReq();
```

(b) El siguiente es declarar la configuración de la solicitud HTTP y enviar la solicitud:

```
Map<String,Object>configurationT= new HashMap<String,Object>()
Map<String,String>headers = new HashMap<String,String>()
configurationT.put("MethodType", "GET")
headers.put(çontent-type", "text/plain");
headers.put("Authorization", "ASJdni32njsdu54");
configurationT.put("headers", headers)
String tempData =
    httpreq.sendRequest("http://localhost:17345/temp", configurationT, "")
```

(c) A continuación, analizar la temperatura y compararla con los valores descritos anteriormente:

```
if(double.parse(tempData) <10 || double.parse(tempData) >= 30)
if(double.parse(tempData) >= 10 && double.parse(tempData) <20)
if(double.parse(tempData) >= 20 && double.parse(tempData) <30)
```

(d) En caso de que alguna de las condiciones se cumpla, se procederá a declarar la configuración de la solicitud de la librería *KnxReq*, con el tipo de datos como un número decimal entre 0 y 1. Es importante destacar que en KNX la representación de este tipo de datos es `5.001`, como se puede ver en el código a continuación. Posteriormente, se utilizará el parámetro del grupo y el método de escritura para definir en la librería que se desea cambiar el estado de la persiana.

```
Map<String,Object>configurationS = new HashMap<String,Object>()
configurationS.put("group","1/0/1")
configurationS.put("dataType","5.001")
configurationS.put("requestType","write")
knxreq.sendRequest("10.0.0.4", configuration, 0.0)
```

KNX es un protocolo de comunicación de los que anteriormente denominamos *persistentes*. Este protocolo permite suscripciones a una dirección de grupo específica. En

este tipo de protocolos se tiene que implementar un `listener` que decide lo que se debe hacer cuando se recibe un mensaje. No obstante, WoTnectivity-KNX ya cuenta con una acción "listener" por defecto que lo que hace es imprimir por pantalla los datos recibidos en el mensaje. Si se quisiera implementar otro comportamiento al utilizar el protocolo, se necesitaría incluir un `listener` con una serie de métodos específicos en el constructor, estos métodos vienen declarados en la interfaz *ProcessListener*. Un ejemplo de esta implementación se puede encontrar en la carpeta *utils* de la librería en forma de *KnxProcessListener* (el `listener` KNX predeterminado).

En la carpeta *utils* de cada librería, está disponible la implementación predeterminada de varios `listeners`, como *KnxProcessListener* o *WsListener*, entre otros. Además de los `listeners`, estas librerías incluyen varias herramientas destacadas, como la clase *KnxUtils*, que permite a los usuarios descubrir servidores KNX-IP o monitorear grupos en una red de área local (LAN), así como un manejador de cuerpo JSON para analizar objetos JSON en WoTnectivity.

El objetivo final de Digital Dice es lograr una verdadera interoperabilidad y gestión de dispositivos IoT y ciberfísicosheterogéneos. El uso de microservicios proporciona la flexibilidad necesaria que el sistema requiere para ser robusto y fácilmente mantenible. La capacidad de replicar microservicios individuales sólo cuando se necesita ayuda a trabajar con los recursos disponibles con la máxima eficiencia. Para lograr este objetivo, el primer paso de la propuesta fue crear una forma de conectar con diferentes dispositivos sin ser necesario tener una gran experiencia previa en el manejo de cientos de protocolos diferentes para controlar dichos dispositivos.

3.3. Relaciones de Causalidad

Internet de las Cosas (IoT) es un dominio tecnológico que abarca una amplia variedad de dispositivos interconectados que generan, procesan y comparten datos en tiempo real. Uno de los mayores desafíos en el desarrollo de sistemas IoT es la interoperabilidad y la integración, ya que estos dispositivos suelen utilizar diferentes protocolos y tecnologías de comunicación. La falta de interoperabilidad dificulta la integración de diferentes dispositivos en sistemas ciberfísicos y limita la eficiencia y la escalabilidad del sistema.

Existen diversos mecanismos de comunicación que se utilizan en IoT, como patrones de comunicación como publicador-suscriptor [Kashyap et al., 2018], interfaces proporcionadas-requeridas [De et al., 2011] o comunicación dirigida por eventos [Zhang et al., 2014]. Aunque estos enfoques habilitan la interacción entre dispositivos IoT, la naturaleza heterogénea de los protocolos y de las tecnologías utilizadas sigue planteando un desafío para los desarrolladores y usuarios de este dominio.

Las recomendaciones de la Web de las Cosas (WoT) del World Wide Web Consortium (W3C) [W3C, 2022a] han establecido una capa de abstracción que ayuda a superar el desafío de la interoperabilidad en sistemas IoT y ciberfísicos. La WoT proporciona una forma uniforme de realizar la comunicación y la descripción de la funcionalidad de los dispositivos IoT, utilizando estándares y tecnologías web. Sin embargo, una de las limitaciones de la WoT es que no tiene en cuenta la posible automatización de la comunicación entre dispositivos sin intervención humana.

En esta sección se establece un sistema de interoperabilidad entre dispositivos IoT y ciberfísicos a través de un lenguaje basado en **causas** y **efectos**, ya presentado en el capítulo anterior. Este lenguaje hace uso de la potencia proporcionada por la especificación Thing Description de la WoT [W3C, 2022b], para establecer una capa de abstracción en la que se pueden declarar interacciones entre dispositivos o software que representan esos dispositivos (*servients*). La Figura 3.7 muestra una relación de causalidad propuesta entre diferentes *servients*, donde una propiedad de una Thing A provoca un efecto que depara en una acción de una Thing B a través de una relación de causalidad.

Durante el desarrollo del modelo Digital Dice se pudo comprobar que las recomendaciones WoT no permitían describir con facilidad la comunicación entre dispositivos, siendo esto una limitación de la WoT. Por tanto, se estudió una primera aproximación mediante el uso de sistemas procesamiento de eventos complejos (Complex Event Procesing, CEP) para habilitar la interacción entre Things como una forma de mitigar este problema. Sin embargo, esta apoximación entraba en conflicto con los principios de los Digital Dice, como el principio de responsabilidad única o la descentralización. Estos principios son propios de las arquitecturas de microservicios, como la implementada para Digital Dice, y ayudan a evitar fallos del sistema.

La Figura 3.7 muestra la propuesta de comunicación entre dispositivos IoT implementados con Digital Dice. El sistema establece una relación de causalidad entre diferentes Things, lo que les permite escuchar las diferentes propiedades, acciones o eventos (causas) de las otras Things, solo en el caso de que tengan un impacto en las interacciones predefinidas de la Thing representada. Existe restricción particular: los efectos sólo pueden actuar sobre las interacciones mostradas en la Thing Description relacionada con la relación de causalidad declarada, es decir cada Thing se debe de preocupar por los efectos que deparen en cambios sobre si misma. Esta restricción está justificada por motivos de seguridad, ya que de esta manera el sistema evita la posibilidad de que una Thing afecte negativamente a otras con la posible inyección de código malicioso.

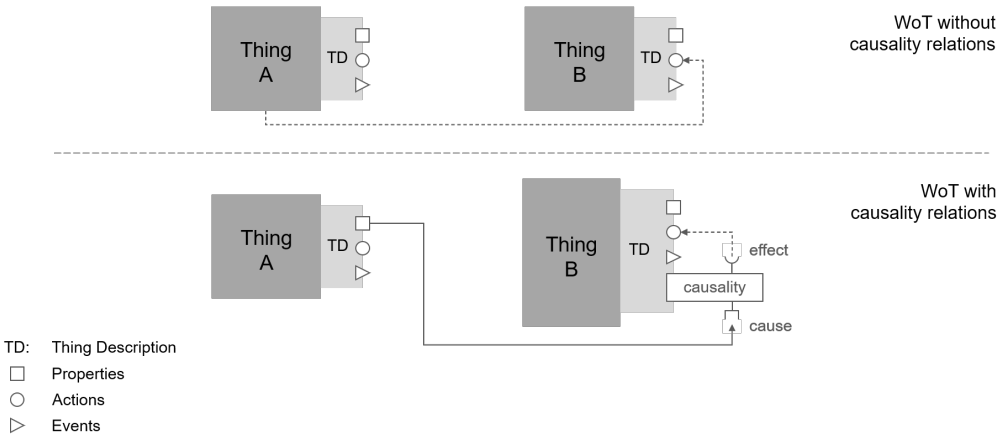


Figura 3.7: Relación de causalidad.

En la Thing Description propuesta por la WoT de la W3C, si una acción de una Thing A se ejecuta al cambiar el valor de una propiedad de una Thing B, dicha relación está oculta en la implementación de la Thing B como una caja negra (ver la parte superior de la Figura 3.7). Esta representación de la relación puede ser útil. Con este fin, la propuesta presentada en esta sección es crear un lenguaje que pueda manejar relaciones entre dispositivos donde las interacciones de la Thing A (causas) pueden tener un impacto directo en las interacciones de la Thing B (efectos). En la situación mencionada, el enlace entre la acción de la Thing B que se ejecuta debido al cambio en el valor de una propiedad de la Thing A sería representado por un enlace causal como parte del modelo de relación de causalidad que controla la Thing B.

Esta solución nos permite establecer lo que se puede considerar un sistema descentralizado de procesamiento de eventos, en el cual cada dispositivo maneja los eventos que impactan en sí mismo. Esto trae consigo una serie de ventajas sobre los enfoques más típicos en los sistemas CEP [Boubeta-Puig et al., 2019, Sun et al., 2019, Grez et al., 2019, Mdhaffar et al., 2017]. Algunas de las principales ventajas de un sistema de eventos descentralizado son la mejora del rendimiento del sistema o la prevención de posibles puntos de fallo. En un sistema de eventos centralizado, un tiempo de inactividad del servicio de eventos afecta a todos los eventos complejos controlados por él. Diferente es en un sistema descentralizado, donde un fallo en uno de los dispositivos sólo afecta a las interacciones donde este sistema es necesario, es decir en los eventos donde una propiedad o evento de ese dispositivo forme parte. Sin embargo, la idea de un sistema de eventos descentralizado tiene el problema de que todos los componentes (*servients* WoT) que hacen uso de este lenguaje deben saber cómo manejarlo. Para permitir una solución universal, se propone un sistema que pueda funcionar simultáneamente como un sistema descentralizado, con aquellos *servients* que saben cómo manejar relaciones de causalidad y que contienen un interprete para dichas relaciones, y de manera centralizada para aquellos *servients* que dependen de implementaciones típicas de la WoT, donde se cuenta con un servicio externo que se encarga de gestionar esas relaciones.

En la Subsección 3.3.2 se muestra un ejemplo de escenario completo de interacción entre dispositivos mediante relaciones de causalidad.

3.3.1. Modelado de interacciones entre Things

En esta sección se explica cómo se usarán las relaciones de causalidad del modelo Digital Dice para habilitar la interacción entre los *servients* WoT sin la intervención del experto. En primer lugar, hay que entender que las Things se definen por sus capacidades de interacción (representadas por las acciones, propiedades y eventos) dentro de un documento Thing Description (TD). Es importante también comprender esto porque en la relación de causalidad, esas capacidades de interacción son las “causas” y “efectos” que se utilizan para la comunicación entre las Things.

Se entiende por *causa* a una interacción de una Thing que provoca la ejecución de un efecto; por ejemplo, cuando la propiedad temperatura de un sensor supere determinado valor. Un *efecto* es una interacción que se ejecutará si una o más causas relacionadas con ese efecto cumplen una serie de requisitos relacionados con esas causas; por ejemplo, activar los rociadores de agua en un sistema antiincencios en la casa.

En una relación de causalidad, las *causas* pueden provenir de diferentes Things, mientras que los *efectos* sólo pueden ejecutar interacciones de la Thing que está controlando la relación de causalidad. Este es el primer precepto en nuestra propuesta, una Thing sólo puede tener una relación de causalidad, en esta relación se han de representar todos los efectos que deben ser controlados por la Thing. Esto introduce una nueva capa de seguridad ya que los efectos de una Thing solo pueden tener repercusiones sobre si mismas. Al mismo tiempo, esto evita la necesidad de consultas más complejas cuando un *servient* WoT, en nuestro caso un Digital Dice, que puede manejar relaciones de causalidad, accede al subsistema para preguntar por los efectos que tiene que manejar, ya que todos están concentrados en un solo documento.

En la Figura 3.8 se muestra la propuesta para las relaciones de causalidad. En este metamodelo, se definen cuatro clases o entidades diferentes que respaldan la idea de la relación de causalidad. En primer lugar, la clase `Thing`, que define las propiedades para el dispositivo (real o virtual) que la relación de causalidad en particular está controlando. Esta clase tiene dos propiedades importantes que deben tenerse en cuenta. En primer lugar, la propiedad `thingID`, que debe ser la misma que el `id` de la Thing en la Thing Description donde se están produciendo los efectos. En segundo lugar, la propiedad `isManaged` es un valor booleano que indica si la relación de causalidad debe ser gestionada por el sistema de causalidad (centralizado) o si es gestionada por el propio *servient* WoT (descentralizado).

La segunda clase a tener en cuenta en este metamodelo es la clase `Link`. Esta clase es similar a la clase `Link` de la Thing Description; en ella se declaran las cosas que se deben de tener en cuenta por la relación de causalidad, con tres propiedades diferentes: `rel`, que indica el nombre de la relación, `href`, la dirección HTTP de la Thing Description que representa la Thing relacionada, y `content-type`, que en esta primera iteración de las relaciones de causalidad siempre deben ser del tipo de medio IANA [Melnikov et al., 2023] *application/td+json*, que hace referencia a una Thing Description de W3C representada en notación de objetos JavaScript (JSON).

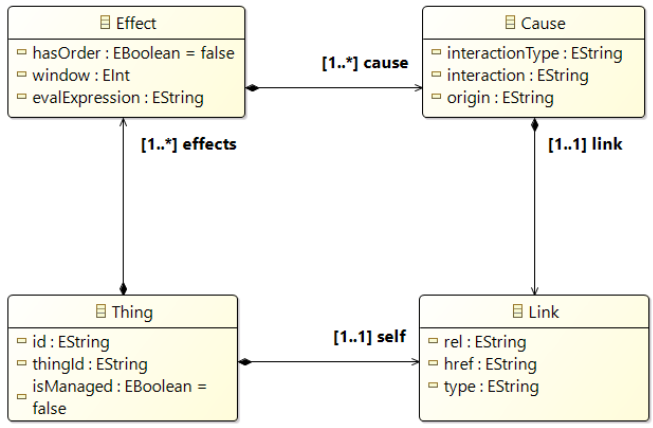


Figura 3.8: Metamodelo de relación de causalidad.

La siguiente clase que se muestra en la Figura 3.8 es la clase `Effect`. La clase `Effect` tiene una serie de aspectos derivados del mecanismo de procesamiento de eventos complejos CEP, y que se declaran aquí como propiedades, siendo estas las siguientes:

- `hasOrder`. Esta propiedad indica si las causas del efecto tienen que producirse en ese orden particular.
- `window`. Esta propiedad determina si las causas deben ejecutarse en una ventana de tiempo particular para que tenga lugar el efecto.
- `evalExpression`. Esta propiedad tiene la lógica de selección, así como lo que el sistema tendrá que hacer si ocurre la lógica de selección y cumple con las propiedades declaradas anteriormente. El lenguaje utilizado en `evalExpression` es JavaScript. La ventaja de este enfoque es que podemos utilizar toda la expresividad del lenguaje para llevar a cabo acciones que normalmente no serían posibles con un lenguaje de consulta simple, como bucles, condicionales, expresiones matemáticas, multi-acciones en una regla simple, etc. La principal desventaja de usar un lenguaje completo es que necesitamos tener un *evaluador* de la variable `evalExpression` que tenga en cuenta el hecho de que un *efecto* sólo puede ejecutar interacciones de la Thing relacionada con la causalidad que controla la Thing en particular. Este *evaluador* añade complejidad al *servient*, pero es necesario para aislar el comportamiento de una Thing.
- `causes`. Relacionado con la clase `Cause` que veremos a continuación.

Por último, está la clase `Cause` del metamodelo, encargada de definir las causas de un efecto particular. Sus propiedades son las siguientes:

- `interactionType`. Esta propiedad determina el tipo de interacción de la causa. En la primera iteración de la relación de causalidad, solo puede tener cuatro valores diferentes: `property`, `action`, `event` o `effect`. Añadir `effect` a los tipos de interacciones controlados por la relación de causalidad, viene dado por la posibilidad de tener cierta retroalimentación, y de esta manera los `effects` puedan convertirse en nuevas causas de otros `effects` distintos.
- `interaction`. Declara el nombre de la interacción.
- `origin`. Declara el origen de la causa, como si proviene del dispositivo físico, de la acción humana o si es una consecuencia de otro efecto.
- `link`. Esta propiedad indica si la `Cause` proviene del dispositivo en sí, con la palabra clave `self`, o proviene de un `link` que apunta a otra Thing Description que contenga el valor de la interacción que buscamos.

La relación de causalidad sigue un patrón de trabajo similar al de un sistema de CEP clásico, pero con un enfoque diferente. A diferencia de los CEP clásicos, que son sistemas centralizados con reglas evaluadas cada vez que ocurre un evento, las relaciones de causalidad ofrecen dos enfoques distintos. Por un lado, el sistema puede funcionar en modo descentralizado, que es el modo de trabajo por defecto, donde cada Thing maneja su lógica requerida, reduciendo el riesgo de fallo total. Por otro lado, nuestro sistema puede operar de manera centralizada, como los CEP clásicos, si el *servient* WoT particular no tiene lo que se denomina *causality core*. Este es un artefacto de software capaz de analizar modelos de relaciones de causalidad y se utiliza tanto en el subsistema de causalidad, cuando el sistema funciona de manera centralizada, como en los *servients* WoT compatibles, cuando los propios *servients* procesan las causalidades.

Trabajar con estos dos enfoques diferentes dota al sistema de varias ventajas en términos de manejo de errores. Los posibles errores en las relaciones de causalidad pueden ser de dos tipos: los generados por los dispositivos involucrados en una relación de causalidad o los generados en servicio centralizado. Si el error proviene de un dispositivo, el impacto se limita a las relaciones de causalidad donde interviene ese dispositivo. Sin embargo, si el error proviene del servicio centralizado, tendrá un mayor impacto que si proviene de un *servient*, pero aquellos *servients* que ya están administrando sus relaciones causales seguirán evaluándolas sin problemas. En contraste, en la mayoría de los sistemas CEP, si falla el núcleo, todo el sistema deja de funcionar.

Al igual que en la mayoría de los sistemas CEP, la relación de causalidad implica una fase de selección, la cual proviene de las causas que manejan un efecto específico, así como fases de coincidencia y derivación que ocurren al ejecutar la expresión de evaluación.

Las relaciones de causalidad permiten el establecimiento de un flujo de comunicación entre diferentes Things, gobernado por *causas* y *efectos*, formalizando así los patrones de interacción entre Things sin intervención humana. El Anexo A contiene el esquema usado para validar las relaciones de causalidad. El esquema representa los tipos y relaciones de cada una de las variables de nuestro modelo de causalidad, de esta manera, a través de un validador, se puede, antes de pasar a controlar los *efectos* de esa relación, comprobar que esta bien declarada.

3.3.2. Escenario ejemplo

A continuación, se describirá cómo definir las relaciones de causalidad usando, para ello, un escenario concreto: un sistema domótico de una habitación para una casa inteligente (Hogar Digital). Esta habitación tendrá cuatro Things diferentes; una persiana (`blind`) con una propiedad de estado `status`, cuyo valor es número entre 0 y 100, indicando el 0 que la persiana está totalmente cerrada y 100 totalmente abierta (parcialmente para valores intermedios); un sensor de temperatura (`tempSensor`) con la propiedad `temp` que proporciona la temperatura de la habitación; un sistema de aire acondicionado (`AC`) donde se puede establecer la temperatura de la habitación con `setTemperature` y apagarlo con la interacción `shutDown`; y un sensor de presencia `movement` capaz de detectar movimiento a partir del evento `movement`.

Para este escenario, hay que establecer una lógica de comportamiento para los eventos. La Figura 3.9 muestra un diagrama causa/efecto ejemplo para el escenario presentado. El comportamiento del sistema que se muestra en la figura establece que la persiana se cierra si la temperatura supera los 24°C. Si la persiana está cerrada, la temperatura supera los 28°C y hubo movimiento en los últimos 10 minutos, el sistema de aire acondicionado se encenderá con la temperatura ajustada a 20°C. Además, si la temperatura es inferior a 22°C, o si no hubo movimiento en los últimos 10 minutos, el aire acondicionado se apagará. La citada figura muestra las Things como rectángulos, las *causas* como elipses y los *efectos* como rectángulos redondeados. Los efectos están dirigidos hacia la Thing donde se declararán mediante lineas discontinuas.

En la Figura 3.10 se muestra un diagrama de objetos como instancia ejemplo del modelo de relación de causalidad para el sistema de aire acondicionado `AC`. En la Figura 3.11 se muestra un fragmento de código donde se declara una relación de causalidad.

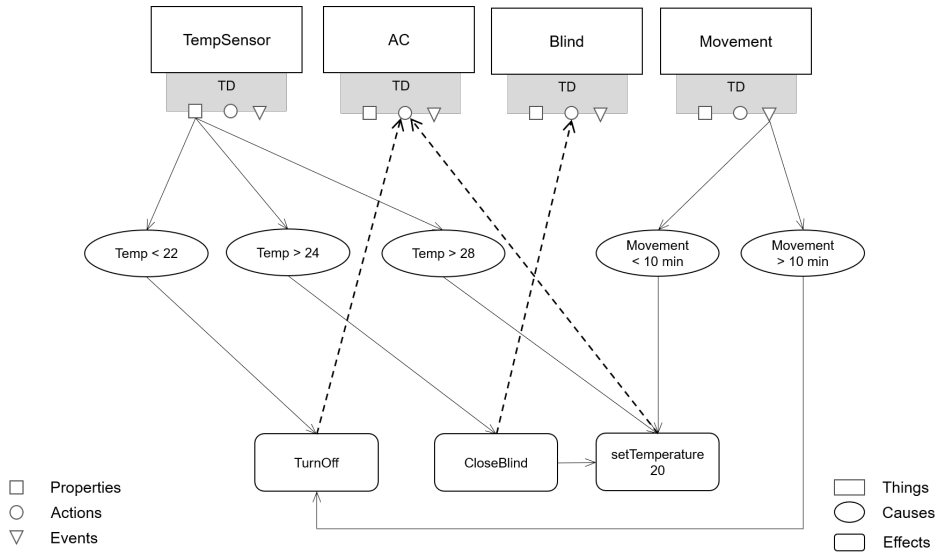


Figura 3.9: Ejemplo de relación causa/efecto de un escenario domótico.

El código de la relación de casusalidad del aire acondicionado (Figura 3.11), comienza con los siguientes parámetros (líneas 1–4): (a) un identificador de causalidad `id`, una clave generada automáticamente para guardar la relación; (b) un identificador `thingID`, que se relaciona con el `id` de la Thing Description donde tienen lugar los efectos; y (c) el atributo `isManaged` establecido como falso, para indicar que la relación de causalidad es administrada por el propio *servient* WoT, en este caso un Digital Dice.

El siguiente parámetro en el código es `self`, que apunta a un enlace donde se encuentra la Thing Description de la Thing relacionada, aquí es donde tienen lugar todos los *efectos* (líneas 5–8).

Seguidamente, en el código, se definen los *efectos*, que en el ejemplo son dos: `turnOff` y `activateAC`. El primero es el efecto `turnOff` (líneas 10–30), que tiene dos causas (líneas 12–28). Como se observa, las causas hacen referencia a varios enlaces (líneas 12–16 y 21–25) donde reside el documento Thing Description de donde proceden las causas. La primera causa proviene del sensor de temperatura `tempSensor`, con una propiedad `property` llamada `temp` que se origina en el dispositivo físico, `physicalDevice`. La segunda causa es generada por el sensor de presencia `movement` con el evento `movement`. En el efecto `turnOff` no hay una propiedad de `window`, lo cual significa que los efectos no necesitan ocurrir en una ventana de tiempo determinada, ni tampoco necesitan tener un orden. Después de eso, se define la expresión `evalExpresion`, una condición abreviada que evalúa la primera causa y la compara para ver si la temperatura es inferior a 22°C. La segunda causa verifica si hubo movimiento en los últimos 10 minutos. Si ocurre una de las dos condiciones, `processInteraction` ejecuta la acción `shutdown`; no hay segundo parámetro porque esta acción en particular, no necesita ninguna entrada.

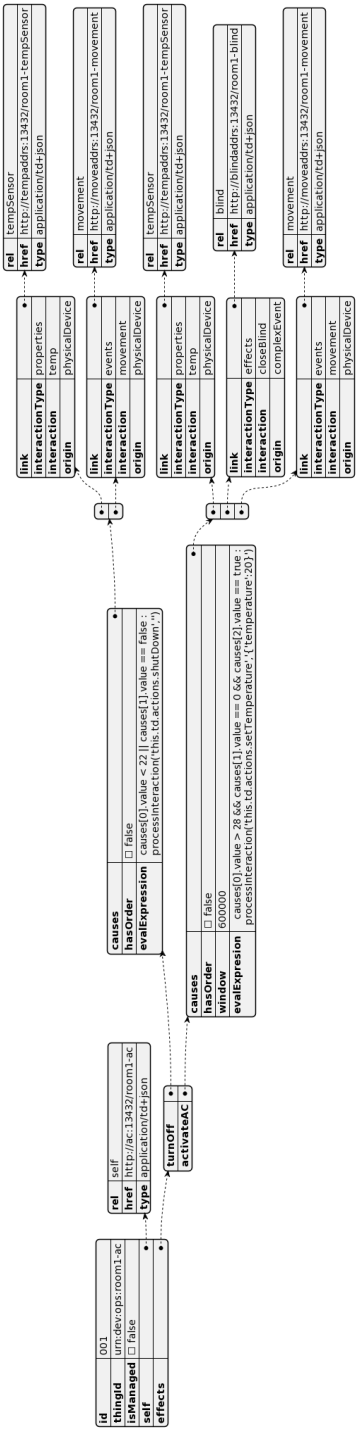


Figura 3.10: Relación de causalidad del sistema *Aire Acondicionado*.

