Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DEL SOFTWARE

Framework para integración de redes de sensores y actuadores con plataformas de Internet de las Cosas

---

Alumno: Diego Fidalgo Moreno

Tutor: Benjamín Sahelices Fernández

A mis padres

Agradecimientos

Llega el momento de agradecer, y no son pocas las personas que han sido fundamentales para que hoy pueda estar aquí, firmando este documento.

Quiero comenzar por mi familia, agradeciéndoles a todos ellos su paciencia y apoyo; con especial mención a mi padre, Fernando, que es quien lleva más tiempo esperando este momento; y a Emilia, mi madre.

A los profesores de la escuela, y en especial a Félix y a María Luisa; personas sin las cuáles ahora, probablemente, no fuera informático. Agradecer a Benjamín que acogiera este trabajo, y que me haya guiado hasta este punto en tiempo récord. A Yania, por su amabilidad y gran corazón y dedicación, por ayudarme a plantear parte del trabajo.

A los compañeros y compañeras de la carrera, y en especial a Luis Javier, a David Soler y a Christiam Mansutti, grandes amigos sin los cuáles hoy no estaría escribiendo esto.

A los técnicos de la escuela, con especial mención a Javier Rodríguez y Javier Ramos, que para mí han sido grandes maestros, y a los que debo un montón de conocimiento.

Al Grupo Universitario de Informática y a los compañeros y compañeras que lo formamos, por haber sido estos años un lugar fundamental donde alimentar la pasión por lo que hacemos, compartir informática y amistad; y adquirir una serie de competencias fundamentales en mi desarrollo profesional.

A Ester, por su apoyo y paciencia. A mis amigos y amigas, por estar ahí siempre.

Y por último, y no por ello menos importante, a todo Argotec, a cada uno de los compañeros; con especial atención en Francisco Huidobro, que me ha guiado durante el desarrollo del trabajo, a Dani, Oliver; por su ayuda en los temas de bajo nivel, a Carlos, Albano y Miguel; por el soporte hardware; y a Antonio, que ha arrojado luz sobre varios temas, y me ha ayudado a ser un poco más práctico.

Probablemente, me deje a otras muchas grandes personas.

A todos vosotros, ¡muchísimas gracias!, de corazón.

Resumen

En un desarrollo de Internet de las Cosas, los “gateways” son una parte fundamental del sistema. Éstos permiten gestionar la interconexión de hardware (sensores, actuadores) con software de más alto nivel (aplicaciones concretas de negocio, servicios en la “nube”, etc.).

La implementación de este tipo de componente requiere conocimiento especializado en protocolos de comunicación, gestión de dispositivos y configuración de hardware. Típicamente, la integración del hardware con otras plataformas forma parte de un desarrollo ad-hoc que proporcionan o bien los fabricantes del hardware, o bien los desarrolladores del software de más alto nivel.

IoTGate nace en este punto para aportar una manera más ágil y cómoda de crear y gestionar una red de sensores y actuadores, permitiendo reutilizar todo el trabajo no generalizado en el desarrollo de soluciones ad-hoc de integración.

IoTGate es un framework que fundamentalmente proporciona implementaciones de diversos protocolos de comunicación -a diferentes niveles de abstracción-, una conceptualización compartida que permite interactuar -en términos lógicos- homogéneamente con todos los dispositivos que conformen la red, un servicio de persistencia de datos y un conjunto de herramientas para integrar estas funcionalidades rápidamente con software de más alto nivel de abstracción.

Abstract

Developing Internet of Things applications, the gateways are an essential part of the system. They enable the managing sensor and actuator networks conformed by many different communication hardware devices, in order to integrate them with another software, i.e specific business applications, services in the “ cloud ”.

Implementing this type of component requires specialized knowledge about hardware: communication protocols, device management and configuration of different devices. Typically, hardware integration with other platforms is part of an ad-hoc development that is provided by hardware manufacturers or by higher-level of abstraction software developers.

At this point, IoTGate is born to provide a more flexible and convenient way to create and manage sensors and actuators networks, allowing to reuse all non-generalized work from ad-hoc integrations solutions.

IoTGate is a framework that primarily gives implementations of various communication protocols -at different levels of abstraction-, a shared conceptualization that allows -in logical terms- interact homogeneously with all devices that conform the network; service and data persistence; and a set of tools to integrate quickly these features with higher level of abstraction software.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

Internet de las Cosas. Año 2016. Como veremos a lo largo de este trabajo, Internet de las Cosas es un concepto que va más allá de una simple moda. Ha venido para quedarse, y está llegando poco a poco.

Después de dos años empleado en Argotec, donde trabajamos principalmente en proyectos englobados dentro del concepto "IoT", hemos experimentado, disfrutado y sufrido mucho hardware, muchas tecnologías, y muchas integraciones con software de terceros.

La idea de este trabajo -llamémoslo IoTGate- lleva varios años en el tintero del i+D de Argotec, y por fin llegó el momento. Hay mucho hardware, muchas tecnologías para la comunicación, una gran diversidad y único ganador claro: el "depende de para qué". Un mundo lleno de oportunidades, que en ningún caso queremos perdernos.

La idea fundamental de este trabajo es poder integrar redes de sensores y actuadores rápidamente con otros servicios de Internet de las Cosas. Para ello, proporcionaremos una serie de herramientas que nos permitirán ser compatibles con muchos productos diferentes, y otra serie de utilerías que por experiencia, suelen ser necesarias en muchas integraciones.

1.2. Objetivos

El objetivo fundamental de este trabajo es desarrollar un framework que permita fácilmente crear y gestionar redes de sensores y actuadores multitecnología, para su posterior integración con plataformas de Internet de las Cosas. Después, el siguiente objetivo del trabajo es crear pequeños desarrollos que hagan uso de él. Los objetivos fundamentales de este framework son:

1. Proporcionar una arquitectura que permita integrar fácilmente dispositivos hardware nuevos orientados a ser utilizados en escenarios de Internet de las Cosas.
2. Ofrecer una conceptualización compartida que sea de base compatible con muchos escenarios posibles, a la par que simple de entender; en el contexto de las redes de sensores y actuadores.
3. Permitir la rápida integración con otros programas/plataformas, y el desarrollo de nuevas soluciones IoT.

Una vez tengamos lista la versión de entrega, presentaremos pequeños programas que harán uso del framework, consiguiendo por una parte demostrar sus funcionalidades y por otra enseñar como se utiliza.

1.3. Herramientas empleadas

Para desarrollar este trabajo, utilizaré las siguientes herramientas:

- Pycharm: es un IDE para programar con Python [10]. Incorpora, entre otras cosas:
  - Integración con depurador de Python.
  - Un "profiler"
  - Auto-completado de sintaxis, detección de errores antes del tiempo de ejecución, ayudas en formato para documentación
  - Dibujado de diagramas (Jerarquía de herencia, esquema relacional de base de datos)
  - Herramientas para trabajar con varios frameworks célebres de Python
  - Herramientas para trabajar con diferentes entornos virtuales del intérprete de Python (tanto locales como remotos)
  - Herramientas para facilitar el auto-despliegue
  - Integración son diferentes softwares de control de versiones (git, svn, mercurial, etc.)
- Stash (git): Stash es un servidor de repositorios de control de versiones basado en git [8].
- Jira: JIRA es una aplicación web para realizar gestión operativa de proyectos [9].
- Diferentes versiones de Raspberry Pi (principalmente la RPi2 y la RPi3): la Rapsberry Pi es un SBC (Single Board Computer) de tamaño reducido y muy bajo coste; ideal para el proyecto por presentar una GPIO que la permite ser ampliada en cuanto a hardware se refiere. [14]
- Hardware fabricado en Argotec:

    1. Placa expansión para Raspberry Pi, que permite la conexión de componentes a la GPIO de la RPi.
    2. Módulo transceptor MRF24J40MC adaptado a la placa anterior. [60]
    3. Diferentes dispositivos basados en Netbows, un producto de Argotec. [16]
    4. Dispositivos de otros fabricantes, como el MCP4728, que es un conversor analógio-digital de cuatro canales, fabricado por Microchip [15]

- Máquina virtual con Debian 8.5 (amd64) proporcionada por la escuela.
- Kile y LaTeX: Kile es un editor que proporciona herramientas para facilitar la edición de textos con LaTeX. LaTeXes un célebre editor de textos. [12] [11]
- Astah: es un editor de diagramas UML 2. [13]
- Gimp e Inkscape: Gimp es un programa de edición de imágenes [17], e Inkscape es un programa para dibujar con vectores [18]. Ambos han sido utilizados para retocar y dibujar algunas de las figuras que se presentan durante el trabajo.

Se han utilizado muchas herramientas más, como sqlitebrowser, phpmyadmin, phppgadmin para disponer de una manera amable de visualizar datos de bases de datos durante el proceso de desarrollo, otros editores de texto como vim y geany, el navegador web Firefox... Indispensables todos ellos para el desarrollo de este trabajo.

Capítulo 2

Estado del arte. Contexto tecnológico

2.1. Introducción

Durante el desarrollo de este capítulo, veremos los siguientes puntos:

1. Comenzaremos explicando que se entiende por Internet de las Cosas y para qué sirve.
2. Continuaremos analizando el contexto tecnológico en el necesitamos desenvolvernos. Definiremos conceptos básicos que utilizaremos a lo largo de todo el trabajo y diferentes tecnologías y teorías que darán soporte a los objetivos que pretendemos.
3. Para finalizar, estudiaremos rápidamente el estado del mercado actual, con el objetivo de encontrar diferentes soluciones a los retos que plantea el concepto IoT.

2.2. ¿Qué es Internet of Things (IoT)?

Durante el desarrollo de este capítulo se introducirá el concepto de internet de las cosas. Estudiaremos rápidamente el concepto en el siguiente orden:

1. Definición genérica del concepto
2. Áreas de aplicación, ¿para que sirve internet de las cosas?, ¿en qué áreas se aplica?
3. Arquitectura típica en un desarrollo de internet de las cosas, ¿qué partes generales están presentes en la mayoría de desarrollos de internet de las cosas actuales?

Después de esta introducción, veremos que soluciones existen en el mercado para suplir las necesidades que este trabajo de fin de grado pretende resolver.

2.2.1. Conceptos básicos

El término de internet de las cosas (en inglés Internet of Things, abreviado IoT) es un concepto que se refiere a la interconexión de diferentes objetos cotidianos a internet.

Actualmente ya existen muchos dispositivos que se conectan a internet, como ordenadores personales y smartphones. Desde el punto de vista de conexión a internet "tradicional", a muy grandes rasgos, contamos con una serie de aparatos que se conectan a internet para ofrecernos información, interactuando nosotros con ellos. La consulta de esa información es proporcionada por una serie de máquinas también conectadas, previa interacción por parte de las personas que distribuyen la información. Por supuesto, la red de redes es mucho más que eso; la intención de este párrafo es constatar que el concepto de internet de las cosas se refiere a otro tipo de interconexión de dispositivos: detrás de los clientes conectados a la red no hay personas, hay aparatos. Aparatos que miden parámetros del entorno mediante sensores y/o realizan acciones sobre éste mediante actuadores. Estos aparatos transmiten datos y esperan peticiones, se pueden comunicar unos con otros y en definitiva, pueden llegar a revelarnos una información de interés.

Los ingredientes para crear sistemas que se ciñan a este concepto llevan existiendo muchos años. Telemetría, telecontrol, análisis de datos... nada de esto es nuevo. ¿Por qué es entonces una revolución ahora?

La respuesta a esta pregunta es relativamente sencilla. Los avances en comunicaciones inalámbricas, en el hardware (capacidades, tamaño, consumo energético...) y en la computación en "nube" han hecho posible que los desarrollos de este tipo de sistemas sean más baratos y fáciles de hacer que nunca.

El valor añadido fundamental que nos brinda este concepto es poder medir cosas de una manera distinta, masiva, increíble. Y con estos datos obtenidos, podemos generar información; con esta información conocimiento; y en base a este conocimiento actuar en consecuencia. En definitiva, dotar de una "inteligencia" artificial a las cosas y conseguir principalmente dos cosas: obtener conocimiento para tomar mejores decisiones - y así ser más eficientes y prácticos- y proporcionar "autonomía" a las cosas, es decir, crear sistemas que actúan en base a la información que ellos mismos recogen y procesan, con la mínima intervención humana.

2.2.2. Áreas de aplicación

Una vez introducido el concepto, vamos a detenernos a hablar de áreas concretas de aplicación, para dar respuesta a una pregunta muy simple: ¿Para qué sirve todo esto?

Hogar (domótica)

Dentro del término de domótica, nos encontramos aplicaciones de internet de las cosas para [19]:

1. **Programación y ahorro energético**: climatización, control de toldos y persianas, mejor control de consumo -colocando un sistema adecuado podríamos conocer cuántos litros de agua gastamos cuando ponemos el lavavajillas o nos duchamos-, cuanta energía consume cada dispositivo de la casa... En base a esta información, podemos programar diferentes cosas, como que temperaturas queremos tener la siguiente semana en las zonas térmicas de nuestro hogar, reducción de intensidad lumínica en base a datos de sensores de luminosidad, recoger el toldo automáticamente si hace mucho aire, etc.
2. **Comodidad**: telecontrol de cualquier dispositivo, gestión de ocio electrónico
3. **Seguridad**: detección de intrusos, cierre automático de persianas y puertas, alarmas de incendio, alertas médicas, cámaras ip, etc.
4. **Accesibilidad**: aplicaciones que mejoren la autonomía de personas con limitaciones funcionales o discapacidad.

Ciudades "inteligentes"

El área de las ciudades "inteligentes" se centra principalmente en soluciones que apoyen a la sostenibilidad de éstas. Ejemplos de aplicación pueden ser:

1. Monitorización del tráfico: informar al ciudadano del estado del tráfico, consiguiendo así que elija el itinerario más óptimo, llegue antes y a la larga, contamine menos; sistemas de autoregulación de semáforos; de apoyo en la toma de decisiones (ampliación de vías de circulación, por ejemplo). [22]
2. Gestión energética: alumbrado público, medición de consumo energético y toma de decisiones (prioridad en provisión de energía mediante fuentes de energía renovables, detección de problemas, etc.) [23]
3. Gestión de residuos: medición de peso de los contenedores para optimizar la recogida, monitorización y gestión eficiente de la flota de camiones, etc.
4. Gestión medioambiental: información en tiempo real sobre polución, calidad del aire, del agua, etc. [24]
5. Transporte público [25]

Agricultura y ganadería

Los sistemas de internet de las cosas orientados al sector agrícola suponen un desafío de futuro muy interesante. El sector agrícola se enfrenta al reto de alimentar a las 9.6 billones de personas que según FAO (Organización de las Naciones Unidas para la Alimentación y la Agricultura) habrá en el planeta para el año 2050. [20]

Este incremento en la población supondrá que la producción se tenga que incrementar en un 70 %. Teniendo en cuenta, además, que el consumo de agua dulce destinado a la agricultura supone actualmente el 70 % del agua disponible en el planeta, y que entre otras cosas, el cambio climático amenaza con cambiar el ciclo de vida de varias plantas y animales (todo esto según la ONU); el reto se torna excesivamente complejo.

En este sector encuentra también su espacio internet de las cosas, con lo que se conoce como "Agricultura de precisión" o "agricultura inteligente". Las soluciones en este pretenden optimizar la producción de explotaciones agrícolas mediante la observación de todos los parámetros posibles y el procesamiento de éstos.

Figura 2.1: Qbic en pruebas: dispositivo IoT orientado a dar soluciones en agricultura

En cuanto a la ganadería, la posibilidad de poder medir y procesar los datos medidos desemboca en una gestión precisa de los rebaños: seguimiento de la salud de los animales, ubicación de estos, eventos de reproducción, vigilancia contra los robos, y una recolección de informa-

ción que permite en base a resultados en el pasado tomar mejores decisiones a futuro. [21]

Otras áreas de aplicación

Para concluir esta sección, vamos a presentar un último listado de áreas clave:

1. Medio ambiente: en las soluciones IoT ya presentadas (hogar, ciudades, agricultura y ganadería) podemos apreciar que muchas de las ideas -por no decir todos- se orientan a conseguir una gestión más eficiente de los recursos para conseguir, entre otras cosas, preservar mejor el medio ambiente. [24]

2. Energía: la innovación en el sector energético pasa también por proyectos de internet de las cosas. El almacenamiento de energía, los contadores inteligentes y los sistemas de pronósticos (demanda, producción), entre otros, tienen un fuerte componente de comunicación entre dispositivos y análisis de información. [23]

3. Automoción: cada día hay más vehículos en el mercado con capacidades de comunicación con internet. Los vehículos conectados a internet pueden proporcionar mucha valiosa información: información sobre la conducción y como mejorarla, información de diagnóstico, ubicación, emisión de gases... [26]

4. Salud: en el campo de salud, internet de las cosas busca prevenir enfermedades y mantener en control los padecimientos crónicos que necesitan estar monitorizados todo el tiempo. También pone a disposición de todos nosotros soluciones que nos permiten conocer más sobre sobre nosotros; mediante relojes inteligentes y pulseras deportivas (wearables). [27]

Llegados a este punto del desarrollo, es fácil darse cuenta de que "internet de las cosas" es un concepto que se puede aplicar en muchísimas áreas; todas aquellas en las que observar, obtener datos, en base a éstos información y en base a ésta conocimiento sea una actividad que aporte un valor añadido.

2.3. Análisis del contexto tecnológico

Para el análisis del contexto, se ha tomado como principal referencia el libro Internet Of Things: a Hands-On Approach [1].

Todos los sistemas IoT deberían contar con estos bloques para poder disponer de las siguientes características esenciales:

1. Toma de medidas de sensores: sin esto, no hay datos.

2. Realización de acciones sobre actuadores.

3. Contextualización de recursos: entendiendo como recursos los sensores y actuadores, es esencial otorgar un contexto (identificación unívoca, tiempo) tanto a los datos como a los actuadores. Si no sabemos cuando y dónde se midió una temperatura; o no sabemos que relé estamos accionando, todo esto no sirve para nada.
4. Gestión de red: para facilitar el orden, aceptar cambios en el contexto, y poder realizar un consumo ordenado de los datos, se debe poder realizar tareas de gestión sobre la red.

Con estas características en mente, presentamos los siguientes bloques de funcionalidades:

1. Dispositivos: Hablaremos en detalle de ellos en la siguiente sección, "Las cosas en IoT" (2.3.1).
2. Comunicación: Hablaremos en detalle de ellos en las secciones "Protocolos en IoT" y "Modelos de comunicación" (secciones 2.3.2 y 2.3.3)
3. Servicios: un sistema IoT suele proporcionar servicios para monitorizar el estado de la red de sensores y actuadores. Entre estos se encuentran servicios de monitorización, servicios de publicación/subscripción, servicios de control (actuadores), servicios para descubrimiento de dispositivos en red, etc.
4. Gestión: este bloque abarca todas las funcionalidades que tengan que ver con la gestión de diferentes aspectos de la red de sensores y actuadores; como pueden ser su jerarquía, contexto, configuración, etc.
5. Seguridad: este bloque abarca funcionalidades que tienen que ver con la seguridad a varios niveles: autenticación, autorización, integridad de la información, seguridad de los datos, etc.
6. Aplicación: este bloque representa la verticalizaciones: una vez dispones de todas las funcionalidades básicas que proporcionan los otros bloques, ya podemos centrarnos en aplicaciones de negocio concretas.

Como podremos observar durante el desarrollo de esta sección, empezaremos con "las cosas" y continuaremos con "la internet".

2.3.1. Las "cosas" en IoT

En los siguientes apartados, llamaremos a las "cosas" dispositivos. Un dispositivo en Internet de las cosas se refiere a un aparato que cumple estas características:

- Capacidad de comunicación para transmitir y/o recibir datos.
- Contextualización de los recursos que ofrecen (identificación unívoca, tiempo). Por ejemplo, si recibimos un valor de temperatura y no sabemos de que ubicación viene o a que hora se tomó esa medida, este dato no aporta información, y por tanto, es completamente inútil.

Una vez dicho esto, vamos a considerar que todo dispositivo englobado en este concepto tiene al menos una de las siguientes capacidades:

- Medir parámetros del entorno en el que se encuentra mediante sensores. Estos sensores pueden ser de temperatura, humedad, luminosidad...
- Realizar acciones sobre actuadores. Estos actuadores pueden ser relés, reguladores de tensión...
- Monitorizar y controlar otros dispositivos, es decir, comunicarse directa o indirectamente con otros dispositivos (M2M) y proporcionar las capacidades de éstos a otro dispositivo y/o sistema de más alto nivel de abstracción.

Con esta última idea planteada, podemos llegar a la conclusión de que existen fundamentalmente tres tipos de dispositivos en internet de las cosas:

Tipo I Dispositivos que cuentan con sensores y/o actuadores.

Figura 2.2: Netbows con 8 actuadores, con comunicación por wifi

Tipo II Dispositivos que utilizan sus capacidades de conexión para establecer comunicación con otros, tanto del tipo de los anteriores como del suyo propio; tomando el rol de "puente" entre diferentes tipos de conexiones y conformando las redes. Este tipo de comportamiento da a lugar al concepto M2M.

Tipo III Dispositivos "híbridos" que proporcionan las mismas funcionalidades que uno de Tipo I y otro de Tipo II.

La mayoría de los dispositivos de internet de las cosas que tienen las capacidades de medida y actuación son placas microcontroladas, de bajo consumo energético y reducido tamaño. Las capacidades de comunicación de estos dispositivos son, en muchas ocasiones, diferentes a las capacidades que se necesitan para establecer una conexión directa con Internet (es decir, no tienen la posibilidad de hacer uso de la familia de protocolos TCP/IP).

Teniendo en mente la pila de protocolos de internet en cinco capas (física, enlace, red, transporte y aplicación), podemos afirmar que los dispositivos considerados de Tipo I no suelen implementar más que la capa física y de enlace (Capítulo 1, apartado 5, del libro Fundamentos de Redes de Computadoras [6]).

Dentro de los denominados dispositivos de Tipo II (y Tipo III por la relación que tienen) existe un subtipo especial que aparece en la mayoría de desarrollos IoT que utilizan dispositivos de Tipo I. A este subtipo lo llamaremos "gateway" a lo largo de este trabajo.

La características claves de los gateways son:

- Disponen de todas las capacidades de comunicación necesarias para establecer conexión con dispositivos de Tipo I, independientemente de las tecnologías que utilicen éstos para transmitir y recibir información.
- Disponen de conectividad a Internet (es decir, pueden hacer uso de los protocolos de la familia TCP/IP). Este es el camino a la capa de aplicación, y por tanto al uso e integración de la red de sensores y actuadores con otras aplicaciones de negocio.

La figura de "gateway" es esencial a lo largo de este trabajo de fin de grado: el objetivo es desarrollar un sistema software para este particular tipo de dispositivo.

2.3.2. Protocolos en IoT

Durante esta sección hablaremos de los diferentes protocolos que nos encontramos en dispositivos de internet de las cosas. Para organizarlos, tomaremos de referencia la pila de protocolos de internet de cinco capas que se presenta en capítulo 1, apartado 5 del libro "Redes de computadoras" [6] y el apartado 1.1.1 del libro "Internet of Things: a hands on Approach" [1]. Las demás referencias consultadas se indican junto a la afirmación a constatar.

Capa física

El objetivo de la familia de protocolos que están contenidos en esta capa es dar soporte a la siguiente capa que se va a prensentar, la capa de enlace. Los protocolos de esta capa dependen de los protocolos de la capa de enlace y del medio físico por el que se transmiten los

bits individuales (cable de par trenzado, fibra óptica monomodo, radiofrecuencia, infrarrojos, etc.).

Por la relación de dependencia que establece con la capa de enlace, se hablará de protocolos concretos en la siguiente sección.

Capa de enlace

Lidiando con la capa física se encuentra la capa de enlace. El objetivo de esta capa es dar soporte a la capa de red, que se encarga de mover datagramas de un dispositivo a otro. Para que los datagramas lleguen a su destino, es muy posible que necesiten pasar por varios enlaces durante su ruta. A lo largo de este trabajo, llamaremos tramas a los paquetes de información que se transmiten en la capa de enlace. Para dar soporte a la transmisión de tramas, existen varios protocolos. Los más relevantes en este contexto -desarrollos IoT- son:

1. 802.3 - Ethernet: IEEE 802.3 es una colección de estándares para redes cableadas ethernet. Por ejemplo 802.3 es el estándar para conexiones mediante cable coaxial 10BASE5, y también existen 802.3i y 802.3j, que son para par trenzado y para fibra óptica; 10BASE-T y 10BASE-F respectivamente.

2. 802.11 - Wifi: IEEE 802.11 es una colección de protocolos para redes de área local inalámbricas. Ejemplos son 802.11a, 802.11b, 802.11g, 802.11.n. Cada una tiene unas características que las definen, como pueden ser la capacidad de transmisión y la banda (GHz) en la que operan.

3. 802.16 - WiMax: IEEE 802.16 es una colección de protocolos para banda ancha inalámbrica. Un ejemplo de protocolo de este conjunto es el protocolo 802.16m, que permite anchos de banda de hasta 1 Gbit/s.

4. 802.15.4 - LR-WPAN: 802.15.4 es una colección de protocolos para redes de baja tasa de transmisión de datos -y bajo consumo energético. Sirve de base para otros protocolos de más alto nivel como Zigbee, 6lowpan y MiWi.

5. Bluetooth 4.0 (Low Energy, LE): Al igual que el conjunto de protocolos anterior, Bluetooth 4.0 es una tecnología para crear redes WPAN (wireless personal area network). En sí no es un protocolo de capa enlace, es mucho más. La pila Bluetooth suele dividirse en dos partes fundamentales: “Controller”(HCI) y “Host”. La parte del “Controller” será la incluida en un dispositivo de “Tipo I”, mientras que la de “Host” irá incluida en uno de “Tipo II” (ver sección 2.3.1). Desde el “Host” haremos una interacción inmediata con la capa de enlace.[29]

6. 2G/3G/4G - Comunicación móvil: 2G, 3G y 4G hacen referencia a diferentes generaciones de estándares de comunicación mediante las redes móviles convencionales. 2G hace uso de GSM y CDMA, 3G hace uso de UMTS y CDMA2000, y 4G hace uso de LTE. Todos ellos permiten que dispositivos IoT se comuniquen a través de las mismas redes que los teléfonos móviles, lo cual supone una ventaja fundamental en muchos casos.

7. LoRa: LoRa es una tecnología de capa física inalámbrica. Los transceptores comerciales proporcionan una capa de enlace básica para comunicaciones P2P entre nodos. En este sentido, recuerda mucho al dispositivo "Controller" que está presente en el stack Bluetooth. Sobre LoRa, al igual que sucede en la familia de protocolos 802.15.4, se establecen varios protocolos cuya misión es completar el resto de capas del modelo OSI. Éstas se llaman LoRaWAN y Symphony Link; son de diferentes fabricantes y proporcionan diferentes características a sus clientes. La principal ventaja de LoRa frente a bluetooth y a 802.15.4 es la capacidad de transmisión: mientras bluetooth y 802.15.4 consiguen (máximos teóricos) 100 y 75 metros respectivamente, LoRa es capaz de transmitir con la misma cantidad de energía entre 1 y 22 kms. [30]

8. Sigfox: Sigfox es la empresa que está detrás de esta tecnología homónima. Ofrecen la posibilidad de conexión a una red mundial Sigfox que ellos mismos han desplegado, y que tiene un alcance de transmisión muy alto. La gestión de los datos que generen los dispositivos a través de su "nube". Para ello, hay que pagar una licencia. Por otra parte, la mayor parte de los módulos comerciales contienen, como hemos podido comprobar, una mínima capa de enlace que permite comunicar dispositivos mediante Sigfox en modo "P2P" (y así hacer redes locales, con los que llamamos dispotivos de Tipo I y Tipo II en la sección 2.3.1). Para hacer uso de estos módulos transceptores en modo P2P, no hace falta pagar licencia a la empresa. [31]

Hay muchas más tecnologías como enOcean [32], KNX[33], ZWave[34], Narrow-Band IOT[35], etc. Todas ellas son interesantes también. El mundo de internet de las cosas está en plena efervescencia. Aún no hay un claro ganador, y puede que nunca lo haya. Todas las soluciones que van apareciendo presentan ventajas e inconvenientes dependiendo de la solución que necesites. La conclusión a la que se pretende llegar es la siguiente: la parte de "Internet" en el concepto IoT es muy extensa y variada, no sólo hay dispositivos que se conecten directamente a la internet que todos conocemos y disfrutamos. De hecho, los dispositivos de Tipo I que se conectan directamente a internet son minoría.

Para acabar esta sección, es preciso indicar que algunos de los puntos de la enumeración anterior no se corresponden directamente con protocolos de capa de enlace como tal. La razón por la cual se han incluido es que el uso que se hace de ellos a nivel conceptual -sin olvidar que seguimos dentro del concepto de IoT-, es lo que se hace en la capa de enlace: mover tramas completas de un elemento de red (Tipo I) hasta otro elemento de red adyacente (Tipo II).

Capa de red/internet

El objetivo de la capa de red es enviar datagramas IP de una red de origen a otra red de destino. Esta capa proporciona direccionamiento y enrutamiento de paquetes. Para lograr esto, el datagrama contiene las direcciones de origen y destino necesarias para transmitir el paquete a través de varios elementos de red que se agrupan jerárquicamente. Los protocolos más importantes de esta capa son:

1. IPv4: es el protocolo más extendido actualmente. Utiliza direcciones de 32-bits para

identificar a los dispositivos que se conectan a la red, utilizando éstos un esquema jerárquico. En enero del año 2011, se tocó techo: IANA proporcionó los últimos pools de IPs v4 a RIR [28].

2. IPv6: es el sucesor de IPv4. La principal diferencia entre ambos es que utiliza direcciones de 128-bits, por lo que hay muchas más direcciones disponibles (2^{128} en IPv6 frente a 2^{32} en IPv4).

3. 6LoWPAN: Las siglas de este protocolo significan "IPv6 over Low power Wireless Personal Area Network". Este protocolo proporciona IPv6 a redes de dispositivos de bajos recursos que se conectan mediante IEEE 802.15.4

Capa de transporte

Los protocolos de la capa de transporte se encargan de proporcionar la transferencia de mensajes punto a punto, con independencia de la implementación de las capas inferiores. Principalmente existen dos protocolos a tener en cuenta en esta capa:

1. TCP: es el protocolo más utilizado. Sobre éste se construyen protocolos de la capa de aplicación, como pueden ser HTTP, SMTP y SSH. Es un protocolo orientado a conexión y con estado. TCP asegura la entegra de mensajes, tiene control de flujo y congestión y garantiza la entrega de paquetes en orden. Todas estas garantías pueden llevar a problemas de rendimiento en escenarios muy concretos.

2. UDP: en contraste con TCP, UDP es un protocolo que no está orientado a conexión. Está pensado para escenarios en los que se transmiten muchas pequeñas porciones de información, y éstas porciones han de ser entregadas lo antes posible, sin importar que lleguen repetidas y/o desordenadas. No ofrece ninguna de las garantías que proporciona TCP.

Capa de aplicación

Los protocolos de esta capa definen la interfaz mediante la cual los programas van a enviarse datos mediante la red. Es decir, proporciona comunicación programa a programa. El direccionamiento en programas dentro de una máquina se realiza mediante puertos. Ejemplos de protocolos importantes en Internet de las Cosas son:

1. HTTP/S: HTTP corresponde a las siglas "Hypertext Transfer Protocol". Está basado en el modelo de comunicación Request-Response (ver sección 2.3.3). El protocolo define los comandos GET, PUT, POST, DELETE, OPTIONS, etc.; que el cliente envía para recibir una respuesta. El protocolo HTTP hace uso de URI's para identificar recursos. HTTP hace uso de TCP para transmitir sus mensajes.

2. CoAP: CoAP es la sigla que corresponde a "Constrained Application Protocol". Este protocolo está orientado a dar soporte a aplicaciones M2M, en redes de dispositivos

de recursos limitados. Para ofrecer compatibilidad con HTTP, también hace uso de los comandos GET, PUT, POST... y el modelo de comunicación es Request-Response, pero en vez de hacer uso de TCP, está construido sobre UDP.

3. SSH: SSH es la sigla que corresponde a "Secure Shell". Este protocolo permite la conexión remota a otra máquina. También permite tunelar puertos; esto es, establecer enlaces seguros puerto a puerto de máquinas remotas.

4. WebSocket: Websocket es un protocolo que establece una conexión bidireccional entre dispositivos sobre un único socket. Esto hace que pueda amoldarse a más modelos de comunicación. [37]

5. MQTT: MQTT es un protocolo ligero basado en el modelo de comunicación publicación-subscripción. Está pensado para ser usado en dispositivos de recursos muy limitados, y en redes con poco ancho de banda. MQTT se implementa sobre una arquitectura cliente-servidor, donde el dispositivo que "mide" es cliente publicador de información. El servidor toma el rol de "broker", reenviado la información publicada a los clientes suscriptores (típicamente, otras aplicaciones). [39]

6. AMQP: AMQP corresponde a las siglas "Advanced Message Queueing Protocol". La arquitectura es muy parecida a la de MQTT, aunque AMQP no está pensado para dispositivos de bajos recursos, sino para comunicación entre programas de más alto nivel de abstracción. Aparte de dar soporte a diferentes modelos de comunicación, también ofrece enrutado y encolado de mensajes. [38]

7. ZeroMQ: proporciona mensajería distribuida y permite la implementación de todos los modelos de comunicación principales. Está pensado para comunicación entre procesos -tanto en la misma máquina, con propósito de paralelismo, como remoto - y es muy rápido. También está pensado para comunicar programas en diferentes lenguajes y en diferentes plataformas. [40]

2.3.3. Modelos de comunicación en IoT

Durante el desarrollo de esta sección veremos los modelos de comunicación qué más se repiten en desarrollos de internet de las cosas.

Request-Response

En este modelo el cliente envía una petición al servidor. El servidor hace lo necesario para formar la respuesta y se la devuelve al cliente -es decir, el servidor responde al cliente. Los puntos clave de este modelo son:

1. Es un modelo de comunicación sin estado.

2. Cada par pregunta/respuesta es independiente.

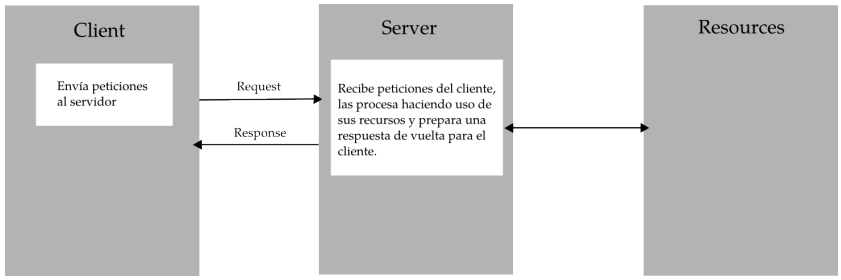

Figura 2.3: Modelo de comunicación "Request-Response"

Publish-Subscribe

En este modelo, están involucrados tres elementos: los publicadores, los suscriptores, y el broker (que media la comunicación entre ellos). Los publicadores son el origen de los datos. Los suscriptores son los consumidores de éstos. Los publicadores se conectan al broker y comienzan a enviarles mensajes identificados mediante una clave única (llamada topic). Los suscriptores se conectar al broker, haciendo uso del topic que les interese consumir. En cuanto el broker recibe un mensaje por un topic determinado, lo envía a todos los suscriptores conectados mediante ese topic. Los puntos clave de este modelo son:

1. Es orientado a conexión.
2. El modelo permite la comunicación uno a muchos (un elemento publica, todos los suscriptores reciben)
3. Permite implementar soluciones de monitorización en tiempo real

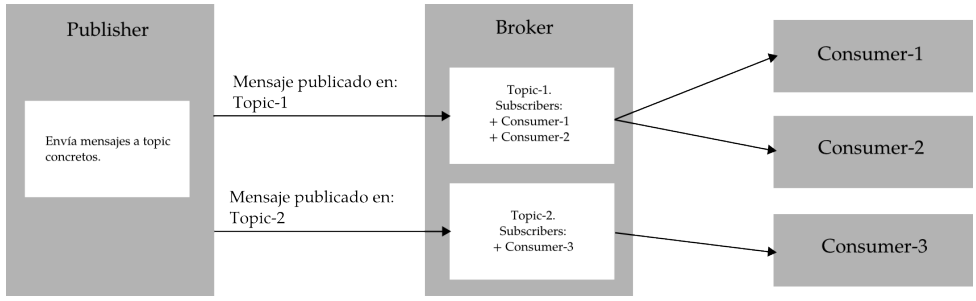

Figura 2.4: Modelo de comunicación "Publish-Subscribe"

Push-Pull

En este modelo, contamos con publicadores y suscriptores, como en el anterior. La clave de este modelo reside en que los productores no tienen que preocuparse de los consumidores de mensajes. En este modelo, los publicadores dejan sus mensajes en diferentes colas. Los productores consumen datos de esas colas, en estricto orden de llegada. El uso de colas lleva a un desacoplamiento de la comunicación entre productores y consumidores. Este modelo resulta útil en escenarios en el que las frecuencia de producción y consumición son distintas, es decir, los productores y consumidores no están sincronizados. Características clave de este modelo son:

1. Es un modelo de comunicación sin necesidad de estado, orientado a conexión desde los puntos de vista publicador-cola (push) y cola-suscriptor (pull).
2. Los mensajes quedan en las colas persistentes hasta que un suscriptor lo consume, es decir, a diferencia del modelo anterior, un mismo mensaje no llega a muchos, "se lo queda" el primero que lo consume.

