Monitorización y Gestión de Invernaderos

Sebastián Salom Fluxá

Junio 2025

Tutor/a: Ainhoa Azqueta Alzúaz

Índice

Resumen

Elaborar un sistema de gestión automática y monitorización de invernaderos.

Este trabajo propone el diseño e implementación de un sistema de gestión automática y monitorización de invernaderos. Tiene como objetivo ofrecer una alternativa de alto nivel que permita controlar las condiciones ambientales en los invernaderos e incorporar algoritmos propios. El invernadero estará compuesto por módulos de producción con diferentes exigencias de temperatura, riego e iluminación y se obtendrán los datos en tiempo real mediante sensores.

La arquitectura del sistema se basa en un modelo Productor/Consumidor con módulos de producción (o simplemente, módulos), que hacen referencia a cada subgrupo de plantas con condiciones de gestión similares. Cada módulo ejecutará su propio algoritmo programable por el usuario, permitiendo una personalización a medida según las necesidades de cada cultivo. Para dotar al sistema de escalabilidad, se empleará Kubernetes [1] como plataforma de orquestación, mientras que la distribución de mensajes entre instancias se gestionará mediante Apache Kafka [2].

Los datos recogidos por los sensores en cada módulo se enviarán desde un servidor recopilador (Productor) a un servidor de procesamiento (Consumidor). Este último tomará acciones para cada módulo de producción siguiendo la lógica de los algoritmos establecidos por el usuario.

Además, se desarrollará una aplicación de monitorización con interfaz de usuario (UI) amigable que permitirá visualizar los estados de los sensores en cada módulo, las últimas decisiones tomadas por el consumidor y ejecutar acciones. La interfaz contará con usuarios y diferentes permisos para acciones como la visualización de métricas de la granja en tiempo real o la ejecución manual de acciones.

El sistema diseñado se basa en el uso de microcontroladores y sensores distribuidos a lo largo del invernadero para captar y enviar la información al servidor Productor. Es importante recalcar que este comportamiento ha sido simulado, debido a limitaciones para su implementación física. El código de la simulación se ha integrado dentro del código del Productor.

Palabras clave

Riego Automatizado, Agricultura Inteligente, Sensores IoT, Automatización de Invernaderos, Riego Inteligente, Apache Kafka

Abstract

Build an automatic management and monitoring system for greenhouses.

This project illustrates the design and implementation of an automatic management and monitoring system for greenhouses. The aim is to provide a high-level solution to control environmental conditions and enable full customization through user-defined algorithms. The greenhouse will be divided into production modules with specific requirements for temperature, irrigation and lighting, monitored by dedicated sensors.

The system follows a Producer/Consumer model with stateful management for each module. Each module will be capable of executing its own user-defined management algorithm, allowing fully customizable configuration based on the specific needs of each crop. To ensure system scalability, **Kubernetes** [1] will be used as the orchestrating platform, while message distribution between instances will be handled by **Apache Kafka** [2].

Sensor data from each module will be collected by a gatherer server (Producer) and sent to a processing server (Consumer). The consumer will process the data using the user-defined algorithms and execute actions for each production module.

Additionally, a monitoring application with a user-friendly interface (UI) will be developed. This interface will allow users to visualize sensor states, recent decisions made by the system, and perform manual actions. The UI will include users with different roles and permission levels for data visualization, or manual execution of actions.

The designed system includes microcontrollers and sensors distributed throughout the greenhouse, whichever produce and send information to the Producer server. It's important to mention that this part of the system has been simulated because of limitations for its physical implementation. The simulation has been integrated inside the Producer server.

Keywords

Automatic Irrigation, Intelligent Agriculture, IoT Sensors, Greenhouse Automation, Smart Irrigation, Apache Kafka

1 INTRODUCCIÓN

Hacer frente a las dificultades climáticas es una prioridad en un sector como el agrario, sensible a sequías y temperaturas extremas. La tendencia es clara, estos fenómenos naturales serán cada vez más frecuentes y la industria debe hacer frente a estos problemas. Adaptarse a la situación implica innovar en el desarrollo de tecnologías de automatización y gestión de los recursos para combatir estos efectos. En la actualidad, las soluciones más modernas ofrecen niveles de eficiencia nunca vistos anteriormente, no obstante, cuentan con configuraciones muy limitadas y requieren de especialización técnica para su mantenimiento.

Esta propuesta busca ayudar a los agricultores españoles a combatir condiciones adversas a las que se enfrentan. Es un sector al que se le ha dado la espalda en numerosas ocasiones y sus condiciones son cada vez más restrictivas, por lo que se ha intentado que en el Trabajo de Fin de Grado una oportunidad para aportar algo útil para su futuro. Por otro lado, desde el primer momento tuve claras mis intenciones de aprender nuevas tecnologías que me sirvieran para el futuro. Eso, combinado con un gran interés personal en tecnologías de código abierto, motivó la idea de elaborar un proyecto con Apache Kafka como plataforma base. Se eligió Kafka, debido a que se trata de uno de los proyectos bajo licencia Apache más populares en la actualidad, y ofrece un potencial enorme en el desarrollo software, especialmente en campos de procesamiento de datos. Gracias a este proyecto, he tenido la oportunidad de introducirme en otras múltiples tecnologías de código abierto, punteras y altamente demandadas en el mercado laboral, como son Kubernetes, MongoDB [3] y Docker [4].

El objetivo del proyecto es diseñar e implementar un sistema de gestión de invernaderos. Mediante el uso de sensores, microcontroladores y un modelo del sistema basado en el Productor/Consumidor, se propone implementar un ecosistema fácil de desplegar y completamente personalizable, sin restricciones de diseño. También se incluyen archivos que gestionan el despliegue a partir de un fichero de configuración intuitivo, para disminuir los requisitos técnicos requeridos en el mantenimiento del sistema. Asimismo, el sistema propuesto debe escalar según las necesidades computacionales del momento, tolerar la migración de datos en tiempo de ejecución entre instancias, y ser tolerante a fallos. Por otro lado, también se busca desarrollar una interfaz gráfica que permita realizar un seguimiento en tiempo real de los módulos del invernadero, así como accionar los mecanismos de estado de los módulos (ventilación, temperatura y riego).

La instalación física en el invernadero, tal como los sensores o microcontroladores, se tienen en cuenta en el diseño del sistema, aunque su implementación queda fuera del alcance del proyecto, siendo esta parte simulada. La simulación debe ser altamente configurable y ofrecer el control de parámetros del clima, velocidad de la simulación, constantes varias, o inclusive el número de plantas en cada módulo del invernadero. Para permitir la ejecución del sistema, se elaborará una simulación optimizada que pueda representar un escenario con miles de plantas sin comprometer el rendimiento del sistema operativo.

2 Estado del Arte

2.1 Evolución de la agricultura

La Revolución Industrial dio origen a un sistema de producción en masa que trajo consigo grandes avances en el nivel de vida y se vio seguido de un incremento de la población mundial. En este escenario, las industrias se vieron obligadas a competir para alcanzar una mayor productividad en el proceso de producción [5] y se inició una carrera por la automatización de los procesos que perdura hasta nuestros días, logrando eficiencias nunca antes vistas en granjas e invernaderos.

A inicios del siglo XVIII, una serie de innovaciones tecnológicas transformaron la agricultura en Europa. Se introdujeron herramientas más eficientes, como la sembradora mecánica (1701). La mecanización facilitó el cultivo de grandes extensiones de tierra, que podían ser gestionadas por menos personas. De este modo, se aumentó la productividad a la vez que se reducía la mano de obra. Las condiciones de vida de los trabajadores en este sector mejoraron radicalmente. Ya en el siglo XIX, se desarrollaron fertilizantes y otros químicos que permitieron el control de plagas y malezas a gran escala. Todo esto fue seguido del inicio de un fenómeno de migración hacia las ciudades que persiste hasta la actualidad.

Este último siglo se ha caracterizado por el desarrollo de nuevas tecnologías, que han revolucionado la agricultura. En los años 60, se inicia en Estados Unidos la Revolución Verde, un periodo de grandes avances en la agricultura cuyo objetivo era combatir el hambre y la desnutrición. Dado el alto crecimiento demográfico, y las dificultades de la industria para atender la demanda, se buscó aumentar el rendimiento por hectárea. Para ello, se desarrollaron variedades de cultivo de alto rendimiento, resistentes a plagas e incluso a condiciones climáticas desfavorables. También aumentó el uso de fertilizantes y de tecnologías de alto coste, suponiendo un periodo inversión que iniciaba una nueva etapa en la agricultura intensiva.