```
{
  "id": 001,
  "thingId": "urn:dev:ops:room1-ac",
  "isManaged": false,
  "self": {
    "rel": "self",
    "href": "http://ac:13432/room1-ac",
    "type": "application/td+json" },
  "effects": {
    "turnOff": {
      "causes": [{
          "link": {
            "rel": "tempSensor",
            "href": "http://tempaddrs:13432/room1-tempSensor",
            "type": "application/td+json"
          },
          "interactionType": "properties",
          "interaction": "temp",
          "origin": "physicalDevice"
        }, {
          "link": {
            "rel": "movement",
            "href": "http://moveaddrs:13432/room1-movement",
            "type": "application/td+json"
          },
          "interactionType": "events",
          "interaction": "movement",
          "origin": "physicalDevice" }],
      "hasOrder": false,
      "evalExpression": "causes[0].value < 22 || causes[1].value == false:
       processInteraction('this.td.actions.shutDown','')" },
    "activateAC":{
      "causes": [{
          "link": {
            "rel": "tempSensor",
            "href": "http://tempaddrs:13432/room1-tempSensor",
            "type": "application/td+json"
          },
          "interactionType": "properties",
          "interaction": "temp",
          "origin": "physicalDevice"
        }, {
          "link": {
            "rel": "blind",
            "href": "http://blindaddrs:13432/room1-blind",
            "type": "application/td+json"
          },
          "interactionType": "effects",
          "interaction": "closeBlind",
          "origin": "complexEvent"
        }, {
          "link": {
            "rel": "movement",
            "href": "http://moveaddrs:13432/room1-movement",
            "type": "application/td+json"
          },
          "interactionType": "events",
          "interaction": "movement",
          "origin": "physicalDevice" }],
      "hasOrder": false,
      "window": 600000,
      "evalExpresion": "causes[0].value > 28 && causes[1].value == 0 &&
     causes[2].value == true : processInteraction('this.td.actions.
     setTemperature','{'temperature':20}')"
}}}
```

Figura 3.11: Código de la relación de causalidad del sistema *Aire Acondicionado*.

El segundo *efecto* `activateAC` (líneas 31–62) es un poco más complejo que el anterior, ya que se origina a partir de tres causas diferentes, las cuales están vinculadas a tres dispositivos diferentes, con las capacidades de interacción (*affordances*) de enlace `tempSensor`, `blind` y `movement`. Es especialmente interesante la *causa* procedente de la persiana, ya que proviene de un *efecto* declarado con origen `complexEvent` de otra Thing Description. A diferencia del primer *efecto* de la Thing Description, este tiene una propiedad de `window` declarada como 600000 (milisegundos), lo que significa que las tres causas deben ocurrir dentro de la misma ventana temporal (10 minutos). Si las tres causas ocurren en esos diez minutos, se evalúa la expresión `evalExpression`. Esta expresión procesa cada *causa* y la compara con los valores indicados en la Figura 3.9. En este caso, si se validan todas las premisas, se procede a establecer la temperatura del sistema de aire acondicionado `AC` a 20°C.

3.4. Alta Disponibilidad en Digital Dice

El aumento en el uso de dispositivos IoT [Laghari et al., 2021] o Sistemas Ciberfísicos (CPS) [Pivoto et al., 2021] en todo tipo de dominios, como ciudades inteligentes, edificios inteligentes e Industria 4.0, entre otros, ha provocado un aumento desenfrenado en la adopción de estos dispositivos. Según un informe de la empresa alemana Statista [Statista, 2023], se espera que el número de dispositivos IoT conectados en todo el mundo alcance los 43.000 millones en 2023, e incluso se llegará a los 500 billones para el año 2030, según Cisco Systems [Zikria et al., 2021]. Este número incluye dispositivos en una variedad de sectores, desde la automatización del hogar hasta la agricultura, la industria y el sector de la salud. Uno de los mayores problemas al trabajar con esta gran cantidad de dispositivos es que usan múltiples protocolos para su manejo y comunicación.

Como se ha comentado anteriormente, la usabilidad, el rendimiento y la disponibilidad son los tres principales problemas de este tipo de dispositivos [Farhan et al., 2017]. Para resolverlos, esta tesis doctoral propone el modelo Digital Dice (DD) [Mena et al., 2021b], cuyo propósito es: (a) crear una representación virtual *altamente disponible* (High-Availability System, HAS) [Piedad and Hawkins, 2001] de una Thing, y (b) ayudar a los usuarios y desarrolladores a interactuar con dichas Things utilizando protocolos web simples, abstrayendo al usuario de las tecnologías subyacentes particulares del dispositivo representado. Para ello, como se ha visto, Digital Dice está basado en la especificación Thing Description (TD) de la Web of Things (WoT) [W3C, 2022a] y se ha implementado mediante microservicios.

La especificación Thing Description de la WoT es un lenguaje de definición que permite declarar las interacciones que se pueden realizar con un dispositivo (real o virtual) y también permite establecer los metadatos de dicho dispositivo. Es similar a lo que OpenAPI es para los servicios RESTful. Usar la Thing Description permite atacar a uno de los problemas que comentamos anteriormente, la usabilidad, una propiedad importante para calidad de un producto software.

Durante el desarrollo del modelo Digital Dice se han tenido en consideración los requisitos no funcionales (RNF) establecidos en la norma ISO/IEC 25010 [Estdale and Georgiadou, 2018] relacionados con los parámetros de calidad de un producto de softwa-

re. En la Tabla 3.1 se comparan los parámetros de calidad de Digital Dice (como producto software), con el uso directo de un dispositivo IoT y la implementación referencia de un *servient* WoT (WoT Scripting API) [Mena et al., 2021b] sobre ese dispositivo.

Digital Dice es una solución arquitectónica basada en la WoT que despliega un conjunto de microservicios en un entorno distribuido y en contenedores. Este hecho permite que Digital Dice se convierta en una arquitectura altamente escalable, lo que constituye el primer paso para lograr el requisito de alta disponibilidad (HA) [Mena et al., 2021b], los cuales están descritos en el Capítulo 1.7. Además, la forma en la que se despliegan los microservicios que implementan los componentes de un Digital Dice, permite escalar la arquitectura de una manera horizontal, aumentando el número de instancias de cada microservicio según sea necesario. Los microservicios de Digital Dice son responsables de facetas (capacidades) específicas para la gestión de un dispositivo, como la comunicación con el mismo, el control de eventos, el uso de una interfaz visual, entre otros. Además, el hecho de que cada microservicio tenga su propio propósito, facilita la mantenibilidad de la arquitectura [Bucchiarone et al., 2020].

En esta sección se describirán diferentes estrategias usadas para convertir Digital Dice en una alternativa de alta disponibilidad. Además, se compararán algunas configuraciones de Digital Dice de conexión directa a un dispositivo IoT para comprobar su rendimiento. Por otro lado, para demostrar la aplicabilidad de Digital Dice en un sistema donde la alta disponibilidad es un requisito, se presentará un escenario virtual para mejorar la eficiencia energética en un escenario de Ciudad Inteligente.

3.4.1. Estrategias de Alta Disponibilidad

En esta sección se examinan las diferentes estrategias que se han utilizado para proporcionar a Digital Dice las capacidades de un sistema de alta disponibilidad y ver cómo estas estrategias ayudan a mejorar el rendimiento del sistema en general.

Estas estrategias de comunicación han sido divididas en tres tipos diferentes: (a) estrategias de comunicación con el dispositivo físico o un sistema ciberfísico particular; (b) estrategias de replicación de microservicios relacionadas con la escalabilidad de servicios (replicación y parada de microservicios); y (c) estrategias de comunicación relacionadas con la intercomunicación entre microservicios y la comunicación entre dichos microservicios y el usuario final.

3.4.1.1. Estrategias de comunicación con el dispositivo físico

En Digital Dice, el componente `Reflection` represnta el único punto de conexión con el dispositivo físico y su función principal es prevenir que exista un cuello de botella en el acceso al dispositivo. Los dispositivos IoT suelen tener limitaciones en cuanto a rendimiento, debido a que algunos dispositivos de ciertas instalaciones IoT están diseñados para priorizar el ahorro de energía, como es el caso de los microcontroladores ESP32. En otro tipo de entornos IoT, como en instalaciones basadas en KNX o Modbus, hay limitaciones en el número de conexiones que se pueden establecer. En este sentido, cuando varios usuarios solicitan datos simultáneamente, el bus puede generar un comportamiento incorrecto y denegar todas las solicitudes.

Tabla 3.1: RNF en IoT, Implementación referencia WoT y Digital Dice.

RNF	IoT	WoT	Digital Dice
Rendimiento Eficiencia	Tiempo de Respuesta Bajo: Conexión directa con el dispositivo. Rendimiento Bajo: Enteramente dependiente del dispositivo.	Tiempo de Respuesta Medio: Hay middleware de por medio. Rendimiento Medio: Entorno de ejecución WoT como factor limitante.	Tiempo de Respuesta Medio: Hay middleware de por medio. Rendimiento Alto: Gracias a la escalabilidad de los microservicios.
Compatibilidad	Baja interoperabilidad: Totalmente dependiente del software utilizado para gestionarlos.	Alta interoperabilidad: Gracias a las capacidades de orquestación de la WoT Scripting API. Media coexistencia: Agregar más dispositivos puede afectar al entorno de ejecución WoT y al tráfico de la red.	Alta interoperabilidad: Gracias al uso del subsistema de causalidad para declarar interoperabilidad entre Things. Alta coexistencia: La carga de la red es el único efecto que tiene agregar un nuevo DD en la arquitectura.
Usabilidad	Bajo nivel de aprendizaje e interoperabilidad: Debido a las diferentes tecnologías que pueden utilizar los dispositivos.	Alto nivel de aprendizaje: La TD representa las interacciones de manera legible, lo que facilita la comprensión de las interacciones proporcionadas. Interoperabilidad media: Debido a que en la forma básica de la TD se pueden usar HTTP, COAP y MQTT.	Alto nivel de aprendizaje: La TD representa las interacciones de manera legible, lo que facilita la comprensión de las interacciones proporcionadas. Alta interoperabilidad: La TD de una DD está restringida a usar HTTP estándar (API REST), lo que facilita su aprendizaje y uso.
Confiabilidad	Completamente dependiente de la implementación de hardware.	Baja disponibilidad: Además de depender del dispositivo físico, si el entorno de ejecución WoT sufre un problema, el sistema puede caerse.	Alta disponibilidad: DD implementa una serie de estrategias (relacionadas con dispositivos físicos, comunicación y microservicios) para convertirse en un artefacto de software de alta disponibilidad. Alta tolerancia a fallos: DD implementa comportamientos de seguridad incluso cuando los dispositivos físicos dejan de funcionar, y el aislamiento de DD hace que uno no afecte a otros. Alta recuperabilidad: Debido a que se guarda todo el estado de las interacciones relacionadas con un dispositivo en particular. DD puede ver tiene constancia de la última instantánea registrada de todas las propiedades aun cuando el dispositivo está desactivado.
Seguridad	Totalmente dependiente de la implementación de hardware/software.	Alta confidencialidad, integridad y autenticidad: debido a la posibilidad de declarar parámetros de seguridad/autenticación en el TD siguiendo las directrices de seguridad WoT.	Integridad media: la comunicación del DD es segura gracias al uso de SSL. Alta responsabilidad: DD registra todas las interacciones solicitadas por los usuarios con un dispositivo en particular.
Mantenibilidad	Completamente dependiente del fabricante de hardware.	Alta testabilidad: Debido a la capacidad de introducir un servient consumidor para probar las características de diferentes servient productores. Alta reusabilidad: Una vez que se declara una TD, puede ser utilizada en otras implementaciones de WoT.	Alta modularidad, reusabilidad, modificabilidad: Gracias a la arquitectura de microservicios y gracias a que se basa en el TD para sus posibles interacciones.
Portabilidad	Adaptabilidad media: Pueden haber muchas piezas de software diferentes para manejar un dispositivo IoT, pero es depende del hardware. Alta instalabilidad: Depende del hardware, pero en algunos casos puede ser plug-and-play. Reemplazabilidad media: Depende del software.	Adaptabilidad baja: Depende del hardware que representa, y en su forma básica solo es capaz de consumir las tres tecnologías mencionadas anteriormente. Instalabilidad baja: La instalación de la WoT Scripting API puede requerir un conocimiento profundo del marco de trabajo.	Adaptabilidad baja: Mejor adaptabilidad que la WoT Scripting API, debido a su capacidad para utilizar dispositivos definidos por más tecnologías como KNX, zigbee, etc., pero todavía depende del hardware que representa. Instalabilidad media: Una vez que se define un DD, se puede instalar fácilmente mediante la ejecución de sus archivos de configuración de Kubernetes. Pero crear un nuevo DD desde cero es difícil y requiere un conocimiento profundo.

El uso de un microservicio para el componente `Reflection` y la base de datos como sistema intermediario reduce el número de peticiones de lectura al dispositivo físico. Además, al tener un único punto de conexión con el dispositivo, se asegura que las operaciones de escritura se realicen en orden de llegada. El funcionamiento del microservicio `Reflection` se basa en suscribirse a las interacciones de lectura del dispositivo. Si este no permite la suscripción a las interacciones, `Reflection` se encarga de realizar solicitudes de *long-polling* cada `X` segundos, siendo `X` un tiempo configurable. Además, `Reflection` escucha las operaciones de escritura relacionadas con el dispositivo controlado. Cuando el microservicio detecta que una operación dada solicita cambios en el dispositivo, lo ejecuta. La elección de MongoDB permite que el microservicio `Reflection` no requiera estar continuamente solicitando datos a la base de datos. MongoDB permite suscribirse a consultas particulares mediante un patrón reactivo [Malhotra, 2019], haciendo que la base de datos sea responsable de notificar al suscriptor cuando hay nuevos datos que cumplen con los parámetros establecidos en la consulta.

3.4.1.2. Estrategias de replicación de microservicios

En cuanto a las estrategias de replicación de microservicios, es importante destacar que los microservicios que forman parte de Digital Dice se despliegan mediante contenedores del tipo Docker [Nickoloff and Kuenzli, 2019]. Gracias al uso de contenedores, se puede aislar las ejecuciones de los microservicios de los sistemas donde fueron desarrollados. Utilizar contenedores ofrece la gran ventaja de poder implementar nuestra solución en cualquier máquina, ya sea en máquinas locales o en infraestructuras en la nube, siempre y cuando estas sean compatibles con contenedores Docker.

Para desplegar contenedores de Docker, se pueden usar diferentes clientes, como el cliente original de Docker (docker CLI) [Jangla and Jangla, 2018], docker-compose [Ibrahim et al., 2021], Rancher [Buchanan et al., 2020] y Kubernetes [Burns et al., 2022], entre otros, cada uno con sus ventajas y desventajas. Sin embargo, debido a su compatibilidad directa con diferentes servicios en la nube y la posibilidad de instalarlos en servidores locales, Digital Dice utiliza principalmente Kubernetes.

La compatibilidad entre plataformas no es la única razón por la que se ha decidido utilizar Kubernetes. A diferencia de algunas de sus alternativas, este sistema tiene una serie de servicios preinstalados, necesarios para que Digital Dice funcione, además de que presenta una serie de ventajas:

(a) Ofrece la generación automática de subredes para aislar los microservicios de cada uno de nuestros Digital Dice.
(b) Permite la comunicación entre los diferentes contenedores utilizando el nombre del contenedor como DNS interno para hacer referencia al microservicio, evitando así la necesidad de instalar un servicio de descubrimiento.
(c) Posee una serie de herramientas para el seguimiento del rendimiento y otras métricas de cada contenedor, como Prometheus, que junto con Grafana, permite diseñar un panel de control para facilitar el seguimiento de estas métricas.

Kubernetes también permite replicar contenedores automáticamente, a diferencia de otros clientes. Para hacerlo, se establecen una serie de parámetros predefinidos: el número de CPU solicitadas, el límite de CPU, el número mínimo y máximo de réplicas que cada

microservicio puede tener, y la utilización de CPU objetivo para cada contenedor de un microservicio en particular. Una vez establecidos estos parámetros, cuando el sistema de Kubernetes detecta que cualquier contenedor está recibiendo una carga de CPU cercana al límite establecido, ejecuta automáticamente otra réplica de ese microservicio. Por el contrario, cuando un contenedor no está siendo utilizado lo suficiente, Kubernetes lo detiene. Por defecto, Kubernetes no sólo utiliza el nivel de uso de CPU para la replicación de microservicios, sino también el nivel de memoria. A partir de la experiencia de usuario adquirida durante el manejo de microservicios de Digital Dice se ha podido comprobar que el uso de la CPU, en lugar de la memoria, tiene un mejor comportamiento en la replicación de los microservicios.

Cuando se utilizan contenedores con Kubernetes, delante de cada uno de ellos hay un *servicio*, comúnmente llamado *entryPoint*, que se encarga de distribuir las solicitudes a cada una de las réplicas del microservicio correspondiente, a través de una estrategia predefinida que permite el balanceo de la carga entre réplicas. En el ejemplo del escenario que introduce en la siguiente subsección, se puede ver un archivo de configuración de implementación para Digital Dice[4]. Cada uno de los microservicios (implementaciones) está emparejado con una configuración de tipo *service*. Además, cada implementación tiene otra configuración de tipo *HorizontalPodAutoscaler* para manejar las réplicas de un microservicio particular. En el archivo de configuración de implementación de Digital Dice (ver Sección 2.5), se observan las especificaciones para cada microservicio, incluyendo la cantidad de réplicas que se deben crear, los recursos necesarios, como el límite y la solicitud de CPU, y el nivel de utilización de CPU deseado.

3.4.1.3. Estrategias de comunicación del componente digital

Al hablar de las estrategias de comunicación del componente Digital Dice, debemos dividirlas en dos puntos de vista diferentes: (a) cómo el componente digital se comunica con el usuario final; y (b) qué sucede con la comunicación entre los microservicios.

Como ya se ha mencionado, Digital Dice utiliza la especificación Thing Description de la WoT para la descripción de su funcionalidad. Esta especificación permite crear un documento Thing Description, donde se encapsulan todas las interacciones que el componente Digital Dice puede gestionar. La Thing Description es para los dispositivos físicos o virtuales lo que Open API [Swagger, 2022] es para un servicio web. Una de las premisas de Digital Dice es facilitar su uso por cualquier desarrollador web o móvil, a través de una API. Digital Dice utiliza un patrón de conexión híbrido: (a) con una API REST para peticiones de consulta síncronas; y (b) mediante gestión de eventos SSE (Server-Sent Events). Ambos tipos de peticiones son consideradas *stateless*, dado que no se almacena información sobre el estado o la actividad de un cliente entre solicitudes sucesivas. Es decir, el servidor trata cada petición de manera independiente, sin tener en cuenta las interacciones previas con el cliente. En Digital Dice, uno se puede suscribir a cualquier interacción de lectura a través de SSE, como a una propiedad del dispositivo a través de una solicitud HTTP y luego recibir los cambios que se producen en esa propiedad en tiempo real. El uso del protocolo SSE en Digital Dice presentó un problema inicial: y es que en las conexiones HTTP/1 [Fielding et al., 1999], un servicio sólo puede

[4]Archivo Kubernetes del escenario ejemplo *recolector de basura*: `https://cutt.ly/YIgtLAW`

mantener cinco conexiones abiertas con cada usuario. HTTP/2 [Stenberg, 2014] y HTTP/3 [Perna et al., 2022] resuelven este problema eliminando esa limitación. Para Digital Dice se optó finalmente por establecer la conexión con el usuario a través del protocolo HTTP/2, ya que HTTP/3, aunque es más rápido que HTTP/2 (por funcionar sobre UDP) todavía no es compatible con la mayoría de los navegadores. Además, el uso de HTTP/2 requiere que cada ruta de Digital Dice esté asegurada por un certificado SSL, lo cual conlleva la necesidad de establecer conexiones seguras cifrando el tráfico generado por Digital Dice. El uso de HTTP/2 en las comunicaciones con el usuario significa que las comunicaciones internas en cada subred de Digital Dice también deben estar aseguradas internamente, por lo que también se utiliza el protocolo HTTP/2.

Por otro lado, en la arquitectura de microservicios se pudo observar que al activar o desactivar replicas puede surgir lo que comúnmente se denomina la pesadilla de los microservicios (microservices nightmare). Esto ocurre cuando se están haciendo solicitudes a replicas de microservicios que ya no están disponibles o algunos de los contenedores están causando continuamente errores, lo que produce una sobrecarga de comunicación en las subredes sin que nos demos cuenta. Para evitar esto, Digital Dice utiliza una *malla de servicios* (service mesh), específicamente `envoy` e `istio` [Calcote and Butcher, 2019], para resolver los problemas de comunicación entre microservicios. Una *malla de servicios* es una capa integrada en la aplicación que controla cómo los microservicios comparten datos. A diferencia de otros sistemas que también pueden gestionar la comunicación, la *malla de servicios* es una capa visible que permite registrar si la interacción entre diferentes partes de la aplicación es buena o mala. Facilita la optimización de la comunicación y evita tiempos de inactividad a medida que la aplicación crece. Además de la monitorización, también se aplican una serie de patrones recomendados en arquitecturas de microservicios. Uno de ellos es el *circuit breaker*, que evita la comunicación con un método de microservicio si genera continuamente problemas, forzando una respuesta predeterminada. Además, el uso de `envoy` e `istio` permite el balanceo de carga entre las diferentes réplicas de un microservicio automáticamente y sin necesidad de implementarlo en el microservicio en sí. Además, estas herramientas permiten el uso de políticas de seguridad, control de tráfico y observabilidad.

3.4.2. Evaluación del rendimiento

Para evaluar el rendimiento de Digital Dice es necesario establecer una línea base de las características de la infraestructura donde se desplegará. Aunque Digital Dice puede desplegarse en un servidor dedicado, se decidió desplegarlo en un servicio en la nube, en este caso, Google Cloud, lo que implica una penalización en los tiempos de respuesta, pero permite tener un mayor control en la asignación de recursos para cada microservicio.

En la Tabla 3.2, se muestran los diferentes parámetros requeridos por los microservicios utilizados para medir el rendimiento de Digital Dice. Estos parámetros son el número mínimo y máximo de réplicas, el nivel de CPU solicitado medido en milicores (1000m = 1 núcleo), el nivel máximo y uso de la CPU objetivo, solicitada por el sistema para cada contenedor de microservicio. Además de los parámetros de cada microservicio, es necesario definir el número de réplicas utilizado para el despliegue de MongoDB. Para este experimento, se empleó un conjunto de réplicas con dos máquinas equipadas con 2

Servicio	ReqCPU	MaxCPU	Min Réplicas	Max Réplicas	Average Utilization
Controller	300m	600m	1	5 ‖ 7	50 %
Data Handler	300m	600m	1	5 ‖ 7	50 %
Reflection	200m	1000m	1	1	No Aplicable

Tabla 3.2: Configuración de parámetros del Horizontal-pod-autoscaler.

CPU virtuales (vCPU) y 2 GB de RAM cada una. Para estudiar la diferencia en el rendimiento (que se verá a continuación), se decidió utilizar dos configuraciones diferentes, en cuanto al número máximo permitido de réplicas. La configuración `#1` permite hasta 5 réplicas, como máximo, y la configuración `#2` hasta 7 replicas.

El dispositivo físico utilizado para medir el rendimiento de Digital Dice ha sido el de una bombilla simple. Para las pruebas, se han empleado tres modelos de bombilla: una de marca Philips, controlada por el protocolo Zigbee, otra de tipo KNX, y una tercera bombilla controlada por un servidor web, desplegado en un microcontrolador ESP32. Esta elección permite realizar una comparativa de rendimiento, independientemente de las tecnologías utilizadas. Al mismo tiempo, este dispositivo permite trabajar con los tres tipos de operaciones *básicas* de una Thing: de lectura, escritura y de suscripciones.

En la Figura 3.12, se puede ver el código en JSON de una Thing Description del componente Digital Dice que rige estos dispositivos, que es la misma en los tres casos, ya que aunque la implementación interna es diferente en el caso de `Reflection` (microservicio que establece la conexión con el dispositivo físico) para cada uno de ellos, todos tienen las mismas operaciones. Recuérdese que la Thing Description es una caja negra que el usuario utiliza al operar con nuestro Digital Dice.