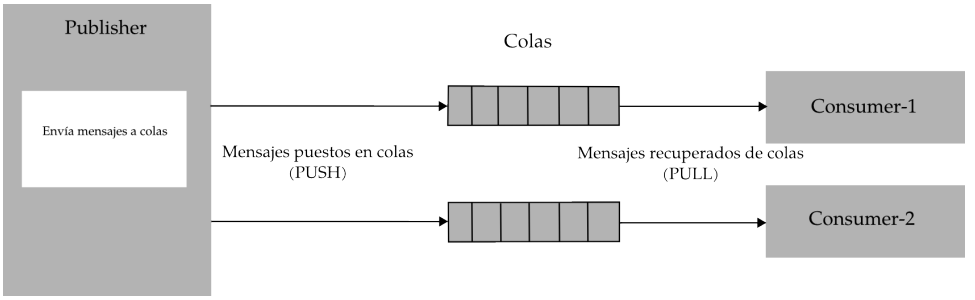

Figura 2.5: Modelo de comunicación "Push-Pull"

Exclusive-pair

En este modelo, tenemos dos roles fundamentales: cliente y servidor. Entre ellos se establece una conexión persistente y bidireccional (full-duplex). La comunicación es iniciada por el cliente, se intercambian los mensajes que precise el protocolo, y finaliza generalmente con una petición de fin de comunicación por parte del cliente, a la que el servidor responde con una confirmación (es decir, lo normal es que el servidor envíe el último mensaje). Características clave de este modelo son:

1. Es un modelo de comunicación con estado, orientado a conexión.
2. El servidor suele ser capaz de atender y establecer comunicación con varios clientes simultáneamente, es decir, el servidor es consciente de todas las conexiones abiertas.

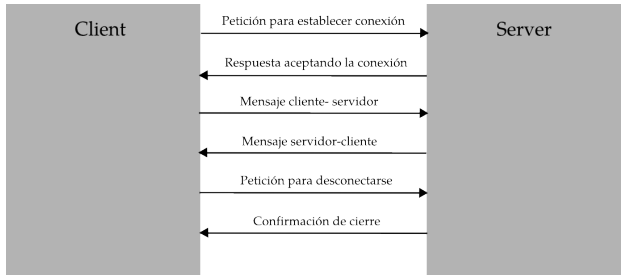

Figura 2.6: Modelo de comunicación "Exclusive pair"

2.4. Diferentes soluciones del mercado

Como anticipábamos en la introducción, el propósito fundamental de este trabajo es crear un framework que permita integrar redes de sensores y actuadores con plataformas de Internet de las Cosas. Como hemos visto a lo largo de este capítulo, los elementos arquitectónicos que toman estas responsabilidades - lidiar directamente con dispositivos y proporcionar su funcionalidad a capas de más alto nivel- se suelen llamar "gateways".

Teniendo en cuenta lo que hemos visto hasta ahora, podemos concluir que hay principalmente dos tipos de productos y servicios que resultan ser de interés máximo para este trabajo:

Tipo I Productos y servicios que den solución a la integración de dispositivos multitecnología con otras plataformas cuyo principal punto de entrada sea TCP/IP. Normalmente, se ceñirán a los conceptos "gateway", "framework", "toolkit", etc.

Tipo II Plataformas de Internet de las Cosas de más alto nivel. ¿Qué ofrecen? ¿Cómo se integran productos en ellas? ¿Qué necesitamos hacer para ser compatibles con ellas? Como veremos, la mayor parte de la integración con el tipo de software del punto anterior se realizará a través de APIs HTTP, y otras tecnologías de las que hemos hablado en la sección 2.3.2, y suelen ofrecer generalmente almacenamiento, analítica y visualización de datos. También suelen soportar reglas y capacidad de actuación. Suelen contar con mecanismos para establecer verticalizaciones, dependiendo del negocio.

Aunque se trata de una clasificación un poco gruesa, nos servirá para entender cómo están trabajando otras personas con los conceptos que este trabajo de fin de grado maneja.

2.4.1. Soluciones de Tipo I

En esta sección veremos varios desarrollos de los considerados "Soluciones de Tipo I". Después de una búsqueda intensa, hemos encontrado varias soluciones, de entre las cuáles destacamos las siguientes:

- Eclipse Kura [42]
- Linux Foundation IoTivity [43]

Existen más soluciones que resultan de interés, como Emutex Ubiworx [44], IoT Toolkit [45], VSCP [46] y DeviceHive [47].

Una conclusión interesante que he alcanzado durante esta búsqueda es que la mayoría de los proyectos que pretenden dar una respuesta a este problema son soluciones de código abierto y/o libre.

A continuación, veremos con un poco más de detalle por qué resultan interesantes Kura e IoTivity.

Eclipse Kura

Eclipse Kura es uno de los desarrollos englobados dentro del proyecto "Eclipse IoT" [41], cuya misión es contar con un conjunto de herramientas Open Source para realizar desarrollos de Internet de las Cosas.

Kura es un conjunto de librerías Java y servicios OSGi que son comúnmente necesarios en gateways de Internet de las Cosas, entre los que se incluyen:

- Servicios de Entrada/Salida
- Servicios de tratamiento de datos
- Servicios para integración con plataformas "cloud".
- Gestión de redes
- etc.

En la siguiente figura, se muestra un diagrama de funcionalidades de Kura:

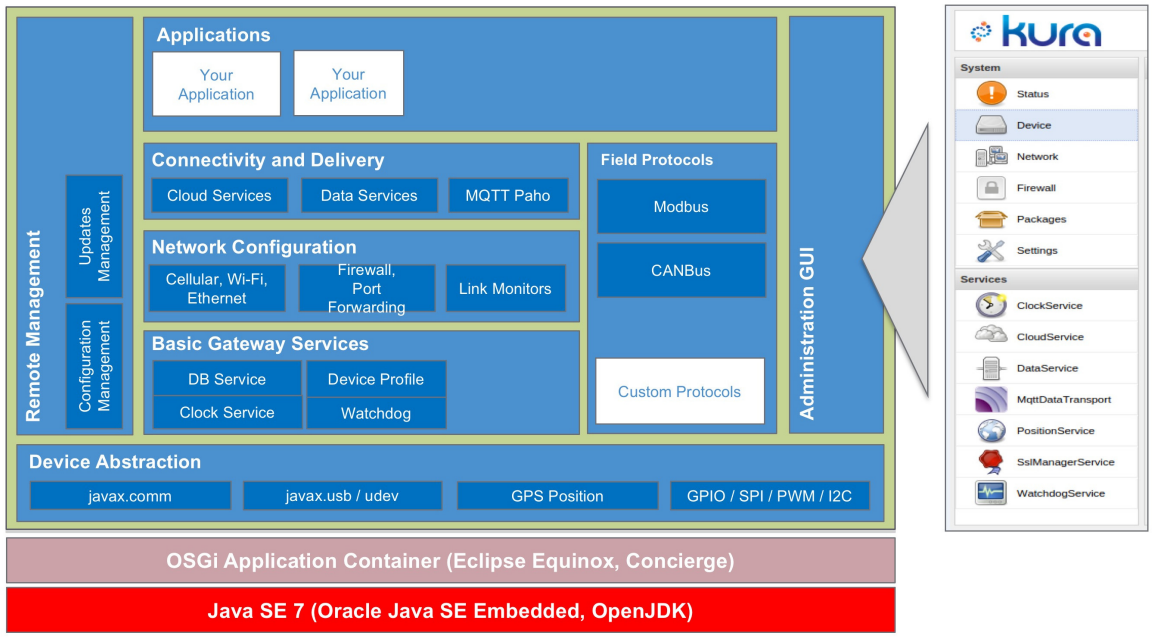

Figura 2.7: Arquitectura funcional de Eclipse Kura

Cuenta con varias cosas interesantes:

- Depende de la máquina virtual de Java y de una distribución de GNU/Linux, lo que le permite ser utilizado en muchas plataformas.
- En la capa "Device Abstraction", cuenta con varias de las tecnologías que vamos a necesitar durante el desarrollo de este trabajo: SPI, I2c, GPIO... hablaremos de ellas en el siguiente capítulo.
- En la capa "Basic Gateway Services", proporciona, entre otras cosas, un servicio de persistencia
- En la capa "Field Protocols", nos encontramos con implementaciones de protocolos industriales "Modbus", "CAN Bus" y "Custom".
- En la capa "Connectivity and Delivery", cuenta con MQTT.

Los elementos resaltados son aquellos que de más interés resultan de cara al desarrollo del presente trabajo. Más adelante veremos que este trabajo comparte objetivos con Kura, aunque con una visión ligeramente distinta: nuestro desarrollo pretende centrarse mucho en la comunicación con dispositivos, y por la necesidad de ser simple, contará con muchas menos características que Kura; a la par que aspira a ser compatible con muchos más dispositivos de internet de las cosas, y más versátil en cuanto al tema de integración.

Linux Foundation IoTivity

IoTivity resulta ser un desarrollo más centrado en la comunicación y puesta a disposición de las capacidades de los dispositivos, comparado con Kura. Esta desarrollado en C/C++, y como podemos observar en la siguiente figura, cuenta con los bloques de funcionalidades básicos que hemos presentado en la sección 2.3:

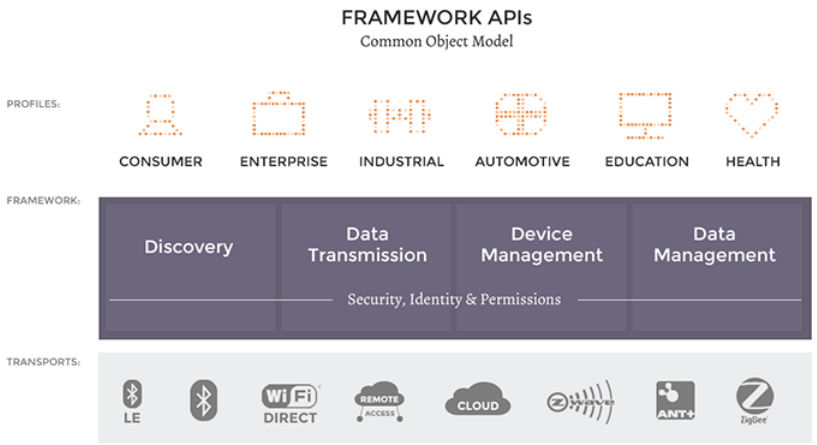

Figura 2.8: Visión general de funcionalidades de IoTivity

Como podemos observar en la figura, se centra en ser multitecnología, y proporciona servicios de descubrimiento, transmisión de información, gestión de dispositivos y gestión de información. Por encima de este bloque de funcionalidades, proporciona una API cuyo principal objetivo es dar soporte a muy diversos negocios.

En la siguiente figura, se presenta un diagrama de arquitectura global del sistema:

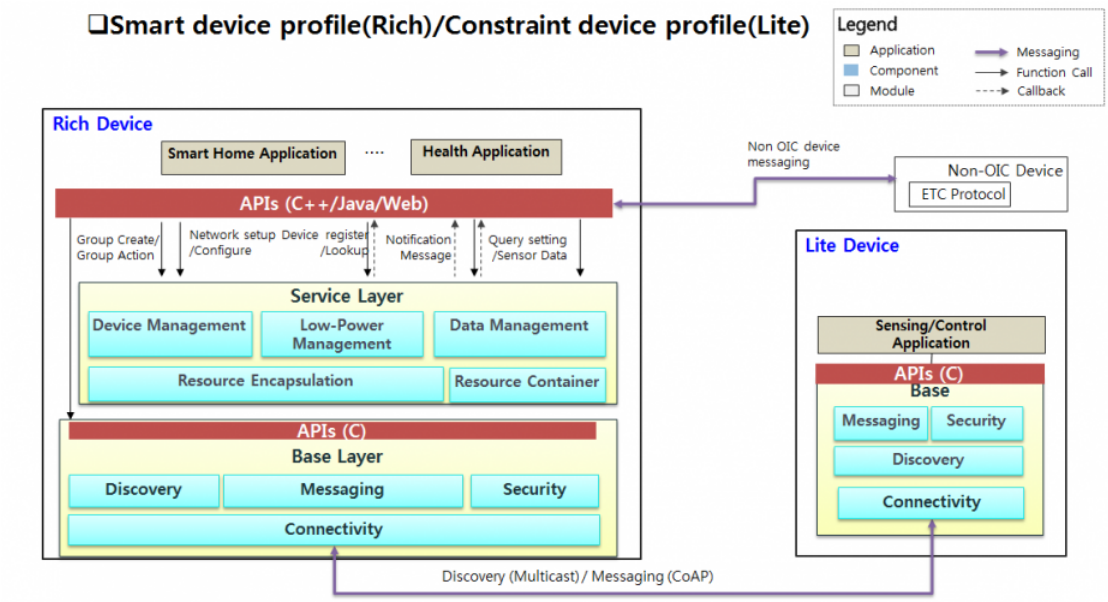

Figura 2.9: Arquitectura de IoTivity

Los puntos más interesantes de esta solución son:

1. Proporciona software para lo que considera "Rich Device" (gateway) y para "Lite Device" (dispositivos de Tipo I).
2. Hace uso de CoAP para la comunicación con los dispositivos, por lo que tiene una versatilidad grande; dado que con CoAP se pueden implementar varios paradigmas.

Que proporcione el firmware para dispositivos es, a la par que interesante, una restricción bastante seria: en muchas ocasiones, por no decir en la mayoría, no podremos modificar el firmware de los dispositivos.

Aunque cuenta con una documentación excelente, no es fácil trabajar con él: presenta muchos elementos de muy bajo nivel de abstracción.

2.4.2. Soluciones de Tipo II

En esta sección veremos varios desarrollos de los considerados "Soluciones de Tipo II". Después de una búsqueda intensa, hemos encontrado varias soluciones, de entre las cuáles destacamos Kaa [48], ThingSpeak [49], Carriots [50], Particle [51], Lhings [52], Fiware, ThingWorx

[54], Xively [55], SiteWhere [56], Blynk [57] y Zatar [58].

En esta ocasión, hacemos esta vista general porque son desarrollos que estarían en un nivel de abstracción más alto que el desarrollo que se pretende en este trabajo. Consultando las referencias, se puede observar que en aquellos que proporcionan documentación para desarrolladores, las labores de integración se realizan:

- Mediante una API REST, a la que hay que hacer peticiones cuando un dispositivo cambia de estado (es decir, la comunicación es request-response, y es el dispositivo el que hace las peticiones)
- Mediante WebSockets, lo que permite la implementación de más modelos de comunicación.
- Mediante colas de mensajes, con protocolos como AMQP y MQTT
- Mediante librerías que ellos proporcionan, y que en ocasiones sabemos qué hacen y en otras ocasiones, no.

Capítulo 3

Análisis

3.1. Introducción

Durante el desarrollo de este capítulo se analizarán y estudiarán los aspectos fundamentales que IoTGate ha de cubrir para cumplir sus objetivos. Por ello,

1. La primera sección explicará las conclusiones alcanzadas durante el desarrollo del contexto tecnológico en el que IoTGate se ubica (capítulo anterior, sección 2.3); nos permitirá entender mejor las necesidades que llevan a su creación.
2. Con el proyecto contextualizado y meramente acotado, especificaremos una serie de requisitos sobre los cuáles, más tarde, comenzaremos el diseño de la solución.
3. Por último, contaremos con un diagrama de clases de análisis, con el objetivo de organizar los conceptos detectados durante el desarrollo de este capítulo.

3.2. Ubicación del proyecto en el contexto

Para comenzar esta sección, vamos primero a hacer un resumen de las conclusiones clave a las que hemos llegado hasta ahora:

1. Existen fundamentalmente dos tipos de dispositivos, de Tipo I (miden, actúan, envían, reciben) y de Tipo II (ponen en comunicación diversos dispositivos, M2M). (ver sección 2.3.1). El objetivo de este trabajo es crear un software que cubra las necesidades de un subtipo especial de dispositivo de Tipo II: el llamado "gateway".
2. Hay muchísimas tecnologías para comunicación entre dispositivos (M2M), y la mayoría no se entienden con protocolos de la familia TCP/IP (es decir, la mayoría de los dispositivos no pueden conectarse directamente a Internet).

3. Las tecnologías de comunicación no-ip que hemos visto se ciñen de una manera u otra al modelo de cinco capas (ver sección 2.3.2). Por regla general, los dispositivos de Tipo I contienen como mínimo elementos que a nivel funcional proporcionan la capa física y de enlace.

4. La figura de los "gateways" aparece para establecer comunicación con dispositivos que no tengan capacidades de conectividad a internet, y poner a disposición de internet (a software de más alto nivel generalmente, a través de APIs de capa de aplicación) las funcionalidades de la red de sensores y actuadores (a nivel de capa de aplicación).

5. Otro punto fundamental, según comentábamos en la sección 2.3, es la contextualización de los datos de sensores y capacidades de los actuadores, y la gestión de los elementos que conforman la red.

Nuestro particular desarrollo se sitúa entre los dispositivos de Tipo I y las plataformas de almacenamiento, analítica y visualización de datos. Su principal objetivo será abstraer al desarrollador de las particularidades de cada tecnología para la comunicación (comunicación con dispositivos de Tipo I). A la par, ofrecerá pequeñas herramientas para hacer más fácil añadir compatibilidad con diferentes tecnologías, y también presentará elementos orientados a crear plataformas de almacenamiento, analítica y visualización de datos.

3.3. Especificación de requisitos de sistema

Durante el desarrollo de este apartado se presentarán los requisitos funcionales y no funcionales del usuario. Para llevar a cabo su redacción y clasificación, se han seguido los criterios establecidos en [3] (Capítulo 6: Requerimientos del software), teniendo en cuenta los siguientes aspectos y matizando en consecuencia:

1. Los usuarios de este sistema son, fundamentalmente, otros programas; es decir, no estamos ante un sistema interactivo.

2. Uno de los objetivos fundamentales de este desarrollo es disponer de un conjunto de herramientas que permita la integración rápida de dispositivos hardware con sistemas de más alto nivel.

3. Considerando los dos puntos anteriores, concluimos:

   - Una de las formas principales de uso de este sistema será a modo de "caja negra": otro desarrollador software utilizará IoTGate para integrar sus dispositivos con la API proporcionada por la plataforma objetivo.
   - Otra de las formas de uso de este sistema consistirá en su ampliación, añadiéndole compatibilidad con nuevos dispositivos.
   - Por tanto, en cualquier caso, el perfil de usuario de IoTGate será siempre técnico (desarrollador de software), dado que estamos enfrentando el desarrollo de un framework [59]

Por otra parte, para llevar a cabo la elaboración de los requisitos, se seguirá el siguiente formulario:

Identificador	<id único de requisito>
Nombre	<Nombre del requisito>
Descripción	<Descripción del requisito>
Obligatoriedad	<Deseable u Obligatorio>
Más información	<Justificación, fundamento...>

Cuadro 3.1: Plantilla de especificación para requisitos de usuario

3.3.1. Objetivos principales del sistema

Con estos puntos clave, vamos a presentar los cuatro bloques fundamentales que hay que atender en el desarrollo de este proyecto. El título de las secciones que vienen a continuación representan una necesidad general. Se expondrá un análisis de como suplirla, y en base a este análisis, se establecerán después los requisitos funcionales y no funcionales necesarios para comenzar a tramar una solución.

Conexión con dispositivos de diversas tecnologías

Para entender esta necesidad, tenemos que partir de la diversidad de tecnologías de comunicación presentadas en la sección 2.3.2. En este punto, tenemos que hacer una pequeña reflexión: ¿Cómo conseguimos conectividad con todo este tipo de redes?

Necesitamos para comunicaciones que pueda integrarse fácilmente con una plataforma que proporcione un sistema operativo. Este hardware son módulos transceptores como los que se presentan a continuación:

- MRF24J40 [60]: es un módulo transceptor que proporciona conectividad con dispositivos de redes 802.15.4:

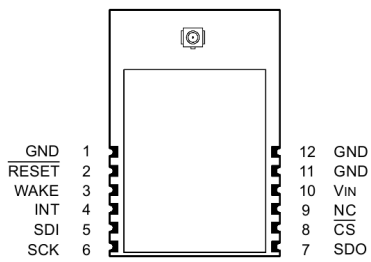

Figura 3.1: Diagrama de pines del módulo MRF24J40MC

- SX1272 [61]: es un módulo transceptor que proporciona conectividad con dispositivos de redes LoRa:

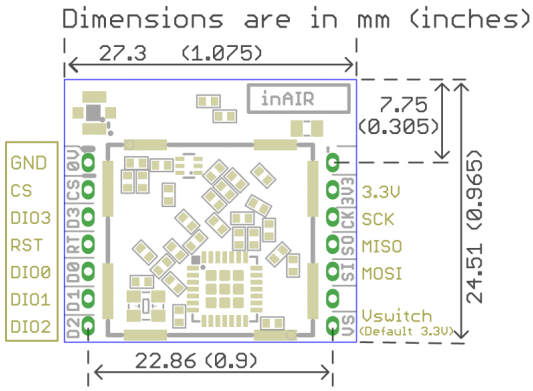

Figura 3.2: Diagrama de pines del módulo SX1272

Como podemos observar en las figuras, ambos presentan un esquema de pines muy parecido. Ambos módulos cuentan con una interfaz SPI (MISO, MOSI, SCK y CS), y varias entradas/salidas digitales.

Para controlar estos módulos desde una plataforma controladora, vamos a necesitar:

1. Que la plataforma disponga de buses para comunicación en serie con éstos módulos (SPI[62], UART[64], I2C[63], RS232[65], RS485, etc.)
2. Que la plataforma disponga de pines entrada/salida de propósito general (GPIO[66]). Esto nos permite encender/apagar el módulo (salida), detectar interrupciones (entrada), etc.
3. Implementar el driver siguiendo las indicaciones que el fabricante proporcione en su datasheet.

Por otra parte, numerosos dispositivos de los considerados de Tipo I cuentan con una interfaz a bus para comunicación en serie (por ejemplo, el MCP4728[15]); por lo que disponer de estas características también es deseable para la integración directa de estos dispositivos con el gateway.

En este contexto, una plataforma ideal en la que aparecen todos los elementos necesarios es la Raspberry Pi. La Raspberry Pi es un SBC [14] de tamaño muy reducido, que tiene la posibilidad de ejecutar una distribución GNU/Linux, conectividad a internet, y una GPIO de 40 pines (con SPI, UART, I2C y entradas y salidas digitales):

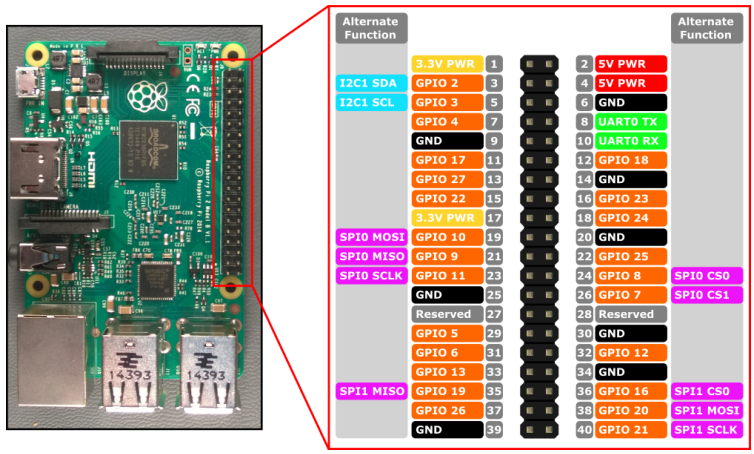

Figura 3.3: Diagrama de pines de la GPIO de la RPi2 y RPi3

La Raspberry Pi es la placa más famosa de estas características, pero no es la única: otras placas interesantes son la Radxa Rock [67] y la BeagleBone Black[68]. Todas ellas cuentan con una GPIO que permite ampliarlas en cuanto a hardware se refiere. En el mercado hay cientos de placas de características similares.

Monitorización y control de los dispositivos

Teniendo en cuenta lo presentado en la sección anterior, partimos de un punto en el que tenemos la posibilidad de comenzar a comunicarnos con dispositivos de Tipo I. Disponemos de varias vías de comunicación:

1. a través de un módulo transceptor (como el MRF24J40MC[60])
2. a través de buses de comunicación en serie (como I2C, por ejemplo)
3. a través de hardware gestionado por el sistema operativo (ethernet/wifi, bluetooth, etc.)

Ahora, necesitamos implementar los protocolos que nos permitan comunicarnos con los dispositivos de Tipo I. Al establecer esta comunicación, podremos poner a disposición de nuestro sistema los recursos de la red de sensores y actuadores que conforman los dispositivos de Tipo I con los que nos conectamos. Después de implementar el protocolo, podremos recoger los datos de los sensores, hacer acciones con los actuadores y configurar los dispositivos.

Cada dispositivo tiene una serie de características y funciones. Dependiendo del fabricante y el modelo, el dispositivo se comunicará con nosotros de una forma. Por tanto, concluimos con que cada tipo de dispositivo necesitará un protocolo. El protocolo puede ser un estándar

(modbus[69], bluetooth[29], etc.), puede ser propietario (definido por el propio fabricante del dispositivo), o puede ser una mezcla. En cualquier caso, el protocolo se podrá implementar haciendo uso de los modelos de comunicación presentados en la sección 2.3.3.

Gestión de los elementos de red

Llegados a este punto, disponemos de una serie de dispositivos lógicos que nos permiten hacer uso (leer datos de sensores, realizar acciones sobre actuadores y configurar dispositivos) de los dispositivos de Tipo I reales. Ahora, vamos a necesitar contextualizar los datos, como se establecía en la sección 2.3. La necesidad que da título a esta sección pretende dar soporte al bloque 4 (gestión).

Para añadir contexto a nuestra red necesitaremos poder personalizar los nombres de los elementos que la conforman y establecer una organización coherente de todos ellos.

También necesitaremos mecanismos que nos permitan dar de alta y baja a los diferentes elementos. En la sección 3.3.1 hablábamos de que cada dispositivo cuenta con un fabricante, un modelo; en definitiva, de unas especificaciones que determinan el protocolo a usar para establecer comunicación. A este concepto lo llamaremos producto, y todo dispositivo corresponderá con un producto.

Integración y acceso remoto

Con la visión global de objetivos presentados en las tres secciones anteriores, presentamos el último objetivo. Este último objetivo supone el "para qué" de este proyecto. Resulta clave en este proyecto poner a disposición de otros programas sus capacidades.

Por ello, necesitaremos poner a disposición de otros programas las capacidades que ofrezca, teniendo en cuenta varios puntos importantes:

1. Los programas que necesiten hacer uso de este proyecto pueden estar programados en lenguajes de programación distintos al del proyecto.
2. La arquitectura física del sistema dependerá enormemente del uso que se esté haciendo del framework. No podremos definir una arquitectura concreta, pero si recomendar varias en base al tipo de funcionalidades que se estén utilizando.
3. La mayor parte de los programas con los que nos integremos existirán previamente; es decir, ninguno estará diseñado para acoplarse perfectamente con el nuestro.

Estos últimos puntos llevan irremediablemente a tener en cuenta los siguientes puntos:

- La integración y despliegue de este proyecto contará con una parte muy importante **distribuida**.

- Vamos a necesitar dar soporte a varios modelos de comunicación genéricos, para que el uso de este software sea rápidamente entendible por otras personas.
- Este desarrollo pretende, en gran medida, reducir la complejidad que supone integrar hardware de Internet de las Cosas con software de más alto nivel. Por ello, hay que centrarse en que la arquitectura esté muy clara, haya pocos conceptos que aprender y en definitiva, sea una solución extremadamente simple y versátil.

3.3.2. Requisitos funcionales

Identificador	RFU-01
Nombre	Comunicación con módulos transceptores
Descripción	El sistema proporcionará comunicación en serie con módulos transceptores
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.2: RFU-01: Comunicación con módulos transceptores

Identificador	RFU-02
Nombre	Control de módulos transceptores
Descripción	El sistema dispondrá el uso de las entradas/salidas digitales que la plataforma proporcione para controlar los módulos transceptores
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.3: RFU-02: Control de módulos transceptores

Identificador	RFU-03
Nombre	Instalación/desinstalación de paquetes de compatibilidad con buses
Descripción	El sistema facilitará la instalación/desinstalación de paquetes que amplíen las capacidades de comunicación en serie
Obligatoriedad	Deseable
Más información	Ver secciones: 3.3.1

Cuadro 3.4: RFU-03: Instalación/desinstalación de paquetes de compatibilidad con buses

Identificador	RFU-04
Nombre	Facilidad en creación de drivers para transceptores
Descripción	El sistema facilitará la creación de drivers para módulos transceptores, poniendo al servicio del programador la comunicación en serie por bus y el control de las entradas/salidas digitales
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.5: RFU-04: Facilidad en creación de drivers para transceptores

Identificador	RFU-05
Nombre	Instalación/desinstalación de paquetes de compatibilidad con transceptores
Descripción	El sistema facilitará la instalación/desinstalación de paquetes que contengan drivers para transceptores
Obligatoriedad	Deseable
Más información	Ver secciones: 3.3.1

Cuadro 3.6: RFU-05: Instalación/desinstalación de paquetes de compatibilidad con transceptores

Identificador	RFU-06
Nombre	Facilidad en creación de drivers para dispositivos de Tipo I
Descripción	El sistema facilitará la creación de drivers para dispositivos de Tipo I, poniendo al servicio del programador la comunicación en serie por bus y el control de las entradas/salidas digitales
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.7: RFU-06: Facilidad en creación de drivers para dispositivos de Tipo I

Identificador	RFU-07
Nombre	Instalación/desinstalación de paquetes con drivers para dispositivos de Tipo I
Descripción	El sistema facilitará la instalación/desinstalación de drivers para dispositivos de Tipo I.
Obligatoriedad	Deseable
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.8: RFU-07: Instalación/desinstalación de paquetes con drivers para dispositivos de Tipo I

Identificador	RFU-08
Nombre	Envío de tramas a dispositivos de Tipo I
Descripción	El sistema debe permitir el envío de tramas a dispositivos de Tipo I.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.9: RFU-08: Envío de tramas a dispositivos de Tipo I

Identificador	RFU-09
Nombre	Recepción de tramas de dispositivos de Tipo I
Descripción	El sistema debe permitir la recepción de tramas de dispositivos de Tipo I.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.10: RFU-09: Recepción de tramas de dispositivos de Tipo I

Identificador	RFU-10
Nombre	Facilidad de creación de protocolos
Descripción	El sistema facilitará la implementación de protocolos para comunicación con dispositivos de Tipo I, poniendo a disposición del desarrollador implementaciones de protocolos estándar y otros elementos de utilidad (como conversores de un tipo de dato a otro, por ejemplo).
Obligatoriedad	Deseable
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.11: RFU-10: Facilidad de creación de protocolos

Identificador	RFU-11
Nombre	Instalación/desinstalación de paquetes con protocolos para comunicación con dispositivos de Tipo I
Descripción	El sistema facilitará la instalación/desinstalación de protocolos para comunicación con dispositivos de Tipo I.
Obligatoriedad	Deseable
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.12: RFU-11: Instalación/desinstalación de paquetes con protocolos para comunicación con dispositivos de Tipo I

Identificador	RFU-12
Nombre	Crear, editar y borrar productos
Descripción	El sistema debe permitir crear, editar y borrar especificaciones de producto.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.13: RFU-12: Crear, editar y borrar productos

Identificador	RFU-13
Nombre	Crear, editar y borrar grupos
Descripción	El sistema debe poder crear, editar y borrar grupos de dispositivos
Obligatoriedad	Obligatoria
Más información	Ver secciones: 3.3.1

Cuadro 3.14: RFU-13: Crear, editar y borrar grupos

Identificador	RFU-14
Nombre	Añadir y quitar dispositivos de grupos
Descripción	El sistema debe permitir el añadido de dispositivos a un grupo, así como el cambio de un dispositivo de un grupo a otro.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.15: RFU-14: Añadir y quitar dispositivos de grupos

Identificador	RFU-15
Nombre	Crear, editar y borrar dispositivos
Descripción	El sistema debe poder crear, editar y borrar dispositivos.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.16: RFU-15: Crear, editar y borrar dispositivos

Identificador	RFU-16
Nombre	Nombrar elementos de la red
Descripción	El sistema debe obligar a que cada elemento de red tenga un nombre personalizado que permita contextualizar los valores de los sensores y actuadores.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.17: RFU-16: Nombrar elementos de la red

Identificador	RFU-17
Nombre	Marcado de timestamp de observación en valores provenientes de sensores o actuadores.
Descripción	El sistema apuntará la hora a la que se midió/observó el dato; siempre y cuando sea posible.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.18: RFU-17: Marcado de timestamp de observación en valores provenientes de sensores o actuadores

Identificador	RFU-18
Nombre	Marcado de timestamp de recepción en valores provenientes de sensores o actuadores
Descripción	El sistema apuntará siempre la hora de recepción del dato recibido.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.19: RFU-18: Marcado de timestamp de recepción en valores provenientes de sensores o actuadores

Identificador	RFU-19
Nombre	Consulta de históricos de valores de dispositivos
Descripción	El sistema debe permitir la consulta de valores de sensores/ estado de actuadores entre dos fechas dadas.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.20: RFU-19: Consulta de históricos de valores de dispositivos

Identificador	RFU-20
Nombre	Suscripción a notificaciones de dispositivos
Descripción	El sistema debe permitir la suscripción a notificaciones sobre valores de sensores y realización de acciones sobre actuadores.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.3, 2.3.2

Cuadro 3.21: RFU-20: Suscripción a notificaciones de dispositivos

Identificador	RFU-21
Nombre	Suscripción a notificaciones del sistema
Descripción	El sistema debe permitir la suscripción a notificaciones sobre eventos clave que sucedan en el sistema.
Obligatoriedad	Deseable
Más información	Ver secciones: 3.3.1, 2.3.3, 2.3.2

Cuadro 3.22: RFU-21: Suscripción a notificaciones del sistema

Identificador	RFU-22
Nombre	Log del sistema
Descripción	El sistema debe crear un log en el que detalle los eventos clave que han sucedido.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.3, 2.3.2

Cuadro 3.23: RFU-22: Log del sistema

Identificador	RFU-23
Nombre	Exposición de métodos a través de API
Descripción	El sistema debe permitir la gestión de los elementos de red y la configuración de las conexiones a través de una API.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.3, 2.3.2

Cuadro 3.24: RFU-23: Exposición de métodos a través de API

Identificador	RFU-24
Nombre	Serialización de objetos
Descripción	El sistema debe proporcionar un serializador de objetos que permita que toda interacción con la API esté en un formato estándar, de cara a facilitar la integración con otros lenguajes y a facilitar la creación de interfaces remotas.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.3, 2.3.2

Cuadro 3.25: RFU-24: Serialización de objetos

Identificador	RFU-25
Nombre	Servicio de persistencia
Descripción	El sistema ofrecerá un servicio de persistencia que proporcionará soporte para un catálogo de productos, guardado de estado de la red y de los datos que ésta genere.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.26: RFU-25: Servicio de persistencia

3.3.3. Requisitos no funcionales

Identificador	RNFU-01
Nombre	Formato de hora y zona horaria
Descripción	Todos los timestamps, independientemente del formato en el que los dispositivos comuniquen el tiempo, harán uso de UNIX Timestamp [ref].
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.27: RNFU-01: Formato de hora y zona horaria

Identificador	RNFU-02
Nombre	Representación lógica del dispositivo
Descripción	Todos los dispositivos con los que se establezca comunicación han de estar representados de la misma manera lógica en el sistema, para poder hacer un uso homogéneo de éstos. Esta representación lógica pondrá a disposición del resto del sistema las capacidades del dispositivo (estado de sensores, actuadores, configuración del dispositivo)
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.28: RNFU-02: Representación lógica del dispositivo

Identificador	RNFU-03
Nombre	Representación lógica del producto
Descripción	Todos los dispositivos con los que se establezca comunicación han de estar asociados a un producto. Los productos especificarán las capacidades del dispositivo, así como las configuraciones posibles de éste y el tipo de conexión/protocolo necesario para la comunicación
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.29: RNFU-03: Representación lógica del producto

Identificador	RFNU-04
Nombre	Asociar producto a dispositivo
Descripción	El sistema debe obligar a que todo dispositivo corresponda a un producto
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.1

Cuadro 3.30: RFNU-04: Asociar producto a dispositivo

Identificador	RNFU-05
Nombre	Formato de serialización de entidades del sistema
Descripción	La serialización de los objetos del sistema se realizará utilizando el formato JSON.
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1, 2.3.3, 2.3.2

Cuadro 3.31: RNFU-05: Formato de serialización de entidades del sistema

Identificador	RNFU-06
Nombre	Formato del log
Descripción	Cada línea del log indicará la fecha y la hora (en UTC) del suceso del evento, así como que elemento ha lanzado el evento y un mensaje descriptivo.
Obligatoriedad	Obligatorio
Más información	Ver secciones:

Cuadro 3.32: RNFU-06: Formato del log

Identificador	RNFU-07
Nombre	Compatibilidad de servicio de persistencia con SGBD
Descripción	El servicio de persistencia tiene que ser compatible con MySQL, PostgreSQL y SQLite como mínimo.
Obligatoriedad	Obligatorio
Más información	Ver secciones:

Cuadro 3.33: RNFU-07: Compatibilidad de servicio de persistencia con SGBD

Identificador	RNFU-08
Nombre	Compatibilidad del sistema con diferentes arquitecturas
Descripción	El sistema tiene que funcionar tanto en SBCs como la RPi (arquitecturas ARM) como en otras plataformas más convencionales (x86, amd64).
Obligatoriedad	Obligatorio
Más información	Ver secciones: 3.3.1

Cuadro 3.34: RNFU-08: Compatibilidad del sistema con diferentes arquitecturas

3.4. Diagrama de clases

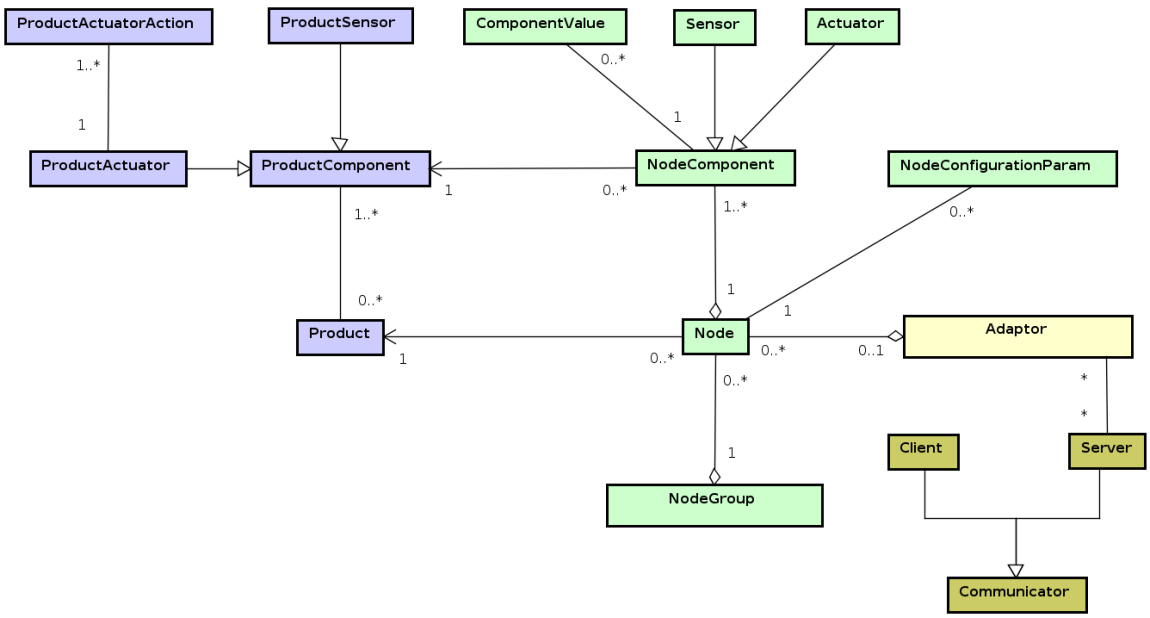

Figura 3.4: Diagrama de clases de análisis

En las siguientes secciones, vamos a explicar cuál es la misión de las clases principales de cada bloque (hay cuatro bloques, uno por color).