A finales del siglo XX, empiezan a producirse microcontroladores y sensores electrónicos para automatizar el riego. El desarrollo de estas tecnologías han llevado a la automatización de las tareas, y han desplazado al humano en tareas como la siembra, riego, recolección y procesamiento del cultivo. La automatización no solo optimiza procesos agrícolas, sino que también mejora la calidad de vida de los trabajadores. Gracias a estas tecnologías, las condiciones de trabajo se han visto favorecidas, reduciendo la aparición de problemas de salud históricamente vinculados al sector. También ha permitido reducir el error humano en tareas como el riego, en favor de otras soluciones fundamentadas en estrategias más precisas. De este modo, se ha dado paso a modelos más eficientes que favorecen la reducición de la huella de carbono en beneficio del medio ambiente [6].

De modo similar, se han introducido nuevas formas de monitorización, que mediante la digitalización de los datos de la granja, se ha facilitado el trabajo del granjero en el seguimiento de los cultivos. Estas tecnologías han revolucionado el mercado agrícola y disparado la eficiencia en las tareas de seguimiento del cultivo [7].

Si bien es cierto que el desarrollo tecnológico ha traído muchos beneficios, también ha generado impactos significativos, especialmente en el ámbito de la agricultura intensiva. Este modelo productivo ha contribuido al deterioro ambiental en la producción de aguas residuales con restos de productos agroquímico, sedimentos y otros contaminantes. Además, la expansión de monocultivos puede afectar la flora y fauna, promover la deforestación y acelerar la degradación del suelo.

En este contexto, las tecnologías de gestión y monitorización en tiempo real juegan un papel crucial para optimizar el consumo de los recursos y mejorar la sostenibilidad en la agricultura. El uso de sensores y sistemas informáticos pueden simplificar el proceso, con alternativas eficientes para la gestión de los recursos.

Para la revisión de este estado del arte, se han consultado bases de datos públicas como *Arxiv* y *Google Scholar* y también se ha utilizado buscadores basados en Inteligencia Artificial (IA), como *Perplexity*. Se priorizaron artículos y documentos recientes que correspondieran al contexto tecnológico actual, especialmente en lo referente a la tecnología. Se ha dado preferencia a los artículos publicados a partir de 2020.

2.2 Estrategias de riego en invernaderos

Hoy en dia, existen una gran variedad de sistemas de riego, cada uno cumple con funciones diferentes y cuentan con características particulares que se adecúan a distintos tipos de cultivos y necesidades hídricas. En invernaderos, la necesidad de tener una emisión de agua en zonas controladas, hace que se recurra al subgrupo de sistemas de riego localizados. Un sistema localizado es una técnica de irrigación basada en el desprendimiento de agua directamente en la zona radicular, siendo un modo muy eficiente de regado. Algunas de las estrategias más destacadas en la actualidad son las siguientes:

- **Riego por goteo**

En 1959, el ingeniero israelí Simcha Blass, diseñó el riego de precisión que conocemos hoy en día, basado en la liberación pausada del agua directamente en las raíces mediante un sistema de tuberías y goteros. Esta técnica permite una aplicación eficiente del agua por goteros a baja presión, que expulsan pequeñas cantidades de agua durante un periodo de tiempo. Este método de riego cuenta con grandes puntos a favor que la han establecido como uno de los más populares en los invernaderos. Entre los múltiples beneficios que ofrece, destacan el ahorro del agua, control de malezas, mejor uso de fertilizantes y aumento de nutrientes.

No obstante, este sistema presenta limitaciones que pueden dificultar su eficacia en el cultivo. Una desventaja de este sistema sobre otros es que puede provocar el endurecimiento de la tierra no hidratada. Al regar directamente en la zona radicular de las plantas, se produce un área deshidratada que puede compactarse, dificultando el crecimiento saludable del cultivo. Ademaś, la infraestructura del sistema, es sensible a problemas que pueden surgir con el tiempo, por lo que es fundamental llevar a cabo un mantenimiento constante. Algunas averías comunes son: problemas de presión, obstrucción de goteros y tuberías, o fugas de agua, entre otros.

- **Tuberías exudantes**

En los años 80, el agricultor francés René Petit, tras analizar los sistemas de riego localizados tradicionales, se propuso diseñar un sistema que cuyos materiales no se deterioraran con facilidad y fuera sencillo de limpiar. Ideó un sistema de riego basado en un tubo textil y poroso, que ofrecía riego uniforme y una gran resistencia a obstrucciones y roturas [8].

En esta técnica, el agua se libera de manera uniforme a lo largo del tejido externo de una manguera, de modo que no se producen zonas secas y favoreciendo el desarrollo equilibrado de las raíces. Debido a ello, es una estrategia óptima en cultivos con una distancia reducida entre cada planta, donde destaca la facilidad de la instalación, la durabilidad y su funcionamiento a baja presión.

Por otro lado, el sistema tiene algunas desventajas que pueden convertirlo en una solución menos atractiva. Su instalación en terrenos con pendientes inclinadas puede producir una distribución desigual del agua. A esto se le suma el alto coste de las tuberías microporosas y filtros, que hacen de la instalación una opción menos accesible que otros sistemas.

- **Riego por microaspersión**

Este sistema expulsa agua en forma de pequeñas gotas que hidratan un área específica, lo que la convierte en una opción perfecta para cultivos que requieren de humedad en el ambiente. A diferencia de las estrategias vistas anteriormente, esta necesita una mayor presión, y permite regar superficies

más extensas. No obstante, mediante el riego por microaspersión, no se garantiza el control de las malas hierbas. Además, pueden darse pérdidas de agua por evaporación o viento y supone un consumo energético elevado, debido al bombeo del agua requerido para llegar a la presión necesaria [9] [10].

En este proyecto se va a usar como referencia el sistema de riego por goteo por los beneficios que ofrece en cuanto a la gestión de los recursos. Además, se alinea con la idea de hacer un uso calculado de los recursos y mantener el control de cada planta mediante sensores.

2.3 Sistemas automatizados en la actualidad

Los sistemas de automatización para invernaderos actuales varían en complejidad y eficiencia. Utilizan diferentes tecnologías y metodologías de funcionamiento, y están disponibles en una gran variedad de presupuestos.

Los sistemas más primitivos se basan exclusivamente en el uso de temporizadores que activan el riego al pasar un periodo específico. Estos sistemas son mas comunes en el uso doméstico por su accesibilidad y fácil instalación. Sin embargo, no ofrecen ninguna lógica para tomar decisiones, como por ejemplo desactivar el riego en días lluviosos. Además, tampoco incorporan mecanismos para otras necesidades como la ventilación o la temperatura, imprescindibles en los invernaderos actuales.

Por otro lado, algunos sistemas más avanzados incorporan tecnologías como el Internet de las Cosas (*IoT*) y sensores, a través de los cuales obtienen información sobre la humedad, temperatura e incluso iluminación del invernadero. Estos sistemas optimizan notablemente la gestión de los recursos, no obstante, la mayoría de estos sistemas solamente ofrecen un catálogo de algoritmos básicos predefinidos, que mediante la captura de datos en sensores y tras su procesado, ejecutan las decisiones. Estos sistemas han significado un avance imprescindible en la búsqueda del sistema de gestión perfecto. La inclusión de IoT en invernaderos cuenta con muchas ventajas como pueden ser la optimización del uso de recursos, mejor calidad del producto, mayor productividad y la reducción de costes [11]. Por otro lado, el mayor problema de este tipo de instalaciones, es la limitación que supone tener que usar algoritmos predefinidos sin la opción de introducir los propios al sistema [12].

Algunos ejemplos destacados de este tipo de sistemas son los siguientes:

- **Priva Connext**

La solución de Priva, ofrece una alternativa para invernaderos que gestiona la iluminación, clima, consumo de agua e incluso CO2. Se trata de un sistema altamente escalable que incluye algoritmos predefinidos cuyo comportamiento se puede modificar mediante la configuración de parámetros como la transpiración, temperatura estimada, humedad o iluminación. También se incluye un sistema de monitorización y gestión desde su aplicación *Priva Office Direct*, accesible desde dispositivos móviles, PCs y tablets [13]. En los últimos años, Priva ha realizado un gran trabajo en la búsqueda de crear un sistema de gestión perfecto, incorporando muchos algoritmos desarrollados por *Blue Radix* para invernaderos en todo el mundo. Adicionalmente, cuenta con APIs que permiten conectar el sistema con aplicaciones externas, permitiendo al usuario establecer sus propios algoritmos y que ejecute acciones.