```
{
"@context": "https://www.w3.org/2019/wot/td/v1",
"id": "acg:lab:light",
"title": "ACG Lab Light",
"properties": {
 "status": {
  "type": "object",
  "properties": {"value": { "type": "boolean" }},
  "forms": [{
   "href": "https://example.com/acg:lab:light/property/status/",
   "contentType": "application/json"}, {
   "href": "https://example.com/acg:lab:light/property/status/sse",
   "subprotocol": "sse", "response": {"contentType": "text/event-stream"
    }}]
}},
"actions": {
 "switch": {
  "input": {
  "type": "object",
  "properties": {"value": { "type": "boolean"}}},
  "required": [ "value" ],
  "forms": [{
   "href": "https://example.com/acg:lab:light/action/switch/",
   "contentType": "application/json" }]} }
}
```

Figura 3.12: Thing Description de Digital Dice Light.

En la Figura 3.13 se muestra el rendimiento de Digital Dice frente a protocolos IoT, para las dos configuraciones consideradas, indicadas en la figura como `Config#1` y `Config#2`. Los parámetros utilizados para ambas configuraaciones son los establecidos en la Tabla 3.2. El rendimiento del sistema se ha medido mediante la comprobación del nivel de respuestas esperadas junto con los errores que se producen conforme aumenta el número de solicitudes al sistema. En la Figura 3.13 de rendimiento, el parámetro *errores* hace referencia a aquellas solicitudes cuya respuestas forman parte de los códigos de estado de error del cliente de la serie 400, como las respuestas *404 No encontrado* o *408 Tiempo de espera de la solicitud superado*. Para realizar la medición, se utilizaron dos plataformas diferentes para la generación sintética de solicitudes en el sistema, Locust [Locust, 2023] y JMeter [Apache, 2023].

Es importante tener en cuenta que los resultados obtenidos en la Figura 3.13 son específicos para la infraestructura y configuraciones utilizadas. Por tanto, estos resultados no son generalizables a otros sistemas y configuraciones, pero dan una idea bastante clara de cómo evoluciona el rendimiento de Digital Dice al cambiar el número de replicas.

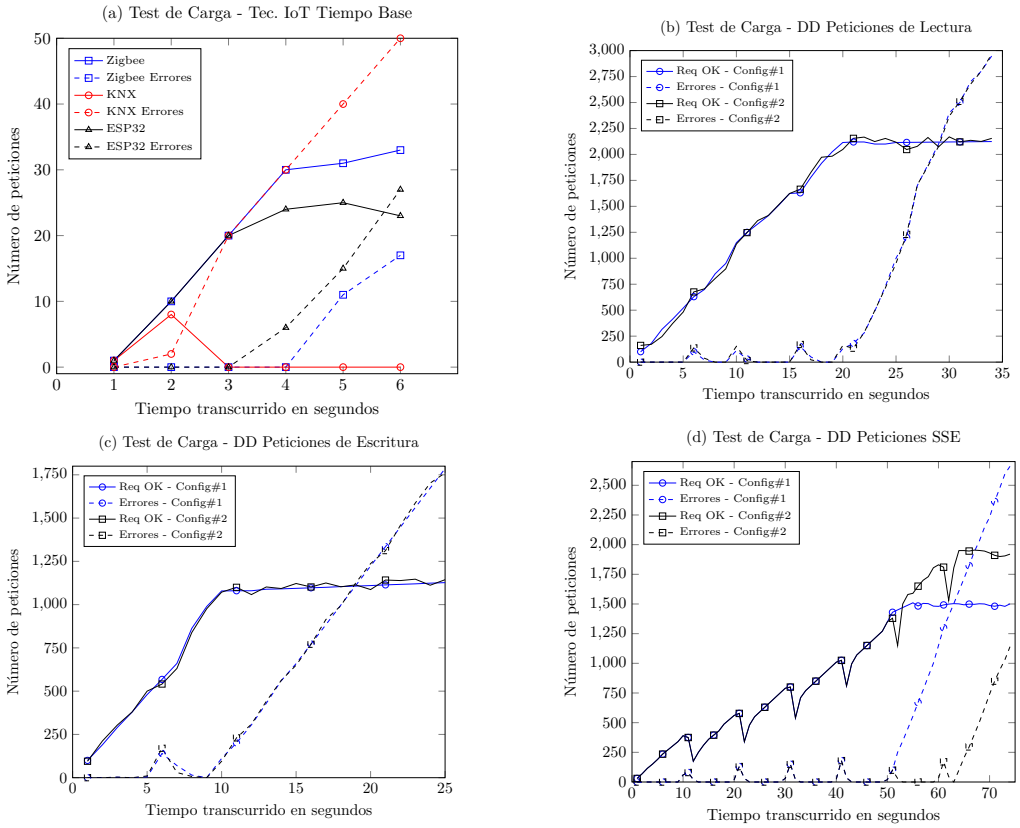


Figura 3.13: Gráficas de rendimiento: Protocolos IoT frente a Digital Dice.

Por último, es importante destacar que el rendimiento de un sistema de microservicios como el que implementa Digital Dice depende de muchos factores, como el tamaño y la complejidad de la infraestructura, el número de microservicios, la carga de trabajo y la configuración utilizada. Por lo tanto, es importante realizar pruebas periódicas del sistema y ajustar la configuración de la arquitectura de microservicios de Digital Dice según sea necesario, para garantizar un rendimiento óptimo.

En la gráfica de la Figura 3.13(a), podemos observar que los errores aparecen a partir de un número relativamente bajo de solicitudes por segundo, en los diferentes dispositivos físicos utilizados como referencia. Curiosamente, en las conexiones basadas en bus, como KNX, se puede observar que después de ocho operaciones de lectura simultáneas, comienzan a aparecer tiempos de espera importantes, ya que el búfer de datos de este sistema comienza a funcionar de forma incorrecta.

Por el contrario, en la gráfica de la Figura 3.13(b), se observa cómo responde Digital Dice frente a las operaciones de lectura. A medida que aumenta el número de solicitudes, el sistema responde mucho mejor que cualquier dispositivo de referencia mostrado en la gráfica, ya que no es necesario establecer una comunicación directa con los dispositivos físicos. Sin embargo, en la gráfica se puede ver cómo aparecen algunos errores cada (aproximadamente) 500 usuarios realizando peticiones a Digital Dice. Los errores surgen cuando se está iniciando una nueva réplica, lo que hace que las solicitudes se dirijan a ese nuevo contenedor antes de que termine de iniciarse. En este punto, el número de errores parece estabilizarse de nuevo. Es importante destacar que, aunque se haya definido previamente que el sistema puede funcionar hasta con cinco réplicas (en la configuración `#1`), en la gráfica, no aparece el quinto pico en el número de errores. Sin embargo, a partir de cierto momento, el sistema alcanza el límite para el número de solicitudes que puede manejar. Este punto se alcanza cuando se llega a la cantidad máxima de solicitudes que puede manejar la base de datos simultáneamente. En ese momento, la base de datos se convierte en el factor limitante. Después de aumentar los recursos de la base de datos, se observa que el número de solicitudes aumentó según lo esperado. Además, si se aumenta el número de réplicas de los microservicios y el número de réplicas de la base de datos en la implementación a más de cinco, se alcanza otro factor limitante: el ancho de banda desde donde se lanzan las solicitudes. Seguramente, si se usara una red de nodos para lanzar las solicitudes de manera paralela, el sistema seguiría escalando las solicitudes según lo esperado, pero debido al costo que esto implicaría, aunqu esto no ha sido probado en el trabajo. Lo mismo ocurre para la configuración `#2` anteriormente establecida, ya que Digital Dice no es el factor limitante.

La gráfica de la Figura 3.13(c) representa las solicitudes de escritura que el sistema puede manejar. Nuevamente, el comportamiento es similar al de las operaciones de lectura, pero el funcionamiento incorrecto ocurre mucho antes en las dos configuraciones. El colapso ocurre porque el componente de microservicio `Reflection` de Digital Dice establece una cola con las solicitudes de escritura que llegan a la base de datos, solicitudes que se enviarán secuencialmente al dispositivo físico. Este comportamiento ocurre cuando el número de solicitudes que llegan a la base de datos supera el número de solicitudes que el dispositivo físico puede aceptar, y por tanto, la cola en `Reflection` se satura. Por ello, de la gráfica se puede deducir que esto presenta una limitación de las operaciones de escritura en el dispositivo físico.

Finalmente, el resultado mostrado en la gráfica de la Figura 3.13(d) permite ver el comportamiento de los eventos enviados por el servidor (SSE). Este tipo de solicitud es bastante difícil de probar ya que prácticamente ninguna de las herramientas de prueba de carga puede trabajar con estas peticiones. Para probar SSE, se han generado *scripts* de prueba personalizados utilizando un muestreador de Java. Una vez lanzado, se observa que el comportamiento difiere de las operaciones de lectura habituales. En primer lugar, se pueden procesar menos solicitudes al mismo tiempo. Además, los picos de error, aunque parecen estar sincronizados con la replicación de un servicio, son más pronunciados y parecen afectar al número de solicitudes procesadas por el sistema. También, utilizando un valor de 5 como factor de replicación máximo, el sistema no encuentra limitaciones en la base de datos, a diferencia de las operaciones de lectura. En la configuración `#2`, cuando el factor de replicación para los microservicios se establece en 7, ya comienzan a aparecer los mismos factores limitantes que con las operaciones de lectura.

El comportamiento de la replicación se vuelve errático con SSE al utilizar el límite de CPU como métrica de replicación, debido a que los contenedores están casi inactivos, hasta que se activa un mensaje. En la actualidad, se está estudiando cómo establecer otras métricas personalizadas de replicación, como el número de peticiones por segundo que recibe el sistema. Con esta métrica, los microservicios anticiparán la necesidad de replicación, lo que hará del sistema que sea más resiliente.

Aumentar el número de réplicas o dar más recursos a la base de datos ciertamente mejora el rendimiento de las diferentes operaciones en Digital Dice. Sin embargo, debido al costo de la operación de utilizar más réplicas, se decidió que las configuraciones utilizadas eran lo suficientemente buenas para comprobar el comportamiento de Digital Dice para todos los tipos de operaciones diferentes.

3.4.3. Escenario de Experimentación

En este escenario, se persigue demostrar la viabilidad de Digital Dice en un entorno de ciudad inteligente. Para ello, se desarrolló un sistema virtualizado basado en un proceso muy importante para una ciudad, el proceso de recolección de basura. Se eligió este escenario por estar alineado con las fortalezas de Digital Dice, al ser un escenario en el que prima la escalabilidad, la usabilidad y interoperabilidad entre diferentes dispositivos que intervendrían en el hipotético escenario de recogida de basura. En este proceso, tanto los ciudadanos y los propios trabajadores del sistema de recogida, actúan como *stakeholders*, por lo que el hecho de disponer de una alternativa de alta disponibilidad podría ser de gran utilidad y de interés para ellos, ya que se les puede proporcionar información en tiempo real sobre el estado de los contenedores y de las rutas que deben seguir los camiones de basura. Además, como se trata de un ejemplo de sistema virtual, se puede considerar Digital Dice como una implementación Digital Twins.

El escenario consta de tres Things para controlar: contenedores, camiones y rutas. Las posibles interacciones disponibles en cada una de las Things se definen en la Tabla 3.3. Hay que destacar que cada una de las Thing Descriptions del escenario, no representan una entidad individual. Esto significa que, por ejemplo, en los contenedores, la Thing Description no representa sólo un contenedor individual; representa un conjunto completo. Una de las ventajas significativas de WoT es su flexibilidad al definir el con-

Interacción	Nombre	Descripción
Containers		
Propiedades	- containerDetails - containersDetails	- Detalle de un contenedor. - Detalle de todos los contenedores.
Acciones	- emptyContainer - throwGarbage - changeTemperature - moveContainer	- Vaciar un contenedor. - Tirar basura en contenedor. - Cambiar temperatura de contenedor. - Mover posición del contenedor.
Eventos	- fire - garbageCollected	- Se incendió el contenedor. - La basura ha sido recogida.
Trucks		
Propiedades	- truckDetails - fleetDetails	- Detalle de un camión. - Detalle de la flota.
Acciones	- collectGarbage - moveTruck - consumeFuel - startRoute - stopRoute - refillFuel - emptyTruck - changeActualRoute - finishRoute	- Carga una cantidad de basura. - Mueve la posición. - Consume cierta cantidad de gasolina. - Comienza una nueva ruta. - Para la ruta activa. - Rellena la gasolina. - Vacía la carga de basura. - Cambia la ruta actual. - Finaliza ruta actual.
Eventos	- noFuel - nextRoute - routeFinished	- Se agota la gasolina. - Una nueva ruta ha sido asignada. - Se termino la ruta activa.
Routes		
Propiedades	- routeDetails - routesDetails - activeRoutesDetails	- Detalle de la ruta. - Detalle todas las rutas. - Detalle todas las rutas activas.
Acciones	- generateRoute - addContainer - stopRoute - finishRoute	- Genera ruta para camión. - Añade contenedor a ruta. - Para ruta. - Finaliza ruta.
Eventos	- routeStarted - routeStopped - routeModified - routeFinished	- Una ruta ha comenzado. - Una ruta ha parado. - Una ruta ha sido modificada. - Una ruta ha terminado.

Tabla 3.3: Escenario de recolección de basura - Interacciones.

cepto de Thing. Conceptualmente, la Thing expuesta en el ejemplo podría considerarse como una Thing agregada en términos WoT, es decir, una Thing de Things. El repositorio de código público está en disponible en línea[5] y proporciona más detalles sobre las diferentes interacciones, así como los parámetros de entrada y salida requeridos. Este repositorio contiene todas las descripciones de las *Things* del escenario, el código fuente de los diferentes microservicios y los archivos de configuración, tanto de Docker como de Kubernetes, para replicar el escenario cuando sea necesario.

Este escenario tiene como objetivo optimizar las rutas de los camiones en tiempo real conforme los contenedores se van llenando. Para simular este comportamiento, es necesario declarar una serie de relaciones de causalidad representadas en la Figura 3.14, cuyas definiciones se encuentran en los componentes `Event Handler` de cada Thing que interviene en el escenario. La decisión de incluir las relaciones de causalidad directamente en dicho componente es para evitar la necesidad de tener que levantar el servicio externo de relaciones de causalidad, y de esa manera poder ofrecer el escenario de manera más contenida, dado que este escenario no requiere funcionar de manera centralizada y son los propios Digital Dice los que interpretan las relaciones de causalidad. Estas relaciones

[5]Garbage collection repo. - `https://github.com/acgtic211/garbage-iiot-dd-public`

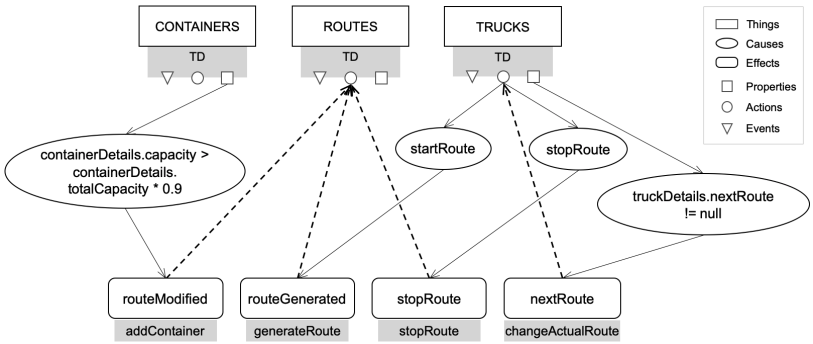


Figura 3.14: Escenario de recolección de basura - Relaciones de causalidad.

conducen a la ejecución de una serie de acciones o modificaciones de una determinada propiedad en sus Things. Por ejemplo, en la Figura 3.14 se puede ver que, cuando un contenedor supera el 90 % de su capacidad (causa), la regla trata de agregarlo a la ruta a través de la acción `addContainer` (efecto), que buscará internamente la ruta óptima para agregar el contenedor, priorizando las rutas activas que no excedan la capacidad de combustible ni la capacidad de carga del camión que está trabajando en ella.

En este escenario, los servicios de virtualización se encargan de imitar el comportamiento de los contenedores y los camiones. Las rutas que hacen estos últimos se establecen a partir de rutas calculadas y teniendo en consideración los contenedores a recoger. La ruta óptima se calcula con Google Directions API. A menos que se indique manualmente, los contenedores pueden ejecutar un comportamiento de virtualización que llena cada contenedor cada x segundos, con un peso semialeatorio, entre un valor máximo y mínimo de peso, dependiendo del contenedor tratado.

En el laboratorio web WoT-Lab[6] del grupo de investigación ACG/UAL (al que pertenezco) hay disponible una interfaz gráfica para la gestión y monitoreo de este escenario. Además, la herramienta WoT-Lab también ofrece un escenario relacionado con hogares inteligentes y permite el acceso a Thing Descriptions que representan dispositivos físicos, virtuales y Digital Dice, estando disponibles a través de un servicio de descubrimiento público para experimentar con esta tecnología. En el siguiente capítulo se ofrecen más detalles de la implementación y de la interfaz gráfica del escenario ejemplo.

En el contexto de una ciudad inteligente, este escenario ilustra cómo Digital Dice puede ser utilizado para mejorar la eficiencia y la calidad de vida de los ciudadanos. Al proporcionar información en tiempo real sobre el estado de los contenedores y las rutas de los camiones de basura, se pueden reducir los tiempos de espera y los tiempos de recolección, mejorando así la higiene y la seguridad en la ciudad. Además, al virtualizar el comportamiento de los contenedores y los camiones, se pueden evitar errores humanos y minimizar el impacto ambiental al optimizar las rutas de recolección. Estos beneficios demuestran el potencial de Digital Dice como herramienta para mejorar la gestión de servicios públicos y fomentar la creación de ciudades inteligentes y sostenibles.

[6]ACG WoT-Lab – `https://acg.ual.es/projects/cosmart/wot-lab/`

Capítulo 4

Implementación de Digital Dice

Capítulo 4

Implementación de Digital Dice

Contenido

4.1. Introducción

En este capítulo se describen detalles de la construcción de cada uno de los microservicios que componen Digital Dice, tanto a nivel de implementación como de despliegue. Para ello, se parte de un ejemplo de implementación de un Digital Dice basado en un escenario de uso real en una SmartHome[1] que utiliza el protocolo KNX como tecnología central. No obstante, la implementación de Digital Dice es independiente del protocolo de comunicación utilizado, por lo que cualquier otro protocolo de comunicación puede ser utilizado en su lugar. El ejemplo se enfoca en la descripción de los aspectos relevantes de la implementación de Digital Dice y no necesariamente en lo que sería el escenario completo de una casa inteligente. El escenario utilizado esta encapsulado en un banco de pruebas establecido en el maletín que se muestra en la Figura 4.1. Este escenario, cuenta con un detector de movimiento, dos bombillas sin regulador de intensidad, dos luces con regulador de intensidad, y dos sistemas visuales de aviso de fuego e inundaciones. Todos los dispositivos están controlados por un Digital Dice que permite interactuar con todos los dispositivos a través de tecnologías web.

Figura 4.1: Laboratorio físico de pruebas que simula una instalación de casa inteligente basada en el protocolo KNX.

[1]SmartHome-DD-project - `https://github.com/acgtic211/SmartHome-DD-project`

En primer lugar, es necesario establecer una Thing Description que describa las posibles interacciones que el Digital Dice va a controlar, así como las rutas con las que se realiza la comunicación. La Figura 4.2 establece un pequeño fragmento del código en formato JSON de esa Thing Description. El código completo del escenario SmartHome[2] se encuentra disponible en el portal web Digital Dice. En la figura se definen dos propiedades, `status-light1` (líneas 15–41) y `status-light2` (líneas 42–46), que representan los dos luces de la SmartHome declarado como `acg:lab:suitcase-dd` (línea 3).