3.4.1. Communicator

La clase "Communicator" aparece en el diagrama para dar soporte a las necesidades/requisitos presentados en el apartado 3.3.1. Sus especializaciones ("Client" y "Server") proporcionarán conexión con diferentes dispositivos de Tipo I. "Client" y "Server" aparecen para dar soporte a los diferentes modelos de comunicación (2.3.3).

Si el dispositivo al que pretendemos conectarnos inicia la conexión contra IoTGate, éste estará tomando el rol de cliente, y su petición será atendida por una instancia de "Server". En este punto, si el dispositivo sólo está notificando, podemos dar soporte al modelo publish-subscribe. Si el dispositivo necesita que completemos un protocolo en el que la instancia de "Server" y él se intercambian varias tramas, mediante "Server" e instanciaciones de "Client", podremos dar soporte al modelo "Exclusive-pair".

Contra otro tipo de dispositivos, la conexión se iniciará desde IoTGate, por lo que se instanciará "Client" para realizar la petición. A partir de aquí, volvemos a tener la posibilidad de implementar el modelo de comunicación. Concluyendo, los "Communicator" proporcionan las posibilidades. En la siguiente sección, los "Adaptor" se harán cargo de utilizar estas posibilidades.

3.4.2. Adaptor

Esta clase representa las necesidades/requisitos presentados en la sección 3.3.1. Gracias a la funcionalidad provista por los "Communicators", y el soporte que ofrecen para implementar modelos de comunicación mediante sus especializaciones "Server" y "Client", podremos implementar aquí los protocolos de comunicación que nos permitirán poner a disposición lógica las capacidades de los dispositivos conectados a IoTGate. Llegarán los mensajes, serán interpretados, y servirán para instanciar la representación lógica del dispositivo (siguiente apartado, "Node").

Dependiendo de la implementación concreta que tengan los "Server" y "Client" de "Communicator", podremos utilizar el mismo protocolo conectándonos a los dispositivos mediante diferentes tecnologías.

3.4.3. Node

La clase "Node" representa a un dispositivo de Tipo I, y teniendo en cuenta los objetivos de este trabajo, es fácil entender que es una de las clases más importantes del sistema. Las capas de arriba nunca trabajarán directamente con las instancias de "Adaptor" y "Communicator", sino que trabajarán con "Node". Los conjuntos de "Node" (representados con

"NodeGroup") serán los que conformen la red de sensores y actuadores.

Si llega al sistema una petición de acción, esta acabará la instancia de "Node", "Node" hará uso de "Adaptor", "Adaptor" hará uso de "Client" y al final, llegará al dispositivo de Tipo I.

3.4.4. Product

La clase "Product" representa el conjunto de especificaciones compartidas por varios "Node" de la red, como explicábamos en la sección 3.3.1. Todo dispositivo de "Tipo I" habrá sido fabricado por alguien, y contará con una serie de capacidades (sensores, actuadores) y funcionalidades (configuración). Todo "Node" contará con un "Product" que describa sus características.

Capítulo 4

Diseño e implementación

4.1. Introducción

Durante el desarrollo de este capítulo, estudiaremos el diseño y la implementación de la solución dada al problema que enfrentamos.

Comenzaremos describiendo características del diseño arquitectónico, siguiendo los criterios presentados en el capítulo 11 del libro "Ingeniería del software", por Ian Sommerville [3].

Después, iremos desgranando los elementos que conforman el sistema, estudiando su diseño, las implicaciones de éste y su posterior implementación.

4.2. Diseño arquitectónico

Durante el desarrollo de esta sección hablaremos de las decisiones tomadas con respecto al diseño arquitectónico. Es fruto del conocimiento y de la experiencia que se han obtenido a la hora de enfrentar, de la forma más simple posible, toda la complejidad que supone convertir lo heterogéneo en homogéneo.

Para llevar a cabo el diseño arquitectónico de IoTGate, se ha considerado un modelo de capas. Como hemos presentado en el capítulo de análisis (ver sección 3.2), se ha tomado, en gran medida, la pila de protocolos de Internet de cinco capas como referencia. La arquitectura también cuenta con características del modelo cliente-servidor (ver sección 2.3.3). Con IoTGate queremos conectarnos a dispositivos, conformar una red de sensores y actuadores y poner a disposición de otros programas sus capacidades; tanto de forma local como remota.

Contaremos con un arquitectura lógica de cuatro capas; cada capa suple una de las necesidades presentadas en 3.3.1.

Por otra parte, la estrategia tomada para desarrollar la descomposición modular del sistema es la estrategia de descomposición orientada a objetos. En cuanto a estilos de control, contamos con un control en mayor medida centralizado (capa "core"), pero con ciertos elementos de control basados en eventos, dadas las características de sistema de tiempo real que IoTGate presenta.

Para desarrollar este capítulo, tomaremos un enfoque descendente:

1. Primero introduciremos qué objetivo tiene la capa y cómo es la arquitectura lógica de ésta.
2. Continuaremos hablando de la descomposición modular de la capa. Hablaremos de los detalles clave del diseño.
3. Una vez vista la descomposición modular, se presentarán las dependencias que ésta tenga con otras capas - sin tener en cuenta elementos que queden fuera del sistema, como otras librerías.
4. Para finalizar, se comentarán los detalles más relevantes acerca de la implementación.

Antes de empezar el desarrollo del trabajo capa a capa, hablaremos del sistema en general, presentando su arquitectura lógica global, y que hace cada capa a grandes rasgos.

El proyecto está programado en Python 3.4, y ha sido desarrollado y probado sobre una Raspberry 2 y una Rapberry 3. En el capítulo de análisis justificaba el uso de Raspberry Pi (3.3.1). Por otra parte, varios requisitos no funcionales presentados en el capítulo anterior (3.3.3) se satisfacen en gran medida gracias a la adopción de Python como lenguaje de desarrollo principal del sistema.

4.2.1. Arquitectura lógica

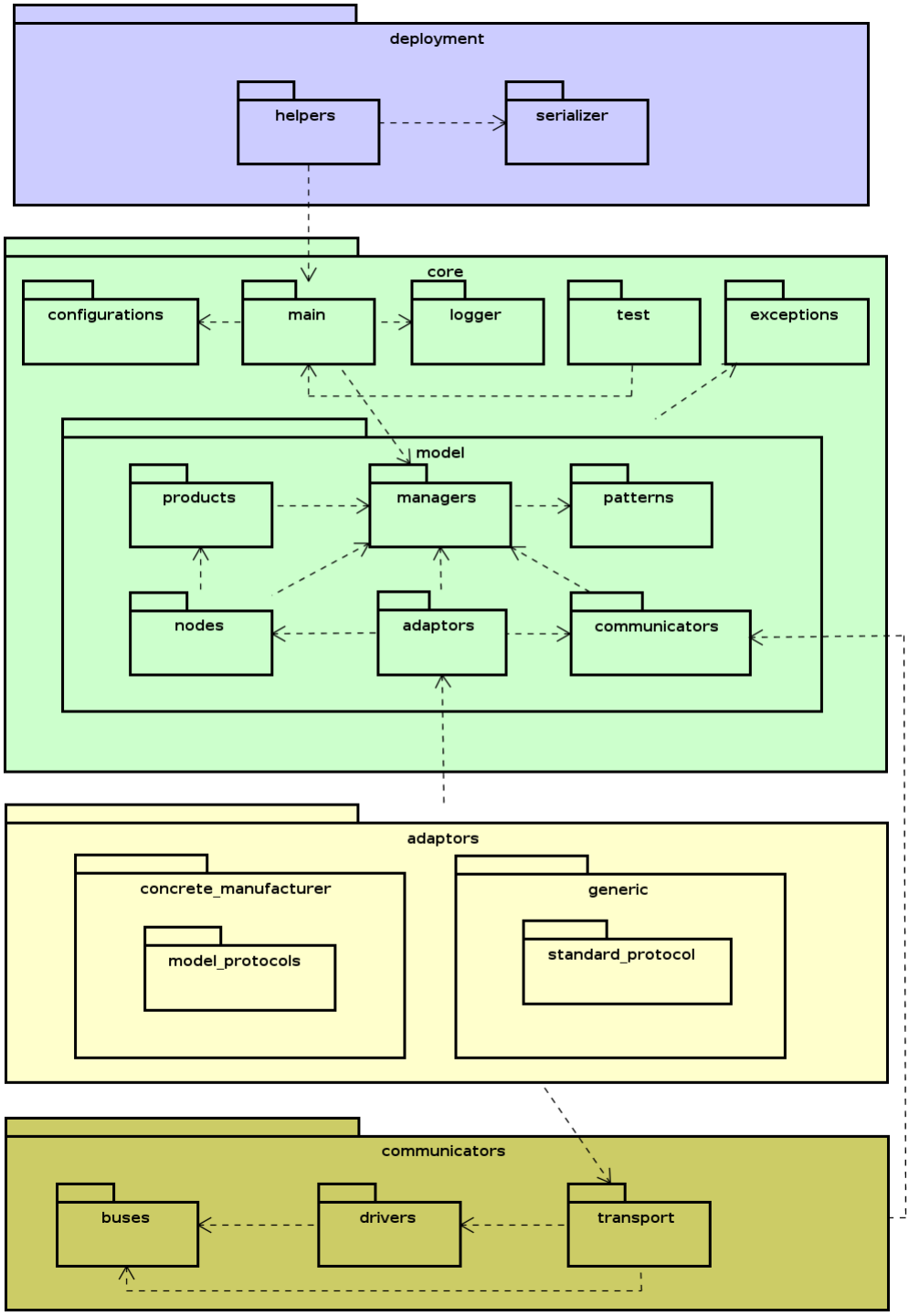

Figura 4.1: Diagrama de arquitectura lógica del sistema

Como podemos observar en la figura, contamos con cuatro capas:

1. La capa "core" es el corazón de IoTGate. Cuenta con "model", donde se encuentra el modelo de dominio del sistema con todos sus elementos relacionados: proporciona las clases abstractas (communicators y adaptors) que permiten extender las funcionalidades del sistema. El modelo está integrado con un ORM, proporcionando así el servicio de persistencia mínimo que necesitamos, y managers, que es el punto de entrada a esta simbolización del sistema. Fuera de "model", nos encontramos con: un servicio para hacer log, un gestor para cargar la configuración (de la base de datos y el servicio de creación de logs), el conjunto de excepciones posibles para el core, main, que es el punto de instanciabilidad del sistema, y por último, "test", donde se encuentran diferentes pruebas del sistema.

2. La capa "communicators" proporciona la conexión con los dispositivos. En "transport", nos encontraremos agrupados por tecnologías las implementaciones de "Client" y "Server" concretas que utilizaremos en la siguiente capa, adaptors. En "drivers", encontraremos las implementaciones de los controladores para módulos transceptores. En buses, encontraremos las implementaciones necesarias para comunicación en serie con los módulos transceptores y otros dispositivos (es decir, existe la posibilidad de que haya "Client" y/o "Server" de un bus que no dependa de ningún elemento de "drivers"). La carga de esta capa es dinámica, por lo que "core" puede instanciarse sin necesitar que esta exista, y el contenido de cualquiera de los espacios de nombres de "communicators" puede variar de una distribución de IoTGate a otra.

3. La capa "adaptors" cuenta con las implementaciones de protocolos que nos permitirán hacer uso de los dispositivos. Esta capa es la encargada de proporcionar a los nodos ("Node") sus datos y funcionalidades. Al igual que la capa anterior, es completamente dinámica.

4. La capa "deployment" nos permite proporcionar las capacidades de IoTGate a otras aplicaciones. Para ello, cuenta con dos elementos fundamentales: "helpers" contiene los elementos que dan soporte a los modelos de comunicación pregunta-respuesta y a publicación suscripción. Extendiendo los elementos básicos de "helpers", podremos crear API's personalizadas. "helpers" hace uso de "serializer": la misión de "serializer" es serializar cualquier objeto del modelo de dominio del sistema, transformándolo en formato JSON. Esto es especialmente interesante en entornos distribuidos, donde muy probablemente se pretenda hacer uso de IoTGate en lenguajes diferentes a Python; o se pretenda hacer viajar objetos de negocio por la red.

4.3. Diseño e implementación de las capas

En la siguiente sección de este capítulo, iremos desgranando los entresijos del diseño y posterior implementación capa a capa.

4.3.1. Capa "core"

La capa "core" es la base de IoTGate, supondrá ser el punto clave de extensibilidad e instanciación. Modela todos los elementos fundamentales del sistema, por lo que hemos considerado adecuado comenzar por esta capa. Proporciona servicio de log, persistencia y la organización de la red de sensores y actuadores. También da soporte al catálogo de productos.

Arquitectura lógica

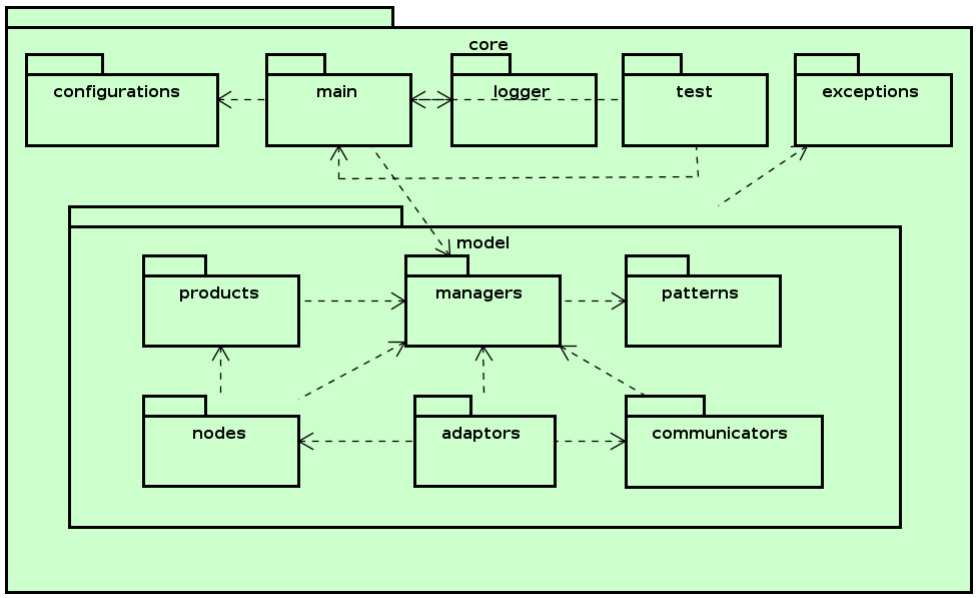

Figura 4.2: Diagrama de arquitectura lógica de "core"

Comenzamos describiendo los elementos que están dentro de "model". "model" es la parte de "core" donde están modelados todos los elementos que conforman el sistema:

1. **Products**: en este módulo, nos encontraremos todas las clases relacionadas con el elemento Producto, que como comentamos en el capítulo de análisis, es el elemento que especifica que características tienen los dispositivos (y por tanto todos los nodos "Node" que representen a un dispositivo físico).

2. **Communicators**: en este módulo, nos encontraremos con todos los elementos que representan de forma abstracta a los communicators. También se establecen en este módulo que relaciones tienen los communicators con el resto del sistema.

3. **Adaptors**: en este módulo, nos encontraremos los elementos abstractos que representan cómo ha de ser un adaptor concreto, y cómo se relaciona éste con el sistema.

4. **Nodes**: en este módulo, nos encontraremos todos los elementos que tienen que ver con la representación y organización de dispositivos lógicos.

5. **Managers**: en este espacio de nombres nos encontraremos con cuatro managers: uno de productos, otro para communicators, otro para adaptors y otro para nodes. Los managers nos permitirán manipular todo elemento del modelo de manera sencilla, y supondrán el punto de entrada al sistema para las capas superiores.

6. **Patterns**: contiene implementaciones abstractas de patrones que necesitaremos en varios puntos del desarrollo.

Las clases del paquete "model" hacen uso de un ORM que proporciona persistencia. Esta persistencia guarda la estructura de la red, las relaciones entre nodos y adaptors, la relación entre adaptors y communicators, y la configuración de los communicators. También guarda información de contexto y configuraciones de nodos.

Fuera de "model", contamos con:

1. **"configuration"**: contiene las clases que cargan y validan la configuración. La configuración se describe mediante ficheros externos al sistema, en formato "*.ini".

2. **"logger"**: implementa el servicio de logs del sistema. Dependiendo de la configuración, se comportará de diferentes maneras.

3. **"exceptions"**: cuenta con las excepciones concretas que pueden suceder en el core.

4. **"test"**: contiene las pruebas del sistema.

5. **"main"**: supone ser el punto de instanciación del sistema. A través de él, cargaremos la configuración (de la base de datos para el ORM y el log), instanciaremos el servicio de log (que quedará suscrito a los eventos del sistema), y realizaremos los tests cuando estemos probando el sistema. Será el elemento del que haga uso la capa deployment para poner a disposición del terceros las capacidades del sistema.

Descomposición modular

La capa "core" contiene muchos elementos, y por tanto, iremos desgranando su diseño poco a poco. Al igual que en la sección anterior, comenzaremos presentando "model":

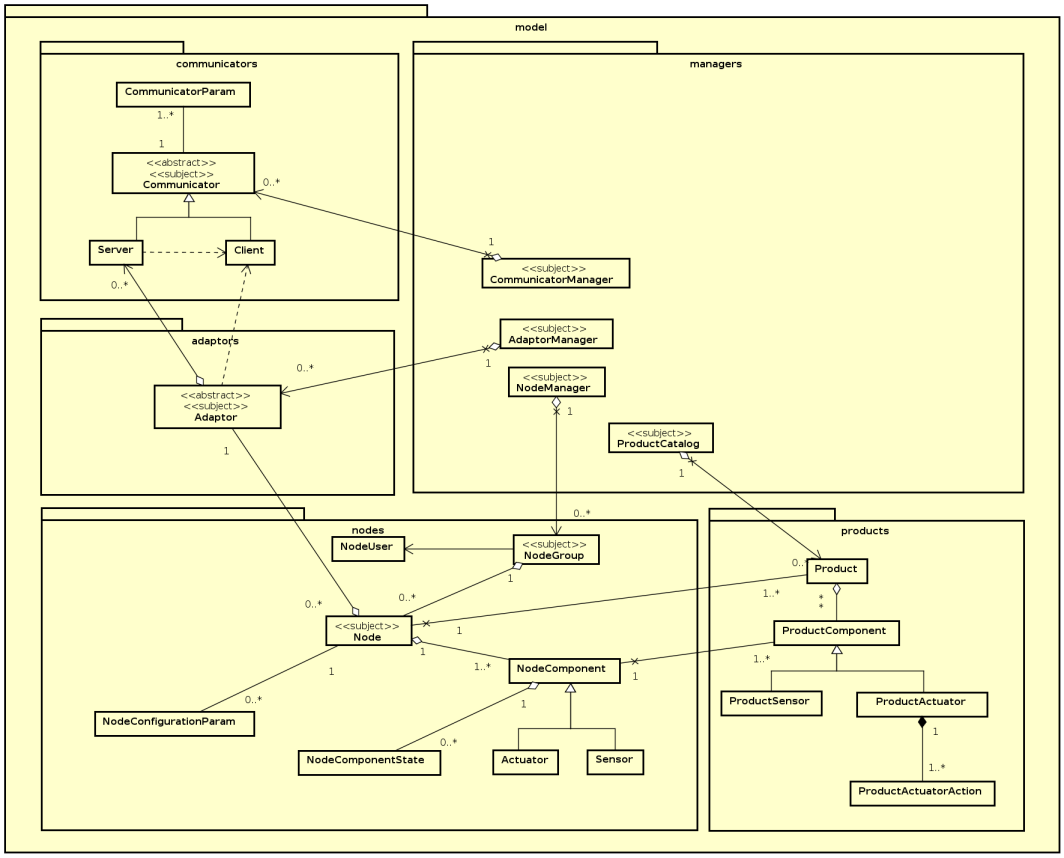

Figura 4.3: Diagrama de descomposición lógica de “core::model”

En el diagrama, podemos observar dos cosas llamativas:

1. **Las clases no tienen métodos ni atributos**. La idea en un primer instante es mostrar las relaciones entre los elementos que conforman “model”. Iremos detallando cómo son estas clases a lo largo de la sección.

2. El **estereotipo <<subject >>**: es un mecanismo que indica que esta clase implementa “Subject”, una de las clases abstractas que contiene el paquete “patterns”. **Significa que la clase puede lanzar eventos**. De los tipos de eventos que cada clase lanza, y de cómo lo hacen, hablaremos más tarde.

Vamos a ir desgranando como son las clases de cada paquete:

1. Communicators:

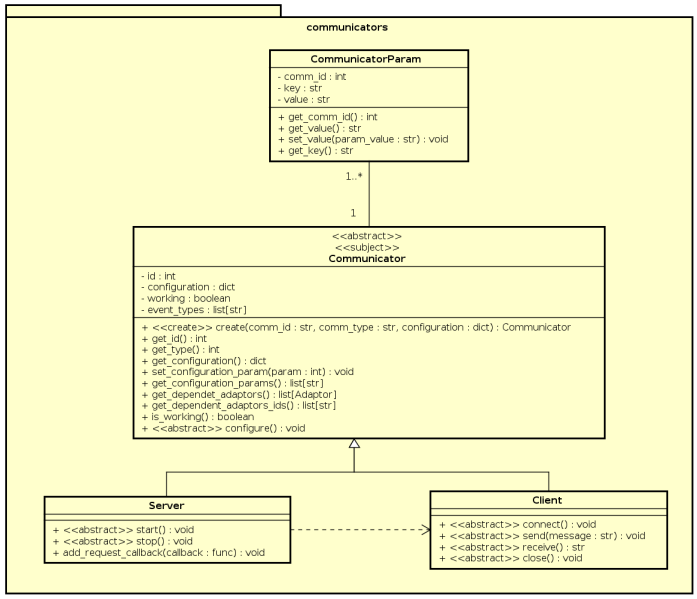

Figura 4.4: Diagrama de descomposición lógica de "core::model::communicators"

- La clase "CommunicatorParam" permite guardar la configuración de cada "Communicator" en términos de clave-valor. Por ejemplo, si tenemos un servidor ("Communicator" de tipo "Server") que va a comunicarse con dispositivos mediante TCP/IP, habrá al menos dos instancias de "CommunicatorParam": una indicará el host (0.0.0.0, por ejemplo) y otra el puerto (8000, por ejemplo). Si contamos con un servidor de 802.15.4, tendremos tres instancias, cuyas claves serán "controller_id" (2000, por ejemplo), "pan_id" (bcda, por ejemplo) y "channel (18, por ejemplo)"
- La clase "Communicator" permite conocer el estado de la instancia de "Communicator" concreto con la que estemos trabajando. Por ejemplo, si contamos con un cliente y no sabemos si está conectado, haremos uso del método "is_working()". Si queremos conocer que instancias de "Adaptor" lo están utilizando, usaremos "get_dependent_adaptors()". Uno de los métodos importantes de esta clase es "configure()": es el método a implementar en los "Communicators" concretos, indicando como se configura el "Communicator".
- La clase "Server" permite crear un servidor que al comenzar a funcionar, quedará a la espera de tramas por parte de los dispositivos. Para implementar un "Server", necesitamos definir los métodos "start()" y "stop()", que serán siempre llamados después que el método "configure()". Cuando llegue una trama del dispositivo, se llamará a las funciones que han sido añadidas a través del método "add_request_callback(callback:func)", y a estas funciones se las llamará pasándolas "address" y una instancia de "Client" lista para establecer conexión con el dispositivo.
- La clase "Client" nos permitirá, por una parte, iniciar a nosotros una conexión

contra un dispositivo, y por otra, como se comentaba en el punto anterior, comenzar a "mantener" una conversación con el dispositivo, cuando éste haya sido el iniciador de la conexión (ver modelo de comunicación par exclusivo, 2.3.3).

2. Adaptors:

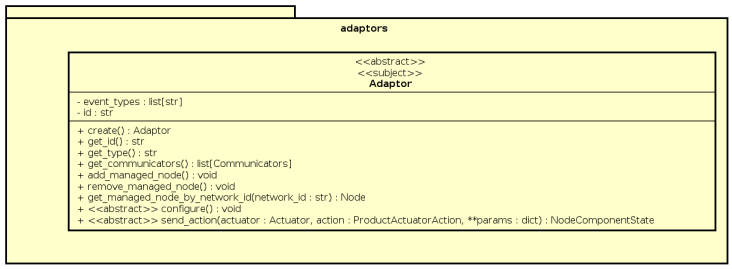

Figura 4.5: Diagrama de descomposición lógica de "core::model::adaptors"

- Las clases concretas que hereden de "Adaptor" nos permitirán implementar diferentes protocolos de comunicación. Por una parte, la clase cuenta con los métodos "add_managed_node()" y "remove_managed_node()", que nos permitirán añadir instancias de "Node" a considerar por ese "Adaptor" concreto.

  Contamos con el método "get_managed_node_by_network_id(network_id: str)", que nos permitirá recuperar una instancia de "Node" en base a un identificador que nos proporcione el dispositivo por red. Contamos con "configure()": en la implementación de este método, nos encontraremos llamadas a "add_request_callback(callback: func)".

  Los métodos "callback" que reciben "address" y "client" -hablábamos de ellos en la explicación de los "Server", "Communicator"- estarán implementados en clases de tipo "Adaptor". En las implementaciones del método "send_action(actuator: Actuator, action ProductActuatorAction, **params: dict)", nos encontraremos definido cómo se envían acciones a los dispositivos (normalmente, haremos uso de un cliente con el que enviaremos una trama).

3. Nodes:

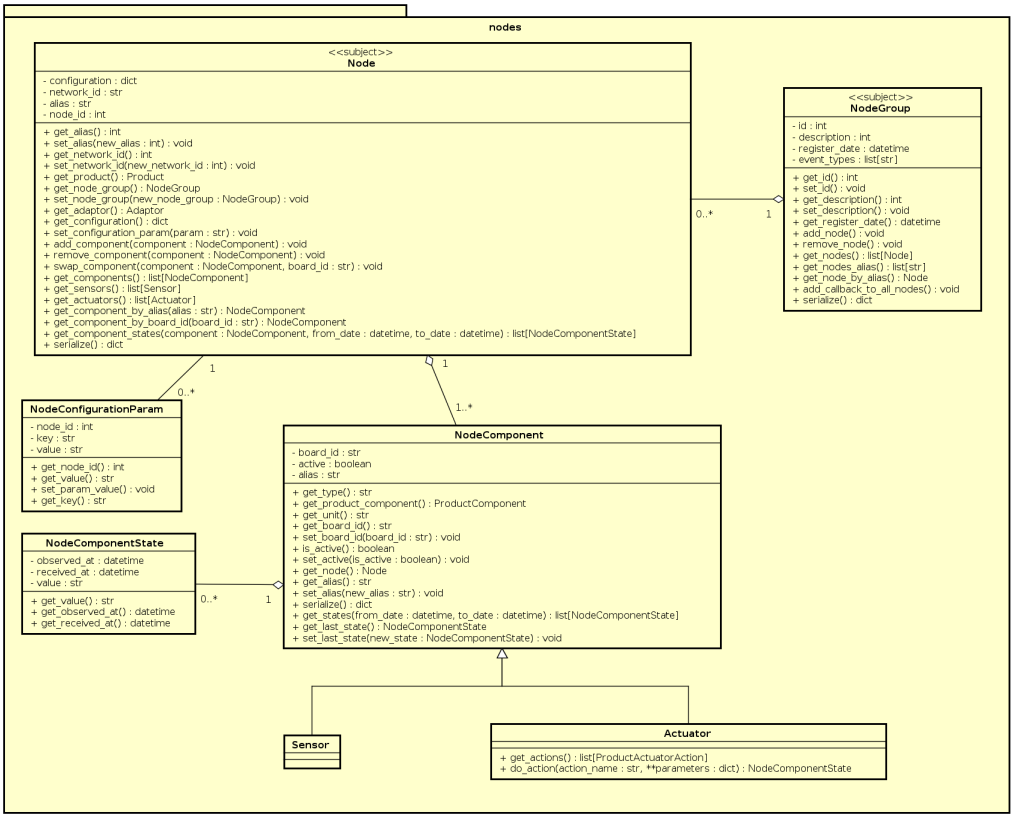

Figura 4.6: Diagrama de descomposición lógica de "core::model::nodes"

- La clase "NodeGroup" representa un grupo de nombres. Su atributo "id" es un nombre personalizado que ha de ser único en el sistema. También cuenta con una descripción que fija el usuario y permite justificar su existencia; y una fecha de creación. Nos permitirá gestionar/localizar nodos y aplicar operaciones sobre todos ellos.
- La clase "Node" representa un dispositivo real, como hemos comentado en otras secciones. Todo "Node" cuenta con un producto "Product" asociado, y ha de estar bajo un único grupo "NodeGroup". También cuentan con un "alias", que es un nombre personalizado que ha de ser único bajo el grupo en el que se encuentre "Node".

  También cuenta con "network_id", que es el identificador único que el dispositivo proporciona por la parte de la red (por ejemplo, la dirección MAC del dispositivo). Otro detalle importante es que todo "Node" está asociado a un adaptor "Adaptor". Como comentábamos en la enumeración anterior, las instancias de "Adaptor" cuentan con una serie de instancias de "Node", con el principal objetivo de ser ellos los que actualicen el estado de los componentes (a través de

"Node"). Esta instancia de "Adaptor" se usará también para enviar acciones a los dispositivos que lo soporten.

- La clase "NodeConfigurationParam" es similar a la presentada en el apartado de "Communicators", "CommunicatorParam". Ésta modela, en términos de clave-valor, la configuración del dispositivo. Se ha implementado así porque cada dispositivo tiene sus propios valores a configurar, y algunos ni siquiera cuentan con una configuración que se pueda realizar en remoto. Entre las instancias de esta clase nos encontraremos configuraciones como puedan ser intervalos de envío, políticas de energía, etc. -lo que el dispositivo que "Node" representa necesite.
- La clase "NodeComponent" representa un componente del dispositivo real. Los componentes serán sensores y/o actuadores. Cuenta con un alias, al mismo estilo que en "Node". En esta caso, será un nombre personalizado que ha de ser único bajo el "Node" concreto. Por defecto, el valor de "alias" será el mismo que el de "board_id". El atributo "board_id" es el identificador que nos permite localizar el valor de componente en una trama, o formar la petición para obtenerlo.
- Los valores que recojamos mediante los "Adaptor" y "Communicator" serán, en cualquier caso (dato de sensor o de actuador), representados con instancias de "NodeComponentState". Toda instancia de "NodeComponentState" cuenta con una fecha de observación y una fecha de recepción. En muchas ocasiones -sobretodo cuando trabajamos con dispositivos de bajo consumo- recibimos varias muestras "de repente" (tiempo de recepción), que fueron tomadas justo en el "tiempo de observación". Otro detalle importante a tener en cuenta es el tipo del atributo "value". Es "str", dado que muchos dispositivos reportan datos que no son numéricos. El tipo del valor que emite el dispositivo estará especificado y disponible en la instancia de producto.

4. Products:

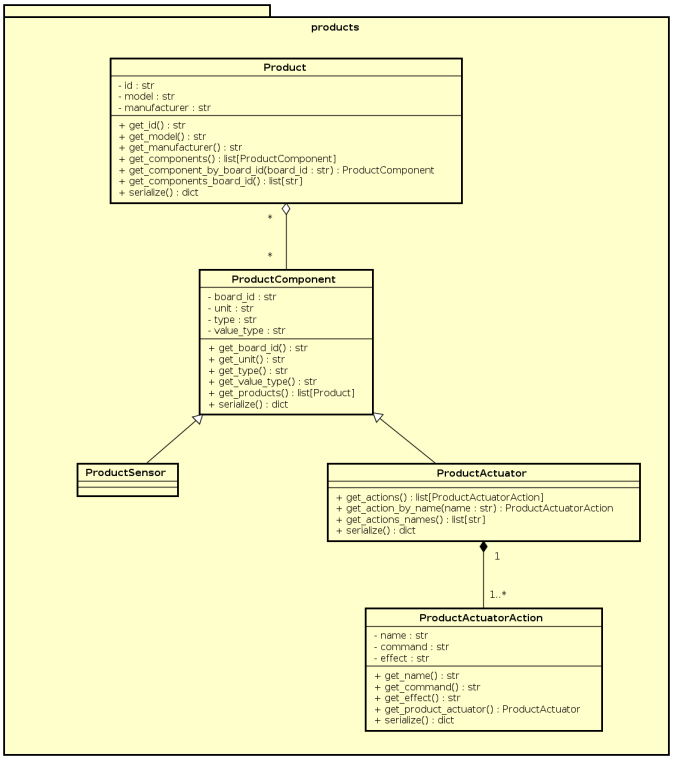

Figura 4.7: Diagrama de descomposición lógica de "core::model::products"

a) La clase "Product" contiene la especificación necesaria para dar de alta un "Node" en nuestra particular red.

b) La clase "ProductComponent" especifica cómo es el componente del que recibiremos datos o al que enviaremos acciones. Como se comentaba anteriormente, en éste se indican el tipo de valor que vamos a consumir (str, int, float), y sus unidades (celsius, lúmenes, milivoltios). También se cuenta aquí con "board_id".

c) La clase "ProductActuatorAction" representa las acciones que el actuador es capaz de llevar a cabo. En el atributo "name" se puede poner lo que uno precise, siempre y cuando no haya dos acciones diferentes con el mismo nombre bajo un mismo actuador. En el atributo "command", guardaremos un comando parametrizado cuyos parámetros serán resuelto en el método "send_action" del "Adaptor" que gestione este tipo de dispositivos.

5. Managers:

Figura 4.8: Diagrama de descomposición lógica de "core::model::managers"

- Como podemos observar en la figura 4.3.1, contamos con un "manager" por bloque conceptual (Communicator, Adaptor, Node y Product). Este paquete supone ser el punto de entrada al modelo. Pone a disposición de las capas de fuera el uso de todas las clases que hemos explicado hasta ahora. Hacen uso del ORM para instanciar todo el modelo, por lo que no necesitan conocerse entre ellos. Los managers, como veremos en deployment, son clases fundamentales, y la especificación de la API será a grandes rasgos la especificación de los métodos de estas clases.

Para continuar con esta sección, vamos a ver la descomposición modular de los elementos principales de "core" que están fuera de "model":

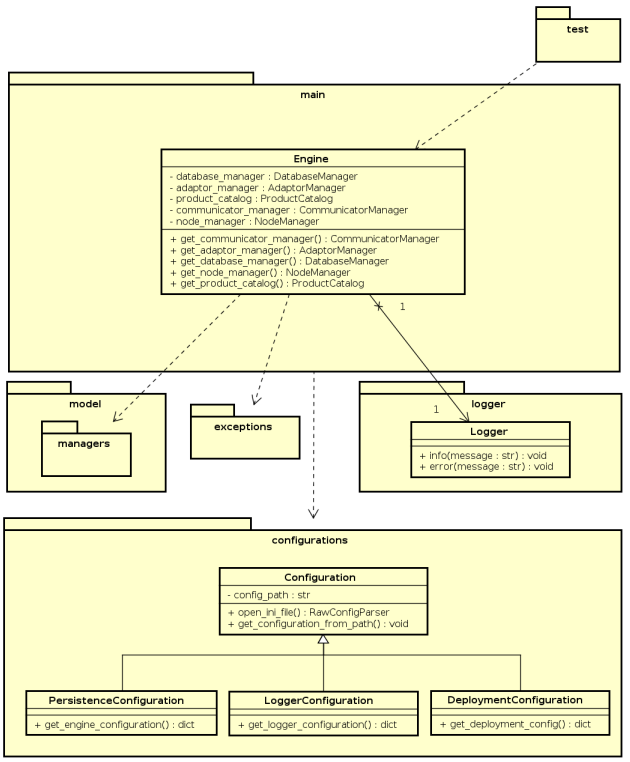

Figura 4.9: Diagrama de descomposición lógica de "core::main, core::logger, y core::configurations"

- Como podemos observar en la figura, el módulo "main" contiene la clase "Engine". Esta clase es el punto de instanciación principal del sistema. Al ser instanciado, pone todo en marcha.