Priva Connext ofrece una buena solución si lo que se busca es personalización. Es un sistema puntero que permite el acceso al sistema desde APIs externas, por lo que la personalización no tiene límites. Sin embargo, esta solución no ofrece ningún tipo de plantilla para estas API externas, por lo que además de los algoritmos, el usuario debe crear su propia API, lo que puede resultar en muchos casos más exigente que la programación de los propios algoritmos. En este proyecto se busca ofrecer un *framework* para que el usuario final se ocupe de la distribución de los mensajes, escalabilidad y tolerancia a fallos, de modo que el usuario tenga la única responsabilidad de introducir los algoritmos desarrollados en la plantilla.

- **Dusun IoT**

Otra solución a destacar es la que ofrece la empresa de soluciones IoT, *Dusun IoT*. El sistema que se propone incluye el uso de sensores con conexiones *Bluetooth Low Energy* (BLE) que envían sus datos a un dispositivo inteligente programable `DSGW-030 IoT gateway` y son reenviados a una API externa, que procesa los algoritmos y envía las acciones determinadas a la *gateway*. Una vez recibida la respuesta, la *gateway*, que se conecta al invernadero, ejecuta las directivas [14]. Gracias a protocolos inalámbricos, la instalación es muy sencilla y mediante el uso del *Bluetooth Mesh Networking Standard* [15], se ofrece escalablidad en el sistema permitiendo a los nuevos sensores su conexión con el recolector de datos. Las señales bluetooth limitan la distancia de emisión de las señales inalámbricas a un radio de 50 a 100 metros en condiciones óptimas, lo que puede ser un factor limitante a gran escala.

En este proyecto, se pretende realizar una arquitectura similar a la que propone *Dusun IoT*, una solución basada en una API local con interfaz gráfica de monitoreo y un servidor de procesamiento de datos. No obstante, el uso de tecnologías bluetooth está fuera del alcance de este proyecto, que se limita a simular la información del clima y de los sensores y se basa en una conexión alámbrica de todos los componentes.

2.4 Limitaciones de estos sistemas

A pesar de las mejoras ofrecidas por los sistemas más modernos, en la actualidad persisten muchas limitaciones que hacen de estos sistemas, modelos inaccesibles o inadecuados para su aplicación en muchos casos. Las principales problemáticas se listan a continuación:

- **Personalización del sistema limitada**

La mayor parte de estos sistemas, se limitan a permitir algoritmos predefinidos donde en ocasiones se permite configurar parámetros como la temperatura o humedad ideales. Sin embargo, al estar restringidos a lógicas predefinidas, se está perdiendo la oportunidad de implementar librerías de código, IA, o acudir a ciertas APIs externas que podrían eliminar muchas limitaciones en la configuración del sistema. Además, para lograr un mayor grado de personalización, es fundamental ofrecer la opción de introducir algoritmos con estado, de modo que se puedan guardar parámetros y objetos de la ejecución en una base de datos persistente.

Un sistema completamente personalizable debe ofrecer un entorno compatible con la inclusión de código propio. Para ello, es necesario facilitar su incorporación al sistema de gestión de invernaderos mediante el uso de una serie de interfaces nativas que asenten las bases en la lógica del algoritmo.

- **Falta de Tolerancia a Fallos**

Depender de un servidor de procesamiento único puede dar lugar a múltiples escenarios de error, bloqueando el sistema de procesamiento y comprometiendo la gestión del invernadero. Por ello, en la actualidad existen muchas tecnologías como Docker Swarm [16] o Kubernetes, que facilitan la gestión y orquestado de contenedores, y permiten despliegues con mecanismos de tolerancia a fallos. Asimismo, en la actualidad, existen proveedores de servicio en la nube como *Amazon Web Services* [17] que están ideados para facilitar el despliegue de estos sistemas en la nube. De este modo se puede asegurar la continuidad y salud del sistema.

- **Falta de Escalabilidad en los Sistemas**

La computación de los servidores está limitada por componentes como la tarjeta gráfica o el procesador. Procesar una gran cantidad de algoritmos en un servidor sin la capacidad de cómputo requerida puede poner en situación crítica el estado del servidor. Para lograr una solución a este problema, plataformas como Kubernetes también ofrecen recursos que facilitan la escalabilidad de servidores basándose en parámetros internos como el porcentaje de uso del procesador o externos, como la cantidad de peticiones por segundo que recibe.

Asimismo, para la distribución de grandes volúmenes de datos entre las instancias existentes, tecnologías como Apache Kafka o RabbitMQ [18] son muy eficaces, permitiendo distribuir equitativamente la carga a procesar entre todas las instancias existentes en un momento dado.

- **Grandes Costes de instalación**

Los costes de instalación son un aspecto a tener en cuenta en la mayoría de casos. La cantidad de sensores, microcontroladores, módulos WiFi y servidores puede ser un problema a la hora de acordar presupuestos. Algunos sistemas vistos anteriormente como el de *Dusun*, implementaban sensores BLE inteligentes y autónomos que funcionan por bluetooth. Estos sensores en invernaderos a grandes escalas pueden aumentar considerablemente el coste de la instalación. Para reducir este problema, se propone hacer uso de los sensores más económicos del mercado:

- AZDelivery B07HJ6N1S4 (Sensor de Humedad)
- AZDelivery B01LXQF9B5 (Sensor de Temperatura)
- ARCELI BH1750FVI (Sensor de Iluminación)

Además, se propone alojar los consumidores en la nube. Gracias a los mecanismos de escalabilidad de Kubernetes, el sistema ejecutará en todo momento las instancias mínimas que sean necesarias, asegurando el buen rendimiento de los servidores y reduciendo los gastos de manera considerable. Con esta solución, el cliente se podrá desentender del mantenimiento y seguridad de los servidores de cómputo al mismo tiempo que se beneficia de la economía de escala de los *Cloud Service Providers*.

Adicionalmente, el uso de licencias comerciales para el sistema de gestión y monitoreo pueden aumentar los costes. Con el fin de reducirlos, se hará uso de las siguientes tecnologías de código abierto y con licencia pública:

- **Apache Kafka** (Apache 2.0 License)
- **Node-RED** [19] (Apache 2.0 License)

- **Docker** (Apache 2.0 License)
- **Kubernetes** (Apache 2.0 License)
- **MongoDB** (SSPL License)

2.5 Conclusión del estado del arte

El análisis del estado del arte ha evidenciado que la evolución de los sistemas de gestión de invernaderos se basa en el uso de sensores, IoT y tecnologías de procesado masivo de datos. La tecnología apunta a ser un factor clave en la lucha contra los problemas climáticos y la escasez de recursos. Además, estos sistemas deben solventar varios desafíos significativos, como la falta de personalización, escalabilidad y falta de accesibilidad por parte los pequeños productores. Todos estos aspectos deben abordarse si se pretende optimizar el consumo de recursos en el sector.

3 Análisis del problema

3.1 Definición del Problema

La industria agraria es un pilar fundamental para la civilización, que alimenta cada día a miles de millones de personas. El sector agrario cuenta con un gran impacto económico en todo el mundo, representando el 4% del producto interior bruto (PIB) global [20] y creando millones de puestos de trabajo. No obstante, la industria se enfrenta a importantes desafíos como el cambio climático y la escasez de recursos, que ponen en riesgo la seguridad alimentaria y la sostenibilidad del sector a largo plazo [21]. Asimismo, el incremento de la población mundial exige una mayor producción alimentaria que se ve dificultada por estas crisis emergentes.

El aumento de las temperaturas globales dificulta la supervivencia de muchas especies vegetales. Su incapacidad para adaptarse a los cambios en el entorno está llevando a numerosas especies a la extinción. Además, en diversas regiones áridas del planeta el agua potable es un recurso cada vez más escaso, lo que limita la capacidad de los cultivos. El cambio climático ha alterado el clima global, generando no solo una tendencia creciente de las temperaturas, sino también un aumento de la frecuencia de sequías en zonas áridas [22] [23].

Un ejemplo de clima árido lo encontramos en España, concretamente en la región mediterránea. El clima mediterráneo se caracteriza por sequías prolongadas durante la estación estival, con periodos que pueden alcanzar hasta tres meses sin precipitaciones. En consecuencia, en estos lugares, la agricultura local se ha visto profundamente afectada por los efectos del cambio climático [24], lo que obliga a optimizar el consumo del agua disponible.

Otro desafío clave es la escasez de recursos como el agua dulce [25]. Organismos internacionales advierten de una emergente crisis hídrica [26] que puede acarrear graves consecuencias. En las últimas dos décadas, los recursos de agua dulce por persona han disminuido un 20% y se estima que alrededor de 2.400 millones de personas viven en países afectados por el estrés hídrico [27]. Aunque el agua dulce supone aproximadamente un 2,5% del agua disponible, la mayoría se ubica en glaciares, hielo y aguas subterráneas [28], estando solamente el 0.3% en ríos, lagos y pantanos, y gran parte ya se encuentra contaminada por el uso excesivo de contaminantes como pesticidas, fertilizantes, metales pesados o microplásticos [29].

