Universidad Autónoma de Madrid

Escuela Politécnica Superior

24|25

Trabajo fin de grado

Central domótica de gestión de excedentes fotovoltaicos

Jose Ramón Morales León

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\ Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Central domótica de gestión de excedentes fotovoltaicos

Autor: Jose Ramón Morales León
Tutor: José Luis García Dorado

noviembre 2024

Jose Ramón Morales León
Central domótica de gestión de excedentes fotovoltaicos

Jose Ramón Morales León

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Cuando un usuario de una pequeña instalación fotovoltaica quiere aprovechar los excedentes que produce porque no le interesa inyectar en ese momento energía a la red puede hacer uso de esa energía encendiendo manualmente algunos dispositivos de su hogar, otros optan ir un paso más allá gracias a la domótica, comprando dispositivos que pueden regularse automáticamente como puede ser una estación de recarga de vehículo eléctrico o un enchufe inteligente. El problema que se encuentra este usuario es que los dispositivos que ha adquirido no se comunican entre ellos y en lugar de organizarse para aprovechar la energía compiten por ella.

Este trabajo propone dar solución tanto a los usuarios que quieran empezar a gestionar sus excedentes tanto como a los que ya lo han intentado mediante la domótica pero no lo han conseguido debido a los problemas de coordinación entre dispositivos.

Para ello se desarrolla un *core* que conoce y controla estos dispositivos mientras que distintos módulos o integraciones podrán ser añadidos para ampliar la capacidad de comunicación con dispositivos de distintos fabricantes y con distintos protocolos de comunicación. La conexión con la sistema *Home Assistant* es también parte fundamental del proyecto.

El desarrollo será de código abierto y se publicará en un repositorio de *GitHub* para que cualquier persona pueda colaborar en el desarrollo del proyecto.

Tras el diseño y desarrollo del sistema se realizan las pruebas para comprobar su correcto funcionamiento gestionando la energía y la capacidad de integración con distintos dispositivos y con el ecosistema de *Home Assistant*.

Palabras clave

domótica, internet de las cosas, energía, fotovoltaica, código abierto

Abstract

When a user of the small photovoltaic installation wants to make use of the surplus energy produced because they are not interested in feeding this energy into the grid in that moment they can take advantage of that energy by manually switching on some devices in their home, others choose to go a step further by using home automation buying devices that can do it automatically like a electric vehicle charging station or a smart plug. The problem that this user encounters is that the acquired devices cannot communicate with each other and instead of organizing themselves to use that energy they compete for it.

This thesis propose to provide a solution for both users who have no way to take advantage of their surplus energy and those who have already tried it through home automation but have not succeeded due to the previously mentioned reasons.

To do this, a core will be developed that knows and controls the devices that are included in the system while different modules or integrations can be added to expand the communication capacity with devices from different manufacturers and with different communication protocols. The connection with the *Home Assistant* system is also a fundamental part of the project.

The development will be open source and will be published in a *GitHub* repository so that anyone can collaborate in the development of the project.

After the design and development of the system, tests will be carried out to verify its correct operation managing the energy and the integration capacity with different devices and with the *Home Assistant* ecosystem.

Keywords

home automation, internet of things, energy, photovoltaic, open source

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de figuras

1

Introducción

Cuando un usuario de una pequeña instalación generadora de autoconsumo conectada a red, normalmente fotovoltaica, genera más energía de la que consume se denomina a esa energía extra como excedentes. El usuario tiene varías opciones en cuanto a la gestión de estos excedentes: limitar la producción para no producir de más, inyectar los excedentes a la red a cambio de un precio negociado en el contrato de su comercializadora, almacenarlos en baterías para su posterior uso o aumentar su consumo mientras la producción se lo permita [1]. Es este último caso el que pretende abordar este proyecto.

1.1. Motivación

Las instalaciones fotovoltaicas de autoconsumo están a la orden del día, solo en 2023 se instalaron en torno a 84545 nuevas instalaciones residenciales según un estudio de la Unión Española Fotovoltaica (UNEF) [2]. Los precios altos de la energía y la necesidad de reducir las emisiones de gases de efecto invernadero son dos de los motivos por los que cada vez más personas optan por instalar paneles solares en sus viviendas. Sin embargo, la gestión de los excedentes fotovoltaicos es un problema que no se ha resuelto de forma satisfactoria.

Las razones por las que son un problema para los usuarios la gestión de excedentes fotovoltaicos son varias:

La primera es el precio de las baterías para almacenar los excedentes que es aún elevado por lo que no todos los usuarios optan por instalarlas además que una vez llena la batería no se puede seguir almacenando energía y se vuelve a tener el problema de qué hacer con los excedentes.

La segunda razón es que la inyección de excedentes a la red no siempre es rentable ya que el precio que se paga por la energía inyectada es, en la mayoría de ocasiones, menor que el precio que se paga por la energía consumida, dependiendo del contrato con la comercializadora. Por lo que si en un mismo día tienes que gastar sí o sí cierta cantidad de energía (cargando un coche, calentando el agua o depurando una piscina) es más beneficioso consumir la energía en ese momento que inyectarla a la red y consumir más adelante. Esta gestión de la energía suele hacerse de forma manual estando

pendiente de la producción de tu instalación y de tu consumo, lo que puede ser tedioso y en ocasiones no se hace de la forma más eficiente. Si bien existen sistemas que pretenden solucionar este problema, son solo un parche que soluciona una pequeña parte pero crea otros problemas como veremos más adelante.

Es el crecimiento de las instalaciones fotovoltaicas de autoconsumo y la necesidad de sus usuarios de gestionar de forma eficiente los excedentes fotovoltaicos lo que ha motivado este proyecto.

1.2. Objetivos

El objetivo principal de este trabajo de Fin de Grado es diseñar e implementar un *software* que permita a los usuarios de pequeñas instalaciones fotovoltaicas gestionar sus excedentes de forma automática, basándose en unas reglas dadas por el usuario, los excedentes fotovoltaicos.

Este *software* deberá ser ampliable a través de integraciones con el fin de que sea compatible con el máximo número de sistemas y que otros desarrolladores puedan adaptar el *software* a sus necesidades y dispositivos.

También deberá ser capaz de integrarse con la plataforma domótica de referencia, *Home Assistant* e incluso podrá ser instalado dentro de el sistema operativo de *Home Assistant OS*, ampliando así el alcance del proyecto a un amplio número de usuarios.

Para la implementación se deberá seguir una metodología de desarrollo iterativo permitiendo dividir el proyecto en pequeños hitos donde el primero sea la creación de un producto mínimo viable que cumpla el objetivo principal. De esta forma se simula un entorno de desarrollo ágil propio de la ingeniería de *software*, tal como se imparte en el grado y cercano a como se trabaja en la industria.

Todo el *software* será documentado y de código abierto para que cualquier persona pueda hacer uso de él o contribuir a su desarrollo.

2

Estado del arte

2.1. Situación tecnológica actual

La situación tecnológica actual es la que ha creado la necesidad de este sistema. Aparte de la gestión manual de los consumos, el usuario tiene múltiples opciones para gestionar sus excedentes. Existen medidores de consumo que, en base a ciertos valores de producción, pueden ejecutar automatizaciones, existen relés y enchufes inteligentes que se activan con la domótica (como la solución comercial de Shelly [3]), cargadores de vehículos eléctricos como el *V2C Trydan* son capaces de adaptar la potencia de carga según el consumo/producción de la casa, incluso hay reguladores PWM (*Pulse-width modulation* o modulación por ancho de pulsos) gestionados por un microcontrolador que pueden hacer que un termo eléctrico consuma la energía sobrante, como es el caso del *FreeDS* [4], otro proyecto de código abierto que se asemeja al que se propone en este trabajo pero con distinta filosofía, o la solución de *Ibepower Technologies* [5] otro derivador de energía de comercial, por último puede ser el usuario el que programe las automatizaciones a traves de *software* como *Node-Red* o *Home Assistant* pero dada lo complejidad del problema a resolver los motores de estos sistemas se quedan cortos.

Una vez el usuario decide domotizar la gestión de los excedentes se encuentra con la siguiente situación: tiene configurados distintos dispositivos inteligentes que constan por ejemplo de un cargador de vehículo eléctrico, un regulador de potencia para el termo, un relé inteligente para la depuradora y un enchufe inteligente para un radiador eléctrico. Todos estos dispositivos que se suponen inteligentes son programados, cada con su sistema propio, para encenderse cuando la producción de energía solar supere el consumo. Cuando la radiación solar permite que se de esta situación se da el siguiente caso: los primeros en encenderse son los dispositivos que hacen uso de las resistencias eléctricas, poco más tarde van el resto de dispositivos. En ese sentido los dispositivos están cumpliendo su función pero el problema viene cuando se actualizan los datos de consumo de energía y resulta que, al no haber ningún orden, jerarquía o reparto de esa energía, la vivienda está consumiendo más de lo que produce, acto seguido los dispositivos que son inteligentes se apagan para evitar consumir de más de la misma forma que se encendieron, sin orden ninguno. Tras esto el usuario vuelve a tener exceso de energía repitiéndose el proceso una y otra vez.

2.2. Filosofía

Esta situación que se da en en la gestión de excedentes es parecida a la que se da en la gestión de dispositivos domóticos donde un usuario compra distintos dispositivos de distintos fabricantes y cada uno tiene su propia aplicación para ser controlado. Frente a ello surgieron alternativas de código abierto que buscaban centralizar todos los dispositivos en un mismo lugar y poder programar automatizaciones, como es el caso de *Home Assistant*. *Home Assistant* es un *software* de automatización que prioriza el control local y la privacidad y, aunque es es visto como un *software* para entusiastas del *DIY* (*Do It Yourself* o hazlo tú mismo) ya que requiere cierto conocimiento técnico, sus desarrolladores estiman que hay un millón de instalaciones activas de este *software* [6] de los cuales hay al menos 35,000 usuarios que hacen uso de la integración de energía solar [7] siendo así potenciales usuarios de este proyecto.

La filosofía de diseño de este proyecto no es solo similar a la de *Home Assistant* como concepción sino también como diseño ya que para hacer que el *software* sea compatible con el mayor número de dispositivos y protocolos del mundo *IoT* (*Internet Of Things* o internet de las cosas) se hacen uso de integraciones que extienden la funcionalidad del sistema original.

Tal es la importancia de este *software* que se pretende que lo desarrollado en este proyecto sea integrable como *add-on* (extensión del programa) en *Home Assistant* y sea ese el entorno de ejecución recomendado pese a que también pueda usarse de forma independiente. También se desea que *Home Assistant* funcione como *frontend* para el sistema de gestión de excedentes fotovoltaicos.

Dada la importancia de que el código sea abierto se ha decidido llamar a este proyecto *Open Surplus Manager* y así será referenciado durante todo el documento.

3

Diseño

3.1. Esquema general

Para comprender adecuadamente las partes que componen el sistema se realiza un esquema general donde se muestran de forma visual las relaciones entre los distintos elementos que lo componen.