  Primero carga la configuración, en base a esta instancia el servicio de logs y se conecta a la base de datos, y con la base de datos cargada, se pone a funcionar el ORM, instanciando el modelo de dominio. Al estar instanciado el modelo, contamos con todos los elementos del sistema cargados y con la configuración que hubiera en la base de datos.

  Por último, llama al método "start()" los servidores. Las tramas comienzan a llegar, los adaptors comienzan a tratarlas. Todo queda listo para ser usado. Como veremos más adelante, la instancia de este objeto es utilizada por la capa "deployment", por lo que todas las funcionalidades del sistema quedan expuestas y dispuestas a ser utilizadas por otros programas.

- Por simplificar, en el paquete "model" sólo se ha incluido el paquete "managers", que resulta ser el último paquete explicado de "model" (ver 4.3.1). "Engine", la instanciar

los managers, les pasa una instancia de "DatabaseManager". Mediante esta instancia, los "managers" se encargan de instanciar el resto del sistema. La creación de objetos se va relegando a las diferentes clases clave, por lo que la complejidad de instanciación del sistema es mínima; dado que la complejidad está muy repartida. En la sección "Acerca de la implementación" (4.3.1), que se encuentra más adelante, hablaremos de ello.

- En el paquete "configurations" contamos con clases que nos ayudan a cargar y convertir los parámetros de configuración de los distintos elementos en diccionarios. Hablaremos en implementación de cómo se carga la configuración.
- *Queda pendiente el tema de excepciones!*

Para acabar esta sección, es preciso hablar del mecanismo establecido para <<subject >>y de la localización de "DatabaseManager", clase responsable de proporcionar las instancias que el ORM necesita para funcionar. Por ello, mostraremos la descomposición modular de los paquetes especiales "patterns" e "__init__.py":

1. Patterns (Subject y otros):

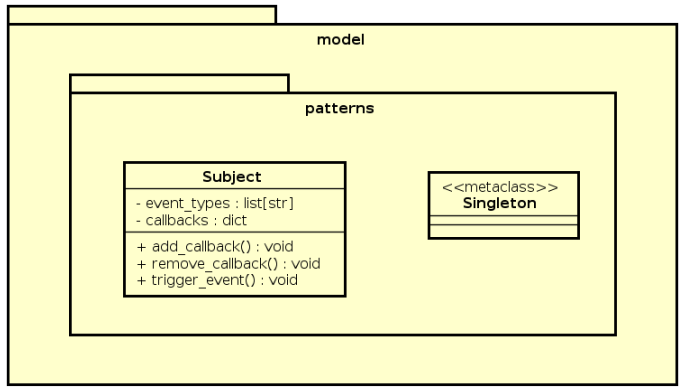

Figura 4.10: Diagrama de descomposición lógica de "core::model::patterns"

- Subject es la clase padre de la que heredan todas las clases marcadas con el esteoreotipo <<subject >>. Esto las proporciona la capacidad de lanzar eventos. Cada clase <<subject >>tiene un tipo de eventos predefinido, cuyo significado se describe en la documentación del código, y veremos más adelante.

  Podemos observar en la figura que "Subject" cuenta con un atributo "callbacks", que es de tipo "dict". En este diccionario, se guarda como clave el tipo de evento, y el valor, es una lista de funciones (añadidas mediante el método "add_callback(event_type:str, callback:func)").

  Todas estas funciones serán llamadas invocando al método "trigger_event(event_type:str,

**kwargs: dict)". Los párametros (kwargs) que han de tener las "callbacks" según el tipo de evento se encuentran especificados en la autodocumentación del código.

- Singleton es una metaclase. En la sección de implemetación hablaremos de como utlizarla. Básicamente, esta clase nos permite marcar otras clases como Singleton, y de una manera transparente, el patrón se aplica, y sólo existe una instancia de la clase marcada en todo el sistema.

2. __init__.py (DatabaseManager):

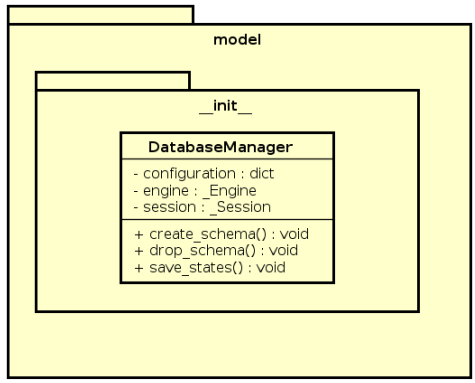

Figura 4.11: Diagrama de descomposición lógica de "core::model::__init__" (DatabaseManager, ORM)

- La clase "DatabaseManager" proporciona las instancias necesarias de objetos de SQLAlchemy, el toolkit de base de datos escogido para integrar el modelo de IoTGate con un ORM.

  SQLAlchemy establece conexión con las bases de datos a través del objeto "_Engine" (no confundir con la "Engine" de IoTGate (core::main:Engine)). El ORM de SQLAlchemy se gestiona a través de instancias de la clase "Session". "DatabaseManager" crea y proporciona acceso a estos objetos. Hablaremos de ello con más detalle en la sección de implementación.

Para acabar esta sección, vamos a hablar del diagrama relacional que da soporte al ORM. Este diagrama fue diseñado aparte, y luego se trabajó en que el ORM se adaptara a él.

Como comentaremos en la sección de implementación, no todos los atributos de las clases del modelo precisan ser persistentes.

Por otra parte, incluimos este diagrama en esta sección a pesar de no formar parte de la descomposición modular del sistema dado que da un soporte fundamental al ORM, que como ya se ha dicho, está completamente integrado con el modelo de dominio del sistema.

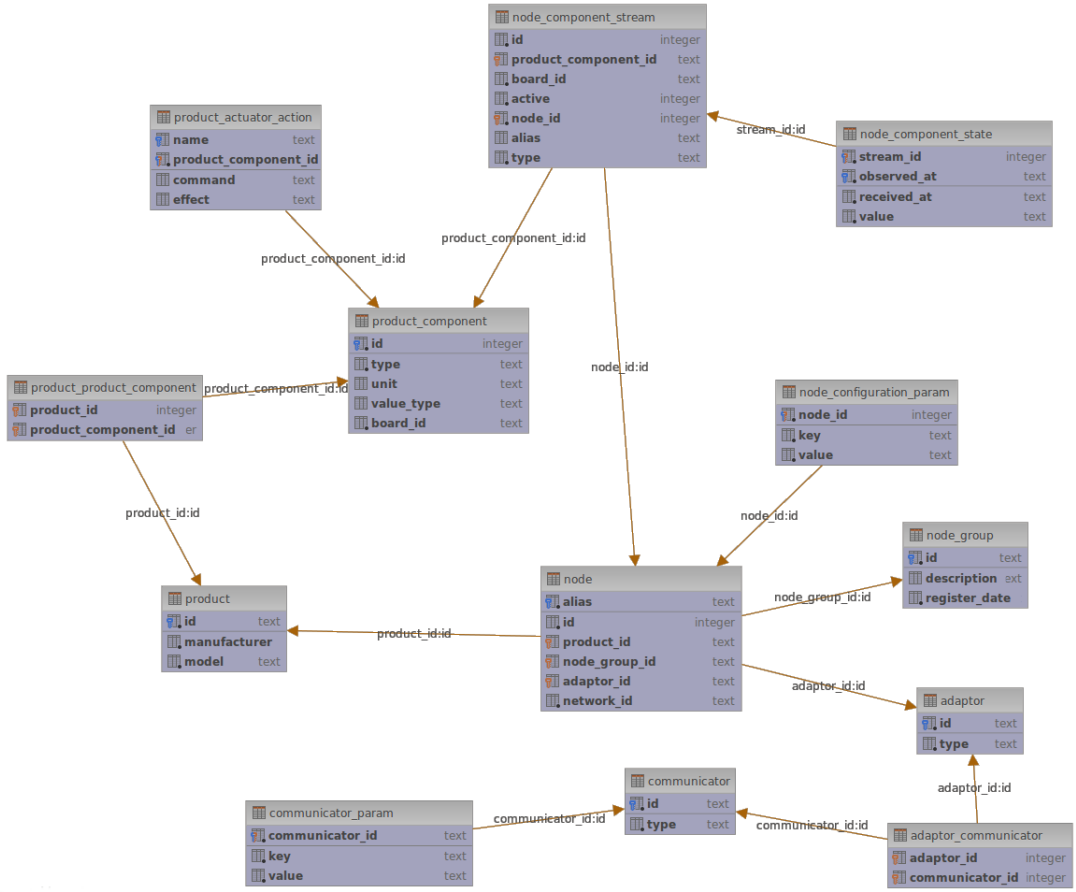

Figura 4.12: Diagrama relacional de la base de datos

Vamos a comentar los detalles más interesantes de este diagrama relacional:

- Tanto en las tablas para "Communicator" y "Adaptor", contamos con un atributo "type". Este atributo es especialmente importante; guarda de que tipo son tanto el "adaptor" como el "communicator" concreto. Este dato se utilizará para cargar dinámicamente las clases que especializan las abtractas que hemos visto en "core::model" (4.3.1).
- La tabla "node_component_stream" representa al componente de un nodo concreto. Bajo ella, están todas las relaciones de la tabla "node_component_state", donde se guardan los datos. Los "stream" están relacionados con un componente del catálogo para poder conocer las unidades (celsius, lúmen, etc.) y el tipo de dato (str, float, integer) de los datos que están bajo "stream", cuyos valores se guardan siempre en string.
- Como podemos comprobar fijándonos en "node", cada "node" está relacionado

con un grupo de nodos, y con el "adaptor" que es capaz de recoger y enviar tramas mediante los "communicators" pertinentes, y "poblar" de datos el objeto "Node", aparte de proporcionar la capacidad de acción.

- Tanto "node" como "communicator" cuentan con una configuración dependiente de concreciones (no todas las configuraciones de los dispositivos son iguales, y no todas las redes con las que trabajamos tienen los mismos parámetros de configuración.) Por ello, se ha establecido una especie de metamodelo clave-valor. Las claves y los tipos de los valores han de ser conocidos -y documentados- por el desarrollador del conjunto de "communicators" y/o "adaptors" que necesite un dispositivo para ser entendido. Veremos con más detalle esto en la sección "Capa communicators".

Dependencias

La capa "core" no presenta dependencias con el resto de elementos del sistema. Sin embargo, todas las otras capas del sistema (communicators, adaptors, deployment) dependen de ella:

1. La capa "communicators" depende de "core::model::communicators::Client" y "core::model::communicators::Server". Veremos esta dependencia con mayor detalle en la siguiente sección.
2. La capa "adaptors" depende de "core::model::adaptors::Adaptor". Veremos más adelante en qué consiste la dependencia con detalle.
3. La capa "deployment" depende de "core::main::Engine". Veremos en qué consiste la dependencia en la última sección.

Acerca de la implementación

Como comentábamos en secciones anteriores, durante el desarrollo de esta sección hablaremos de los puntos clave de la implementación. Para comenzar, hablaremos de la clase "Engine". Esta supone ser el punto principal de instanciación de todo el sistema:

```
1  def __init__(self, configuration_path, reset_database=False,
2               before_start=None, after_start=None):
3      Subject.__init__(self)
4      self.work = True
5      self.configuration_path = configuration_path
6
7      # We prepare the environment
8      self._create_logger_service()
9      self.logger.info('IoTGate %s starting...' % __version__)
10
11     self._create_persistence_management()
12     if reset_database:
13         self._reset_database()
14
15     self._create_model_managers()
16     self._create_deployment_helpers()
17
18     if before_start:
19         before_start(self)
20
21     self.communicator_manager.start_communicators()
22
23     if after_start:
24         after_start(self)
```

Figura 4.13: Constructor de la clase "Engine"

Vamos a comentar los aspectos fundamentales:

- "Engine" hereda de "core::model::patterns::Subject", lo que significa que es una clase que lanza eventos.
- Como podemos observar a partir de la línea 7, vamos instanciando todos los elementos del sistema:
  - Primero, creamos el servicio de log, y acto seguido, probamos que funciona escribiendo el primer mensaje, que contiene la versión del software.
  - Depués, continuamos instanciando los elementos que necesitamos para utilizar la persistencia. A partir de ese punto contamos con el "_Engine" y la "_Session" de SQLAlchemy. El objeto _engine nos permite establecer conexión con la base de datos. El objeto _session nos permite la instanciación y uso del ORM.
  - Con estos dos elementos de SQLAlchemy listos para funcionar, instanciamos los "managers" del modelo. Esta instanciación provoca la creación de todos los objetos del dominio.
  - Con todo el negocio instanciado, instanciamos los elementos de la capa "deployment".
- Como podemos observar, contamos con tres argumentos espciales: "reset_database", "before_start" y "after_start":
  - "reset_database" es de tipo boolean, y nos permite borrar todos los datos de la base y volver a crear las tablas, vacías. Esto sucede cuando se instancia el "Engine" con esta variable en valor "True".

- Tanto “before_start” como “after_start” nos permiten pasar funciones que se hayan de ejecutar antes o después de comenzar a funcionar. Principalmente viene bien para pruebas; como puedan ser introducir una población en la base de datos o pasar un conjunto de tests.

- Con todo instanciado ya, pedimos al manager de communicators que comience a escuchar (es decir, llamamos al método “start()” de todos los communicators de tipo “Server”)

Podemos concluir con que instanciar la clase “Engine” supone iniciar todo el sistema. Otro punto importante es la configuración. Tiene el siguiente formato:

```
[logger]
log_path=./log/
verbose=True
own=False
syslog=False

[persistence]
driver=sqlite
database_file=./db/iotgate.db

[persistence-mysql]
driver=mysql
database_name=iotgatetest
user=*******
password=*******
host=localhost
port=3306

[persistence-psql]
driver=postgresql
database_name=netbows_db
user=*****
password=*****
host=localhost
port=5432
```

Figura 4.14: Formato de la configuración

La configuración ha de contar con dos secciones:

1. “logger”: indica la configuración de log a utilizar:
   - “log_path” y “own” van estrechamente unidos. Si “own” está en True, el sistema escribirá en el directorio de log indicado los ficheros “iotgate.log” e “iotgate.err”. En el primer fichero escribirá eventos del sistema (como la hora de comienzo, que communicators de tipo Server se han levantado, la conexión de un dispositivo concreto...), y en el segundo escribirá los errores que sucedan.
   - el valor de syslog indica a IoTGate si ha de escribir o no en el servicio de log de GNU/Linux “syslog”.
   - “verbose” indica si el log ha de ser escrito también por pantalla. Esto resulta útil para tareas de depuración.

2. “persistence”: indica la configuración de base de datos:
   - Sólo cargará una configuración, la que esté marcada como “persistence”. En la figura, observamos que se cargaría la configuración para funcionar con SQLite.
   - Las otras dos se adjuntan por tener un ejemplo de como trabajar con otros SGBD.

Ahora, vamos a comentar sobre la integración con el ORM. Para ello, vamos a tomar un ejemplo de “core::model::nodes”, la clase Node:

```
class Node(Base, Subject):
    event_types = ['new_state', 'change', 'component_added',
                   'component_removed', 'component_swapped']

    __tablename__ = 'node'

    id = Column(Integer, autoincrement=True, primary_key=True)
    product_id = Column(String(128),
                        ForeignKey('product.id',
                                   onupdate='CASCADE',
                                   ondelete='SET NULL'))
    product = relationship('Product')
    node_group_id = Column(String(128),
                           ForeignKey('node_group.id',
                           onupdate='CASCADE',
                           ondelete='CASCADE'))
    node_group = relationship('NodeGroup')
    adaptor_id = Column(String(128), ForeignKey('adaptor.id',
                                                onupdate='CASCADE',
                                                ondelete='SET NULL'))
    adaptor = relationship('Adaptor')
    params = relationship('NodeConfigurationParam',
                          collection_class=attribute_mapped_collection('key'))
    configuration = association_proxy('params', 'value',
                    creator=lambda k, v: NodeConfigurationParam(key=k, value=v))
    components = relationship('NodeComponent')
    network_id = Column(String(128), nullable=False)
    alias = Column(String(128), nullable=False)

    # Node alias must be unique under a concrete node group
    UniqueConstraint(node_group_id, alias, name='unique_node_in_ngroup')

    def __init__(self, **kwargs):
        Base.__init__(self, **kwargs)
        self._init_on_load()

    @reconstructor
    def _init_on_load(self):
        Subject.__init__(self)
```

Figura 4.15: Integración de la clase “Node” con el ORM de SQLAlchemy

En la figura se muestra parte de la clase Node. En este fragmento, podemos observar varias cosas:

- “Node” hereda de las clases “Base” y “Subject”. Al heredar de “Base”, indicamos al ORM que esta clase ha de ser representada con una tabla en la base de datos. Como ya hemos explicado, “Subject” indica que esta clase puede lanzar eventos. Sobre los eventos, hablaremos más adelante.

- Cada atributo de la clase puede ser:

  1. una columna de la base de datos. Para indicarlo, se hace uso de la clase "Column" de SQLAlchemy. Una vez instanciado, el uso de ese atributo es transparente: por ejemplo, el atributo "id" será más tarde usado como si de un valor de tipo string se tratara.
  2. otro objeto del modelo de dominio. Como hemos visto anteriormente, toda instancia de "Node" pertenece a un grupo y cuenta con un producto que establece sus características. Para contar con la instancias de "NodeGroup" y "Product" correspondientes, se hace uso del método "relationship(<nombre de la clase >)". Al instanciarse, disponemos del objeto de dominio para su uso.
  3. un conjunto de objetos de domino mapeados y convertidos a otro tipo de objeto. Para poder disponer y utilizar la configuración como si de un diccionario de Python se tratara, utilizamos la función de SQLAlchemy "association_proxy()" (línea 24 en la figura). Este tipo de asociación se presenta también en la clase "Communicator", con el mismo objetivo, contar con una configuración cuyo tipo sea diccionario.

- Como último comentario, es necesario en algunos casos utilizar la anotación reconstructor. Cuando comienza a funcionar el sistema y los managers hacen uso del ORM para instanciar el modelo, la construcción de estos es responsabilidad del ORM de SQLAlchemy. Éste no construye los objetos por la vía tradicional, invocando a __init__. Por ello, hay que definir un método en el que se instancien los elementos de la clase que se encuentren en el constructor habitual.

Otro tema de interés acerca de la integración con el ORM tiene que ver con los mecanismos de herencia. Como Node no hereda de ningún elemento de dominio, no nos sirve como ejemplo. Vamos a ver un ejemplo de esto observando la clase "NodeComponent", que resulta ser la clase padre de "Sensor" y "Actuator":

```
1  class NodeComponent(Base):
2      __tablename__ = 'node_component_stream'
3
4      id = Column(Integer, autoincrement=True, primary_key=True)
5      product_component_id = Column(String(128),
6                                    ForeignKey('product_component.id',
7                                               onupdate='CASCADE', ondelete='SET NULL'))
8      product_component = relationship('ProductComponent')
9      board_id = Column(String(128), nullable=False)
10     active = Column(Boolean(), default=True, nullable=False)
11     node_id = Column(Integer,
12                      ForeignKey('node.id', onupdate='CASCADE', ondelete='CASCADE'),
13                      nullable=False)
14     node = relationship('Node')
15     states = relationship('NodeComponentState', lazy='dynamic')
16     alias = Column(String(128))
17
18     type = Column(String(128), nullable=False)
19     __mapper_args__ = {'polymorphic_on': type}
```

Figura 4.16: Fragmento de la clase "NodeComponent"

En esta ocasión, observamos:

- Contamos con un atributo “type”, mediante este guardaremos en la tabla de la base de datos el tipo del objeto. Cuando lo recuperemos, necesitaremos construirlo con el tipo concreto; no tendremos nunca un objeto que sea solo de tipo “NodeComponent”: siempre será “Sensor” o “Actuator”. Además, el conjunto es obligatoriamente disjunto (ningún “Sensor” puede ser a la vez “Actuator”).

  Mediante la sentencia __mapper_args__ = {’polymorphic_on’: type}, indicamos al ORM que tendrá que decidir como instancia el elemento fijándose en el valor de “type”.

- Observemos ahora la clase “Sensor”:

```
1  class Sensor(NodeComponent):
2      __mapper_args__ = {'polymorphic_identity': 'sensor'}
3
4
5  class Actuator(NodeComponent):
6      __mapper_args__ = {'polymorphic_identity': 'actuator'}
7
8      def get_actions(self) -> list:
9          return self.get_product_component().get_actions()
10
11     def do_action(self, name, **params):
12         action = self.get_product_component().get_action(name)
13         new_state = self.get_node().get_adaptor()\
14                                    .send_action(self, action, **params)
15         self.set_last_state(new_state)
16
17         return new_state
```

Figura 4.17: Implementación de las clases “Sensor” y “Actuator”

  Observamos la sentencia __mapper_args__ = {’polymorphic_identity’: ’sensor’}. Aquí es donde indicamos el valor que permite al ORM conocer de que tipo ha de ser la instancia que tiene que crear cuando recupera una fila de la tabla ’node_component_stream’.

- Otro detalle interesante a tener en cuenta sobre “NodeComponent” es su relación con “NodeComponentState”. La tabla de la base de datos que da soporte a “NodeComponentState” (node_component_state) es la tabla que más filas va a tener de toda la base de datos, dado que es en esta tabla donde se guardan los datos que provienen de la red de sensores y actuadores.

  Por ello, establecemos con la sentencia “states = relationship(’NodeComponentState’, lazy=’dynamic’)” que la recuperación de filas de esta tabla sea “lazy”. Así, los objetos “NodeComponentState” serán creados bajo demanda, y no nos traeremos todos siempre (podría ser un gran problema hacer eso: supondría un problema de rendimiento tan grave que provocaría que el sistema no funcionara cuando se contara con una cantidad grande de datos).

Para continuar esta sección, resulta imprescindible hablar del contenido de “core::model::patterns” y del sistema de eventos:

```python
class Subject(object):
    event_types = []

    def __init__(self):
        self.callbacks = {}
        for event_type in self.event_types:
            self.callbacks[event_type] = []

    def add_callback(self, event_type, callback):
        if callback not in self.callbacks[event_type]:
            self.callbacks[event_type].append(callback)

    def remove_callback(self, event_type, callback):
        try:
            self.callbacks[event_type].remove(callback)
        except ValueError:
            pass
            # TODO: Raise exception

    def trigger_event(self, event_type, **kwargs):
        for callback in self.callbacks[event_type]:
            try:
                callback(**kwargs)
            except Exception as err:
                print("Exc from pattern:", err)
                print(traceback.format_exc())


class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]
```

Figura 4.18: Contenido del módulo “patterns.py”

1. La clase “Singleton” es realmente una metaclase. Se utiliza de la siguiente manera:

```python
class MiClase(object, metaclass=Singleton):
    pass
```

Figura 4.19: Uso de la metaclase “Singleton”

Como podemos observar en la figura, basta con indicarlo al crear una clase cualquiera.

Como podemos observar en la implementación de la clase Subject, cada clase que herede de ella contará con una serie de funcionalidades:

- “event_types” es una lista de str cuyos valores son los nombres clave de tipo de eventos. Este atributo ha de ser sobreescrito en las clases que hereden. Vamos a ver un ejemplo con “Node”:

```
"""
    Events
        new_state: called after receive a new state.
            param - component: NodeComponent => The component.
            param - data: NodeComponentState => The state received.
        change: called after change an owner field of the node.
            param - type: str => The name of the field changed.
            param - old_value => The previous value of the field.
            param - new_value => The new value of the field.
        component_added: called after plug a component to the node.
            param - component: NodeComponent => The component
                                                connected to the node.
            param - node: Node => The current node.
        component_removed: called after unplug a component from the node.
            param - component: NodeComponent => The component
                                                unconnected from the node.
            param - node: Node => The current node.
        component_swapped: called after swap a component in the same node.
            param - component: NodeComponent => The component swapped.
            param - old_board_id: str => The previous board
                                         identifier of the component in the node.
            param - new_board_id: str => The new board identifier of
                                         the component in the node.
            param - node: Node => The current node.
    """
    event_types = ['new_state', 'change', 'component_added',
                   'component_removed', 'component_swapped']
```

Figura 4.20: Eventos de "Node"

- Observemos el método "add_callback(event_type: str, callback:func)". "event_type" será uno de los strings de la lista. "callback" será una función que será ejecutada al invocar "trigger_event(event_type, **kwargs)".
- Todas las clases "Subject" cuentan con una documentación dentro del código en el que se indica el significado del evento y los parámetros que recibirá la función "callback" que se añada, como se puede observar en la figura.

Para acabar esta sección, vamos a hablar de la parte "más reflexiva" de la capa core. Como hemos comentado a lo largo de esta sección, las capas "adaptors" y "communicators" dependen de "core", que es quien se encarga de crearlas.

Para este fin, contamos con dos métodos "create()" estáticos, uno en la clase "core::model::adaptors ::Adaptor" y otro en la clase "core::model::communicators::Communicator":

```
    @staticmethod
    def create(adaptor_id: str, adaptor_type: str, communicators: list):
        module_name = ''
        for item in adaptor_type.split('.')[:-1]:
            module_name += '.' + item

        class_name = adaptor_type.split('.')[-1]
        module = __import__('iotgate.adaptors' + module_name, fromlist=[class_name])

        adaptor_class = getattr(module, class_name)
        adaptor = adaptor_class(id=adaptor_id, communicators=communicators)

        return adaptor
```

Figura 4.21: Método factoría para instancias de "Adaptor"

```
    @staticmethod
    def create(comm_id: str, comm_type: str, configuration: dict):
        module_name = ''
        for item in comm_type.split(".")[:-1]:
            module_name += "." + item
        class_name = comm_type.split(".")[-1]
        module = __import__('iotgate.communicators.transport'\
                            + module_name, fromlist=[class_name])

        comm_class = getattr(module, class_name)
        communicator = comm_class(comm_id, configuration)

        return communicator
```

Figura 4.22: Método factoría para instancias de "Communicator"

Como podemos observar, ambos son prácticamente iguales; con la salvedades:

- Los "adaptors" reciben una lista de communicators.
- Las rutas para importar módulos son distintas:
  1. "iotgate.communicators.transport" es la ruta donde estarán las implementaciones concretas de "Client" y "Server", como veremos en la siguiente sección.
  2. "iotgate.adaptors"es la ruta donde estarán las implementaciones concretas de "Adaptor".

Disponemos de los tipos a establecer (adaptor_type y comm_type) preguntando por ello a los "managers" correspondientes (get_communicator_types() y get_adaptor_types(), devuelven listas de strings).

En este punto del sistema aplicamos reflexión, permitiendo así la existencia de varias distribuciones de IoTGate, dependiendo del contenido de sus capas "communicators" y "adaptors". También es fácil comprobar que la ruta de estas capas está prefijada, lo que obliga a mantener una estructura mínima.

4.3.2. Capa "communicators"

La capa "communicators" supone ser la capa de más bajo nivel de abstracción de todo el sistema. En ella, se encuentran las implementaciones de "Client" y "Server", las clases abstractas que contiene "core::model::communicators". Estas implementaciones nos permiten conectarnos con diferentes dispositivos de Tipo I,

1. a través de un módulo trasnceptor,
2. directamente con un dispositivo, a través de un bus de comunicación en serie
3. a través de herramientas que nos proporcione el sistema operativo (como sockets para dispositivos TCP/IP, o Bluetooth)

La capa es completamente dinámica: los módulos a instanciar/utilizar son detectados en tiempo de ejecución, y ni siquiera es necesaria su existencia para instanciar el sistema. Como veremos en la sección que viene a continuación, se proporciona una estructura a seguir para facilitar al sistema (a "core" en este caso) la detección y categorización de los elementos a cargar.

Arquitectura lógica

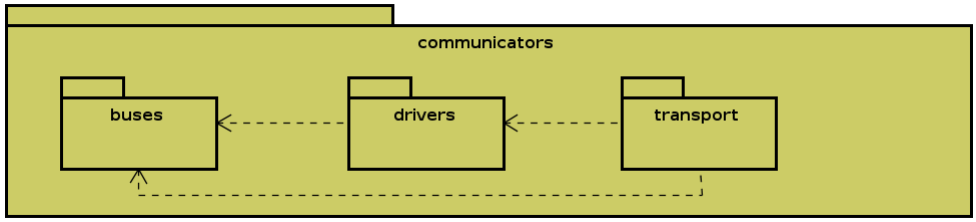

Figura 4.23: Diagrama de arquitectura lógica de "communicators"

Como podemos observar en la figura anterior, la arquitectura lógica de communicators cuenta con tres bloques fundamentales:

- Buses: contiene las implementaciones necesarias para hacer uso de la comunicación en serie. Nos servirá para comunicarnos con módulos transceptores y dispositivos de Tipo I directamente.
- Drivers: contiene las implementaciones de los controladores necesarios para hacer uso de los módulos transceptores.
- Transport: contiene las implementaciones de "Client" y "Server". Dependiendo de la necesidad que tengamos a la hora de integrarnos con un dispositivo, la fuente de datos de las clases de "transport" serán:
  - clases contenidas en el namespace "communicators::buses"
  - clases contenidas en el namespace "communicators::drivers"
  - clases proporcionadas por Python 3, que nos permitan hacer uso de recursos para comunicación que ofrezca el sistema operativo (como un socket tcp/ip, por ejemplo).

Descomposición modular

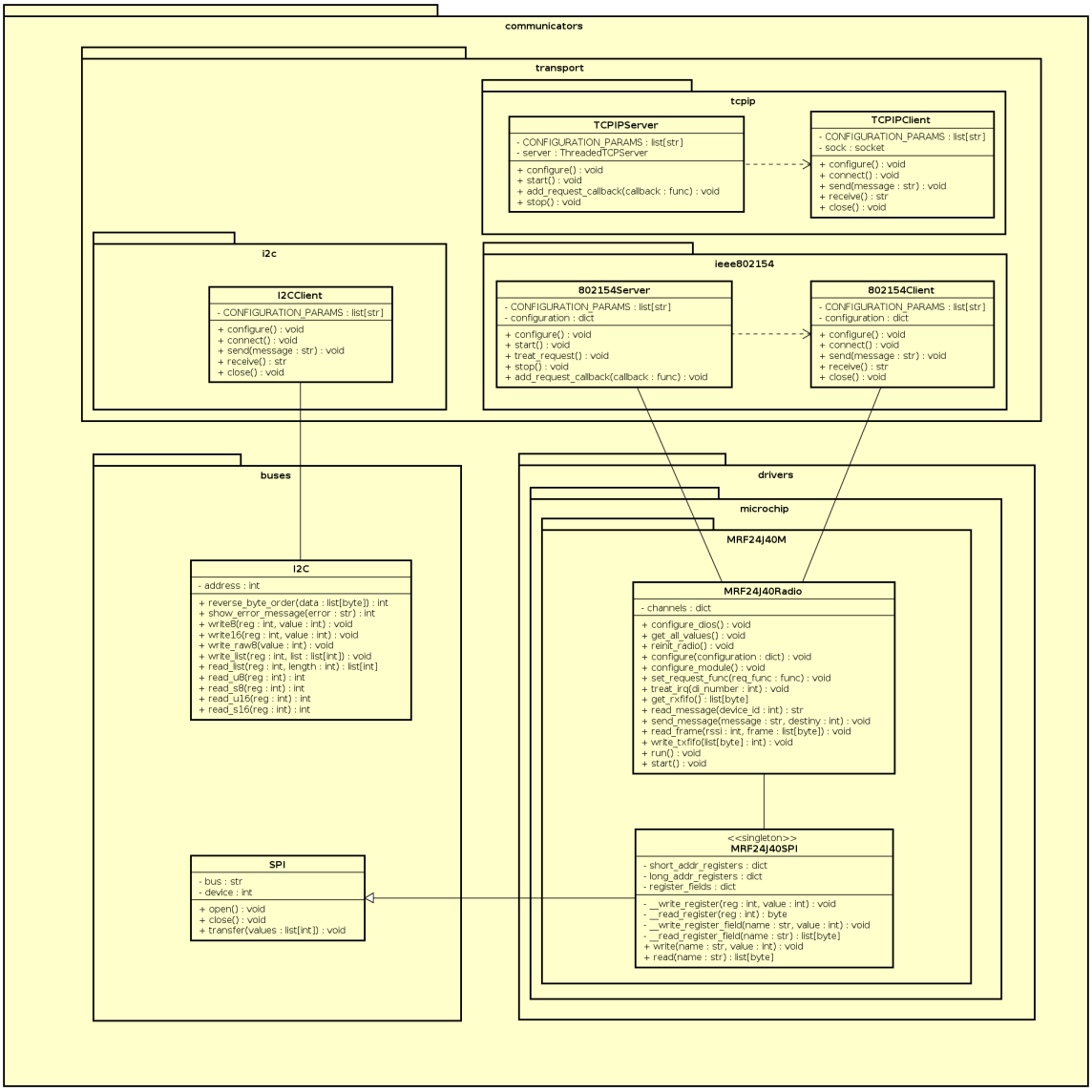

Figura 4.24: Descomposición modular de “communicators”

Teniendo en cuenta los comentarios sobre el dinamismo de esta capa en la sección anterior, la descomposición modular que aquí se presenta resulta ser una “fotografía” de una distribución concreta de IoTGate. Esta distribución cuenta con los tres casos representativos (para la conexión con dispositivos de Tipo I) que ya hemos comentado. Partiendo del namespace

"communicators::transport", resulta interesante fijarse en:

- Cada tecnología para la conexión tiene su propio namespace.
- El namespace "communicators::transport::tcpip" contiene implementaciones de "Client" y "Server", que hacen uso de la implementación de socket que proporciona Python3.
- El namespace "communicators::transport::ieee80154" contiene implementaciones de "Client" y "Server" que hacen uso de la implementación del controlador del módulo transceptor situado en "communicators::drivers::microchip::MRF24J40M"
- El namespace "communicators::transport::i2c" contiene implementaciones de "Client" y "Server" que hacen uso de la implementación del controlador del bus i2c, que a su vez hace uso de librerías de Python3 que lidian con módulos del sistema operativo.

El primer detalle importante es que el uso que se haga de la capa "communicators" en "adaptors" será siempre a partir de las clases ubicadas en "communicators::transport", es decir, siempre se hará a través de las implementaciones de "Client" y "Server". Nos encontraremos estas implementaciones agrupadas bajo módulos cuyo nombre representa a una tecnología de comunicación (tcpip, ieee802154, i2c).

Fijándonos en "communicators::drivers", podemos observar que el orden a seguir para establecer los namespaces es "communicators::drivers::<fabricante del módulo>::<modelo del módulo transceptor>". Dentro de ese namespace, el desarrollador es libre de establecer las clases o jerarquías que considere oportunas. La integración será completa cuando implemente las clases "Server" y/o "Client" a partir de su controlador.

Como último comentario antes de cerrar esta sección, vamos a fijarnos en la implementación de los clientes y servidores que están contenidos en "communicators::transport".

En el diagrama, podemos observar que todos ellos tienen un atributo llamado "CONFIGURATION_PARAMS". Éste es una lista de strings, en la que se indica que parámetros hay que configurar para poder hacer uso de los "communicators".

Por ejemplo, en los communicators "tcpip", "CONFIGURATION_PARAMS" tiene el valor "["host", "port"]", y en "ieee802154" el valor es "["channel", "controller_id", "pan_id"]". Esto resulta necesario cuando se quieren validar y crear communicators desde capas superiores. Dado el carácter heterogéneo de estas configuraciones, resulta necesario para modelarlas utilizar pares clave-valor. Todo valor establecido para un parámetro de configuración tiene el tipo string. La responsabilidad de validar y convertir estos parámetros a sus tipos correspondientes es de las implementaciones concretas de "Client" y "Server".

Dependencias

Las principal dependencia que presenta la capa "communicators" es con la capa "core":

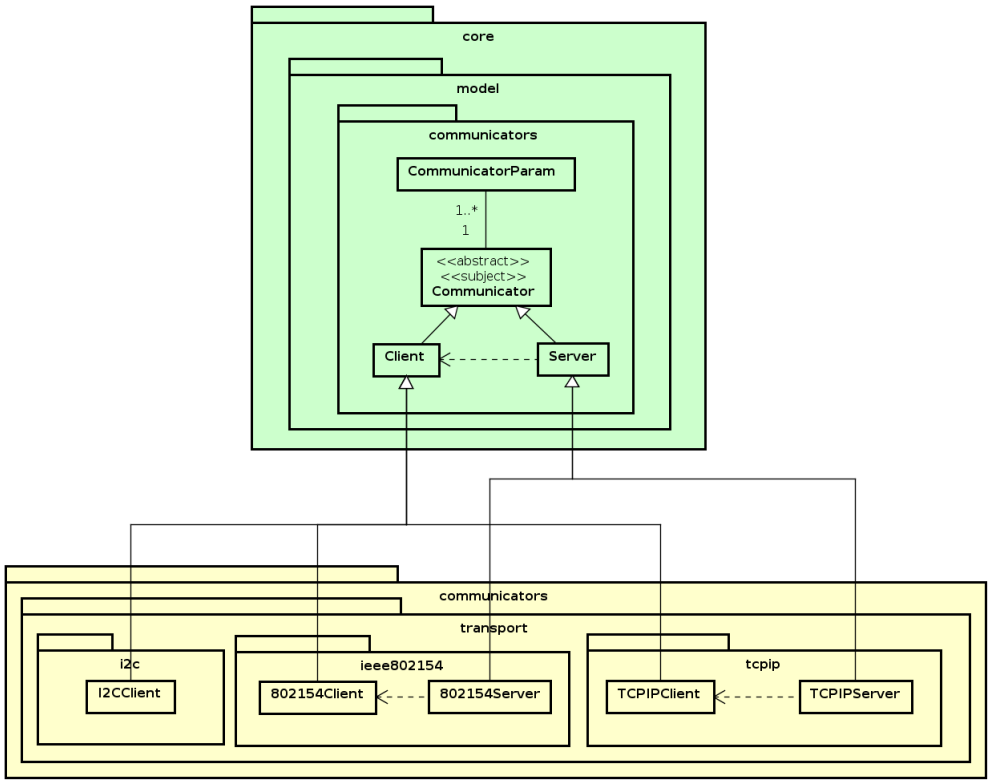

Figura 4.25: Diagrama de dependencia de “communicators” con “core”

Como podemos observar en la figura anterior, la principal relación de dependencia que existe entre la capa “communicators” y “core” son las implementaciones de las clases abstractas “Client” y “Server”.