```
1  {
2    "@context": "https://www.w3.org/2019/wot/td/v1",
3    "id": "acg:lab:suitcase-dd",
4    "title": "ACG Lab Digital Dice Suitcase",
5    "base":"https://acg.ual.es/things",
6    "securityDefinitions": {
7      "basic_sc": {
8        "scheme": "nosec"
9      }
10   },
11   "security": [
12     "nosec"
13   ],
14   "properties": {
15     "status-light1": {
16       "type": "object",
17       "properties": {
18         "value": {
19           "type": "boolean",
20         }
21       },
22       "required": [
23         "value",
24       ],
25       "forms": [{
26         "op": [
27           "readproperty",
28           "writeproperty"
29         ],
30         "href": "#/acg:lab:suitcase-dd/property/status-light1/",
31         "contentType": "application/json"
32       },
33       {
34         "op": [
35           "readproperty",
36         ],
37         "href": "#/acg:lab:suitcase-dd/property/status-light1/sse",
38         "subprotocol": "sse",
39         "response": { "contentType": "text/event-stream" }
40       }]
41     },
42     "status-light2": {
43       "type": "object",
44       "properties": {
45         "value": {
46     ...
47
```

Figura 4.2: Fragmento de la Thing Descripion de la SmartHome.

[2]Portal Digital Dice TD-SmatHome: `https://acg.ual.es/projects/cosmart/digitaldice`

Cada una de estas propiedades tiene un valor booleano que indica si la luz está encendida o apagada (línea 19). Además, ambas propiedades tienen dos formas de acceso, una de lectura y escritura, a tráves de una dirección HTTP (líneas 26–31) y otra de solo lectura, a través de un protocolo de comunicación SSE (líneas 34–39). Como se puede observar, esta Thing Description representa un conjunto de luces de la casa inteligente. El fragmento de código de la Figura 4.2 es solo un ejemplo parcial, ya que en una instalación real, la Thing Description sería mucho más compleja y tendría que incluir más propiedades y formas de acceso.

Es importante partir de las interacciones que ofrece la Thing Description para poder implementar el Digital Dice dado que, como se describe en las siguientes secciones, muchos de lo comportamientos de los microservicios se basan en parametrizaciones de estas interacciones. Por ejemplo, las rutas generadas en Digital Dice de lectura y escritura de las propiedades de la Thing Description que se está manejando vienen dadas por las formas de acceso definidas en la Thing Description. Por lo tanto, si en la Thing Description se definen formas de acceso a propiedades que no incluyen operaciones de escritura, Digital Dice no generará rutas escritura para esas propiedades.

Según lo indicado en el Capítulo 2, los distintos componentes o microservicios que forman parte de un Digital Dice vienen dados por el documento de la Thing Description

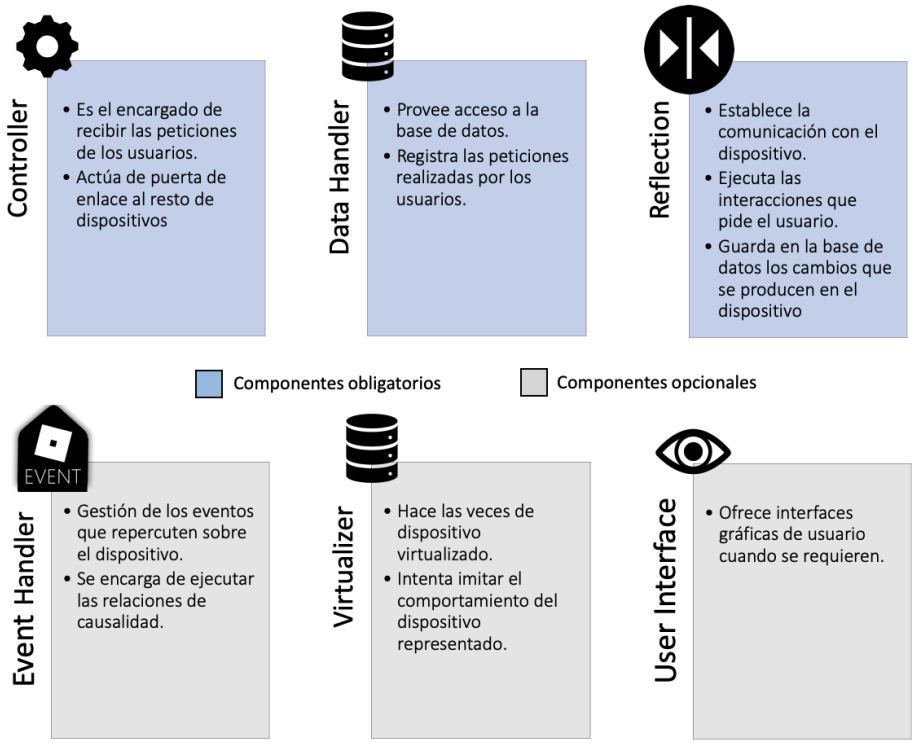


Figura 4.3: Componentes que forman parte de Digital Dice.

que lo define. En la Figura 4.3, se resumen los componentes que pueden formar parte de un Digital Dice, diferenciados entre obligatorios y opcionales, aspecto que dependerá de la Thing Description construida.

El resto de este capítulo se subdivide en cinco secciones, que profundizan en aspectos de implementación y despliegue de cada uno de los componentes que pueden formar parte de un Digital Dice. En la Sección 4.2 se describe el componente `Controller`, que es el encargado de la gestión de las peticiones por parte del usuario, ejerciendo de puerta de enlace con el resto de componentes de Digital Dice. En la Sección 4.3 se describe el componente `Data Handler`, cuya función principal consiste en proveer acceso a los datos requeridos por los usuarios y, al mismo tiempo, almacenar las solicitudes que implican un cambio de estado en el dispositivo por parte de los usuarios. En la Sección 4.4 se describe el componente `Reflection`, el cual se encarga de establecer la comunicación con el dispositivo físico para, por un lado, reflejar las peticiones que modifican el estado en el dispositivo y, por otro lado, reflejar en la base de datos cambios de estado que ocurren en el dispositivo. La Sección 4.5 describe el componente `Event Handler`, cuyo objetivo es encargarse de la gestión de eventos. Por último, en la Sección 4.6 se estudian los últimos dos componentes definidos en nuestra solución: (a) el componente `Virtualizer`, que se encarga de establecer el comportamiento de virtualización siempre y cuando sea necesario; (b) el componente `User Interface (UI)`, que se encarga de ofrecer interfaces de usuario gráficas cuando se requieran.

4.2. El componente Controller

Este componente de Digital Dice actúa como la interfaz principal para los usuarios que quieren interactuar con el sistema. Además, es el responsable de enrutar las solicitudes de los usuarios a los diferentes microservicios que componen Digital Dice. El `Controller` actúa como la puerta de enlace al sistema y es el primer punto de contacto para los usuarios, lo que lo convierte en un componente crítico para el correcto funcionamiento del sistema en su conjunto. Si tenemos en cuenta la Thing Description del listado de la Figura 4.2, las `href` de cada uno de las formas de acceso a las interacciones de la Thing apuntan directamente a este componente, es decir, cualquier interacción que queremos hacer en el dispositivo (p.ej., cambiar `status-light1`) tiene que pasar necesariamente por este microservicio.

En las Subsecciones 4.2.1 y 4.2.2 se describen detalles sobre la implementación y el despliegue de `Controller` respectivamente.

4.2.1. Consideraciones de implementación

El `Controller` de Digital Dice es una implementación de un microservicio en Node.js [Cantelon et al., 2014] utilizando el framework Express [Brown, 2019]. A la hora de implementar este microservicio, era necesario establecer un servidor web; para ello utilizamos la librería del framework Express. Junto con Express, utilizamos otras librerías a modo de plugins, como *morgan* [Wilson, Doug, 2023] para el manejo de logs, *dotenv* [Motte, Scott and jclbw, 2023] para cargar variables de entorno, *cors* [Wilson, Doug

and Goode, Troy, 2023] para permitir solicitudes desde otros orígenes y *promBundle* [Schweizer, Jochen, 2023] para crear métricas de Prometheus [Turnbull, 2018], una herramienta para monitorizar el estado del microservicio y realizar su correcto despliegue en Kubernetes. Prometheus es una de las API a las que Kubernetes puede tener acceso para para decidir si un microservicio requiere replicarse o no.

Este microservicio presenta el siguiente comportamiento. En primer lugar, carga la Thing Description e intenta validar su estructura para comprobar si está bien definida. Una vez validada, se establecen una serie de rutas parametrizadas dependiendo de las interacciones que puede ejecutar la Thing o el conjunto de Things que están siendo representadas. Por ejemplo, si la Thing Description tiene sólo una propiedad que es de tipo lectura (`readproperty`) se generan dos rutas para su acceso: una ruta para devolver el estado de esa propiedad en un momento dado, y otra que devuelve un *text/event-stream*; es decir, una serie de líneas de texto que se reciben mediante eventos en tiempo real con los cambios que se vayan realizando en esa propiedad. Esta última ruta permite realizar una subscripción a esa propiedad mediante el protocolo SSE. Este comportamiento de establecer dos rutas por cada propiedad de lectura es el que se establece por defecto en Digital Dice. No obstante, este comportamiento puede ser modificado a través de la configuración del microservicio. Por ejemplo, si se desea que el microservicio solamente establezca una ruta para cada propiedad de lectura, se debe establecer una única forma (`form` de la TD) de acceso a dicha propiedad. El disponer de las dos rutas nos permite ampliar las posibilidades de interacción con el sistema y mejorar las capacidades de la Thing, ya que si un usuario quiere acceder a una propiedad de lectura de forma simple, puede hacerlo a través de la ruta simple, pero si quiere acceder a la propiedad en tiempo real, puede hacerlo a través de la ruta de suscripción. Por otro lado, si la Thing Description tuviese una propiedad de escritura, se establecería una tercera ruta para modificar el valor de esa propiedad. En el caso de que la Thing Description tuviese una acción, se establecería una ruta para ejecutar esa acción de forma muy similar a las propiedades de escritura. En el caso de que la Thing Description tuviese un evento, se establecería una ruta para suscribirse a ese evento tal y como hacemos con las propiedades de lectura. En el listado de código de la Figura 4.4 se muestran dos rutas de ejemplo.

Las dos rutas expuestas en la Figura 4.4 permiten obtener y actualizar propiedades declaradas en una Thing Description a través de una solicitud HTTP GET y una solicitud HTTP POST, respectivamente. El componente `Controller` almacena la Thing Description en la variable `td`. Lo primero que se observa es que cualquiera que sea la ruta que se solicite, la `td.id` (línea 1 y 9) debe coincidir con la que se ha definido en la TD dado que, de no ser así, no se procesarían las peticiones, ya que la ruta no existiría.

La ruta GET usa la sintaxis `app.get()` para definir la ruta `/td.id/property/:propertyName`, donde `:propertyName` es el nombre de la propiedad a la que se accede. Si la propiedad no existe en la TD, se devuelve una respuesta HTTP 404 con un mensaje de error. En caso contrario, se realiza una solicitud HTTP GET a la dirección `https://dd-suitcase-dh-entrypoint:8063/+td.id+/property/:propertyName`, que corresponde con la dirección del `Data Handler`, componente que se encarga de la comunicación con la base de datos, y se envía la respuesta de la solicitud como una respuesta HTTP al cliente a través de la función `pipe()`, que establece el `Controller` como proxy de la petición y respuesta del `Data Handler`. Gracias a ello se consigue aislar el `Data`

```
1  app.get('/'+td.id+'/property/:propertyName', async (req, res) => {
2
3      if(!td.properties[req.params.propertyName]) res.status(404).send("That
           property doesn't exist");
4
5      request("https://dd-suitcase-dh-entrypoint:8063"+"/"+td.id+"/property/"+
           req.params.propertyName).pipe(res);
6
7  })
8
9  app.post('/'+td.id+'/property/:propertyName', async (req, res) => {
10     if(!td.properties[req.params.propertyName]) res.status(404).send("That
           property doesn't exist");
11
12     td.properties[req.params.propertyName].required.forEach(element => {
13         if(req.body[element] == undefined) res.status(404).send("Some of the
               necessary properties are not in the request - "+td.properties[req
               .params.propertyName].required);
14     });
15
16     res.writeHead(200, headers);
17     request.post("https://dd-suitcase-dh-entrypoint:8063"+"/"+td.id+"/property
           /"+req.params.propertyName, {"json":req.body}).pipe(res)
18
19 })
```

Figura 4.4: Propiedades de lectura y escritura en el `Controller`.

`Handler` del exterior de la subred donde residen todos los microservicios de el correspondiente Digital Dice, y de esa manera evitar en primera instancia la introducción de código malicioso en el mismo.

Algo muy similar ocurre en la segunda ruta, la ruta POST, con la particularidad de que se comprueba que se proporcionen todos los valores requeridos para actualizar dicha propiedad. Si faltan valores, se devuelve una respuesta HTTP 404 con un mensaje de error que indica los valores faltantes. De manera similar a estas dos rutas, se establece el funcionamiento de las rutas para las acciones y los eventos, en cuyo caso se establece una ruta siguiendo el patrón `/+td.id+/action/:actionName` en el caso de las acciones y `/+td.id+/event/:eventName` en el caso de los eventos.

4.2.2. Consideraciones de despliegue

Una vez analizado el código del componente `Controller`, se verán algunas consideraciones para su despliegue en el clúster de Kubernetes. El primer paso ha sido la creación del archivo de configuración de Docker para establecer la imagen que permita desplegar el microservicio en el clúster. Para ello, se ha creado un archivo `dockerfile`, mostrado en la Figura 4.5, que contiene la definición de la imagen que se va a utilizar para desplegar el microservicio. En este, se define la imagen desde que se parte (línea 1), en este caso `node:14`, que ya viene con el intérprete de Node.js instalado, se declara un directorio donde se va a trabajar (línea 3), en el que se copian los archivos del proyecto (línea 4), se instalan las dependencias necesarias para ejecutar el proyecto (línea 5), se expone el puerto 443 (línea 6), puerto por el que se establece la comunicación con el servidor, y se ejecuta el comando para iniciar el servidor dentro del contenedor (línea 7).

```
1  FROM node:14
2
3  WORKDIR /usr/src/app
4  COPY . .
5  RUN npm install
6  EXPOSE 443
7  CMD [ "node", "src/index.js" ]
```

Figura 4.5: Archivo de configuración docker del componente Controller.

Tras construir la imagen con el comando *dockerbuild*, el microservicio `Controller` podrá ser desplegado por Kubernetes. Para ello, se ha creado un archivo YAML, mostrado en listado de la Figura 4.6, con las tres configuraciones necesarias para el correcto despliegue del microservicio. Las configuraciones en el archivo YAML se declaran con la propiedad `kind` y son las siguientes:

(a) El **Deployment** (líneas 1–27). Configuración que especifica los parámetros principales del microservicio. El primer parámetro está relacionado con los metadatos para el despliegue (líneas 3–5). En segundo lugar, el número de réplicas con las que se inicia el microservicio (línea 7). En tercer lugar, las anotaciones necesarias para que Prometheus (el sistema de monitorización usado) pueda extraer las métricas necesarias con las que monitorizar los contenedores, `pods` en Kubernetes (líneas 13–18). Por último, la sección `spec` (líneas 20–26), en las que se declara la imagen a partir de la cual se van a generar los `pods` (línea 22). En ella también se especifica el límite de uso de la CPU de cada `pod` (línea 26), para evitar que estos consuman más recursos de los necesarios.

(b) El **Service** (líneas 28–42). Esta configuración se encarga de generar un punto de entrada con el cual establecer acceso a los `pods`. En este caso, se ha definido un servicio de tipo `NodePort` (línea 34), el cual establece un puerto externo el 30011 (línea 41) que redirige todo el tráfico TCP (línea 38) al puerto 443 los contenedores.

(c) Por último, la configuración **HorizontalPodAutoscaler (HPA)** (líneas 44–61) se encarga de ajustar automáticamente el número de réplicas de la aplicación en función de la demanda. En este caso, se ha especificado que se quiere tener entre 1 y 5 réplicas (líneas 53–54), y que se utilice la CPU como métrica para escalar (líneas 57–61). En concreto, se quiere mantener una utilización media de CPU del 50 %. Si la utilización de CPU supera este valor, la configuración **HPA** creará nuevas réplicas para manejar el tráfico adicional. Si la utilización de CPU baja por debajo de este valor, la configuración **HPA** eliminará algunas de las réplicas para ahorrar recursos. Las características de las métricas de los `pods` son cambiadas dependiendo de la demanda prevista a la hora de establecer tanto el número de réplicas como el límite de CPU y memoria de cada `pod`.

En el escenario de la SmartHome, introducido en la Sección 4.1, cuya configuración de despliegue se muestra en el código de la Figura 4.6, los recursos reservados para el despliegue son bajos porque no se espera un uso intenso de los dispositivos que forman parte del escenario. Por ello, se estableció un límite de CPU mínimo.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dd-suitcase-controller
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dd-suitcase-controller
  template:
    metadata:
      annotations:
        prometheus.io/scrapehttps: 'true'
        prometheus.io/path: /metrics
        prometheus.io/port: '443'
      labels:
        app: dd-suitcase-controller
    spec:
      containers:
      - name: dd-suitcase-controller
        image: 150.214.150.155:5000/dd-suitcase-controller:latest
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            cpu: 100m
---
apiVersion: v1
kind: Service
metadata:
  name: dd-suitcase-controller-entrypoint
  namespace: default
spec:
  type: NodePort
  selector:
    app: dd-suitcase-controller
  ports:
  - protocol: TCP
    port: 443
    targetPort: 443
    nodePort: 30011
    name: https
---
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: dd-suitcase-controller
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: dd-suitcase-controller
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Figura 4.6: Configuración de despliegue del microservicio `Controller`.

4.3. El componente Data Handler

El componente Data Handler se encarga de gestionar la comunicación con la base de datos. Su función principal es la de facilitar el acceso a los datos solicitados por los usuarios, al mismo tiempo que almacena las solicitudes que implican un cambio de estado en alguna interacción del dispositivo.

Este microservicio se encarga de la recuperación de los datos de una manera eficiente y efectiva, asegurando que los usuarios tengan acceso a la información que necesitan, de manera rápida y sencilla. Además, también se encarga de registrar cualquier interacción que se realice en el sistema, lo que permite tener un control completo sobre todas las peticiones y cambios de estado en el sistema.

En el caso de los dispositivos *actuadores*, como por ejemplo, las bombillas definidas para el escenario SmartHome presentado anteriormente, el microservicio registra no sólo los cambios de estado del actuador, sino también el origen de quién realizó la interacción en cuestión (`user`, `device`, `virtualDevice` o `event`), mediante un modelo de datos de interacción. De esta manera, se puede tener un registro completo de las acciones realizadas en el dispositivo, lo cual ayuda a identificar cualquier problema o malfuncionamiento que pueda ocurrir en el sistema.

En el caso de los dispositivos *sensores* (p.ej., un sensor de temperatura o un sensor de humedad), el microservicio registra cualquier cambio en el estado del sensor, así como quién lo consultó y en qué momento, lo que nos permite analizar los datos y tomar decisiones informadas sobre el funcionamiento del dispositivo. Además, también se puede recuperar un historial completo de todos los cambios de estado del dispositivo, lo que es útil para identificar patrones y extraer información valiosa.

4.3.1. Consideraciones de implementación

Al igual que el componente `Controller`, `Data Handler` se ha implementado utilizando el framework Express. De hecho, su código es muy parecido, ya que ambos microservicios se encargan de exponer una API REST pero con funcionalidades y propósitos diferentes. El objetivo del `Data Handler` es almacenar y recuperar datos de la base de datos proporcionados o requeridos por los usuarios, mientras que el `Controller` se encarga de gestionar las interacciones con el usuario. Por este motivo, el `Data Handler` tiene que tener en cuenta la lógica de negocio de la aplicación, mientras que el `Controller` no tiene por qué preocuparse de ello.

Como se ha comentado, Digital Dice trabaja con un modelo de datos para registrar las interacciones, definido por los siguientes parámetros:

- `_id`: Identificador único de la interacción.
- `device`: Identificador del dispositivo que ha realizado la interacción. Este identificador es el mismo que el que se utiliza para identificar el dispositivo en la Thing Description, por ejemplo el que le correspondería en la SmartHome de la Thing Description de la Figura 4.2 es `acg:lab:suitcase-dd`.
- `source`: Indica el origen de la interacción, es decir, si la interacción ha sido realizada por un usuario, por el propio dispositivo, a consecuencia de un evento o proviene de un dispositivo virtual.

— `interaction`: Indica la interacción concreta que se ha realizado. Esta interacción debe estar declarada en la Thing Description del dispositivo en el cual el parámetro `device` hace referencia.
— `lastInteraction`: Indica si esta es la última interacción del tipo definido en `interaction`. Este parámetro es útil para saber si una interacción es la última de un tipo determinado, ya que sobretodo en operaciones de lectura podemos seleccionar rápidamente y sin tener que realizar comparaciones de los timestamps de las interacciones.
— `createdAt`: Fecha y hora en la que se realizó la interacción.
— `updatedAt`: Fecha y hora en la que se actualizó la interacción.
— `data`: Datos asociados a la interacción. Estos son instancias de los tipos de datos que estarán definidos en cada capacidad de interacción (`Interaction Affordance`) de la Thing Description.

El modelo de datos utilizado se oculta al usuario y no se expone en la API REST. El usuario no tiene que preocuparse de cómo se almacenan los datos, ni de cómo se recuperan, sólo tiene que preocuparse de enviar la petición correcta y de interpretar la respuesta. Estas peticiones estarán definidas mediante las `InteractionAffordances` de la Thing Description y coinciden con el campo `data` de nuestro modelo de interacción por lo que esto es lo único que ve el usuario.

En la Figura 4.7 se puede ver cómo se implementan las distintas peticiones posibles sobre una propiedad definida en la Thing Description, declaradas a través de las formas de acceso (`forms`) de la Thing Description que se puede ver en la Figura 4.2. En particular, esta API proporciona tres *endpoints* diferentes que permiten interactuar con las propiedades del dispositivo: obtener el estado actual de una propiedad (líneas 18–23), obtener actualizaciones en tiempo real de una propiedad (líneas 1–16) y actualizar el valor de una propiedad (líneas 25–38), las dos primeras mediante operaciones de tipo GET y la última mediante una operación de tipo POST.

El primer *endpoint*, especifica la ruta `/td.id/property/propertyName/sse`. En este caso, el valor del parámetro `td.id` es el ID del dispositivo y el de `propertyName` el nombre de la propiedad que se está solicitando. La opción `/sse` indica que este *endpoint* utiliza el protocolo SSE para enviar actualizaciones en tiempo real desde el servidor al cliente a través de una conexión HTTP persistente. Por ejemplo, con esta ruta se podría recibir, en tiempo real, el cambio de estado de la propiedad `status-light1` declarada en el escenario SmartHome, es decir, cuándo se enciende o se apaga la bombilla. En la sección de procesado de este *endpoint*, se configuran las cabeceras de la respuesta HTTP utilizando el objeto `headers`. En este caso, se especifica el tipo de contenido `text/event-stream`, que se mantenga la conexión abierta con `Connection: keep-alive`, y se desactiva la caché mediante `Cache-Control: no-cache`. Tras ello, se escribe la cabecera de la respuesta HTTP utilizando el método `res.writeHead`; Y por último, se utiliza el método `watch` del modelo `ThingInteraction` para suscribirse a los cambios en la base de datos que corresponden a la propiedad específica que se está solicitando. Esto se hace mediante el patrón *pipeline* de agregación de MongoDB, el cual permite establecer la propiedad específica que queremos recibir y el `source`, en este caso (`physicalDevice`), que refleja que solo se van a devolver las interacciones a la propiedad que se esta solicitando cuando el origen sea un dispositivo físico.

```
1  app.get('/' + td.id + '/property/:propertyName/sse', async (req, res) => {
2      const headers = {
3          'Content-Type': 'text/event-stream',
4          'Connection': 'keep-alive',
5          'Cache-Control': 'no-cache'
6      };
7      res.writeHead(200, headers);
8      ThingInteraction.watch([{$match: {$and: [  { "fullDocument.source": "
           physicalDevice"}, {"fullDocument.interaction": "property."+req.params
           .propertyName }]}}]).on('change', change => {
9          var response = new ThingInteraction(change.fullDocument);
10         res.write(`data: ${JSON.stringify(response.data)}\n\n`)
11     }
12     );
13     req.on('close', () => {
14         console.log(`Connection closed`);
15     });
16 })
17
18 app.get('/' + td.id + '/property/:propertyName', async (req, res) => {
19     ThingInteraction.findOne({ interaction: "property."+req.params.
           propertyName, source: "physicalDevice", lastInteraction: true}, {}, {
            sort: { 'createdAt': -1 } }, function (err, data) {
20         var response = new ThingInteraction(data);
21         res.send(response.data)
22     });
23 })
24
25 app.post('/' + td.id + '/property/:propertyName', async (req, res) => {
26     var interactionValue = new ThingInteraction({
27         device: "acg:lab:suitcase-dd",
28         source: "user",
29         interaction: "property." + req.params.propertyName,
30         data: req.body
31     })
32     interactionValue.save(function (err, doc) {
33         if (err) res.status(400).send(err);
34         else {;
35             res.send(doc.data);
36         }
37     })
38 })
```

Figura 4.7: Operaciones de acceso a la base de datos (endpoints).

Cuando se detecta un cambio en la base de datos, se construye un objeto del tipo `ThingInteraction` a partir del documento completo que representa el cambio y se envía al cliente, utilizando el método `res.write` (el valor de la propiedad mediante SSE), lo que se repetirá mientras el canal siga abierto y haya algún cambio en la propiedad a la cual el cliente esté suscrito. Como se puede observar, sólo se envía el parámetro `data` del modelo de interacción, ya que es el contenido de esta variable lo que espera el usuario. Finalmente, se define un *listener* para el evento `close` del objeto `req` de la petición del cliente, que se activa cuando se cierra la conexión entre el cliente y el servidor.

El segundo *endpoint*, definido entre las líneas 18 y 23, se utiliza para obtener el estado actual de una propiedad, y tiene la misma ruta que el *endpoint* anterior, pero sin la opción `/sse`. Para obtener el estado de la propiedad se utiliza el método `findOne` del modelo `ThingInteraction` el cual busca en la base de datos el documento más reciente

que se corresponde con la propiedad y el `source` del cambio que sea `physicalDevice`. De esta manera, se recibirá el último cambio que se produzco en la propiedad por parte del dispositivo físico. Una vez localizado el documento, se construye un objeto `ThingInteraction` a partir de él y se envía el valor de la propiedad al cliente haciendo uso de la operación `res.send` (línea 21).

El tercer *endpoint*, que se muestra entre las líneas 25 y 38 de la Figura 4.7, se utiliza para actualizar el valor de una propiedad. Este punto de acceso tiene el mismo formato de ruta que los dos anteriores, aunque en este caso se utiliza el método HTTP POST en lugar de GET. Para actualizar el valor de una propiedad, se construye un objeto `ThingInteraction` a partir de los datos recibidos en la solicitud HTTP, que incluyen el nuevo valor de la propiedad y se almacena en la base de datos utilizando el método `save`. Es importante destacar que determinados parámetros del objeto `ThingInteraction`, como el `source`, `device` e `interaction`, se establecen explícitamente en el código porque son conocidos por el servidor. Por ejemplo, el parámetro `source` de la interacción siempre será `user`, ya que es el usuario el que está enviando la solicitud HTTP. Por otro lado, el parámetro `device` siempre será el dispositivo declarado en la Thing Description, ya que es el único dispositivo con el que se puede interactuar. Por último, la interacción siempre será una propiedad cuyo valor venga definido en el campo `:propertyName`, valor que previamente se ha comprobado y validado en el microservicio `Controller`, por lo que se sabe que existe. Esto sucede porque no es posible acceder al microservicio `Data Handler` desde fuera del servidor, por lo que no hay ningún riesgo de que un usuario pueda enviar una solicitud HTTP con un valor de propiedad que no exista.

Si ocurre un error al guardar el objeto `ThingInteraction` en la base de datos, se envía una respuesta HTTP con un código de estado 400, indicando que la solicitud no se realizó con éxito. De lo contrario, se envía una respuesta HTTP con el nuevo valor de la propiedad actualizada, utilizando el método `res.send`. En el listado de la Figura 4.8 se observa un ejemplo del modelo de datos de interacción del encendido de un dispositivo del luz del escenario SmartHome. En la figura se aprecia que se trata de la interacción de un usuario (línea 4) sobre el dispositivo `acg:lab:suitcase-dd` (línea 3) que corresponde con la modificación de la propiedad `status-light1` (línea 5), cuyo valor se le asigna `true` (línea 7).

```
1  {
2    "_id" : ObjectId("6324599374a4a6182ea90538"),
3    "device" : "acg:lab:suitcase-dd",
4    "source" : "user",
5    "interaction" : "property.status-light1",
6    "data" : {
7      "value" : true
8    },
9    "createdAt" : ISODate("2022-09-16T11:10:11.418Z"),
10   "updatedAt" : ISODate("2022-09-16T11:10:11.418Z"),
11   "__v" : 0
12 }
```

Figura 4.8: Ejemplo de instancia del modelo de datos de interacción del encendido de un dispositivo de luz.

4.3.2. Consideraciones de despliegue

El proceso de despliegue del componente es similar al del componente `Controller`. Ambos requerirán la creación de una imagen de Docker mediante un archivo Dockerfile que permita su despliegue en un clúster de Kubernetes. Asimismo, se debe establecer un configuración para el **Deployment** que permite establecer el lanzamiento de un número de réplicas del microservicio, y generar una configuración de autoescalado horizontal (Horizontal-Pod-Autoscaler, **HPA**) que se encargue de ajustar el número de réplicas en función de la carga de trabajo. La única diferencia de despliegue entre ambos componentes (`Controller` y `Data Handler`) es en cuanto a la configuración de los servicios que dan acceso a los `pods`. En la Figura 4.9 se muestra el código de configuración de ambos microservicios: el de `Controller`, definido entre las líneas 1 a la 15, y de `Data Handler`, entre las líneas 17 a la 29.

