COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

DESARROLLO DE UN SISTEMA FLEXIBLE PARA PERSONALIZAR EL COMPORTAMIENTO DE UNA VIVIENDA INTELIGENTE

Autor: Ignacio García Vera

Directores:

D. Jaime Boal Martín-Larrauri

D. Miguel Martín Lopo

Madrid

Julio de 2019

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. Ignacio García Vera

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: Desarrollo de un sistema flexible para personalizar el comportamiento de una vivienda inteligente, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente).*

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 15 de Julio de 2019

ACEPTA

Fdo. Ignacio García

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

DESAROLLO DE UN SISTEMA FLEXIBLE PARA PERSONALIZAR EL COMPORTAMIENTO DE UNA VIVIENDA INTELIGENTE

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2018/2019 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Ignacio García Vera Fecha: 15/07/2019

Autorizada la entrega del proyecto

LOS DIRECTORES DEL PROYECTO

Fdo.: Jaime Boal Martín-Larrauri Fecha: 15/07/2019

Fdo.: Miguel Martín Lopo Fecha: 15/07/2019

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

DESARROLLO DE UN SISTEMA FLEXIBLE PARA PERSONALIZAR EL COMPORTAMIENTO DE UNA VIVIENDA INTELIGENTE

Autor: Ignacio García Vera

Directores:

D. Jaime Boal Martín-Larrauri

D. Miguel Martín Lopo

Madrid

Julio de 2019

Desarrollo de un sistema flexible para personalizar el comportamiento de una vivienda inteligente

Autor: García Vera, Ignacio
Directores: Boal Martín-Larrauri, Jaime y Martín Lopo, Miguel.
Entidad colaboradora: ICAI - Universidad Pontificia Comillas.

Resumen del proyecto

Introducción

El Internet de las Cosas está posibilitando la aparición de nuevos hábitos de consumo dentro de la vivienda. El concepto de casa domótica incluye la gestión de los recursos energéticos y las operaciones necesarias para adecuar el estado de esta a los criterios de confort del inquilino. El rápido aumento de los requisitos que estos sistemas deben cumplir están causando que las soluciones sean cada vez más complejas. Una de las formas más novedosas de gestión de casas domóticas y en general de todo el sistema eléctrico es a través de metasimuladores de energía, como el propuesto en [1].

Los metasimuladores utilizan la filosofía de los metajuegos, es decir, se busca que el sistema no sea experto en un juego en concreto, sino que sea capaz de jugar a todos ellos, conociendo los principios fundamentales de los juegos y los bienes materiales, la información y la interacción con otros jugadores disponible. La idea es crear un metasimulador para una parte concreta del sistema eléctrico: los usuarios finales y en concreto para cubrir la gestión de la temperatura en las casas.

Estado del Arte

La creación de una plataforma global del sistema eléctrico se ha planteado en [1]. En él, se propone una arquitectura para gestionar los diferentes dispositivos que se conectan. Consta de un Master Kernel, que se utiliza para coordinar las diferentes plataformas. Existe una plataforma por cada una de las funcionalidades típicas del sistema eléctrico: generación, operador del sistema, operador del mercado, regulador, entre otras. Las plataformas serán las responsables de los agentes, que implementan funcionalidades requeridas en las plataformas y realizan las tareas.

Para que las diferentes plataformas y los agentes trabajen de forma correcta, se implementan reglas de funcionamiento entre ellos. Se identifican tres tipos de reglas: reglas de primer nivel, que describen las relaciones entre el Master Kernel, las plataformas y los agentes; reglas de plataforma, que permiten determinar a la plataforma cuando deben llamar a los agentes; y reglas de agentes, para ejecutar las tareas.

Para este Trabajo Fin de Máster únicamente se ha considerado una plataforma, la

de los consumidores finales, que incorpora un agente: los prosumidores. Para ellos, se ha añadido una unidad encargada de la gestión de la temperatura en la casa, que incorpora un control de temperatura, sensores y actuadores.

Metodología

El proyecto se desarrolla tanto en Python como C++. Se incorpora las llamadas de funciones de Python desde C++. Al ser Python un lenguaje interpretado de alto nivel, no es necesario compilar y se puede cambiar mientras es ejecutado por C++, sin necesidad de parar la ejecución del programa principal [2].

La interacción diseñada consta de sensores de temperatura y actuadores, que son radiadores o aire acondicionado, gestionados por un control que genera referencias de temperatura para mantener esta según los criterios de confort del usuario.

Para diseñarlo, se ha enfocado hacia la idea del metasimulador posibilitando la incorporación de criterios de flexibilidad. En concreto, se busca poder conectar múltiples dispositivos con diferentes características técnicas y usando diferentes tipos de control. Además, se debe permitir que el usuario establezca los parámetros según los cuales se va a controlar el sistema sin que esto afecte a la plataforma en su conjunto.

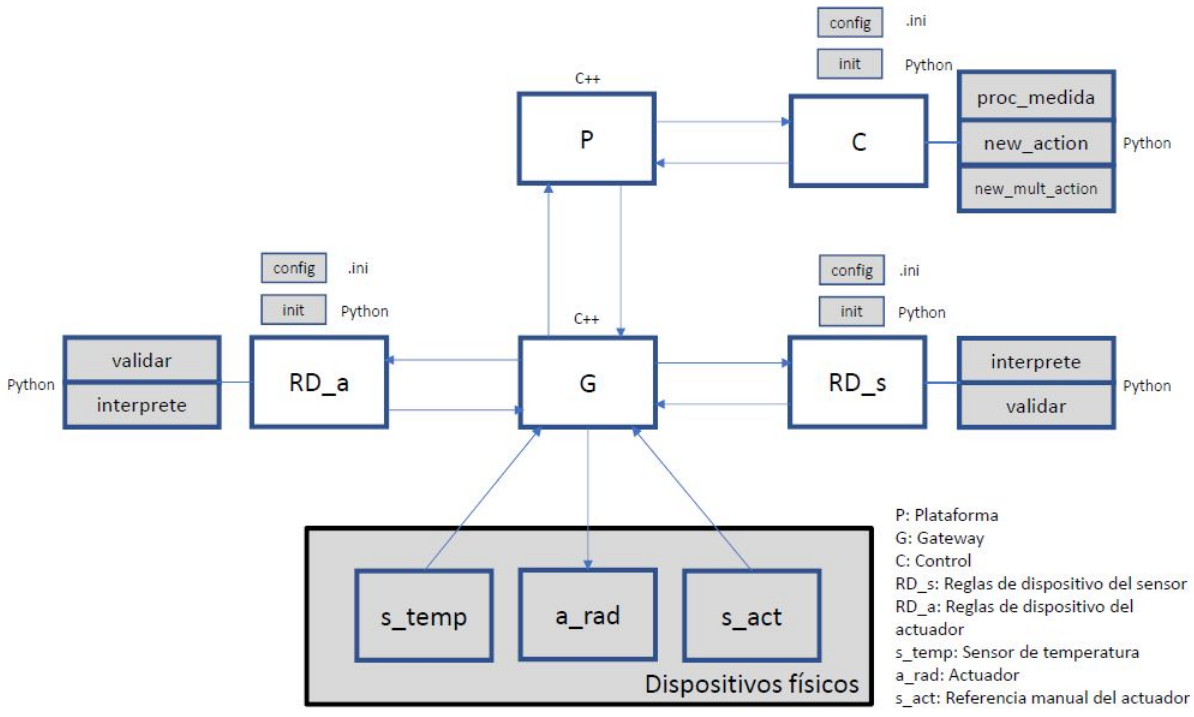

Figura 1: Esquema general de funcionamiento.

Se plantea un esquema de funcionamiento con la presencia de un *gateway*, encargado de gestionar al sensor y al actuador, y de una plataforma, que gestiona al *gateway* y al control. El sensor, el control y el actuador tienen diferentes funciones, que tienen que ser

implementadas por el usuario en Python, como se muestra en la Figura 1.

Las funciones que se han implementado en Python se muestran en la Figura 2. Además de las ilustradas, se necesita una función de inicialización de las reglas de dispositivo del sensor, control y actuador. La función de intérprete, presente en las reglas del sensor y del actuador, se utiliza para pasar los datos de las unidades de los dispositivos a aquellas utilizadas en el sistema de control y viceversa. Las funciones de validación, también presentes en el sensor y en el actuador, comprueban que tanto la medida de temperatura del sensor como el valor de la actuación son coherentes. Las reglas del control incorporan otras tres funciones que se utilizan para gestionar los datos validados que llegan procedentes del sensor y para generar las actuaciones correspondientes para mantener la temperatura dentro de los rangos de confort del usuario.

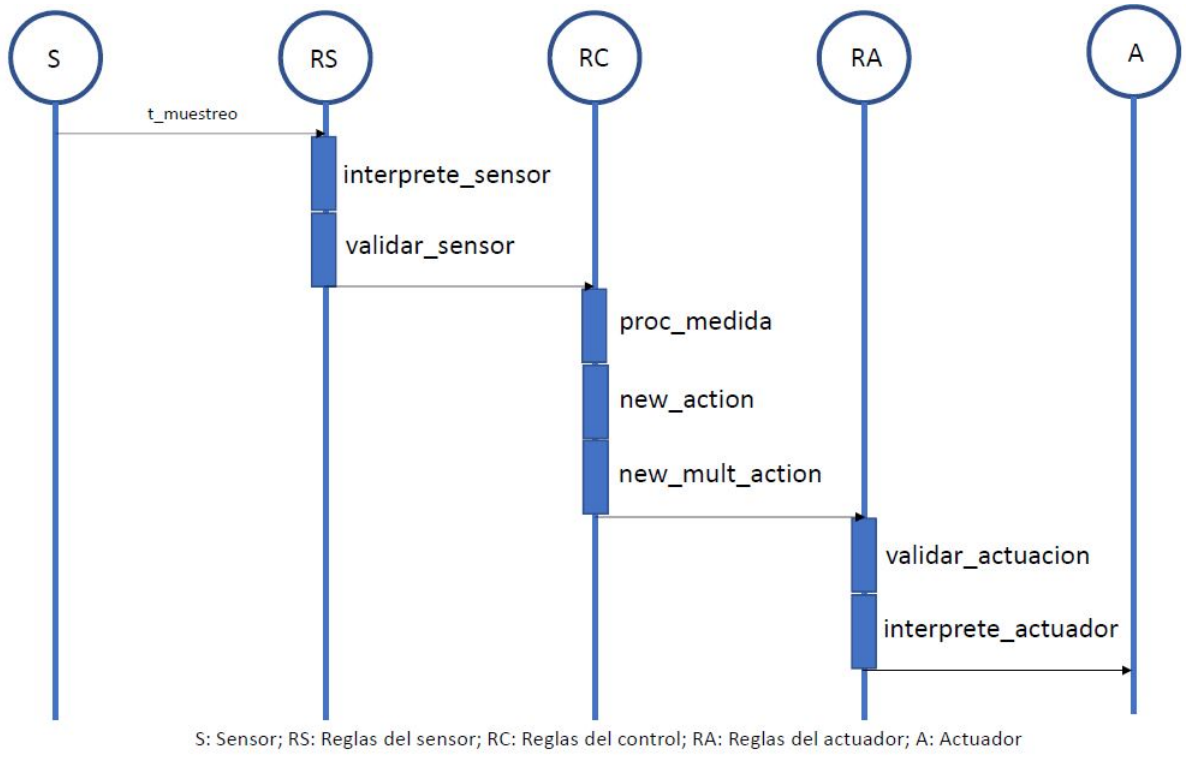

Figura 2: Diagrama UML con la evolución temporal de las funciones para el funcionamiento normal de la aplicación.

Para realizar la implementación del sistema se han requerido diferentes aspectos de programación. Entre ellos, la integración entre C++ y Python, la creación de clases para gestionar los diferentes dispositivos, la creación de múltiples hilos de ejecución en C++ y la lectura de los ficheros de configuración.

Resultados

El sistema se ha probado usando dos casos de estudio. En el primero, hay un sensor, un control y un actuador y generan datos de temperatura cada 5 segundos. Para cada

dato generado, este se valida por las reglas del sensor, se pasa al control para que genere una actuación y la actuación es finalmente validada por las reglas del actuador. La Figura 3 muestra un ejemplo de los resultados. Se comprueba que para los valores que se encuentran entre -20°C y 30°C, se ha generado una actuación, ya que se consideran datos de temperatura válidos. Además, solo las actuaciones que se encuentran entre 20°C y 23°C son actuaciones válidas, por lo que las restantes, son descartadas.

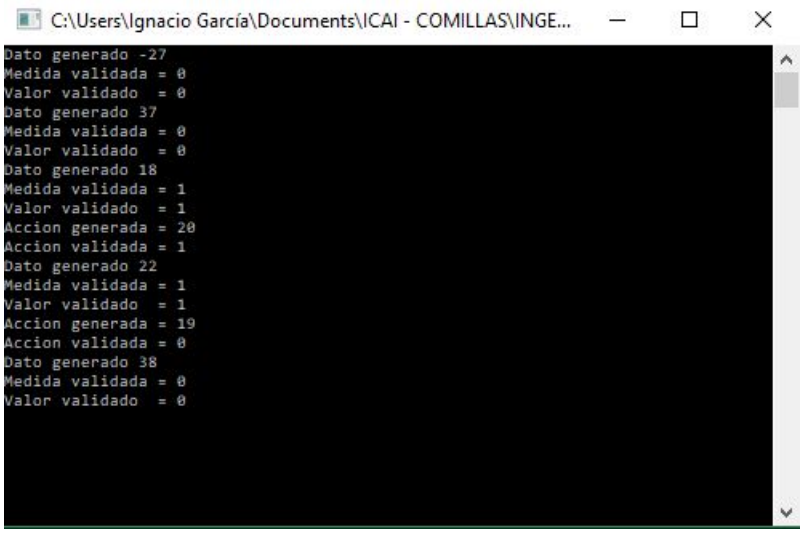

Figura 3: Ejemplo de ejecución del caso de estudio para un hilo.

En el segundo caso de estudio, se han considerado tres sensores y un único control. Los tres sensores envían sus datos al control existente. Hasta que los tres no presenten un dato válido, no se ejecuta el control, que entonces genera una actuación. La validez de la actuación se comprueba mediante las reglas del actuador, como en el caso de estudio anterior.

Conclusiones

Las principales conclusiones obtenidas en el desarrollo de este Trabajo Fin de Máster son:

- Controlar la temperatura de una vivienda de forma flexible, no limitando el número de dispositivos que se puedan conectar y permitiendo al usuario establecer sus propios criterios de confort, encaja como una parte de la plataforma de gestión eléctrica que se ha presentado en [1].
- Para controlar la temperatura, se propone un sistema conformado por sensor de temperatura, un control y un actuador. Existe un *gateway* que controla las comunicaciones con el sensor y el actuador. Además, una plataforma de encargará de gestionar al *gateway* y al control.

- El sistema creado es flexible en dos ámbitos: se pueden incorporar tantos dispositivos como se necesiten y el usuario puede modificar el funcionamiento sin alterar el resto del sistema.

Referencias

- M. Martín Lopo, J. Boal y A, Sánchez, "Transitioning from a meta-simulator to electrical applications: an arquitecture", Simulation Modelling Practice and Theory, vol. 94, p. 198, jul 2019.
- API Python/C, "docs.python.org/3/c-api/intro.html", visitado 12/07/2019.

Development of a flexible system to customize the behavior of a Smart House

Summary

Author: García Vera, Ignacio
Supervisors: Boal Martín-Larrauri, Jaime y Martín Lopo, Miguel.
Collaborating institution: ICAI - Universidad Pontificia Comillas.

Introduction

The Internet of Things is enabling the emergence of new consumption habits within the household. The concept of Smart Home includes the management of energy resources and the necessary operations to adapt the state of this to the criteria of comfort of the user. The rapid increase in the requirements for these systems is causing the solutions to become increasingly complex. One of the newest forms of management of Smart Houses and in general of the whole electrical system is through energy metasimulators, as proposed in [1].