Acerca de la implementación

Como hemos visto a lo largo de las secciones anteriores, aunque la capa “communicators” es dinámica y puede variar dependiendo de la distribución, mantiene una estructura básica (buses, drivers y transport). El paquete que lidia con la siguiente capa (“adaptors”) es “transport”, por lo que vamos a comenzar el estudio de la implementación por él.

Dentro de “communicators::transport”, contamos con implementaciones de “core::model ::communicators:: Client” y “core::model::communicators::Server”, agrupados en módulos Python cuyo nombre es una referencia a la tecnología que emplean para comunicarse. Vamos a ver un ejemplo:

```python
class IEEE802154Server(Server):

    CONFIGURATION_PARAMS = ['controller_id', 'pan_id', 'channel']

    def __init__(self, id: str, configuration: dict):
        self.radio = None
        self.working = False
        Server.__init__(self, id, 'ieee802154.'\
                        + self.__class__.__name__, configuration)

    def start(self):
        self.working = True
        self.radio.start()
        self.trigger_event('server_start', communicator=self,
                           configuration=self.configuration)

    def treat_request(self, configuration: dict, device_id: str):
        client_address = configuration
        client_address['device_id'] = device_id
        radio_client = IEEE802154Client('802154_tmp_client', configuration)
        self.trigger_event('server_request',
                           address=client_address, client=radio_client)

    def configure(self):
        self.radio = MRF24J40MRadio()
        self.radio.set_request_func(self.treat_request)
        self.radio.configure(self.configuration)

    def stop(self):
        message = 'Succesfully close'
        try:
            self.radio.stop()
        except Exception as err:
            message = str(err)
        self.trigger_event('server_stop', communicator=self, message=message)
```

Figura 4.26: Implementación de "core::model::communicator::Server": IEEE80154Server

En la figura anterior, podemos observar una de las implementaciones de la clase abstracta "core :: model :: communicators :: Server". En esta ocasión, se trata de una implementación que hace uso de un driver para módulo transceptor del fabricante microchip (MRF24j40MC). Éste nos proporciona comunicación vía IEEE 802.15.4. Por ello, el módulo donde se encuentra esta implementación se llama "ieee802154.py", que a su vez se sitúa dentro de "transport". En este módulo también se encuentra la implementación del cliente.

Vamos a fijarnos en los detalles más importantes de esta implementación:

- La conexión de esta implementación con el driver se hace a través del objeto "radio", el cuál es instanciado en el método "configure()" (self.radio = MRF24J40MRadio()).
- El método "treat_request(self, configuration, device_id)" es invocado por el driver, cuando éste detecta una interrupción (irq) en recepción.
- En ese momento, contamos con un mensaje nuevo que puede ser o no consumido. Para proporcionar la posibilidad de consumirlo, se instancia un "Client" del tipo correspondiente (radio_client = IEEE802154Client('802154_tmp_client', configuration)). Éste es proporcionado al "Adaptor" y éste, dependiendo del device_id (network_id) puede decidir si quiere consumirlo o no.

De la implementación del driver hablaremos en el siguiente capítulo ("Instanciación y extensibilidad", 5), dado que es un ejemplo ideal para explicar como se puede extender el sistema añadiéndole compatibilidad con nuevos módulos transceptores.

Para terminar esta sección, vamos a poner un ejemplo de "Client". La peculiaridad de éste es que no tiene dependencia ni de elementos de "drivers" ni de elementos de "buses". Se trata de un cliente TCP/IP, por lo que la dependencia se establece con librerías del lenguaje de programación, en este caso Python. A su vez, éste depende de su intérprete, y el intérprete depende del sistema operativo. Gracias al intérprete, para nosotros esta dependencia será transparente. Está incluida en el módulo "tcpip.py", que a su vez está dentro del paquete "transport".

```
class TCPIPClient(Client):
    CONFIGURATION_PARAMS = ["host, port"]

    def __init__(self, id: str, configuration: dict):
        Client.__init__(self, id, 'tcpip.TCPIPClient', configuration)
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server_address = (configuration["host"], int(configuration["port"]))

    def connect(self):
        self.sock.connect(self.server_address)
        self.trigger_event("client_connect",
                           communicator=self, configuration=self.configuration)

    def configure(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server_address = (self.configuration["host"],
                               int(self.configuration["port"]))

    def send(self, message: str):
        try:
            self.sock.sendall(bytes(message, "utf-8"))
            self.trigger_event("client_send", communicator=self, frame=message)
        except socket.error as err:
            message = "Failed to send frame ( " +\
                      str(self.server_address[0]) + "):\nException: " + str(err)
            self.trigger_event("error", message=message)

    def receive(self) -> str:
        response = self.sock.recv(2**10)
        self.trigger_event("client_rcv", communicator=self, frame=response)

        return response.decode('utf-8')

    def close(self):
        message = "Close successfully"
        try:
            self.sock.close()
        except Exception as message:
            message = str(message)
        self.trigger_event("client_close", communicator=self, message=message)
```

Figura 4.27: Implementación de "core::model::communicator::Client": TCPIPClient

Como podemos observar en la figura, hacemos uso de la implementación de "socket" que proporciona el lenguaje.

Hablaremos de la implementación de esta capa con mucho más detalle en el siguiente capítulo ("Instanciación y extensibilidad", 5), dado que esta capa es clave en términos de extensibilidad del sistema.

4.3.3. Capa "adaptors"

En la capa "adaptors" encontraremos las implementaciones de los protocolos que nos permitirán poner a disposición lógica los recursos de los dispositivos de Tipo I. En esta capa se hará uso de las clases contenidas en "communicators::transport" para implementar los modelos de comunicación necesarios, pudiendo así obtener datos de sensores y realizar acciones sobre actuadores.

Toda instancia de "adaptor" dispondrá de una lista de instancias de "core::model:: nodes::Node" que serán gestionadas por él, de manera que los estados de los sensores/actuadores se obtendrán a través de la instancia de "Node". También la realización de acciones se realizará a través de la instancia de "Node".

Arquitectura lógica

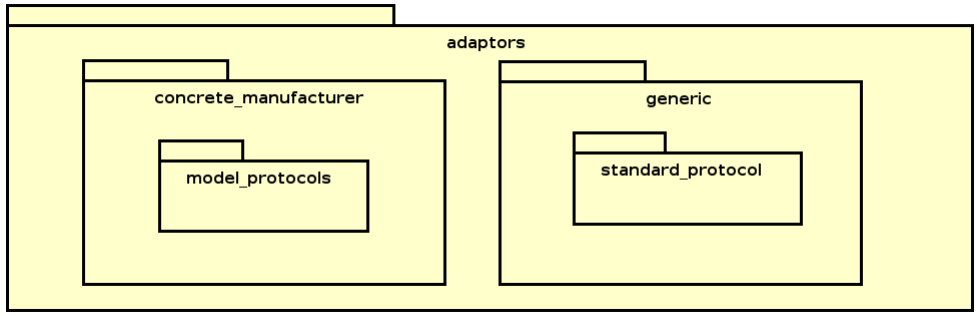

Figura 4.28: Diagrama de arquitectura lógica de "adaptors"

En la figura anterior podemos observar varios detalles importantes de esta capa:

- Al igual que la capa "communicators", esta capa es completamente dinámica y su contenido variará, dependiendo de la distribución de IoTGate que se esté utilizando.
- Se presenta un aspecto genérico; que se propone como una forma ordenada de organizar los elementos que se incluyan. En cualquier caso, no deja de ser una propuesta: como veremos más adelante, el desarrollador que esté trabajando con IoTGate puede organizarse los elementos de este capa como considere más cómodo.

- En la figura se aprecian dos grupos fundamentales de tipos de protocolos:
  - "concrete_manufacturer", se refiere a protocolos propietarios. Se refiere a protocolos que no se ciñen a ningún estándar, y que implementa un fabricante para un producto concreto.
  - "generic", se refiere a protocolos estándar que se presentan en varios dispositivos. En varias ocasiones, los protocolos propietarios serán especializaciones de estos protocolos.

    También sucede con cierta frecuencia que un dispositivo tome ciertos elementos de un protocolo estándar, siendo sólo útiles ciertos métodos; por lo que también puede existir la posibilidad de que protocolos incluidos dentro de "generic" se utilicen a modo de utilería -como "math"- para analizar una trama o convertir de un tipo de datos a otro, por ejemplo.

Descomposición modular

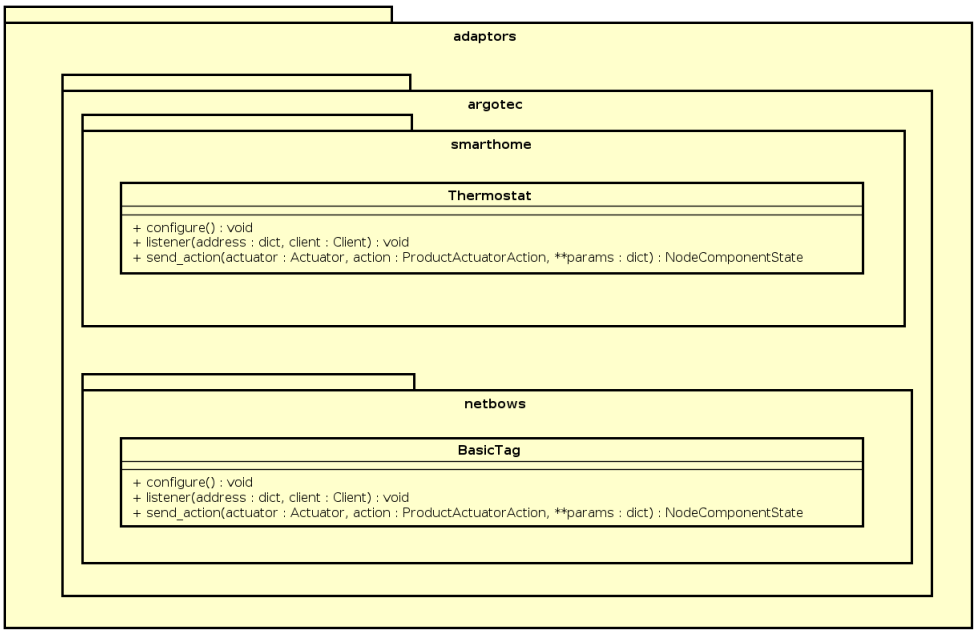

Figura 4.29: Descomposición modular de "adaptors"

En la figura anterior, está representada una distribución concreta de IoTGate. En esta ocasión, contamos con dos "adaptors", ambos del fabricante "argotec". Dentro del namespace "adaptors::argotec", ambos adaptors están agrupados usando el criterio "familia del producto". En el namespace "adaptors::argotec::smarthome", contamos con la implementación de

un protocolo que gestiona un modelo de producto concreto, un termostato. En el otro namespace "adaptors::argotec::netbows" contamos con la implementación de un modelo concreto del producto "netbows", que se llama BasicTag.

- Los dos cuentan con la misma estructura, ya que funcionan de forma muy similar. Ambos dispositivos emiten tramas mediante ieee802154, a un controller_id, pan_id y por un canal concreto. Cuando llega una trama, ésta es captada por una instancia de "Server", y éste hace una llamada a la función "callback":
  - En esta ocasión, la función "callback" de ambos "adaptors" resulta ser "listener".
  - Si recordamos al sección anterior, las instancias de "Server" cuentan con un método llamado "add_request_callback(callback: func)".
  - El método "configure" de las implementaciones de "Adaptor" es llamado desde el constructor. En éste, indicaremos que el método "listener" es la función que trata la petición añadiendo a la instancia de "Server" la callback mendiante la llamada al método "server.add_request_callback(listener)".
  - Podemos observar que el método "listener" recibe como argumentos "client_address" y "client". Mediante "client_address" y el atributo "network_id" de "core::model :: nodes::Node", el adaptor podrá conocer si la trama recibida corresponde a un dispositivo gestionado por él. Si es así, usará el objeto "client" para consumir el mensaje y ponerlo en la instancia de "Node" correspondiente. Si no, el mensaje quedará en un buffer a la espera de ser consumido por otro "adaptor".
- Para enviar acciones, se ejecutará desde una instancia de "core::model::nodes::Actuator" el método "do_action", que a su vez hará uso del método "send_action" que está implementado en el "adaptor" que lo esté gestionando. Mediante este método y la información del modelo sobre qué acciones hay disponibles, se formará la trama necesaria y se enviará haciendo uso del "network_id" del dispositivo y una instancia de "Client" creada para la ocasión.

Como conclusión, vamos a plantear una de las características clave de IoTGate: la implementación concreta de "Client" y "Server" a utilizar en el "Adaptor" puede variar. Hemos comentado que ambos dispositivos hacen uso de ieee 802.15.4. Si se fabricará una nueva versión que funcionara con el mismo "firmware" pero emitiera y recibiera las tramas mediante otra tecnología (como wifi o Bluetooth, por ejemplo), con un ligero cambio de configuración bastaría para hacerlo funcionar. Aquí reside el motivo por el cuál la capa "adaptors" y "communicators" se encuentran separadas, ofreciendo una mayor reutilización del código y una capacidad de comunicación multitecnología.

Dependencias

Las única dependencia que establece la capa "adaptors" con el resto del sistema es, como sucede con el resto de las capas, con la capa "core".

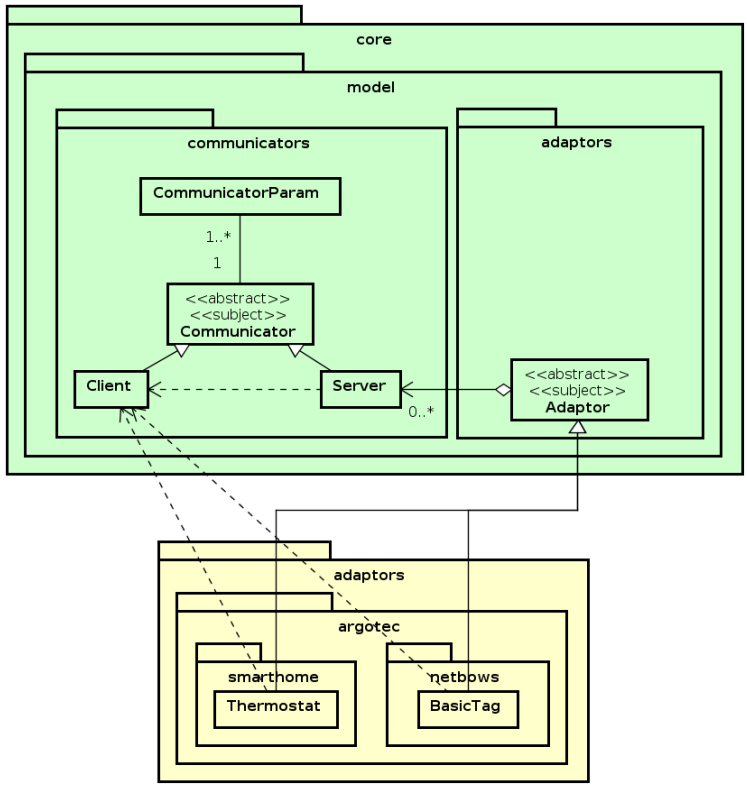

Figura 4.30: Diagrama de dependencia de “adaptors” con “core”

Vamos a explicar en qué consisten estas dependencias, y por qué existen. Fijándonos en la figura, determinamos:

- Las clases que componen la capa “adaptors” son implementaciones concreta de la clase abstracta “Adaptor”, que se encuentra en “core::model::adaptors::Adaptor”.
- Como podemos observar en la figura -fijarse en la capa “core”, en verde-, todo “Adaptor” -que lo necesite- puede contener instancias de “Communicator”, de tipo Server. Esto permite establecer qué método del “adaptor” concreto será utilizado como “callback” cuando llegue una “request”. Como comentábamos en la sección anterior, el añadido de “callback” se realiza en el método “configure()” del “adaptor”.
- Se establece una dependencia con “communicators” de tipo “Client”, por el hecho de tener la posibilidad de enviar acciones. Un “adaptor” que no necesite enviar acciones a su dispositivo -es decir, sólo utilice servidores- no tendrá esta dependencia. En cambio, los “adaptors” que si necesiten enviar acciones, necesitarán una instancia de “Client”. Normalmente, el uso de esta instancia debería suceder en el método “send_action” del “Adaptor” concreto.

Acerca de la implementación

Como comentábamos durante el desarrollo de la sección anterior, la estructura de esta capa depende de la distribución. Por tanto, durante esa sección vamos a describir la implementación de uno de los "adaptors" mostrado en la descomposición modular. Este "Adaptor" implementa el protocolo que nos permite trabajar como si de un "Node" se tratara con un producto fabricado en Argotec, "BasicTag".

Esta placa se comunica mediante IEEE 802.15.4, y reporta valores de temperatura y humedad cada cierto tiempo. Siempre inicia ella la conexión. Resulta ser interesante para el estudio porque se trata de un protocolo muy sencillo: cada cierto tiempo envía una trama con unos valores, los obtenemos y listo. Esta placa no dispone de actuadores y no se puede configurar de forma inalámbrica, por lo que no es necesario enviarla comandos (sólo haremos uso de "Client" para obtener mensajes).

```
class BasicTag(Adaptor):
    def __init__(self, id: str, communicators):
        Adaptor.__init__(self, id, 'argotec.netbows.BasicTag', communicators)

    def configure(self):
        comm = self.get_communicator_by_type('Server')
        comm.add_callback('server_request', self.listener)

    def listener(self, address, client):
        # Unknown devices, return
        if address['device_id'] not in self.managed_nodes:
            self.trigger_event('info', is_known=False, device='basic tag',
                               device_address=address)
            return
        network_id = str(address['device_id'])
        try:
            netbows = self.get_managed_node(network_id)
            message = client.receive()
            if message:
                rssi = message['rssi']
                message = message['message']
                try:
                    collected_data = {
                        'humidity': message.split(':')[0],
                        'temperature': message.split(':')[1],
                        'rssi': rssi
                    }
                    netbows.save_states(collected_data)
                except IndexError:
                    pass
        except KeyError:
            pass
    def send_action(self, actuator, action, **params):
        raise Exception('BasicTag can not do actions')
```

Figura 4.31: Implementación concreta de "Adaptor": "BasicTag"

Veamos los detalles más interesantes de este "Adaptor":

- Utilizamos el método "configure()" para indicar cuál de los métodos de este "Adaptor" es el que ha de ser ejecutado cuando un dispositivo inicia conexión contra nosotros.

- En el método "listener" aunamos las instancias de "core::model::nodes::Node" con el modelo de comunicación. A partir de aquí, el dispositivo real tiene su correspondiente virtual dentro del sistema.
- Hacemos uso de la integración del ORM con el modelo, dando la responsabilidad a la instancia de "Node" para guardar los valores en la base de datos.

Hablaremos de la implementación de esta capa con mucho más detalle en el siguiente capítulo ("Instanciación y extensibilidad", 5), dado que esta capa es clave en términos de extensibilidad del sistema.

4.3.4. Capa "deployment"

La capa "deployment" se encarga de hacer sencillo el proceso de integración de IoTGate con otros desarrollos de más alto nivel. En esta capa se hace uso de reflexión e introspección, por lo que la mayor parte de los cambios que se realicen en la capa "core" quedarán expuestos inmediatamente en las APIs que se construyan a partir de ella.

Por otra parte, otra idea interesante que presenta su concepción es la facilidad con la que se podrían exponer otros elementos del sistema.

Arquitectura lógica

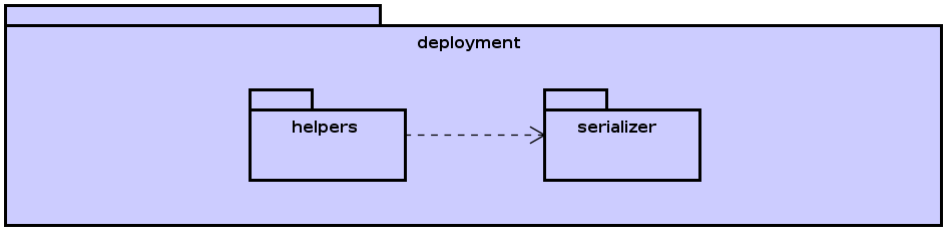

Figura 4.32: Diagrama de arquitectura lógica de "deployment"

Como podemos observar en la figura anterior, esta capa está compuesta por:

1. "helpers": contiene implementaciones que dan soporte a los modelos de comunicación básicos que se necesitan en software de más alto nivel. Estos son:
   a) "Request-Response": mediante este modelo, podremos enviar mensajes a los objetos expuestos del sistema (los "managers" de "core::model::managers"). Gracias a ello, podremos hacer operaciones como consultar qué productos están registrados, que nodos hay en un grupo, que "servers" están escuchando, etc.

b) "Publish-subscribe": mediante este modelo, daremos soporte a la observación de cambios en la red: cuando un dispositivo cambie de estado, una trama llegará a un adaptor, ese adaptor la tratará y cambiará un nodo en consecuencia. Este cambio será notificado a través del sistema descrito en la sección 4.3.1, y en esta capa "deployment", se dispone de los recursos necesarios para consumir los eventos que se lancen en el core.

2. "serializer" contiene las clases necesarias para serializar todos los objetos de dominio que haya que comunicar a un formato estándar, facilitando la transmisión por red de los objetos y la interpretación de estos mensajes en lenguajes de programación diferentes a Python3.

Descomposición modular

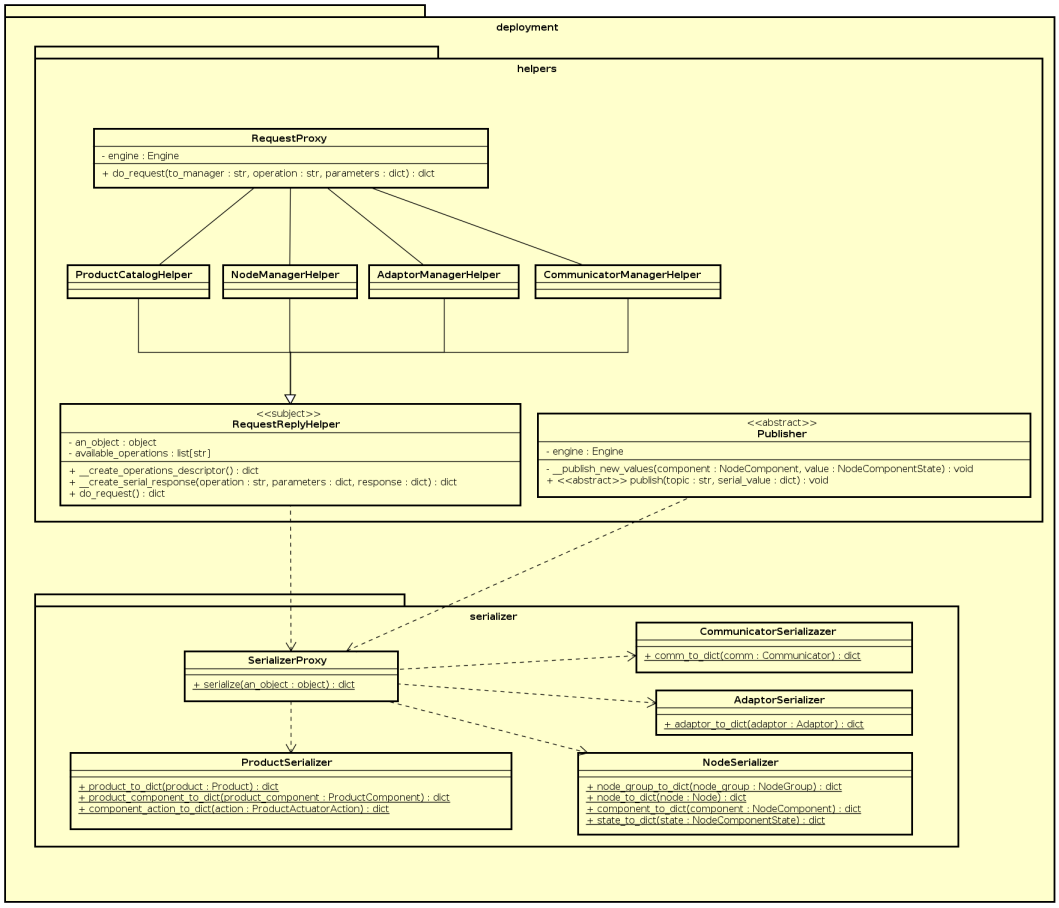

Figura 4.33: Descomposición modular de "deployment"

Vamos a comenzar esta sección hablando de "deployment::serializer":

- Como podemos observar en la figura, contamos con cuatro serializadores, uno por cada bloque conceptual básico del sistema (productos, dispositivos, adaptors y communicators). Todos ellos son una clase de métodos estáticos, que analizan el objeto dado y devuelven un diccionario de Python, cuyas claves y valores son de un tipo básico (int, str, float, etc.)
- Contamos con la clase "SerializerProxy". Es en este punto donde utilizamos introspección. Llamamos a al método "SerializerProxy.serialize(objeto)", éste analiza el tipo del objeto y determina que serializador ha de utilizar para convertir el dato. El algoritmo de "serialize" es recursivo, y por tanto, siempre devuelve un "dict" cuyos pares clave-valor son de un tipo básico.
- Una vez está el objeto representado mediante un "dict", se hace uso de la utilería que proporciona el lenguaje (json) para convertir el dato definitivamente.

El uso que se hace de serializer es el mismo por parte de todos los elementos de "deployment::helpers". Ahora, vamos a hablar de ellos. Principalmente, tenemos que dar soporte a dos modelos de comunicación, como comentábamos en la introducción a esta sección. Por ello, contamos con dos clases fundamentales aquí:

- "Publisher": es una clase abstracta que se proporciona para ser implementada en desarrollos que quedan fuera del alcance de IoTGate. Da soporte al modelo de publicación-subscripción.
- "RequestProxy": es una clase que se proporciona para ser utilizada como único punto de entrada al sistema. A través de él se exponen las operaciones de los "managers" de "core::model::managers" (4.3.1).
  - La clase "RequestReplyHelper" hace uso de reflexión para analizar el objeto (en este caso, una instancia de manager). Mediante la reflexión, analiza que operaciones públicas tiene el objeto y que argumentos tienen estas operaciones. En base a este primer análisis, proporciona un descriptor de operaciones-argumentos, a través de la invocación al método "get_operations()". Cuando le llega una petición (es decir, se invoca a do_request(operation:str, arguments: dict), hace uso de los mecanismos de reflexión que Python proporciona e invoca al método correspondiente del "manager" que toque)
  - Las clases que especializan a "RequestReplyHelper" permiten hacer filtro: su principal misión es quitar ciertas operaciones que no han de ser expuestas (por falta de necesidad, o en base a criterios de configuración). Modifican el atributo "available_operations", cuyo valor es una lista de strings. Cada string resulta ser el nombre de una operación expuesta. Si la operación no está en ese lista, no se puede realizar desde fuera.
  - La clase "RequestProxy" se encarga de recibir la petición, pasar los argumentos al "Helper" correspondiente y de responder al programa cliente con el objeto ya serializado.

En la sección de implementación detallaremos las características de este diseño. En el siguiente capítulo (Instanciación y extensibilidad, 5), explicaremos como utilizarlo y mostraremos diferentes ejemplos.

Dependencias

Como pasa con las demás capas, "deployment" depende de "core". En el siguiente diagrama, podemos comprobar que la única dependencia es "core::main::Engine":

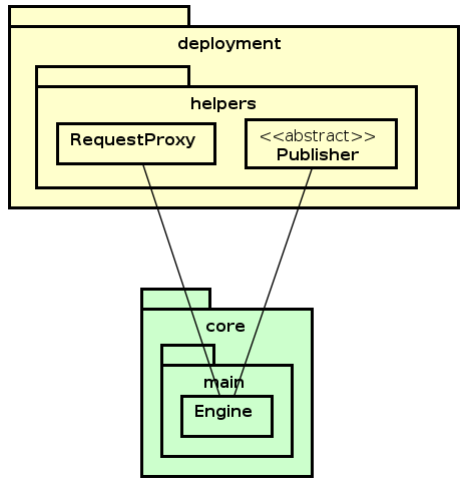

Figura 4.34: Diagrama de dependencia de "deployment" con "core"

Sólo hay una instancia de "Engine" por instancia del sistema. El "Engine" es compartido por ambas clases. Al crear objetos de la clases "Publisher" y "RequestProxy", la instancia de "Engine" es inyectada. Como comentábamos en la sección anterior, durante la sección "Acerca de la implementación" veremos los detalles clave que ayudarán a entender este diseño.

Acerca de la implementación

Para comenzar esta sección, vamos a estudiar el contenido del paquete "deployment::helpers". Nuestra primera parada es la implementación de Publisher:

```python
class Publisher:
  def __init__(self, engine):
      engine.get_node_manager().add_callback_to_all_nodes(self._publish_new_values)

  def _publish_new_values(self, component, data):
      node_group = component.get_node().get_node_group().get_id()
      node_alias = component.get_node().get_alias()
      node_component = component.get_alias()
      topic = "new_value" + "." + node_group\
              + "." + node_alias + "." + node_component
      value = NodeSerializer.component_value_to_dict(data)
      self.publish(topic, value)

  @abstractmethod
  def publish(self, topic, serial_value):
      pass
```

Figura 4.35: Implementación de “deployment::helpers::Publisher”

En la implementación de “Publisher” podemos observar varias cosas:

- Hace uso de una instancia de “core::main::Engine” para funcionar. A partir de esta, obtiene el “manager” de nodos y se suscribe al tipo de evento “new_value”, por lo que el método “publish(topic: str, serial_value: dict)” será ejecutado cada vez que un nodo cambie de estado.
- El método “_publish_new_values(component: NodeComponent, data: NodeComponentState)” se encarga de lidiar con las instancias de clases del modelo de dominio, proporcionado al método “publish”:
  - topic: es una ruta única que permite contextualizar el dato. Tiene el formato <tipo de evento>.<nombre del grupo de nodos>.<alias del nodo>.<alias del componente>.
  - serial_value es un diccionario cuyas claves son la fecha de recepción, la fecha de medida y el valor observado.
- El método “publish” es abstracto, por lo que el uso de esta clase se hará a través de la especialización. En el siguiente capítulo (“Instanciación y extensibilidad”, 5) veremos como se utiliza con un ejemplo.

Nuestra siguiente parada es la implementación de “RequestProxy”:

```python
class RequestProxy(Subject):
    event_types = ["on_message", "on_error"]

    def __init__(self, engine):
        Subject.__init__(self)
        self.request_helpers = engine.get_request_helpers()

    def get_available_helpers(self):
        return self.request_helpers.keys()

    def do_request(self, to_manager: str, operation: str, parameters: dict) -> dict:
        try:
            reply = self.request_helpers[to_manager].do_request(operation, parameters)
            self.trigger_event("on_message", message="Received request to "
                               + to_manager + ", operation <" + operation + ">,"
                                "parameters: " + json.dumps(parameters, indent=3))
            return reply
        except Exception as err:
            message = "Error: something terrible happens -> " + str(err)
            self.trigger_event("on_error", message="error:::" + message)
            return {
                    "from": self.__class__.__name__,
                    "return": "Error: something terrible happens: " + str(err),
                    "operation": "None"
            }
```

Figura 4.36: Implementación de "deployment::helpers::RequestProxy"

En esta implementación, observamos lo siguiente:

- Contamos con 4 instancias, una por instancia de manager, de clases de tipo "RequestReplyHelper".
- El cliente tiene que establecer a que manager quiere hacer una petición (los disponibles se pueden consultar a través del método get_available_helpers()), pasar el nombre de la operación (en string) y un diccionario con los pares argumento-valor.
- La respuesta que se obtenga (variable reply) es una variable serializada, que tiene el siguiente formato:

```
{
  "from": str,
  "return": [ dict | str | int | float ]
  "operation": str
}
```

Figura 4.37: Formato de respuesta de "RequestReplyHelper"

Observamos en la figura anterior:

- El tipo de "reply" es siempre diccionario.
- Contiene las claves from, return y operation:
  - from indica de que clase (manager, salvo error; helper) proviene la respuesta.
  - return indica cual es la respuesta. Esta siempre será un tipo básico y serializable. Normalmente es otro diccionario.

- operation indica cuál fue la operación que se ordenaba realizar en la petición.

- En caso de excepción, la última palabra la tiene el proxy: no debemos lanzar excepciones a tratar más allá de este punto. Por ello, devolvemos siempre la respuesta en el mismo formato. En caso de error, se adjunta un mensaje y se indica que el valor de la clave "operation" es "None".

La siguiente implementación importante es la de la clase "RequestReplyHelper":

```
class RequestReplyHelper:
    def __init__(self, an_object):
        self.an_object = an_object
        self.available_operations = [method for method in dir(self.an_object)
                                     if isinstance(getattr(self.an_object, method),
                                        collections.Callable) and method[0] != "_"]
        self.operations_desc = self._create_operations_descriptor()
```

Figura 4.38: Constructor de la clase "deployment::helpers::RequestReplyHelper"

Como podemos observar en la figura, esta clase recibe una instancia de clase cualquiera. En esta caso, sólo la utilizamos con las cuatro instancias de "core::model::managers". Al crearse "RequestReplyHelper", el objeto es analizado, obteniendo una lista con los métodos de éste. Después se llama al método "self._create_operations_descriptor()":

```
def _create_operations_descriptor(self):
    operations = {}
    for operation in self.available_operations:
        param_list = list(getattr(self.an_object, operation)\
                                  .__code__.co_varnames)
        param_list = param_list[:getattr(self.an_object, operation)\
                                  .__code__.co_argcount]
        try:
            param_list.remove('self')
        except ValueError:
            pass
        operations[operation] = param_list
    return operations
```

Figura 4.39: Método "_create_operations_descriptor()", de la clase "RequestReplyHelper"

En la figura anterior, podemos observar que mediante este método creamos un descriptor de operaciones disponibles. Este descriptor es un diccionario cuyas claves son las operaciones, y cuyos valores son listas con los argumentos de la operación. Veamos ahora el último método clave de esta clase, "do_request(operation:str, parameters:dict)"

```
def do_request(self, operation, parameters):
    if operation in self.available_operations:
        try:
            response = getattr(self.an_object, operation)(**parameters)
            reply = self._create_serial_response(operation, parameters, response)
        except KeyError as err:
            response = "Exception (Does not exists): " + str(err)
            reply = self._create_serial_response(operation,  parameters, response)
        except TypeError as err:
            reply = self._create_serial_response(operation,
                                                 parameters,
                                                 'Not recognized operation: invalid\
                                                 parameters <' + str(err) + '>')
    elif operation == "get_operations":
        reply = self._create_serial_response(operation, {},
                self.operations_desc)
    else:
        reply = self._create_serial_response(operation, {},
                    "Requested operation\
                    doesn't exists. Privileges?")
    return reply
```

Figura 4.40: Método "do_request(operation, parameters)", de la clase "RequestReplyHelper"

En la figura anterior podemos observar que el método do_request recibe el nombre de la operación en string y un diccionario cuyas claves son los argumentos del método, y cuyos valores son los valores de ellos.

Hay otra serie de detalles interesantes:

- Esta concepción de "deployment" es la que lleva a que los argumentos de las operaciones de los "managers" sean todos tipos base, dado que los mensajes que lleguen del exterior llegarán serializados y habrá que hacer una conversión. Cuando más sencilla sea, mejor. Por ello, repartimos un poco la complejidad por diferentes partes del sistema.
- Introducimos aquí un "método virtual" (get_operations), que nos permitirá aplicar reflexión desde software exterior al sistema; dado que se obtendrá una lista de las posibles operaciones que ofrece un determinado objeto.
- No se muestra la implementación del método "_create_serial_response" porque pensamos que no resulta de interés. Básicamente, devuelve un diccionario con el formato definido en la figura 4.37.

Para terminar esta sección, vamos a mostrar fragmentos del serializador. Como se comentaba anteriormente, el serializador se encarga de pasar instancias de objetos de nuestro modelo de dominio a un diccionario cuyas claves-valores son tipos base. Esta operación nos permitirá después utilizar los serializadores que proporciona el propio lenguaje de programación (json).

```
class SerializerProxy(object):
   class SerializerProxy(object):
    @staticmethod
    def serialize(an_object):
        if isinstance(an_object, str):
            return an_object
        if isinstance(an_object, int):
            return an_object
        if isinstance(an_object, float):
            return an_object
        if isinstance(an_object, bytes):
            return int(an_object)
        if not an_object:
            return None
        if isinstance(an_object, list):
            return [SerializerProxy.serialize(item) for item in an_object]
        if isinstance(an_object, dict):
            serial_dict = {}
            for key in an_object:
                serial_dict[key] = SerializerProxy.serialize(an_object[key])
            return serial_dict
        if isinstance(an_object, NodeGroup):
            return NodeSerializer.node_group_to_dict(an_object)
        if isinstance(an_object, Product):
            return ProductSerializer.product_to_dict(an_object)
        if isinstance(an_object, Adaptor):
            return AdaptorSerializer.adaptor_to_dict(an_object)
        if isinstance(an_object, Communicator):
            return CommunicatorSerializer.communicator_to_dict(an_object)
```

Figura 4.41: Implementación de la clase “deployment::serializer::SerializerProxy”

Como podemos observar en la figura anterior, “SerializerProxy” es una función que aplica instrospección y recursividad, consiguiendo así representaciones de nuestras instancias del modelo de negocio en objetos base serializables.

En el siguiente capítulo (Instanciación y extensibilidad, 5), explicaremos como utilizar todas las herramientas presentadas mediante diferentes ejemplos.

Capítulo 5

Instanciación y extensibilidad

5.1. Introducción

A lo largo de este capítulo, entenderemos como se utiliza IoTGate, viendo primero que funcionalidades proporciona al ser instanciado, y que posibilidades de integración con software proporciona.

Continuaremos hablando de cómo ampliar sus capacidades, dónde están sus "hooks" y qué herramientas pone a disposición del desarrollador para llevar a cabo estas tareas.