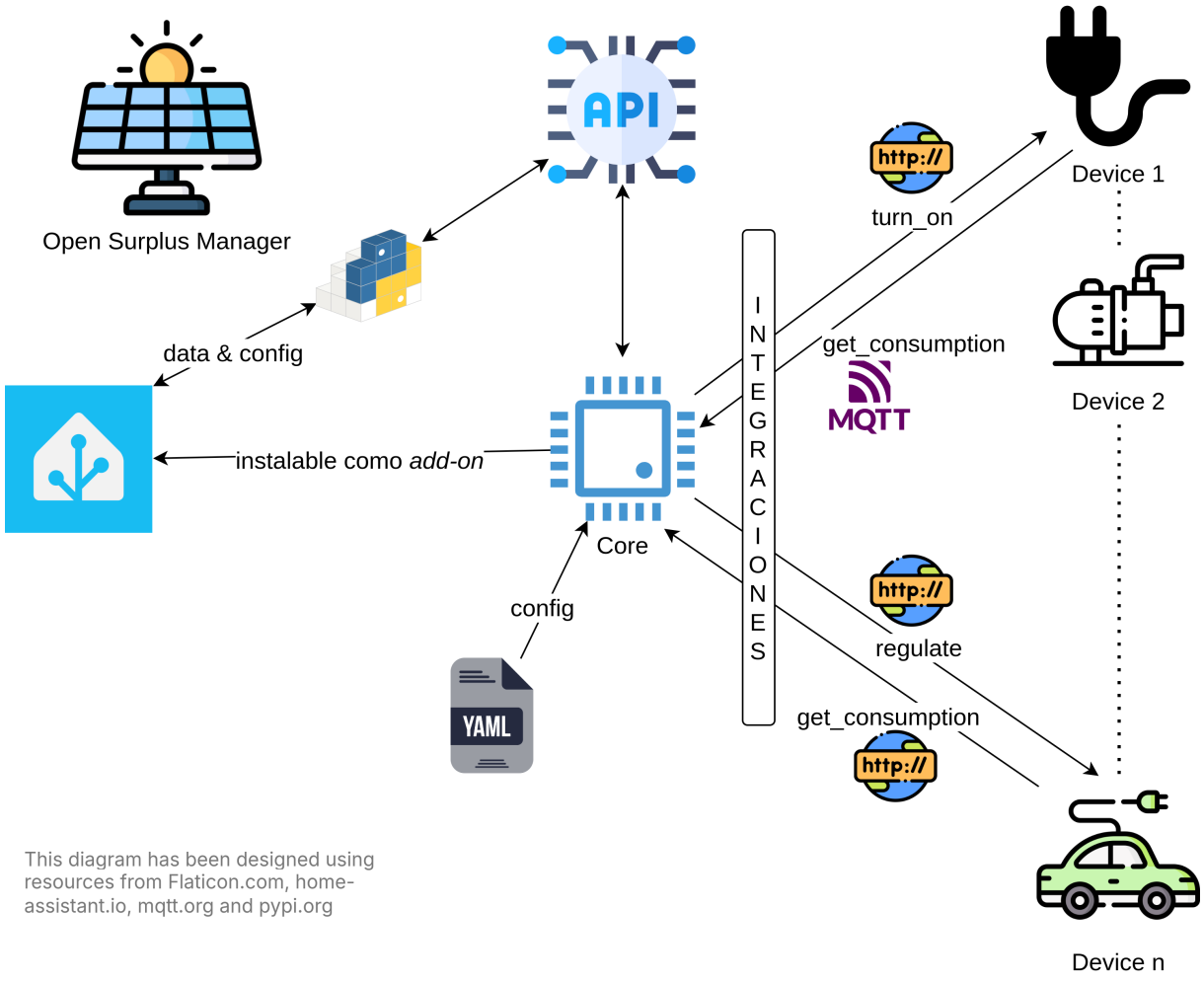

Figura 3.1: Esquema general del sistema

En el centro de la figura 3.1 se encuentra el *core* que es el que realiza todo el trabajo de proce-

samiento de los datos y de control de dispositivos. El *core* también es el encargado de ejecutar las integraciones que le permiten al sistema comunicarse con otros dispositivos. Cada comunicación puede ser mediante una integración completamente distinta, como por ejemplo, mediante métodos *HTTP* (protocolo estándar de la web) o a través de lecturas y escrituras en un *broker MQTT* (programa que permite la comunicación de clientes a través del protocolo de publicación/suscripción *MQTT*), estos ejemplos son independientes al *core* y pueden utilizarse otras tecnologías para la comunicación. La cantidad de dispositivos compatibles puede ampliarse gracias a las integraciones que permiten al usuario o desarrollador añadir nuevas formas de comunicación sin tener que modificar el funcionamiento del *core*, solo respetando unas normas para añadir o desarrollar nuevas integraciones.

Por otra parte los datos pueden ser consultados via *API REST* (interfaz de programación de aplicaciones según la arquitectura REST), que es el estándar de facto para la comunicación entre servicios en la web y además es necesaria para la integración con *Home Assistant*. El sistema permite ser configurado a través de un archivo de configuración *YAML* (tipo de fichero para serialización muy popular en domótica) o a través de *Home Assistant* que funcionaría como interfaz del del sistema.

3.2. Requisitos de *software*

En esta sección se describen los requisitos mínimos de *software* necesarios para que el sistema sea usable en un escenario real.

3.2.1. Requisitos funcionales

RF-1.– El usuario modificará el sistema a través de un fichero de configuración *YAML*.

- **RF-1.1.–** El fichero de configuración puede ser editado por el usuario, se cargará cada vez que se inicie el sistema y será la vía principal de configurar el sistema.
- **RF-1.2.–** El contenido del fichero de configuración será documentado a través de un fichero de ejemplo y explicado en la documentación.
- **RF-1.3.–** La configuración puede ser modificada a través de *Home Assistant*.

RF-2.– Las integraciones se añadirán a una carpeta *integrations* y se cargarán automáticamente al arrancar el sistema.

RF-3.– Las integraciones pueden ser de dos tipos: de lectura de consumo o de control de dispositivos.

- **RF-3.1.–** Las integraciones de consumo pueden ser de un dispositivo o del total de excedentes. En caso de que sea del total de excedentes la integración actualizará su atributo en el *core* y en el caso de que sea de un dispositivo, la integración actualizará el atributo de consumo del dispositivo.
- **RF-3.2.–** Las integraciones de control son llamadas directamente por el *core* cuando el algoritmo de control lo requiera.

RF-4.– El sistema puede ser consultado a través de una *API REST*.

RF-5.– Los datos de del sistema pueden ser consultados través de *Home Assistant*.

RF-6.– El algoritmo de control balanceará la carga entre los dispositivos. El orden de prioridad de los dispositivos se establecerá en el fichero de configuración.

RF-7.– El usuario podrá ampliar la funcionalidad del sistema creando sus propias integraciones.

3.2.2. Requisitos no funcionales

RNF-1.– El sistema se desarrollará en el lenguaje de programación *Python*, el entorno recomendado de ejecución será *Docker* y se proporcionarán esas imágenes a través del repositorio de paquetes *Github Registry*.

RNF-2.– La imagen de Docker también podrá ser ejecutada como *add-on* de *Home Assistant*.

RNF-3.– Se establecerá un sistema de *logging* (registro) para que el usuario pueda ver los errores que se produzcan.

- **RNF-3.1.–** El sistema guardará los errores en un fichero de *log* y los imprimirá por la salida estándar.
- **RNF-3.2.–** El sistema tendrá un nivel de *logging* que se podrá configurar con variables de entorno.

RNF-4.– El sistema se ejecutará en un único hilo pero hará uso de herramientas de concurrencia como *asyncio* para permitir la ejecución de varias integraciones a la vez junto con el algoritmo de control.

RNF-5.– Las integraciones son ejecutadas desde el programa principal a través un método *setup* y si fuera necesario se cerrarán cuando el programa principal lo ordene a través de un método *close*.

RNF-6.– El *software* debe de ser documentado.

- **RNF-6.1.–** Se documentará el código con *docstrings*.
- **RNF-6.2.–** Se creará una wiki en *Github* con la documentación del proyecto.

RNF-7.– El sistema debe de ser *Open Source* (código abierto).

- **RNF-7.1.–** Se almacenará en repositorios de *Github* tanto el programa principal como las herramientas complementarias.
- **RNF-7.2.–** Se creará un sistema de contribuciones a través de *Github*.
- **RNF-7.3.–** Se creará una guía de desarrollo de integraciones.

3.3. Diagrama de clases

El diagrama de clases de la figura 3.1 nos permite tener una visión general de las clases del sistema y sus relaciones. En este diagrama únicamente se representan las clases en si mismas, *Python* permite el uso de funciones y variables fuera de las clases y estas no se representan en este diagrama, como por ejemplo la función *main* o punto de entrada del programa y otras utilidades que se han utilizado en el desarrollo del sistema. Además la dependencia con otros paquetes de terceros tampoco está representada.

Aún así, esta forma de representación da a entender decisiones de diseño como que el *core* es el encargado de gestionar la configuración, crear los dispositivos y gestionarlos. Los modelos de datos se encuentran en el paquete *models* donde se define cómo es un dispositivo o cómo deben ser las integraciones y entidades que debemos añadir a *integrations*.

Forman parte del diagrama un paquete de integración de control y otro de consumo a modo de ejemplo pero todas las integraciones que se quieran añadir para extender el funcionamiento del sistema formarán parte del paquete *integrations* y heredarán de las clases abstractas de *models*.

3.4. Diagramas Secuencia

En los diagramas de secuencia se representan como funciona en conjunto de entidades cuando se realiza cierta acción en un tiempo. En este caso se representarán dos acciones principales, cuando se obtiene un dato de consumo (figura 3.3) y cuando se actualiza el estado de los dispositivos según el dato de excedentes (figura 3.4).

Otras acciones como la carga de la configuración del sistema, la carga de las integraciones o el funcionamiento de la *API REST* no se representan en estos diagramas ya que no son acciones que tengan procesos de interacción entre las distintas entidades del sistema y el diagrama no aportaría información relevante.

3.5. Lenguaje y librerías

El desarrollo de este sistema será en *Python*, un lenguaje de alto nivel que permite desarrollar distintos tipos de aplicaciones de forma rápida y sencilla. *Python* es un lenguaje muy popular en el desarrollo en general pero lo es sobre todo en el mundo *IoT* y en domótica donde *software* importante como *Home Assistant* está desarrollado en este lenguaje. También dispone de una gran cantidad de librerías que permiten la comunicación con una gran cantidad de dispositivos distintos, muy importante para el desarrollo de integraciones de *Open Surplus Manager*.

Para el desarrollo será necesario el uso de herramientas que permitan la programación asíncrona, ya que el sistema deberá ser capaz de gestionar múltiples tareas simultáneamente siendo algunas de ellas bloqueantes. [8]. Algunas de las librerías o *frameworks* (conjunto de código o conceptos que sirve de base para el desarrollo) a utilizar son: *asyncio* para la programación asíncrona, *aiohttp*, tanto para la creación de una *API REST* como para usarlo como cliente, y *aiomqtt* para la obtención de datos *MQTT*. La elección de *MQTT* para el desarrollo de una de las integraciones nativas se debe a que es un protocolo muy popular e idóneo para la comunicación entre dispositivos *IoT*. [9]

Además de *Python* para la programación del sistema se hará de uso de *YAML* para la configuración, *Docker* para la creación de contenedores donde pueda ejecutarse el sistema y *Docker Compose* para su definición y ejecución cuando se usa fuera de *Home Assistant OS*.