El agua también se usa intensivamente en muchos otros sectores, como en la producción y transformación de metales, o en la refrigeración de la maquinaria [30]. Un ejemplo controvertido de este último caso es el de OpenAI, donde se estima que se consumieron 216 millones de litros de agua en solo cinco días para refrigerar los sistemas informáticos utilizados en la generación masiva de imágenes con temática de *Studio Ghibli* [31].

3.2 Justificación de la necesidad

Actualmente los sistemas de riego basados en lógicas básicas, no resultan suficientes para abordar las demandas actuales del sector. Los algoritmos basados en temporizadores o en la humedad del suelo se han vuelto anticuados y ni garantizan un consumo eficiente del agua, ni ofrecen la personalización suficiente para los agricultores. La mayor parte de soluciones actuales ofrecen un catálogo reducido de algoritmos, mientras que otras más modernas como la de *Priva*, recurren a

APIs externas del cliente. Esto último aumenta la necesidad de un perfil técnico para la instalación y mantenimiento del sistema.

Además, muchas de las instalaciones no incorporan interfaz gráfica que muestre información necesaria de cada uno de los cultivos, o no permiten ejecutar acciones manuales en tiempo real. Esto dificulta el seguimiento dedicado del invernadero e incluso puede causar problemas al no poder accionar los mecanismos de riego, ventilación o temperatura al instante.

3.3 Relevancia del Proyecto

A diferencia de los sistemas actuales que limitan la configuración a algoritmos predefinidos o recurren a aplicaciones externas para su funcionamiento, se propone un sistema que hace posible la ejecución de algoritmos de forma nativa en la aplicación. Esto elimina la necesidad de recurrir a sistemas externos y permite crear cualquier tipo de lógica sin restricciones de diseño.

Además, muchos sistemas actuales necesitan un mínimo nivel de especialización de la mano de obra para su instalación. En esta propuesta se busca ofrecer un ecosistema fácil de configurar y accesible a todo tipo de perfiles sin experiencia técnica. Para ello se ha introducido un fichero de configuración y archivos de instalación automatizada.

Para una mayor transparencia, se presenta un repositorio *open-source* [32], que busca desarrollarse mediante aportaciones comunitarias, así como aprovecharse de la publicación de algoritmos públicos.

En definitiva, el proyecto busca asentar las bases de un sistema escalable, fácil de instalar y personalizable, que pueda ser aprovechado en otros escenarios como jardines o huertos y no requiera de mano de obra especialista su despliegue.

3.4 Preguntas clave a responder

Para configurar el diseño del sistema, es necesario responder a las siguientes preguntas clave:

¿Cómo se van a transmitir los datos a través del sistema?

¿Cuánta personalización permitirán los algoritmos?

¿Cómo se logrará la escalabilidad del sistema?

¿Cómo se logrará la tolerancia a fallos?

4 Definición de Requerimientos

En este apartado se definen los principales requisitos funcionales y no funcionales con los que va a contar el sistema.

4.1 Requisitos funcionales

- **Simulación (Productor)**

El servidor simulador debe poder ser configurado desde *server_utilities.h*, que contendrá parámetros para la simulación del clima, tiempo y las estaciones.

El servidor productor debe enviar mensajes con los parámetros siguientes: identificador del plan, humedades, temperatura y estado de los mecanismos de ventilación, riego y calefación.

Los mensajes de cada módulo de producción deben enviarse de manera asíncrona.

La ejecución debe cerrarse de manera controlada, asegurando la liberación de todos los recursos y evitando excepciones o bloqueos.

- **API REST**

La API REST debe recopilar los datos de cada módulo de producción (ID del plan, fecha y hora, temperatura interna del invernadero, humedad de cada sensor, estados de la válvula de riego, ventilación y calefacción) en un objeto JSON.

La API REST debe responder a las solicitudes HTTP de la interfaz gráfica con el último estado captado por los sensores.

La API REST debe responder a las solicitudes HTTP de acciones y modificar el estado del invernadero accionando los mecanismos que sean necesarios.

- **Servidor Consumidor**

El servidor debe poder ser configurado con el fichero de configuración *config.yaml*.

El servidor debe establecer una conexión con el *broker* Kafka especificado en el *config.yaml*.

El servidor debe establecer una conexión con la base de datos nativa MongoDB.

El servidor debe contar con el *topic* riego y tantas particiones como módulos de producción especificados en el archivo *config.yaml*.

El servidor debe poder construirse con un *script* una vez rellenado el archivo *config.yaml*.

El servidor debe consumir mensajes del *broker* Kafka en tiempo real.

El servidor debe obtener mensajes de cada módulo de produccion en distintas particiones.

El servidor debe procesar los algoritmos de los módulo de producción de manera aislada.

El servidor debe ejecutar los algoritmos programados de cada módulo de producción en paralelo.

El servidor debe enviar las decisiones tomadas a la simulación.

El servidor debe poder implementar algoritmos (con y sin estado) para cada módulo de producción.

El servidor debe poder escalar horizontalmente al añadirse nuevos consumidores.

El servidor debe ser capaz de migrar información a otros servidores consumidores al producirse una redistribución de particiones.

El servidor debe poder iniciarse en un contenedor.

El servidor debe poder autoescalar haciendo uso de una plataforma de orquestación.

El servidor debe generar reportes de las acciones tomadas y los recursos gastados.

- **Interfaz Gráfica**

La aplicación debe poder obtener los datos en tiempo real de la API REST del sistema.

La aplicación permitirá a un granjero visualizar los datos de la API REST del sistema.

La aplicación permitirá visualizar los estados de las válvulas y temperaturas internas en cada invernadero.

La aplicación permitirá visualizar el historial de las acciones ocurridas en los invernaderos.

La aplicación permitirá interactuar con la simulación, permitiendo controlar los mecanismos de cada invernadero.

La aplicación permitirá que el usuario cree un escenario del sistema desde el archivo *config.yaml*.

El sistema permitirá cargar el escenario de configuración a partir del archivo *config.yaml*.

4.2 Requisitos no funcionales

El sistema debe ser capaz de gestionar 1000 plantas sin generar latencia.

El sistema debe ser escalable y mejorar su rendimiento al añadir nuevas instancias del servidor consumidor.

La simulación no debe retrasarse más de 1 segundo respecto a la realidad durante su ejecución.

4.3 Tecnologías empleadas

- **Apache Kafka** - *Distribución de los mensajes.*
- **Java** [33], **C++** [34] - *Backend.*
- **CUDA** [35]- *Simulación de sensores.*
- **YAML** - *Ficheros de Configuración.*
- **JSON** - *Transmisión de datos.*
- **Node-RED** - *Frontend.*
- **Docker** - *Despliegue de los servidores productores.*

- **Kubernetes** - *Orquestración y escalado de los servidores consumidores.*
- **Minikube** - *Clúster de Kubernetes*
- **GNU/Linux** - *Sistema Operativo de referencia para la ejecución del sistema.*
- **Bash** - *Generación del entorno*

5 Desarrollo

5.1 Diseño de la Arquitectura

5.1.1 Arquitectura general del Sistema

El sistema está compuesto por dos capas interdependientes: la capa local y la capa externa, cada una encargada de diferentes funcionalidades. En la primera se generan los datos y se ejecutan las acciones, mientras que en la segunda, se lleva a cabo el procesamiento de los algoritmos. A continuación, se explican con más detalle cada una de las capas:

Local

Situada dentro del espacio físico del invernadero. El ciclo de vida de los datos comienza en la captura de los sensores. Cada módulo de producción está equipado con un solo sensor térmico, que mide la temperatura interna, y con tantos sensores de humedad como plantas tenga el módulo.

Inicialmente se captura el estado del módulo mediante sensores, que están conectados alámbricamente a un microcontrolador. Este último se encarga de construir los mensajes resultantes de los datos capturados y darles formato para finalmente enviarlos al productor del invernadero mediante un cable USB.

El productor recibe los datos de los microcontroladores y forma mensajes con la información procedente de los diferentes módulos de producción. Estos mensajes se distribuyen a diferentes particiones del tópico "riego" dentro del *broker*, lo que permite la escalabilidad horizontal en el sistema. El servidor productor, está diseñado para optimizar el envío de mensajes y permite una reasignación dinámica del tamaño de los paquetes. La transmisión de datos del *broker* al consumidor se realiza mediante el protocolo binario de Kafka, basado en TCP.