Cada sección de este capítulo irá acompañada de ejemplos sencillos que permitirán al lector entender mejor el funcionamiento.

5.2. Instanciación

En IotGate, contamos con tres niveles de instanciación:

- **Nivel I**: en este nivel, hacemos uso únicamente de la capa "Communicators".
- **Nivel II**: en este nivel, hacemos uso de "core::main::Engine" (4.3.1), por lo que disponemos de persistencia, productos y nodos organizados.
- **Nivel III**: en este nivel, hacemos uso de la capa "deployment", la cual proporciona las herramientas necesarias para desarrollar APIs a partir de las funcionalidades del sistema.

A lo largo de esta sección explicaremos como se hace uso de estos niveles. Otro detalle importante a tener en cuenta es que los niveles superiores engloban a los inferiores, es decir,

cuando hacemos uso del Nivel II, por debajo estamos utilizando el Nivel I, y cuando hacemos uso del Nivel III, estamos haciendo uso del Nivel II y el Nivel I. En cualquier caso, no son excluyentes: por ejemplo, se puede trabajar mezclando el Nivel II y el III.

5.2.1. Nivel I: sólo communicators

Cuando hacemos uso de este nivel, estamos haciendo principalmente uso de la capa communicators, y por tanto, de las implementaciones de "Client" y "Server" multitecnología. Esto nos vendrá bien en los siguientes casos:

- Nos encontramos desarrollando elementos de la capa "communicator" (un driver para un transceptor, una implementación de "Client", una de "Server"...). Para poder probarlo, puede ser tedioso -e incómodo por tanto- tener que instanciar todo el sistema, configurar la base de datos, registrar productos, nodos, etc.
- Necesitamos hacer un desarrollo muy sencillo en el que no necesitamos las capas superiores.

Para comenzar a trabajar con este Nivel I, es necesario hablar de la clase "core::model:: managers:: CommunicatorManager". Ésta proporciona tres métodos **estáticos** que ayudan a instanciar "communicators":

1. **get_commmunicator_types() ->list**: devuelve una lista de strings. Cada string es el nombre de un "communicator" instalado y listo para usar en IoTGate.
2. **get_commmunicator_type_config_params(comm_type: str) ->list**: devuelve una lista de strings. Cada string es uno de los parámetros a configurar para establecer la conexión. Como argumento, recibe uno de los valores de la lista del método explicado en el punto anterior.
3. **create_commmunicator(comm_type:str, configuration:dict) ->Communicator**: este método es una factoría de "communicators". A partir del tipo del "communicator" (obtenido mediante el primer método) y con la configuración (establecido por el usuario, los parámetros de configuración se obtienen con el segundo método), crea y devuelve un "Communicator" listo para trabajar.

Es importante tener en cuenta que los métodos son estáticos: no necesitamos instanciar "CommunicatorManager". Como veremos más adelante, la instanciación de los "managers" se realiza a través de la instanciación de la clase "Engine", ya que supone uso de la base de datos y otros elementos del sistema. Como en este Nivel I no nos interesan los demás elementos del sistema, no instanciamos "CommunicatorManager".

Ahora que ya tenemos creado el "communicator", el siguiente paso es entender cómo funcionan. Como comentábamos en capítulos anteriores, contamos con dos tipos de "communicators":

1. Server: este tipo de "communicator" lo utilizaremos cuando los dispositivos con los que vamos a trabajar son los que inician la conexión, y por tanto, debemos esperar a que "se pongan en contacto". Para trabajar con instancias de "Server" haremos uso principalmente de los siguientes métodos:

   a) **add_callback(event_type:str, callback:func)**: nos permite añadir funciones a ejecutar cuando sucede un tipo de evento concreto.

   b) **start()**: Provoca que el server comience a escuchar. Cuando llegue un mensaje, lanzará un evento de tipo "server_request", provocando la ejecución de las funciones añadidas con el método del punto anterior.

   c) **stop()**: Provoca que el server deje de escuchar.

2. Client: usaremos este tipo de communicator cuando seamos nosotros los que necesitamos iniciar la conexión contra el dispositivo. Client será utilizado también por los "Server": cuando llegue un mensaje, los parámetros que se pasan a la función callback son "address" y "client". Veremos esto con un ejemplo. Los métodos de "client" que usaremos principalmente son:

   a) **send(message:str)**: nos permite enviar un mensaje al dispositivo.

   b) **receive() ->str**: nos permite recibir mensajes de dispositivos.

A continuación, veremos dos ejemplos sencillos de uso. En el primero, haremos uso de los "communicators" de manera interactiva, mediante el intérprete de Python. En el segundo, haremos un pequeño programa que nos permitirá acabar de entender cómo funciona esta capa.

Ejemplo I: uso interactivo, en Nivel I

Para poder seguir los pasos de este ejemplo, se ha de tener instalado "IoTGate" tal y cómo se explica en el apéndice "Manual de Instalación" (A). Los pasos los estamos llevando a cabo en una Raspberry Pi 2, con una imagen del sistema operativo "Raspbian", en su versión "2016-02-09-raspbian-jessie-lite.img"

Nuestra particular Rasberry cuenta además con hardware adaptado a ella en Argotec, como se muestra en la siguiente figura:

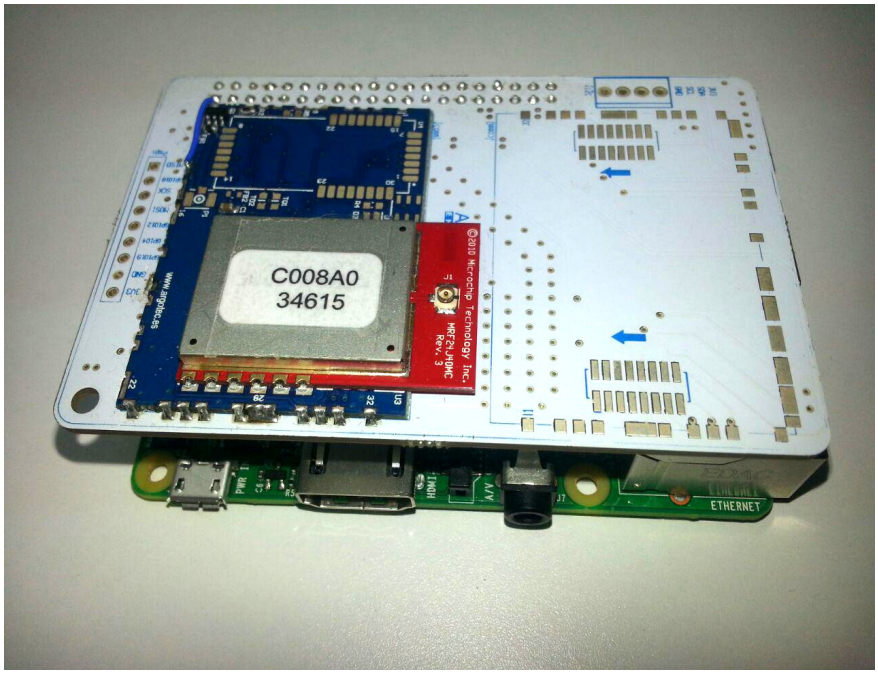

Figura 5.1: Raspberry Pi 2 con placa de expansión y el módulo MRF24J40MC

Nuestro objetivo durante este ejemplo va a ser conectarnos a un termostato fabricado en Argotec, que utiliza para comunicarse 802.15.4. Vamos a seguir los siguientes pasos:

1. Abrimos un terminal de Python 3 e importamos el manager de communicators:

```
diego@iotgate-tfg:~$ python3
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from iotgate.core.model.managers import CommunicatorManager as comm_man
```

2. Ahora, vamos a consultar qué "communicators" tenemos instalados en nuestra distribución de IoTGate:

```
>>> comm_man.get_communicator_types()
['ieee802154.IEEE802154Client', 'ieee802154.IEEE802154Server', 'tcp.TCPClient', 'tcp.TCPServer']
```

Como podemos observar, en este instante contamos con cuatro tipos de comunicadores:

- **ieee802154.IEEE802154Client**: es la implementación de "Client" para comunicación con dispositivos IEEE 802.15.4.

- **ieee802154.IEEE802154Server**: es la implementación de "Server" para comunicación con dispositivos IEEE 802.15.4.
- **tcp.TCPClient**: es la implementación de "Client" para comunicación con dispositivos TCP/IP.
- **tcp.TCPServer**: es la implementación de "Server" para comunicación con dispositivos TCP/IP.

3. Ahora que ya sabemos los tipos de "communicator" disponibles, tenemos que decidir qué configuración vamos a establecer. Para conocer los valores de configuración que tenemos que establecer para el "Communicator" de tipo "ieee802154.IEEE802154Server":

```
>>> comm_man.get_communicator_type_config_params('ieee802154.IEEE802154Server')
['controller_id', 'pan_id', 'channel']
```

4. En este punto, ya tenemos toda la información que necesitamos para crear este "communicator". Vamos allá:

```
>>> server = comm_man.create_communicator(comm_type='ieee802154.IEEE802154Server', configuration={
... "controller_id": "2000",
... "pan_id": "bcda",
... "channel":"18"})
```

5. Durante la sección anterior, explicábamos que tanto "Server" como "Client" hacen uso del sistema de eventos que proporciona IoTGate. Como comentábamos en el capítulo anterior (ver figura 4.20), toda clase que lance eventos en el sistema cuenta con una documentación asociada:

```
class Server(Communicator):
    """
        Events
            server_request: called after receive any request.
                param - address: dict => Device address information.
                param - client: Client => Client instance to communicate with device.
            server_start: called just before server start to listen.
                param - configuration: str => Server's configuration.
            server_stop: called just before stop the server.
                param - message: str => Everything stop well?.
            error: called for any error occurred.
                param - message: str => Error message.
    """
    event_types = ["server_request", "server_start", "server_stop", "error"]
```

Figura 5.2: Documentación de los eventos de "Server"

Como podemos observar en la figura, nosotros necesitamos crear una función que reciba address y client, para poder consumir los mensajes que llegan de nuestro dispositivo:

```
>>> def foo(address, client):
...   print(address, client)
...   print(client.receive())
...
```

Esta función básica nos imprimirá por pantalla la información que llega cada vez que el dispositivo nos envíe un mensaje.

6. El siguiente paso es asociar nuestra función a nuestro "Server". Eso lo haremos así:

```
>>> server.add_callback(event_type='server_request', callback=foo)
```

Ahora, cada vez que llegue un mensaje, se imprimirá por pantalla algo de información.

7. Sólo nos queda comenzar a escuchar:

```
>>> server.start()
```

Y en el momento en el que el dispositivo nos ponga un mensaje, veremos:

```
{'pan_id': 'bcda', 'controller_id': '2000', 'channel': '18', 'device_id': 1}
<iotgate.communicators.transport.ieee802154.IEEE802154Client object at 0x75ef4770>
{'message': '12:35.2,27.1,0-0,0.0:10:0', 'rssi': -20}
```

Para dejar de escuchar, ejecutaremos:

```
>>> server.stop()
```

Ejemplo II: un pequeño "puente de red"

Este ejemplo lo vamos a desarrollar sobre la misma plataforma que el anterior (Raspberri Pi 2 con hardware adicional).

En esta ocasión, vamos a montar un "pequeño puente de red". Este puente consistirá en reenviar vía TCP/IP los mensajes que nos envía el termostato (que se comunica vía IEEE 802.15.4) del ejemplo anterior. Para ello, necesitaremos hacer uso de varios communicators:

```
import time
from iotgate.core.model.managers import CommunicatorManager as comm_man

server_config = {
                 "host":"localhost",
                 "port": "8080"
                }
tcp_server = comm_man.create_communicator('tcp.TCPServer', server_config)

def server_callback(address, client):
    print("[-] Message from ip thermostat:",
           address, " message:", client.receive())

tcp_server.add_callback("server_request", server_callback)

tcp_server.start()
try:
    time.sleep(100000)
except KeyboardInterrupt:
    tcp_server.stop()
```

Figura 5.3: Ejemplo II: Implementación simple de servidor TCP/IP

Como podemos observar en la figura, es sencillo poner a escuchar a un socket TCP/IP con IoTGate. Cuando nos entre una petición, el sistema se encargará de llamar a la función definida por el usuario "server_callback(address, client)". En este ejemplo, que pretende ser sencillo, sólo imprimiremos por pantalla el mensaje que nos llegue. Dejemos ahora ese código ejecutándose, a la espera de recibir mensajes.

Veamos la siguiente pieza de código:

```
import time
from iotgate.core.model.managers import CommunicatorManager as comm_man

tcp_client_config = {
                     "host":"localhost",
                     "port": "8080"
                    }
server_config = {
                 "controller_id":"2000",
                 "pan_id": "bcda",
                 "channel": "18"
                }
tcp_client = comm_man.create_communicator('tcp.TCPClient',tcp_client_config)
server = comm_man.create_communicator('ieee802154.IEEE802154Server',server_config)

def server_callback (address, client):
    print("[+] Message from: ", address,)
    tcp_client.connect()
    tcp_client.send(client.receive())
    tcp_client.close()
    print("[*] Sent to ", tcp_client.get_configuration())

server.add_callback("server_request", server_callback)
server.start()

try:
    time.sleep(10000)
except KeyboardInterrupt:
    server.stop()
```

Figura 5.4: Ejemplo II: Implementación del "puente" de IEEE 802.15.4 a TCP/IP

En este último código, sucede lo mismo que durante el Ejemplo I, con una salvedad: utilizamos otro "communicator", en este caso de tipo "tcp.TCPClient", mediante el cuál reenviamos los mensajes que nos llegan por el "Communicator" de IEEE 802.15.4 al servidor TCP/IP mostrado en la figura anterior.

5.2.2. Nivel II: integración a través del "Engine"

Como vimos en capítulos anteriores (4.3.1), la clase "core::main::Engine" supone ser un punto de instanciación del sistema fundamental.

Cuando estemos trabajando en Nivel II, instanciaremos un "Engine" y haremos uso de instancias de las clases de "core::model::managers" para manipular el modelo de dominio. Esto nos permite un control total de todos los elementos que se ofrecen en el sistema para integración con otras aplicaciones.

```
1 from iotgate.core.main import Engine
2
3 engine = Engine(configuration_path='/etc/iotgate/tfg-config.conf')
4
5 pm = engine.product_manager
6 cm = engine.communicator_manager
7 am = engine.adaptor_manager
8 nm = engine.node_manager
```

Figura 5.5: Ejemplo de uso: creación del "Engine" y obtención de los managers

Por ello, durante el desarrollo de esta sección vamos a hablar de cómo se utilizan los managers, y vamos a terminar con un ejemplo en el que se implementa una pequeña API REST haciendo uso de las bondades del "Engine". Por otra parte, existe toda la documentación dentro del código:

```
1 diego@debianito: $ python3
2 Python 3.4.2 (default, Oct  8 2014, 10:45:20)
3 [GCC 4.9.1] on linux
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> from iotgate.core.model.managers import NodeManager
6 >>> help(NodeManager)
```

Figura 5.6: Ejemplo de uso: como consultar la documentación de "NodeManager"

El comando "help(objeto)" permite obtener también la documentación introduciéndole como parámetros instancias de la clase.

Este desarrollo es también compatible con la librería "pydoc", que nos permitirá obtener una documentación autogenerada (previa definición bajo formato concreto por mi parte en el código, nada es mágico).

Database Manager

"DatabaseManager" proporciona control sobre la base de datos al resto de "managers". Como comentábamos en la sección 4.3.1, para hacer uso del ORM necesitaremos principalmente los objetos _Engine (de SQLAlchemy, no de IoTGate), que gestiona la conexión con el SGBD y el objeto "_Session", que nos proporciona uso del ORM (add, delete, update, commit, etc.)

Product Manager

"ProductManager" proporciona control sobre el catálogo de productos. Es uno de los managers más simples, implementa las siguientes operaciones:

1. **register_product(product_descriptor: dict)**: nos permite registrar un producto a partir de un descriptor. Un descriptor tiene el siguiente aspecto:

```
{
   "id":"802154Thermostat",
   "manufacturer":"Argotec",
   "model":"netbows",
   "components":[
      {
         "type":"actuator",
         "board_id":"comfort",
         "unit":"celsius",
         "value_type":"float",
         "actions":[
            {
               "name":"change_comfort",
               "command":"at!ct%(temperature)s\r",
               "effect":"Changes comfort temperature"
            }
         ]
      },
      {
         "type":"sensor",
         "board_id":"temperature",
         "unit":"celsius",
         "value_type":"float"
      },
      {
         "type":"sensor",
         "board_id":"humidity",
         "unit":"percentage",
         "value_type":"float"
      },
      {
         "type":"sensor",
         "board_id":"rssi",
         "unit":"dBm",
         "value_type":"int"
      }
   ]
}
```

Figura 5.7: Ejemplo de descriptor de producto: Termostato

En la figura anterior, podemos observar lo siguiente:

- Nuestro descriptor cuenta con cuatro claves principales: id, manufacturer, model y components.
- components es una lista que describe los componentes que tiene el dispositivo que representa. Contamos con dos tipos de componentes (lo indicamos con el valor de la clave type), como hemos visto a lo largo del trabajo:
  - sensor: contiene board_id, unit, y value_type. La clave board_id será importante cuando nos encontremos trabajando en un "Adaptor": mediante este valor identificaremos el componente.
  - actuator: es exactamente igual que sensor, pero con una clave más: actions. "actions" es una lista de acciones. Las acciones se describen mediante las claves name, command y effect. name es el nombre de la acción mediante el cuál la identificaremos, command contiene el mensaje parametrizado que hay que enviar al dispositivo. El formato de command se establece teniendo en cuenta cómo funciona el formateador de strings de Python ([75]). effect describe que hace esta acción; se puede dejar en blanco si no es necesario describirla.
- Otra cosa interesante de este descriptor es uno de los sensores: "rssi". Realmente, la fuerza de señal no es un componente del dispositivo, tampoco es un sensor. Entendiendo "sensor" y "actuator" como entrada y salida, podemos jugar con ello. Por ejemplo, en algunos dispositivos podemos utilizar un componente sensor para describir un estado (batería, mensajes de diagnóstico, pulsación de botones en el dispositivo, etc.); y actuadores para enviar comandos que tengan que ver con configuraciones del dispositivo, por ejemplo.

2. **unregister_product(product_id: str)**: nos permite borrar un producto del catálogo a partir del identificador de producto.
3. **get_product(product_id: str)**: nos permite recuperar un producto del catálogo a partir de su identificador.
4. **get_products_ids()**: nos devuelve una lista con todos los identificadores de productos disponibles.

Aparte de nosotros, hará uso de este manager el manager de nodos: en el momento de registro de nodo, el nodo se asocia a un producto concreto. Mediante esta asociación, conoceremos que serie de datos nos va a proporcionar el dispositivo concreto, y qué acciones tiene disponibles.

Communicator Manager

El communicator manager nos permitirá gestionar los communicators; especialmente aquellos que sean de tipo server. Cuenta con los siguientes métodos:

1. **get_communicator_types()**: este método es estático, y devuelve una lista de strings con los nombres de los tipos de communicators que tenemos instalados en la distribución que estemos utilizando de IoTGate.

2. **get_communicator_type_config_params(comm_type: str)**: este método es estático y devuelve una lista de las claves que tendrá que contener el diccionario "configuration" en los métodos de creación que vienen a continuación.

3. **create_communicator(comm_type: str, configuration: dict)**: este método es estático, y nos devuelve un communicator listo para ser utilizado.

4. **new_communicator(comm_id: str, comm_type: str, configuration: dict)**: este método hace lo mismo que el anterior, con la salvedad de que recibe un nombre con el cuál se guardará en la base de datos, quedando así persistente para futuros usos. Cuando instanciemos en "Engine", y por tanto los managers, los communicators creados a través de este método estarán disponibles para su uso.

5. **start_communicator_by_id(comm_id: str)**: a partir del identificador de communicator, permite que éste comience a escuchar. Los communicators que se levanten serán de tipo "Server".

6. **stop_communicator_by_id(comm_id: str)**: permite parar un communicator de tipo "Server" a partir de su identificador.

7. **restart_communicator_by_id(communicator_id: str)**: permite reiniciar un communicator de tipo "Server" a partir de su identificador.

8. **get_communicators_status()**: devuelve un diccionario cuyas claves son identificadores de communicators, y cuyos valores son diccionarios que describen el communicator (nombre, tipo) y su estado (parado, escuchando, conectado, etc.)

Aparte de nosotros, de este manager hará uso también el manager de Adaptors: aquellos Adaptors que necesiten utilizar para funcionar un communicator de tipo "Server", consultarán a este manager por el communicator que necesiten en su momento de creación.

Adaptor Manager

Como hemos comentado en varios puntos de esta memoria, los adaptors son los elementos que lidian entre los Communicators (los mensajes que enviamos y recibimos) y el modelo (objetos "Node", la representación lógica de dispositivos físicos reales). Este manager cuenta con los siguientes métodos:

1. **get_adaptors_types()**: devuelve un lista de strings con los nombres de los tipos de los adaptors que están instalados en la distribución de IoTGate que se está utilizando.

2. **new_adaptor(name: str, adaptor_type:str, communicators_ids:list)**: permite crear un adaptor pasándole un nombre único (para poder identificarlo unívocamente más tarde), el nombre del tipo (obtenido mediante el método descrito en el punto anterior) y una lista de identificadores de communicators (se pueden consultar los communicators disponibles mediante el communicator manager).

3. **get_managing_adaptors_names()**: devuelve una lista de strings con los identificadores de los adaptors que están registrados en el sistema.

4. **get_adaptors()**: devuelve una lista con todos las instancias de adaptors registradas en el sistema.

5. **get_adaptor_by_name(adaptor_name: str)**: permite obtener una instancia de adaptor a partir de su nombre único.

6. **add_node(node: Node)**: permite añadir una instancia de nodo a un adaptor, para que éste le vaya actualizando a lo largo de la ejecución el estado de sus componentes y permita el envío de acciones. Este método será utilizado, normalmente, por una instancia de "NodeManager".

7. **delete_node(node: Node)**: permite que el adaptor deje de gestionar el nodo escogido. Este método será utilizado, normalmente, por una instancia de "NodeManager".

Aparte de nosotros, también hará uso de este manager el manager de nodos: Cuando sea instanciado y cargue la red, o cuando creemos un nuevo nodo, hará uso de este manager para añadir a cada adaptor los nodos que tiene que gestionar.

Node Manager

Por último, tenemos el manager de nodos. Mediante este manager podremos hacer gestión y uso de nuestra red de dispositivos virtuales, cuya información y funcionalidad está soportada por los elementos anteriores. Este manager proporciona las siguientes operaciones:

1. **create_node_group(node_group_id: str, description: str)**: esta operación crea un grupo de nodos vacío a partir de un nombre único y de una descripción (el usuario es libre de poner la descripción que considere). Devuelve una instancia de "NodeGroup".

2. **delete_node_group(node_group_id: str)**: borra un nodo de grupos a partir del nombre del grupo a borrar. Para borrar un grupo, primero ha de estar vacío, es decir, no se puede borrar ningún grupo que contenga nodos.

3. **create_node(alias: str, network_id:str, node_group:str, product:str, adaptor:str, configuration={})**: esta operación crea un nodo. Para crear un nodo, necesitamos todos estos parámetros:

   - alias: nombre amigable para el usuario, ha de ser único bajo el grupo de nodos en el que se vaya a introducir este nodo.
   - network_id: es un identificador único de nodo con el que se trabajará en los adaptors.
   - node_group: es el nombre del grupo en el que se va encontrar el nodo en un primer instante. Todos los nodos han de tener grupo, y su alias ha de ser único dentro del grupo. El grupo se puede cambiar más tarde.
   - product: es el identificador de producto al que este nodo corresponde.

- adaptor: es el identificador del adaptor que va a gestionar este nodo.
- configuration: es la configuración del nodo, en términos clave-valor. Esta configuración es opcional, y si no se establece, por defecto estará vacío. Este elemento nos permite guardar cualquier información adicional en el nodo, y normalmente esta información sera utilizada en el adaptor.

Devuelve una instancia de "Node" lista para usar (y ubicada ya en el grupo indicado).

4. **delete_node(node_group_id: str, node_alias: str)**: desde fuera, la manera de identificar un nodo será a través de la unicidad de la tupla (identificador de grupo, alias del nodo). Una vez localizado, esta operación borra el nodo del modelo.

5. **swap_node(node: str, from_group: str, to_group: str)**: Esta operación permite cambiar un nodo de grupo. Estableciendo un identificador de grupo y un alias, se localiza un nodo que será cambiado al grupo identificado con el nombre "to_group".

6. **set_node_config(node_group_name, node_alias, configuration)**: permite cambiar la configuración de un nodo concreto.

7. **get_node_groups_names()**: devuelve una lista de strings con todos los identificadores de grupos registrados en el sistema.

8. **get_node_group(name)**: devuelve la instancia de "NodeGroup" cuyo nombre sea el facilitado.

9. **get_nodes_groups()**: devuelve una lista de instancias de "NodeGroup".

10. **get_node_by_alias(node_group_id: int, node_alias: str)**: devuelve una instancia de "Node", introduciendo el identificador de grupo al que pertenece y su alias.

11. **listen_all_groups(event_type: str, callback)**: permite añadir una función callback que se ejecute cuando cualquiera de las instancias de "NodeGroup" lancen el evento especificado en la documentación NodeGroup.

12. **listen_all_nodes_of_a_group(node_group_id: str, event_type: str, callback)**: permite añadir una función callback a todos los nodos de un grupo. Para conocer que tipos de evento lanzan las instancias de "Node" (y como implementar las callbacks en consecuencia), consúltese la documentación de "Node".

13. **listen_all_nodes(event_type: str, callback)**: es una operación similar a la anterior, sólo que en esta ocasión la callback se añade a todos los nodos registrados en el sistema.

14. **get_state_by_date(node_group_id: str, node_id: int, board_id: str, from_date, to_date)**: devuelve una lista de estados de componente (instancias de NodeComponentState), a partir del identificador de grupo, alias de nodo y board_id del componente, entre las fechas establecidas.

15. **get_last_state(node_group_id: str, node_id: int, board_id: str)**: parecido al método anterior, sólo devuelve el último estado registrado (es decir, una instancia de NodeComponentState). Si nunca se ha recibido información de este componente, también se devuelve una instancia de "NodeComponentState", pero los atributos de éste se encuentran en el valor "None" (null de Python).

En esta ocasión, de este manager sólo haremos uso los usuarios; ningún elemento del sistema más lo manipula. Por otra parte, al estar el ORM integrado con el sistema, los grupos, nodos (y su configuración) y datos de nodos van quedando persistentes.

Ejemplo III: creación de API REST mediante Flask

Durante el desarrollo de este ejemplo, vamos a implementar una pequeña API REST que permita el uso remoto de ciertas partes del sistema utilizando "Flask", un microframerk web para Python [76].

Aunque la integración con todas las funcionalidades de IoTGate no será completa, nos permitirá explicar el uso descrito en la sección anterior.

Vamos a seguir los siguientes pasos:

1. Primero, vamos a importar y a instanciar el "Engine":

```
1  import json
2  from flask import Flask
3  from flask import request
4  from iotgate.core.main import Engine
5
6  IOTGATE_ETC_FILE = '/etc/iotgate/test.conf'
7  HOST = "0.0.0.0"
8  PORT = 5000
9
10 app = Flask(__name__)
11 engine = Engine(configuration_path=IOTGATE_ETC_FILE)
```

Figura 5.8: Ejemplo III: instanciando el "Engine"

En la figura anterior, observamos:

- Importamos la librería json, que nos permitirá serializar/deserializar los objetos diccionario cuyas calves-valores son también tipos básicos.
- Importamos los elementos de Flask que necesitamos para trabajar:
  - la clase Flask nos permitirá instanciar lo que se conoce como "app" en este framework. La "app" hace de servidor: nos permite una configuración donde podemos elegir en que host y en qué puerto tiene que escuchar.
  - el objeto request, que nos permite recuperar la información que el cliente nos adjunte en su petición HTTP.
- Para instanciar un Engine, necesitamos como mínimo indicarle la ruta de la configuración, que debe ser un fichero como el que se especifica en la figura 4.14.

En este punto, estamos listos para empezar a usar IoTGate. En esta ocasión, nuestro objetivo es poner a disposición remota el uso del mismo.

2. Vamos a comenzar exponiendo la funcionalidad del manager de productos:

```
@app.route('/products')
def get_product_ids_list():
    products_list = engine.product_manager.get_products_id()

    return json.dumps(products_list)


@app.route('/products/<product_id>')
def get_product_by_id(product_id: str):
    product = engine.product_manager.get_product(product_id)

    return json.dumps(product.serialize())


@app.route('/products/register',  methods=['POST'])
def register_product():
    product_descriptor = request.get_json()
    engine.product_manager.register_product(product_descriptor)

    return json.dumps(product_descriptor)


@app.route('/products/unregister/<product_id>',  methods=['DELETE'])
def unregister_product(product_id: str):
    engine.product_manager.unregister_product(product_id)

    return ""
```

Figura 5.9: Ejemplo III: exposición de funcionalidad de "ProductManager" a través de API REST

3. Vamos a continuar exponiendo las funcionalidades de "CommunicatorManager":

```
@app.route('/communicators')
def get_comms_status():
    comm_states = engine.communicator_manager.get_communicators_status()
    return json.dumps(comm_states)


@app.route('/communicators/create', methods=['POST'])
def create_comm():
    creation_params = request.get_json()
    comm = engine.communicator_manager.new_communicator(**creation_params)
    return json.dumps(comm.serialize())


@app.route('/communicators/types')
def get_comm_types():
    comm_types = engine.communicator_manager.get_communicator_types()
    return json.dumps(comm_types)


@app.route('/communicators/<comm_type>/params')
def get_comm_type_config_param(comm_type: str):
    comm_config_params = engine.communicator_manager\
        .get_communicator_type_config_params(comm_type)
    return json.dumps(comm_config_params)
```

Figura 5.10: Ejemplo III: exposición de funcionalidad de "CommunicatorManager" a través de API REST

4. Continuamos con “AdaptorManager”:

```
@app.route('/adaptors')
def get_adaptor_ids_list():
    adaptors_list = engine.adaptor_manager.get_manage_adaptors_names()
    return json.dumps(adaptors_list)


@app.route('/adaptors/<adp_id>')
def get_adaptor_info_by_id(adp_id: str):
    adaptor = engine.adaptor_manager.get_adaptor_by_name(adp_id)
    return json.dumps(adaptor.serialize())


@app.route('/adaptors/types')
def get_adp_types():
    adp_types = engine.adaptor_manager.get_adaptors_types()
    return json.dumps(adp_types)


@app.route('/adaptors/create',  methods=['POST'])
def new_adaptor():
    adaptor_descriptor = request.get_json()
    engine.adaptor_manager.new_adaptor(**adaptor_descriptor)

    return json.dumps(adaptor_descriptor)
```

Figura 5.11: Ejemplo III: exposición de funcionalidad de “AdaptorManager” a través de API REST

5. Terminamos exponiendo “NodeManager”:

```
@app.route('/nodes/group')
def get_node_groups():
    node_groups = engine.node_manager.get_nodes_groups()
    list_of_groups = [node_group.serialize() for node_group in node_groups]

    return json.dumps(list_of_groups)

@app.route('/nodes/group/create')
def create_node_group():
    group_desc = request.get_json()
    engine.node_manager.create_node_group(**group_desc)

    return json.dumps(group_desc)

@app.route('/nodes/node/create')
def create_node():
    node_desc = request.get_json()
    engine.node_manager.create_node(**node_desc)

    return json.dumps(node_desc)

@app.route('/nodes/action', methods=['POST'])
def do_action():
    action_desc = request.get_json()
    node = engine.node_manager.get_node(action_desc['node_group'], action_des['node_alias'])
    actuator = node.get_actuator(action_desc['actuator'])
    state = actuator.do_action(action_desc['action_name'], action_desc['action_params'])

    return json.dumps(state.serialize())
```

Figura 5.12: Ejemplo III: exposición de funcionalidad de “NodeManager” a través de API REST

6. Por último, añadimos las líneas que van a permitir la instanciación y escucha de peticiones:

```
if __name__ == '__main__':
    app.run(host=HOST, port=PORT)
    engine.safe_close()
```

Figura 5.13: Ejemplo III: puesta en marcha de la API REST

5.2.3. Nivel III: integrando a través de "deployment"

Cuando hagamos uso de este último nivel, relegaremos el uso de los elementos del "Nivel II" a las instancias de los objetos de la capa "deployment::helpers" (ver sección 4.3.4).

Con las ideas presentadas en esa sección, recordamos que:

- Utilizaremos "deployment::helpers::Publisher" para dar soporte al modelo de comunicación Publish-subcribe (ver sección 2.3.3).
- Utilizaremos "RequestProxy" para dar soporte al modelo de comunicación Resquest-Response (ver sección 2.3.3)

Publisher

Publisher es una clase abstracta. Para usarla, lo único que tenemos que hacer es heredar de ella e implementar el método "publish(topic: str, serial_value:dict).

Veamos un ejemplo:

```
import time
from iotgate.core.main import Engine
from iotgate.deployment.helpers import Publisher

class ConcretePublisher(Publisher):
    def publish(topic: str, serial_value:dict):
        print(topic, serial_value)

if __name__ == '__main__':
    engine = Engine(configuration_path='/etc/iotgate/tfg-config.conf')
    publisher = ConcretePublisher(engine)

    try:
      while True:
          time.sleep(1)
    except KeyboardInterrupt:
      engine.safe_close()
```

Figura 5.14: Ejemplo: uso de "deployment::helpers::Publisher

Cuando ejecutemos el programa de la figura, veremos por pantalla una línea indicando un cambio de estado en un nodo. El formato de "topic" y "serial_value" está explicado en la sección 4.3.4.

Request Proxy

Como vimos en la sección 4.3.4, RequestProxy es una clase que mediante reflexión expone los elementos de la capa "core::model::managers". Para este fin, proporciona un único método a utilizar:

- **do_request(to_manager: str, operation: str, parameters: dict)**: permite hacer peticiones a los managers del sistema, y devuelve una respuesta serializada. Sus argumentos son los siguientes:
  - **to_manager**: indica el nombre del manager a quien queremos hacer la petición. Este valor puede ser CommunicatorManager, AdaptorManager, ProductManager o NodeManager.
  - **operation**: indica la operación del adaptor concreto que queremos realizar. Independientemente del manager, existe una operación especial "get_operations", que devuelve un diccionario cuyas claves son el nombre de la operación y su valores son listas de argumentos a establecer, para realizar la operación.
  - **parameters**: es un diccionario cuyas claves son el nombre de argumento, y sus valores son el valor que queremos que tome el argumento durante la operación.

En las siguientes secciones, veremos una serie de ejemplos de uso de RequestProxy y Publisher junto a otras librerías.

Ejemplo IV: creación de API mediante AMQP

En este ejemplo, vamos a realizar una integración mínima con el protocolo AMQP. AMQP es un protocolo estándar abierto de la capa de aplicación (ver sección 2.3.2).

Para la realización de este ejemplo, utilizaremos:

- **RabbitMQ**, un broker compatible con AMQP v 0.9.1. [78]
- **pika**, una librería de Python que implementa la versión 0.9.1 del protocolo AMQP. [79]

Por ello, vamos a importar las librerías "iotgate" y "pika" y a definir las siguientes constantes:

```
import pika
import json
import time
from threading import Thread
from iotgate.core.main import Engine
from iotgate.deployment.helpers import Publisher
from iotgate.deployment.helpers import RequestProxy

EXCHANGE = 'iotgate_test'
IOTGATE_CONFIGURATION_PATH = '/etc/iotgate/tfg-config.conf'
AMQP_URL = 'amqp://iotgate:*******@virtual.lab.inf.uva.es:20072/'
```

Figura 5.15: Ejemplo IV: importación de librerías y definición de constantes

Vamos a necesitar implementar varios elementos, para dar soporte a los dos modelos de comunicación que necesitamos: request-response y publish-subscribe:

1. Comenzamos con la implementación del publicador:

```
class AMQPPublisher(Publisher):
    def __init__(self, engine, amqp_url):
        Publisher.__init__(self, engine)
        self.connection = pika.BlockingConnection(pika.URLParameters(amqp_url))
        self.channel = self.connection.channel()

        self.channel.exchange_declare(exchange=EXCHANGE,
                                      type='topic')

    def publish(self, topic, serial_value):
        message = json.dumps(serial_value)
        self.channel.basic_publish(exchange=EXCHANGE,
                                   routing_key=topic,
                                   body=message)
```

Figura 5.16: Ejemplo IV: implementación de publicador AMQP

2. Ahora, continuamos con la implementación del servidor "RPC" con AMQP:

```
class AMQPServer(Thread):
    def __init__(self, engine, amqp_url):
        Thread.__init__(self)
        self.proxy = RequestProxy(engine)
        self.connection = pika.BlockingConnection(pika.URLParameters(amqp_url))
        self.channel = connection.channel()
        self.channel.queue_declare(queue='rpc_queue')

    def on_request(ch, method, props, body):
        request = json.loads(body)
        reply = self.proxy.do_request(request['to'], request['operation'], request['params'])

        ch.basic_publish(exchange='',
                         routing_key=props.reply_to,
                         proprties=pika.BasicProperties(correlation_id = \
                                                        props.correlation_id),
                         body=json.dumps(reply))
        ch.basic_ack(delivery_tag = method.delivery_tag)

    def run(self):
        self.channel.basic_qos(prefetch_count=1)
        self.channel.basic_consume(self.on_request, queue='rpc_queue')
        self.channel.start_consuming()
```

Figura 5.17: Ejemplo IV: implementación del servidor RPC AMQP

3. Para finalizar, pongamos a funcionar todos los elementos:

```
if __name__ == "__main__":
    engine = Engine(IOTGATE_CONFIGURATION_PATH)
    publisher = AMQPPublisher(engine, AMQP_URL)
    server = AMQPServer(engine, AMQP_URL)
    server.start()

    try:
      while True:
          time.sleep(1)
    except KeyboardInterrupt:
      engine.safe_close()
```

Figura 5.18: Ejemplo IV: implementación de main de la integración con AMQP

Consultando la referencia sobre RabbitMQ [78], podemos comprobar que existen múltiples librerías compatibles en diferentes lenguajes. Podríamos, por tanto, crear un Suscriptor y un cliente RPC en Java. Como todas las funcionalidades del sistema quedan expuestas, podríamos tener un control absoluto en remoto y desde otros lenguajes.