```
 1  apiVersion: v1
 2  kind: Service
 3  metadata:
 4      name: dd-suitcase-controller-entrypoint
 5      namespace: default
 6  spec:
 7      type: NodePort
 8      selector:
 9      app: dd-suitcase-controller
10      ports:
11      - protocol: TCP
12        port: 443
13        targetPort: 443
14        nodePort: 30011
15        name: https
16  ---
17  apiVersion: v1
18  kind: Service
19  metadata:
20    name: dd-suitcase-dh-entrypoint
21    namespace: default
22  spec:
23    type: ClusterIP
24    selector:
25      app: dd-suitcase-dh
26    ports:
27    - protocol: TCP
28      port: 8063
29      targetPort: 8063
```

Figura 4.9: Configuración de los servicios `Controller` y `Data Handler`

Como se puede observar en el archivo de configuración, en el caso del componente `Controller`, este se describe mediante un servicio de tipo `NodePort` en Kubernetes, que sirve para exponer un conjunto de `pods` de un clúster de Kubernetes a través de un puerto específico. Con este tipo de servicio, los `pods` del clúster son accesibles desde fuera del clúster utilizando la dirección IP de cualquier nodo del cluster y el puerto asignado al servicio. Por otro lado, en el caso del componente `Data Handler`, este se describe mediante un servicio de tipo `ClusterIP` en Kubernetes. Este tipo de servicio expone un conjunto de `pods` en un clúster de Kubernetes a través de un puerto específico. Con este

tipo de servicio, los `pods` del clúster son accesibles desde dentro del clúster, utilizando el nombre DNS del servicio y el puerto asignado, pero no desde fuera. De este modo, el componente no se expone al exterior del clúster, sino que se accede a él desde otros servicios internos del mismo clúster, como en el caso del microservicio `Controller`. El uso de un servicio de tipo `ClusterIP` para exponer el componente `Data Handler` presenta múltiples beneficios en términos de seguridad, eficiencia, escalabilidad y flexibilidad. Al no exponerse fuera del clúster, el `Data Handler` está protegido contra posibles amenazas externas, lo que aumenta la seguridad y reduce el riesgo de vulnerabilidades.

4.4. El componente Reflection

El componente `Reflection` se encarga de establecer la comunicación bidireccional con el dispositivo físico y de reflejar los cambios de estado del dispositivo físico en la base de datos y la petición del usuario en el dispositivo físico. Además, este componente también es responsable de traducir los mensajes que se envían y reciben del dispositivo físico a un formato que pueda ser interpretado por el resto de los microservicios del sistema. Por ejemplo, si el dispositivo físico utiliza un protocolo de comunicación como MQTT, el componente `Reflection` traduce los mensajes que se envían y reciben al modelo de interacción usado por la base de datos, que pueden interpretar el resto de los componentes. Este componente representa el único punto de conexión con el dispositivo físico. La principal ventaja de ello es que el resto de los componentes no necesitan conocer el protocolo de comunicación que utiliza el dispositivo físico, lo que facilita la escalabilidad del sistema. Además, al ser el único punto de comunicación, tiene implicaciones importantes a nivel de seguridad y de rendimiento. Por ejemplo, si el dispositivo físico se encuentra en una red privada, el resto de los componentes no necesitarán conocer la dirección IP del dispositivo físico. Otra ventaja de que este microservicio sea el único punto de comunicación con el dispositivo físico es que las peticiones de escritura de los usuarios serán atendidas por este microservicio. Esto permite establecer distintos comportamientos en función de la situación, como por ejemplo la aplicación de políticas de seguridad específicas o la gestión de errores de manera más eficiente.

El componente `Reflection` se encarga de suscribirse a las interacciones de lectura del dispositivo para su correcto funcionamiento. En el caso de que el dispositivo no permita la suscripción, este componente realiza solicitudes de tipo *long-polling* cada `X` segundos (siendo `X` un tiempo configurable), para ir recuperando los cambios de estado del dispositivo y reflejar esos cambios en al base de datos. Además, `Reflection` observa las operaciones de escritura relacionadas con el dispositivo controlado en la base de datos, para ejecutarlas en el dispositivo (físico o virtual) cuando es necesario.

La elección de MongoDB como sistema de base de datos ofrece importantes beneficios para el funcionamiento de `Reflection`. En concreto, la capacidad de suscribirse a consultas particulares mediante un patrón reactivo [Davis and Davis, 2019] hace que la base de datos sea responsable de notificar al suscriptor cuándo hay nuevos datos que cumplen con los parámetros establecidos en la consulta. Esto permite que el microservicio no tenga que estar continuamente solicitando datos a la base de datos, lo que reduce la carga de trabajo en el sistema y aumenta la eficiencia en la comunicación.

Para el correcto funcionamiento del componente `Reflection`, es necesario que este tenga acceso tanto al dispositivo físico como a la base de datos. El microservicio que implementa este componente es independiente del resto y funciona de manera autónoma, por lo que no necesita conectarse a ningún otro microservicio del sistema. En este sentido, es importante destacar que `Reflection` puede ejecutarse fuera del clúster de Kubernetes, lo cual ofrece mayor flexibilidad en el despliegue. No obstante, es recomendable que el microservicio del compomente `Reflection` se ejecute en el mismo entorno donde está conectado el dispositivo físico, ya que esto reduce la latencia en la comunicación y mejora la eficiencia del sistema en general. Es importante señalar que `Reflection` es un microservicio que trabaja sin ningún tipo de réplica, lo que significa que no requiere de múltiples instancias en ejecución. Esto puede suponer un cuello de botella a nivel software, al trabajar con una única replica. No obstante, se decidió establecer así por los beneficios descritos en el Capítulo 3, entre otros, la capacidad de poder asegurar que las operaciones de escritura se realicen en orden de llegada y, en el caso de que fuera necesario, establecer ciertas heurísticas a la hora de decidir si se van a ejecutar ciertas peticiones o no, por ejemplo, si una luz ya esta encendida no es necesario volver a ejecutar una petición que la encienda. Por otro lado, al tener sólo un punto de conexión con el dispositivo, se evitan errores de mal funcionamiento propios de las tecnologías basadas en bus (como en KNX, Modbus o Netbus) que pueden llegar a tener limitado el número de agentes que pueden estar conectados al bus en un momento dado.

La independencia de `Reflection` en el sistema y su capacidad de ejecución en diferentes entornos (i.e., Docker o Kubernetes), hacen que este componente sea más flexible y sencillo de desplegar que otros microservicios del Digital Dice. Además, su capacidad de trabajar de forma autónoma y sin réplicas facilita su gestión y mantenimiento.

4.4.1. Consideraciones de implementación

La implementación de `Reflection` es muy distinta a las vistas en los otros microservicios, ya que este no necesita ningún tipo de sistema de comunicación con el exterior y no necsita una API para su correcto funcionamiento. Su propósito es observar lo que está ocurriendo en el dispositivo (físico o virtual) y reflejarlo en el formato correcto dentro de la base de datos. De igual manera, cuando se realiza una operación de escritura en la base de datos, `Reflection` se encarga de reflejarla en el dispositivo físico.

En el listado de la Figura 4.10 se muestra un fragmento del código del componente `Reflection`. Como se puede observar en el código, este componente utiliza la librería WoTnectivity, presentada en el Capítulo 3, encargada de realizar las operaciones de escritura y lectura en el dispositivo físico. En la primera parte del código, se importan las dependencias necesarias, se establece la conexión con la base de datos y se define un modelo de datos para las interacciones con el dispositivo. Además, se definen dos funciones: `executeInteractions` y `subToInteractions`. La operación `subToInteractions` se encarga de suscribirse a las interacciones de lectura que se encuentran en el dispositivo físico. Cada vez que se detecta un cambio en el dispositivo físico, esto queda reflejado en la base de datos. En el ejemplo concreto que se ve en la Figura 4.10, correspondiente con el código de Reflection del escenario SmartHome, la función `subToInteractions` utiliza la librería `wotnectivity-knx` para enviar una solicitud de suscripción al dispositivo físi-

```
const db = require('./config');
const wotnectivity = require('wotnectivity-knx');
var thingInteractionSchema = require('./models');
const { set } = require('./models');
var ThingInteraction = db.model('ThingInteraction', thingInteractionSchema);

async function subToInteractions(){
    var statusProp = await wotnectivity.sendRequest(process.env.SUITCASE_URI
        ,{ requestType: "subscribe", groups: [{group: "2/0/1", dataType: "
        DPT1.001"},{group: "2/1/1", dataType: "DPT1.001"}]})
    statusProp.subscribe(async (data)=>{
        var interactionValue=new ThingInteraction({
            device: "acg:lab:suitcase-dd",source: "physicalDevice",
                interaction:"property.status",data: data
        })
        switch (data.group) {
            case "2/0/1":
                interactionValue.device = "acg:lab:suitcase-dd";
                interactionValue.interaction = "property.status-light1";
                interactionValue.data = {value:data.value}
                break;
            default:
                interactionValue.device = "acg:lab:suitcase-dd";
                interactionValue.interaction = "unknown";
                break;
        }
        interactionValue.save(function(err){
            if(err){
                 console.log(err);
                 return;
            }
        })
    })
}

async function executeInteractions(){

    ThingInteraction.watch([{$match: {$and: [  { "fullDocument.source": "user
        "}, {"fullDocument.device": "acg:lab:suitcase-dd"}]}}]).on('change',
         async change => {
        var response = new ThingInteraction(change.fullDocument);
        switch (response.interaction){
            case "property.status-light1":
                var reqProp = await wotnectivity.sendRequest(process.env.
                    SUITCASE_URI, { requestType: "write", group: "2/0/0",
                    dataType: "DPT1.001" }, response.data.value);
                break;
            case "action.switch-light1":
                var reqProp = await wotnectivity.sendRequest(process.env.
                    SUITCASE_URI, { requestType: "write", group: "2/0/0",
                    dataType: "DPT1.001" }, response.data.value);
                break;
            default:
                break;
        }
    }
}
subToInteractions()
executeInteractions()
```

Figura 4.10: Implementación parcial del componente `Reflection`.

co. En esta solicitud, se especifica el tipo de petición, que en este caso es "subscribe", y se definen los parámetros de las interacciones a las que se tiene que suscribir, indicando su dirección de grupo (por ejemplo, "2/0/1"), párametro que establece una dirección física del dispositivo concreto al que se conecta, y su tipo de datos (por ejemplo, "DPT1.001", valor binario), que indica el tipo de datos que se tiene que mandar a la dirección de grupo definida [KNXAssoc, 2021]. Cuando se recibe un cambio de estado en alguno de los grupos suscritos, se crea un objeto de tipo `ThingInteraction` con los datos del cambio de estado y se almacena en la base de datos MongoDB.

El método `executeInteractions` se encarga de ejecutar las interacciones de escritura que se encuentran en la base de datos. Para ello, se suscribe a la colección correspondiente referente a las operaciones de escritura (propiedades de escritura y acciones) de la Thing concreta que el Digital Dice está manejando, y cada vez que se detecta una nueva interacción, se ejecuta la operación de escritura correspondiente en el dispositivo físico. Al ser un método asíncrono, este se ejecuta en un hilo de ejecución independiente, por lo que no se bloquea el resto de la ejecución del microservicio. En el ejemplo concreto del escenario SmartHome, la función `executeInteractions` se encarga de realizar las operaciones de escritura en el dispositivo en función las interacciones que se hayan registrado en la base de datos. Para ello, se utiliza la función `watch` de la librería de MongoDB para detectar cambios en la colección `ThingInteraction` que correspondan con el dispositivo manejado. Cuando se detecta un cambio, se ejecuta la operación correspondiente en el dispositivo utilizando la función `sendRequest` de la librería `wotnectivity-knx`. Puesto que el componente `Reflection` controla un dispositivo de tipo KNX, es importante conocer los valores de la dirección de grupo que controla cada interacción, así como los tipos de datos con los que cada una de esas direcciones de grupo trabajan.

4.4.2. Consideraciones de despliegue

Al contrario que `Data Handler` y `Controller`, el componente `Reflection` no requiere ser desplegado en un clúster de Kubernetes. Por su carácter atómico, `Reflection` puede desplegarse como un servicio local (*standalone*), un servicio Docker o en Kubernetes. En el escenario ejemplo SmartHome, el componente `Reflection` ha sido desplegado como un servicio Docker, por lo que dentro de la carpeta `reflection` se encuentra el archivo `Dockerfile` necesario para construir la imagen. En la Figura 4.11 se puede observar el contenido de este archivo de despliegue. En primer lugar, se define la imagen base para construir la imagen Docker de `Reflection`. En este caso, se ha utilizado la imagen oficial de Node.js 14, al venir ya con el interprete de este lenguaje de programación y por ser este lenguaje usado para desarrollar el resto de los componentes de Digital Dice. A continuación, se define el directorio de trabajo y se copia el archivo `package.json`, que contiene las dependencias del componente `Reflection`, y que se encuentran en la carpeta `reflection`. Tras copiar los archivos, se ejecuta el comando `npm install` para instalar las dependencias necesarias para ejecutar el componente. Finalmente, se copia el resto de los archivos de la carpeta `reflection` en el directorio de trabajo y se define el comando que se va a ejecutar al arrancar el contenedor. No obstante, también se puede lanzar el componente `Reflection` de forma local utilizando el comando `node src/index.js`, siempre y cuando se hayan descargado las dependencias.

```
1  FROM node:14
2
3  WORKDIR /usr/src/app
4  COPY package*.json ./
5  RUN npm install
6  COPY . .
7  CMD [ "node", "src/index.js" ]
```

Figura 4.11: Archivo de configuración Docker del componente `Reflection`.

4.5. El componente Event Handler

Una vez visto el comportamiento de los componentes obligatorios de un Digital Dice, a continuación se describirán los opcionales: `Event Handler`, `Virtualizer` y `UI`. El componente `Event Handler` tiene como objetivo principal gestionar los eventos producidos por el dispositivo físico, registrado previamente por el microservicio `Reflection`. También es el encargado de escuchar eventos tanto externos como internos que puedan afectar al dispositivo que controla, encargándose de las relaciones de causalidad que afecten al dispositivo. Como se ha descrito, las relaciones de causalidad permiten establecer un sistema de interacción entre dispositivos basado en *causas* y *efectos*. Esto permite que el componente pueda actuar como un gestor de eventos, como se detalla en el Capítulo 3.

Desde el punto de vista de la comunicación, las peticiones se realizan de manera similar a los componentes `Controller` y `Data Handler`, aunque en este caso se utiliza únicamente el protocolo de comunicación SSE, ya que este componente se encarga de gestionar los eventos que genera el dispositivo físico, (p.ej., temperatura de la sala) y los eventos complejos (p.ej., se activa el aire acondicionado porque ha habido un aumento de la temperatura y se ha detectado movimiento dentro de la sala). Las rutas de los eventos siguen el mismo patrón que `Controller` o `Data Handler` es decir `/'+td.title+'/event/:eventName`. El funcionamiento interno para los eventos simples es también muy similar al funcionamiento del componente `Data Handler`: se requiere de una conexión a la base de datos con un patrón reactivo a la espera de que se produzca un cambio en la colección de interacciones hasta que se dispare un evento del dispositivo en cuestión. Una vez se produce el evento, se envía el evento al cliente que se haya suscrito a dicho evento a través del `Controller`.

El componente `Event Handler` tiene la particularidad de que también debe controlar los eventos producidos a partir de las relaciones de causalidad. Para ello, `Event Handler` en primer lugar obtiene el modelo de causalidad correspondiente a la Thing que el Digital Dice está manejando. Una vez `Event Handler` cuenta con el documento que describe la relación de causalidad de la Thing, se encarga de interpretarlo y de gestionar los eventos que se produzcan. La Figura 4.12 muestra el código del `Event Handler` que se encarga de manejar estas relaciones de causalidad. Para explicar este código, se toma como referencia una relación de causalidad introducida en el Capítulo 2. La relación de causalidad se muestra en la Figura 4.13, la cual contiene el efecto `turnoff` (líneas 10–31). Este efecto se dispara siempre y cuando pase una de las dos causas, que la temperatura baje de los 22°C (líneas 11–20 y línea 30) o que no se detecte movimiento o presencia de personas dentro de la sala (líneas 20–28 y línea 30).

```
async function effectsControl(){
    var causes=[[]];
    var effectsKeys = Object.keys(td.effects)
    console.log(effectsKeys)
    effectsKeys.forEach((key,effectIndex)=>{

        td.effects[key].causes.forEach((cause, causeIndex)=>{
        console.log("CausesIn")
        ThingInteraction.watch([{$match: {$and: [  { "fullDocument.source": "
             physicalDevice"}, {"fullDocument.interaction": cause.
             interactionType+"."+cause.interaction }]}}]).on('change', async
             change => {
          causes[effectIndex][causeIndex]=change.fullDocument;
          console.log("interaction watched")
          if(td.effects[key].window){
            var inWindow = await calcWindow(causes[effectIndex], td.effects[
                 key])
            console.log(inWindow);
            if(inWindow) evalExpresion(causes[effectIndex], td.effects[key])
        } else evalExpresion(causes[effectIndex], td.effects[key])
      })
    });
  });

}

async function calcWindow(causes, effect){
    if(causes.length == effect.causes.length){
        if(!effect.hasOrder){
            causes.sort((a, b) => {return a.createdAt.getTime()-b.createdAt.
                 getTime()})
        }
        var timeBetween = 0;
        causes.forEach((element,indexElement) => {
          if(indexElement==0);
          else{
            var auxTime = (element.createdAt.getTime()-causes[indexElement
                 -1].createdAt.getTime())/1000;
            if(auxTime < 0){
              timeBetween+= -Infinity;
            }
            else{
              timeBetween+= auxTime;
            }
        }
    });
        console.log(effect.window);
        console.log(timeBetween);
        if(effect.window-Math.abs(timeBetween)>0 && effect.window-Math.abs(
             timeBetween)<effect.window) return true;
    }
    return false;
}

async function evalExpresion(causes, effect){
    eval(effect.evalExpression);
}

effectsControl();
```

Figura 4.12: Componente Event Handler: gestión de las relaciones de causalidad.

```
{
  "id": 001,
  "thingId": "urn:dev:ops:room1-ac",
  "isManaged": false,
  "self": {
    "rel": "self",
    "href": "http://ac:13432/room1-ac",
    "type": "application/td+json" },
  "effects": {
    "turnOff": {
      "causes": [{
          "link": {
            "rel": "tempSensor",
            "href": "http://tempaddrs:13432/room1-tempSensor",
            "type": "application/td+json"
          },
          "interactionType": "properties",
          "interaction": "temp",
          "origin": "physicalDevice"
        }, {
          "link": {
            "rel": "movement",
            "href": "http://moveaddrs:13432/room1-movement",
            "type": "application/td+json"
          },
          "interactionType": "events",
          "interaction": "movement",
          "origin": "physicalDevice" }}],
      "hasOrder": false,
      "evalExpression": "causes[0].value < 22 || causes[1].value == false
     : processInteraction('this.td.actions.shutDown','')"
}  }  } }
```

Figura 4.13: Relación de causalidad que controla un aire acondicionado.

En el código de la Figura 4.12 se observa cómo se llama a la función `effectsControl` que se encarga de controlar los efectos establecidos en las relaciones de causalidad. En esta función (líneas 1–21), se declara la lista de causas en la variable `causes`, que almacenará las causas de los distintos efectos. En el ejemplo, la causa `tempSensor` hace referencia a la interacción `property.temperature` de un sensor de temperatura y la causa `movement` hace referencia a la interacción. Luego se obtienen las claves de los efectos (`effectsKeys`) que se van a controlar y se realiza un bucle para recorrer cada una de ellas (en el ejemplo sólo está el efecto `turnOff`). Dentro del bucle, se recorre cada una de las causas asociadas a cada efecto y se realiza una suscripción a la base de datos para escuchar los cambios de cada una de las causas. Posteriormente, se llama a la función `calcWindow` (líneas 23–46) que calcula si las causas van ocurriendo en el periodo de tiempo indicado por la ventana temporal (`window`) establecida en la declaración del efecto. Si se cumple la ventana temporal, se llama a la función `evalExpresion` (líneas 48–50) que evalúa la expresión lógica asociada al efecto. La función `calcWindow` recibe como parámetros las causas asociadas y un objeto efecto, y compara si el número de causas es igual al indicado en el efecto. Además, si el efecto tiene un orden específico, se ordenan las causas cronológicamente para su posterior comparación. Seguidamente, se recorren las causas y se calcula el tiempo transcurrido entre ellas. Si alguna causa ocurrió en un momento anterior, se establece el tiempo entre ellas como `-Infinity`, para discriminar

la ejecución del efecto. Por último, esta función compara el tiempo transcurrido con el de la ventana temporal del efecto. Si el tiempo transcurrido es inferior, la función devuelve `true`. Finalmente, si la ventana temporal se cumple, se evalúa la expresión lógica asociada al efecto (`evalExpression`) mediante la función `evalExpression`. En caso de que no haya ventana de tiempo cuando se recibe alguna de las causas, se evalúa directamente el efecto. En el ejemplo se comprueba si la temperatura es menor de 22°C o no hay movimiento en la sala, momento en el cual se ejecuta la acción `shutdown` del aire acondicionado. En resumen, la función `effectsControl` es una función que se encarga de controlar los efectos producidos por una serie de causas en el sistema. Para ello, se suscribe a los cambios de las causas, calcula si estas tienen que ocurrir en un periodo de tiempo específico, y si es así, evalúa la expresión lógica asociada.

A nivel de despliegue, el `Event Handler` utiliza las mismas configuraciones descritas para el componente `Data Handler`.

4.6. Los componentes Virtualizer y UI

Por último, existen otros dos componentes opcionales más, el de virtualización y el de interfaz de usuario. El microservicio de virtualización permite simular el comportamiento de un dispositivo físico real. Por otro lado, el componente de interfaz de usuario permite visualizar la información de los dispositivos físicos y virtuales, así como la información de los sensores y actuadores de los dispositivos físicos y virtuales. Se describen ambos servicios de forma simultánea porque dichos servicios se deben crear a medida para cada tipo de dispositivo IoT, ya que requieren de conocimiento del experto y no son generalizables. A continuación, se describirán dos ejemplos de implementación.

El componente de virtualización actúa como sustituto del dispositivo físico, de manera similar a un Digital Twin, ofreciendo una interfaz de programación de aplicaciones (API) que es idéntica a la utilizada para interactuar con el dispositivo físico real. Esta interfaz permite realizar operaciones de lectura y escritura en la representación virtual del dispositivo, permitiendo a los desarrolladores de sistemas IoT probar su funcionamiento de forma independiente y crear escenarios de prueba complejos y variados sin tener que preocuparse por la disponibilidad y el estado del dispositivo físico real.

A nivel de implementación, depende mucho de la complejidad de la virtualización que se quiera realizar. La Figura 4.14, muestra parte del código de un microservicio de virtualización que simula el ejemplo de un contenedor de basura de escenario de ciudad inteligente presentado en el Capítulo 3. En el código se observa la implementación de la acción de vaciado del contenedor. Esta acción se realiza mediante una petición POST a la API del microservicio de virtualización. La petición POST se realiza sobre la ruta `/container/:serialNumber/action/empty_container`, donde `:serialNumber` es el número de serie del contenedor. La petición POST recibe como parámetro el número de serie del contenedor, y devuelve como respuesta el estado del contenedor tras la acción de vaciado. Por otro lado, también se dispone de una virtualziación en la operación `ejecutar`, donde se simula la generación de residuos en cada uno de los contenedores. Este comportamiento se ejecuta cada 5 minutos, pero se puede acelerar o ralentizar el tiempo de ejecución mediante el parámetro `factorAceleracion`.