La automatización del sistema de calefacción, ventilación y aire acondicionado (HVAC), que incluye también riego, viene dada por el servidor productor, que incorpora una API que permite modificar el estado del invernadero desde aplicaciones externas. Para ello, el productor se conecta alámbricamente con el sistema HVAC para enviar señales de encendido/apagado.

Externa

Por otro lado, la infraestructura externa se ubica en la nube, y está formada por un clúster de Kubernetes que incluye el *broker*, servidores consumidores y una base de datos no relacional MongoDB. Para su despliegue, se propone la infraestructura de *Amazon Web Services* debido a las facilidades que ofrece para el despliegue de aplicaciones en Kubernetes, no obstante, esto último queda afuera del alcance del proyecto.

El *broker* almacena los mensajes, que esperan ser consumidos, en las múltiples particiones del tópico "riego". La recepción de los mensajes se inicia por el consumidor, siguiendo un modelo *pull*. En sistemas Productor/Consumidor, este modelo consiste en que el consumidor es quien solicita los mensajes, en contraste al modelo *push*, donde el *broker* los envía directamente. Al igual que en la producción de los mensajes, la transmisión de datos del *broker* al consumidor se realiza mediante el protocolo binario de Kafka.

Para ofrecer persistencia de datos, se ha integrado dentro del clúster una base de datos MongoDB, que almacena la información necesaria para el funcionamiento de los consumidores.

En la siguiente figura, se muestra un esquema del sistema con todos los componentes que participan en el invernadero así como sus conexiones:

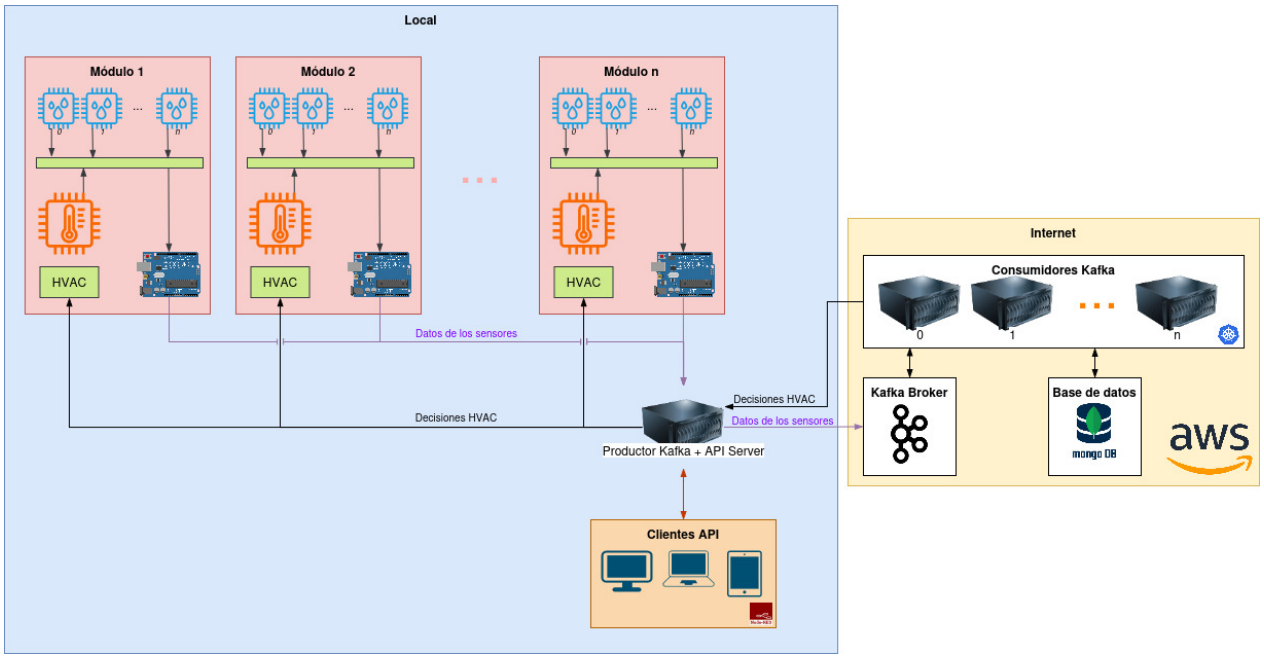

Figura 1: Diseño del Sistema

La imagen ilustra la arquitectura del sistema, compuesta por dos partes: la instalación local y la infraestructura externa (en la nube).

En la instalación local se encuentran los sensores, microcontroladores, el servidor productor y los clientes. Se pueden ver las conexiones que los vinculan. Se puede ver como cada módulo cuenta con un sensor térmico y varios de humedad, así como su microcontrolador y el sistema HVAC. Nótese como la API está conectada con el sistema HVAC y permite conexiones desde clientes con la aplicación web para Node-RED.

La instalación externa está compuesta por un clúster de Kubernetes que incluye los consumidores, el *broker* y la base de datos. También se muestran las conexiones broker-productor, broker-consumidor, consumidor-almacenamiento y consumidor-API, así como donde se transmiten las decisiones finales y los datos de los sensores.

En el siguiente diagrama se muestra el camino que siguen los datos dentro del sistema, desde su generación hasta la modificación de los mecanismos de estado del invernadero.

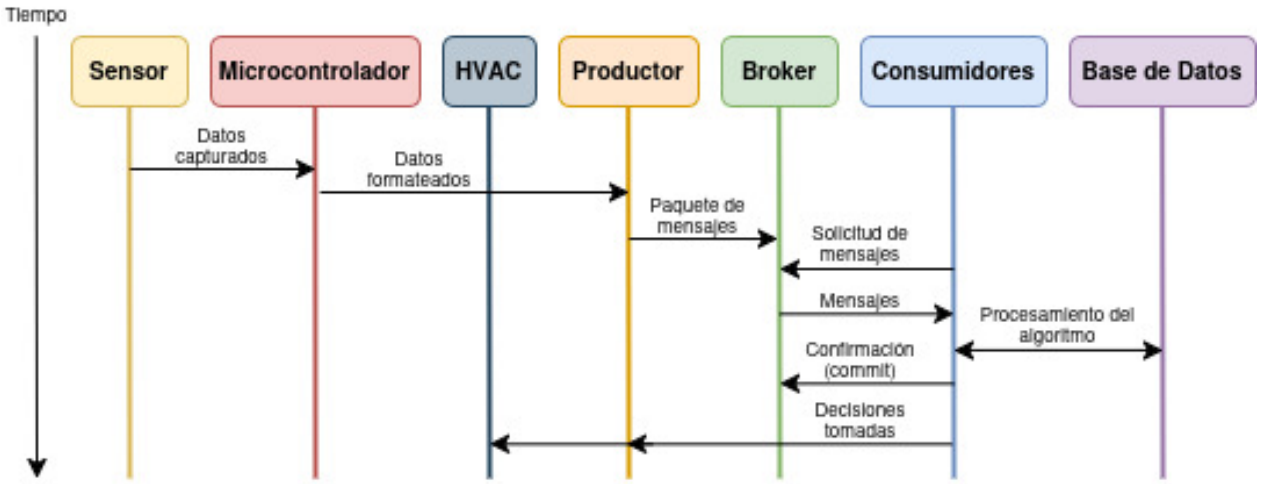

Figura 2: Diagrama de secuencia que siguen los datos

El diagrama de secuencia describe el ciclo de vida de los datos generados por los sensores. Los microcontroladores transmiten los datos formateados al productor, que los envía al *broker* en paquetes. El consumidor solicita y consume mensajes del *broker*. Para procesarlos, puede ayudarse de la base de datos. Posteriormente, se envía la confirmación al *broker* y procede a enviar las decisiones a la API, que retransmite estas instrucciones al sistema HVAC.

Este proyecto se enfoca en el despliegue de un clúster en Kubernetes haciendo uso de los recursos definidos en la carpeta `KubernetesResources`. No obstante, también se ofrece la opción de desplegar el entorno en docker.

5.1.2 Arquitectura de clases del servidor local

El servidor local ejecuta simultáneamente tres componentes: el servidor productor, la API y la simulación. En este proyecto, la simulación y la producción se ejecutan de forma secuencial mientras que la API se ejecuta en paralelo, lo que requiere la implementación de mecanismos de sincronización que garanticen la exclusión mútua.