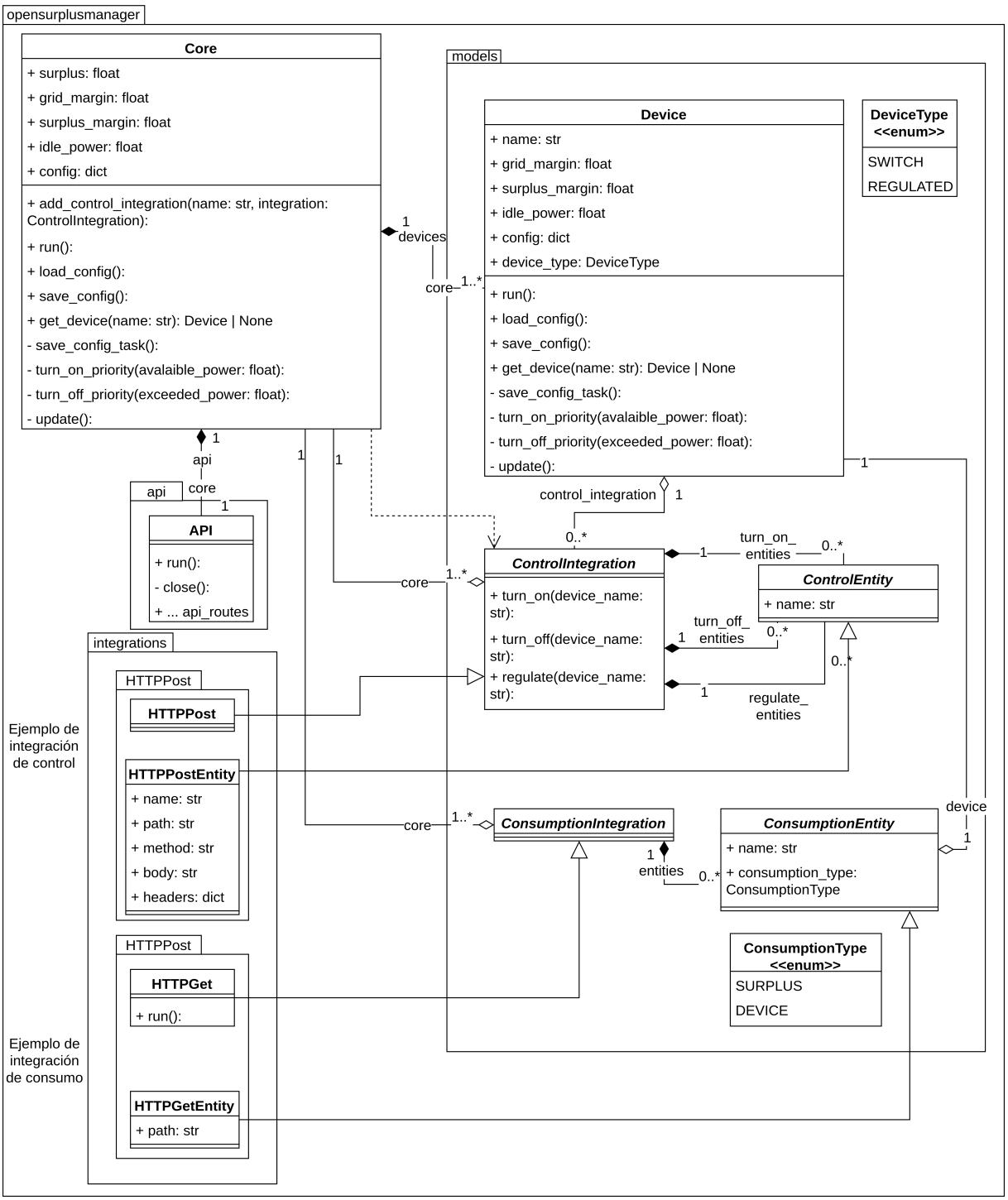

Figura 3.2: Diagrama de clases del sistema

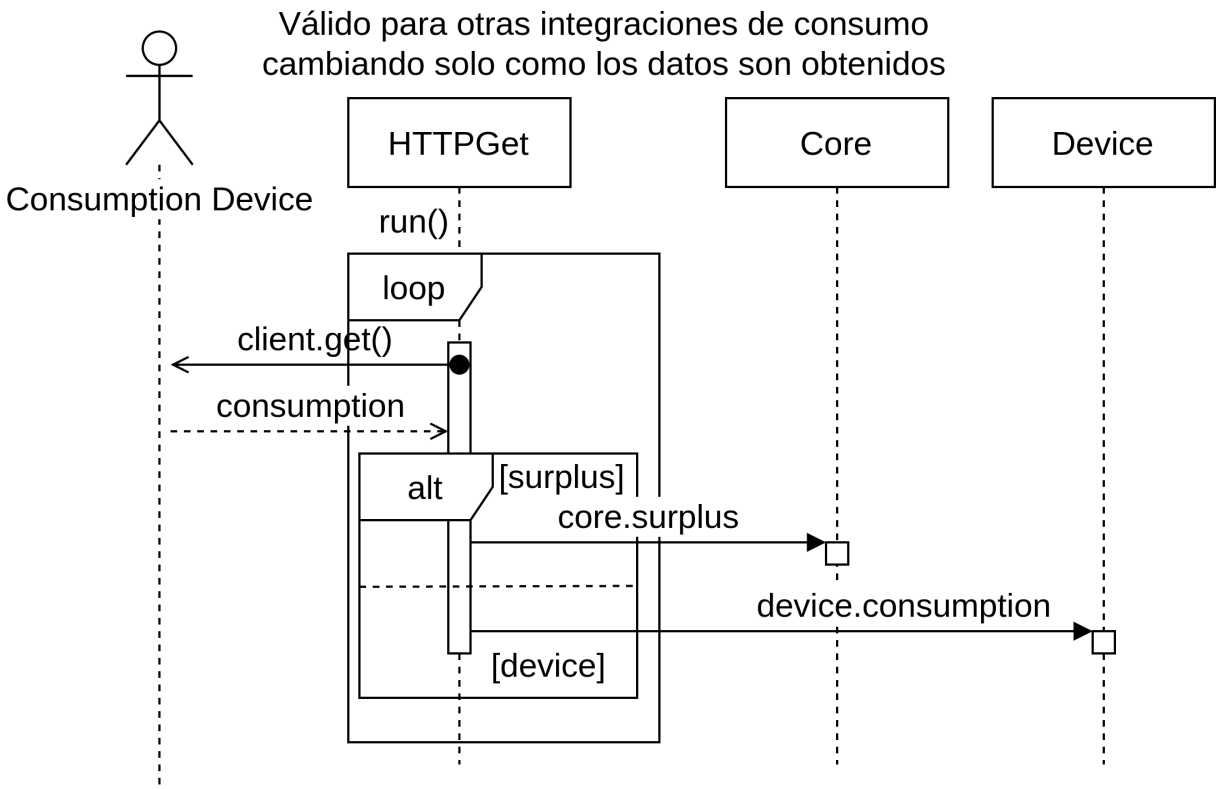

Figura 3.3: Diagrama de secuencia de obtención del dato de consumo

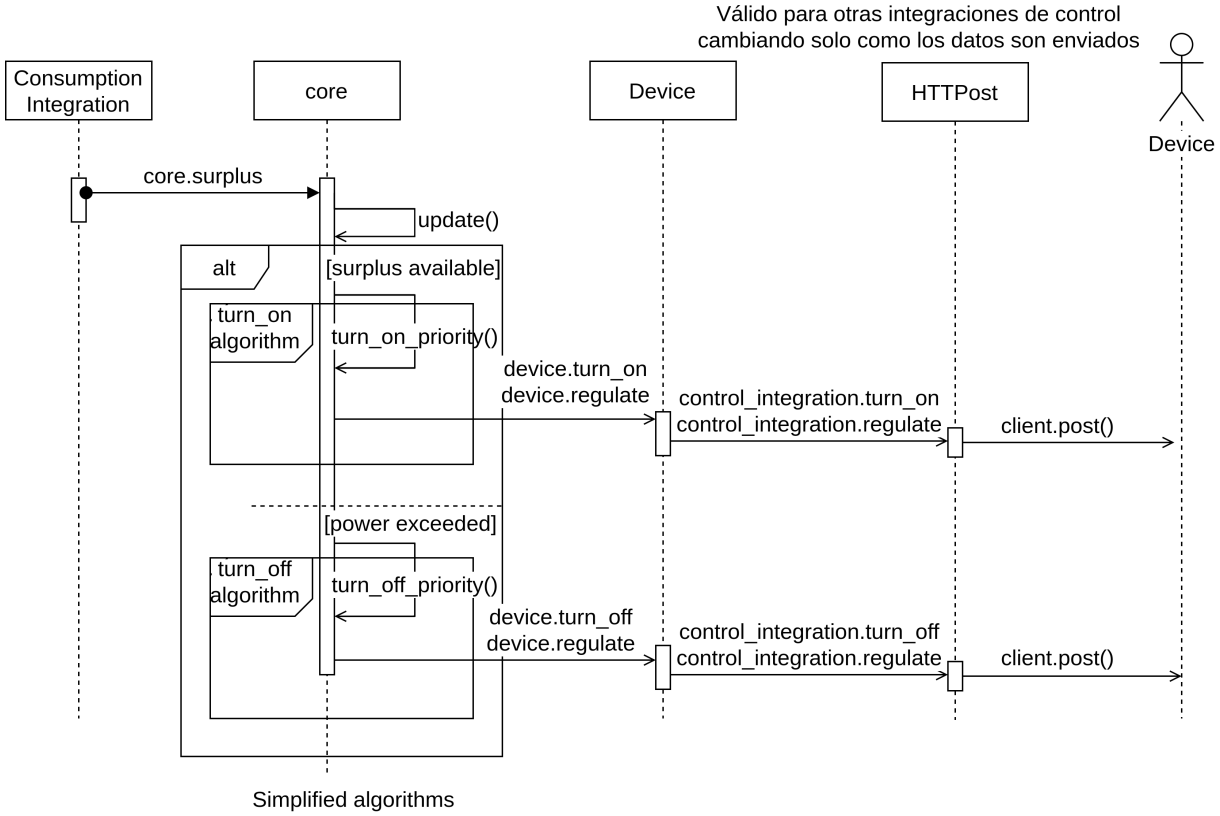

Figura 3.4: Diagrama de secuencia de control de dispositivos

4

Implementación

Para la implementación se ha decidido dividir el desarrollo en iteraciones tal y como se haría en un equipo de desarrollo que busca entregar un producto en un tiempo determinado, el fin de esta metodología es poder crear en la primera iteración un producto mínimo viable, o *MVP* por sus siglas en inglés, que permita comprobar si la idea es viable y si los resultados obtenidos cumplen con los objetivos marcados. En las siguientes iteraciones se irán añadiendo funcionalidades y mejoras al sistema hasta completar la lista de requisitos marcados en el diseño. Cada iteración tendrá el fin de cumplir con alguno de los objetivos marcados en el diseño y englobará los requisitos que permitan cumplir con ese objetivo.

4.1. Primera iteración

Esta primera iteración es la más importante y extensa ya que contiene toda la lógica que permite al sistema funcionar. Se incluirá aquí lo mínimo imprescindible para que el sistema pueda ser probado y así comprobar si los resultados obtenidos son los esperados.

4.1.1. Punto de entrada e inicio de la aplicación

Toda la aplicación queda encapsulada en un paquete de *Python* llamado *opensurplusmanager*. La ejecución de este paquete lanza un *script* *__main__.py* y dicho *script* hace uso del método *run* de *asyncio* para cargar las configuraciones y poner en marcha tanto las integraciones como el *core* dentro del bucle de eventos de *asyncio* [10].

Para el correcto funcionamiento del sistema se necesita un fichero de configuración *yaml* que servirá tanto para configurar el sistema como para guardar los datos de configuración si se modifica algún parámetro a través de otra fuente. Un fichero de configuración de ejemplo se podrá encontrar en el repositorio del proyecto. Para la configuración son necesarios los parámetros *surplus*, donde se indica la información necesaria de la integración que se encarga de recopilar el dato de excedente de energía, *surplus_margin*, que es el número de vatios que se desea dejar de margen para inyectar a la red,

grid_margin, que es el número de vatios que se desea dejar de margen para consumir de la red en caso de pico de consumo evitando modificar los dispositivos por un pico de consumo poco relevante y, por último, *devices*, que es una lista con los dispositivos que se quieren controlar.

Para configurar algún parámetro de las integraciones haremos uso de la sección *integrations* de la configuración.

De esta forma se cumple el requisito funcional RF1.1.

Configurado el sistema, el *script* de entrada se encarga de buscar las integraciones en la carpeta *integrations* tal y como se describe en el requisito funcional RF2. Estas integraciones son colocadas en forma de paquetes en la carpeta *integrations* y cada integración debe tener un fichero *__init__.py* con un método *setup* que se encarga de inicializar la clase que contiene la lógica de la integración. Esta inicialización puede requerir de la creación de una nueva tarea dentro del bucle de eventos de *asyncio* para que la integración pueda funcionar de forma asíncrona cumpliendo con los requisitos no funcionales RNF4 y RNF5.

Por último, antes de implementar la lógica del *core* y de las integraciones, se añade un sistema de *logging* que permita al usuario y al desarrollador comprender el funcionamiento del programa. Este sistema escribe los mensajes en un fichero dentro de una carpeta *logs* y en la salida estándar. El nivel de *logging* se puede configurar a través de la variable de entorno *LOG_LEVEL* y se establece en *INFO* por defecto. De esta forma se cumplen los requisitos no funcionales RNF3.1 y RNF3.2.

4.1.2. *Core* y algoritmo de control

Respetando el diseño del sistema, se ha implementado el *core* que es el encargado de dirigir a los dispositivos. El funcionamiento del *core* es bastante sencillo, aparte de cargar los dispositivos y de recibir los datos de las integraciones, la función principal del *core* es la de ejecutar el algoritmo de control. Este algoritmo se ejecuta cada vez que alguna integración actualiza el valor de excedentes (*surplus*), gracias al decorador *@property* que permite que en cada actualización de este atributo de la clase *core* se haga una ejecución asíncrona del algoritmo.

En el algoritmo, cada vez que se actualiza el valor de excedentes se comprueban dos condiciones en el valor de los excedentes. Si el excedente es mayor que el margen de excedentes (*surplus_margin*) se ejecuta el algoritmo de encendido con prioridad, es decir, según el orden en el que se han añadido los dispositivos a la lista de dispositivos. Si el excedente es menor que el margen de consumir de la red (*grid_margin*) se ejecuta el algoritmo de apagado con prioridad.