Metasimulators use the philosophy of metagames, that is, the system is not expert in a particular game, but capable of playing all of them, knowing the fundamental principles of games and the availability of material goods, information and interaction with other players. The idea is to create a metasimulator for a specific part of the electrical system: the end users and specifically to cover the management of temperature in houses.

State-of-the-Art

The creation of a global platform for the electrical system has been raised in [1]. It proposes an architecture to manage the different devices that are connected to it. It consists of a Master Kernel, which is used to coordinate the different platforms. There is a platform for each one of the typical functionalities of the electrical system: generation, system operator, market operator, regulator, among others. The platforms are responsible for the agents, who implement the functionalities required in the platforms and carry out the tasks.

In order for the different platforms and agents to work correctly, it is necessary to implement operating rules between them. Three types of rules are identified: first level rules, which describe the relationships between the Master Kernel, the platforms and the agents; platform rules, which allow the platform to determine when to call the agents; and agent rules, to execute the tasks.

For this Master Thesis, only one platform has been considered, the one of final consumers, which incorporates an agent: the prosumers. For them, a unit will be added in charge of managing the temperature in the house, which will incorporate a temperature control, sensors and actuators.

Methodology

The project is developed both in Python and C++. Python function calls from C++ are incorporated. As Python is a interpreted, high-level programming language, meaning that it does not need to be compiled and can be changed while it is executed by C++, without having to stop the execution of the main program [2].

The interaction designed consists of temperature sensors and actuators, which are radiators or air conditioning, managed by a control that generates temperature references to maintain it according to the comfort criteria of the user.

In order to design it, it has focused on the idea of the metasimulator and flexibility criteria had to be incorporated. Specifically, the aim is to be able to connect multiple devices with different technical characteristics and using different types of control. In addition, the user must be allowed to establish the system parameters without affecting the rest of the platform.

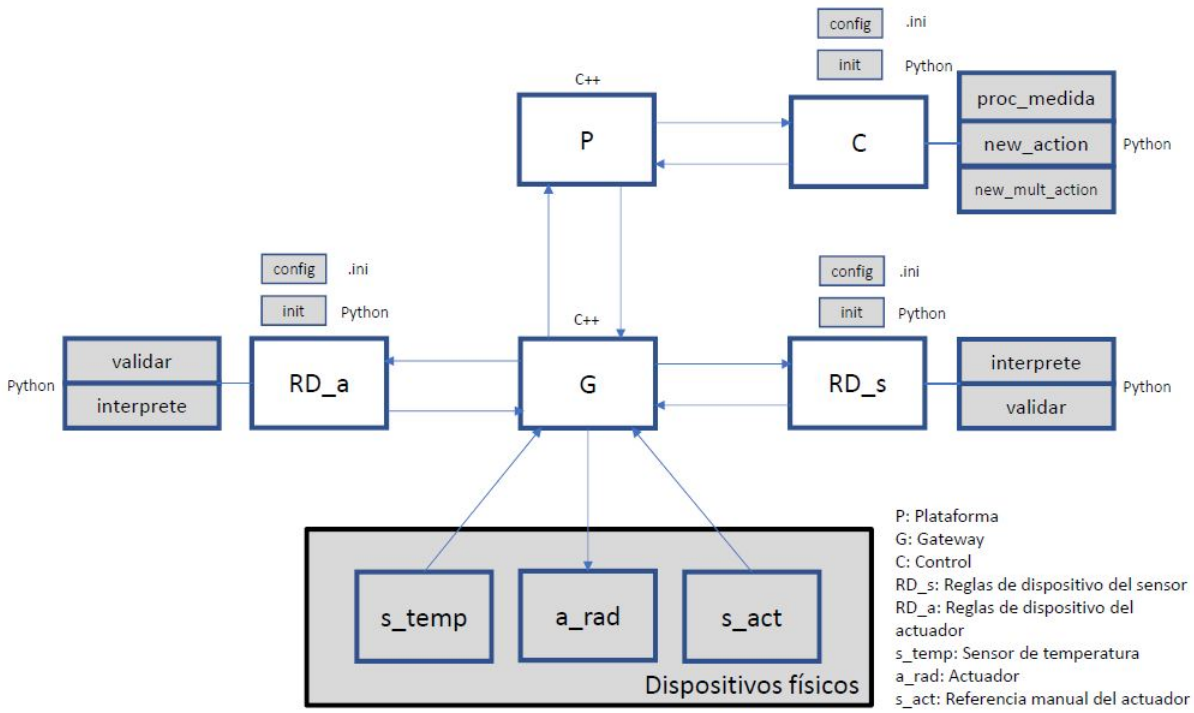

Figura 4: General arquitecture of the system.

The proposed arquitecture counts with the presence of a gateway, in charge of managing the sensor and the actuator, and of a platform, which manages the gateway and

the control. The sensor, control and actuator have different functions, which have to be implemented by the user in Python, as shown in Figure 4.

The functions that have been implemented in Python are shown in the Figure 5. In addition to those functions, an initialization function of the sensor, control and actuator device rules is required. The interpreter function, present in the sensor and actuator rules, is used to pass data from the units of the devices to those used in the control system and vice versa. The validation functions, also present in the sensor and actuator, verify that both the sensor temperature measurement and the actuation value are consistent. The control rules incorporate three other functions that are used to manage the validated data coming from the sensor and to generate the corresponding actions to maintain the temperature within the comfort ranges of the user.

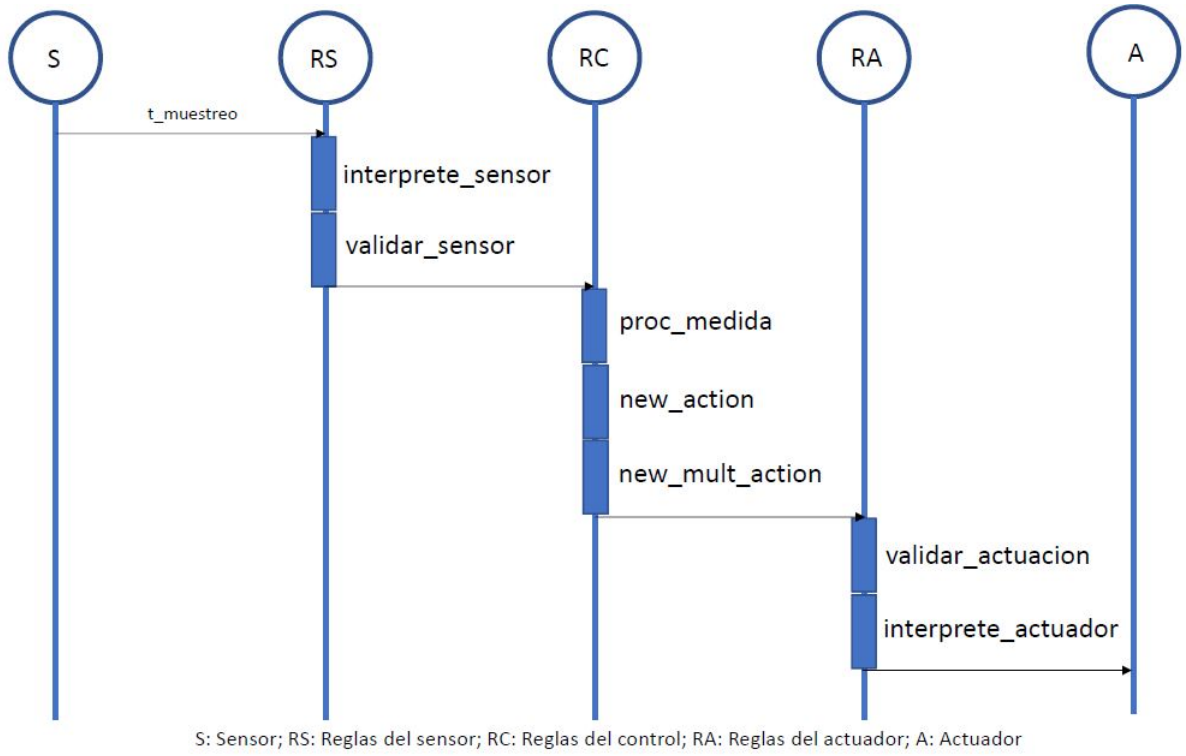

Figura 5: UML diagram with the temporal evolution of the functions for the normal operation of the application.

In order to implement the system, different aspects of programming have been required. Among them, the integration between C++ and Python, the creation of classes to manage the different devices, the creation of multiple execution threads in C++ and the reading of the configuration files.

Results

The system has been tested using two case studies. In the first one, there will be a sensor, a control and an actuator and temperature data is generated every 5 seconds. For

each value generated, it is validated by the sensor rules, it goes through the control to generate an actuation and the actuation is validated by the actuator rules. Figure 6 shows an example of the results. Check that for values between -20°C and 30°C, an action has been generated, as valid temperature data is considered. Also, only actuations between 20°C and 23°C are valid actuations, so the remaining ones are discarded.

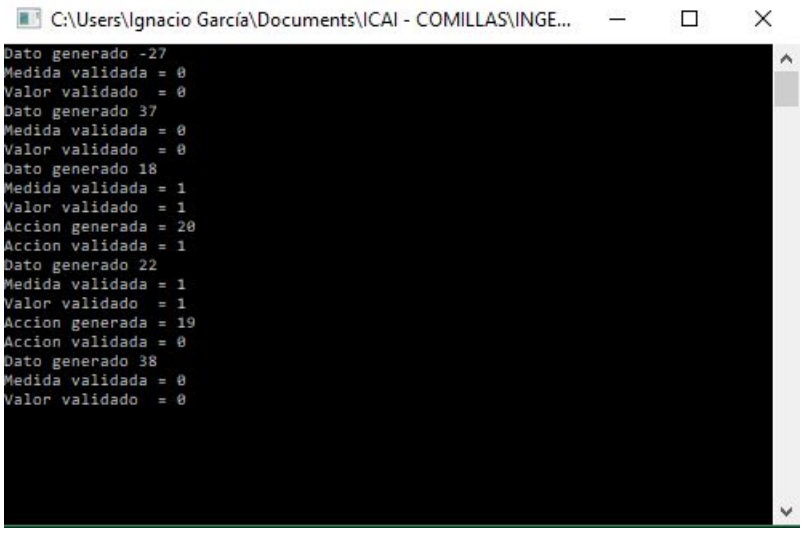

Figura 6: Example of the execution for a single thread.

In the second case study, three sensors and a single control were considered. All three sensors send their data to the existing control. Until all three sensors present a valid data, the control is not executed, which then generates an action. The validity of the actuation is checked by the actuator rules, as in the previous case study.

Conclusions

The main conclusions obtained in the development of this Master Thesis are:

- Controlling the temperature of a home in a flexible way, not limiting the number of devices that can be connected and allowing the user to establish their own comfort criteria, fits as a part of the electrical management platform that has been presented in [1].
- To control the temperature, a system consisting of a temperature sensor, a control and an actuator is proposed. There is a gateway that controls communications with the sensor and the actuator. In addition, a platform will manage the gateway and the control.
- The system created is flexible in two areas: multiple devices could be included and the user can modify the operation without modifying the rest of the system.

References

- M. Martín Lopo, J. Boal y A, Sánchez, "Transitioning from a meta-simulator to electrical applications: an arquitecture", Simulation Modelling Practice and Theory, vol. 94, p. 198, jul 2019.
- API Python/C, "docs.python.org/3/c-api/intro.html", visited 12/07/2019.

Índice general

Índice de figuras

Índice de tablas

Índice de código

Capítulo 1

Introducción

La llegada del Internet de las Cosas y las las mejoras y abaratamiento de las tecnologías de comunicación, junto a nuevas tecnologías de generación distribuida, permiten la aparición de nuevas formas de gestión de los recursos energéticos y de los hábitos de consumo. El concepto de casa domótica abarca todo esto y permite el control y la optimización de los comportamientos de las tecnologías de generación o las cargas presentes en el ámbito que se considere. La presencia o no de generación distribuida es opcional y las cargas presentes (iluminación, climatización, seguridad), tienen un funcionamiento parecido. Sin embargo, es necesario que exista un sistema flexible basado en reglas que permita establecer que hacer en cada situación y limitar las acciones disponibles. Esto es necesario tanto a nivel de cada sistema domótico individual, como si existieran sistemas de gestión comunes (agrupadores, comercializadoras).

Una forma de modelar el sistema eléctrico con el objetivo de su gestión es a través de metasimuladores, como se propone en [1]. Para entender lo que un metasimulador quiere alcanzar, se puede hacer uso de los metajuegos. La filosofía de los metajuegos es la creación de un "jugador" que pueda jugar a una gran variedad de juegos a partir de unas reglas generales de funcionamiento comunes a todos ellos. Dichas reglas generales se basan en los recursos de los que dispone un jugador en la partida: bienes materiales, información disponible o interacción con otros jugadores [4].

En el lado opuesto se encuentra la creación de un programa para jugar a un determinado juego concreto al que se le puede añadir información adicional a las reglas proveniente de la experiencia del programador en el juego. La diferencia se ilustra en la Figura 1.1. En la parte izquierda, donde se ilustra al programador que conoce las reglas del juego y tiene experiencia jugando, que programa para que el ordenador sepa jugar a dicho juego. Por su parte, el esquema de la derecha muestra al programador que conoce las reglas generales por las que se rigen los juegos y que le permite programar a un metajugador que podrá jugar a los diferentes juegos [3].

Adicionalmente, es importante reseñar el papel que tienen los sistemas expertos en el desarrollo de un metasimulador. En los sistemas expertos se implementan reglas generales

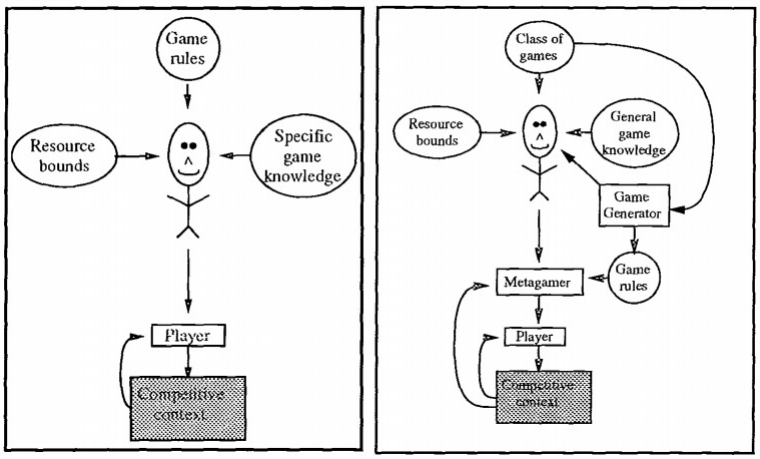

Figura 1.1: Comparación entre jugador de un único juego y metajugador [3].

que permiten controlar el funcionamiento. Los sistemas expertos son sistemas computacionales que emulan la capacidad de tomar decisiones de un humano. Están compuestos de tres estructuras: la base de conocimiento, donde se especifican las reglas y los hechos, el motor de inferencia, que permite obtener nueva información aplicando los hechos a las reglas y la interfaz de usuario, que permite al programador tomar decisiones.

El proceso de diseño de un metasimulador comienza identificando los diferentes elementos existentes en la realidad que se quiere emular o controlar. En este caso, un sistema global de gestión del sistema eléctrico dispone de características comunes entre los elementos que lo componen que se deben abstraer [1].