```
var containers = await container.find({});
app.post('/container/:serialNumber/action/empty_container', function(req
    , res) {
    container.findOne({serialNumber: req.params.serialNumber}, {}, {
        sort: { 'createdAt': -1 } }, function (err, data) {
        var response = new container(data);
        response.capacity = 0;
        response.save();
        res.send(response);
    });
});
async function ejecutar(){
    setTimeout(function(){
        ejecutar();
    }, tiempo)

        await containers.forEach(async (container) => {


            await throwGarbage(container);

            if(container.capacity > container.totalCapacity){

            }else{

                var cont = await container.save();

            }
        }
    })
    tiempo = (300000 * (1 / factorAceleracion));
}
```

Figura 4.14: Virtualización del ejemplo de *contenedor de basura*.

El componente `UI` comparte ciertas particularidades con el de virtualización. Son componentes completamente personalizados y adaptados al entorno de experimentación que se quiera realizar. En el caso de la interfaz gráfica de usuario, los Digital Dice del WoT-Lab han sido implementados mediante la tecnología de Web Components de Svelte [Libby, 2022]. En la Figura 4.15, se puede observar un ejemplo de una interfaz gráfica de usuario para el escenario de los *contenedores de basura*, la cual está disponible en el laboratorio WoT Lab[3] del Grupo de Investigación de Informática Aplicada (TIC-211) de la Universidad de Almería, y también está accesible en el Portal Web Digital Dice[4]. Para embeber la interfaz proporcionada por el componente `UI`, se puede realizar mediante un *iframe*, o mediante la API de Web Components de Svelte.

En cuanto al proceso de despliegue de los componente `Virtualizer` y `UI`, estos se realizan de la misma forma que el resto de los componentes Digital Dice: mediante Docker y Kubernetes. Para ambos, el despliegue se realiza en el mismo espacio de nombres que los componentes `Controller` y `Data Handler`, ya que todos ellos son componentes de uso interno al Digital Dice.

[3]ACG WoT Lab - https://acg.ual.es/projects/cosmart/wot-lab/
[4]Portal Digital Dice: `https://acg.ual.es/projects/cosmart/digitaldice`

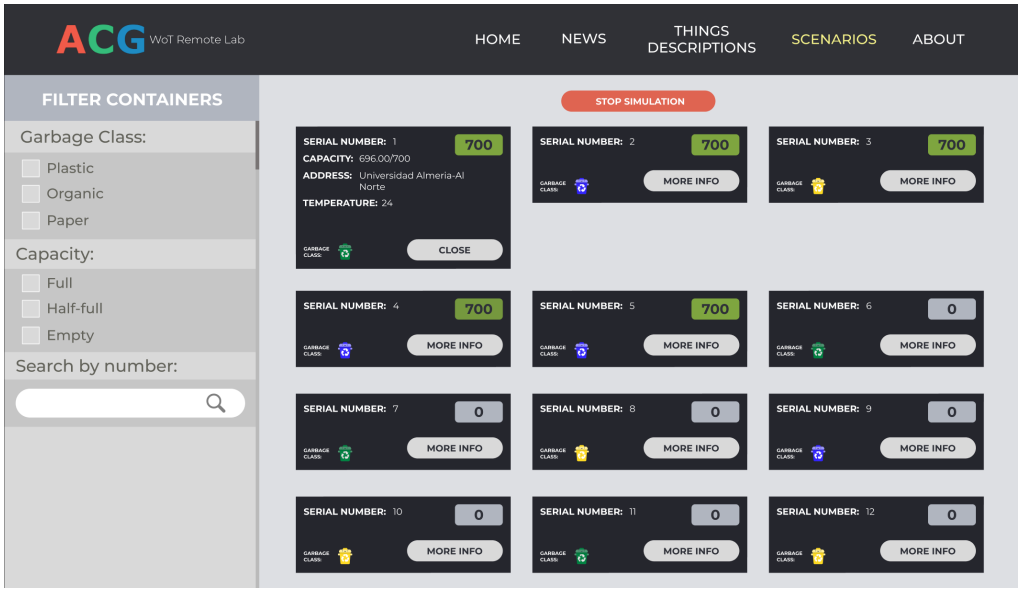


Figura 4.15: Ejemplo de interfaz gráfica de un Digital Dice.

Capítulo 5

Conclusiones y Trabajo Futuro

Capítulo 5

Conclusiones y Trabajo Futuro

Contenido

5.1. Introducción

En este capítulo final se resumen las principales contribuciones del trabajo de investigación de la tesis doctoral, así como sus limitaciones y líneas de investigación abiertas pendientes de explorar en futuros trabajos.

En el trabajo de tesis doctoral se han abordado varias cuestiones relacionadas con el desarrollo, la gestión y el uso de componentes digitales que forman parte de soluciones basadas en el Internet de las Cosas (IoT) y de Sistemas Ciberfísicos (CPS). En primer lugar, se han estudiado las soluciones existentes en la literatura, para analizar la oportunidad y viabilidad de llevar a cabo el desarrollo una nueva propuesta de componente digital. La nueva solución propuesta, que recibe el nombre de *Digital Dice*, es una aproximación al desarrollo de componentes digitales que se basa en la descomposición de cada componente en módulos que agrupan funcionalidades con características comunes. Esta agrupación facilita el desarrollo, el mantenimiento, y la reutilización, pero el principal objetivo es mejorar la gestión de la ejecución de los componentes digitales.

La división de los componentes digitales en partes más pequeñas se puede ver beneficiada por una aproximación basada en *microservicios*, en la que es posible encapsular cada uno de los diferentes módulos que conforman un componente en un artefacto software de un tamaño reducido. Esta aproximación proporciona una distribución más práctica de las operaciones que forman parte de los componentes, siempre que la funcionalidad de cada microservicio esté bien definida y sea accesible a través de una *interfaz*. En este sentido, tanto la comunicación entre las diferentes partes en las que se divide un componente digital como la comunicación entre los componentes digitales de una solución software debe llevarse a cabo de manera efectiva y eficiente. Para conseguir esta comunicación, así como para satisfacer el resto de requerimientos comunes a las aplicaciones IoT y a los sistemas ciberfísicos, se propone una *arquitectura* de microservicios para componentes digitales.

Tanto para el ámbito del IoT, en el que los principales objetivos se basan en habilitar la conexión de los dispositivos físicos con Internet permitiendo la transmisión de datos mediante redes de comunicación; como para el ámbito de los sistemas ciberfísicos, en el que los esfuerzos están enfocados en mejorar la interacción entre los elementos físicos de un sistema con los elementos digitales, existe un problema de *heterogeneidad* en las tecnologías y los protocolos utilizados por los diferentes dispositivos. Esta heterogeneidad, que supone una limitación en la comunicación entre los componentes de un sistema, puede ser reducida, ocultada o solventada si se construye una capa de abstracción que permita *homogeneizar* la definición y el uso de dichos componentes.

Una alternativa de la que disponen los ingenieros de software para incorporar una capa de abstracción en la construcción de soluciones IoT y sistemas ciberfísicos es utilizar las recomendaciones de la *Web de las Cosas* (WoT) del W3C. El conjunto de recomendaciones de la WoT tiene como objetivo mejorar la interoperabilidad entre los distintos dispositivos y las distintas plataformas existentes en el dominio del IoT. Esto se consi-

gue mediante la aplicación de una metodología de desarrollo basada en tecnologías web. Esta aproximación permite beneficiarse de la madurez de estas tecnologías y aplicar la experiencia de los desarrolladores en este ámbito para reducir los costes en el proceso de desarrollo de mantenimiento de soluciones IoT y sistemas ciberfísicos.

En este trabajo de tesis doctoral se propone una arquitectura de microservicios que permite gestionar y utilizar soluciones basadas en dispositivos IoT y en sistemas ciberfísicos. La arquitectura desarrollada se basa en los estándares de la WoT para reducir la fragmentación y heterogeneidad existentes en este tipo de sistemas, proporcionando una capa homogénea para la definición, la implementación y la comunicación de sus componentes. Esta arquitectura se basa en una propuesta de componente digital denominado Digital Dice, cuyo nombre proviene de las siguientes consideraciones. Digital Dice es una representación digital de un dispositivo (físico o virtual) que forma parte de un sistema ciberfísico o una instalación IoT. Este componente digital está constituido por seis partes o módulos, que encapsulan la funcionalidad asociada una de las siguientes *capacidades* o *facetas*: controlador de la comunicación, gestor de datos, gestor de eventos, interfaz de usuario, sincronizador con el dispositivo físico, y virtualizador.

El hecho de que se trate de una representación digital de un dispositivo, incluyendo la capacidad específica encargada de su virtualización, así como el nombre de Digital Dice, establece una analogía con el concepto de gemelo digital o *Digital Twin*. Una aproximación basada en el concepto de Digital Twin se centra principalmente en replicar digitalmente un dispositivo físico en un componente virtual (con sus funcionalidades, características y comportamiento). No obstante, este tipo de soluciones no establecen una aproximación a varios niveles que permitan abordar de forma específica las diferentes capacidades de un dispositivo, como pueden ser la gestión de la interacción de los usuarios, el control de los eventos de los dispositivos, o el aprovisionamiento de recursos según la demanda, entre otros posibles ejemplos.

La solución propuesta de Digital Dice presenta una serie de mejoras con respecto al concepto de Digital Twin. Además de las capacidades de virtualización y simulación (que implican también la sincronización con el dispositivo físico), Digital Dice incluye las capacidades adicionales mencionadas anteriormente de control de la comunicación, gestión de datos y eventos, e interfaz de usuario. El desarrollo de cada una de estas facetas se aborda mediante una aproximación basada en microservicios, lo que permite escalar únicamente aquellas capacidades que necesiten más recursos en un momento determinado, y gestionar cada una de las capacidades de manera independiente. De esta forma, es posible reducir las limitaciones que presentan los dispositivos de una aplicación IoT o de un sistema ciberfísico, como pueden ser unas reducidas capacidades de computación o de almacenamiento, tiempos de respuesta elevados, o un alto número de peticiones concurrentes, entre otros posibles ejemplos.

Debido a que la arquitectura propuesta y Digital Dice se basan en los estándares de la WoT, existe una capa homogénea de comunicación que facilita la integración y la interoperabilidad, y que permite gestionar y utilizar los dispositivos IoT y ciberfísicos a través de una interfaz de programación de aplicaciones (API). Por un lado, la recomendación que describe la arquitectura de la WoT permite definir los requerimientos y la estructura de los elementos que participan en nuestra aproximación, así como las relaciones y las interacciones entre estos elementos. Por otro lado, la recomendación que

establece los principios para describir los elementos individuales que forman parte de la arquitectura permiten homogeneizar la interacción con los dispositivos y su descripción.

Este capítulo final se estructura en las siguientes cuatro secciones. La Sección 5.2 describe las principales contribuciones que el trabajo de investigación desarrollado aporta a la comunidad científica. En la Sección 5.3 se identifican las principales limitaciones de la propuesta descrita en este trabajo de tesis. La Sección 5.4 presenta las líneas de investigación que quedan abiertas. Por último, la Sección 5.5 ofrece un listado de las publicaciones derivadas del trabajo de investigación realizado.

5.2. Contribuciones de la Tesis Doctoral

En esta sección se describen las contribuciones aportadas a lo largo de este trabajo de tesis doctoral. Las contribuciones se han dividido en los cuatro hitos o en las cuatro capacidades principales descritas en profundidad a lo largo de esta tesis: Digital Dice como solución para el manejo de dispositivos, WoTnectivity como librería para la comunicación con dispositivos, las relaciones de causalidad para describir las interacciones entre dispositivos, y Digital Dice como sistema de alta disponibilidad.

5.2.1. Digital Dice

A lo largo de la tesis doctoral, se ha presentado Digital Dice como una solución basada en microservicios para el manejo y gestión de dispositivos IoT y sistemas ciberfísicos. Digital Dice abstrae a los usuarios y desarrolladores de aplicaciones de este tipo de dispositivos de las tecnologías que estos utilizan para comunicarse con el mundo exterior. Se han llevado a cabo las siguientes contribuciones relacionadas con Digital Dice:

— Se ha establecido y desarrollado una arquitectura de microservicios para el manejo de dispositivos IoT y CPS, los Digital Dice. Una de las piezas centrales de Digital Dice es el uso de la Web of Things, específicamente la Thing Description para la declaración de la funcionalidad del mismo a través de este estandar, estableciendo el vinculo fundamental que hay entre Thing Description y Digital Dice [Mena et al., 2019a] [Mena et al., 2019d] [Criado et al., 2020] [Llopis et al., 2021] [Llopis et al., 2022][Mena et al., 2023] (*Capítulo 2*).

— Se han definido las distintas partes que forman un Digital Dice, tanto microservicios como servicios auxiliares. Para ello hemos estudiado la arquitectura de Digital Dice junto con las distintas capas que la componen [Mena et al., 2019a] [Mena et al., 2019d] [Mena et al., 2023] (*Capítulo 2*).

— Se ha desarrollado un lenguaje específico de dominio (DSL) para la definición de las partes de un Digital Dice [Mena et al., 2023] (*Capítulo 2*).

— Se ha establecido un modelo de datos que Digital Dice utiliza para registrar las interacciones por parte de los usuarios, así como los datos generados por los dispositivos que nuestros DD están representando [Mena et al., 2023] (*Capítulo 2* y *Capítulo 4*).

— Se ha desarrollado un proceso de transformación de modelo a texto (M2T) para establecer la base de código de los distintos componentes de un Digital Dice a partir de la Thing Description que lo representa [Mena et al., 2019c] [Mena et al., 2021b] (*Capítulo 2*).

— Se ha explicado de forma detallada la implementación de los distintos componentes que forman parte de un Digital Dice. A la vez se ha estudiado los detalles de despliegue de cada uno de los componentes de Digital Dice y se han visto distintos ejemplos de Digital Dice de distinta índole mediante escenarios relacionados con las Smart Home y las Smart Cities [Mena et al., 2023] (*Capítulo 4*).

5.2.2. WoTnectivity

Para establecer conexiones con los diferentes protocolos encontrados en el dominio del IoT, se ha desarrollado WoTnectivity, una librería multiprotocolo para trabajar con diferentes tipos de protocolos, como HTTP [Krishnamurthy and Rexford, 2001], MQTT [Hunkeler et al., 2008], WebSocket [Wang et al., 2013] o KNX [KNXAssoc, 2009], y que tiene la posibilidad de ampliar la cantidad de protocolos soportados a través de la interfaz que la librería proporciona. Las contribuciones derivadas son las siguientes:

— Se ha desarrollado una librería multiprotocolo para la comunicación con dispositivos IoT y CPS. En esta librería se ha establecido un comportamiento similar para la comunicación con distintos dispositivos IoT, independientemente de la tecnología que estos utilicen [Mena et al., 2020] (*Capítulo 3*).

— Se ha desarrollado la librería WoTnectivity para los lenguajes de programación Java y Node.js, en esta última con una aceptación bastante interesante aportando una media de descarga en alguna de sus vertientes de unos 15 usuarios semanales[1]. Se ha comparado el uso esta librería con distintas librerías de uso común para conectarnos con diferentes dispositivos [Mena et al., 2020] (*Capítulo 3*).

— Se ha establecido un patrón de utilización común para todas las vertientes de WoTnectivity. Se ha estudiado la interfaz que establece el patrón de comunicación, la interfaz `IRequester`. Además, se han descrito un conjunto de ejemplos que permiten ilustrar cómo se implementan nuevos protocolos en WoTnectivity [Mena et al., 2020] (*Capítulo 3*).

— Se ha desarrollado una serie de ejemplos de uso de WoTnectivity en sus vertientes para HTTP y para KNX [Mena et al., 2020] (*Capítulo 3*).

5.2.3. Relaciones de Causalidad

Para definir un mecanismo comunicación entre Things que permita la interoperabilidad entre dispositivos IoT, se ha desarrollado un lenguaje específico de dominio que se basa en un modelo de relación causa/efecto. Este lenguaje hace uso de la potencia proporcionada por la Thing Description de la WoT [W3C, 2022b] para establecer una capa

[1] https://www.npmjs.com/package/wotnectivity-knx

de abstracción en la que se puedan declarar interacciones entre los dispositivos o los componentes software que representan a los dispositivos (*servients*). Las contribuciones vinculadas a las relaciones de causalidad se pueden resumir en las siguientes:

— Se ha desarrollado un lenguaje específico de dominio (DSL) para la definición de causas y efectos [Mena et al., 2021c] [Mena et al., 2023] (*Capítulo 3*).

— Se ha establecido un vínculo entre ese lenguaje y las interacciones de la Thing Description [Mena et al., 2021c] (*Capítulo 3*).

— Se ha desarrollado un servicio que permite la consulta de las relaciones de causalidad. Donde se ha desarrollado un motor de evaluación de las relaciones de causalidad [Mena et al., 2021c] (*Capítulo 3*).

— Se ha definido los distintos modos de ejecución de las relaciones de causalidad, centralizado y descentralizado [Mena et al., 2021c] (*Capítulo 3*).

— Se ha desarrollado un ejemplo de uso de las relaciones de causalidad. [Mena et al., 2021c] [Mena et al., 2023] (*Capítulo 3*).

5.2.4. Alta disponibilidad en Digital Dice

Se han desarrollado diferentes estrategias y mecanismos para garantizar la alta disponibilidad (HA) de los Digital Dice. El concepto de Digital Dice fue diseñado con el objetivo de ser un sistema altamente escalable, que es uno de los requisitos que se consideraron como necesarios para garantizar HA. Lo que, unido a estrategias de distinta índole, como la replicación de los servicios, la utilización de un sistema de caché, el uso de mallas de servicios, etc., nos permite definir Digital Dice como un sistema altamente disponible. Las contribuciones derivadas de este punto son las siguientes:

— Se han establecido los requisitos funcionales y no funcionales que debe cumplir un sistema para garantizar la alta disponibilidad [Mena et al., 2019b], [Mena et al., 2023] (*Capítulo 3*).

— Se ha comparado la propuesta realizada por Digital Dice con respecto a conexiones directas con dispositivos IoT e implementaciones referencia de la WoT (WoT Scripting API) en lo que respecta a los requisitos no funcionales [Mena et al., 2019b] [Mena et al., 2023] (*Capítulo 3*).

— Se han establecido estrategias para garantizar la alta disponibilidad en los Digital Dice. Más concretamente, estrategias de comunicación con el dispositivo físico, estrategias de replicación de microservicios y estrategias de comunicación entre microservicios [Mena et al., 2019b] [Mena et al., 2023] (*Capítulo 3*).

— Se ha desarrollado un sistema de monitorización de los microservicios de Digital Dice, y se han establecido diferentes métricas que nos permiten medir el estado de los distintos microservicios [Mena et al., 2023] (*Capítulo 3*).

— Se ha definido una malla de servicios que ayuda a implementar los distintos patrones de comunicación entre microservicios [Mena et al., 2023] (*Capítulo 3*).

— Se ha realizado una evaluación de la propuesta de Digital Dice a nivel de rendimiento para comparar el desempeño de diferentes dispositivos físicos cuando se utiliza y cuando no se utiliza Digital Dice [Mena et al., 2023] (*Capítulo 3*).

— Se ha desarrollado un escenario de ejemplo que maneja el proceso de recogida de basuras en una SmartCity, puesto que en este tipo de escenarios, la alta disponibilidad es un requisito deseable [Mena et al., 2023] (*Capítulo 3*).

5.3. Limitaciones de la propuesta

En esta sección, se identifican las principales limitaciones o restricciones que deben ser tenidas en cuenta sobre la propuesta desarrollada en este trabajo de tesis doctoral:

— El despliegue de Digital Dice puede ser complejo, ya que requiere de un despliegue de múltiples servicios. Para que un desarrollador pueda aplicar la propuesta presentada, debe conocer y ser capaz de utilizar las tecnologías como Docker, Kubernetes, etc. Por lo que, en un primer momento, puede ser un poco complejo para un desarrollador no experto.

— La transformación de modelo a texto para la conversión entre Digital Dice es parcial, por lo que la intervención por parte del usuario es necesaria para completar la implementación del sistema, lo cual requiere de conocimientos sobre el funcionamiento de Digital Dice. Por otro lado, la conversión se realiza mediante un sistema de reglas completamente a medida, por lo que no se trata de un proceso totalmente automático, siendo necesaria la intervención del experto. La implementación de las reglas de conversión con un sistema más generalista como EMF o Xtext, aumentaría la robustez del proceso de conversión, ya que el sistema de reglas que utilizamos para la conversión es muy sensible a los cambios en el modelo.

— El soporte de Digital Dice a nivel de tecnologías de comunicación es limitado, ya que solamente soporta los protocolos MQTT, HTTP, SSE, Websocket y KNX. Por lo que, si se quiere utilizar otro protocolo de comunicación, se deberá implementar una nueva variante de WoTnectivity.

— La implementación de `Virtualizer` y `UI` debe ser realizada por los desarrolladores que hacen uso de Digital Dice, ya que ahora mismo no existe un servicio que permita la generación automática de estos componentes.

— Los experimentos realizados sobre distintos entornos, como Ciudades Inteligentes, Hogar Digital o Edificios Inteligentes, son experimentos a nivel de prototipo, por lo que no se puede garantizar que los resultados obtenidos sean totalmente extrapolables a entornos con una gran cantidad de dispositivos. En el caso del experimento realizado sobre uno de los procesos Smart Cities, el sistema de recolección de basura, se ha realizado sobre un escenario virtual y aunque en este caso el número de

dispositivos controlados supera los 200, no se puede garantizar que los resultados obtenidos sean exactamente iguales a los obtenidos en un despliegue real.

El objetivo de la identificación de las limitaciones de la propuesta de este trabajo de tesis doctoral está orientado a localizar los distintos aspectos mejorables, para que estos sean abordados en trabajos de investigación futuros.

5.4. Líneas de investigación abiertas

A partir del trabajo de investigación desarrollado y de algunas de las limitaciones descritas sección anterior, han sido identificadas una serie de líneas de investigación que quedan abiertas y que pueden ser abordadas como trabajo de investigación futuro:

(a) En primer lugar, puede ser de utilidad desarrollar un servicio que permita el establecimiento de una transformación modelo-a-modelo para hacer compatible el modelo de datos de Digital Dice con modelos y plataformas de datos abiertos existentes. Este servicio establecerá interoperabilidad entre Digital Dice y dichas plataformas, lo que permitirá la integración de los datos generados por Digital Dice en repositorios de datos abiertos.

(b) Desarrollo automático de interfaces de usuario a partir de las definiciones existentes de un Digital Dice. Se pretende generar un lenguaje específico de dominio para definir los componentes la interfaz de usuario que permitan interaccionar con el Digital Dice. También deberá desarrollarse una transformación que genere de forma automática el código asociado.

(c) Desarrollar un sistema de recomendación para las interfaces de usuario definidas con el lenguaje propuesto en el punto anterior. Este sistema de recomendación puede sugerir al usuario la mejor interfaz de usuario para cada situación. Este sistema de recomendación se basará en el uso de técnicas de inteligencia artificial para la generación de modelos.

(d) Desarrollo de un sistema de generación de microservicios de virtualización que sea capaz de establecer su comportamiento a partir de los datos almacenados previamente en la base de datos y los datos que forman parte de la definición de un Digital Dice. Este sistema de generación de microservicios de virtualización permitirá crear microservicios de virtualización de forma automática, sin la intervención del experto o del usuario.

(e) Creación de extensiones de Digital Dice para plataformas abiertas de visualización de entornos IoT como Home Assitant. Estas extensión tendrán como objetivo la integración de Digital Dice con estas plataformas. De esta forma, se podrá conseguir la visualización de la información y la interacción con los Digital Dice en estas plataformas de manera remota.

(f) Estudio y creación de nuevas métricas personalizadas que permitan mejorar el rendimiento de las peticiones SSE cuando se levantan réplicas de los microservicios de Digital Dice.

5.5. Publicaciones derivadas de la Tesis

En esta sección, se presentan las publicaciones que han sido resultado de esta tesis doctoral, incluyendo artículos y contribuciones a actas de congresos internacionales y nacionales. Todas estas publicaciones han sido realizadas durante el período de investigación dedicado a la preparación de la tesis doctoral. En resumen, los resultados incluyen:

(a) 3 publicaciones en revistas internacionales indexadas en JCR: *IEEE Access* (Q1), *Computing* (Q2), and *Journal of Universal Computer Science* (Q3).

(b) 6 publicaciones en conferencias internacionales: *9th International Conference (MEDI 2019)*, *Current Trends in Web Engineering (ICWE 2020) International Workshops: WoT4H*, *IEEE 44th Annual Computers, Software, and Applications Conference, (COMPSAC 2020)*, *IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC 2021)*, *13th International Conference on Management of Digital EcoSystems (MEDES 2021)*, and *14th International Conference on Management of Digital EcoSystems (MEDES 2022)*.

(c) 3 publicaciones en conferencias nacionales: *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2019)*, *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021 - CEDI 2021)*, *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2022)*.

La siguiente lista detalla las publicaciones en orden cronológico:

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2019). Una arquitectura de microservicios para componentes digitales en la Web de las Cosas. Jornadas de Ingeniería de Software y Bases de Datos (JISBD). SISTEDES, Biblioteca Digital-Sistedes. Cáceres. Handle: 11705/JISBD/2019/025.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2019). Digital Dices: Towards the Integration of Cyber-Physical Systems Merging the Web of Things and Microservices. In Model and Data Engineering: 9th International Conference, MEDI 2019, Toulouse, France, October 28–31, 2019, Proceedings 9, 195-205. Springer International Publishing. ISBN: 978-3-030-32064-5. DOI: 10.1007/978-3-030-32065-2_14.

— **Mena, M.**, Corral, A., Iribarne, L., & Criado, J. (2019). A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things. IEEE Access, 7, 104577-104590. ISSN: 2169-3536. DOI: 10.1007/978-3-319-46963-8_18. [JCR SCIE, COMPUTER SCIENCE, INFORMATION SYSTEMS, Impact Factor (2019): 3.745, 35/156 in 2021, (**Q1**)]

— Criado, J., Boubeta-Puig, J., **Mena, M.**, Llopis, J. A., Ortiz, G., & Iribarne, L. (2020). Towards the Integration of Web of Things Applications Based on Service Discovery. In Current Trends in Web Engineering: ICWE 2020 International Workshops, WoT4H, Helsinki, Finland, 24-29. Cham: Springer International Publishing. ISBN: 978-3-030-65664-5. DOI: 10.1007/978-3-030-65665-2_3.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2020). WoTnectivity: A communication pattern for different Web of Things connection protocols. In 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC 2020), pp. 1059-1064. IEEE. DOI: 10.1109/COMPSAC48688.2020.0-133.

— Llopis, J. A., **Mena, M.**, Criado, J., & Iribarne, L. (2021). MI-FIWARE: A web component development method for FIWARE using microservices. In 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC 2021), pp. 1058-1065. IEEE. DOI: 10.1109/COMPSAC51774.2021.00144

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2021). Assembling the Web of Things and Microservices for the Management of Cyber-Physical Systems. J. Univers. Comput. Sci., 27(7), 734-754. ISSN: 0948-695X. DOI: 10.3897/jucs.70325. [JCR SCIE, COMPUTER SCIENCE, THEORY & METHODS, Impact Factor (2021): 1.056, 80/110 in 2021, (**Q3**)]

— **Mena, M.**, Criado, J., & Iribarne, L. (2021). Un lenguaje para definir componentes WoT basados en microservicios. Jornadas de Ingeniería de Software y Bases de Datos (JISBD), SISTEDES, CEDI, Biblioteca Digital-Sistedes. Málaga. Handle: 11705/JISBD/2021/044.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2021). Defining interactions of WoT servients with causality relations. In Proceedings of the 13th International Conference on Management of Digital EcoSystems, MEDES 2021, Hammamet, Tunisia, 112-119. DOI: 10.1145/3444757.3485102.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2022). Alta disponibilidad en una arquitectura de microservicios para IoT. Jornadas de Ingeniería del Software y Bases de Datos (JISBD), SISTEDES, Biblioteca Digital-Sistedes. Santiago de Compostela. Handle: 11705/JISBD/2022/5426.

— Llopis, J. A., **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2022). Towards a Discovery Model for the Web of Things. In Proceedings of the 14th International Conference on Management of Digital EcoSystems, MEDES 2022, Venice, Italy, 96-103. DOI: 10.1145/3508397.3564827.

— **Mena, M.**, Criado, J., Iribarne, L., Corral, A., Chbeir, R., & Manolopoulos, Y. (2023). Towards High-Availability Cyber-Physical Systems using a Microservice Architecture. Computing, Springer, 1–25. DOI: 10.1007/s00607-023-01165-x. [JCR SCIE, COMPUTER SCIENCE, COMPUTER SCIENCE, THEORY & METHODS, Impact Factor (2021): 2.420, 47/110 in 2021, (**Q2**)]

Conclusions and Future Work

Introduction

In this final chapter, we summarize the main contributions of the research work carried out in the doctoral thesis, as well as its limitations and the open research lines pending exploration in future works. The doctoral thesis work addressed several issues related to the development, management, and use of digital components that are part of solutions based on the Internet of Things (IoT) and Cyber-Physical Systems (CPS). Firstly, existing solutions in the literature were studied to analyze the opportunity and feasibility of developing a new digital component proposal. The proposed new solution, called *Digital Dice*, is an approach to the development of digital components that is based on breaking down each component into modules that group functionalities with common characteristics. This grouping facilitates development, maintenance, and reuse, but the main objective is to improve the management of digital component execution.

Breaking down digital components into smaller parts can be aided by a *microservices* approach, in which each of the different modules that make up a component can be encapsulated in a small-sized software artifact. This approach provides a more practical distribution of the operations that make up the components, provided that the functionality of each microservice is well-defined and accessible through an interface. In this regard, both the communication between the different parts that make up a digital component and the communication between digital components of a software solution must be carried out effectively and efficiently. To achieve this communication, as well as to meet the other common requirements of IoT applications and cyber-physical systems, a microservices architecture for digital components is proposed.

Both in the IoT domain, where the main objectives are based on enabling the connection of physical devices to the Internet, allowing data transmission through communication networks, and in the cyber-physical systems domain, where efforts are focused on improving the interaction between the physical elements of a system with the digital elements, there is a problem of *heterogeneity* in the technologies and protocols used by different devices. This heterogeneity, which poses a limitation on communication between the components of a system, can be reduced, hidden, or solved if an abstraction layer is built that allows for the standardization and use of these components.

One alternative available to software engineers to incorporate an abstraction layer in the construction of IoT solutions and cyber-physical systems is to use the recommendations of the Web of Things (WoT) from W3C. The set of recommendations from the WoT aims to improve interoperability between different devices and platforms existing in the IoT domain. This is achieved by applying a development methodology based on web technologies. This approach allows benefiting from the maturity of these technologies and applying the experience of developers in this area to reduce costs in the process of developing and maintaining IoT solutions and cyber-physical systems.

In this doctoral thesis work, a microservices architecture is proposed that allows for the management and use of solutions based on IoT devices and cyber-physical systems.

The developed architecture is based on WoT standards to reduce the existing fragmentation and heterogeneity in this type of systems, providing a homogeneous layer for the definition, implementation, and communication of its components. This architecture is based on a proposed digital component called Digital Dice, whose name comes from the following considerations. Digital Dice is a digital representation of a device (physical or virtual) that is part of a cyber-physical system or an IoT installation. This digital component consists of six parts or modules that encapsulate the functionality associated with one of the following *capabilities* or *facets*: communication controller, data manager, event manager, user interface, synchronizer with the physical device, and virtualizer.

The fact that it is a digital representation of a device, including the specific capability responsible for its virtualization, as well as the name Digital Dice, establishes an analogy with the concept of a digital twin. An approach based on the concept of a digital twin primarily focuses on digitally replicating a physical device in a virtual component (with its functionalities, features, and behaviour). However, this type of solution does not provide a multi-level approach that allows addressing the different capabilities of a device specifically, such as user interaction management, device event control, or resource provisioning according to demand, among other possible examples.

The proposed solution of Digital Dice presents several improvements over the concept of a digital twin. In addition to the virtualization and simulation capabilities (which also involve synchronization with the physical device), Digital Dice includes the additional capabilities mentioned earlier of communication control, data and event management, and user interface. The development of each of these facets is addressed through a microservices-based approach, which allows scaling only those capabilities that require more resources at a particular moment and managing each of the capabilities independently. In this way, it is possible to reduce the limitations presented by IoT applications or cyber-physical systems, such as reduced computing or storage capabilities, high response times, or a high number of concurrent requests, among other possible examples.

Because the proposed architecture and Digital Dice are based on WoT standards, there is a homogeneous communication layer that facilitates integration and interoperability and allows managing and using IoT and cyber-physical devices through an application programming interface (API). On the one hand, the recommendation that describes the WoT architecture allows defining the requirements and structure of the elements that participate in our approach, as well as the relationships and interactions between these elements. On the other hand, the recommendation that establishes the principles for describing the individual elements that are part of the architecture allows homogenizing the interaction with devices and their description.

This final chapter is structured into the following four sections. The first section describes the main contributions that the research work developed brings to the scientific community. In the second section, the main limitations of the proposal described in this thesis are identified. The third section presents the open research lines. Finally, the last section provides a list of the publications derived from the research work carried out.

Contributions of the Doctoral Thesis

In this section, we describe the contributions made throughout this doctoral thesis. The contributions are divided into the four milestones or the four main capabilities described in detail throughout this thesis: Digital Dice as a solution for device management, WoT-nectivity as a library for communication with devices, causality relations to describe interactions between devices, and Digital Dice as a high availability system.

Digital Dice

Throughout the doctoral thesis, Digital Dice has been presented as a microservices-based solution for managing and controlling IoT devices and cyber-physical systems. Digital Dice abstracts users and application developers from the technologies used by these devices to communicate with the outside world. The following contributions related to Digital Dice have been made:

— An architecture of microservices for managing IoT and CPS devices, Digital Dice, has been established and developed. One of the central components of Digital Dice is the use of the Web of Things, specifically the Thing Description, to declare the functionality of the devices through this standard, establishing the fundamental link between Thing Description and Digital Dice [Mena et al., 2019a] [Mena et al., 2019d] [Criado et al., 2020] [Llopis et al., 2021] [Llopis et al., 2022][Mena et al., 2023] (*Chapter 2*).

— The various parts that make up a Digital Dice, including microservices and auxiliary services, have been defined. To do this, we studied the architecture of Digital Dice along with the various layers that make it up [Mena et al., 2019a] [Mena et al., 2019d] [Mena et al., 2023] (*Chapter 2*).

— A domain-specific language (DSL) has been developed for defining the parts of a Digital Dice [Mena et al., 2023] (*Chapter 2*).

— A data model that Digital Dice uses to record interactions by users, as well as data generated by the devices that our DDs are representing, has been established [Mena et al., 2023] (*Chapter 2* and *Chapter 4*).

— A model-to-text (M2T) transformation process has been developed to establish the code base of the various components of a Digital Dice from the Thing Description that represents it [Mena et al., 2019c] [Mena et al., 2021b] (*Chapter 2*).

— The implementation of the various components that make up a Digital Dice has been explained in detail. At the same time, we have studied the deployment details of each of the components of Digital Dice and seen different examples of Digital Dice of different kinds through scenarios related to Smart Homes and Smart Cities [Mena et al., 2023] (*Chapter 4*).

WoTnectivity

In order to establish connections with the different protocols found in the IoT domain, WoTnectivity has been developed as a multiprotocol library for working with different types of protocols such as HTTP [Krishnamurthy and Rexford, 2001], MQTT [Hunkeler et al., 2008], WebSocket [Wang et al., 2013], and KNX [KNXAssoc, 2009]. The library provides an interface that allows the extension of the supported protocols. The contributions derived from WoTnectivity are as follows:

— A multiprotocol library has been developed for communication with IoT and CPS devices. This library provides a similar behaviour for communication with different IoT devices, regardless of the technology they use [Mena et al., 2020] (*Chapter 3*).

— The WoTnectivity library has been developed for Java and Node.js programming languages, with the latter showing a significant interest, with an average download rate of around 15 users per week[2]. The use of this library has been compared with other commonly used libraries for connecting to different devices [Mena et al., 2020] (*Chapter 3*).

— A common usage pattern has been established for all versions of WoTnectivity. The interface that establishes the communication pattern, `IRequester`, has been studied, and a set of examples have been described to illustrate how to implement new protocols in WoTnectivity [Mena et al., 2020] (*Chapter 3*).

— A series of usage examples of WoTnectivity have been developed for both HTTP and KNX versions [Mena et al., 2020] (*Chapter 3*).

Causality Relations

To define a communication mechanism between Things that allows interoperability among IoT devices, a specific domain language has been developed based on a cause/effect relationship model. This language leverages the power provided by the Thing Description of the WoT [W3C, 2022b] to establish an abstraction layer in which interactions between devices or the software components that represent the devices (*servients*) can be declared. The contributions related to causality relations can be summarized as follows:

— A specific domain language (DSL) has been developed for the definition of causes and effects [Mena et al., 2021c] [Mena et al., 2023] (*Chapter 3*).

— A link has been established between this language and the interactions of the Thing Description [Mena et al., 2021c] (*Chapter 3*).

— A service has been developed that allows the querying of causality relations. An evaluation engine for causality relations has been developed [Mena et al., 2021c] (*Chapter 3*).

[2]https://www.npmjs.com/package/wotnectivity-knx

— Different execution modes of causality relations have been defined, centralized and decentralized [Mena et al., 2021c] (*Chapter 3*).

— An example of the use of causality relations has been developed [Mena et al., 2021c] [Mena et al., 2023] (*Chapter 3*).

High Availability in Digital Dice

Different strategies and mechanisms have been developed to ensure high availability (HA) of Digital Dice. The concept of Digital Dice was designed with the objective of being a highly scalable system, which is one of the requirements considered necessary to guarantee HA. This, together with strategies of different kinds, such as service replication, the use of a cache system, the use of service meshes, etc., allows us to define Digital Dice as a highly available system. The contributions derived from this point are as follows:

— Functional and non-functional requirements that a system must meet to ensure high availability have been established [Mena et al., 2019b], [Mena et al., 2023] (*Chapter 3*).

— The proposal made by Digital Dice has been compared with direct connections to IoT devices and reference implementations of WoT (WoT Scripting API) in terms of non-functional requirements [Mena et al., 2019b] [Mena et al., 2023] (*Chapter 3*).

— Strategies to ensure high availability in Digital Dice have been established. More specifically, communication strategies with the physical device, microservice replication strategies, and communication strategies between microservices [Mena et al., 2019b] [Mena et al., 2023] (*Chapter 3*).

— A monitoring system for the microservices of Digital Dice has been developed, and different metrics have been established that allow us to measure the state of the different microservices [Mena et al., 2023] (*Chapter 3*).

— A service mesh has been defined to help implement the different communication patterns between microservices [Mena et al., 2023] (*Chapter 3*).

— An evaluation of the Digital Dice proposal has been carried out in terms of performance to compare the performance of different physical devices when Digital Dice is used and when it is not used [Mena et al., 2023] (*Chapter 3*).

— An example scenario that handles the garbage collection process in a SmartCity has been developed, since high availability is a desirable requirement in this type of scenario [Mena et al., 2023] (*Chapter 3*).

Limitations of the proposal

In this section, we identify the main limitations or constraints that must be taken into account regarding the proposal developed in this doctoral thesis:

— The deployment of Digital Dice can be complex, as it requires the deployment of multiple services. To apply the proposed approach, a developer must have knowledge and be able to use technologies such as Docker, Kubernetes, etc. Therefore, it may be challenging for a non-expert developer at first.

— The model-to-text transformation for the conversion between Digital Dice is partial, so user intervention is necessary to complete the implementation of the system, which requires knowledge of Digital Dice's operation. Moreover, the conversion is carried out through a completely bespoke rule-based system, so it is not a fully automated process, and expert intervention is necessary. Implementing the conversion rules with a more general system such as EMF or Xtext would increase the robustness of the conversion process, as the rule-based system we use for conversion is very sensitive to changes in the model.

— The support of Digital Dice for different communication technologies is limited, as it only supports MQTT, HTTP, SSE, Websocket, and KNX protocols. Therefore, if another communication protocol is desired, a new WoTnectivity variant must be implemented.

— The implementation of the `Virtualizer` and `UI` must be carried out by the developers who use Digital Dice, as there is currently no service that allows automatic generation of these components.

— The experiments carried out on different environments, such as Smart Cities, Digital Home, or Smart Buildings, are prototype-level experiments. Therefore, it cannot be guaranteed that the results obtained are entirely extrapolatable to environments with a large number of devices. In the case of the experiment conducted on one of the Smart Cities' processes, the garbage collection system, it was carried out on a virtual scenario, and although in this case, the number of controlled devices exceeds 200, it cannot be guaranteed that the results obtained are exactly the same as those obtained in a real deployment.

The objective of identifying the limitations of the proposal in this doctoral thesis is to locate the different areas that need improvement, so that they can be addressed in future research works.

Open Research Lines

Based on the research work carried out and some of the limitations described in the previous section, a series of open research lines have been identified that can be addressed as future research work:

(a) First, it may be useful to develop a service that allows for the establishment of a model-to-model transformation to make the Digital Dice data model compatible with existing open data models and platforms. This service will establish interoperability between Digital Dice and such platforms, allowing for the integration of data generated by Digital Dice into open data repositories.

(b) Automated development of user interfaces based on existing definitions in Digital Dice. A domain-specific language will be generated to define the user interface components that allow interaction with Digital Dice. A transformation that generates the associated code automatically will also be developed.

(c) Develop a recommendation system for user interfaces defined with the language proposed in the previous point. This recommendation system can suggest the best user interface for each situation to the user. This recommendation system will be based on the use of artificial intelligence techniques for model generation.

(d) Development of a virtualization microservice generation system that can establish its behaviour based on data previously stored in the database and data that is part of the definition of a Digital Dice. This virtualization microservice generation system will allow the automatic creation of virtualization microservices, without the intervention of an expert or user.

(e) Creation of Digital Dice extensions for open IoT environment visualization platforms such as Home Assistant. These extensions aim to integrate Digital Dice with these platforms. This will allow the visualization of information and interaction with Digital Dice on these platforms remotely.

(f) Study and creation of new customized metrics that allow for the improvement of SSE request performance when replicas of Digital Dice microservices are raised. Currently, the metrics used to measure the performance of SSE requests are not the most suitable for this type of request, as seen in Chapter 3.

Publications derived from the Thesis

In this section, we present the publications that have resulted from this doctoral thesis, including articles and contributions to international and national conference proceedings. All these publications have been made during the research period dedicated to the preparation of the doctoral thesis. In summary, the results include:

(a) 3 publications in JCR-indexed international journals: *IEEE Access* (Q1), *Computing* (Q2), and *Journal of Universal Computer Science* (Q3).

(b) 6 publications in international conferences: *9th International Conference (MEDI 2019)*, *Current Trends in Web Engineering (ICWE 2020) International Workshops: WoT4H*, *IEEE 44th Annual Computers, Software, and Applications Conference, (COMPSAC 2020)*, *IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC 2021)*, *13th International Conference on Management of*

Digital EcoSystems (MEDES 2021), and *14th International Conference on Management of Digital EcoSystems (MEDES 2022)*.

(c) 3 publications in national conferences: *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2019)*, *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021 - CEDI 2021)*, *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2022)*.

The following list details the publications in chronological order:

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2019). Una arquitectura de microservicios para componentes digitales en la Web de las Cosas. Jornadas de Ingeniería de Software y Bases de Datos (JISBD). SISTEDES, Biblioteca Digital-Sistedes. Cáceres. Handle: 11705/JISBD/2019/025.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2019). Digital Dices: Towards the Integration of Cyber-Physical Systems Merging the Web of Things and Microservices. In Model and Data Engineering: 9th International Conference, MEDI 2019, Toulouse, France, October 28–31, 2019, Proceedings 9, 195-205. Springer International Publishing. ISBN: 978-3-030-32064-5. DOI: 10.1007/978-3-030-32065-2_14.

— **Mena, M.**, Corral, A., Iribarne, L., & Criado, J. (2019). A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things. IEEE Access, 7, 104577-104590. ISSN: 2169-3536. DOI: 10.1007/978-3-319-46963-8_18. [JCR SCIE, COMPUTER SCIENCE, INFORMATION SYSTEMS, Impact Factor (2019): 3.745, 35/156 in 2021, (**Q1**)]

— Criado, J., Boubeta-Puig, J., **Mena, M.**, Llopis, J. A., Ortiz, G., & Iribarne, L. (2020). Towards the Integration of Web of Things Applications Based on Service Discovery. In Current Trends in Web Engineering: ICWE 2020 International Workshops, WoT4H, Helsinki, Finland, 24-29. Cham: Springer International Publishing. ISBN: 978-3-030-65664-5. DOI: 10.1007/978-3-030-65665-2_3.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2020). WoTnectivity: A communication pattern for different Web of Things connection protocols. In 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC 2020), pp. 1059-1064. IEEE. DOI: 10.1109/COMPSAC48688.2020.0-133.

— Llopis, J. A., **Mena, M.**, Criado, J., & Iribarne, L. (2021). MI-FIWARE: A web component development method for FIWARE using microservices. In 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC 2021), pp. 1058-1065. IEEE. DOI: 10.1109/COMPSAC51774.2021.00144

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2021). Assembling the Web of Things and Microservices for the Management of Cyber-Physical Systems. J. Univers. Comput. Sci., 27(7), 734-754. ISSN: 0948-695X. DOI: 10.3897/jucs.70325. [JCR SCIE, COMPUTER SCIENCE, THEORY & METHODS, Impact Factor (2021): 1.056, 80/110 in 2021, (**Q3**)]

— **Mena, M.**, Criado, J., & Iribarne, L. (2021). Un lenguaje para definir componentes WoT basados en microservicios. Jornadas de Ingeniería de Software y Bases de Datos (JISBD), SISTEDES, CEDI, Biblioteca Digital-Sistedes. Málaga. Handle: 11705/JISBD/2021/044.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2021). Defining interactions of WoT servients with causality relations. In Proceedings of the 13th International Conference on Management of Digital EcoSystems, MEDES 2021, Hammamet, Tunisia, 112-119. DOI: 10.1145/3444757.3485102.

— **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2022). Alta disponibilidad en una arquitectura de microservicios para IoT. Jornadas de Ingeniería del Software y Bases de Datos (JISBD), SISTEDES, Biblioteca Digital-Sistedes. Santiago de Compostela. Handle: 11705/JISBD/2022/5426.

— Llopis, J. A., **Mena, M.**, Criado, J., Iribarne, L., & Corral, A. (2022). Towards a Discovery Model for the Web of Things. In Proceedings of the 14th International Conference on Management of Digital EcoSystems, MEDES 2022, Venice, Italy, 96-103. DOI: 10.1145/3508397.3564827.

— **Mena, M.**, Criado, J., Iribarne, L., Corral, A., Chbeir, R., & Manolopoulos, Y. (2023). Towards High-Availability Cyber-Physical Systems using a Microservice Architecture. Computing, Springer, 1–25. DOI: 10.1007/s00607-023-01165-x. [JCR SCIE, COMPUTER SCIENCE, COMPUTER SCIENCE, THEORY & METHODS, Impact Factor (2021): 2.420, 47/110 in 2021, (**Q2**)]

Anexo A

Causality JSON-Schema

En este anexo se muestra el esquema JSON usado para validar las relaciones de Causalidad manejadas por los Digital Dice. Este esquema JSON es una herramienta de gran utilidad para validar la estructura de los datos y comprobar si los modelos de las relaciones de causalidad contienen datos precisos y bien formados.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
   "id": {
    "type": "integer"
   },
   "thingId": {
    "type": "string"
   },
   "isManaged": {
    "type": "boolean"
   },
   "self": {
    "type": "object",
    "properties": {
     "rel": {
      "type": "string"
     },
     "href": {
      "type": "string"
     },
     "type": {
      "type": "string"
     }
    },
    "required": [
     "rel",
     "href",
     "type"
    ]
   },
   "effects": {
    "type": "object",
    "patternProperties": {
     "^[A-Za-z0-9\\._%\\+-]+@[A-Za-z0-9\\.-]+\\.[A-Za-z]{2,6}$": {
      "type": "object",
      "properties": {
       "causes": {
        "type": "array",
        "items": [
         {
          "type": "object",
          "properties": {
           "link": {
            "type": "object",
            "properties": {
             "rel": {
              "type": "string"
             },
             "href": {
              "type": "string"
             },
             "type": {
              "type": "string"
             }
            },
            "required": [
```

```
              "rel",
              "href",
              "type"
             ]
            },
            "interactionType": {
             "type": "string"
            },
            "interaction": {
             "type": "string"
            },
            "origin": {
             "type": "string"
            }
           },
           "required": [
            "link",
            "interactionType",
            "interaction",
            "origin"
           ]
          },
          {
           "type": "object",
           "properties": {
            "link": {
             "type": "object",
             "properties": {
              "rel": {
               "type": "string"
              },
              "href": {
               "type": "string"
              },
              "type": {
               "type": "string"
              }
             },
             "required": [
              "rel",
              "href",
              "type"
             ]
            },
            "interactionType": {
             "type": "string"
            },
            "interaction": {
             "type": "string"
            },
            "origin": {
             "type": "string"
            }
           },
           "required": [
            "link",
            "interactionType",
            "interaction",
            "origin"
           ]
          },
          {
           "type": "object",
           "properties": {
            "link": {
             "type": "object",
             "properties": {
```

```
126                "rel": {
127                 "type": "string"
128                },
129                "href": {
130                 "type": "string"
131                },
132                "type": {
133                 "type": "string"
134                }
135               },
136               "required": [
137                "rel",
138                "href",
139                "type"
140               ]
141              },
142              "interactionType": {
143               "type": "string"
144              },
145              "interaction": {
146               "type": "string"
147              },
148              "origin": {
149               "type": "string"
150              }
151             },
152             "required": [
153              "link",
154              "interactionType",
155              "interaction",
156              "origin"
157             ]
158            }
159           ]
160          },
161          "hasOrder": {
162           "type": "boolean"
163          },
164          "window": {
165           "type": "integer"
166          },
167          "evalExpresion": {
168           "type": "string"
169          }
170         },
171         "required": [
172          "causes",
173          "hasOrder",
174          "window",
175          "evalExpresion"
176         ]
177        }
178       }
179    }
180    },
181    "required": [
182     "id",
183     "thingId",
184     "isManaged",
185     "self",
186     "effects"
187    ]}
```