Para su desarrollo, se ha hecho uso de tres ficheros de código principales:

- **server.cpp**

Es el fichero principal del servidor productor. Contiene la función `main` e inicia la simulación. En su código se definen de los recursos de la API. Para gestionar las peticiones, se hace uso de la clase `UserConnection`, cuyos contenidos se muestran a continuación:

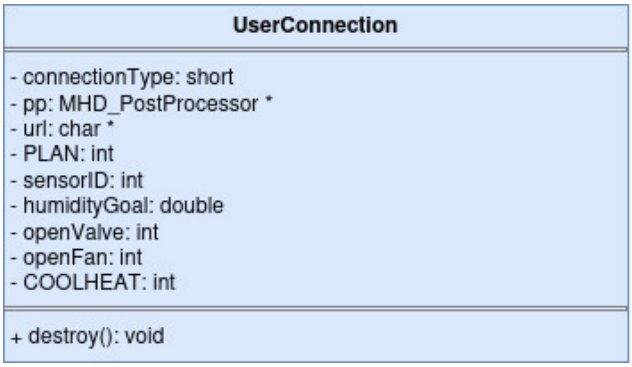

Figura 3: Diagrama de clases para UserConnection

La imagen muestra los contenidos de la clase `UserConnection`. Contiene los parámetros que gestionan las peticiones de la API (connectionType, pp, url) y otros para modificar el estado del invernadero (PLAN, sensorID, humidityGoal, openValve, openFan, COOLHEAT).

Todos los atributos han sido encapsulados mediante métodos *getters* y *setters*, que se han eludido en la imagen para simplificar su representación.

- **weather_simulation.cpp**

Lleva a cabo la simulación de los parámetros del clima, como la iluminación o la temperatura mediante la clase `WeatherSimulation`, que a su vez recurre a una estructura `Day` para guardar los parámetros climáticos para cada hora del día.

- **world_simulation.cu**

Inicia el servidor productor. Para ejecutar la simulación, se introduce la clase `WorldSimulation`, que genera los datos del escenario.

Los contenidos las clases usadas para la simulación, así como sus relaciones, se muestran a continuación:

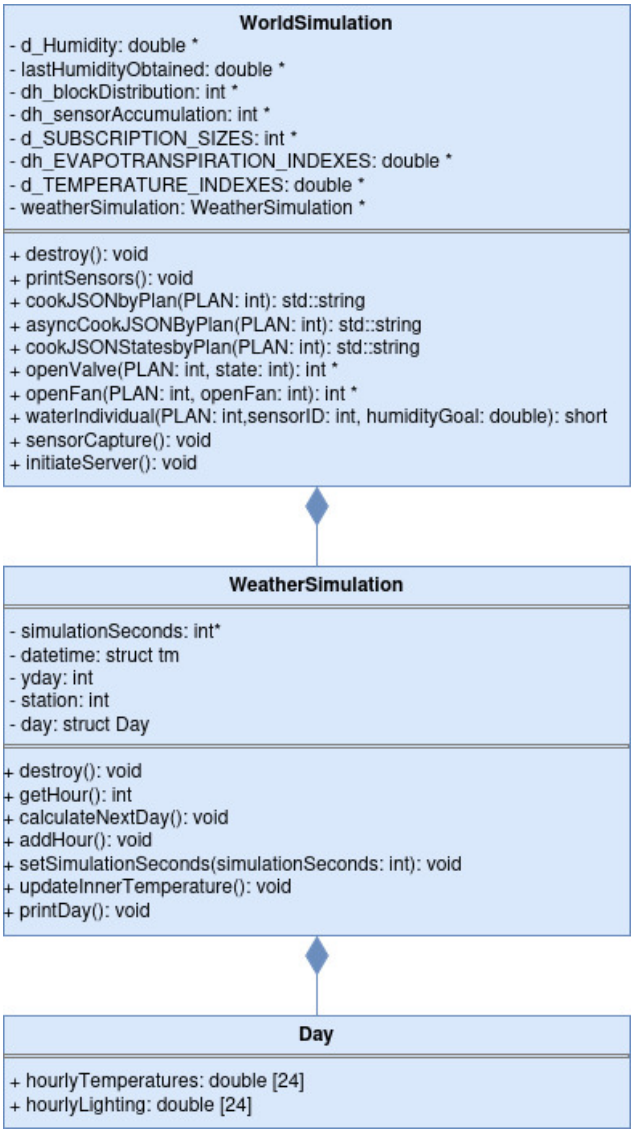

Figura 4: Diagrama de Clases UML de la Simulación

En el diagrama se muestran los contenidos de las clases principales de la simulación: `WorldSimulation`, `WeatherSimulation` y `Day`. Las relaciones que las vinculan son de tipo composición. Cada clase contiene una función `destroy`, para eliminar los datos de las clases contenidas. Para seguir los principios de encapsulamiento, cada variable tiene *getters* y *setters*, que no se muestran por claridad en la representación. Las variables que residen exclusivamente en la memoria de la GPU, se nombran con el prefijo "d_" (de *device*). Por otro lado, las que se comparten entre el procesador y la tarjeta gráfica, usan el prefijo "dh_", (de *device-host*). El resto de variables, residen exclusivamente en la memoria del procesador.

La interacción entre estas clases viene resumida por la siguiente imagen:

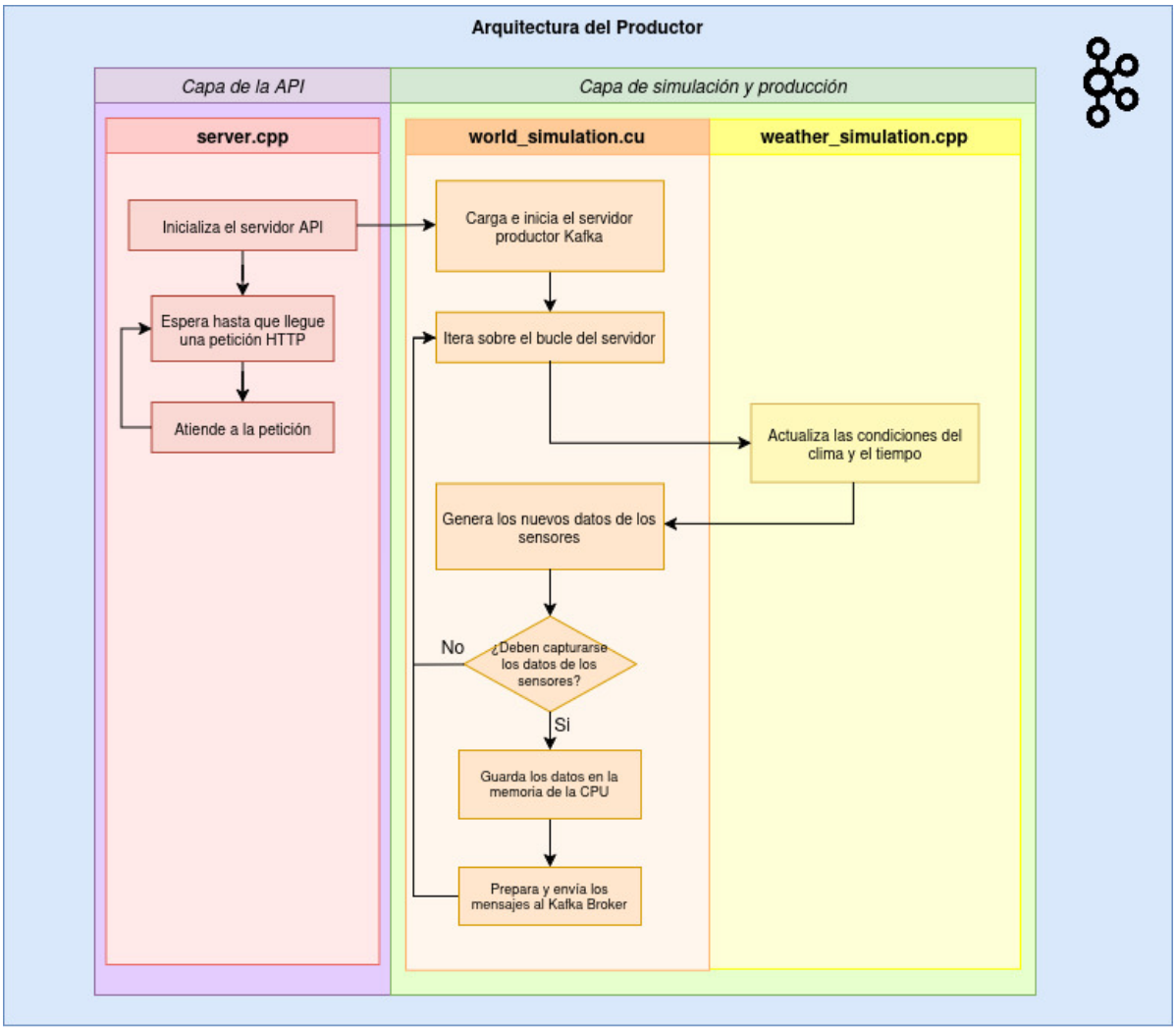

Figura 5: Diseño de las Clases en el Productor

En el diagrama de flujo se ilustran las dos capas del sistema. Por un lado la API trabajando de forma asíncrona, y por el otro la simulación y producción trabajando de forma secuencial. También se ven las conexiones entre los tres ficheros principales mencionados previamente. Inicialmente la ejecución comienza por la función `main` definida en el archivo *server.cpp*, responsable de gestionar la API. La simulación se inicia desde la función `main`, y sigue la lógica especificada en el archivo `world_simulation.cu`, que a su vez recurre a los contenidos de `weather_simulation.cpp` para calcular el clima y tiempo en cada iteración de la simulación.