Con esta perspectiva planteada, los sistemas domóticos presentan características idóneas para que puedan ser controlados mediante sistemas expertos, de forma similar al funcionamiento de los metajuegos, estableciendo reglas generales de funcionamiento para cualquier situación que pueda darse en el sector eléctrico y que se particularizan para casos de uso concretos, como puede ser la gestión energética de un prosumidor con diferentes cargas, generación distribuida o sensores de medida. No obstante, se presentan retos en el desarrollo de plataformas de energía que deben ser resueltos para una correcta implementación. Entre ellos, la escalabilidad de las soluciones propuestas ya que el sistema debe ser capaz de gestionar millones de dispositivos y la información que van generando o problemas de seguridad y de acceso a los datos de los usuarios.

Capítulo 2

Revisión del estado del arte

2.1. Plataforma de gestión del sistema eléctrico

Con el objetivo de crear el metasimulador que permita controlar el sistema eléctrico, en [1] se presenta una arquitectura que permite integrar el Master Kernel (sistema central), las diferentes plataformas y sus agentes, la interfaz de usuario, las reglas de comportamiento y el almacenamiento de datos. En el Master Kernel se controla todo el sistema y debe comunicarse con las diferentes plataformas que existen, además de controlar los tiempos de computación, como se muestra en la Figura 2.1.

El Master Kernel se encuentra en el centro de la arquitectura y todo se encuentra directa o indirectamente conectado a él. De este modo tiene acceso a la información de todo el sistema. Se encuentra sombreado, indicando que debe estar en manos del desarrollador del sistema y que deberá ser incluido en cualquier implementación que se haga de este, aunque no sean necesarios en alguna de ellas y no siendo modificable por el usuario final. El resto de bloques sombreados también deben ser definidos por el desarrollador. En blanco, por el contrario, aparecen los bloques que deben ser introducidos por el usuario.

Existe una justificación para aceptar que debe haber bloques que no puedan ser modificados por el usuario y otros que, por el contrario, deben ser especificados por él. Por ejemplo, el desarrollador del sistema debe especificar las plataformas que existen. Por ejemplo, la plataforma de generación. Los agentes de una plataforma de generación son las diferentes empresas de generación. El usuario tendrá libertad para seleccionar el número de agentes (número de unidades de generación) que existen en la realidad que está modelando.

Tal y como se muestra en la Figura 2.1, el Master Kernel se gestiona gracias a unas reglas de comportamiento general que deben ser introducidas por el usuario. Estas reglas permiten al Master Kernel saber que tipos de plataformas pueden conectarse y que puede pedirles a ellas. Otra de sus funcionalidades es la de comunicar diferentes plataformas. Por ejemplo, la plataforma de generación comunica al Master Kernel las ofertas de sus agentes y este las hace llegar al operador del mercado, que resuelve el mercado y comunica

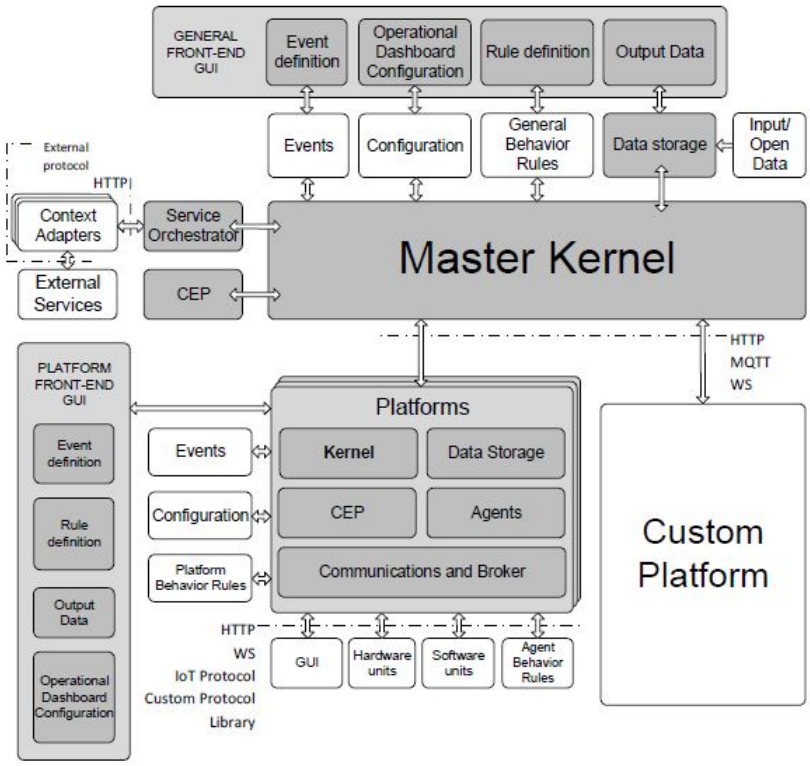

Figura 2.1: Arquitectura del sistema eléctrico propuesta [1].

a la plataforma de generación los resultados vía el Master Kernel.

El siguiente elemento que aparece siguiendo el orden jerárquico de la arquitectura que se propone en [1] son las plataformas. Las plataformas se utilizan para coordinar y gestionar los diferentes agentes que componen el sistema eléctrico. Las plataformas se identifican con las principales funcionalidades del sistema eléctrico (generación, operador del sistema, operador del mercado, regulador). Puede existir más de una plataforma para cada función, como en el caso de que exista más de una compañía generadora, aunque es común que las funcionalidades estén representadas por una única plataforma, como en el caso de la plataforma del operador del mercado. Cada plataforma puede ser responsable de más de un agente como podría ser el caso de la plataforma de transmisión que tendría como agentes al operador de la red y dueño de la red de transporte.

La estructura seguida entre el Master Kernel y las plataformas se replica entre la plataforma y los diferentes agentes que la componen. Así, la plataforma se encarga de enviarle al Kernel los mensajes de los agentes. Las plataformas se encuentran controladas mediante dos tipos de reglas: las de comportamiento general, que como se ha comentado, especifican los tipos de plataformas y las características que deben de tener y las de comportamiento de plataforma, que determinan cómo opera la plataforma y que varían de plataforma a plataforma aunque están sean del mismo tipo. Ambas reglas deben ser especificadas por el usuario. El Master Kernel permite la conexión de diferentes tipos de plataformas y de varias plataformas del mismo tipo.

Los agentes aparecen dentro de las plataformas y son los encargados de realizar las diferentes tareas. Por ejemplo, en las plataformas de generación los agentes serían los encargados de la producción eléctrica o de determinar cuánta electricidad se va a generar. Los agentes pueden venir definidos o pueden ser definidos por el usuario si fuera necesario. No obstante, en ambos casos, el comportamiento de los agentes debe ser el esperado por las reglas de la plataforma.

Los agentes se encuentran definidos por cuatro bloques, los que se conectan por la parte baja al bloque de las plataformas en la Figura 2.1. Estos son: la interfaz de usuario, unidades de hardware, unidades de software y las reglas de comportamiento de los agentes. La interfaz de usuario la pueden utilizar los usuarios del sistema para interactuar con el proceso. Las unidades de software buscan simular el comportamiento de infraestructuras o sistemas mientras que las unidades de hardware interaccionan con el entorno.

Las diferentes partes del esquema de la Figura 2.1 que se han explicado funcionan y se coordinan gracias a reglas, como se ha hecho mención. Las reglas permiten que los agentes puedan realizar una determinada transformación, una transacción con otro agente son las reglas de comportamiento o ser coordinados por una plataforma. Se pueden identificar tres tipos de reglas dependiendo que buscan controlar:

- Reglas de primer nivel: describen las relaciones que existen entre el Master Kernel, las plataformas y los agentes. Es decir, que características debe tener una plataforma para poder conectarse al kernel o que tipos de agente pueden existir dentro de una plataforma determinada y cuáles son sus características para que puedan ser identificados como tal. Estas reglas informan al Master Kernel sobre cuándo deben llamar a las diferentes plataformas para que realicen una determinada función.

- Reglas de plataforma: permiten a cada plataforma saber cuando deben llamar a los agentes que tiene para que realicen una determinada tarea. Por ejemplo, la plataforma encargada de operar el sistema ha recibido del Master Kernel una petición para realizar un flujo de cargas para conocer el estado de la red. La plataforma en cuestión dispondrá de uno o varios agentes que realizarán una serie de tareas para que la plataforma pueda reportar al Master Kernel de vuelta los resultados.

- Reglas de agentes: permite a los agentes hacer uso de las unidades de las que dispone para realizar las tareas que le han sido enviadas desde la plataforma. Siguiendo el mismo ejemplo, el operador del sistema deberá disponer de una unidad que pueda ejecutar flujos de carga que será la encargada de realizarlo y reportar los resultados a la plataforma.

Por último, en [1] proponen aplicar la arquitectura que se ha explicado para el sistema eléctrico español, considerando todas las entidades que participan en él. En total, existirán siete plataformas, cada una de ellas realizando una función específica. En el caso que

concierne a este Trabajo Fin de Máster, la plataforma de los consumidores finales incorpora un único agente, los prosumidores, que tienen asociados una gran lista de unidades (generación renovable, *energy box*, contadores inteligentes, carga, baterías y sensores y actuadores), que pueden poseer o no. De esta forma es más fácil abstraer la idea de un consumidor final, independientemente de las unidades que tenga implementadas.

2.2. Sistemas de integración de reglas

En la literatura consultada, existen ya sistemas expertos aplicados a diferentes procesos en campos tan variados como el sector de la energía o la medicina. Cómo se comentó previamente, los sistemas expertos con base de conocimiento se utilizan para emular la toma de decisiones de los humanos y es este hecho el que provoca que, independientemente del campo de aplicación, tengan características similares que permiten comparar los diferentes sistemas expertos.

Una de las aplicaciones que más interés suscitan es las de sistemas expertos en el sector eléctrico. [5] realiza una revisión bibliográfica de sistemas expertos a aplicados a problemas en plantas de generación eléctrica. El objetivo que persigue la incorporación de sistemas expertos para controlar plantas de generación es el de reducir la cantidad de trabajo de operadores y expertos.

Los sistemas expertos que se proponen en [5] se pueden clasificar en grupos en función de su cometido. Se identifican cuatro grandes grupos:

- Diagnóstico de faltas: los sistemas expertos le da directrices a los operarios para diagnosticar la falta y explica como las acciones correctivas se tienen que ejecutar para recuperar el sistema lo más rápido posible [6], [7], [8].
- Apoyo a operarios: diseño de plantas, programación horaria de plantas, reducción de sobrecargas y predicción de demanda y procesado de datos y análisis [9].
- Procesamiento de alarmas [10]
- Entrenamiento de operadores [11], [12], [13].

Junto a la clasificación anterior, para cada tipo de aplicación de los sistemas expertos, existen diferentes tipos de metodologías para la ejecución del motor de inferencia:

- Sistema experto basado en reglas: El conocimiento se almacena en forma de reglas *if-then* de las que se extraerá conocimiento mediante razonamiento delantero, trasero o híbrido para alcanzar las conclusiones. La calidad de la información extraída dependerá de la cantidad de información contenida en la base de conocimiento [9], [11], [12], [14].

- Sistema experto borroso: Para que un sistema basado en reglas tenga un comportamiento satisfactorio es necesario que toda la información se encuentra en la base de conocimiento. Como esto es difícil de alcanzar y pueden existir situaciones en las que el sistema experto basado en reglas no sea capaz de extraer ninguna conclusión válida, se plantean los sistemas expertos borrosos. A cada regla se le asigna un factor de certeza [6], [13], [15].

- Sistema experto híbrido: Este tipo de sistemas incorporan, además de un sistema experto, otros tipos de mecanismos para aportar información al sistema. Por ejemplo, se pueden incorporar redes neuronales, lógica borrosa ó sistemas de recolección de datos en tiempo real (SCADA) [7], [8], [16], [17].

Aunque los sistemas expertos basados en reglas permiten deducir grandes cantidades de información, encuentran una especial limitación cuando la información disponible para su análisis no es discreta. Por ejemplo, en un sistema experto que reconoce animales, se ha incorporado una regla que determina que un animal de color verde debe ser identificado como una rana. Sin embargo, si el color de la rana tiene tonalidades de verde o la lectura del color no es exacta, se descarta que sea una rana. En un caso como este, tiene más sentido utilizar sistemas expertos borrosos, que ante estas situaciones, te ofrecen la probabilidad de que el animal sea una rana y no es un sistema todo o nada.

Considerando los sistemas híbridos, en la literatura aparecen sistemas que combinan redes neuronales y un sistema experto basado en reglas. De forma general, las redes neuronales pueden utilizarse para modelar el proceso y detectar anomalías en el normal desarrollo de este [16]. Una vez que las faltas están localizadas, el sistema experto analiza los resultados y los confirma u ofrece una alternativa. La principal limitación de esta combinación se encuentra si la falta localizada por las redes neuronales está fuera del límite de conocimiento del sistema experto y, por lo tanto, no tiene reglas específicas para dicha situación.

Esta combinación de redes neuronales y sistemas expertos se presenta en [17] donde se busca crear un sistema que detecte fallos en la planta antes que el sistema tradicional de alertas. De esto se encargan las redes neuronales, que monitorizan las desviaciones entre los valores reales de la planta y los predichos por las propias redes neuronales. El sistema experto aparece a continuación, toma información de las redes neuronales y trata rápidamente grandes cantidades de información que permiten tomar decisiones en entornos complejos.

2.3. Lenguajes de programación

Para desarrollar este Trabajo Fin de Máster se ha utilizado tanto C++ como Python, integrándose entre ellos. Se llama a funciones escritas en Python desde el código en C++.

La integración de código de Python en C++ se debe a un interés para que usuarios avanzados realicen o cambien partes de un programa sin tener acceso a su totalidad o si se desea utilizar alguna funcionalidad de Python que es difícil o costoso implementar en C++. Python proporciona una forma rápida de escribir código de forma flexible. La llamada de funciones de Python desde C++ se denomina *embedding* mientras que el proceso contrario sería *extending*.

C++ y Python tienen características diferentes que hacen interesante la integración que se propone. Mientras que C++ se implementa como un lenguaje compilado y se necesita un compilador de C++, Python es un lenguaje interpretado de alto nivel. Por ejemplo, un programa en C++ debe ser compilado en cada sistema operativo en el que se desee ejecutar mientras que Python se puede escribir una vez el código y ejecutarlo en todos los dispositivos que tengan Python instalado. C++ también soporta algunas funcionalidades de los lenguajes de alto nivel, es un lenguaje de programación orientado a objetos, permitiendo el concepto de clases.

Para integrar C++ y Python se hará uso de la "Application Programmers Interface Python/C" que incluye las diferentes funciones necesarias para iniciar y finalizar el intérprete de Python. También permite llamar a funciones y clases de Python y toda las funciones necesarias para integrar ambos lenguajes [2].

2.4. Motivación

Con la idea del sistema flexible que permita controlar diferentes fenómenos del sector eléctrico, la principal motivación del trabajo es la de obtener dicho sistema que permita la incorporación de diferentes funcionalidades (sensor de medida, sistema de generación) sin tener que modificar ni parar la ejecución principal del programa. De este modo, se pueden analizar diferentes casos de uso para comprobar el rendimiento de la herramienta creada.

2.5. Objetivos

El realización de este Trabajo Fin de Máster abarca el cumplimiento de los siguientes objetivos:

- Desarrollar un sistema flexible para controlar la temperatura basado en reglas de comportamiento asignadas a las diferentes unidades que forman parte de dicho sistema. El sistema deberá ser capaz de integrar múltiples dispositivos y diferentes tipos de control.
- Permitir que el usuario pueda indicar sus criterios para controlar la temperatura, y modificarlos a lo largo de la ejecución, sin que esto afecte al resto del sistema.

Capítulo 3

Diseño conceptual y tecnológico

En el Capítulo 2 se ha presentado una plataforma de gestión del sistema eléctrico. La Figura 2.1 ilustra las relaciones existentes entre los diferentes elementos que deben existir para que pueda crearse y funcionar dicha plataforma. El objetivo de este Trabajo Fin de Master es diseñar una parte de dicha plataforma, para desarrollar un sistema flexible que personalice el comportamiento de una vivienda inteligente. En concreto, se pretende crear un sistema con dos tipos de dispositivos (sensores de temperatura y actuadores, que puede ser un radiador o aire acondicionado) que se comporte de manera flexible.