Listing 1: Causality JSON Schema

Acrónimos

Acrónimos

ACG	Applied Computing Group
API	Application Programming Interface
CD	Continuous Delivery
CEP	Complex Event Processing
CI	Continuous Integration
COAP	Constrained Application Protocol
CPS	Cyber Physical System
CPU	Central Processing Unit
DB	Database
DD	Digital Dice
DSL	Domain Specific Language
DT	Digital Twin
FaaS	Function as a Service
HA	High Availability
HAS	High-Availability System
HPA	Horizontal Pod Autoscaler
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
IoT	Internet of Things
JSON	JavaScript Object Notation

LAN	Local Area Network
M2M	Model-To-Model (Transformation)
M2T	Model-To-Text (Transformation)
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MINECO	Ministry of Economy and Competitiveness
MOM	Message-Oriented-Middleware
MQ	Message Queue
MQTT	MQ Telemetry Transport
ORB	Object Request Broker
PaaS	Platform as a Service
PWA	Progressive Web Application
REST	Representational State Transfer
ROA	Resource Oriented Architecture
RPC	Remote Procedure Call
SaaS	Software as a Service
SE	Software Engineering
SOA	Service-Oriented Architecture
SOM	Service-Oriented Middleware
SSE	Server-Sent Events
SSL	Secure Sockets Layer
TD	Thing Description
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WoT	Web of Things
XML	eXtensible Markup Language
YAML	Yet Another Markup Language

Bibliografía

Bibliografía

[Al-Jaroodi and Mohamed, 2012] Al-Jaroodi, J. and Mohamed, N. (2012). Middleware is STILL everywhere!!! *Concurrency and Computation: Practice and Experience*, 24(16):1919–1926.