Para el encendido se ha implementado el algoritmo 4.1. Este algoritmo itera sobre los dispositivos que están habilitados, un dispositivo puede estar deshabilitado durante un tiempo tras su encendido o apagado para evitar que esté cambiando su estado continuamente dañando algunos tipos de disposi-

Data: $devices$: a list of devices, $available\_power$: the amount of power available to use
Result: Turn on/regulate devices according to the available power

```
1  foreach device in Enabled( devices ) do
2      if DeviceType( device ) = SWITCH then
3          if ExpectedConsumption( device ) < available_power and ¬
            Powered( device ) then
4              try await TurnOn( device )
5              catch continue
6              available_power ← available_power − ExpectedConsumption( device
                )
7          end
8      else if DeviceType( device ) = REGULATED then
9          if ExpectedConsumption( device ) < available_power and ¬
            Powered( device ) then
10             try await TurnOn( device )
11             catch continue
12             device_power ← { MaxConsumption( device ),  if available_power >
                                                            MaxConsumption( device )
                              { available_power,            else
13             try await Regulate( device, device_power )
14             catch continue
15             available_power ← available_power − device_power
16         else if Powered( device ) and Consumption( device ) < idle_power
            then
17             total_device_power ← Consumption( device ) + available_power
18             device_power ← { MaxConsumption( device ),  if total_device_power >
                                                            MaxConsumption( device )
                              { total_device_power,         else
19             try await Regulate( device, device_power )
20             catch continue
21             added_power ← device_power − Consumption(device)
22             available_power ← available_power − added_power
23         end
24     end
25 end
```

Algoritmo 4.1: Representación del algoritmo de encendido de dispositivos

tivos como motores. Este tiempo (*time_out*) se puede configurar para cada dispositivo en el fichero de configuración.

Una vez el algoritmo tiene los dispositivos disponibles para su cambio de estado identifica si el dispositivo es de tipo encendido/apagado o por el contrario si es capaz de regular su potencia. Para el primer caso, tras comprobar que la potencia disponible es suficiente y que el dispositivo está apagado procede a encenderlo, si es satisfactorio el encendido disminuye la potencia disponible para el resto de dispositivos.

Para el caso de los dispositivos regulables, si el dispositivo está apagado y la potencia disponible es suficiente, se enciende y se regula a la potencia máxima del dispositivo, si no lo es, se regula a la potencia disponible. Si el dispositivo ya estaba encendido y su consumo es mayor que la potencia de reposo (*idlePower*) se regula a la potencia máxima disponible o a la potencia máxima del dispositivo si la potencia disponible es mayor que la máxima del dispositivo. Al igual que con el otro tipo de dispositivo se calcula la potencia disponible para el resto.

La razón por la que se comprueba la energía en reposo es porque algunos dispositivos pese a estar encendidos deciden dejar de consumir si se dan ciertas condiciones (por ejemplo, un calentador de agua que deja de calentar si el agua está a la temperatura deseada), en ese caso no tiene sentido aumentar la potencia de ese dispositivo porque no va a consumir más.

El pseudocódigo del apagado con prioridad se puede ver en el algoritmo 4.2. Este algoritmo itera sobre los dispositivos habilitados en orden inverso, es decir, desde el último dispositivo añadido en la configuración hasta el primero.

Para liberar carga se hace un proceso parecido al encendido. Para los dispositivos de tipo encendido/apagado se apaga si está encendido y su consumo es mayor que la potencia de reposo. En cambio para los dispositivos regulables se decidirá apagar o regular a una potencia menor según la potencia que se necesite liberar. Cuando se consigue llegar a la potencia necesaria se termina el bucle.

Con la implementación de estos algoritmos se cumple el requisito funcional RF6.

4.1.3. Integraciones

Las integraciones se encargan de ampliar la funcionalidad del sistema principalmente sirviendo como medio para comunicarse con otros servicios o dispositivos inteligentes.

Para conseguir el *MVP* de esta iteración y basado en los diseños de puntos anteriores se han implementado dos integraciones. Una que hará la función de control de dispositivos y otra que consultará los datos de consumo de estos.

```
Input: devices: a list of devices, exceeded_power: the amount of power that exceeds the grid margin
Output: Turn off/regulate devices according to the exceeded power
1  foreach device in Reversed( Enabled( devices ) ) do
2      if Powered( device ) and Consumption( device ) > idle_power then
3          if DeviceType( device ) = SWITCH then
4              try await TurnOff( device )
5              catch continue
6              exceeded_power ← exceeded_power − ExpectedConsumption( device )
7          else if DeviceType( device ) = REGULATED then
8              if exceeded_power > Consumption( device ) − ExpectedConsumption( device ) then
9                  try await TurnOff( device )
10                 catch continue
11                 exceeded_power ← exceeded_power − ExpectedConsumption( device )
12             else
13                 await Regulate( device, Consumption( device ) − exceeded_power )
14                 break
15             end
16         end
17     if exceeded_power < 0 then
18         break
19 end
```

Algoritmo 4.2: Algoritmo de apagado de dispositivos

Integración de consumo *HTTP Get*

Se desarrolla esta integración con el fin de recopilar los datos de consumo de nuestros dispositivos. Se crea un paquete con el nombre de nuestra integración y en el fichero *__init__.py* se sitúan dos elementos: la función *setup* encargada de inicializar la clase de la integración para ponerla en marcha y la clase de la integración que hereda de la clase *ConsumptionIntegration* tal y como se propone en la etapa de diseño.

Para la configuración de esta integración es necesario crear una tarea en el bucle de eventos de *asyncio* que llame al método *run* de la clase ya que necesita actualizarse cada cierto tiempo. También es necesario devolver la instancia de la clase al programa principal ya que este se encargará de llamar al método *close* cuando cierre el programa.

La clase, al heredar de *ConsumptionIntegration*, no tiene métodos que implementar pero si recibe los atributos *entities* para almacenar los datos de consumo de cada dispositivo y *core* para acceder a los datos de la clase *core*.

En el caso particular de esta integración se añaden los atributos *client*, para almacenar el cliente de *aiohttp* [11] que se encargará de hacer las peticiones *HTTP*, y *timeout* para establecer un tiempo de espera en las peticiones.

Cada integración puede añadir a su lista de entidades una entidad personalizada que herede de la clase *ConsumptionEntity* de esta forma se puede añadir información necesaria para realizar la consulta de consumo, como es el *path* en este caso.

La clase de esta integración se inicializa creando la sesión de *aiohttp* y guardando su configuración. Esta configuración es particular de cada integración y se puede añadir en el fichero de configuración, es por ello importante que cada integración documente los datos que sean necesarios para su correcto funcionamiento. Al requerir de una sesión de *aiohttp* es necesario cerrarla cuando se cierre la integración con el método *close*.

Para la carga de las entidades se extrae de la configuración el tipo que es y se añade a la lista de entidades. En caso de que la entidad corresponda a un dispositivo se obtiene este dispositivo del *core* para poder modificar su consumo.

En cuanto a la obtención del dato de consumo: cada cierto tiempo establecido se itera sobre las entidades haciendo peticiones *HTTP GET* y si es satisfactoria se actualiza el consumo.

Esta integración cumple con el requisito funcional RF3.1.

Integración de control *HTTP Post*

La integración de control se implementa de forma similar a la integración de consumo, en este caso la integración no tiene que estar continuamente ejecutándose sino que se ejecutará cuando el *core* lo requiera.

Al igual que la integración *HTTP Get* se crea un paquete con el nombre de la integración y se añaden los métodos *setup* y la clase de la integración. Al inicializarse se crea la sesión de *aiohttp* por lo que también es necesario devolver la instancia de la clase al programa principal para que este pueda llamar al método *close* cuando cierre el programa.

La clase de la integración hereda de la clase *ControlIntegration* y recibe los atributos *core* y las entidades de encendido, apagado y regulación. Un dispositivo puede tener una integración de encendido/apagado distinta a la de regulación (por ejemplo, un punto de recarga de vehículo eléctrico que se enciende y apaga con un contactor y se regula la potencia via *API*), a su vez el dispositivo puede tener distinta configuración para cada petición, por ejemplo, un dispositivo puede tener un atributo *body* de la petición distinto para encenderlo que para apagarlo. Por ello las integraciones de control dividen las entidades según el tipo de control.

La integración, una vez cargada la configuración, se incluye en el dispositivo con una llamada al *core*, con ello logramos que el *core* cuando mande una orden de control hará que el propio dispositivo se encargue de llamar a la integración correspondiente.

La integración al heredar de *ControlIntegration* tiene que implementar los métodos *turn_on*, *turn_off* y *regulate*. Estos métodos se encargan de hacer las peticiones *HTTP POST* con los datos necesarios almacenados en cada entidad, que también es personalizada para esta integración. En este caso la entidad *HTTPostEntity* almacena el *path*, el *body*, *method* y *headers*.

Esta integración cumple con el requisito funcional RF3.2.

Con la implementación de estas integraciones se finaliza la primera iteración del proyecto y se finaliza el *MVP*. Con el *software* en este estado se puede probar el correcto funcionamiento del sistema y comprobar si los resultados obtenidos son los esperados. Estos resultados se muestran en el capitulo 5.

4.2. Segunda iteración

Una parte importante del proyecto es poder desplegar el *software* de forma sencilla en un servidor y que sea compatible con *Home Assistant*. *Home Assistant* permite la ejecución de*software* en forma de *add-ons* que se pueden instalar en el sistema *Home Assistant* OS y que corren sobre *Docker*. Además, se comunicará el *software* con *Home Assistant* a través una integración de *Home Assistant*.

4.2.1. Despliegue en contenedor *Docker*

Docker es una herramienta que nos permite empaquetar una aplicación y todas sus dependencias en un contenedor que se puede ejecutar en cualquier máquina de forma aislada. Para poder tener nuestro *software* en un contenedor *Docker* necesitamos un archivo *Dockerfile* que describa cómo se debe construir el contenedor. Al ser un desarrollo en *Python* existen imágenes base de *Python* que podemos usar para construir nuestro contenedor. En el archivo *Dockerfile* se especifica que se va a usar una imagen base de *Python*. Se copian los archivos, se crean las carpetas necesarias para la ejecución, se listan los puertos a utilizar y por último se ejecuta el punto de entrada de la aplicación.

Una vez tenemos el *Dockerfile* podemos construir la imagen, ya sea de manera local o en un repositorio. En este caso se hace uso del sistema propio de *GitHub* que permite construir imágenes y publicarlas en el mismo repositorio donde se almacena el código. Para ello se necesita un archivo de configuración para hacer uso de las *GitHub Actions*. En este archivo se especifica que cada vez que realizamos un despliegue en *GitHub* se va a construir una imagen de *Docker*, se le va dar un nombre y se le asignará una etiqueta.

Si el despliegue es correcto la imagen podrá ser usada y descargada a través de *GitHub Container Registry*. Además, para facilitar el despliegue se crean los archivos de *Docker Compose* que permite ejecutar los contenedores evitando los comandos de *Docker*.

Este proceso junto al desarrollo en *Python* cumple el requisito no funcional RNF1.

4.2.2. *API REST*

Para poder desarrollar la integración de *Home Assistant* es necesaria una forma de comunicarse con el sistema, para ello se ha desarrollado una *API REST*. La *API REST* es una interfaz que permite la comunicación entre dos sistemas a través de peticiones *HTTP*. Anteriormente se ha usado el *framework aiohttp* en forma de cliente y ahora se usará para crear un servidor. El *framework* permite una comunicación asíncrona durante el manejo de las peticiones, pero al iniciar el servidor no como un programa independiente sino como parte de otro programa más grande, también asíncrono, no se puede hacer uso de los métodos recomendados por la documentación y según se indica en ella se debe usar la API alternativa de *Application runners*, que permite un control a más bajo nivel del inicio y cierre del servidor asíncrono.

Como suele ser habitual con *API REST* se hace uso de los métodos *GET* para obtener información y *POST* para actualizar los datos del programa. En este caso podemos consultar la información del *core* que incluye el dato de *surplus* y las configuraciones *surplus_margin*, *grid_margin* e *idle_power*. Todos estas configuraciones son modificables con el método POST.

Además de la información del *core* se puede consultar la información completa de los dispositivos y

se pueden modificar los datos de *max_consumption*, *expected_consumption* y *cooldown* de cada uno de ellos.

Con esta implementación se cumple el requisito funcional RF4.

4.2.3. Cliente de *Python*

Antes de desarrollar la integración de *Home Assistant* se ha desarrollado un cliente de *Python* que permite la comunicación con la *API*. Esto es un requisito según la documentación oficial de *HomeAssistant* donde se indica que la comunicación con dispositivos y servicios solo podrá darse mediante la interacción entre objetos y no haciendo llamadas directas a la *API*, además se indica que la librería debe ser alojada en *PyPi* [12].