El desarrollo se ha realizado asemejándose a lo propuesto por [1] en la mencionada arquitectura. Si se enfoca el problema de abajo hacia arriba, los dos tipos de dispositivos que se han utilizado, sensores y actuadores, se consideran unidades, que tienen sus propias reglas de comportamiento. Estas dos unidades pertenecen a un determinado agente, que tal y como se definió en el Capítulo 2, es *prosumer*, que era el único agente que pertenecía a la plataforma de los usuarios finales, que es donde con una perspectiva más global se encuentran los sistemas domóticos que pretenden personalizar el comportamiento de una vivienda inteligente.

Para el diseño de la interacción entre los sensores y actuadores es necesario mantener la misma filosofía seguida en el resto del sistema. Es decir, se busca aumentar los niveles de flexibilidad en dos aspectos: poder conectar sensores y actuadores con diferentes características técnicas y permitir diferentes tipos de control, para satisfacer las necesidades de confort de diferentes usuarios. Para ello, es necesario que el usuario proporcione información sobre sus requisitos y las características técnicas de dichos dispositivos. Por tanto, parte del desarrollo se le deja al usuario, aunque de forma intuitiva. Desde un punto de vista operativo, autorizar al usuario a realizar cambios debe realizarse siempre que no interfiera en el normal funcionamiento del resto de la plataforma. Es decir, si un usuario pretende realizar algún cambio (añadir algún sensor o actuador o cambiar los criterios de control), esto debería afectar de forma mínima al resto de la plataforma.

3.1. Arquitectura propuesta

Para el problema planteado, se presenta en la Figura 3.1 la arquitectura propuesta. En ella se integran los sensores y actuadores (parte física), con las reglas de comportamiento de dichos dispositivos. Existe un *gateway* encargado de lidiar con dichos dispositivos: recibe las medidas desde los sensores y se encarga de enviar las señales de actuación al radiador. Además, tiene comunicación con la plataforma, para enviarle la información necesaria para ejecutar el control y recibir los resultados de este. Por último el control, se encarga de procesar la información de temperatura y decidir, según los criterios de confort del usuario, que acción tomar.

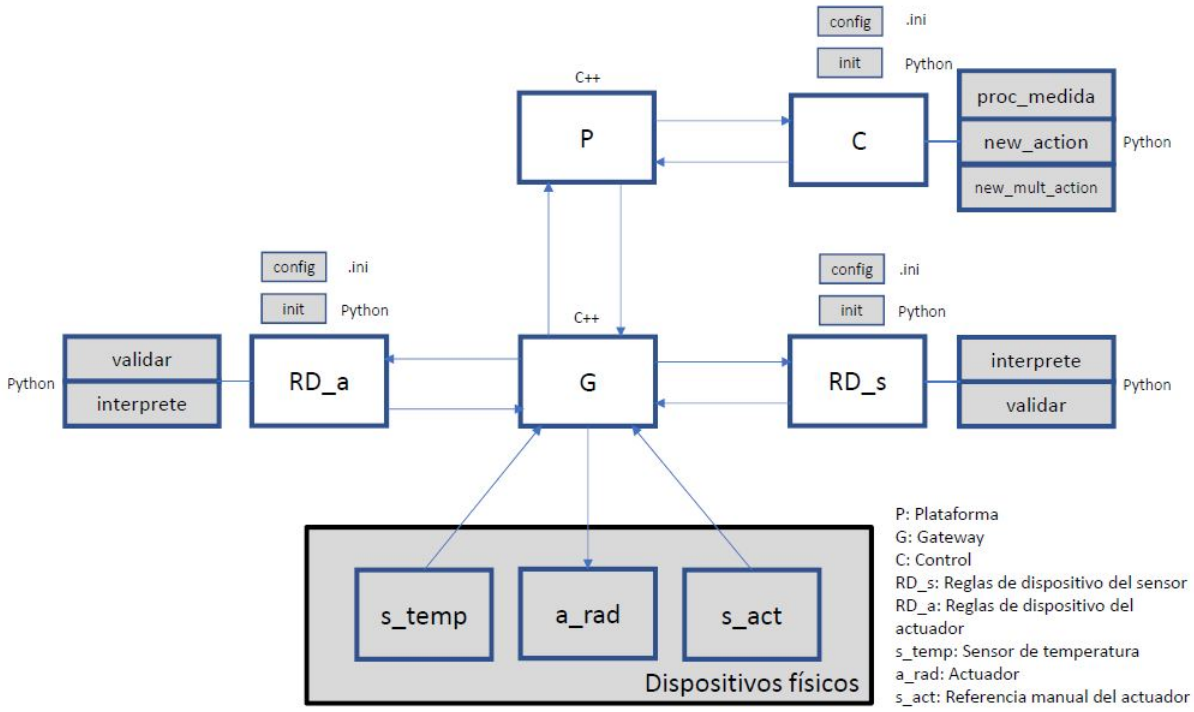

Figura 3.1: Esquema general de funcionamiento.

Los principales bloques funcionales presentes en la Figura 3.1 se enumeran a continuación:

- Dispositivos físicos: se consideran hasta tres dispositivos físicos.
  - Sensor de temperatura: detecta la magnitud física, la temperatura en este caso, y genera una señal que incluye la variable, en una determinada unidad de medida y otro tipo de información, como un identificador del propio sensor o la información horaria en la que se generó la medida.
  - Actuador: recibe la orden del control y ajusta la señal de temperatura del radiador para cambiar la temperatura de la ubicación donde se encuentra.

    - Sensor en el actuador: junto a las referencias de temperatura que le llegan desde el control al actuador, el usuario puede tener la posibilidad de indicarle al radiador o aire acondicionado la temperatura a la que quiere que funcione durante un tiempo concreto. Estas acciones del usuario deben ser tenidas en cuenta.

- *Gateway* (G): es la primera capa de gestión de los dispositivos. A su cargo se encuentran los diferentes dispositivos físicos y las reglas de dispositivo. Es el responsable de las comunicaciones directas con los dispositivos.

- Reglas de dispositivo del sensor (RD_s): Reciben los datos del sensor desde el *gateway* y le devuelven el valor procesado, comprobando que se trata de un dato válido y en las unidades correctas con las que trabaja la plataforma.

- Reglas de dispositivo del actuador (RD_a): se utilizan para enviar la señal de la actuación al dispositivo final, garantizando que es una actuación válida y que será entendida por el dispositivo físico.

- Control (C): incluye las reglas que permiten gestionar las medidas que llegan desde los dispositivos y generar las correspondientes actuaciones, puntuales o una planificación de ellas.

- Plataforma (P): es la capa de gestión más alta de la interacción que se está diseñando. Actúa de mensajero entre el control y los dispositivos y se utilizaría también par almacenar en la base de datos toda la información relevante.

3.2. Definición de las funciones

Para que los diferentes bloques cumplan con las tareas que se le han asignado, es necesario implementar diferentes funciones en ellos. La Figura 3.2 muestra las diferentes funciones, el bloque en el que se implementan y la evolución temporal en la que se procesan, si atendemos a la lógica de que una vez que aparece un dato nuevo, este pasa por las reglas del sensor, va a las reglas del controlador, que genera una actuación que, por último, recibe el actuador. No obstante, es posible que la generación de actuaciones no esté ligada a que el sensor haya enviado un nuevo dato si no que el control se ejecute cada cierto período de tiempo. Una vez transcurrido ese tiempo, tomará las últimas medidas de temperaturas validadas por el sensor y generará una actuación. Ya se elija una u otra opción, las funciones que hay que implementar son las mismas: en las reglas del sensor, hay que incorporar un intérprete y una función de validación. En el control se procesa la medida y existen otras dos funciones para generar actuaciones. Por último, en las reglas del actuador, se verifica que la actuación se encuentra dentro de los parámetros del actuador

al que se está enviando y la medida debe ser interpretada para que esté en las mismas unidades que el dispositivo físico. La explicación de dichas funciones se muestra en 10 tablas (Tabla 3.1 a 3.10).

Como se ha hecho mención al inicio del capítulo, parte de la flexibilidad que aporta el sistema es a costa de que el usuario deba participar en la definición de determinadas funciones. Esto, sin embargo, no deja de ser lógico, pues es para el usuario final para quien se personaliza el sistema. Por tanto, como se ilustra en la Figura 3.1, el usuario es el encargado de implementar los bloques sombreados, mientras que el desarrollador se encarga de los bloques blancos. Continuando con la lógica de abajo hacia arriba, el usuario es el encargado de elegir que dispositivos físicos quiere conectar a sus sistema. Los desarrolladores incorporan una lista de dispositivos típicos para los cuales su configuración en el sistema será inmediata, pero el usuario tiene libertad para añadir otros que considere pero es el encargado de su configuración. El usuario no se encarga de la creación del *gateway* ni de la gestión que este hace de las reglas de dispositivo (tanto del sensor como del actuador), lo que es parte del trabajo de los desarrolladores. Sí que debe incluir las funciones que se llaman desde estas reglas y que se especifican a continuación. es también responsabilidad de los desarrolladores gestionar las comunicaciones entre el *gateway* y la plataforma y entre esta y el control. Por último, de forma similar a lo que ocurría para las reglas de dispositivo, el usuario es el encargado de especificar las funciones que se usan en el control.

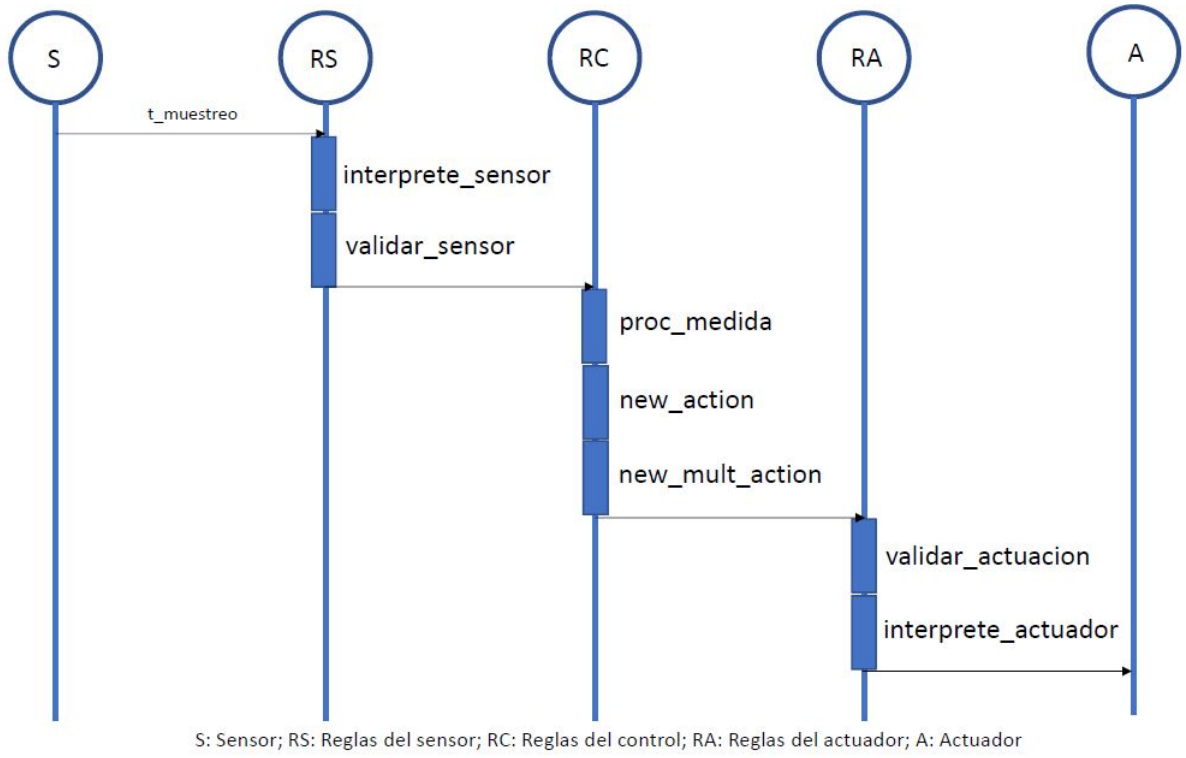

Figura 3.2: Diagrama UML con la evolución temporal de las funciones para el funcionamiento normal de la aplicación.

Tabla 3.1: Descripción de la función `init_sensor`.

Nombre	`init_sensor`
Pertenece a	Reglas del sensor
Entradas	Ninguna, se ejecuta al iniciar el sistema y no necesita ningún argumento para ejecutarse.
Salidas	Nombre de fichero de configuración.
Descripción	La función de inicialización incorpora el nombre del archivo de configuración que tiene los parámetros de funcionamiento del sensor para que el *gateway* pueda conocerlos y, de este modo, pasarle los argumentos necesarios a las distintas funciones del sensor. Además, puede servir para realizar comprobaciones de funcionamiento del sensor.
Trigger	Inicialización del sistema.
Posibles errores	En primer lugar, que no exista información sobre el archivo de configuración que tiene que enviarle al *gateway* y que no se pueda realizar la correcta configuración del sensor.

El *gateway* debe disponer de la información característica de cada uno de los sensores que estén conectados a él. Cuando se quiere conectar un nuevo sensor, esté tiene que tener sus características especificadas en un archivo de configuración que es leído por el *gateway* durante el proceso de inicialización. Como se ha comentado en la Tabla 3.1, la ruta a dicho archivo se especifica en la función de inicialización del sensor, que tiene que ser creada por el usuario. El archivo de configuración, que tiene formato `.ini` incluye los parámetros y la información necesaria para controlar el comportamiento del sensor. Entre estos parámetros se pueden incluir el tipo de sensor que se conecta, un parámetro único identificador del sensor, la ubicación del mismo, el agente al que se va a conectar, el intérprete que debe usar y el tiempo de muestreo, uno o varios, que va a utilizar, entre otros.

Tabla 3.2: Descripción de la función `interprete_sensor`.

Nombre	`interprete_sensor`
Pertenece a	Reglas del sensor
Entradas	Variable de temperatura medida por el sensor en las unidades en las que trabaja el sensor.
Salidas	Variable de temperatura en las unidades en las que trabaja el sistema.
Descripción	Transforma la variable medida a las unidades características de la plataforma para homogeneizar los datos adquiridos de los diferentes sensores.
Trigger	Cada vez que llega un dato a las reglas del dispositivo del sensor.
Posibles errores	La información que llegue esté defectuosa, que no se pueda llamar al intérprete o que no exista.

La presencia de un intérprete es necesaria para garantizar que la plataforma y el control reciben todos los datos en las mismas unidades. La función que controla esto se muestra en la Tabla 3.2. Un ejemplo que muestra esto, es el caso de un control que toma las medidas de dos sensores diferentes. Sin embargo, uno le está enviando las medidas en grados Celsius y el otro en grados Fahrenheit. Aunque el sistema se puede desarrollar para que el control pueda trabajar con datos en diferentes unidades, es una solución difícilmente escalable. Lógicamente, se debe aceptar que dispositivos que trabajan en diferentes unidades se conecten al sistema. El procedimiento más simple para que todo funcione bien es que los datos se pasen a las unidades en las que trabajan el *gateway* y la plataforma antes de acceder a ellas (en el caso del sensor) y se vuelva a pasar a las unidades de los dispositivos antes de salir de ellas (en el caso del actuador).

Tabla 3.3: Descripción de la función `validar_sensor`.