Ejemplo V: creación de API mediante ZeroMQ

Durante el desarrollo de este ejemplo, veremos como integrar IoTGate con una tecnología similar a AMQP, pero lo suficientemente distinta como para que suponga ser un ejemplo interesante.

ZeroMQ, más que un protocolo, es una implementación alternativa de los sockets convencionales; que permite implementar muy fácilmente diferentes modelos de comunicación. Al

estar implementado en varios lenguajes, resulta ser una solución muy interesante para integrar software en distintos lenguajes de programación.

Durante este ejemplo, crearemos los mismos elementos que en el anterior, con un nuevo añadido: ZeroMQ no tiene "broker" como tal, es decir, no existe el homólogo de RabbitMQ, por lo que tendremos que implementar un broker muy simple:

1. Empezamos importando las librerías necesarias:

```
import time
import zmq
from threading import Thread
from iotgate.core.main import Engine
from iotgate.deployment.helpers import Publisher
from iotgate.deployment.helpers import RequestProxy
```

Figura 5.19: Ejemplo V: importando las librerías necesarias

2. Continuamos con la implementación de un publicador ZeroMQ:

```
class ZMQPublisher(Publisher):
    def __init__(self, engine, host, port):
        Publisher.__init__(self, engine)
        self.context = zmq.Context()
        self.publisher = self.context.socket(zmq.PUB)
        self.publisher.bind(host + ":%s" % port)

    def publish(topic: str, serial_value:dict):
        self.publisher.send_multipart([topic.encode(), json.dumps(serial_value).encode()])
```

Figura 5.20: Ejemplo V: implementación de publicador mediante ZeroMQ

3. Seguimos con la implementación del broker:

```
class ZMQBroker(Thread):
    def __init__(self, frontend_host, backend_host):
        context = zmq.Context()
        self.frontend = context.socket(zmq.ROUTER)
        self.backend = context.socket(zmq.DEALER)

        self.frontend.bind(frontend_host)
        self.backend.bind(backend_host)

        self.poller = zmq.Poller()
        self.poller.register(frontend, zmq.POLLIN)
        self.poller.register(backend, zmq.POLLIN)

        self.work = True

     def stop(self):
        self.work = False

     def run(self):
        while self.work:
            socks = dict(self.poller.poll())

            if socks.get(self.frontend) == zmq.POLLIN:
                message = self.frontend.recv_multipart()
                self.backend.send_multipart(message)

            if socks.get(self.backend) == zmq.POLLIN:
                message = self.backend.recv_multipart()
                self.frontend.send_multipart(message)
```

Figura 5.21: Ejemplo V: implementación de broker mediante ZeroMQ

4. El único elemento que nos queda es una clase que atienda a las peticiones que lleguen desde el broker:

```
class ZMQWorker(Thread):
     def __init__(self, engine, host):
         context = zmq.Context()
         self.proxy = RequestProxy(engine)
         self.socket = context.socket(zmq.REP)
         self.socket.connect(host)

         self.work = True

     def stop(self):
        self.work = False

     def run(self):
         while self.work:
             request = json.loads(self.socket.recv())
             reply = self.proxy.do_request(request['to'], request['operation'], request['params'])
             socket.send(json.dumps(reply).encode())
```

Figura 5.22: Ejemplo V: implementación de "worker" mediante ZeroMQ

5. Por útlimo, ponemos todo en marcha:

```
if __name__ == '__main__':
    engine = Engine(configuration_path='/etc/iotgate/tfg-config.conf')
    publisher = ZMQPublisher(engine)

    broker = ZMQBroker('tcp://*:5559', 'tcp://*:5560')
    broker.start()
    worker = ZMQWorker(engine, 'tcp://localhost:5560')
    worker.start()

    try:
      while True:
          time.sleep(1)
    except KeyboardInterrupt:
      engine.safe_close()
```

Figura 5.23: Ejemplo V: puesta en marcha de la integración de IoTGate con ZeroMQ

Al igual que en el ejemplo anterior, consultando la referencia sobre ZeroMQ [40] podemos encontrar implementaciones de la librería de ZeroMQ en muchos otros lenguajes. ZeroMQ resulta ser una librería bastante agradable para montar "wrappers" en otros lenguajes. Mediante esto, podríamos ofrecer la misma API en muchos lenguajes de programación diferentes (recomiendo ver la "zguide" de ZeromMQ, es una documentación fantástica).

Ejemplo VI: creación de API mediante Flask

Como vimos en el desarrollo del ejemplo III (ver 5.2.2), podíamos integrar IoTGate con otros software mediante el considerado "Nivel II", sin hacer uso de la capa "deployment". Durante el desarrollo de este ejemplo, vamos a presentar la diferencia que hay -en código y responsabilidad - de utilizar el Nivel III en vez del II.

En este caso, no vamos a ser nosotros los que hagamos un uso directo de los managers que nos proporciona "Engine", sino que este uso se hará desde "deployment":

1. Primero, importemos las librerías necesarias y definamos constantes:

```
import json
from flask import Flask
from flask import request
from iotgate.core.main import Engine
from iotgate.deployment.helpers import RequestProxy

IOTGATE_ETC_FILE = '/etc/iotgate/config-tfg.conf'
HOST = "0.0.0.0"
PORT = 5000
```

Figura 5.24: Ejemplo VI: importación de librerías y definición de constantes

2. Continuemos escribiendo las rutas que nos permitirán obtener información acerca de las operaciones disponibles:

```
@app.route('/', methods=['GET'])
def get_info():
    doc = """
        <h2> To start using this API: </h2>
        <ul>
            <li>[GET] /CommunicatorManager -> info about communicator manager's available operations</li>
            <li>[GET] /AdaptorManager -> info about adaptor manager's available operations</li>
            <li>[GET] /ProductManager -> info about product manager's available operations</li>
            <li>[GET] /NodeManager -> info about node manager's available operations</li>
        </ul>
    """

    return doc


@app.route('/<to>', methods=['GET'])
def get_operations_info(to: str):
    info = proxy.do_request(to, 'get_operations', {})

    return json.dumps(info['return'])


@app.route('/<to>/<operation>', methods=['GET'])
def get_operation_info(to: str, operation: str):
    info = proxy.do_request(to, 'get_operations', {})

    return json.dumps(info['return'][operation])
```

Figura 5.25: Ejemplo VI: provisión de ayuda mínima para el cliente

3. Ahora, vamos a escribir la ruta mediante la cual se podrán hacer operaciones contra IoTGate:

```
@app.route('/<to>/<operation>', methods=['POST'])
def do_request(to: str, operation: str):
    kwargs = request.get_json()
    reply = proxy.do_request(to, operation, kwargs)

    return json.dumps(reply)
```

Figura 5.26: Ejemplo VI: uso de RequestProxy con Flask

4. Por último, pongamos todo a funcionar:

```
if __name__ == '__main__':
    app = Flask(__name__)
    engine = Engine(configuration_path=IOTGATE_ETC_FILE)
    proxy = RequestProxy(engine)

    app.run(host=HOST, port=PORT)
    print("Closing engine..")
    engine.safe_close()
```

Figura 5.27: Ejemplo VI: Código "main" para puesta en marcha de la API

Como podemos observar en este ejemplo, sucede una cosa bastante interesante: ahora, la responsabilidad de hacer uso del engine corre de cuenta de la capa "deployment", lo que hace muchísimo más rápido la exposición de funcionalidades de IoTGate. Por otra parte, estamos

incrementando la complejidad en el cliente de la API.

Otro tema que puede resultar interesante es el uso de flask-socketio [77], que es una extensión de Flask que permite, muy fácilmente, añadir compatibilidad con WebSockets. Esto nos permitiría hacer uso de "Publisher", para dar soporte al modelo de comunicación de publicación-subscripción.

5.3. Extensibilidad

Como hemos comentado en capítulos anteriores, la extensibilidad de IoTGate tiene que ser una de sus principales ventajas frente a otros desarrollos similares. La extensibilidad de IoTGate se encuentra, principalmente, en las capas "communicators" y "adaptors". Para extender IoTGate, hay que cubrir comunicación con dispositivos a varios niveles:

1. comunicación bus en serie: en el paquete "communicators::buses", encontraremos implementaciones que nos permitirán utilizar I2C [63] y SPI [62]. Para poder hacer uso de otro tipo de buses, habría que implementarlos e incluirlos; para poder proporcionar comunicación a las clases de "communicators::drivers" o a las de "communicators::transport".
2. con el soporte de los elementos de la capa anterior, hay que implementar el driver del dispositivo a controlar, o envolverlo en implementaciones de "Client" y "Server" (si pretendemos conectarnos a un dispositivo de Tipo I directamente).
3. Si estamos implementando un driver, tendremos que consultar el datasheet del módulo en cuestión y hacer lo que se nos indique; o encontrar alguna implementación realizada para otra plataforma y adaptarla.
4. Cuando dispongamos ya de una forma de trabajar a través de instancias de "Client" y "Server", tendremos que implementar un "Adaptor" que nos permita hacer uso en "core" de nuestras instancias de "Node".

Por estas razones, los siguientes apartados describirán cómo se han integrado dos elementos hardware en IoTGate: un módulo transceptor y un dispostivo de Tipo I.

5.3.1. Integración de módulos transceptores

Para comenzar esta sección, es importante recalcar los siguientes puntos:

- La mayoría de los módulos transceptores cuentan con una interfaz SPI [62] o UART [64].
- Existen varios módulos comerciales que dan la posibilidad de elegir entre cualquiera de los dos.

- En cualquier caso, estos módulos contarán con una serie de patillas que no tendrán que ver directamente con la comunicación mediante su interfaz (SPI, UART), pero que serán determinantes para su correcto funcionamiento:
  - Alimentación: todos los módulos cuentan con un pin que normalmente se alimenta a 5V, y otra patilla para tierra (GND).
  - Entradas/salidas digitales (RESET, WAKE, SLEEP,INT, CS...). Este tipo de patillas nos permiten apagar, encender, reiniciar, depertar/dormir (en dispositivos de bajo consumo) el módulo (WAKE, SLEEP, RESET); nos notifican de interrupciones (INT); nos permiten indicar que vamos a comunicar (CS, chip select...).

  Con estos puntos en mente, determinamos que para integrar drivers necesitamos:

  - Hacer uso de las entradas y salidas digitales que nos proporcione el SBC [14], para poder controlar el módulo que estamos integrando, y detectar sus interrupciones.
  - Hacer uso de las comunicaciones en serie (SPI y UART, entre otras)

Con esta pequeña síntesis entre manos, vamos a presentar las partes fundamentales de un elemento de drivers en la siguiente sección, para que sirva de ejemplo.

Ejemplo VII: Integración de módulo transceptor

Durante el desarrollo de este ejemplo, veremos cómo integrar el módulo transceptor MRF24J40MC [60] con IoTGate. Para ello, tomaremos como plataforma una Raspberry Pi 2 [14], dado que nos proporciona todas las características que necesitamos a través de su GPIO [66]:

- Tiene la capacidad de alimentar el módulo.
- Cuenta con una serie de entradas y salidas digitales que son configurables.
- Cuenta con una interfaz SPI, que es la que proporciona el módulo MRF24J40MC.

Para realizar la integración de este hardware, llevaremos a cabo los siguientes pasos:

1. Conectar adecuadamente el módulo transceptor al la GPIO de la Raspberry:
   - Esta tarea requiere conocimientos específicos sobre electrónica. Existen varios módulos comerciales que cuentan con integraciones específicas para Rapberry. En esta ocasión, vamos a utilizar una adaptación fabricada por Argotec:*imagen*
   - Como podemos observar en la figura anterior, ya contamos con el nuevo hardware conectado al SBC. Necesitaremos conocer las especificaciones de la placa de expansión para conocer que patilla del dispositivo está conectada a que patilla de la GPIO del SBC.
2. El siguiente paso será documentarnos, es decir, ¿cómo utilizamos el módulo?

- En este paso, lo más fácil suele ser localizar un driver ya implementado (aunque sea en un lenguaje distinto a Python), para poder hacerse rápidamente una idea de cómo funciona. Siempre hay que tener a mano el datasheet del fabricante, pero es mucho más rápido consultar cosas concretas que "descrifrarlo".
- Si no se localiza, no quedará más remedio que armarse de paciencia e ir desgranando las instrucciones que proporcione el fabricante del módulo.
- En esta ocasión, localizamos varias implementaciones del driver:

Driver I Driver en Python para RPi, última consulta a 14 de julio de 2016
Driver II Driver en C para el SO Contiki, última consulta a 14 de julio de 2016
Driver III Driver en C para el kernel de Linux, última consulta a 14 de julio de 2016

3. Ahora que contamos con toda la información necesaria, podemos comenzar a trabajar. Nuestro objetivo es dar soporte a elementos de la capa "communicators::transport" mediante los cuáles estableceremos conexión con los dispositivos. Tomaremos como referencia el primer código propuesto, dado que está en Python y es el más simple (sobre todo por que es el más simple):

   - Primero, vamos a importar las librerías que necesitamos:

```
1 import datetime
2 import _thread
3 import time
4 import traceback
5 from threading import Lock
6
7 import RPi.GPIO as GPIO
8 from iotgate.communicators.buses.spi import SPI
9 from iotgate.core.model.patterns import Singleton
```

Figura 5.28: Ejemplo VII: importación de librerías necesarias

4. Vamos a continuar estableciendo una serie de constantes:

```
1 # MRF24J40MC pins -> RPi GPIO Pins
2 RESET = 21
3 WAKE = 16
4 INTERRUPT = 24
5 STATE = 26
6 CHIP_SELECT = 7
7 # Interrupt signals values
8 RX_INTERRUPT_SIGNAL = 8
```

Figura 5.29: Ejemplo VII: definición de constantes pins mrf24j40 ->GPIO RPi

5. Vamos a continuar configurando la GPIO de la Raspberry,

```
    def init_rpi_gpio(self):
        GPIO.setmode(GPIO.BCM)
        # nReset
        GPIO.setup(RESET, GPIO.OUT)
        # GPIO0 WAKE from module
        GPIO.setup(WAKE, GPIO.OUT)
        # GPIO1 INTERRUPT from module
        GPIO.setup(INTERRUPT, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
        GPIO.add_event_detect(INTERRUPT, GPIO.FALLING, callback=self.treat_irq, bouncetime=10)
        # State
        GPIO.setup(STATE, GPIO.OUT)
        # Chip select
        GPIO.setup(CHIP_SELECT, GPIO.OUT)
```

Figura 5.30: Ejemplo VII: configuración de las E/S digitales de la GPIO (RPi)

6. Acto seguido, vamos a a poner a funcionar las E/S digitales como se indica en el datasheet, y a configurar el módulo tal cual lo hace el creador de "Driver I":

   a) Primero, ponemos a funcionar la E/S:

```
    def configure_module(self):
        # Wake up!
        GPIO.output(WAKE, True)
        GPIO.output(CHIP_SELECT, False)
        # We reset the module
        GPIO.output(RESET, False)
        time.sleep(0.1)
        GPIO.output(RESET, True)
```

Figura 5.31: Ejemplo VII: valores de inicio de las E/S de la GPIO (RPi)

   b) Continuamos limpiando las colas FIFO de envío/recepción, y estableciendo el modo de funcionamiento:

```
        # 0x07 - Perform a software Reset. Bit is auto-reset
        self.spi.write("SOFTRST", 0x07)
        # 0x98 - Initialize FIFOEN = 1 and TXONTS = 0x6.
        self.spi.write("PACON2", 0x98)
        # 0x95 - Initialize RFSTBL = 0x9.
        self.spi.write("TXSTBL", 0x95)
        # clean rx fifo
        self.spi.write("RXFLUSH", 0x01)
        # 11. BBREG2 (0x3A) = 0x80 - Set CCA mode to ED.
        self.spi.write("BBREG2", 0x80)
        # 12. CCAEDTH = 0x60 - Set CCA ED threshold.
        self.spi.write("CCAEDTH", 0x60)
```

Figura 5.32: Ejemplo VII: Configuración del módulo MRF24J40MC (I)

   c) Ahora, vamos a configurarle el reloj interno y a configurarle para que nos reporte el RSSI con el que llegan los mensajes:

```
        # 0x01 - Initialize VCOOPT = 0x02.
        self.spi.write("RFCON1", 0x01)
        # 0x80 - Enable PLL (PLLEN = 1).
        self.spi.write("RFCON2", 0x80)
        # 0x90 - Initialize TXFIL = 1 and 20MRECVR = 1.
        self.spi.write("RFCON6", 0x90)
        # 0x80 - Initialize SLPCLKSEL = 0x2 (100 kHz Internal oscillator).
        self.spi.write("RFCON7", 0x80)
        # 0x10 - Initialize RFVCO = 1.
        self.spi.write("RFCON8", 0x10)
        # 0x21 - Initialize CLKOUTEN = 1 and SLPCLKDIV = 0x01.
        self.spi.write("SLPCON0", 0x01)
        self.spi.write("SLPCON1", 0x20)
        # 13. BBREG6 (0x3E) = 0x40 - Set appended RSSI value to RXFIFO.
        self.spi.write("BBREG6", 0x40)
```

Figura 5.33: Ejemplo VII: Configuración del módulo MRF24J40MC (II)

d) Continuamos configurando la interrupción (nos indicará cuando está llegando un mensaje -entre otras cosas-, para que actuemos en consecuencia); después configuramos los parámetros de red (qué canal, que potencia, controller_id, PAN_id). Después, limpiamos el registro de direcciones de red extendidas, y establecemos el PAN_coordinator. Para finalizar, reiniciamos la máquina de estados interna del módulo. Estamos listos para enviar y recibir ahora:

```
        ##
        # Interrupt config
        ##
        self.spi.write("INTCON", 0x00)
        # Set channel
        self.spi.write("RFCON0", channel[int(self.configuration['channel'])] | 0x02)
        self.spi.write("RFCTL", 0x04)
        self.spi.write("RFCTL", 0x00)

        time.sleep(0.1)
        # Transmission set to maximum level (0 dBm) (MC Version)
        self.spi.write("TRISGPIO", 0x08)
        self.spi.write("GPIO", 0x08)
        self.spi.write("RFCON3", 0x28)
        self.spi.write("TESTMODE", 0xf)
        # Set controller ID
        controller_id = str(self.configuration["controller_id"])
        high, low = int("0x" + controller_id[:2], 16), int("0x" + controller_id[2:], 16)
        self.spi.write("SADRL", low)
        self.spi.write("SADRH", high)
        # Set PANID
        pan_id = str(self.configuration["pan_id"])
        high, low = int("0x" + pan_id[:2], 16), int("0x" + pan_id[2:], 16)
        self.spi.write("PANIDL", low)
        self.spi.write("PANIDH", high)
        # Crear EADR0-7 extended address register
        for i in range(8):
            self.spi.write("EADR" + str(i), i)
        self.spi.write("RXMCR", 0x23)

        # 0x04 - Reset RF state machine.
        self.spi.write("RFCTL", 0x04)
        self.spi.write("RFCTL", 0x00)
```

Figura 5.34: Ejemplo VII: Configuración del módulo MRF24J40MC (III)

e) Ahora, vamos a ver el código de la callback que se ejecuta en caso de interrupción:

```
    def treat_irq(self, gpio_number):
        int_reg = self.spi.read("INTSTAT")
        if int_reg == RX_INTERRUPT_SIGNAL:
            frame = self.get_rxfifo()
            device_id = self.read_frame(frame["rssi"], frame["bytes"])
            self.server_request_func(self.configuration, device_id)
```

Figura 5.35: Ejemplo VII: código para el tratamiento de la interrupción MRF24J40MC

Este punto es clave, como veremos más adelante en la implementación del "Client" y el "Server".

f) Ahora, vamos a echar un vistazo a las funciones de envío y recepción de tramas:

```
    def get_rxfifo(self):
        with self.interrupt:
            self.spi.write("BBREG1", 0x04)
            mplusnplus2 = self.spi.read(0x300)
            read_bytes = []
            for i in range(mplusnplus2 + 2):
                read_bytes.append(self.spi.read(0x301 + i))
            self.spi.write("RXDECINV", 0)
            rssi = int(((read_bytes[-1] / 255.) * 70) - 90)
            data = dict(bytes=read_bytes[:-4], fcs=read_bytes[-4:-2], lqi=read_bytes[-2], rssi=rssi)
            # Allow to receive multiple message at same time
            self.spi.write("RXFLUSH", 0x61)
            self.spi.write("BBREG1", 0x00)
            time.sleep(0.150)
            # Clean FIFO
            self.spi.write("RXFLUSH", 0x01)

            # Reboot module
            self.reinit_radio()
        return data
```

Figura 5.36: Ejemplo VII: Lectura de la FIFO de recepción del módulo MRF24J40MC

```
    def read_message(self, device_id):
        if device_id in self.frame_register:
            if len(self.frame_register[device_id]) != 0:
                return self.frame_register[device_id].pop()
            else:
                return False
        else:
            return False
```

Figura 5.37: Ejemplo VII: implementación de pila de mensajes para el driver MRF24J40MC

```
    def send_message(self, message, destiny):
        legacy = self.configuration.get('legacy', False)
        controller_id = (0x20, 0x00)
        destiny = (0x00, destiny)
        frame_bytes = [65, 200, 166, 69, 35, destiny[1], destiny[0], controller_id[1], controller_id[0],
                       0, 0, 0, 0, 0, 0]
        if legacy:
            frame_bytes.append(79)
        message = [ord(character) for character in message]
        frame_bytes += message
        if legacy:
            frame_bytes.append(79)

        self.write_txfifo(frame_bytes)
```

Figura 5.38: Ejemplo VII: envío de mensajes mediante el driver del MRF24J40MC

```
    def write_txfifo(self, bytes):
        while self.interrupt.locked():
            time.sleep(0.1)
        self.spi.write(0x000, len(bytes))
        self.spi.write(0x001, len(bytes))
        for i in range(2, len(bytes) + 2):
            self.spi.write(0x000 + i, bytes[i - 2])
        self.spi.write("TXNCON", 1)
        time.sleep(0.1)
        self.spi.write("RXFLUSH", 0x01)
        self.reinit_radio()
```

Figura 5.39: Ejemplo VII: escritura en la FIFO de transmisión del MRF24J40MC

El driver contiene más código que tiene que ver con la escritura en los registros mediante SPI, se puede consultar en el código de IoTGate, está localizado en communicators::drivers::microchip::MRF24J40M. La intención de esta sección es comentar los puntos clave del desarrollo.

7. Ahora, sólo nos queda proporcionar un “Client” y “Server”, para poder proporcionar su uso a “Nivel I”; o a los adaptors, para los usos a “Nivel II” y “Nivel III”:

```python
class IEEE802154Client(Client):

    CONFIGURATION_PARAMS = ['controller_id', 'device_id', 'pan_id', 'channel']

    def __init__(self, id: str, configuration: dict):
        self.radio = None
        Client.__init__(self, id, 'ieee802154.IEEE802154Client', configuration)

    def configure(self):
        self.radio = MRF24J40MRadio()

    def receive(self):
        message = self.radio.read_message(self.configuration['device_id'])
        self.trigger_event('client_rcv', communicator=self, frame=message)
        return message

    def send(self, message):
        self.radio.send_message(message, self.configuration['device_id'])
        self.trigger_event('client_send', communicator=self, frame=message)

    def connect(self):
        self.trigger_event('client_connection', communicator=self, configuration=self.configuration)

    def close(self):
        self.trigger_event('client_close', communicator=self, message='Close successfully')
```

Figura 5.40: Ejemplo VII: implementación de "Client" para comunicación mediante el MRF24J40MC

El código de servidor se mostraba en la Figura 4.26.

Como podemos observar, a falta de implementar un driver que proporcione un control completo de todas las funcionalidades que ofrece el módulo, contamos con todas las funcionalidades básicas para empezar a trabajar rápidamente, y contamos con la expresividad y herramientas que ofrece el uso de un lenguaje de programación de alto nivel como Python.

5.3.2. Integración de dispositivos de Tipo I

Para comenzar esta sección, es importante recalcar los siguientes puntos:

- La mayor parte de este tipo de dispositivos harán uso de I2C o UART, aunque pueden comunicarse con otro tipo de interfaces.
- Cuando integremos este tipo de dispositivos, no trabajaremos con elementos del paquete driver:
  - Haremos uso de elementos de "buses" a través de elementos de "transport".
  - Esto significa que el "driver" del dispositivo estará implementado en la capa adaptors.
- Existen dispositivos que nos permiten escuchar a muchos otros dispositivos (concentradores, convertidores). En estos casos, aunque queda libre a la elección del programador, puede ser más interesante considerar la integración en términos parecidos a los presentados en la sección anterior:

- Cuando tengamos un dispositivo que se comunique con varios, y pretendamos que esos varios aparezcan representados como varias instancias de "Node", entonces preferiremos el esquema de integración bus-driver-transport-adaptor.
- Cuando tengamos uno o varios dispositivos, y pretendamos que aparezca una sola instancia de "Node" representando a ese dispositivo o dispositivos, preferiremos el esquema bus-transport-adaptor.

Dicho esto, vamos a ver mediante un ejemplo (sección siguiente) como integrar un dispositivo en estas condiciones.

Ejemplo VIII: Integración de dispositivo de Tipo I

Para llevar a cabo este ejemplo, integraremos un dispositivo de Microchip, el MCP4728 [15]. Se trata de un conversor digital-analógico de de 4 canales. Para llevar a cabo la integración, consideraremos que es un nodo que cuenta con 4 actuadores, los cuales pueden regularse entre 0 y 10v:

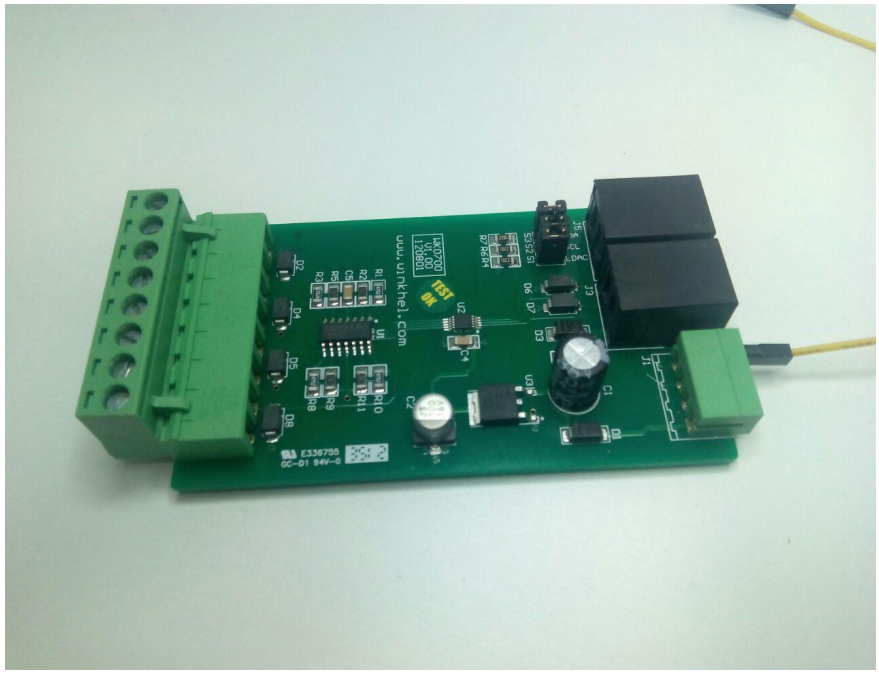

Figura 5.41: El dispositivo MCP4728

Como se trata de un sólo dispositivo que hace uso de I2C para funcionar, llevaremos a cabo la integración siguiendo el esquema bus-transport-adaptor:

1. Primero, vamos a implementar un "Client" que haga uso de la utilería de I2C que proporcionamos:

```
class I2CClient(Client):
    __mapper_args__ = {'polymorphic_identity': 'I2CClient'}

    def __init__(self, id, configuration):
        Client.__init__(self, id, 'i2c.I2CClient', configuration)
        self.port = None
        self.address = None
        self.connector = None
        self.register = None
        self.length = None
        self.configuration = None

    def configure(self):
        if "port" in self.configuration:
            self.port = int(self.configuration["port"])
        self.address = int(self.configuration["address"], 16)
        if "register" in self.configuration:
            self.register = int(self.configuration["register"])
            self.length = int(self.configuration["length"])

    def set_configuration(self, configuration):
        self.configuration = configuration

    def connect(self):
        if self.port:
            self.connector = I2C(address=self.address, bus_num=self.port)
            self.trigger_event("client_connect", communicator=self, configuration=self.configuration)
        else:
            self.connector = I2C(address=self.address)
            self.trigger_event("client_connect", communicator=self, configuration=self.configuration)

    def send(self, message):
        register = message["register"]
        values_to_write = message["values"]
        self.connector.write_list(reg=register, list=values_to_write)
        self.trigger_event("client_send", communicator=self, frame=message)

    def receive(self):
        response = self.connector.read_list(self.register, self.length)
        self.trigger_event("client_rcv", communicator=self, frame=response)

        return response

    def close(self):
        self.connector.close()
```

Figura 5.42: Ejemplo VIII: implementación de "Client" para hacer uso de bus I2C

2. Ahora, tenemos que implementar un Adaptor:

```
class MCP4728(Adaptor):
    __mapper_args__ = {'polymorphic_identity': 'argotec.netbows.MCP4728'}

    registers = {
        'a': 0x58,
        'b': 0x5A,
        'c': 0x5C,
        'd': 0x5E
    }

    def __init__(self, id, communicators):
        Adaptor.__init__(self, id, MCP4728, communicators)
        self.client = CommunicatorManager.create_communicator('i2c.I2CClient', self.configuration)

    def configure(self):
        self.client.connect()

    def set_vdd_to_channel(self, board_id, value):
        original_value = value
        value = float(value)
        channel = board_id
        value = int(value * 1000)

        value = int(value / 2.96)
        if value > 4095:
            value = 4095

        values_to_write = [(value >> 8) & 0xFF,
                           value & 0xFF]
        message = {
            'register': self.registers[channel],
            'values': values_to_write,
        }
        self.get_managed_node('mpc4728').update_values({channel: original_value})
        self.client.send(message)

        return str(original_value)

    def send_action(self, actuator, action, **params):
        self.set_vdd_to_channel(**params)
        return NodeComponentState(received_at=datetime.datetime.utcnow(),
                                  observad_at=datetime.datetime.utcnow(),
                                  value=str(params))

    def __del__(self):
        self.client.close()
```

Figura 5.43: Ejemplo VII: implementación de Adaptor para MCP4728

Capítulo 6

Líneas futuras

6.1. Mejoras en la capa "communicators"

Como hemos visto en capítulos anteriores, la capa "communicators" resulta ser una de las capas más importante del sistema. Las implementaciones que incluye de "Client" y "Server" en su paquete "transport" resultan ser la base sobre la que se establece prácticamente toda la funcionalidad del framework.

Durante las siguientes secciones, vamos a establecer cuál es el futuro de esta capa en líneas generales.

6.1.1. Mejora en el diseño de la capa

Esta capa resulta ser la más compleja en implementación, pero su diseño resulta simple. Partiendo de esta idea, hay que tomar en consideración los siguientes puntos en el futuro:

- Esta capa es la que más dependencias establece directamente con la plataforma hardware sobre la que se ejecute el sistema. Deberíamos concentrar todas las dependencias en los mismos puntos, para favorecer la reutilización máxima del código:
  - Sería interesante que toda implementación de "driver" tuviera dependencias sólo de la utilería básica (librería estándar) de Python y del propio framework (IoT-Gate).
  - El último punto implica una provisión mejor de herramientas, unas interfaces/clases abstractas (de la capa "buses") mucho mejor definidas.
- En esta capa trabajamos muy cerca del hardware, y esto tiene las siguientes implicaciones:

  - El acceso a ciertos elementos (como las entradas y salidas digitales de la GPIO, los buses de comunicación en serie) ha de ser exclusivo, es decir, tenemos recursos que no pueden ser utilizados por varios elementos hardware a la vez.
  - Hay otros elementos, como el SPI, que sí que pueden ser compartidos, pero ha de hacerse una gestión ordenada del recurso, estableciendo quien accede al medio cada vez.
  - Concluimos con que sería interesante hacer una gestión centralizada de estos recursos -quizá mediante una representación lógica de la placa/plataforma (SBC, PC, VM...)- permitiendo así el acceso controlado a los medios, y una mejor gestión de posibles problemas en configuración/uso.
  - Este hipotético "gestor de recursos" debería proporcionar la configuración, por una parte, la configuración de las entradas y salida digitales del sistema, por otra, acceso a la comunicación por bus/serie; y debería estar integrado con el sistema de eventos de IoTGate

- Como último punto a recalcar, debería replantearse el esquema de herencia de "Communicator": "Client" y "Server", ¿son verdaderamente especializaciones?
  - Quizá "Communicator" debería ser una implementación de "socket", mediante la cual, proporcionáramos una implementación de "Server".
  - ¿Cómo podríamos mejorar la integración con otras tecnologías como 6lowpan, LoRaWAN, KNX...? ¿Sería más interesante intentar una integración con las implementaciones que proporciona el kernel de Linux, o sería mejor contar con una implementación propia y más simple en este desarrollo? Si la respuesta correcta fuera "integrar con IoTGate", entonces ¿estas integraciones deberían hacerse en esta capa, o deberían aparecer en otra capa..?
- Si hiciéramos un planteamiento de desarrollo basado en componentes y servicios, ¿tendría sentido definir interfaces remotas aquí?

Como podemos ver, "communicators" es una de las capas más interesantes. Desde luego, va a crecer, y su crecimiento, ha de ser sometido a profunda reflexión.

6.1.2. Integración con más dispositivos hardware

A corto plazo, se planea integrar con IoTGate los siguientes módulos, proporcionándole así el entendimiento con nuevas tecnologías (y por tanto con más dispositivos): LoRa [61], KNX [33] y MBUS [83].

6.1.3. Compatibilidad con otras plataformas

Como planteábamos al principio de esta sección, muchos de los recursos del sistema deberían ser gestionados desde un único punto para hacer un mejor y más sólido aprovechamiento.

Ahora mismo, gracias a la elección de Python 3 como lenguaje de programación base, contamos con la compatibilidad multiplataforma que ofrece el intérprete. Al contar con "communicators" y "adaptors" que pueden ser instalados y desinstalados, no suele haber problemas en ese sentido. Pero si cambiáramos, por ejemplo, de una Rapberry a otro tipo de SBC, y quisiéramos reutilizar el driver implementado para el módulo MRF24J40, ¿cuánto esfuerzo supondría?

Para que el esfuerzo fuera mínimo, tendríamos que diseñar mejor la capa "communicators", para que las librerías dependientes de la plataforma (como RPi.GPIO) fueran fácilmente reemplazables. En la implementación de driver deberíamos hacer uso de una interfaz, y deberíamos poder cambiar la implementación de la misma con facilidad, incluyendo en esta las dependencias con la plataforma.

6.2. Mejoras en la capa "adaptors"

Como hemos visto a lo largo del trabajo, la misión fundamental de la capa "adaptors" es convertir la comunicación con los dispositivos físicos en representaciones lógicas muy sencillas de utilizar. Sus objetivos secundarios son ser legibles, permitiendo de un vistazo entender cómo se comunica el dispositivo, y conseguir tener implementaciones de protocolos que puedan ser utilizados con diferentes dispositivos independientemente de la tecnología que utilicen para establecer conexión.

Estos objetivos están, de un modo u otro, cubiertos. Sin embargo, la continuación del desarrollo del sistema nos hace plantearnos los siguientes puntos, presentados en las secciones que vienen a continuación.

6.2.1. Implementación de protocolos estándar

En muchas ocasiones, los dispositivos con los que hay que integrarse implementan -completa o parcialmente- protocolos estándar de comunicación.

Por ello, resultaría verdaderamente útil contar, a modo de utilería, con implementaciones de protocolos como MBUS y MODBUS, que pudieran ser utilizados más tarde en "Adaptors" para productos concretos. También sería interesante contar con herramientas de conversión de un tipo de datos a otro, haciendo más legibles y mantenibles los tratamientos de mensajes ("parseo" de tramas, digamos) que se presentan en los adaptors.

6.2.2. Relación de elementos de adaptor con el resto del sistema

Como hemos visto a lo largo del desarrollo del trabajo, los "Adaptors" tenían relación tanto con "Communicators" como con "Nodes", estableciéndose así como puente fundamental entre los mensajes y nuestra red lógica. Vamos a plantear las siguientes dudas a este diseño:

- Como explicaré en la siguiente sección, los adaptors están estrechamente ligados con el producto. Por ello, quizá debería especificarse en el producto qué adaptor y qué tecnología de comunicación utiliza el dispositivo, y en base a ello, crear los communicators pertinentes.
- El punto anterior llevaría a establecer una estricta relación de uso con los communicators, es decir, los adaptors no contendrían communicators, irían creando los elementos necesarios según se fueran añadiendo instancias de "Node" a gestionar.
- Llegados a este punto, ¿quién debería contener la configuración? No tiene sentido que la contenga "Product", dado que se trata de una especificación genérica. En este caso, debería tenerla "Node". Esto podría llegar a dar problemas (por ejemplo, dos nodos son añadidos a un adaptor con configuraciones incompatibles entre sí, causando que uno de los dos no funcione)
- La última idea planteada en el punto anterior lleva a pensar en el sistema de eventos de IoTGate y en la capa "communicators" como componente: ¿y si definimos una interfaz en la que un Adaptor concreto se encargue de pasar callbacks al componente de Communicators, y este crea un servidor bajo demanda, o añade la callback a un servidor ya creado, o simplemente rechaza la petición por la imposibilidad que supone el cumplimiento de la misma?