Para el desarrollo de este cliente se vuelve a hacer uso del *framework aiohttp*. La herramienta para crear clientes de *aiohttp* permite que las peticiones sean asíncronas. Para su funcionamiento, cuando se crea el objeto de la clase se inicia la sesión de *aiohttp* y el *host* al que dirigir las aplicaciones, en el cliente hay dos métodos privados uno para obtener información de la *API REST* a través de peticiones *GET* y otro para modificar la información a través de peticiones *POST*, el resto de métodos son públicos y, a través de los métodos privados, describen los distintos recursos de la *API REST* y modela la respuesta en una clase. Además incorpora la librería *backoff* que permite reintentar la conexión en caso de fallo.

Como se debe publicar en *PyPi* se ha creado un archivo *setup.py* que contiene la información necesaria para publicar la librería. Este archivo contiene datos como el nombre de la librería, la versión, la descripción, el autor, la licencia, las dependencias y el repositorio de *GitHub*. Para su publicación se vuelve a hacer uso de las *GitHub Actions* tal y como recomienda la documentación de *PyPi*. El paquete será publicado en el repositorio de *PyPi* cada vez que se publique una *release* (cierre de una versión del programa) en *GitHub* [13].

4.2.4. Integración de *Home Assistant*

Ya se cumplen todos los requisitos previos para desarrollar la integración de *Home Assistant*. Esta integración tiene bastante dificultad ya que, pese a que existe documentación para este tipo de desarrollos, esta es escueta y está desactualizada en algunos fragmentos.

En *Home Assistant* existen dos tipos de integraciones, las que están incluidas dentro del *core* y las llamadas *custom components*. En su funcionamiento son idénticas solo difiere que el código no está supervisado por el equipo de *Home Assistant* y por lo tanto no están incluidas en la instalación y se deben añadir desde un repositorio externo.

La lógica de la integración es compleja por como se comunica con el *core* de *Home Assistant*,

esta comunicación debe seguir una serie de reglas que garantizan el correcto funcionamiento de la integración.

El primer paso es la configuración de la integración, en esta se definirá que información debe pedir el *frontend* al usuario, en este caso se debe indicar el *host* donde tiene alojado la instancia de *Open Surplus Manager*. Antes de cargar la integración se comprobará que puede conectarse al servidor.

Para esta integración se van a hacer uso de tres plataformas, una plataforma es la forma que tiene *Home Assistant* para categorizar las distintas tipos de información. En el caso de esta integración se hace uso de las plataformas *sensor* (que representa un dato no modificable), *binary sensor* (una dato booleano no modificable) y *number* (un dato numérico modificable). Cada plataforma tendrá su propio fichero donde se configura la entrada de la integración correspondiente a esa plataforma.

Una vez se tiene el *host* validado entonces se inicia la configuración de la integración y, de forma asíncrona, se obtiene la información de los dispositivos y del *core* de *Open Surplus Manager* para almacenarlo en un coordinador. Las plataformas a utilizar se indican para que *Home Assistant* pueda inicializarlas. Además, se indican las instrucciones para cuando *Home Assistant* requiera eliminar la entrada de esta configuración.

Como se ha explicado anteriormente, hay varias plataformas y para cada una de ellas un fichero de código. En cada uno lo primero que se indica es el método de configuración para dicha plataforma, en este método le indicamos a *Home Assistant* qué entidades le corresponden a esta plataforma, por ejemplo, los datos de consumo y excedentes son dos entidades distintas que corresponden a la misma plataforma *sensor*. Todo el proceso de añadir entidades se hace de forma asíncrona respetando las reglas de diseño de las últimas versiones de *Home Assistant*.

Cada entidad tendrá unos métodos y atributos que dependen de la plataforma y los adquieren mediante herencia de clases. Los atributos de las entidades contienen información de esta como el tipo de dato o las unidades a utilizar, por ejemplo, potencia y vatios. Los métodos dependen del tipo de entidad y de la lógica de actualización de los datos.

Para este tipo de integración se ha hecho uso de unas clases *OSMCore* y *OSMDevice* que están definidas en su propio fichero y almacenan la información que reciben de la *API* a través del cliente de *pyOSManager*. Las distintas entidades tienen acceso a los objetos de estas clases para obtener la información que necesitan. Para evitar sobrecargar la *API* solo una entidad de cada clase se encarga de actualizar la información de los objetos, esta actualización se la indica el *core* de *Home Assistant* a través de un método *async_update*. Además, también tienen métodos para obtener la información la primera vez que se cargan las entidades sin esperar a la llamada del método *async_update*, esta primera actualización también se recibe solo una vez controlando la inicialización de los objetos. Para que *Home Assistant* conozca el valor de las entidades utiliza la propiedad *native_value* de cada entidad, previamente inicializada. En el caso de la plataforma *number*, al ser un dato modificable, se ha hecho uso de la propiedad *async_set_value* para modificar el valor de la entidad, enviando la actua-

lización a través del cliente de *pyOSManager*. Por último *Home Assistant* relaciona las entidades de un mismo dispositivo gracias a los identificadores de la propiedad *device_info*, si los identificadores de varias entidades coinciden en dominio (identificador de la integración) y nombre de dispositivo, *Home Assistant* los mostrará juntos en la interfaz.

Las entidades modificables son aquellas que se describen en el archivo de configuración por lo que si se modifican en la integración el cambio se ve reflejado en la configuración, tal y como especifica el requisito funcional RF1.3.

Otras funcionalidades de la integración es el uso de traducciones para cada idioma que soporta *Home Assistant*, para ello se ha creado un fichero de traducciones que contiene las cadenas de texto utilizadas.

Para mayor comodidad del usuario se ha facilitado la instalación con *HACS*, la tienda de aplicaciones de la comunidad de *Home Assistant*, para ello simplemente definimos el repositorio en un fichero *hacs.json* y respetamos la estructura de directorios que se indica en la documentación de *HACS*. De este modo se podrá instalar *Open Surplus Manager* añadiendo el repositorio de *GitHub* en la integración de *HACS*.

Gracias a esta integración se cumple el requisito funcional RF5.

4.2.5. *Home Assistant* Add-on

Un *add-on* es un *software* que se puede instalar en *Home Assistant OS* y que corre sobre *Docker*. Según la documentación oficial hay varias maneras de convertir un*software* en un *add-on* pero en este caso al ya haber desarrollado una imagen de *Docker* para este proyecto se ha creado un *add-on* a partir de esta. Para ello se define un archivo *config.yaml* que contiene la información necesaria para que *Home Assistant* pueda instalar el *add-on*, datos como el nombre, la descripción, la versión, la URL de la imagen de *Docker* y los puertos a utilizar. También se ha indicado que el *add-on* requiere acceso al directorio de configuración para que el usuario pueda modificar el fichero de configuración de *Open Surplus Manager*. Para instalar este add-on simplemente se ha de añadir la URL del repositorio de *GitHub* en la tienda de add-ons de *Home Assistant OS*.

Este add-on permitirá a los usuarios que tienen en su servidor *Home Assistant OS* instalar *Open Surplus Manager* de forma sencilla y sin necesidad de tener conocimientos de *Docker*.

Con esto se cumple el requisito no funcional RNF2.

4.3. Tercera iteración

Una vez el *software* ya está listo para ser usado se ha documentar y publicar el código para que los usuarios puedan usarlo y los desarrolladores puedan mejorarlo o ampliarlo con integraciones. Para ello además de la documentación del propio código se creará una wiki, se pautará el sistema de contribuciones *Open Source* y además se incluirá una guía de desarrollo de integraciones, como ejemplo a esta guía se crea una nueva integración para obtener los datos de consumo via MQTT. Toda la documentación se hará en inglés para facilitar la comprensión a usuarios de todo el mundo.

4.3.1. Documentación dentro del código

Para facilitar el mantenimiento del código y cumpliendo con la convención *PEP 257* se documentará el código con *docstrings*. [14]

Esta documentación cumple con el requisito no funcional RNF6.1.

Además, en el repositorio se ha añadido un fichero de configuración *config.example.yaml* que permite a los usuarios ver un ejemplo de cómo se debe de configurar el sistema y cada apartado de la configuración es explicado en la wiki. Con ello se cumple con el requisito funcional RNF1.2.

4.3.2. Creación de una wiki en Github

Se ha creado una wiki en Github con la documentación del proyecto, en ella se explica el proyecto, cómo se debe de configurar el sistema, cómo se deben de crear las integraciones y cómo se debe de contribuir al proyecto.

Esta wiki cumple con el requisito no funcional RNF6.2.

Para el resto de repositorios que componen el proyecto se ha incluido toda la información adicional en el *README.md* de cada repositorio y se ha añadido un enlace al repositorio principal.

4.3.3. Repositorios y sistema de contribuciones a través de Github

Además de la wiki en el repositorio principal se completan los repositorios de GitHub con toda la información necesaria para usuarios y desarrolladores.

Además de la descripción del proyecto y los *README.md* de cada repositorio se ha añadido un fichero *CONTRIBUTING.md* que explica cómo se debe de contribuir al proyecto, siguiendo el sistema de contribuciones *Open Source*.

Complementando el fichero *CONTRIBUTING.md* se añaden plantillas para facilitar la creación de

Pull Requests e *Issues*.

También se ha añadido un fichero *LICENSE* con la licencia del proyecto. Esta licencia es la *MIT License*, aprobada por la *Open Source Initiative* [15].

Con todo esto se cumplen los requisitos no funcionales RNF7.1 y RNF7.2.

4.3.4. Guía de desarrollo de integraciones

Uno de los objetivos del proyecto es que los usuarios puedan crear sus propias integraciones de forma sencilla. Como ya se ha definido anteriormente, existen dos tipos de integraciones y entidades declaradas en la carpeta *models*, las de consumo y las de control. Para ambos tipos de integraciones el usuario debe añadir al directorio *integrations* una carpeta con nombre de la integración y dentro de esta los ficheros *__init__.py* y *entity.py*.

Las integraciones de control funcionan de la siguiente manera:

La clase de la integración debe heredar de la integración *ControlIntegration*. Esta clase está definida tal y como se ve en el fragmento listado 4.1. En esta clase tenemos los atributos *core* que es la referencia al objeto de la clase *Core* de la aplicación y un diccionario para cada método de control (encender, apagar y regular), además de los métodos abstractos que se deben de implementar en la clase hija.

Los diccionarios contienen entidades de control, estas entidades se definen en el fragmento listado 4.2. En esta clase abstracta simplemente se define el nombre de la entidad y, posteriormente, cada integración heredará de esta clase e incluirá los atributos que cada tipo de integración necesite.

Con estas dos clases abstractas de base podemos crear una integración de control, para ello primero definimos el tipo de entidad que vamos a necesitar, esta entidad debe heredar de la clase *ControlEntity* y contiene los atributos que dependen de cada integración, por ejemplo, para la integración *HTTP Post* incluye la dirección, el cuerpo y la cabecera de la petición.

Para la integración en sí, en el módulo *__init__.py* se incluye la clase de la integración y el método *setup* que se encarga de crear la clase con la referencia al *core*, también debe de crear la tarea de *asyncio* que ejecutará el método *run* de la aplicación si es que la integración necesita ejecutarse de forma periódica o escuchar eventos, además, si la integración debe de hacer alguna tarea de limpieza antes de cerrarse, debe devolver la referencia al objeto creado en el método *setup* para que el sistema pueda invocar el método *close* del objeto de la integración.

La clase que hereda de *ControlIntegration* debe de realizar los siguientes pasos:

- En el método *__post_init__* se debe de cargar los módulos que necesite para funcionar(por ejemplo un cliente *HTTP*) y debe cargar las entidades desde el atributo *config* del *core*.