Nombre	`validar_sensor`
Pertenece a	Reglas del sensor
Entradas	El dato que ha sido intepretado y ya se encuentra en lenguaje de la plataforma.
Salidas	El mismo dato que ha entrado si ha sido validado o un mensaje de error si no ha sido validado.
Descripción	Sobre los datos que llegan del sensor se aplican una serie de reglas para comprobar que son coherentes. Entre las comprobaciones que se pueden hacer: que el tiempo de la medida sea coherente, que se encuentre dentro de límites o que no existan variaciones muy grandes entre medidas consecutivas.
Trigger	Cuando llega un dato del intérprete.
Posibles errores	Todos aquellos que hacen que la medida no sea validada.

La Tabla 3.3 muestra la función más importante de las reglas del sensor. En la función se comprueba la validez y la coherencia de las medidas que llegan desde el dispositivo físico. Los criterios sobre los que se hacen las comprobaciones deben venir incluidos en el fichero de configuración del sensor, que se especifica en la función de inicialización. Por ejemplo, se deben especificar los límites mínimo y máximo de temperatura que el sensor acepta y, una medida fuera de esos rangos, no es considerada válida. También es posible considerar una medida cómo errónea si difiere mucho de la anterior. Por tanto, se puede establecer un criterio de rampas, que invaliden dicha medida. Estos tres parámetros, temperatura máxima y mínima y la máxima variación entre dos medidas consecutivas deben venir especificadas desde el archivo de configuración. También es posible comprobar la coherencia de la hora de la medida para asegurar que es posterior a la de la última medida validada, para comprobar que es un dato actual y no se encuentra dañado. Por último, y también en la función de validación, es posible añadir elementos de control y verificación de orden superior: enviar un código de error si no se han procesado medidas desde hace un determinado tiempo o calcular el porcentaje de medidas que se validan y

enviar un error si no se están validando suficientes, lo cual puede indicar que el sensor no está funcionando correctamente.

Tabla 3.4: Descripción de la función `init_control`.

Nombre	`init_control`
Pertenece a	Reglas del control
Entradas	Se ejecuta al iniciar el sistema y no tiene ningún argumento de entrada.
Salidas	Nombre del fichero de configuración del control.
Descripción	La función de inicialización del control sirve para que la plataforma conozca el fichero de configuración de cada control que se conecte a él para así poder gestionarlo.
Trigger	Se ejecuta al iniciar el sistema.
Posibles errores	Que no exista el fichero de configuración.

De forma similar a lo explicado para la función de inicialización del sensor (Tabla 3.1), la función de inicialización del control se explica en la Tabla 3.4. Además de inicializar el control, aporta la información necesaria para que se ejecute correctamente, a través del archivo de configuración del control. Lo primero que habrá que indicar es el tipo de control que se realizará. Es decir, que decisiones se van a tomar a partir de los datos de temperatura que se reciban de los sensores asociados. Por ejemplo, se puede realizar un control todo o nada, que a partir de una temperatura se inicie el actuador. También, el control puede permitir enviar referencias de temperatura al actuador y en función de la temperatura medida, se genera una referencia u otra para este. Para el primer caso, sería necesario que el archivo de configuración incluya la temperatura medida por el sensor a partir de la cuál se actuará mientras que para el segundo caso será necesario incluir el rango de temperaturas que se enviarán al actuador. Estos dos tipos de control además se podrán ejecutar cada vez que llegue una medida del sensor o cada cierto tiempo, considerando las medidas que han llegado desde la última actuación. En este último caso, el archivo de configuración deberá incluir también la frecuencia con la que se realizará el control. Independientemente de los sensores que existan, será necesario que el archivo de configuración indique cuáles son aquellos sensores que se van a controlar con el control al que corresponde el archivo de inicialización. Por último, y cómo se explica en las Tablas 3.6 y 3.7, se tiene que indicar si el control puede generar una única actuación o una programación de actuaciones.

Tabla 3.5: Descripción de la función `proc_medida`.

Nombre	`proc_medida`
Pertenece a	Reglas del control
Entradas	El dato validado por la función de validación del sensor.
Salidas	En función del control que se realice: nada o la medida o conjunto de últimas medidas para calcular la actuación a realizar a posteriori.
Descripción	Es una función destinada a la gestión de los datos validados y que su funcionalidad dependerá del tipo de control que se ejecute. Envia a las funciones `new_action` y `new_mult_action` diferentes argumentos de entrada.
Trigger	Cada vez que llega un dato de la plataforma.
Posibles errores	Ninguno

Dentro de las funciones del control se ha incorporado una que permite procesar los datos validados recibidos del sensor. Su funcionalidad depende del tipo de control que se esté utilizando. Si el control se realiza cada vez que llega una medida y se genera una referencia, la función no se necesitaría. Por el contrario, si el control se realiza cada un cierto periodo de tiempo, la función de procesar medida realizará los cálculos necesarios con las medidas que vayan llegando para pasarle los argumentos necesarios a las funciones que generan las actuaciones.

Tabla 3.6: Descripción de la función `new_action`.

Nombre	`new_action`
Pertenece a	Reglas del control
Entradas	Los valores del sensor, procesados o no por proc_medida.
Salidas	Una actuación única para el actuador.
Descripción	Esta función permite generar una actuación y que llegue al actuador para que se ejecute directamente o en el siguiente periodo de ejecución.
Trigger	En función del control que se está ejecutando: cuando se termine la función `proc_medida` o cuando haya pasado el tiempo indicado desde la última ejecución.
Posibles errores	No se genere ninguna actuación.

Tabla 3.7: Descripción de la función `new_mult_action`.

Nombre	`new_mult_action`
Pertenece a	Reglas del control
Entradas	Los valores del sensor, procesados o no por proc_medida.
Salidas	Una programación de actuaciones para el actuador.
Descripción	El control puede enviarle al actuador una programación de diferentes actuaciones para los próximos periodos.
Trigger	En función del control que se está ejecutando: cuando se termine la función `proc_medida` o cuando haya pasado el tiempo indicado desde la última ejecución.
Posibles errores	No se genere ninguna actuación.

Las funciones `new_action` y `new_mult_action` permiten enviar al actuador las diferentes referencias de temperatura que el control considera que debe ejecutar. La existencia de dos funciones proviene del interés que puede existir en determinadas circunstancias de enviar una programación de actuaciones al actuador. Las razones para que pueda interesar el disponer de una programación de actuaciones son diversas. Por ejemplo, si la diferencia entre la temperatura actual y la que se quiere alcanzar es muy grande, se puede realizar una programación de actuaciones para alcanzar gradualmente esa temperatura, para así además tener una mayor eficiencia energética. Otro caso podría ser que se va a desconectar el control y se envían por adelantado las referencias de temperatura para los siguientes períodos. Si se conocen los requerimientos de temperatura del usuario o se ha predicho la temperatura que hará a una determinada hora se pueden programar las referencias de temperatura para mantener el confort. Por último, mediante dichas programaciones se evitan cambios bruscos en las referencias de temperatura que se envían al actuador.

Tabla 3.8: Descripción de la función `init_actuador`.

Nombre	`init_actuador`
Pertenece a	Reglas del actuador
Entradas	Se ejecuta al iniciar el sistema y no tiene ningún argumento.
Salidas	Nombre del fichero de configuración del actuador.
Descripción	Permite que el *gateway* sepa donde se encuentra el archivo de configuración del control y pueda leerlo.
Trigger	Se ejecuta al iniciar el sistema.
Posibles errores	Que no exista el fichero de configuración.

Como ocurría con el sensor y el control, el archivo de configuración del actuador debe indicar la identificación del actuador que se está conectando, así como el control al que se está asociando y que le debe enviar las referencias de actuación. Al igual que ocurría con los sensores, habrá una serie de actuadores incluidos por defecto por el desarrollador, no obstante el usuario tendrá libertad para incluir el que considere. Como es posible que las unidades entre el *gateway* y el actuador sean diferentes, el archivo de inicialización deberá

indicar que intérprete hay que utilizar. En cuanto a la parte operativa, será necesario indicar también las temperaturas máximas y mínimas que soporta el actuador para garantizar que ninguna actuación las supera y también la diferencia máxima de temperatura entre dos actuaciones consecutivas. Por último, si la actuación no se ejecuta cada vez que llega una nueva medida, sino cada cierto periodo de tiempo, esto hay que indicarlo.

Tabla 3.9: Descripción de la función `validar_actuador`.

Nombre	`validar_actuador`
Pertenece a	Reglas del actuador
Entradas	La actuación o programación de actuaciones proveniente del control.
Salidas	La actuación que ha llegado si ha sido validada o un código de error en caso contrario.
Descripción	Las actuaciones que llegan del control deben ser validadas en el actuador antes de que sean ejecutadas para garantizar que cumplen con los requisitos técnicos de este.
Trigger	Cada vez que llega una actuación o programación de actuaciones del control.
Posibles errores	Que la actuación no sea válida por estar fuera de rango o porque varíe mucho de la actuación anterior.

La función de validación de las actuaciones permite establecer un filtro de las actuaciones que llegan desde el control que pueden no ser aceptadas por el actuador que se encuentra asociado a dicho control. La función se encarga de garantizar, mediante una temperatura máxima y mínima que ninguna actuación fuera de rango sea pasada al actuador. También se pueden incorporar comprobaciones adicionales, como las rampas entre dos actuaciones consecutivas o que el tiempo en el que se va a ejecutar la actuación es coherente con el tiempo actual. Otras validaciones de nivel superior se puede integrar, como comprobaciones de que el sistema está funcionando correctamente a través de indicadores como el porcentaje de actuaciones validadas o los tiempo sin validar ninguna actuación.

Tabla 3.10: Descripción de la función `interprete_actuador`.

Nombre	`interprete_actuador`
Pertenece a	Reglas del actuador
Entradas	Cada una de las actuaciones validadas en unidades del *gateway*.
Salidas	La actuación validada en unidades del actuador.
Descripción	Transforma el valor de la actuación de las unidades del *gateway* a las unidades propias del actuador.
Trigger	Cuando llega una actuación que se va a ejecutar.
Posibles errores	No se pueda llamar al intérprete o que no exista.

Por último, antes de que la actuación se envíe al actuador, tiene que pasar por el intérprete. Es necesario que tanto la plataforma como el *gateway* trabajen con unas misma unidades pero los actuadores que se conectan puede tener otras. Por tanto, es necesario pasar a estas últimas antes de que la actuación se ejecute.

3.3. Posibles errores en las funciones

Durante la explicación de las funciones se ha comentado que es posible que existan errores o que el flujo completo, desde el sensor hasta el actuador no se termine de completar. Esto puede deberse a que el valor de retorno de determinadas funciones sea un código de error o simplemente que no exista ningún valor de retorno. Las Figuras 3.3 a 3.5 muestran ejemplo de esto. El sensor envía una medida cada los segundos que marca el tiempo de muestreo y una vez que llega a las reglas de dispositivo del sensor, se comprueba que la medida es válida o no. Si fuera válida, seguiría según el diagrama de la Figura 3.2 pero, por el contrario, si la validación no fuera satisfactoria, el dato pasa a erróneo, generando un código de error que podría indicar la parte de la validación que no superó, cómo se explicó en la Tabla 3.3.

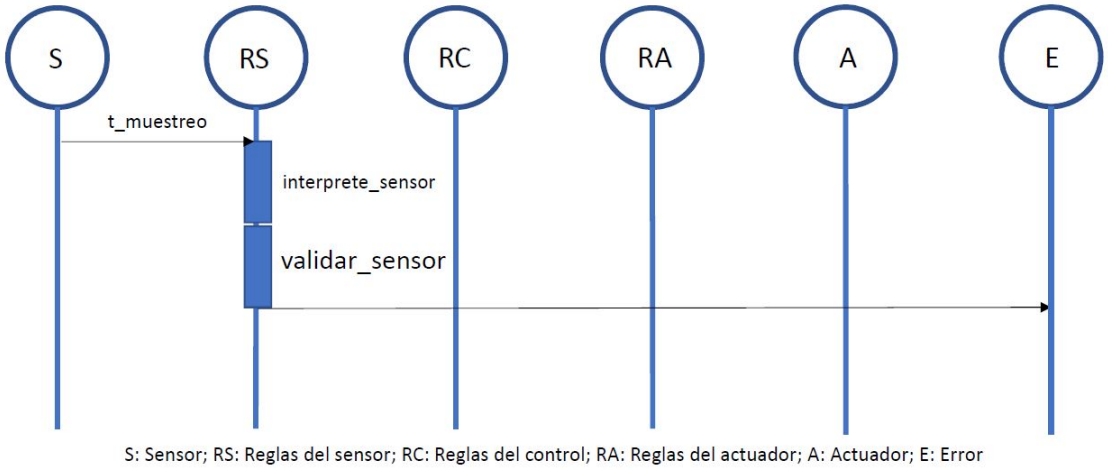

Figura 3.3: Diagrama UML con la evolución temporal de las funciones cuando el dato generado no es válido.

Por su parte, la Figura 3.4 muestra la gestión de los errores en la validación de la actuación, de forma similar a lo que se realiza para los errores en la validación del sensor, lo que permite observar la simetría de la arquitectura propuesta. Si la acción proveniente del control no es validada, está se considera errónea y se genera un código de error para informar de lo sucedido.

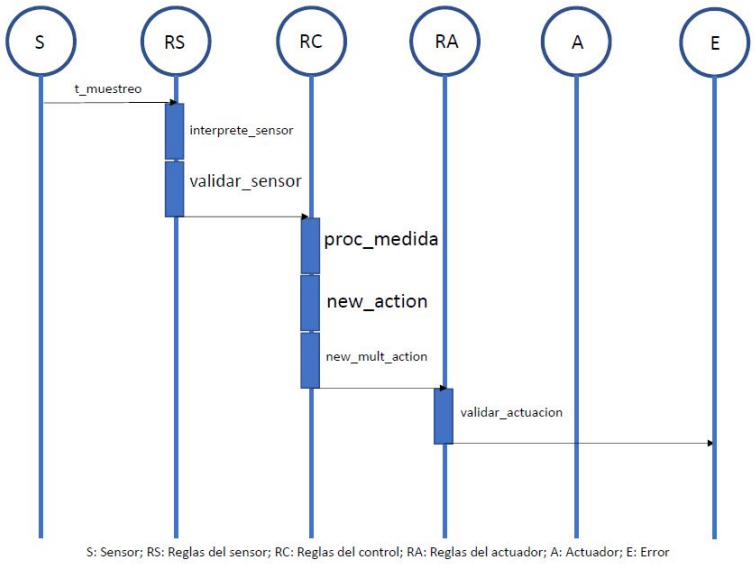

Figura 3.4: Diagrama UML con la evolución temporal de las funciones cuando la actuación generada no es válida.

Otro caso particular en el que el flujo sensor-actuador-control se interrumpe es en el caso de que no se genere ninguna actuación. Esto se puede deber a que el sistema se encuentra en el estado deseado o que las condiciones medidas no han cambiado y no es necesario modificar la referencia. Como se ilustra en la Figura 3.5, si no se genera ni actuación puntual ni una programación de actuaciones, el sistema esperará a que llegue una nueva medida del sensor para volver a ejecutarse.

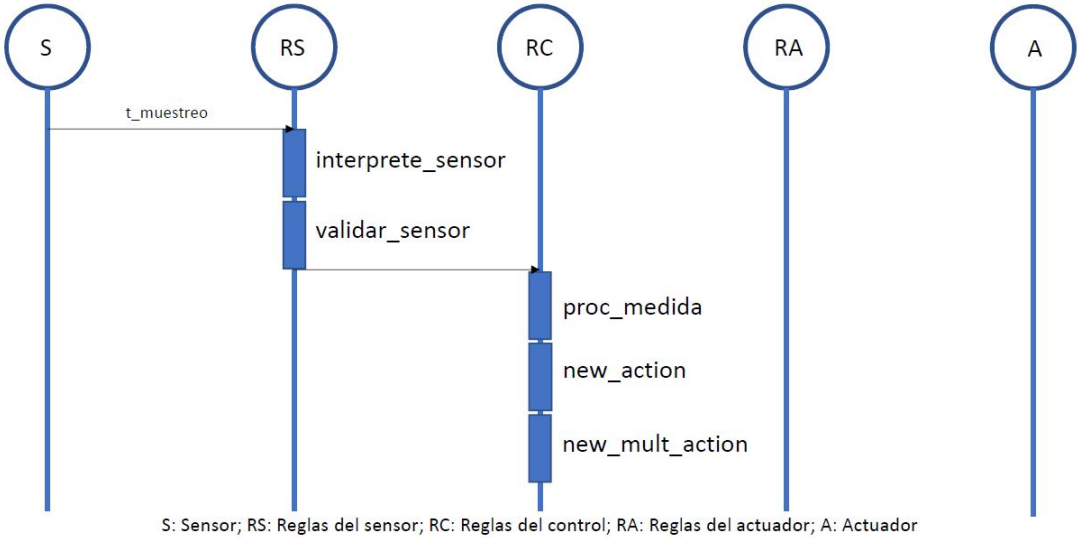

Figura 3.5: Diagrama UML con la evolución temporal de las funciones cuando no se genera ninguna actuación.