6.3. Mejoras en la capa "core"

Como hemos visto a lo largo del trabajo, la capa "core" es la base sobre la que "conectamos" y "orquestamos" el funcionamiento del resto del sistema, a la par que proporcionamos un servicio de persistencia y otro de logs.

La capa "core" es fundamental para la integración con otras aplicaciones, y define el "cómo" de muchas de prácticamente todas las funcionalidades que ofrecemos. Por ello, a futuro, tomaremos los puntos de las siguientes secciones como importantes puntos de reflexión de cara a desarrollar un mejor sistema.

6.3.1. Mejor tratamiento de excepciones

Contamos con una serie de reglas de negocio, establecidas por nosotros, como pueden ser:

- Todo nodo a de estar en un grupo.
- El nombre de cada grupo a de ser único en el sistema
- Bajo un grupo, el alias del nodo tiene que ser único.
- No de puede borrar un grupo que tenga nodos: primero habría que cambiar los nodos a otro grupo o borrarlos.

- Los componentes son de dos tipos, sensores y actuadores.
- Contamos con la tupla (nombre de grupo, alias de nodo) para identificar a un nodo en el sistema, desde las capas superiores. De cara a los adaptors, los nodos son identificados bajo su identificador de red (network_id), que ha de ser único desde el punto de vista de nodos gestionados por un adaptor.
- ...

El incumplimiento de estas reglas, así como conflictos de configuración y otros posibles errores no son tratados con rigor. De hecho, apenas contamos con excepciones personalizadas que nos permitan tratar correctamente todos estos posibles problemas. A futuro, va a ser fundamental revisar este punto.

6.3.2. Pruebas de unidad e integración

Las pruebas de software son fundamentales para ahorrar tiempo descubriendo errores, a la par que permiten entender mejor cómo funciona el sistema, probar su resistencia a los cambios y medir varios parámetros de interés.

Por ello, resulta imprescindible diseñar una serie de pruebas que nos permitan probar cómodamente el sistema. Contar con ellas llevará a poder comenzar a desplegar el sistema con seguridad mediante sistemas de integración continua y comprobar compatibilidad entre diferentes versiones.

Por otra parte, otra idea interesante sería adoptar la metodología TDD(Test driven development) para desarrollar ciertas partes del sistema: al ser este desarrollo un framework, esta metodología se adapta muy bien a nuestras necesidades.

6.4. Mejoras en la capa “deployment”

Como hemos visto a lo largo del trabajo, la capa “deployment” nos permite exponer las funcionalidades del sistema mediante reflexión (get_operations, do_operation) e instrospección (serializador).

Esta función facilita en gran medida integraciones con otras plataformas software, como se explicaba durante el capítulo 5. Sin embargo, lejos de ser perfecta, presentamos las siguientes secciones como líneas de futuro de la capa “deployment”:

6.4.1. Mejoras en “helpers”

Al igual que en la capa “core”, hay una serie de excepciones que no se tratan como es debido en los “helpers”. Por otra parte, en cuanto a funcionalidad, debemos fijarnos en los siguientes

puntos:

- La clase Publisher:
  - Ahora mismo, este publicador se suscribe a todos las instancias de "Node" del sistema. Esto puede dar lugar a excesos de información y a problemas de eficiencia.
  - Por otra parte, a pesar de que el sistema lanza muchos más tipos de evento, esta limitada versión de Publisher sólo se suscribe a los cambios de nodos. Será interesante, a futuro, poder elegir a qué tipos de eventos y de qué instancias queremos suscribirnos
- La clase RequestProxy:
  - El proxy debería informar a los usuarios de qué clases están disponibles para ser utilizadas (en este punto del proyecto, tan sólo es posible trabajar con los managers).
  - Siguiendo la idea del punto anterior, quizá sea interesante añadir la posibilidad de trabajar con muchos más tipos de objetos.
  - El descriptor de operaciones que se forma cuando llega la petición indica qué operaciones hay y que argumentos necesitan para ser invocadas. No indica el tipo que ha de tener el valor que se dé al argumento, lo cual puede causar confusión si no se cuenta con una copia de la documentación a mano.
  - Siguiendo la última idea del punto anterior, otra opción interesante a considerar sería el de poder consultar la documentación en línea, teniendo la opción de obtener las descripciones, tipos de dato de los argumentos, y tipo de dato del valor de retorno. En caso de ser tipo diccionario el objeto, sería muy interesante poder consultar qué claves y que tipo de valores se esperan para esas claves.

6.4.2. Proporcionar APIs para diversos lenguajes

Como vimos durante el desarrollo del capítulo 5, contamos con un diseño que nos permite integrarnos muy rápidamente con tecnologías que pongan nuestras funcionalidades y objetos serializados muy rápidamente en otros software.

Aunque esto nos da gran agilidad a la hora de poder disponer de las bondades de IoT-Gate en remoto, la naturaleza reflexiva de esta capa hace que los clientes/suscriptores a nuestras interfaces remotas tengan mucho trabajo de integración.

Por ello, una de las líneas futuras consistirá en trabajar en un conjunto coherente de APIs en diferentes lenguajes, que hagan uso de las características que ofrecen AMQP, ZeroMQ y similares. Estas APIs deberían trasladar el modelo de dominio de sistema a remoto, y como se comenta en el título, existirán implementaciones de éstas en varios lenguajes: Python, Java y Node.js (javascript), en un primer instante.

Otra idea interesante a considerar es la de integrar estas soluciones con el framework, como

si de una herramienta más se tratara, y poder disponer a golpe de configuración de una API REST (request-response) con WebSockets(publish-subcribe) implementada en Flask, de un servidor/publicador en AMQP, etc.

Capítulo 7

Conclusiones

7.1. Introducción

Hállome finalizando mi posible último trabajo en calidad de estudiante del Grado en Ingeniería Informática (mención en Ingeniería del Software) -permítanme tomármelo con algo de humor-, y no son pocas las conclusiones que éste me evoca. En un intento por ser sintético y directo, voy a organizar las conclusiones que he alcanzado de la siguiente manera:

- Comenzaré hablando sobre la temática del trabajo, "internet de las cosas", y de lo que pienso que he aprendido acerca del concepto - y de lo que me queda.
- Después, hablaré de los objetivos que he alcanzado con el desarrollo del proyecto.
- Continuaré hablando sobre las conclusiones a las que he llegado desde la perspectiva de estudiante; de qué me ha servido lo que he estudiado para desarrollar este trabajo, y qué cosas he conseguido comenzar a entender.
- Por último, hablaré en calidad de desarrollador de software novicio, qué conclusiones saco después de haberme enfrentado a este trabajo; que resulta ser la culminación -y el comienzo- de muchos otros -pasados y futuros.

7.2. Acerca de internet de las cosas

Durante el desarrollo de este trabajo, he tenido la oportunidad de profundizar mucho en el tema. Como denota el trabajo hecho, he intentando documentarme muy bien, y he descubierto un sinfín de posibilidades.

Internet de las Cosas es un campo muy amplio, y ha venido para quedarse. Tiene aplicaciones en prácticamente cualquier campo, y es muy posible que sea esencial en el futuro,

para hacer mejor la vida de las personas. Más información, más conocimiento, mejores decisiones. Eso es Internet de las Cosas. Sin embargo, también plantea, en mi opinión, grandes interrogantes:

- ¿Cómo avanzará este mundo hiper-conectado en términos de intimidad y privacidad?
- ¿Va a servir para hacer mejor la vida de las personas, o para tenerla mejor medida?
- ¿Va a contribuir a un desarrollo más sostenible, más cuidado con el medio ambiente, o va a desembocar en la fabricación de miles de millones de dispositivos, que habrá que reemplazar cada poco tiempo por temas de obsolescencia, creando así un montón de basura tecnológica?
- ¿Va a estar al servicio de sus clientes, o de sus fabricantes?

Estos últimos puntos son reflexiones que quizá inviten al lector a ser algo pesimista. Por otra parte, es un campo en el que nos vamos a encontrar profesionales de muchas áreas: ingenieros industriales, electrónicos, de telecomunicaciones, informáticos, matemáticos, estadísticos, agrarios, médicos...; supone un reto de futuro muy interesante.

Por otra parte, desde una perspectiva más teórica, en "IoT", es fácil encontrarse trabajando, entre otros, con elementos de las siguientes áreas:

- M2M, machine to machine, informática industrial.
- Computación Ubicua.
- Sistemas operativos.
- Sistemas distribuidos.
- Inteligencia artificial, ingeniería del conocimiento, técnicas de aprendizaje automático y minería de datos.
- Un grandísimo etcétera.

Todo esto descrito, desde mi aún reducido punto de vista. Aún me queda, manteniendo el alcance de mi curiosidad a temas que tengan que ver con este trabajo -IoTGate- los siguientes puntos:

- Conocer e implementar muchos protocolos más.
- Entender mejor la integración que hago con el hardware, ¿cómo funciona GNU/Linux en ese sentido? ¿cómo podría hacer mejor aprovechamiento de él?
- Aprender mucha más programación.

7.3. Objetivos alcanzados

Con respecto a los objetivos alcanzados mediante la solución desarrollada a lo largo de este trabajo.

La motivación de este trabajo viene dada por la experiencia de haber desarrollado una cantidad considerable de soluciones ad-hoc a los problemas:

- Alcanzar la estructura, fijar los conceptos, y decidir el cómo está completamente basado en la experiencia.
- Ahora, empezamos a hacer uso de este nuevo desarrollo. Como veíamos en la sección de "Líneas futuras", hay unos cuantos interrogantes, y mucho por hacer. El uso que hagamos irá definiendo si hemos acertado o si no. En cualquier caso, no podemos hablar en términos absolutos: habrá cosas que estén bien y otras que no. El primer paso es equivocarse, el segundo es aprender, y el tercero es continuar. Lo malo sería caer siempre en los mismos errores, y no aprender.

Por otra parte se han alcanzado los siguientes objetivos:

- Hemos conseguido proporcionar una comunicación uniforme a través de diversas tecnologías de comunicación, a través de los elementos de communicators:
  - Se pone a disposición del desarrollador las herramientas básicas que hacen falta para hacer uso de la GPIO de la Raspberry Pi.
  - Se presenta una manera sencilla de implementar drivers, con un lenguaje de alto nivel y una sintaxis más amable que la que habitualmente hay que manejar cuando nos enfrentamos a problemas de este estilo.
  - Aparte de ser ampliable, es perfectamente utilizable: enchufas tu hardware, instalas el framework, escribes cuatro líneas de código y ya estás trabajando.
- Hemos conseguido separar el protocolo del medio de comunicación mediante la distinción entre adaptors y communicators. Esto facilita la comprensión sobre el funcionamiento del dispositivo integrado.
- Proporcionamos una serie de reglas, un orden, y unos conceptos (Grupos, nodos, sensores, actuadores, etc.)
- Proporcionamos un servicio de persistencia y otro de "logging".
- Proporcionamos herramientas que facilitan la integración con otros programas, dando soporte a los modelos de comunicación básicos.

Por otra parte, quedan pendientes, como se planteaba en líneas futuras:

1. Un mejor diseño de la capa communicators.

2. Más APIs distribuidas: dejamos demasiado trabajo para los clientes.
3. Instalación sencilla de las ampliaciones en communicators y adaptors.

Como veremos a continuación, he aprendido mucho de este proyecto. Y en gran medida, y con mucho futuro por delante, hemos alcanzado la mayor parte de los objetivos de este proyecto.

7.4. Desde la perspectiva de estudiante

En esta sección, mi pretensión es poner en valor las diferentes asignaturas que de una forma u otra, han influenciado con determinación el resultado final de este trabajo:

- Matemática Discreta: tardé en aprobarla mucho, pero fue revelador entenderla mínimamente después de haber visto Estructura de datos y Algoritmos, Análisis de algoritmos, Bases de datos, Ingeniería del Software, Orientación a Objetos, Criptografía... Muy reveladora. Si al desarrollar el diseño de este proyecto, no hubiera sido capaz de razonar en términos de conjuntos, operaciones con los mismos, relaciones entre elementos, grafos, redes... Todo en informática tiene demasiado que ver con las matemáticas. Y bueno, todo en general
- Sistemas digitales: me hablaron de cosas como "activa en baja", "se activa en flanco de bajada", los efectos aleatorios, la señal del reloj, síncrono... es lejano ya, pero me ha ayudado a entender muchas de las cosas que estaban escritas en los "datasheets" de los dispositivos que he integrado; cuando he tenido que trabajar con las entradas/salidas digitales, etc.
- Física: al igual que matemáticas, la he tenido en mi repertorio de asignaturas hasta el final. De hecho, es la última asignatura que he aprobado. Pero claro... el día que mi Raspberry tenía demasiadas cosas enchufadas y se arregló cambiando el adaptador de corriente por uno que tuviera mayor amperaje, algo tuvo que ver la física. Los sensores y actuadores con los que trabajo, las fórmulas que utilizamos para entender los datos que nos proporcionan los sensores... todo tiene algo que ver con la física.
- Fundamentos de computadoras: en esta asignatura empecé a entender bien qué era bajo nivel y que era alto nivel. Vimos ensamblador, lo que en cierto modo me ha ayudado a entender la misión de los microcontroladores, y esto desemboca en el entendimiento del término "tiempo real"
- Fundamentos de redes de computadoras: un trabajo que va sobre internet y redes, tiene demasiado que ver con una asignatura como esta. Aquí aprendí la pila de protocolos, el modelo OSI, que era un IP, que era un puerto, que era una trama, que era un datagrama, que era un mensaje, que es TCP, en que se diferencia de UDP.. indispensable.
- Fundamentos de programación: la suspendí un par de veces, hasta que un día empecé a entender. Y desde entonces, me ha gustado tanto que he necesitado quedarme a escuchar el resto de la carrera.

- Paradigmas de programación: en esta asignatura aprendí a programar. Del buen recuerdo, me volví a animar a utilizar Python, el lenguaje con el que nos enseñaron durante esta asignatura. Y así ha quedado el proyecto.
- Estructuras de datos y algoritmos: "¿y aquí que pongo, un mapa o una lista?" "voy a poner aquí búsqueda lineal, no vamos a tener nunca más de 10 elementos", "ahí mejor pon una lista enlazada, que el acceso va a ser siempre secuencial". Imprescindible.
- Fundamentos de sistemas operativos: me hablaron de secciones críticas. De recursos que no podían compartirse. De hilos, que compartían datos mediante sus variables globales. De semáforos; binarios y naturales. De las cosas que sucedían "a la vez".
- Programación orientada a objetos: quizá la orientación objetos sea una de las herramientas más impresionantes con la que contamos los programadores. Sino hubiera entendido que es la herencia, lo que es el polimorfismo, lo que es una clase abstracta, lo que es una interfaz, lo que significa sobrescribir un método, que es la visibilidad de una atributo, que es el estado de un objeto, que es eso de que "se envían mensajes".
- Estructura de sistemas operativos: después de esta asignatura, no volví a equivocarme poniendo un semáforo. Después de entenderla, no volví a sentir de la misma manera a los sistemas operativos. Cambios de contexto, paginación, sistemas de ficheros, virtualización... Consiguió cambiar mi manera de ver a los aparatos. Y también influenció positivamente en mi manera de hacer los programas. Todo un logro, otra imprescindible.
- Sistemas distribuidos: en esta asignatura escuché hablar de socket, de operaciones "bloqueantes", de accepts que lanzaban hilos para atender peticiones... escuché hablar de CORBA. Escuché hablar de RPC.
- Fundamentos de Ingeniería del software: mi primera aproximación a los requisitos, casos de uso, diagramas UML... salí pensando que "todo eso no servía para nada". Ahora pienso que estaba muy equivocado.
- Análisis y diseño de bases de datos: imprescindible, para prácticamente cualquier desarrollo. Diagramas de entidad-relacional, el paso a relacional, las restricciones de integridad, consultas y SQL... No he parado de aplicarla desde que la aprendí.
- Diseño de software: observador, compuesto, experto, factorías, controlador, MVC, estrategia, estado, comando, proxy, responsabilidades, capas... una vez más, imprescindible. Cuantas herramientas, cuantas posibilidades.
- Lenguajes de programación: y así di un paso de gigante como programador. Aprendiendo a programar un lenguaje de programación. Programando el C más serio que he hecho hasta la fecha. Ya no volveré a poner un puntero del revés, vaya. Ahora veo ámbitos, alcances, espacios de nombres,... Completamente imprescindible.
- Desarrollo basado en componentes y servicios: definid buenas interfaces. Interfaces remotas. APIs. Despliegue. Términos que me acompañarán toda la vida.
- Sistemas Empotrados y de Tiempo Real: qué decir sobre esta asignatura. Al final empecé a aprender que era Linux. Que era un demonio. Que era un módulo. Como

podía compilar ciertas cosas. Por qué GNU/Linux no era un sistema operativo de tiempo real. Fue genial, muy reveladora.

Y me dejo muchas en el tintero, pero aquí solo debo poner las que algo tienen que ver con el trabajo. Como decía en los agradecimientos, muchas gracias a todos. Concluyo: a pesar de las dificultades, de la pereza, de los conflictos, y de otra serie de cosas, he disfrutado bastante la carrera. Gracias de nuevo.

7.5. Desde la perspectiva de desarrollador de software

Este proyecto es, hasta la fecha, el más ambicioso que he desarrollado. Estando, por una parte, trabajando en la universidad, de una manera más académica; y por otra parte, en la empresa -Argotec-, desarrollando un producto -con sus clientes, sus plazos, los compañeros...- he llegado a un montón de conclusiones; en este caso, como desarrollador de software recién llegado al mundo:

- **Las pequeñas decisiones importan**: cosas como qué nombres poner a las variables, en qué punto llamo a qué métodos y que parámetros reciben y devuelve estos, que estructura de datos utilizo para implementar tal cosa...
- **Reconocer que lo que parecen grandes decisiones, en realidad, no lo son tanto**: contamos con muchas herramientas para hacer las cosas. Las decisiones sobre qué tipo de herramientas, incluyendo aquí elementos como el lenguaje de programación y sus posibilidades, no son tan importantes en un principio. Todo descansa sobre las mismas bases; lo importante es entender las bases.
- **Saber siempre que utilizar**: poco a poco, la vida va sucediendo, y los problemas se repiten. Al estar frente a un viejo problema, uno empieza a tener en mente un gran catálogo de posibilidades. Un buen programador siempre conoce dos o tres alternativas a la soluciones, y decide rápidamente cual es la mejor.Esto es experiencia.
- **Entender que el código es barato**: el trabajo es lo que importa. El razonamiento, la conceptualización, el orden, las decisiones... Ese es el verdadero valor. El código hay que cambiarlo, borrarlo, escribirlo, usarlo; pero es más una gran herramienta. Vislumbrar el futuro es complicado, llegar a él es inevitable.
- **Saber decir "de esto, no sé, voy a ver"**: no hay mejor manera de perder el tiempo desarrollando que "liándose" con un problema que ya está solucionado.
- **Aprender bajo presión**: el plazo está ahí. Y siempre hay mucho que aprender, y poco tiempo.
- **Ser responsable de mis errores**: en muchas ocasiones, es fácil ver a un programador novato echando la culpa de sus cosas a la "librería esta", o a el "maldito Java". Cuidado con la hostilidad, es mejor abrir la mente, asumir errores y aprender de ellos. Detrás de todo el software hay personas, y las personas nos equivocamos. Nada es perfecto.

- **Saber identificar los verdaderos problemas**: priorizar y estimar es una tarea verdaderamente compleja. Nuestra perspectiva informática nos juega malas pasadas en muchas ocasiones. Hay que intentar "descontaminarse" de vez en cuando y entender desde otra perspectiva cuáles son los verdaderos problemas.
- **Programar con gente que sabe**: creo que no he descubierto mejor manera de aprender a programar que juntarme a programar con gente que de verdad sabe.
- **Leer mucho código**: vivimos, por fortuna, en un mundo en el que contamos con un montón de soluciones y programas de código abierto (e incluso libre). Hay que ser curiosos, hay que leer cómo han resuelto esto otras personas. Hay mucho código ahí fuera.
- **Ser capaz de escribir código malo**: muchas veces, uno siente haberse perdido en detalles, intentando que todo quede lo mejor posible. A veces hay que ser práctico y hacer cosas feas, pero no pasa nada. Los plazos son en muchas ocasiones mucho más importantes que una calidad que no se va a apreciar. Ya habrá tiempo de corregir esos métodos feos.
- **Identificar errores a la velocidad de la luz**: otra cosa que voy notando es la capacidad genial que va forjando la experiencia. Ver un mensaje de error, detectar que es, arreglarlo, que aparezca otro y pensar "bien, voy avanzando".
- **Tener perspectiva de negocio**: en los negocios, el desarrollo de software no es la única tarea. Entender el trabajo desde la perspectiva económica y de equipo es fundamental.
- **Ser positivo y apasionado**: poco más que decir. Sin pasión, esto pierde mucha gracia. Sin ser positivo, se hace tedioso y complicado.

Estos son algunas de las conclusiones y actitudes fundamentales que he ido adquiriendo a lo largo de la realización de este trabajo. A ver qué nos depara el futuro. [75]

Apéndice A

Manual de instalación

A.1. Instalación de IoTGate

Para instalar IoTGate, necesitamos:

1. Un sistema operativo con un intérprete de Python 3.4.
2. Conexión a internet (para descargar las dependencias).
3. Crear un fichero de configuración.
4. Permisos de administrador, a no ser que se esté instalando en un "virtualenv": Documentación sobre virtualenv
5. Si la distribución de IoTGate a instalar cuenta con "communicators" que dependen de hardware específico, necesitaremos configurar previamente ese hardware.

Para configurar el hardware en una Raspberry, se recomienda leer el siguiente artículo: https://www.raspberrypi.org/documentation/configuration/raspi-config.md.

Una vez tengamos nuestro entorno preparado, nos situamos en el directorio donde esté situado el fichero "setup.py". En el CD-ROM, podemos encontrarlo bajo la carpeta "src". Después, tan solo hay que ejecutar el siguiente comando:

```
1 python3 setup.py install
```

Figura A.1: Comando de instalación de IoTGate

Posibles errores en la instalación:

1. El nombre del comando “python3” puede variar según la configuración de variables de entorno del sistema.

2. Aunque se instala por defecto en la mayor parte de las distribuciones del intérprete de Python, es posible que te falten “pip” y “setuptools”. En ese caso, recomiendo leer este artículo: https://packaging.python.org/installing/

3. Otro error posible es la ejecución del script “setup.py” con el intérprete de Python 2. IoTGate es incompatible con Python2. La versión en la que ha sido probado IoTGate es Python 3.4.2.

Apéndice B

Contenidos del CDROM

El CD-ROM incluye:

1. Un fichero llamado memoria.pdf, con el presente documento.
2. Una carpeta "src", que incluye el código fuente del proyecto:
   - Dentro de src, encontraremos un fichero "setup.py". Es el instalador del framework.
   - Una carpeta "iotgate", que contiene el código fuente del proyecto:
     - La carpeta "etc" incluye un fichero de ejemplo del formato que ha de tener la configuración.
     - La carpeta "adaptors" incluye la capa "adaptors". Esta capa es dinámica, si se quiere introducir un nuevo adaptor, hay que copiarlo aquí.
     - La carpeta "communicators", incluye la capa "communicators". Al igual que la anterior, es dinámica. Si se quiere incluir nuevos elementos, hay que copiarlos aquí.
   - El fichero de ejemplo "start$_i otgate.py''$, $queindicacomoseinstanciael framework$.

Bibliografía

Libros

[1] *Internet of things: A Hands-On Approach*,
Arshdeep Bahga y Vijay Madisetti
Editorial: Publicado por los propios autores
ISBN: 978-0996025515

[2] *UML 2*,
Jim Arlow E Ila Neustadt.
Editorial: Anaya
ISBN: 84-415-2033-X

[3] *Ingeniería del software, 7ª Edición*,
Ian Somerville
Editorial: Pearson
ISBN: 84-7829-074-5

[4] *Patrones de diseño*,
E. Gamma, R. Helm, R. Johnson y J. Vlisides
Editorial: Addison Wesley
ISBN: 84-7829-059-1

[5] *Ubiquitous Computing*,
Stefan Poslad
Editorial:
ISBN: 978-0-470-03560-3

[6] *Redes de computadoras: un enfoque descendente*,
James F. Kurose y Keith W. Ross
Editorial: Addison Wesley
ISBN: 978-84-7829-119-9

[7] *Learning Python Design Patterns*,
GENNADIY ZLOBIN
Editorial: Packt Publishing
ISBN: 978-1-78328-337-8

Referencias web

- Capítulo 1: Introducción -

[8] *Acerca de Stash*
ATLASSIAN.COM
https://es.atlassian.com/software/bitbucket/server
Visitado por última vez: 12 de julio de 2016

[9] *Acerca de Jira*
ATLASSIAN.COM
https://es.atlassian.com/software/jira
Visitado por última vez: 12 de julio de 2016

[10] *Acerca de Pycharm*
JETBRAINS.COM
https://www.jetbrains.com/pycharm/
Visitado por última vez: 12 de julio de 2016

[11] *Acerca de LaTeX*
LATEX-PROJECT.ORG
https://latex-project.org/intro.html
Visitado por última vez: 12 de julio de 2016

[12] *Kile - an Integrated LaTeX Environment*
SOURCEFORGE.NET
http://kile.sourceforge.net/
Visitado por última vez: 12 de julio de 2016

[13] *Astah Professional*
ASTAH.NET
http://astah.net/editions/professional
Visitado por última vez: 12 de julio de 2016

[14] *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*
RASPBERRYPI.ORG
https://www.raspberrypi.org/help/what-is-a-raspberry-pi/
Visitado por última vez: 12 de julio de 2016

[15] *MCP4728 - Mixed Signal- System D/A Converters - Microchip*
MICROCHIP.COM
http://www.microchip.com/wwwproducts/en/en541737
Visitado por última vez: 12 de julio de 2016

[16] *Netbows – Small circuits for big things*
NETBOWS.COM
https://netbows.com/
Visitado por última vez: 12 de julio de 2016

[17] *GIMP - GNU Image Manipulation Program*
GIMP.ORG
https://www.gimp.org/
Visitado por última vez: 12 de julio de 2016

[18] *Inkscape: Draw Freely*
INKSCAPE.ORG
https://inkscape.org/en/
Visitado por última vez: 12 de julio de 2016

[19] *Asociación Española de domótica e inmótica*
CEDOM.ES
http://www.cedom.es/sobre-domotica/que-es-domotica
Visitado por última vez: 12 de julio de 2016

[20] *El futuro de la agricultura: la agricultura inteligente*
AGRICULTURERS.COM
http://agriculturers.com/el-futuro-de-la-agricultura-la-agricultura-inteligente/
Visitado por última vez: 12 de julio de 2016

[21] *Telefónica colabora con Cattle-Watch para conectar la industria ganadera al IoT*
SMART-LIGHTING.ES
http://smart-lighting.es/telefonica-colabora-con-cattle-watch-para-conectar-la-industria-ganadera-al-iot/
Visitado por última vez: 12 de julio de 2016

[22] *¿Cómo será la gestión del tráfico en las Smart Cities?*
INSTITUTOTED.COM
http://www.institutoted.com/noticia/47-smart-cities/113-gestion-trafico-smart-cities
Visitado por última vez: 12 de julio de 2016

[23] *Leveraging the Internet of Things and Analytics for Smart Energy Management*
TCS.COM
http://www.tcs.com/SiteCollectionDocuments/White-Papers/BPS-Internet-of-Things-Smart-Energy-Management-1015-1.pdf
Visitado por última vez: 12 de julio de 2016

[24] *Internet de las cosas y medio ambiente*
LOSTIEMPOS.COM
http://www.lostiempos.com/tendencias/tecnologia/20160517/internet-cosas-medio-ambiente
Visitado por última vez: 12 de julio de 2016

[25] *How Big Data And The Internet Of Things Improve Public Transport In London*
FORBES.COM

http://www.forbes.com/sites/bernardmarr/2015/05/27/how-big-data-and-the-internet-of-things-improve-public-transport-in-london/#40973d253ab3
Visitado por última vez: 12 de julio de 2016

[26] *El coche conectado: el Internet de las Cosas pisa el acelerador*
MOBILEWORLDCENTRE.COM
https://www.mobileworldcentre.com/es/-/el-coche-conectado-el-internet-de-las-cosas-pisa-el-acelerador
Visitado por última vez: 12 de julio de 2016

[27] *Salud conectada, el IoT como tu mejor salvavidas*
IOT.TELEFONICA.COM
https://iot.telefonica.com/blog/salud-conectada-el-iot-como-tu-mejor-salvavidas
Visitado por última vez: 12 de julio de 2016

- Capítulo 2: Estado del arte. Contexto tecnológico -

[28] *IPv4 Officially Depleted, Eyes on IPv6*
ENTERPRISENETWORKINGPLANET.COM
http://www.enterprisenetworkingplanet.com/news/article.php/3923391/IPv4-Officially-Depleted-Eyes-on-IPv6.htm
Visitado por última vez: 12 de julio de 2016

[29] *Bluetooth LE Basics*
BLUETOOTH.COM
https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy
Visitado por última vez: 12 de julio de 2016

[30] *LoRa Technology*
LORA-ALLIANCE.ORG
https://www.lora-alliance.org/What-Is-LoRa/Technology
Visitado por última vez: 12 de julio de 2016

[31] *Sigfox - Página oficial*
SIGFOX.COM
https://www.sigfox.com/es
Visitado por última vez: 12 de julio de 2016

[32] *What is enOcean*
ENOCEAN.COM
https://www.enocean.com/en/technology/
Visitado por última vez: 12 de julio de 2016

[33] *¿Qué es KNX?*
KNX.ORG
https://www.knx.org/es/knx/associacion/que-es-knx/index.php
Visitado por última vez: 12 de julio de 2016

[34] *About ZWave*
Z-WAVE.SIGMADESIGNS.COM

http://z-wave.sigmadesigns.com/about$_z$ − *wave*
Visitado por última vez: 12 de julio de 2016

[35] *NB-IOT, Accelerating Cellular IOT*
HUAWEI.COM
http://www.huawei.com/minisite/hwmbbf15/en/nb-iot-accelerating-cellular-iot.html
Visitado por última vez: 12 de julio de 2016

[36] *CoAP — Constrained Application Protocol*
COAP.TECHNOLOGY
http://coap.technology/
Visitado por última vez: 12 de julio de 2016

[37] *About websocket*
WEBSOCKET.ORG
https://www.websocket.org/aboutwebsocket.html
Visitado por última vez: 12 de julio de 2016

[38] *AMQP Overview*
AMQP.ORG
https://www.amqp.org/product/overview
Visitado por última vez: 12 de julio de 2016

[39] *MQTT FAQ*
MQTT.ORG
http://mqtt.org/faq
Visitado por última vez: 12 de julio de 2016

[40] *ZeroMQ: Distributed messaging*
ZEROMQ.ORG
http://zeromq.org/
Visitado por última vez: 12 de julio de 2016

[41] *Eclipse Kura*
IOT.ECLIPSE.ORG
http://iot.eclipse.org/
Visitado por última vez: 14 de julio de 2016

[42] *Eclipse Kura*
ECLIPSE.ORG
https://www.eclipse.org/kura/
Visitado por última vez: 14 de julio de 2016

[43] *Linux Foundation IoTivity*
IOTIVITY.ORG
https://www.iotivity.org/
Visitado por última vez: 14 de julio de 2016

[44] *Ubiworx*
UBIWORX.COM
http://www.ubiworx.com/ubiworx/
Visitado por última vez: 14 de julio de 2016

[45] *Tools for the open source Internet of Things*
IOT-TOOLKIT.COM
http://iot-toolkit.com/
Visitado por última vez: 14 de julio de 2016

[46] *VSCP: Very Simple Control Protocol*
VSCP.ORG
http://www.vscp.org/
Visitado por última vez: 14 de julio de 2016

[47] *Device Hive*
DEVICEHIVE.COM
http://devicehive.com/
Visitado por última vez: 14 de julio de 2016

[48] *Kaa Open-Source IoT Platform*
KAAPROJECT.COM
http://www.kaaproject.org/
Visitado por última vez: 14 de julio de 2016

[49] *ThingSpeak*
THINGSPEAK.COM
https://thingspeak.com/
Visitado por última vez: 14 de julio de 2016

[50] *Carriots*
CARRIOTS.COM
https://www.carriots.com/
Visitado por última vez: 14 de julio de 2016

[51] *Particle*
PARTICLE.IO
https://www.particle.io/prototype
Visitado por última vez: 14 de julio de 2016

[52] *Lhings: how it works*
LHING.COM
http://lhings.com/howitworks.html
Visitado por última vez: 14 de julio de 2016

[53] *IoT Ready Stack, FIWARE*
FIWARE.ORG
https://www.fiware.org/iot-ready-stack/
Visitado por última vez: 14 de julio de 2016

[54] *Thingworx: machine learning and IoT Analytics Platform*
THINGWORX.COM
https://www.thingworx.com/platforms/thingworx-analytics/
Visitado por última vez: 14 de julio de 2016

[55] *Xively: How it works*
XIVELY.COM
https://www.xively.com/xively-iot-platform/how-it-works
Visitado por última vez: 14 de julio de 2016

[56] *SiteWhere Architecture*
SITEWHERE.ORG
http://documentation.sitewhere.org/architecture.html
Visitado por última vez: 14 de julio de 2016

[57] *Blynk*
BLYNK.CC
http://www.blynk.cc/
Visitado por última vez: 14 de julio de 2016

[58] *Zatar Product Showcase*
ZATAR.COM
http://www.zatar.com/product-showcase
Visitado por última vez: 14 de julio de 2016

- Capítulo 3: Análisis -

[59] *Software framework*
WIKIPEDIA.ORG
https://en.wikipedia.org/wiki/Software_framework
Visitado por última vez: 12 de julio de 2016

[60] *MRF24J40MC*
MICROCHIP.COM
http://www.microchip.com/wwwproducts/en/MRF24J40MC
Visitado por última vez: 12 de julio de 2016

[61] *Semtech SX1272*
SEMTECH.COM
http://www.semtech.com/wireless-rf/rf-transceivers/sx1272/
Visitado por última vez: 12 de julio de 2016

[62] *Serial Peripheral Interface Bus*
WIKIPEDIA.ORG
https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
Visitado por última vez: 12 de julio de 2016

[63] *I2C*
WIKIPEDIA.ORG
https://en.wikipedia.org/wiki/I%C2%B2C
Visitado por última vez: 12 de julio de 2016

[64] *Universal asynchronous receiver/transmitter*
WIKIPEDIA.ORG
https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter
Visitado por última vez: 12 de julio de 2016

[65] *RS-232*
WIKIPEDIA.ORG
https://en.wikipedia.org/wiki/RS-232
Visitado por última vez: 12 de julio de 2016

[66] *Entrada/Salida de Propósito General*
WIKIPEDIA.ORG
https://es.wikipedia.org/wiki/GPIO
Visitado por última vez: 12 de julio de 2016

[67] *Radxa Rock(Pro/Lite)*
WIKI.RADXA.COM
http://wiki.radxa.com/Rock
Visitado por última vez: 12 de julio de 2016

[68] *BeagleBone Black*
BEAGLEBOARD.ORG
https://beagleboard.org/black
Visitado por última vez: 12 de julio de 2016

[69] *Modbus*
WIKIPEDIA.ORG
https://en.wikipedia.org/wiki/Modbus
Visitado por última vez: 12 de julio de 2016

- Capítulo 4: Diseño e implementación -

[70] *https://docs.python.org/3.4/library/inspect.html*
PYTHON 3.4.5 DOCUMENTATION
https://docs.python.org/3.4/library/inspect.html
Visitado por última vez: 14 de julio de 2016

[71] *How to use *args and **kwargs in Python*
SALTYCRANE.COM
http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
Visitado por última vez: 14 de julio de 2016

[72] *abc - Abstract Base Classes*
PYTHON 3.4.5 DOCUMENTATION
https://docs.python.org/3.4/library/abc.html
Visitado por última vez: 14 de julio de 2016

[73] *Python Scopes and Namespaces*
PYTHON 3.4.5 DOCUMENTATION
https://docs.python.org/3/tutorial/classes.html
Visitado por última vez: 14 de julio de 2016

[74] *SQLAlchemy: the Python SQL Toolkit and Object Relational Mapper*
SQLALCHEMY.ORG
http://www.sqlalchemy.org/
Visitado por última vez: 14 de julio de 2016

- Capítulo 5: Instanciación y extensibilidad -

[75] *Common string operations. Format examples*
PYTHON 3.4.5 DOCUMENTATION
https://docs.python.org/3.4/library/string.html#format-examples
Visitado por última vez: 14 de julio de 2016

[76] *Flask: a Python microframework*
FLASK.POCOO.ORG
http://flask.pocoo.org/
Visitado por última vez: 14 de julio de 2016

[77] *Flask-SocketIO*
FLASK-SOCKETIO.READTHEDOCS.IO
http://flask-socketio.readthedocs.io/en/latest/
Visitado por última vez: 14 de julio de 2016

[78] *RabbitMQ Getting Started*
FLASK.POCOO.ORG
https://www.rabbitmq.com/getstarted.html
Visitado por última vez: 14 de julio de 2016

[79] *Introduction to Pika*
FLASK.POCOO.ORG
https://pika.readthedocs.io/en/0.10.0/
Visitado por última vez: 14 de julio de 2016

[80] *RPi.GPIO 0.6.2*
PYPI.PYTHON.ORG
https://pypi.python.org/pypi/RPi.GPIO
Visitado por última vez: 14 de julio de 2016

[81] *spidev 3.2*
PYPI.PYTHON.ORG
https://pypi.python.org/pypi/spidev
Visitado por última vez: 14 de julio de 2016

[82] *Adafruit I2C*
GITHUB.COM
https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code/tree/legacy/Adafruit_I2C
Visitado por última vez: 14 de julio de 2016

- Capítulo 6: Líneas Futuras -

[83] *The M-Bus: A Documentation Rev. 4.8*
M-BUS.COM
http://www.m-bus.com/mbusdoc/default.php
Visitado por última vez: 12 de julio de 2016