Código 4.1: Clase base para las integraciones de control.

```python
@dataclass
class ControlIntegration(ABC):
    """Model for a control integration."""

    core: Core
    turn_on_entities: Dict[str, ControlEntity] = field(init=False, default_factory=dict)
    turn_off_entities: Dict[str, ControlEntity] = field(
        init=False, default_factory=dict
    )
    regulate_entities: Dict[str, ControlEntity] = field(
        init=False, default_factory=dict
    )

    @abstractmethod
    async def turn_on(self, device_name: str):
        """
        Abstract method to turn on a device.

        Parameters:
        device_name (str): The name of the device to turn on.
        """

    @abstractmethod
    async def turn_off(self, device_name: str):
        """
        Abstract method to turn off a device.

        Parameters:
        device_name (str): The name of the device to turn off.
        """

    @abstractmethod
    async def regulate(self, device_name: str, power: float):
        """
        Abstract method to regulate a device.

        Parameters:
        device_name (str): The name of the device to regulate.
        power (float): The power to regulate the device to.
        """
```

Código 4.2: Clase base para las entidades de control.

```
1  @dataclass
2  class ControlEntity(ABC):
3      """
4      Model for a control entity. Control integrations entities
5      should inherit from this class.
6      """
7
8      name: str
```

- La carga de entidades debe realizarse buscando la referencia al nombre de la integración por cada tipo de método (encender, apagar y regular), creando la entidad correspondiente y añadiéndola a los diccionarios de control. Para cada dispositivo se debe llamar al método *add_control_integration* del *core* para que este sepa sobre que integración de control debe ejecutar los métodos.
- Se debe implementar los métodos asíncronos de *turn_on*, *turn_off* y *regulate* que se encargarán de ejecutar las acciones de control sobre los dispositivos.
- Si la integración necesita ejecutarse de forma periódica se debe implementar el método *run*.
- Si la integración necesita hacer alguna tarea de limpieza antes de cerrarse se debe implementar el método *close*.

Las integraciones de consumo funcionan de manera parecida:

La clase de la integración debe heredar de *ConsumptionIntegration*. Esta clase está definida tal y como se ve en el fragmento listado 4.3. En esta clase tenemos los atributos *core* que es la referencia al objeto de la clase *Core* de la aplicación y una lista de entidades de consumo.

Código 4.3: Clase base para las integraciones de consumo.

```
1  @dataclass
2  class ConsumptionIntegration(ABC):
3      """Model for a consumption integration."""
4
5      core: Core
6      entities: List[ConsumptionEntity] = field(init=False, default_factory=list)
```

Las entidades de consumo se definen de la misma manera que las entidades de control, en la clase abstracta *ConsumptionEntity* se define el nombre de la entidad, el tipo de consumo (excedentes o de dispositivo) y el dispositivo asociado si es que lo tiene. En el fragmento listado 4.4 se puede ver la definición de la clase.

Como en las integraciones de control, en el módulo *__init__.py* se incluye la clase de la integración y el método *setup* que se encarga de crear la clase con la referencia al *core*, y también debe de crear la tarea de *asyncio* que ejecutará el método *run*, en este caso suele ser necesario ya que las integraciones de consumo suelen escuchar eventos además. Si la integración debe de hacer alguna

Código 4.4: Clase base para las entidades de consumo.

```
1
2
3   @dataclass
4   class ConsumptionEntity(ABC):
5       """
6       Model for a consumption entity. Consumption integrations entities
7       should inherit from this class.
8       """
9
10      name: str
```

tarea de limpieza antes de cerrarse debe devolver la referencia al objeto creado en el método *setup* para que el sistema pueda invocar el método *close* del objeto de la integración.

La clase que hereda de *ConsumptionIntegration* debe de realizar los siguientes pasos:

- En el método *__post_init__* se debe de cargar los módulos que necesite para funcionar (por ejemplo un cliente *MQTT*) y debe cargar las entidades desde el atributo *config* del *core*.
- La carga de entidades debe realizarlo buscando la referencia al nombre de la integración para cada dispositivo o para la entidad *surplus*. Aquí es importante obtener la referencia del dispositivo asociado a la entidad a través del método *get_device* del *core* para que se pueda actualizar el consumo de los dispositivos.
- Con la lista de entidades de consumo a través del método *run* se actualizará el atributo *consumption* de cada dispositivo asociado a la entidad o el atributo *surplus* del *core*.
- Si la integración necesita hacer alguna tarea de limpieza antes de cerrarse se debe implementar el método *close*.

Una guía similar a esta se incluye en la wiki del proyecto para que los usuarios puedan crear sus propias integraciones.

Esta guía cumple con el requisito no funcional RNF7.3.

4.3.5. Ejemplo de integración: *MQTT Subscription*

Con la guía anterior se ha creado una nueva integración que permite obtener los datos de potencia de los dispositivos a través de *MQTT* usando el cliente de *aiomqtt* [16]. De esta forma y sin necesidad de tocar más código que el de la propia integración se puede extender la funcionalidad del *software*. Esta integración resulta muy parecida a la anteriormente desarrollada llamada *HTTP Get*, pero cambiando el cliente *HTTP* por un cliente *MQTT* y en lugar de realizar peticiones *HTTP* cada cierto tiempo se suscribe a un *topic MQTT*.

Con esta nueva integración se demuestra la facilidad de extender el *software* y se cumple con el requisito funcional RF7.

5

Pruebas

Para comprobar el correcto funcionamiento de la implementación se analizan los resultados de varias simulaciones y se certifica que todas las partes que componen este proyecto realizan su función correctamente.

5.1. Simulaciones

Simularemos un día de una vivienda con datos de consumo y producción de una instalación fotovoltaica. Con estos datos, a través de un script de *Python* se calcularán los datos de excedentes. Estos datos de excedentes se publicarán en una *API REST* que consultará *Open Surplus Manager*. Para los dispositivos se hace uso de otro *script* de *Python* que simula el comportamiento de varios dispositivos de distinto tipo (encendido/apagado o regulables) y de distinta potencia, estos datos también se pondrán a disposición del sistema de gestión de excedentes y podrán ser controlados como si fueran dispositivos reales.

5.1.1. Simulación 1

Se trata de un día soleado y la vivienda tiene 4 dispositivos controlables. 3 de ellos solo se pueden controlar el encendido y apagado y tienen consumo de unos 200 vatios cada uno, descripción compatible con un juego de pequeños radiadores eléctricos. El cuarto dispositivo es un dispositivo regulable de entre 50 y 800 vatios, similar al comportamiento de un calefactor eléctrico.

En la figura 5.1 se muestra una gráfica con 3 datos, la curva de consumo original, sin la intervención de los dispositivos simulados (con nombre *Consumption Base*), la curva de producción fotovoltaica y la curva de consumo resultante del consumo original y la intervención de los dispositivos simulados (con nombre *Consumption*). El eje Y representa la potencia en vatios y el eje X el tiempo aunque al tratarse de una simulación no se corresponde con el tiempo real sino que cada punto representa una lectura de potencia instantánea equivalente a leer cada 5 minutos.

En una situación ideal intentaremos acercar lo máximo posible los valores de consumo con los

de producción, en este caso se puede observar que la curva de consumo se ajusta a la curva de producción, en algunas ocasiones se producen picos de potencia demasiado elevados y el sistema reacciona inmediatamente bajando consumos después de hacer esa lectura pero puede ocurrir que para la próxima lectura el consumo esté muy por debajo de la producción, ya que el pico de potencia acabó y el sistema ha bajado consumos, será en ese momento cuando el sistema vuelva a subir consumos para ajustar la curva de consumo a la de producción. Cuantas más lecturas se hagan más preciso será el ajuste.

Es especialmente representativo el comportamiento del sistema en las zonas donde la pendiente de la producción es mayor, ajustándose de manera muy precisa. En la zona intermedia de la gráfica el sistema no puede ajustarse más ya que en esta simulación la suma de los consumos gestionados es menor que la producción en algunas de las horas centrales del día y no se pueden añadir más consumos.

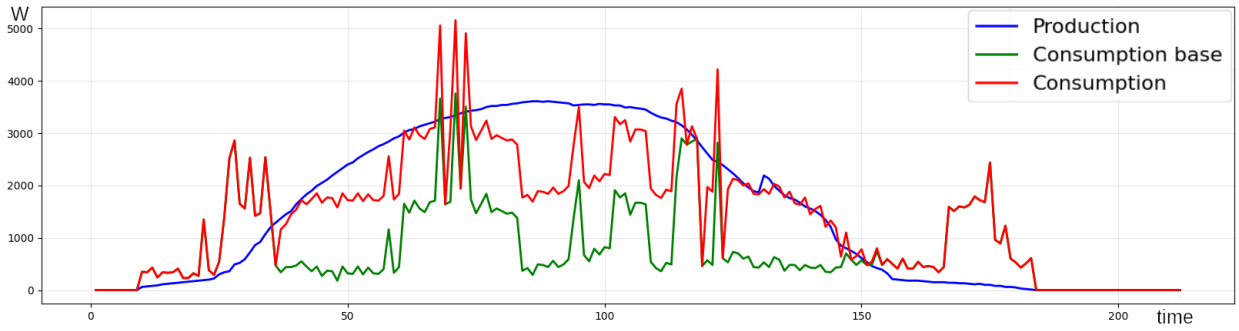

Figura 5.1: Potencia instantanea de la vivienda en simulación 1

En la figura 5.2 se muestra una gráfica con la misma simulación que la figura anterior pero con los datos de los dispositivos individuales. Lo más destacado en esta figura es el como el dispositivo regulable es muy importante para realizar ajustes finos en la curva de consumo.

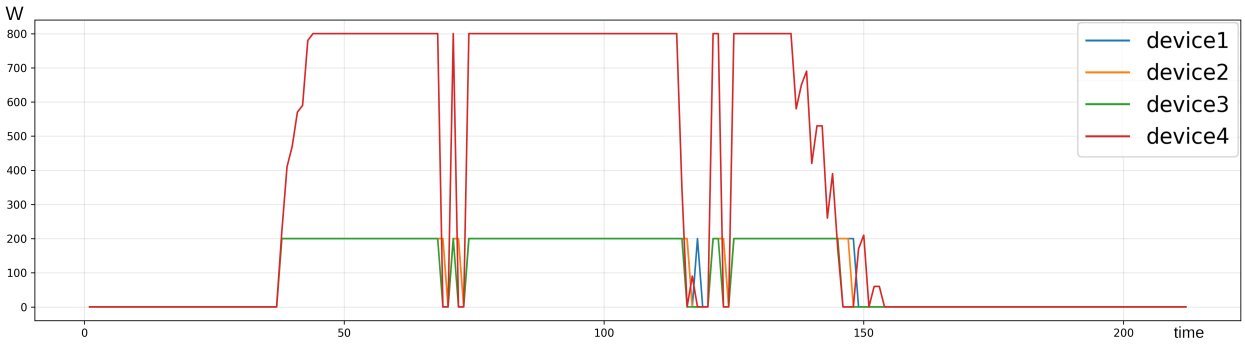

Figura 5.2: Potencia instantanea de la vivienda en simulación 1 con detalle de la intervención de los dispositivos

5.1.2. Simulación 2

En la segunda simulación los datos de consumo y producción pertenecen al mismo día pero algunos dispositivos son diferentes. En este caso los dispositivos 1 y 3 volverían a ser los radiadores eléctricos de 400 vatios no regulables, el dispositivo 2 sería parecido pero de 800 vatios y con un tiempo de enfriamiento de unos 30 minutos o 6 lecturas para el caso de esta simulación. Este tiempo es necesario en algunos dispositivos que no queremos que estén continuamente cambiando de estado ya que puede afectar a la vida útil del mismo. Esta descripción podría corresponder a un aire acondicionado. El dispositivo 4 sería un dispositivo regulable que simula ser un cargador eléctrico con un consumo de entre 1380 y 7360 vatios, también con un tiempo de enfriamiento de 30 minutos o 6 lecturas. Esta simulación además de añadir la complejidad de los tiempos de enfriamiento nos permitirá ver el desempeño con dispositivos de mayor consumo.

En la figura 5.3 podemos comparar, respecto a la figura 5.1, que la curva de consumo consigue algo más acercarse a la curva de producción aunque, pese a conseguir dispositivos de mayor consumo, en este caso se penaliza la potencia mínima del dispositivo regulable que es de 1380 vatios y que en la mayoría de momentos no puede encenderse y regularse ya que la potencia sobrante es menor a la mínima del dispositivo.

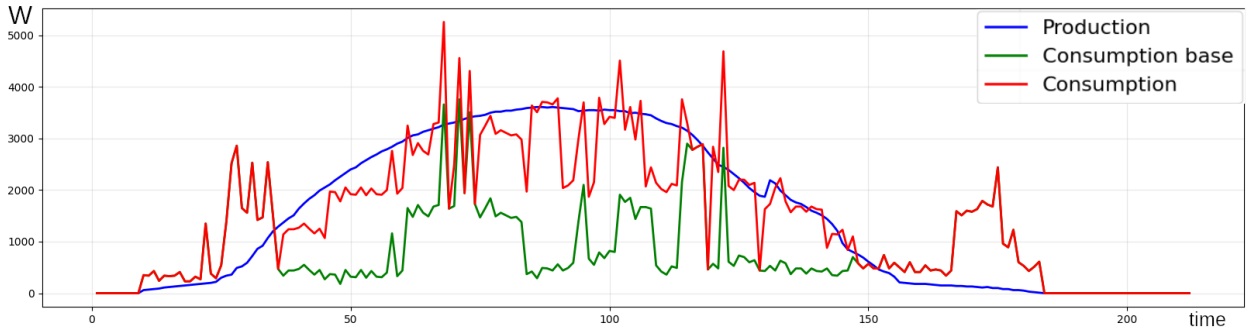

Figura 5.3: Potencia instantanea de la vivienda en simulación 2

En la figura 5.4 y la figura 5.5 se puede ver como los dispositivos con menores consumos tienen más facilidad para consumir mientras que los de mayor consumo tienen que esperar a las horas centrales del día, cuando mayor es la producción para poder encenderse.