3.4. Obtención de la flexibilidad deseada

Una de las principales características de la arquitectura propuesta para desarrollar el sistema flexible para personalizar el comportamiento de una vivienda inteligente es la integración de diferentes lenguajes de programación, como se mostró en la Figura 3.1. Los bloques correspondientes al desarrollador, aquellos mostrados en blanco se van a programar en C++ mientras que la parte del sistema que se muestra sombreada, que corresponde al usuario, se va a programar en Python. Adicionalmente, los archivos de configuración necesarios para el sensor, actuador y control estarán en formato `.ini`. Fundamentalmente, la parte en Python corresponde a las funciones (Tablas 3.1 a 3.10) que son modificables por el usuario mientras que la parte en C++ es toda la arquitectura básica: recepción de medidas por el *gateway*, envío de medidas del *gateway* a las reglas del sensor y devolución de dichas medidas validadas, envió de información de *gateway* a la plataforma, envío y devolución de información entre la plataforma y el control y todo el flujo aguas abajo de información hacia el actuador, de forma simétrica al sensor. Tanto en las reglas del control como en las reglas de dispositivo (sensor y actuador) es donde se produce la integración de las funciones en Python. Esos tres bloques tendrán su funcionalidad implementada a través de funciones de Python que deberán llamar para ejecutarlas.

Las justificación de la necesidad de integrar C++ y Python aparece al recordar el objetivo de crear un sistema flexible. Parte de la flexibilidad provenía del hecho de que se limitara el impacto sobre todo el sistema si alguno de los usuarios deseaba modificar algunas de las características de su aplicación. Con el procedimiento planteado, la parte del sistema modificable por el usuario serán las funciones de Python que se llaman desde C++. De este modo, el usuario puede realizar cambios mientras que el sistema se esté ejecutando y únicamente es necesario reiniciar el sistema, nunca compilar de nuevo el proyecto.

Mientras que una parte de la flexibilidad de la que se quiere dotar al sistema proviene del hecho de integrar Python con C++, también se pretende hacer un sistema flexible en cuanto a que no esté limitado el número de dispositivos que se conectan. Por tanto, y cómo se ha comentado en la explicación de las funciones, se necesita que el sistema permita conectar más de un sensor, control o actuador y que la solución tecnológica propuesta no sea específica para un número concreto de dispositivos. No obstante, en esta parte se entrará en más detalle en el Capítulo 4.

Proporcionar esta flexibilidad permite realizar otras combinaciones que pueden tener sentido en determinadas circunstancias: que un mismo control tenga dos actuadores, que exista un único sensor que sea controlado de dos formas diferentes, etc. En la Figura 3.6 se ha querido representar el diagrama de bloques para un caso más complejo: la existencia de tres sensores, un control y un actuador.

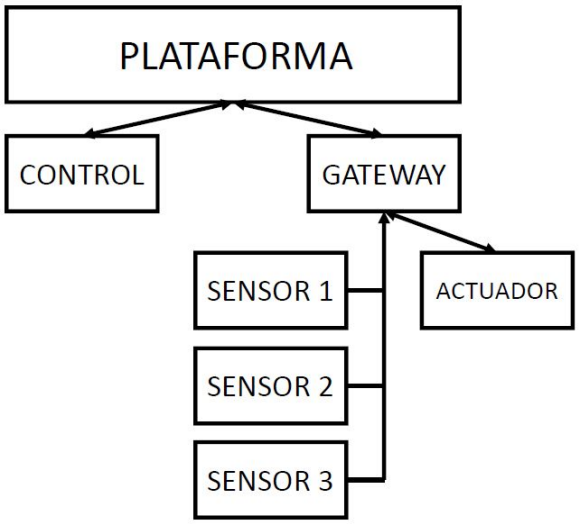

Figura 3.6: Esquema con las comunicaciones del sistema cuando tres sensores y un único control.

La Figura 3.6 cumple un objetivo doble: por un lado muestra como quedarían las comunicaciones entre bloques y por otro garantiza que se cumple la misma arquitectura que lo planteado en [1]. Es decir, todas las comunicaciones de los sensores deben subir hasta la plataforma, que se encarga de llamar al control. Los resultados del control deben subir hasta la plataforma, para después bajar al actuador. Esto se hace así para poder incluir funcionalidades aguas abajo de los dispositivos de forma intuitiva: se puede pensar que el control necesita otro bloque en el que se ejecuta algún tipo de programa para poder obtener los resultados del control y enviarlos a la plataforma. Con la forma planteada de organizar los bloques, es fácil de implementar.

Por último, gran parte de esta flexibilidad que se ha ido buscando se puede lograr gracias a los archivos de configuración, como se ha hecho mención. La estructura que sigue un archivo de configuración, en formato `.ini`, es siempre la misma. Se pueden definir secciones, dentro de las cuáles habrá distintos parámetros con sus respectivos valores. Los valores pueden consistir en texto, números ó listas separadas por comas.

En los ficheros de configuración que se usan para añadir los parámetros de funcionamiento al sistema, se pueden contabilizar hasta cuatro secciones: general, donde se incluye la información común a varios dispositivos, y uno para cada dispositivo: sensor, control y actuador. En estos últimos se incluyen los parámetros necesarios para el funcionamiento de cada dispositivo, el número de dispositivos que hay que crear de cada clase y las relaciones que van a establecerse entre dispositivos.

Capítulo 4

Implementación

En el Capítulo 3 se han analizado los conceptos que se desean incluir en sistema que se está diseñando así como las características tecnológicas que este ha de tener. Sin embargo, es necesario que exista una solución técnica para lo planteado. En este Capítulo se profundiza en los aspectos técnicos de la implementación que permiten alcanzar lo propuesto en el diseño de la interacción. Para alcanzar esto es necesario implementar la integración de C++ y Python, la creación de clases para gestionar los diferentes dispositivos, la creación de múltiples hilos de ejecución en C++ para desligar cada una de las partes de la arquitectura y la lectura de los ficheros de configuración.

De forma introductoria, el problema se ha resuelto creando creado clases que emulan los diferentes dispositivos (sensor, control y actuador), que incluirán funciones desde las que se van a llamar a las funciones incluida en la clase de Python creada, cuando se considere. Por ejemplo, hay una clase que emula al sensor que llama a la función de validar (de Python), para comprobar que el valor del dato de temperatura es correcto, tal y como se explicó en el Capítulo 3. Además, existen clases que permiten almacenar los valores generados y que se han creado de forma independiente a las clases que emulan los dispositivos para que puedan ser memorias compartidas, de forma que a los dispositivos se le pueda dar o no acceso a ellas. Las memorias compartidas son necesarias ya que, por ejemplo, el actuador tiene que tener acceso a los valores generados por el control, que se guardarán en la memoria compartida del propio control. Por último, cuando nos encontremos con un sistema con varios sensores y un único control, al control se le proporcionarán todas las memorias, para lo que se ha creado una clase que permite mapearlas y así facilitar el acceso.

En la arquitectura propuesta en [1] existe una base de datos que almacena la información que va generando el sistema. En el esquema propuesto en el Capítulo 3 no se ha tenido en cuenta para no hacer más compleja la solución técnica. Existen dos razones para esto: no es necesaria para validar el flujo de información propuesto y se pueden utilizar memorias compartidas entre las clases para almacenar la información necesaria para la correcta ejecución del sistema. No obstante, la solución propuesta no es la más adecua-

da en determinadas circunstancias y disponer de una base de datos sería la forma más correcta de almacenar la información, proporcionando acceso o no a las diferentes partes del sistema.

4.1. Estructura general de la implementación

Tabla 4.1: Estructura general de la implementación.

main.c
inicialización del *lock* de Python carga de ficheros de configuración inicialización de memorias compartidas inicialización de dispositivos creación de hilos de ejecución

sharedmemory.c	**memoriamulti.c**	**llamadafuncion.c**	**dataclass.py**
creación de variables de almacenamiento compartido definición de getters y setters de variables	añadir memorias de validación de los sensores getters y setters de las variables de las memorias de validación de cada sensor	inicialización de dispositivos llamadas a funciones de Python creación de while(1)	class config def data_gen def init_sensor def inter_sensor def valida_sensor def init_control def control_med def new_action def new_mult_action def control_med def new_action def inter_actuador

Antes de profundizar en la forma concreta de implementar determinadas funcionalidades, se va a realizar un análisis de la estructura general de la implementación realizada, que se ha realizado en C++ pero realizando llamadas a funciones de Python. La estructura general del código se representa en la Tabla 4.1. El proyecto completo consta de 8 archivos: el archivo principal: `main`, los archivos de origen y encabezado de `sharedmemory`, `memoriamulti` y `llamadafuncion`, así como el archivo de Python, `dataclass`. No obstante, el orden de inclusión reflejado en la Tabla 4.1 únicamente representa la localización de las diferentes funcionalidades en cada uno de los archivos.

Comenzando por el `main`, este se utiliza para inicializar diferentes funcionalidades. En primer lugar, se inicializa el `Global Interpreter Lock (GIL)`, como se verá en la Sección 4.2, para que los diferentes hilos de ejecución que se han creado puedan acceder al intérprete de Python. A continuación, se produce la carga de los ficheros de configuración y se indica al control de qué sensores tiene que tomar los datos. De clases creadas para gestionar los diferentes dispositivos y sus memorias compartidas se tienen que crear objetos

para gestionar el los diferentes dispositivos que se conecten. Esto se realiza en el main. Por último, se crean tantos hilos como dispositivos haya conectados, uno para cada uno, como se verá en la Sección 4.4.

Analizando los diferentes archivos que son necesarios incluir, en `sharedmemory` se crean clases que almacenan las diferentes variables que se utilizan durante la operación. En el Código 4.1, se ofrece un ejemplo de la memoria compartida para el sensor. En el constructor de la clase se inicializa la variable `last_measure`, que almacena la última temperatura que ha sido generada, y `isData_generated` que es una variable binaria que tiene un '1' si se ha generado una variable o un '0' si no, y que se utiliza para llamar a la función de validación de datos cada vez que llegue una medida. Además, se crean *getters* y *setters* para modificar y obtener estos valores desde cualquier dispositivo que tenga acceso a esta memoria.

Código 4.1: Ejemplo de memoria compartida del sensor.

```
1  #include ''sharedmemory.h''
2
3  sensorSharedMemory::sensorSharedMemory(){
4      last_measure = 0;
5      isData_generated = 0;
6  }
7  void sensorSharedMemory::setLast_measure(int data_new){
8      last_measure = data_new;
9  }
10 void sensorSharedMemory::setIsData_generated(int isData_new){
11     isData_generated = isData_new;
12 }
13 int sensorSharedMemory::getLast_measure(){
14     return last_measure;
15 }
16 int sensorSharedMemory::getData_generated(){
17     return isData_generated;
18 }
```

El resto de memorias compartidas tienen una estructura similar, con dos variables de almacenamiento y *getters* y *setters*. La memoria de validación de medidas tiene una variable donde se almacena la última medida validada y otra que informa si la medida ha sido validada o no, para ejecutar el control. El control tiene la última acción generada y otra variable que permite ejecutar la actuación, mientras que esta únicamente tiene una variable que informa si las actuaciones generadas son válidas o no.

En el caso de que haya varios sensores y un único control, o en el caso de que haya controles que no deban tomar datos de determinados sensores, es necesario indicarle al control que valores debe tomar. Esto se resuelve mediante una clase (`MultipleMemory` en el fichero `memoriamulti`), que mapea las memorias de validación que utiliza el control. Esto se consigue mediante una función `addMemory` que mapea, usando el número de identificación del sensor, su memoria de validación. Para utilizar las memorias en el control, es necesario incluir *getters* y *setters* desde los que poder obtener y asignar los valores a

las variables de la validación. Eso se incluye también en el archivo `memoriamulti`.

Por último, en `llamadafuncion` se implementan las funciones necesarias para los diferentes dispositivos. Un ejemplo, para la clase de validación de medida del sensor se muestra en el Código 4.2. Además, de las funciones ahí representadas, en el constructor de la clase hay que pasarle diferentes argumentos, como las memorias compartidas, el estado del GIL o la ruta del archivo de configuración. Según lo mostrado en el Código 4.2, hay una función `run`, equivalente para el resto de clases en la que se implementa el `while(1)`. Antes se llama a una función de inicialización, donde se lee el archivo de configuración y se inicia el GIL para cada hilo de ejecución. La función que se ejecuta permanentemente en el `while(1)` es la que llama a las funciones de Python.

Código 4.2: Ejemplo de las funciones de cada clase.

```
1 void validar::runValidacion(){
2     inicializarValidacion();
3     while (1) {
4         validarMedida();
5     }
6 }
```

4.2. Global Interpreter Lock

La ejecución de diferentes hilos de forma simultánea y la necesidad de que todos ellos deban tener acceso al intérprete de Python para ejecutar sus funciones plantea un problema. Si no se restringe el acceso a este y se libera una vez haya sido usado, únicamente un hilo puede tener acceso a dicho intérprete. Para solucionar el problema, se usa el `Global Interpreter Lock (GIL)` que asegura que solo un hilo tiene el intérprete y que libera su uso una vez ha terminado de ejecutar sus funciones.

Para la correcta implementación del GIL, es necesario inicializarlo en el hilo principal y en cada uno de los hilos secundarios. Además, es necesario adquirir el *lock* cuando se quiera ejecutar una determinada función de Python y liberarlo una vez se haya acabado. El Código 4.3 muestra el código necesario para inicializar el GIL en el hilo principal. En primer lugar, se inicializa el intérprete de Python mediante `Py_Initialize()`. Esta función debe ser llamada antes que cualquier otra función de la C API de Python. Para que el GIL pueda ser accesible por diferentes hilos de ejecución, es necesario inicializarlos mediante `PyEval_InitThreads()`, que inicializa y adquiere el GIL. Es importante que sea llamado en el `main`, antes de crear otros hilos de ejecución. A continuación, se crea un puntero de un objeto de tipo `PyThreadState` que almacena el estado del hilo principal y que se inicializa con valor `NULL`. Finalmente, se obtiene el estado de dicho hilo mediante `PyThreadState_Get()` y la inicialización termina liberando el GIL.

Código 4.3: Inicialización del Global Interpreter Lock en el hilo principal

```
1  Py_Initialize();
2  PyEval_InitThreads();
3  PyThreadState* mainThreadState = NULL;
4  mainThreadState = PyThreadState_Get();
5  PyEval_ReleaseLock();
```

Junto a la inicialización en el hilo principal, es necesario inicializar el GIL en cada uno de los hilos secundarios. Esto se realiza según muestra el Código 4.4. En primer lugar, es necesario adquirir el GIL. A continuación se crea un puntero de un objeto de tipo `PyInterpreterState` donde se almacena el estado del intérprete de `mainThreadState`. Para finalizar la inicialización en cada hilo secundario, antes de liberar el GIL, se crea un puntero de un objeto de tipo `PyThreadState`, con nombre `_myThreadState` donde se almacenará el valor del estado de cada uno de los hilos de ejecución secundarios y que se inicializa mediante `PyThreadState_New(mainInterpreterState)`, lo que permite que este nuevo objeto pertenezca al intérprete.