5.1.3 Arquitectura de clases del consumidor

El consumidor se ha diseñado con dos capas: la capa de control y la del procesamiento en segundo plano.

GreenhouseConsumer es la clase principal del consumidor. Se encarga de organizar la capa de control mediante la recepción de mensajes del *broker* y su distribución y ejecución en tareas (*tasks*). También gestiona la comunicación con el *broker* para comunicar los *commits* al *broker*. Desde la capa de control se inicializa la memoria en el caso de nuevas particiones asignadas, mientras que en el caso contrario es liberada.

Task es la clase usada para procesar los mensajes de cada módulo de producción de forma asíncrona. Una tarea está compuesta por los mensajes asignados a un tópico y clave, y se inicializa desde la clase `GreenhouseConsumer`. La clase `Task` puede entenderse como un paquete de mensajes que se van procesando uno tras otro, con soporte para errores y con facilidades para consultar su estado desde clases externas. Esta clase es la intermediaria entre el *broker* y los algoritmos del usuario, siendo su principal función la ejecución de las funciones base de la clase abstracta `BasicPlan`, vista más adelante.

A continuación se muestran los atributos y funciones de las clases principales en el consumidor:

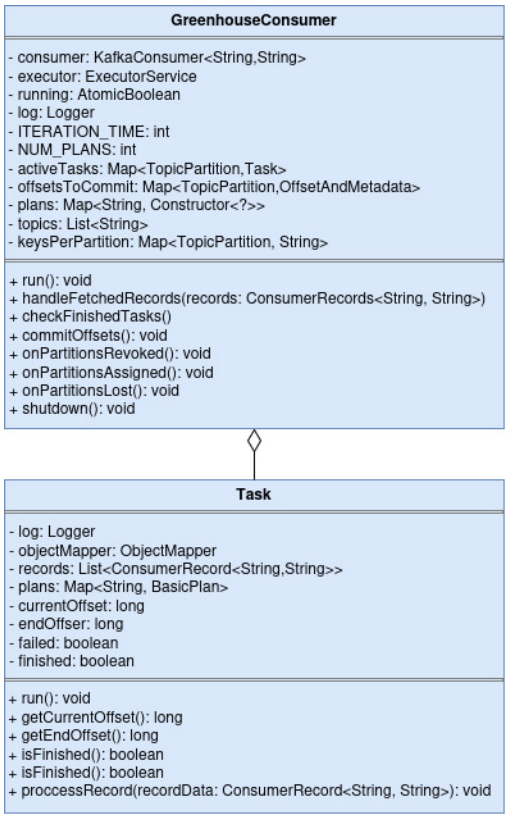

Figura 6: Diagrama de Clases UML del consumidor

La imagen expone los atributos de las clases mencionadas así como su relación. Se puede ver como `GreenhouseConsumer` cuenta con atributos de estado de la instancia como las tareas activas o

offsets a confirmar, así como métodos para controlar la instancia en caso de redistribuciones de las particiones o en el cierre inesperado del sistema. Por otro lado, desde `Task` se controla el estado de la ejecución a nivel de mensajes e incluye métodos para consultar el estado de las tareas. Cabe destacar, que las clase `Task`, cuenta con una relación agregación con la clase `BasicPlan`, donde una *Task* contiene una serie de objetos `BasicPlan` pero son independendientes. Se ha decidido no incluir esta relación por simplificación del diagrama.

Por otro lado, **BasicPlan** es la clase abstracta de la que se extienden las lógicas programadas por el usuario. Contiene lo necesario para interactuar con el invernadero e integrar los algoritmos en el *framework*. Con su uso se busca facilitar el diseño. Esta clase incluye métodos abstractos para implementar los algoritmos además de otras facilidades para el guardado y recuperación de los algoritmos desde los consumidores. La clase `BasicPlan` hace uso de cuatro clases para elaborar su función: `SensorData` es la clase Java a la que se deserializan los contenidos de cada mensaje, `GreenhouseAction` permite accionar los mecanismos del invernadero, `DataCallback` es una clase abstracta usada para definir los contratos, y `DBHashMap` facilita la interacción con la base de datos.

El esquema de las clases queda de la siguiente manera:

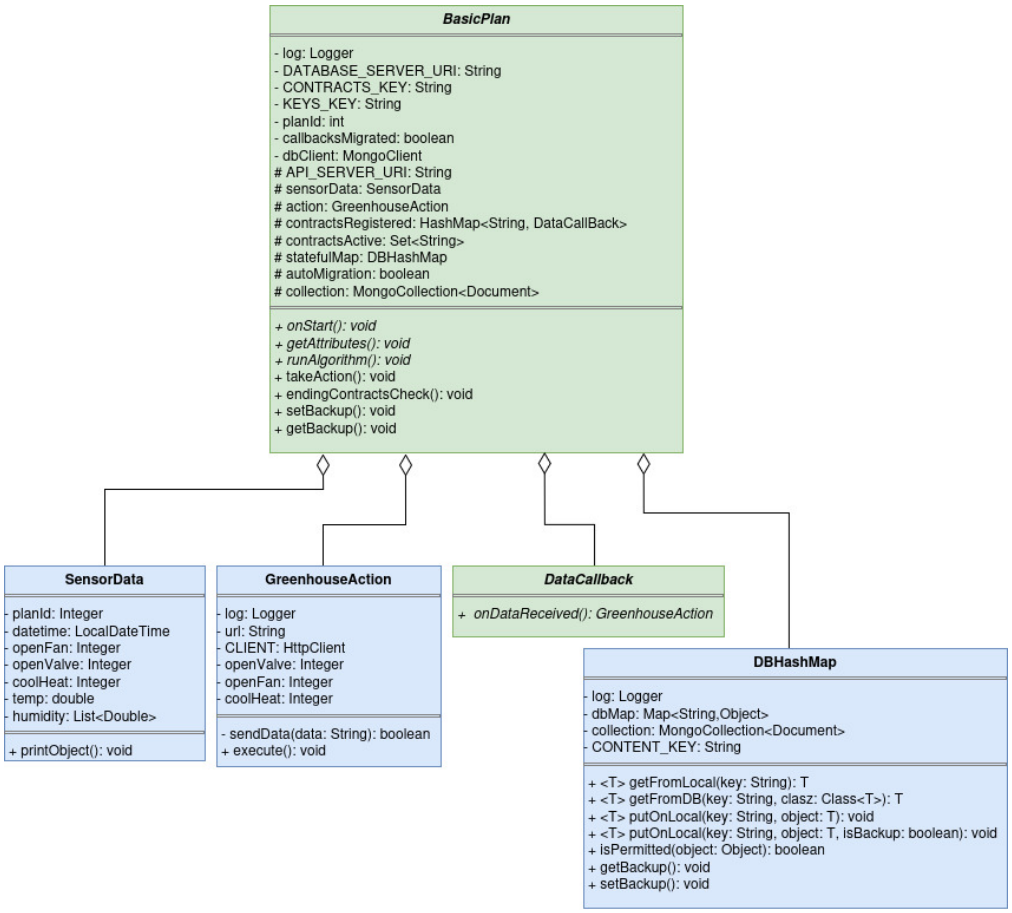

Figura 7: Diagrama de Clases de BasicPlan

El diagrama de clases muestra las relaciones existentes entre las clases implicadas en el diseño de los algoritmos. Las clases abstractas (en color verde), contienen métodos abstractos (mostrados

en cursiva). La clase `BasicPlan` mantiene una relación de agregación con las clases `SensorData`, `GreenhouseAction`, `DataCallback` y `DBHashMap`, que se comentarán en profundidad en el apartado de implementación.