En la figura 5.6 junto con la figura 5.5 se demuestra el funcionamiento del tiempo de enfriamiento donde los dispositivos que cuentan con esta característica quedan desactivados durante el tiempo indicado antes de poder volver a cambiar de estado.

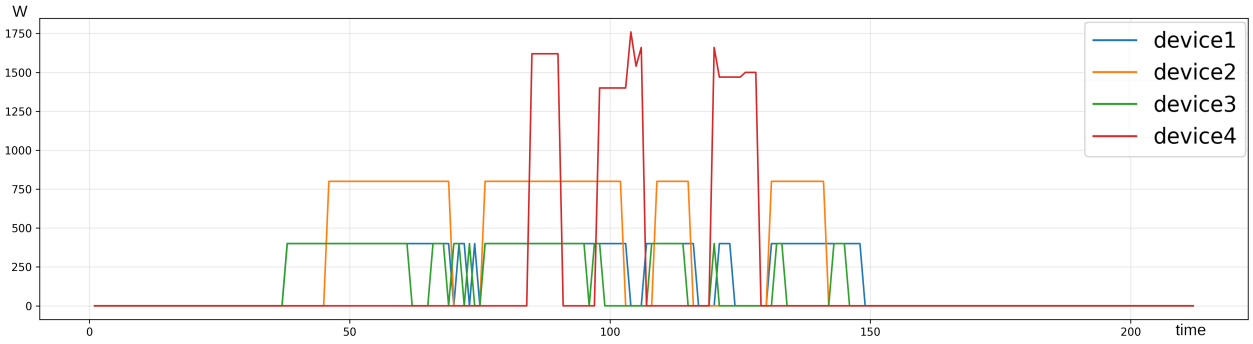

Figura 5.4: Potencia instantanea de la vivienda en simulación 2 con detalle de la intervención de los dispositivos

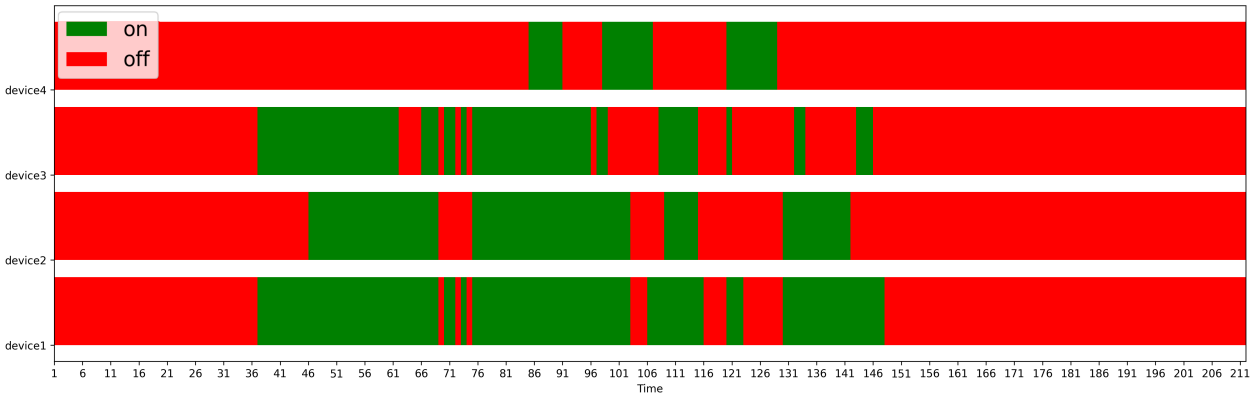

Figura 5.5: Encendido/apagado de los dispositivos en cronología de estados para la simulación 2

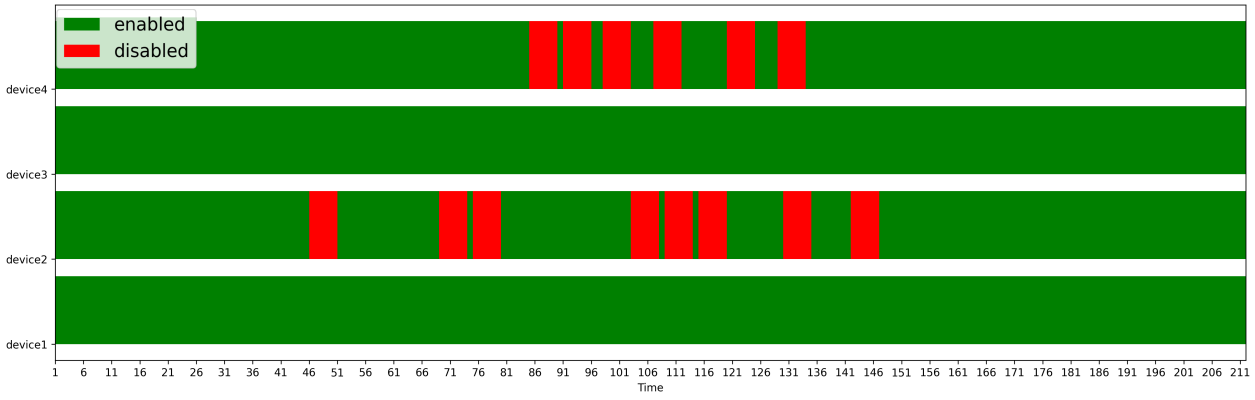

Figura 5.6: Dispositivos habilitados en cronología de estados para la simulación 2

5.1.3. Simulación 3

Para el último caso se ha seleccionado los datos de un día con mayor variabilidad debido a nubes. En este caso se inicializa el sistema con dos dispositivos uno de 800 vatios y otro de entre 50 y 3000 vatios. El primero puede corresponder a un calefactor eléctrico y el segundo a un termo eléctrico de gran capacidad. Gracias a la carga resistiva del termo eléctrico se permite un amplio rango de potencias con los que ajustar el sistema.

En la figura 5.7 se puede observar como el sistema trata siempre de ajustarse a la curva de producción pese a los grandes cambios en la producción debido a las nubes.

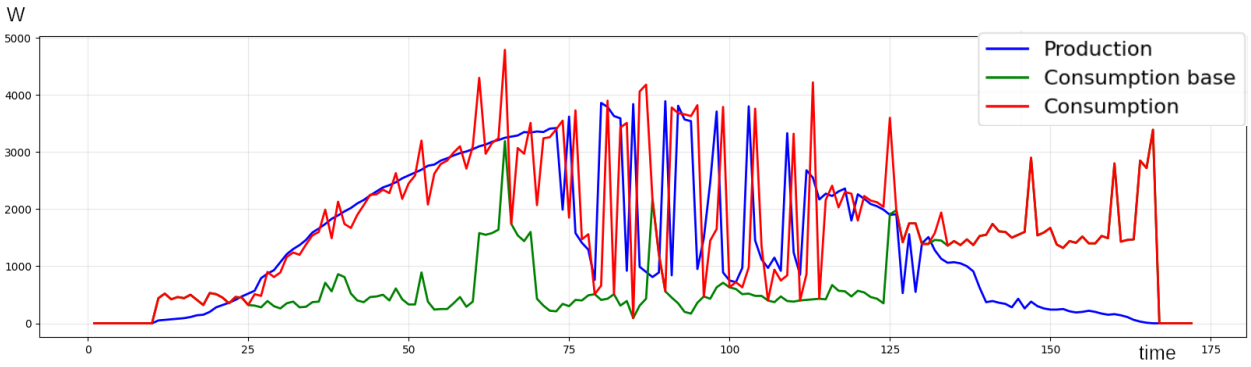

Figura 5.7: Potencia instantanea de la vivienda en simulación 3

En la figura 5.8 se puede observar como el dispositivo regulable de gran potencia es capaz de realizar el ajuste más preciso pero también permite al otro dispositivo de menor potencia entrar en acción en los momentos de mayor producción.

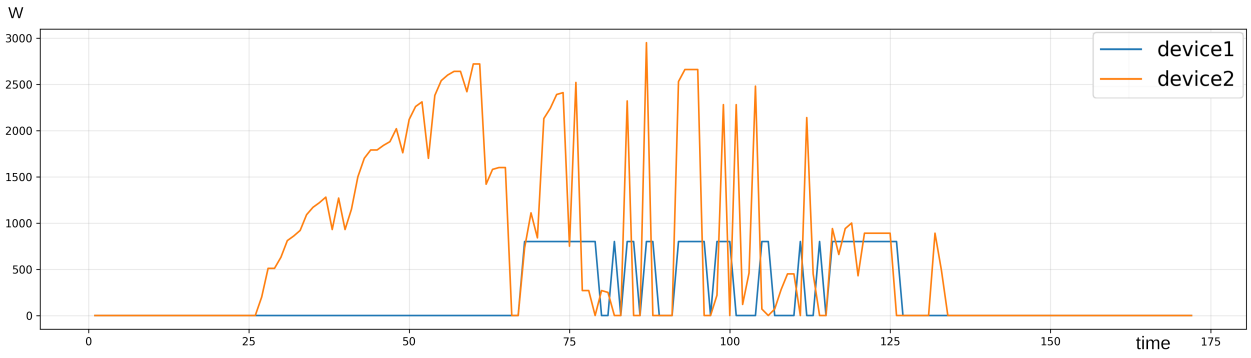

Figura 5.8: Potencia instantanea de la vivienda en simulación 3 con detalle de la intervención de los dispositivos

Con esta última simulación se demuestra que el sistema es capaz de adaptarse a situaciones tanto típicas como de mayor variabilidad y que los dispositivos regulables son una parte fundamental para el aprovechamiento de la producción fotovoltaica.

5.2. Ecosistema

Además del programa principal este trabajo también se ha enfocado en las distintas herramientas que permiten al sistema funcionar de manera más cómoda para el usuario gracias al proyecto de *Home Assistant*. Las distintas partes que componen el ecosistema son:

- *Open Surplus Manager*: el sistema principal que se encarga de gestionar los datos de consumo y de ofrecer una *API REST* para su consulta. Repositorio: `https://github.com/JoseRMorales/OpenSurplusManager`
- *OSM-HA*: la integración de *Home Assistant* que se comunica con el sistema y permite tener los datos de *Open Surplus Manager* en nuestro *Home Assistant*. Repositorio: `https://github.com/JoseRMorales/OSM-HA`
- *pyOSManager*: el cliente de *Python* que se comunica con la *API REST* de *Open Surplus Manager*. Importante ya que es requerido por *OSM-HA* aunque también puede ser usado de manera independiente. Repositorio: `https://github.com/JoseRMorales/pyOSManager`
- *OS-HA-Addon*: el *add-on* de *Home Assistant* que instala *Open Surplus Manager* en el sistema de *Home Assistant OS*, sin necesidad de conocimientos de *Docker* o de la instalación manual de *Open Surplus Manager*. Repositorio: `https://github.com/JoseRMorales/OSM-HA-Addon`

Es por ello que en esta sección se pretende demostrar todo el ecosistema funcionando en conjunto dentro de una instancia de *Home Assistant*.