Código 4.4: Inicialización del Global Interpreter Lock en cada hilo secundario

```
1  PyEval_AcquireLock();
2  PyInterpreterState* mainInterpreterState = _mainThreadState->interp;
3  PyThreadState* _myThreadState = PyThreadState_New(mainInterpreterState);
4  PyEval_ReleaseLock();
```

Por último, la función principal del GIL es limitar el acceso al intérprete de Python, por lo que es necesario adquirirlo para llamar a funciones en este lenguaje y liberarlo una vez se haya acabado. El procedimiento se muestra en el Código 4.5. Después de adquirir el GIL, es necesario cambiar el estado del hilo actual mediante `PyThreadState_Swap(_myThreadState)` para ejecutar el código de Python que se incluiría a continuación. Una vez ejecutado, dado que la llamada a Python se encuentra dentro de un bucle, en el que puede volver a ser llamado, es necesario volver a ejecutar `PyThreadState_Swap(_myThreadState)` para garantizar que el proceso vuelve a repetirse.

Código 4.5: Ejecución de código de Python con el GIL activado.

```
1  PyEval_AcquireLock();
2  PyThreadState_Swap(_myThreadState);
3  // Execute some Python code
4  PyThreadState_Swap(_myThreadState);
5  PyEval_ReleaseLock();
```

4.3. Integración entre Python y C++

Mientras que en la Sección 4.2 se ha visto cómo se limita el acceso al intérprete de Python, la llamada en sí a las funciones también requiere que se ejecuten unas líneas de código correctamente. Se ha optado por crear una clase en Python que almacene las diferentes funciones y las variables de inicialización. El proceso para llamar a una de estas funciones se muestra en el Código 4.6. En primer lugar, se crea un puntero de un objeto de tipo `PyObject` donde se almacena el nombre del módulo de Python. En dicho módulo, se encuentra definida la clase. En este caso, en `module_name` se almacena el nombre del módulo usando `PyUnicode_FromString`. Esto es necesario para almacenar el nombre del módulo de Python en un objeto `PyObject` que se usará posteriormente. El segundo paso es importar el módulo mediante `PyImport_Import` y almacenarlo en otro puntero de `PyObject`.

Código 4.6: Ejemplo de llamada a una función de Python.

```
1 PyObject* module_name = PyUnicode_FromString(''dataclass'');
2 PyObject* module = PyImport_Import(module_name);
3 PyObject* dict = PyModule_GetDict(module);
4 PyObject* clase = PyDict_GetItemString(dict, ''config'');
5 PyObject* instance = PyObject_CallObject(clase, pArgs);
6 PyObject* temp = PyObject_CallMethod(instance, ''valida_sensor'', ''(i)'', last_m);
```

A continuación, es necesario extraer el diccionario del módulo importado. Este es un paso necesario para poder extraer la clase que se ha definido en dicho módulo, en este caso `config`. La principal ventaja de definir las funciones en una clase es que se pueden realizar tantas instancias como se desee de ellas. Para crear una instancia se usa `PyObject_CallObject` a la que se le da como argumento la clase en cuestión y `pArgs` que es una tupla que contiene todas los parámetros de inicialización de la clase. Una vez que se tiene la instancia definida, se pueden llamar a las distintas funciones que hay en la clase mediante `PyObject_CallMethod` a la que hay que indicarle la función que se quiere llamar y darle los argumentos que necesita. En este caso se muestra un ejemplo para la función `valida_sensor`, que se le da el valor de la medida generada (`last_m`) y devuelve `temp`, con un 1 si la medida es válida o un 0 si no, como se muestra en la definición de la función en Python, en el Código 4.7. Los argumentos de la función `valida_sensor` en Python son `self` y `temp`. En `self` se almacenan todos los valores de la instancia, que se incluye en la llamada a la función desde C++ y que son utilizables por las funciones definidas en la clase de Python, mientras que `temp` es la última medida validada.

Código 4.7: Ejemplo de función de clase en Python.

```
def valida_sensor(self, temp):
    if temp >= self.tsensmin and temp <= self.tsensmax:
        r = 1
    else:
        r = 0

    return r
```

4.4. Hilos de ejecución múltiples en C++

Los diferentes dispositivos que conforman el sistema, que como se había comentado, están gestionados mediante clases, van a ejecutarse en hilos de ejecución simultáneos. Será necesario crear un hilo por cada dispositivo que exista en el sistema. De este modo, si el sistema estuviera formado por 2 sensores, 2 validación del sensor, 2 controles y 2 actuadores, este constaría de 8 hilos, además del principal.

En el Código 4.8 se muestra un ejemplo de creación de cuatro hilos, uno por dispositivo, a los que se les da como argumento la función que ejecutarán: `run` de cada clase, tal y como se explicó en el Código 4.2, y un paso por referencia de un objeto de dicha clase.

Código 4.8: Creación de múltiples hilos de ejecución, para cada dispositivo.

```
std::thread ts1(&sensor::runSensor, &sensorObj1);
std::thread tv1(&validar::runValidacion, &validarObj1);
std::thread tc1(&control::runControl, &controlObj1);
std::thread ta1(&actuator::runActuator, &actuatorObj1);

ts1.join();
tv1.join();
tc1.join();
ta1.join();
```

Tras crear los diferentes hilos de ejecución, es necesario esperar a que acaben. Como en este caso, la generación de los hilos de ejecución se hace con funciones que tienen un `while(1)`, no acabarán, por lo que `join` nunca devolverá nada. Como los hilos nunca acaban, no tiene sentido escribir nada debajo de los `join` en el `main` ya que nunca se ejecutaría esa parte del código.

4.5. Lectura de ficheros de configuración

Una de las partes importantes del diseño de la interacción era dotar al usuario de flexibilidad para determinar los diferentes parámetros de funcionamiento del sistema. Estos parámetros se iban a dar al sistema a través de un archivo de configuración, que tiene que ser leído. La lectura de todos los parámetros se muestra en el Código 4.9. Para que la lectura pueda realizarse de forma correcta, en el archivo de configuración deben definirse secciones y dentro de ellas se pueden crear parámetros a los que se les puede asignar velores.

En el ejemplo mostrado, el nombre del fichero se especifica a través de `_myroute` y para los diferentes parámetros, se crean variables locales a las que se le asigna el valor leído.

Código 4.9: Código para la lectura de los ficheros de configuración.

```
1  // Lectura del fichero de inicialización proporcionado
2  boost::property_tree::ptree pt;
3  boost::property_tree::ini_parser::read_ini(_myroute, pt);
4  // Tiempo de muestreo
5  std::string st_muestreo = pt.get<std::string>(``sensor.tmSensor'');
6  std::string::size_type sz;
7  t_muestreo = std::stol(st_muestreo, &sz, 10);
8  // Temperatura máxima del sensor
9  std::string tMax = pt.get<std::string>(``sensor.tMax'');
10 long tempMax = std::stol(tMax, &sz, 10);
11 // Temperatura mínima del sensor
12 std::string tMin = pt.get<std::string>(``sensor.tMin'');
13 long tempMin = std::stol(tMin, &sz, 10);
14 // Temperatura máxima del control
15 std::string tCtrlMax = pt.get<std::string>(``control.TempContMax'');
16 long TempContMax = std::stol(tCtrlMax, &sz, 10);
17 // Temperatura mínima del control
18 std::string tCtrlMin = pt.get<std::string>(``control.TempContMin'');
19 long TempContMin = std::stol(tCtrlMin, &sz, 10);
20 // Temperatura máxima del actuador
21 std::string tActMax = pt.get<std::string>(``actuator.actMax'');
22 long TempActMax = std::stol(tActMax, &sz, 10);
23 // Temperatura mínima del actuador
24 std::string tActMin = pt.get<std::string>(``actuator.actMin'');
25 long TempActMin = std::stol(tActMin, &sz, 10);
```

Capítulo 5

Análisis de casos de estudio

Para comprobar que la implementación ha sido exitosa y se han obtenido las funcionalidades deseadas, se han ejecutardo dos casos de estudio, entre los que variará el número de dispositivos. En primer lugar, se analizará el funcionamiento del sistema con un único hilo de ejecución, que consta de un sensor, un control y un actuador mientras que después se analizarán los resultados de un caso con tres sensores, un control y un actuador.

5.1. Un único hilo de ejecución

En el primer caso de estudio, se va a comprobar el funcionamiento del sistema más básico, que consta de un sensor, que emite medidas cada 5 segundos y las valida en un rango de temperaturas, el control, que toma la última medida validada y genera una actuación, y el actuador, que valida que la actuación generada por el control se encuentra dentro de los límites establecidos. El procedimiento explicado se muestra en la Figura 5.1.

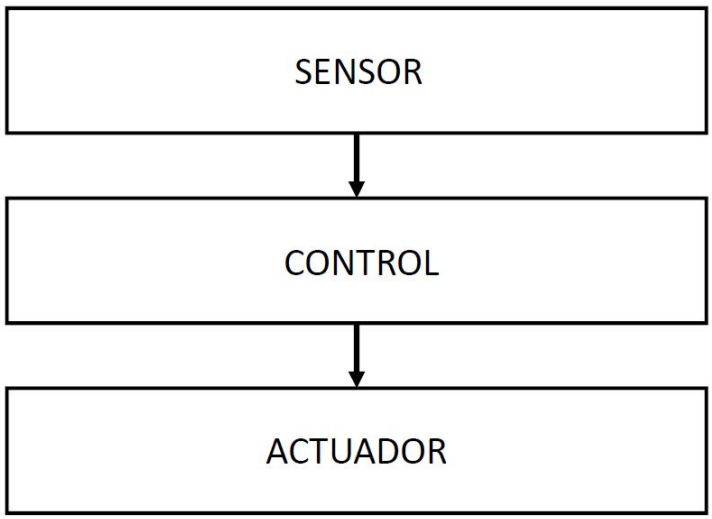

Figura 5.1: Esquema de bloques seguido para el caso de estudio con un único sensor, control y actuador.

Se trata de un proceso secuencial: el sensor genera un dato, y una vez generado, actúa la validación del sensor. Si el dato ha sido validado, se puede llamar al control, si no es válido, se acaba el ciclo. El control, siempre que le llegue una medida, genera una actuación. Solo cuando haya una actuación generada, se llama a la validación del actuador, que valida o no dicha actuación.

Los datos de temperatura del sensor se han generado mediante un generador de números aleatorios, entre -50°C y 50°C y se genera un dato cada cinco segundos, tal y como se muestra en la Tabla 5.1. La medida generada se somete a la validación del sensor, que la considera válida si es menor de 30°C y mayor de -20°C. Si la validación ha tenido éxito, se pasa al control, que para cada medida que le llegue genera una actuación. La actuación es inversamente proporcional al valor de la temperatura. Para una temperatura de -20°C, la actuación vale 25°C, mientras que para una temperatura de 30°C, la actuación generada es de 18°C y para los valores intermedios se interpola linealmente. Se busca que para temperaturas altas se generen actuaciones con temperaturas bajas y viceversa, como se muestra en la Figura 5.2. Por último, las actuaciones se consideran válidas si se encuentran entre los 20°C y los 23°C.

Tabla 5.1: Valores de los diferentes parámetros para el caso de un único hilo.

Sección	Parámetro	Unidades	Valor
Sensor	Tiempo de muestreo del sensor	sg	5
	Temperatura máxima del sensor	°C	30
	Temperatura mínima del sensor	°C	-20
Control	Temperatura máxima del control	°C	25
	Temperatura mínima del control	°C	18
Actuador	Temperatura máxima del actuador	°C	23
	Temperatura mínima del actuador	°C	20

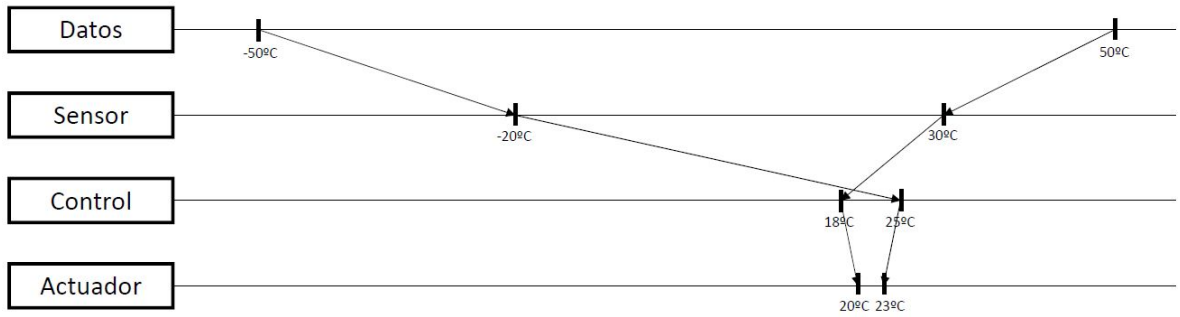

Figura 5.2: Diagrama con las temperaturas permitidas en cada bloque.

Para que lo planteado funcione, es necesario crear las diferentes clases de cada dispositivo y de la memoria compartida, como se comentó en el Capítulo 4. El código que hay que incluir en el `main` se muestra en el Código 5.1. En primer lugar, se indica el nombre del archivo de configuración, que es necesario pasárselo a los diferentes objetos de las clases.

También se le indica al control de qué sensores debe adquirir los datos, en este caso, del único que se va a inicializar. El paso siguiente es inicializar la memoria compartida, creando un objeto de cada una de las clases, la memoria múltiple y los diferentes dispositivos. Con todo lo anterior creado, es necesario crear un hilo de ejecución para cada dispositivo y esperar a que acabe.

Código 5.1: Código del "main" para la ejecución de un único hilo.

```
1  string tipo1 = ''server.ini'';
2  vector<int> sensorescontrol1{1};
3
4  sensorSharedMemory sensorSharedMemoryobj1;
5  validationSharedMemory validationSharedMemoryobj1;
6  controlSharedMemory controlSharedMemoryobj1;
7  actuatorSharedMemory actuatorSharedMemoryobj1;
8
9  MultipleMemory memValidacion1;
10 memValidacion1.addMemory(sensorescontrol1[0], &validationSharedMemoryobj1);
11
12 sensor sensorObj1(sensorSharedMemoryobj1, mainThreadState, tipo1);
13 validar validarObj1(sensorSharedMemoryobj1, validationSharedMemoryobj1,
14 mainThreadState, tipo1);
15 control controlObj1(controlSharedMemoryobj1, mainThreadState, tipo1,
16 sensorescontrol1, memValidacion1);
17 actuator actuatorObj1(controlSharedMemoryobj1, actuatorSharedMemoryobj1,
18 mainThreadState, tipo1);
19
20 std::thread ts1(&sensor::runSensor, &sensorObj1);
21 std::thread tv1(&validar::runValidacion, &validarObj1);
22 std::thread tc1(&control::runControl, &controlObj1);
23 std::thread ta1(&actuator::runActuator, &actuatorObj1);
24
25 ts1.join();
26 tv1.join();
27 tc1.join();
28 ta1.join();
```

Para demostrar el funcionamiento de este caso de estudio, se han impreso por consola unos resultados obtenidos en la simulación, mostrados en la Figura 5.3. En total, se han generado cinco datos, con 5 segundos de intervalo entre ellos. Los dos primeros datos generados corresponden a medidas de -27°C y 37°C, ambos fuera de los limites de validación mostrados en la Figura 5.2, por lo que se imprime por pantalla que la medida validada vale 0, mostrando que no se ha validado. Cuando el dato si está en rango de validación, como en el caso del tercer dato generado, 18°C, se puede pasar al control, que generará una acción, en este caso de 20°C. Con la acció generada, se pasa a la validación del actuador. En este caso, como la medida se encuentra entre 20°C y 23°C, la actuación es válida. El caso de que la medida sea valida pero la actuación no, ocurre en el siguiente dato generado, de 22°C, que da lugar a una actuación de 19°C, que se encuentra fuera del rango de validación de la actuación y no es validada.