[Al-Sarawi et al., 2017] Al-Sarawi, S., Anbar, M., Alieyan, K., and Alzubaidi, M. (2017). Internet of Things (IoT) communication protocols. In *2017 8th International conference on information technology (ICIT)*, pages 685–690. IEEE.

[Ali et al., 2021] Ali, O., Ishak, M. K., and Bhatti, M. K. L. (2021). Emerging IoT domains, current standings and open research challenges: A review. *PeerJ Computer Science*, 7:e659.

[Almonaies et al., 2010] Almonaies, A. A., Cordy, J. R., and Dean, T. R. (2010). Legacy system evolution towards service-oriented architecture. In *International Workshop on SOA Migration and Evolution*, pages 53–62.

[Apache, 2023] Apache (2023). Apache JMeter. `https://jmeter.apache.org/`. Acc.: 2023-02-01.

[Aranha et al., 2013] Aranha, C., Both, C., Dearfield, B., Elkins, C. L., Ross, A., Squibb, J., Taylor, M., et al. (2013). *IBM WebSphere MQ V7. 1 and V7. 5 Features and Enhancements*. IBM Redbooks.

[Ashton, 1999] Ashton, K. (1999). That 'Internet of Things' Thing. *RFID journal*, 22.

[Bandyopadhyay and Sen, 2011] Bandyopadhyay, D. and Sen, J. (2011). Internet of things: Applications and challenges in technology and standardization. *Wireless personal communications*, 58(1):49–69.

[Barry and Gannon, 2003] Barry, D. K. and Gannon, P. J. (2003). *Web services and service-oriented architecture*. Morgan Kaufmann Publishers Inc.

[Biehl, 2016] Biehl, M. (2016). *RESTful Api Design*, volume 3. API-University Press.

[Boubeta-Puig et al., 2019] Boubeta-Puig, J., Díaz, G., Macià, H., Valero, V., and Ortiz, G. (2019). MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored petri nets. *Information Systems*, 81:267–289.

[Brown, 2019] Brown, E. (2019). *Web development with node and express: leveraging the JavaScript stack*. O'Reilly Media.

[Bucchiarone et al., 2020] Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., and Sadovykh, A. (2020). Microservices. *Science and Engineering. Springer*.

[Buchanan et al., 2020] Buchanan, S., Rangama, J., Bellavance, N., Buchanan, S., Rangama, J., and Bellavance, N. (2020). Deploying and using Rancher with Azure Kubernetes service. *Introducing Azure Kubernetes Service: A Practical Guide to Container Orchestration*, pages 79–99.

[Buchmann and Koldehofe, 2009] Buchmann, A. and Koldehofe, B. (2009). *Complex event processing*. Oldenbourg Wissenschaftsverlag GmbH München Germany.

[Burgueño et al., 2018] Burgueño, L., Boubeta-Puig, J., and Vallecillo, A. (2018). Formalizing complex event processing systems in Maude. *IEEE Access*, 6:23222–23241.

[Burgueño et al., 2019] Burgueño, L., Cabot, J., and Gérard, S. (2019). The future of model transformation languages: An open community. *Journal of Object Technology*, 18(3).

[Burns et al., 2022] Burns, B., Beda, J., Hightower, K., and Evenson, L. (2022). *Kubernetes: up and running*. O'Reilly Media.

[Calcote and Butcher, 2019] Calcote, L. and Butcher, Z. (2019). *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media.

[Cantelon et al., 2014] Cantelon, M., Harter, M., Holowaychuk, T., and Rajlich, N. (2014). *Node. js in Action*. Manning Greenwich.

[Chen et al., 2014] Chen, C. Y., Fu, J. H., Sung, T., Wang, P.-F., Jou, E., and Feng, M.-W. (2014). Complex event processing for the internet of things and its applications. In *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1144–1149. IEEE.

[Chodorow, 2019] Chodorow, K. (2019). *MongoDB: The Definitive Guide - Powerful and Scalable Data Storage*. O'Reilly Media, Sebastopol, CA, 2 edition.

[Cook, 2014] Cook, D. (2014). *Data Push Apps with HTML5 SSE: Pragmatic Solutions for Real-World Clients*. O'Reilly Media.

[Criado et al., 2020] Criado, J., Boubeta-Puig, J., Mena, M., Llopis, J. A., Ortiz, G., and Iribarne, L. (2020). Towards the integration of web of things applications based on service discovery. In *International Conference on Web Engineering*, pages 24–29. Springer.

[Davis and Davis, 2019] Davis, A. L. and Davis, A. L. (2019). Existing Models of Concurrency in Java. *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*, pages 5–13.

[De et al., 2011] De, S., Barnaghi, P., Bauer, M., and Meissner, S. (2011). Service modelling for the Internet of Things. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 949–955. IEEE.

[de la Torre et al., 2019] de la Torre, L., Chacon, J., Chaos, D., Dormido, S., and Sánchez, J. (2019). Using server-sent events for event-based control in networked control systems. *IFAC-PapersOnLine*, 52(9):260–265.

[Engels et al., 2001] Engels, D. W., Foley, J., Waldrop, J., Sarma, S. E., and Brock, D. (2001). The networked physical world: an automated identification architecture. In *Proceedings. The Second IEEE Workshop on Internet Applications. WIAPP 2001*, pages 76–77. IEEE.

[Espertech, 2022] Espertech (2022). EsperTech Inc. `https://www.espertech.com/`. Acc.: 2022-05-25.

[Estdale and Georgiadou, 2018] Estdale, J. and Georgiadou, E. (2018). Applying the ISO/IEC 25010 quality models to software product. In *Proc. 25th European Conference on Systems, Software & Services Process Improvement (EuroSPI)*, pages 492–503.

[Etzion and Niblett, 2011] Etzion, O. and Niblett, P. (2011). *Event processing in action*. Manning.

[Farhan et al., 2017] Farhan, L., Shukur, S. T., Alissa, A. E., Alrweg, M., Raza, U., and Kharel, R. (2017). A survey on the challenges and opportunities of the Internet of Things (IoT). In *2017 Eleventh International Conference on Sensing Technology (ICST)*, pages 1–5. IEEE.

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol–HTTP/1.1. Technical report.

[Gabor et al., 2018] Gabor, G. et al. (2018). Application of industrial PROFIBUS-DP protocol. In *Proc. International Conference & Exposition on Electrical & Power Engineering (EPE)*, pages 614–617.

[Giamas, 2022] Giamas, A. (2022). *Mastering MongoDB 6. x: Expert techniques to run high-volume and fault-tolerant database solutions using MongoDB 6. x*. Packt Publishing Ltd.

[Grez et al., 2019] Grez, A., Riveros, C., and Ugarte, M. (2019). A Formal Framework for Complex Event Processing. In *22nd International Conference on Database Theory (ICDT 2019)*, volume 127 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany.

[Guinard and Trifa, 2009] Guinard, D. and Trifa, V. (2009). Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, volume 15, page 8.

[Hazra et al., 2023] Hazra, A., Rana, P., Adhikari, M., and Amgoth, T. (2023). Fog computing for next-generation Internet of Things: Fundamental, state-of-the-art and research challenges. *Computer Science Review*, 48:100549.

[Hunkeler et al., 2008] Hunkeler, U., Truong, H. L., and Stanford-Clark, A. (2008). MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE.

[Ibrahim et al., 2021] Ibrahim, M. H., Sayagh, M., and Hassan, A. E. (2021). A study of how Docker Compose is used to compose multi-component systems. *Empirical Software Engineering*, 26:1–27.

[Iribarne et al., 2004] Iribarne, L., Troya, J. M., and Vallecillo, A. (2004). A Trading Service for COTS Components. *The Computer Journal*, 47(3):342–357.

[ISO/IEC, 2018] ISO/IEC (2018). ISO/IEC 30141:2018 - Internet of Things (IoT) — Reference Architecture. https://www.iso.org/standard/65695.html. Acc.: 2023-02-25.

[ITU-T, 2012] ITU-T (2012). Y.2060 : Visión general de la Internet de las cosas. https://www.itu.int/rec/T-REC-Y.2060-201206-I. Acc.: 2023-02-25.

[Jacobs et al., 2022] Jacobs, G., Konrad, C., Berroth, J., Zerwas, T., Höpfner, G., and Spütz, K. (2022). Function-oriented model-based product development. *Design Methodology for Future Products: Data Driven, Agile and Flexible*, pages 243–263.

[Jangla and Jangla, 2018] Jangla, K. and Jangla, K. (2018). *Docker Basics*. Springer.

[Jonas et al., 2019] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. (2019). Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.

[Kashyap et al., 2018] Kashyap, M., Sharma, V., and Gupta, N. (2018). Taking MQTT and NodeMcu to IOT: Communication in Internet of Things. *Procedia Computer Science*, 132:1611–1618.

[KNXAssoc, 2009] KNXAssoc (2009). *KNX System Specification, KNX Standard, v2.0*, volume 9.

[KNXAssoc, 2021] KNXAssoc (2021). KNX DataTypes. https://cutt.ly/o4S1Xcr. Acc.: 2023-02-01.

[Kosar et al., 2016] Kosar, T., Bohra, S., and Mernik, M. (2016). Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91.

[Krishnamurthy and Rexford, 2001] Krishnamurthy, B. and Rexford, J. (2001). *Web protocols and practice: HTTP/1.1, Networking protocols, caching, and traffic measurement*. Addison-Wesley Professional.

[Kuzminykh et al., 2019] Kuzminykh, I., Carlsson, A., and Yevdokymenko, M. (2019). A Performance Evaluation of Sensor Nodes in the Home Automation System based on Arduino. In *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S T)*, pages 511–516.

[Laghari et al., 2021] Laghari, A. A., Wu, K., Laghari, R. A., Ali, M., and Khan, A. A. (2021). A review and state of art of Internet of Things (IoT). *Archives of Computational Methods in Engineering*, pages 1–19.

[Larsson et al., 2020] Larsson, L., Tärneberg, W., Klein, C., Elmroth, E., and Kihl, M. (2020). Impact of etcd deployment on kubernetes, istio, and application performance. *Software: Practice and Experience*, 50(10):1986–2007.

[Libby, 2022] Libby, A. (2022). *Practical Svelte*. Springer.

[Light, 2017] Light, R. A. (2017). Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software*, 2(13).

[Llopis et al., 2021] Llopis, J. A., Mena, M., Criado, J., and Iribarne, L. (2021). MIFIWARE: A web component development method for FIWARE using microservices. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1058–1065. IEEE.

[Llopis et al., 2022] Llopis, J. A., Mena, M., Criado, J., Iribarne, L., and Corral, A. (2022). Towards a Discovery Model for the Web of Things. In *Proceedings of the 14th International Conference on Management of Digital EcoSystems*, pages 96–103.

[Locust, 2023] Locust (2023). Locust.io. `https://locust.io/`. Acc.: 2023-02-01.

[Longley et al., 2014] Longley, D., Kellogg, G., Lanthaler, M., and Sporny, M. (2014). Json-ld 1.0 processing algorithms and api. *W3C recommendation, W3C*.

[Lu et al., 2020] Lu, Q., Chen, L., Li, S., and Pitt, M. (2020). Semi-automatic geometric digital twinning for existing buildings based on images and CAD drawings. *Automation in Construction*, 115:103183.

[Madakam et al., 2015] Madakam, S., Lake, V., Lake, V., Lake, V., et al. (2015). Internet of Things (IoT): A literature review. *Journal of Computer and Communications*, 3(05):164.

[Malhotra, 2019] Malhotra, R. (2019). Java Reactive Development. *Rapid Java Persistence and Microservices: Persistence Made Easy Using Java EE8, JPA and Spring*, pages 251–265.

[Mangas and Alonso, 2019] Mangas, A. G. and Alonso, F. J. S. (2019). WOTPY: A framework for web of things applications. *Computer Communications*, 147:235–251.

[Mansour et al., 2021] Mansour, E., Chbeir, R., Arnould, P., Allani, S., and Salameh, K. (2021). Data management in connected environments. *Computing*, 103(6).

[Mdhaffar et al., 2017] Mdhaffar, A., Rodriguez, I. B., Charfi, K., Abid, L., and Freisleben, B. (2017). CEP4HFP: Complex event processing for heart failure prediction. *IEEE transactions on nanobioscience*, 16(8):708–717.

[Melnikov et al., 2023] Melnikov, A., Miller, D., and Kucherawy, M. (2023). IANA org.: IANA Media Types. `https://www.iana.org/assignments/media-types/media-types.xhtml`. Acc.: 2023-03-15.

[Mena et al., 2019a] Mena, M., Corral, A., Iribarne, L., and Criado, J. (2019a). A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things. *IEEE Access*, 7:104577–104590.

[Mena et al., 2021a] Mena, M., Criado, J., and Iribarne, L. (2021a). Un lenguaje para definir componentes WoT basados en microservicios. In *JISBD2021*. SISTEDES.

[Mena et al., 2019b] Mena, M., Criado, J., Iribarne, L., and Corral, A. (2019b). Alta disponibilidad en una arquitectura de microservicios para IoT. *Jornadas de Ingenierūa de Software y Bases de Datos (JISBD). Biblioteca Dūgital-Sistedes. Santiago de Compostela*.

[Mena et al., 2019c] Mena, M., Criado, J., Iribarne, L., and Corral, A. (2019c). Digital Dices: Towards the integration of cyber-physical systems merging the Web of Things and microservices. In *Proc. 9th International Conference on Model & Data Engineering (MEDI)*, pages 195–205.

[Mena et al., 2019d] Mena, M., Criado, J., Iribarne, L., and Corral, A. (2019d). Una arquitectura de microservicios para componentes digitales en la Web de las Cosas. *Jornadas de Ingenierūa de Software y Bases de Datos (JISBD). Biblioteca Dūgital-Sistedes. Caceres*.

[Mena et al., 2020] Mena, M., Criado, J., Iribarne, L., and Corral, A. (2020). WoTnectivity: A communication pattern for different Web of Things connection protocols. In *Proc. 44th IEEE Annual Computers, Software & Applications Conference (COMPSAC)*, pages 1059–1064.

[Mena et al., 2021b] Mena, M., Criado, J., Iribarne, L., and Corral, A. (2021b). Assembling the Web of Things and Microservices for the Management of Cyber-Physical Systems. *Journal of Universal Computer Science*, 27(7):734–754.

[Mena et al., 2021c] Mena, M., Criado, J., Iribarne, L., and Corral, A. (2021c). Defining Interactions of WoT Servients with Causality Relations. In *Proceedings of the 13th International Conference on Management of Digital EcoSystems*, MEDES '21, page 112–119, New York, NY, USA. Association for Computing Machinery.

[Mena et al., 2023] Mena, M., Criado, J., Iribarne, L., Corral, A., Chbeir, R., and Manolopoulos, Y. (2023). Towards high-availability cyber-physical systems using a microservice architecture. *Computing*, pages 1–25.

[Moreno et al., 2018] Moreno, N., Bertoa, M. F., Barquero, G., Burgueño, L., Troya, J., García-López, A., and Vallecillo, A. (2018). Managing uncertain complex events in web of things applications. In *International Conference on Web Engineering*, pages 349–357. Springer.

[Motte, Scott and jclbw, 2023] Motte, Scott and jclbw (2023). Dotenv - npmjs.com. `https://www.npmjs.com/package/dotenv`. Acc.: 2023-01-25.

[Nadareishvili et al., 2016] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media.

[Negri et al., 2019] Negri, E., Fumagalli, L., Cimino, C., and Macchi, M. (2019). FMU-supported simulation for CPS Digital Twin. *Procedia Manufacturing*, 28:201–206. 7th International conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2018).

[Ngu et al., 2016] Ngu, A. H., Gutierrez, M., Metsis, V., Nepal, S., and Sheng, Q. Z. (2016). IoT middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20.

[Nickoloff and Kuenzli, 2019] Nickoloff, J. and Kuenzli, S. (2019). *Docker in action*. Simon and Schuster.

[Pacheco, 2018] Pacheco, V. F. (2018). *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*. Packt Publishing Ltd.

[Perera et al., 2015] Perera, C., Liu, C. H., and Jayawardena, S. (2015). The emerging internet of things marketplace from an industrial perspective: A survey. *IEEE transactions on emerging topics in computing*, 3(4):585–598.

[Perna et al., 2022] Perna, G., Trevisan, M., Giordano, D., and Drago, I. (2022). A first look at HTTP/3 adoption and performance. *Computer Communications*, 187:115–124.

[Perrey and Lycett, 2003] Perrey, R. and Lycett, M. (2003). Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, pages 116–119. IEEE.

[Piedad and Hawkins, 2001] Piedad, F. and Hawkins, M. (2001). *High Availability: Design, Techniques, and Processes*. Prentice Hall, Upper Saddle River, NJ.

[Pivoto et al., 2021] Pivoto, D. G., de Almeida, L. F., da Rosa Righi, R., Rodrigues, J. J., Lugli, A. B., and Alberti, A. M. (2021). Cyber-physical systems architectures for industrial internet of things applications in Industry 4.0: A literature review. *Journal of manufacturing systems*, 58:176–192.

[Razzaque et al., 2016] Razzaque, M. A., Milojevic-Jevric, M., Palade, A., and Clarke, S. (2016). Middleware for Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(1):70–95.

[Rose et al., 2012] Rose, L. M., Matragkas, N., Kolovos, D. S., and Paige, R. F. (2012). A feature model for model-to-text transformation languages. In *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, pages 57–63. IEEE.

[Roy, 2017] Roy, G. M. (2017). *RabbitMQ in depth*. Simon and Schuster.

[Ruta et al., 2017] Ruta, M., Scioscia, F., Loseto, G., and Di Sciascio, E. (2017). KNX: A Worldwide Standard Protocol for Home and Building Automation: State of the Art and Perspectives. *Industrial Communication Technology Handbook*, pages 58–1.

[Schweizer, Jochen, 2023] Schweizer, Jochen (2023). Express-prom-bundle - npmjs.com. `https://www.npmjs.com/package/express-prom-bundle`. Acc.: 2023-01-25.

[Sciullo et al., 2019] Sciullo, L., Aguzzi, C., Di Felice, M., and Cinotti, T. S. (2019). WoT Store: Enabling things and applications discovery for the W3C Web of Things. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–8. IEEE.

[Shetty, 2017] Shetty, S. (2017). How to use Digital Twins in your IoT strategy. https://www.gartner.com/smarterwithgartner/how-to-use-digital-twins-in-your-iot-strategy. Acc.: 2022-11-25.

[Singh et al., 2021] Singh, M., Fuenmayor, E., Hinchy, E. P., Qiao, Y., Murray, N., and Devine, D. (2021). Digital Twin: Origin to future. *Applied System Innovation*, 4(2):36.

[Slange, 2013] Slange (2013). Internet of things - architecture - IOT-A: Internet of things architecture. https://www.iot-a.eu/. Acc.: 2023-02-25.

[Sneps-Sneppe and Namiot, 2016] Sneps-Sneppe, M. and Namiot, D. (2016). On physical web models. In *2016 International Siberian Conference on Control and Communications (SIBCON)*, pages 1–6. IEEE.

[Soni and Makwana, 2017] Soni, D. and Makwana, A. (2017). A survey on mqtt: a protocol of internet of things (IoT). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, pages 173–177.

[Sporny et al., 2020] Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., and Lindström, N. (2020). JSON-LD 1.1. *W3C Recommendation, Jul*.

[Statista, 2023] Statista (2023). Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030. `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide`. Acc.: 2023-01-25.

[Stenberg, 2014] Stenberg, D. (2014). HTTP2 explained. *ACM SIGCOMM Computer Communication Review*, 44(3):120–128.

[Sun et al., 2019] Sun, A. Y., Zhong, Z., Jeong, H., and Yang, Q. (2019). Building complex event processing capability for intelligent environmental monitoring. *Environmental Modelling & Software*, 116:1–6.

[Swagger, 2022] Swagger (2022). OpenAPI spec. https://swagger.io/specification/. Acc.: 2023-03-11.

[Turnbull, 2018] Turnbull, J. (2018). *Monitoring with Prometheus*. Turnbull Press.

[Uhlemann et al., 2017] Uhlemann, T. H.-J., Lehmann, C., and Steinhilper, R. (2017). The Digital Twin: Realizing the Cyber-Physical Production System for Industry 4.0. *Procedia CIRP*, 61:335–340. The 24th CIRP Conference on Life Cycle Engineering.

[Union, 2005] Union, I. T. (2005). ITU Internet Report 2005: The Internet of Things.

[W3C, 2022a] W3C (2022a). Web of Things. `https://www.w3.org/WoT/`. Acc.: 2022-05-25.

[W3C, 2022b] W3C (2022b). Web of Things – Thing Description. `https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/`. Acc.: 2022-05-25.

[W3C, 2023a] W3C (2023a). Web of Things – Architecture. `https://www.w3.org/TR/wot-architecture10/`. Acc.: 2023-01-20.

[W3C, 2023b] W3C (2023b). Web of Things – Discovery. `https://www.w3.org/TR/wot-discovery/`. Acc.: 2023-01-20.

[W3C, 2023c] W3C (2023c). Web of Things – Profile. `https://www.w3.org/TR/wot-profile/`. Acc.: 2023-01-20.

[W3C, 2023d] W3C (2023d). Web of Things – Scripting API. `https://www.w3.org/TR/wot-scripting-api/`. Acc.: 2023-01-20.

[W3C, 2023e] W3C (2023e). Web of Things – Security and Privacy Guidelines. `https://www.w3.org/TR/wot-security/`. Acc.: 2023-01-20.

[W3C, 2023f] W3C (2023f). Web of Things (WoT) - Binding Templates. `https://www.w3.org/TR/wot-binding-templates/`. Acc.: 2023-02-01.

[Wang et al., 2013] Wang, V., Salim, F., and Moskovits, P. (2013). *The definitive guide to HTML5 WebSocket*, volume 1. Springer.

[WebThings, 2022] WebThings (2022). WebThings. `https://webthings.io`. Acc.: 2022-05-25.

[Wilson, Doug, 2023] Wilson, Doug (2023). Morgan - npmjs.com. `https://www.npmjs.com/package/morgan`. Acc.: 2023-02-25.

[Wilson, Doug and Goode, Troy, 2023] Wilson, Doug and Goode, Troy (2023). Cors - npmjs.com. `https://www.npmjs.com/package/cors`. Acc.: 2023-01-25.

[Zhang et al., 2014] Zhang, Y., Duan, L., and Chen, J. L. (2014). Event-driven soa for iot services. In *IEEE international conference on services computing*, pages 629–636. IEEE.

[Zhang et al., 2020] Zhang, Y., Tian, G., Zhang, S., and Li, C. (2020). A Knowledge-Based Approach for Multiagent Collaboration in Smart Home: From Activity Recognition to Guidance Service. *IEEE Transactions on Instrumentation and Measurement*, 69(2):317–329.

[Zikria et al., 2021] Zikria, Y. B., Ali, R., Afzal, M. K., and Kim, S. W. (2021). Next-generation internet of things (IoT): Opportunities, challenges, and solutions. *Sensors*, 21(4):1174.

Este archivo ha sido generado usando LaTeX.

Todas las Figuras y Tablas de este archivo son originales

Una arquitectura de microservicios para componentes digitales en la Web de las Cosas.

Manel Mena Vicente
Departamento de Informática
Grupo de Investigación de Informática Aplicada (TIC-211)
Universidad de Almería
Almería, Mayo, 2023

http://acg.ual.es