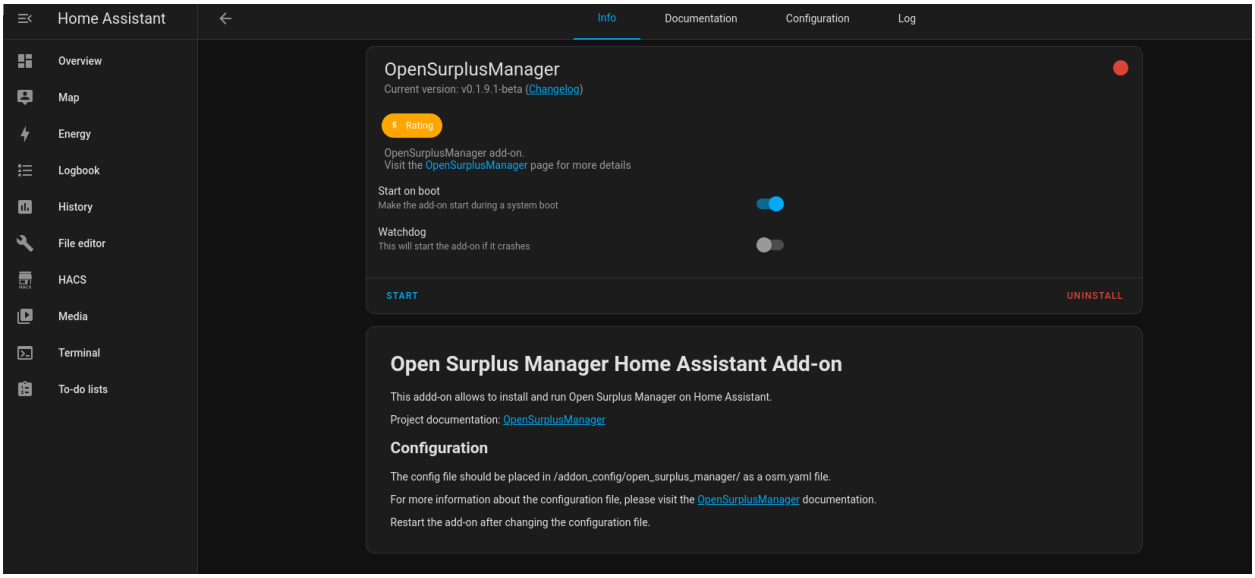

Figura 5.9: Pantalla de inicio del add-on

En la pantalla de inicio del *add-on* (figura 5.9) se puede observar la interfaz de *Home Assistant* con el *add-on* de *Open Surplus Manager* instalado. Desde aquí se puede ejecutar tal y como se haría en una instalación con *Docker* o directamente ejecutando el programa de *Python*.

En cambio en la figura 5.10 se muestra la interfaz de integraciones de *Home Assistant* donde se puede observar que la integración de *Open Surplus Manager* está instalada y configurada.

Con la integración y el *add-on* podemos tener en nuestro *Home Assistant* un *dashboard* como el

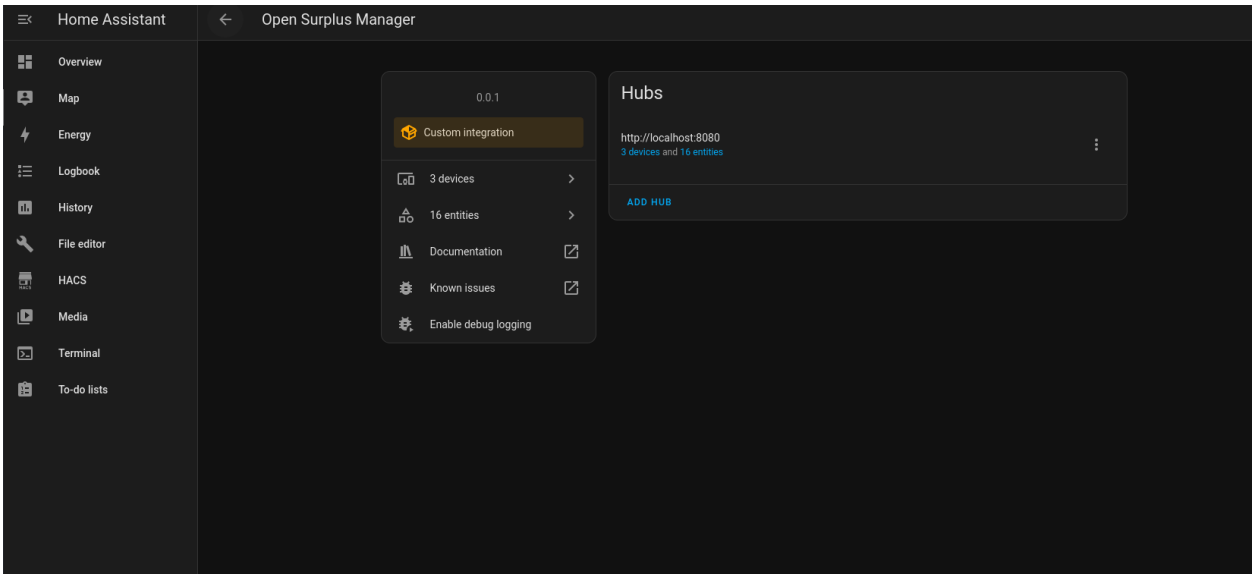

Figura 5.10: Pantalla inicio de la integración

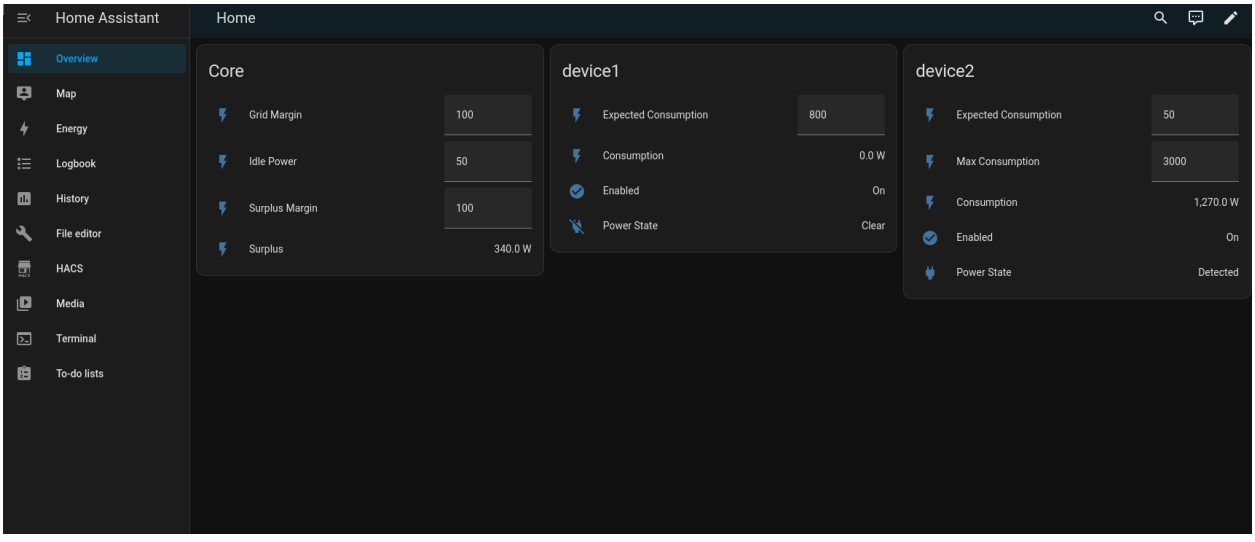

Figura 5.11: *Dashboard* de *Home Assistant* con los datos de *Open Surplus Manager*

de la figura 5.11 donde se disponer de los distintos datos que ofrece *Open Surplus Manager* a través de la *API REST* e incluso permite modificar alguna configuración.

6

Conclusiones

Tras las pruebas realizadas se puede concluir lo siguiente de los objetivos planteados antes de desarrollar el sistema:

El *software* permite al usuario gestionar los excedentes de energía de su instalación fotovoltaica de forma automática basándose en reglas establecidas por el usuario. Se ha demostrado a través de las simulaciones capaz de gestionar la energía de forma eficiente y personalizada gracias a la configuración, aunque también se ha visto que su eficiencia es dependiente de la cantidad y tipo de dispositivos de los que se disponga, en cualquier caso, resulta una mejora respecto a la no gestión o a la gestión manual de los excedentes.

Se ha desarrollado un sistema de código abierto que permite la colaboración de otros desarrolladores en el mantenimiento o mejora del mismo. Esto se ha logrado gracias a la publicación del código en un repositorio público y a la documentación del mismo.

Además de permitir la colaboración con el código de este sistema también se ha conseguido que la creación de nuevas integraciones sea lo más sencilla posible de forma que otros usuarios puedan añadir nuevos tipos de dispositivo que en un principio no estaban soportados.

Uno de los objetivos fundamentales de este trabajo era la integración con el sistema *Home Assistant* y se ha conseguido de forma satisfactoria. La integración creada permite a los usuarios obtener los datos de *Open Surplus Manager* en su instancia de *Home Assistant* y el *add-on* creado facilita la instalación y configuración de *Open Surplus Manager* dentro del propio *Home Assistant OS*.

En general el trabajo ha planteado varios objetivos a cumplir y tras el diseño, implementación y pruebas se ha conseguido cumplir satisfactoriamente con ellos. Pese a ello siempre hay margen de mejora y se proponen varias líneas de trabajo futuro en el siguiente capítulo.

7

Trabajo futuro

El trabajo futuro más inmediato es el mantenimiento del proyecto, al tratarse de un programa de código abierto es importante mantenerlo actualizado ya sea por el autor original o estar pendiente de las colaboraciones de otros desarrolladores.

En cuanto a mejoras o nuevas funcionalidades que se pueden añadir, la más lógica sería la creación de una interfaz web que además de permitir visualizar los datos de la instalación y los dispositivos permita la configuración del sistema a través de ella. La configuración actual a través de ficheros de texto aunque funcional puede ser el mayor impedimento para nuevos usuarios.

Otro trabajo que se puede realizar sería la implementación de nuevos algoritmos que se enfoquen en distintos objetivos, si bien es cierto que con las distintas configuraciones se puede adaptar el funcionamiento a lo que el usuario necesita, puede ser interesante disponer de distintos algoritmos predefinidos que el usuario pueda elegir según que configuración de dispositivos tenga.

Por último, otro de los puntos de mejora más lógicos es el añadir nuevas integraciones que amplíen el número de dispositivos compatibles, pese a que cada usuario puede crearse sus propias integraciones o modificar las existentes para adaptarlas a sus necesidades, siempre es interesante disponer de un mayor número de dispositivos soportados de forma nativa.

Bibliografía

[1] J. Leitão, P. Gil, B. Ribeiro, and A. Cardoso, "A survey on home energy management," *IEEE Access*, vol. 8, pp. 5699–5722, 2020.

[2] UNEF, "En 2023 se instalaron en España 1.706 MW de autoconsumo fotovoltaico." https://www.unef.es/es/comunicacion/comunicacion-post/en-2023-se-instalaron-en-espana-1706-mw-de-autoconsumo-fotovoltaico. [Accedido 14-11-2024].

[3] "Pack gestión de excedente de energía Shelly - Shellyspain." https://shellyspain.com/pack-gestion-de-excedente-de-energia-shelly.html. [Accedido 14-11-2024].

[4] "FreeDS." https://freeds.es/. [Accedido 14-11-2024].

[5] "Derivador de energía solar excedente IbeDiv." https://ibepower.com/product/ibediv-derivador-de-energia-solar. [Accedido 14-11-2024].

[6] "Installations | Home Assistant Analytics." https://analytics.home-assistant.io/. [Accedido 25-10-2024].

[7] "Home assistant Forecast.Solar." https://www.home-assistant.io/integrations/forecast_solar. [Accedido 25-10-2024].

[8] M. Wilkes, *Parallelization and async*, pp. 283–344. Berkeley, CA: Apress, 2020.

[9] A. Cornel Cristian, T. Gabriel, M. Arhip-Calin, and A. Zamfirescu, "Smart home automation with mqtt," in *2019 54th International Universities Power Engineering Conference (UPEC)*, pp. 1–5, 2019.

[10] "asyncio — Asynchronous I/O." https://docs.python.org/3/library/asyncio.html. [Accedido 25-10-2024].

[11] "AIOHTTP documentation." https://docs.aiohttp.org/en/stable/. [Accedido 25-10-2024].

[12] "Home Assistant Developer Docs." https://developers.home-assistant.io/. [Accedido 25-10-2024].

[13] "Publishing package distribution releases using GitHub Actions CI/CD workflows - Python Packaging User Guide." https://packaging.python.org/en/latest/guides/publishing-package-distribution-releases-using-github-actions-ci-cd-workflows/. [Accedido 25-10-2024].

[14] "PEP 257 – Docstring Conventions." https://peps.python.org/pep-0257/. [Accedido 25-10-2024].

[15] "The Open Source Definition." https://opensource.org/. [Accedido 25-10-2024].

[16] "aiomqtt — pypi.org." https://pypi.org/project/aiomqtt/. [Accedido 25-10-2024].


UÁM
Universidad Autónoma
de Madrid