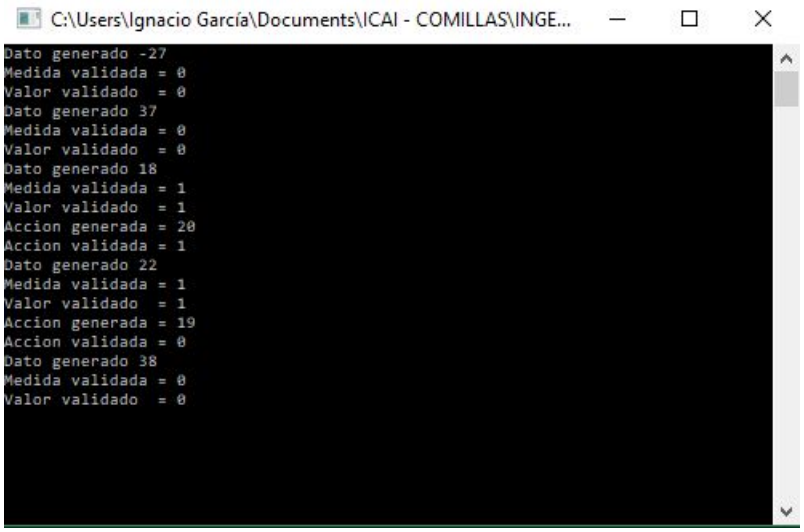

Figura 5.3: Ejemplo de ejecución del caso de estudio para un hilo.

Una de la parte de la flexibilidad del sistema que se ha creado provenía del hecho de que el usuario podía cambiar los parámetros de funcionamiento del sistema. En concreto, puede cambiar el contenido de las funciones de Python sin que sea necesario compilar de nuevo el programa de C++.

Para demostrar que las funciones implementadas en Python se pueden cambiar y que basta con ejecutar de nuevo el programa en C++ para que se implementen los cambios, se ha ejecutado el programa habiendo realizado cambios en el fichero de Python. El cambio realizado en Python ha sido cambiar los límites mínimo y máximo en la generación de números aleatorios. Se han ajustado a -20°C y 30°C, respectivamente. El funcionamiento del programa se ha impreso por consola, como muestra la Figura 5.4.

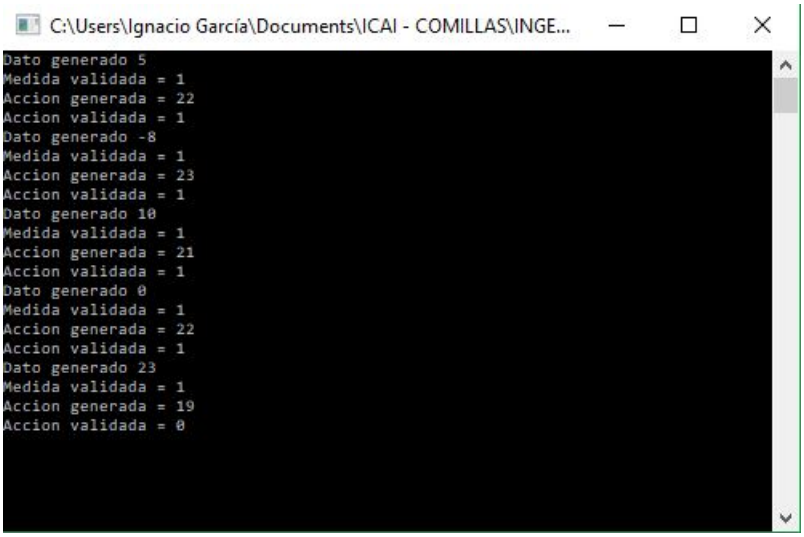

Figura 5.4: Ejemplo de ejecución del caso de estudio tras cambiar el archivo de Python.

Aunque el cambio no se ha realizado sobre las funciones fundamentales del sistema, sino sobre una que durante el funcionamiento real no existirá, sí que permite comprobar que los cambios realizados en el fichero de Python se incluyen en la ejecución del programa principal si se ejecuta de nuevo este.

5.2. Tres sensores y un único controlador

Para demostrar la flexibilidad del sistema creado, se va a realizar un segundo caso de estudio en que habrá tres sensores, un control y un actuador, como se muestra en la Figura 5.5. Los tres sensores generan datos y se validan en hilos independientes, pero la existencia de un único control requeriá que cada última medida de cada sensor sea válida para que se pueda ejecutar el control. El control, solo entonces, genera una acción, que debe ser validada, como en el caso anterior.

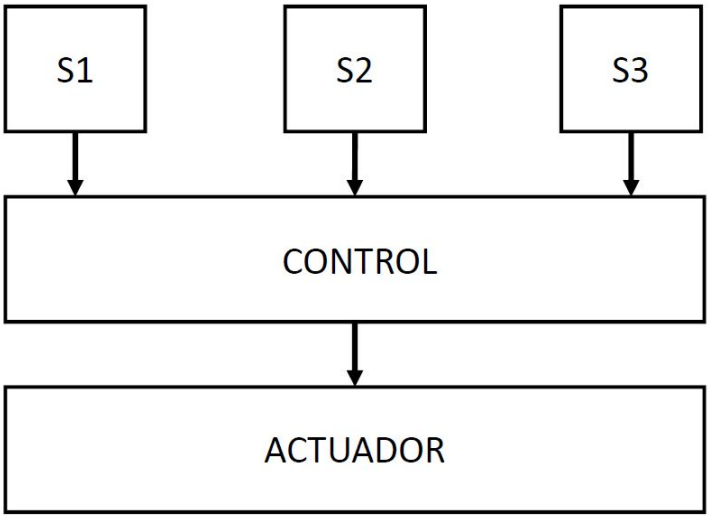

Figura 5.5: Esquema de bloques seguido para el caso de estudio con tres sensores, un control y un actuador.

La estructura del código es la misma que la seguida en el primer caso de estudio. Ahora es necesario crear tantos objetos de las clases como dispositivos haya. En primer lugar, se indica al control que serán tres los sensores de los que deberá tomar las medidas y a partir de estas, generar las actuaciones. Se generan tres memorias compartidas de tipo sensor y validación y una de tipo de control y actuación. Únicamente es necesario crear una memoria múltiple, pero hay que mapearle las tres memorias de validación de cada uno de los sensores. A continuación, se crean los objetos de cada uno de los dispositivos, que se usan para crear los hilos de ejecución.

El funcionamiento propuesto se verifica imprimiendo por la consola los datos generados por cada sensor y las acciones generadas. La Figura 5.6 muestra un ejemplo de esto. Todos

los dispositivos empleados utilizan los mismos parámetros que en el primer caso de estudio, mostrados en la Figura 5.2.

En este caso, el primer sensor genera una medida de 25°C, que es válida. Sin embargo, no se genera ninguna actuación ya que es necesario que la última medida de cada uno de los tres sensores sea válida para que se genere una actuación. En este ejemplo se genera después de que el segundo sensor genere una medida válida de -13°C y el tercero, otra de 22°C. Con las tres medidas validadas, el control genera una actuación con el valor medio de dichas medidas. En este caso es 12°C, por lo que la actuación generada de 21°C tiene sentido, ya que es la temperatura que corresponde al interpolar linealmente entre la temperatura máxima y mínima válidas del sensor.

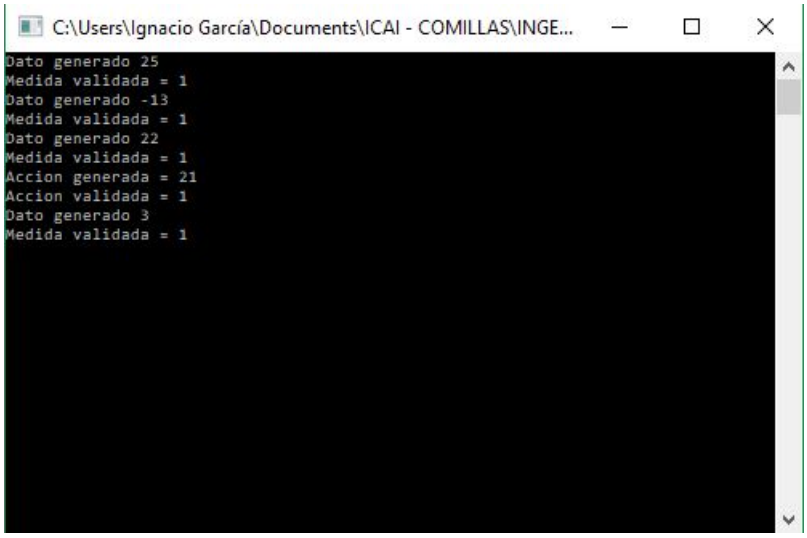

Figura 5.6: Ejemplo de ejecución del caso de estudio para tres sensores y un control.

Capítulo 6

Conclusiones

Las principales conclusiones que se pueden extraer de este Trabajo Fin de Máster son:

- La incorporación de sistemas inteligentes en las casas para controlar los comportamientos de los diferentes equipos allí presentes está provocando que las soluciones domóticas sean cada vez más complejas. Una forma ambiciosa y completa de simular y tomar decisiones en los sistemas domésticos de gestión del consumo eléctrico es mediante la creación de una plataforma de control del sistema eléctrico. La creación de un subsistema para controlar la temperatura encaja en la filosofía de una plataforma global.

- Para controlar la temperatura de un lugar, se ha propuesto un sistema que consta de los sensores de temperatura, un control de temperatura y un actuador, que podrá ser un radiador o aire acondicionado. Para lograr un control flexible haciendo uso de estos dispositivos, se han creado esquemas de funcionamiento que son fácilmente escalable para la presencia de más de una unidad de cada dispositivo. Existe un *gateway* que se encarga de gestionar todos los sensores o actuadores que se conecten a él y la plataforma se comunica con el *gateway* para enviar y recibir información de los dispositivos. Además, la presencia de la plataforma se justifica por la necesidad de independizar la parte del control de los dispositivos.

- Se han identificados 10 funciones que se tienen que ejecutar en diferentes puntos del sistema. Para el sensor, el control y el actuador es necesaria una función de inicialización, en la que se indican los parámetros que se utilizan. Para el sensor y el actuador, hace falta tanto una función interprete, para pasar de las unidades del dispositivos a las de *gateway* como una función de validación, que comprueba que las medidas generadas y las actuaciones son válidas. En el control, se implementan otras tres funciones: una para procesar los datos validados que llegan, otra para generar actuaciones y otra para generar programaciones de actuaciones.

- Se han detectado tres posibles errores en las funciones, para los que se han creado códigos de error. Los dos primeros corresponden a las funciones de validación, del sensor y del actuador, que pueden no validar los valores que les lleguen. El tercero corresponde a la función que genera las actuaciones, que se puede dar el caso que no tenga que generar ninguna actuación y no se pase ningún valor al actuador.
- El sistema creado es flexible en dos ámbitos. En primer lugar, se pueden incorporar tantos dispositivos como se necesiten y establecer entre ellos las relaciones que convengan: varios sensores envían los datos a un único control ó cada sensor tiene su control, por ejemplo. Además, el usuario puede modificar los parámetros de cada uno de los dispositivos sin que esto afecte al sistema total gracias a la integración entre C++ y Python.
- Para cumplir con los objetivos propuestos, se han utilizado clases en C++ para representar los diferentes dispositivos y las memorias compartidas, donde se almacenan los diferentes datos. Cada dispositivo se ha ejecutado en un hilo independiente para así poder crear tantos como se desee. Esto ha obligado a utilizar el *Global Interpreter Lock* de Python, para restringir el acceso al intérprete de Python a los diferentes hilos de ejecución, dentro de los cuáles se llaman a las funciones de Python.
- En los dos casos de estudio planteados se han obtenido los resultados esperados. En el caso con una única unidad de cada dispositivo, se han validado los datos correctamente y las actuaciones se han generado y validado conforme a los criterios previstos. En el caso con tres sensores y un único control, se ha demostrado que hasta que los tres sensores tienen una medida válida, el sistema no genera una actuación.

6.1. Desarrollos futuros

Tras la creación de un sistema para controlar la temperatura de forma flexible, existen algunos aspectos que podrán ser objeto de desarrollos futuros. Entre ellos:

- Independizar cada uno de los dispositivos en ejecutables independientes.
- Creación de la base de datos y sistemas de comunicación entre los diferentes bloques de la estructura.
- Desarrollo de sistemas similares para gestionar otros usos eléctricos de la vivienda o la iluminación.
- Integración del sistema creado como un subconjunto de una plataforma de gestión del sistema eléctrico.

Bibliografía

[1] M. Martin Lopo, J. Boal, and A. Sanchez, "Transitioning from a meta-simulator to electrical applications: an architecture," *Simulation Modelling Practice and Theory*, vol. 94, p. 198, jul 2019.

[2] API Python/C, "docs.python.org/3/c-api/intro.html," *visitado 12/07/2019*.

[3] B. Pell, "Metagame in Symmetric Chess-Like Games," pp. 1–30, 1992.

[4] B. Pell, "A strategic metagame player for general chess-like games," *Computational Intelligence*, vol. 12, no. 1, pp. 177–198, 1996.

[5] Mayadevi, Vinodchandra, and S. Ushakumari, "A Review on Expert System Applications in Power Plants," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 4, no. 1, 2014.

[6] B. Shan, D. Zhao, X. Zhang, F. Guan, and Z. Liu, "Research on Relay Protection Setting Expert System for Main Equipment in Power Plant," *2009 International Conference on Sustainable Power Generation and Supply*.

[7] E. Amaya and A. Alvares, *SIMPREBAL: An expert system for real-time fault diagnosis of hydrogenerators machinery*. oct 2010.

[8] M. T. Veljko, R. T. Predrag, and M. D. Zeljko, "Expert system for fault detection and isolation of coal-shortage in thermal power plants," in *2010 Conference on Control and Fault-Tolerant Systems (SysTol)*, pp. 666–671, 2010.

[9] M. Ross, R. Hidalgo, C. Abbey, and G. Joós, "An expert system for optimal scheduling of a diesel - wind - energy storage isolated power system," *IECON Proceedings (Industrial Electronics Conference)*, pp. 4293–4298, 2009.

[10] G.-s. Jang, J.-y. Keum, J.-Y. Park, and Y.-K. Kim, "Active Alarm Processing in a Nuclear Power Plant," *2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE)*, 2008.

[11] I. Motoiu and L. Caracasian, "Expert Application for Power Plant Operators Traning,"

[12] H. Qudrat-Ullah, "QES-Shell : An Expert System for Nuclear Power Plant Operators Training," *2012 Third International Conference on Intelligent Systems Modelling and Simulation*, pp. 151–156, 2012.

[13] Y. Zhou, X. Fang, and X. He, "Use of an Expert System in a Personnel Evaluation Process," *2011 IEEE International Conference on Quality and Reliability*, pp. 15–19, 2011.

[14] W. R. Nelson, "Reactor: An Expert System for Diagnosis and Treatment of Nuclear Reactor Accidents,"

[15] J. Falqueto and M. S. Telles, "Automation of diagnosis of electric power transformers in Itaipu Hydroelectric Plant with a fuzzy expert system," *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, pp. 577–584, 2007.

[16] W. Becraft, P. Lee, and R. Newell, "Integration of neural networks and expert systems for process fault diagnosis," *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pp. 832–837, 1991.

[17] K. Nabeshima, T. Suzudo, T. Ohno, and K. Kudo, "Nuclear reactor monitoring with the combination of neural network and expert system," *Mathematics and Computers in Simulation*, vol. 60, no. 3-5, pp. 233–244, 2002.