Tal y como se ha visto en este apartado, el consumidor está formado por tres clases principales: `GreenhouseConsumer`, `Task` y `BasicPlan`. En la siguiente imagen, se muestra cómo interactúan entre ellas:

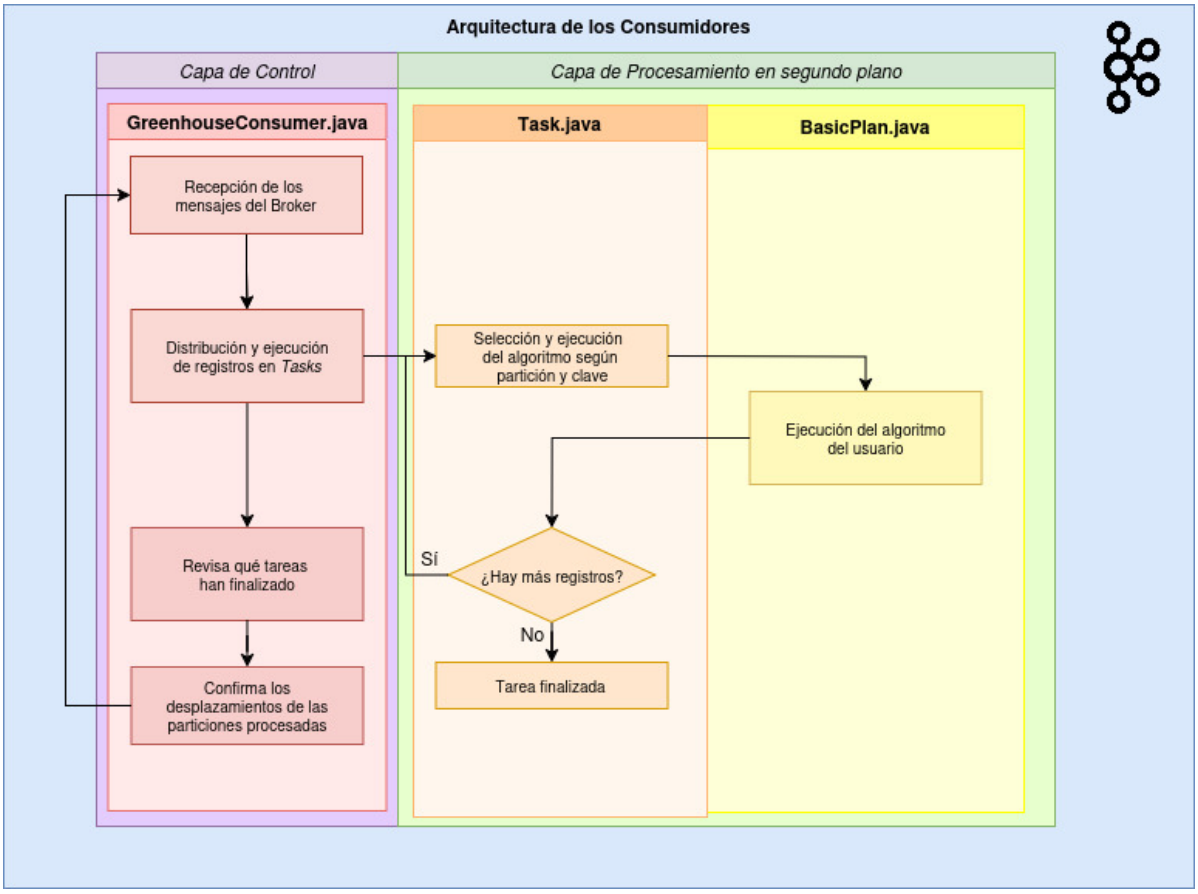

Figura 8: Diseño de las Clases en el Consumidor

En el diagrama de flujos se muestra cómo cooperan las clases principales. Están divididas entre la capa de control y la capa de procesamiento en segundo plano.

1. GreenhouseConsumer: Inicia la ejecución del sistema. Distribuye los mensajes en tareas que se ejecutan asíncronamente. Esta clase es responsable de gestionar tanto la solicitud de mensajes al *broker* como de confirmar los desplazamientos una vez procesados los datos.
2. Task: Esta clase se encarga de ejecutar los algoritmos. Es el punto de entrada a la capa asíncrona del consumidor. Coordina la ejecución de los mensajes con sus respectivos algoritmos y ofrece mecanismos para consultar el estado de cada tarea.
3. BasicPlan: Ofrece un catálogo de métodos necesarios para la inclusión de los algoritmos en el consumidor.

5.2 Implementación de Componentes

En la arquitectura, se han visto los agentes que participan en el sistema automático de gestión de invernaderos. En este apartado se explica cómo se han implementado cada uno de ellos. También se va a profundizar en la lógica que siguen y en las conexiones que establecen (introducidas en la sección previa).

5.2.1 Productor

El servidor productor es responsable de la generación y envío de mensajes al *broker*. Está implementado en la instancia local, que también implementa una API a través de la cual los consumidores comunican sus decisiones al invernadero. En la arquitectura del sistema, la instancia local, está directamente conectada con los microcontroladores de todos los módulos así como con los mecanismos de estado del invernadero (ventilación, temperatura y riego). Al ser todas las conexiones alámbricas, se recomienda que esté ubicado en un punto central y estratégico del invernadero, ya que constituirá el núcleo de la instalación física.

Para la interacción con el *broker* desde C++, se ha usado la librería de Kafka para C++ de *Morgan Stanley*, *modern-cpp-kafka* [36], una librería de código abierto con repositorio público en GitHub que simplifica la implementación de Apache Kafka en C++, con clases que facilitan algunas tareas como la generación de mensajes o la liberación de recursos.

En la interacción con el *broker*, un productor mal configurado puede entorpecer el funcionamiento del sistema. Es por ello, que se han establecido una serie de ajustes que juegan un papel crucial en la salud del *broker*. Con la finalidad de asegurar el procesamiento de los datos en tiempo real, se ha introducido un tiempo límite en el envío de cada mensaje equivalente al periodo en segundos entre envíos. De este modo, solamente se envían datos capturados en tiempo real. Además, para evitar pérdidas de mensajes, se ha establecido una política de confirmaciones (*acknowledge*) en el envío de los paquetes, donde el productor enviará los paquetes hasta recibir una confirmación.

Para que el flujo de datos siga el camino deseado, el productor y consumidor deben compartir el mismo algoritmo de particionado. Por este motivo, se ha configurado en ambas partes el particionador *MurmurHash2*, una función hash rápida que permite distribuir los mensajes en la particiones de forma equitativa. Estos algoritmos obtienen las particiones según dos valores de la ecuación: la clave del mensaje, y el número de particiones totales. Para maximizar el reparto de estas distribuciones, se ha configurado para el tema "riego" un número de particiones equivalente a la cantidad de módulos de producción del invernadero (autogenerado en la construcción del entorno).

Las claves de los mensajes –de ahora en adelante, claves de módulo– producidas, siguen el formato *plan*<*id*>, siendo *id*, la posición del módulo de producción dentro del arreglo *plans* en el fichero de configuración *config.yaml*. Gracias a este formato constante, se asegura que los mensajes de un módulo se guardan en la misma partición durante todo el tiempo de ejecución.

Para el despliegue del servidor productor es importane que exista la variable de entorno `KAFKA_URI`, con la dirección IP (o nombre de dominio), seguida del puerto del servidor Kafka al que debe enviar los datos generados. A continuación se muestra un ejemplo:

```
KAFKA_URI: "192.168.1.100:9092"
```

5.2.2 Simulación

Debido a que este proyecto parte de la base de que el servidor productor dispone de los mensajes para ser producidos, la simulación se ha incorporado dentro del productor, por lo que las tareas de generar los datos (sensores), darles formato (microcontroladores) y enviarlos al *broker* se realizan en la misma ejecución.

Para imitar un entorno real de invernadero con miles de plantas agrupadas en módulos, se ha buscado simular de forma eficiente miles de sensores de humedad y de temperatura. El principal reto para su implementación, es que la generación de miles de datos por segundo, cantidad necesaria para simular entornos de producción, puede sobrecargar el servidor y comprometer su rendimiento. Con este motivo, se ha decidido implementar estos servidores dentro de un programa de alto rendimiento en C++, uno de los lenguajes más eficientes que ofrece un amplio número de librerías que facilitan su aplicación. La clave de la simulación, reside en el uso de la librería de *NVIDIA*, CUDA C++ y su compilador *nvcc*. De este modo, se pueden realizar miles de cálculos en paralelo usando los hilos de la tarjeta gráfica (GPU), y simular miles de sensores sin sobrecargar el procesador (CPU).

También se ha buscado mejorar el aprovechamiento del hardware de la CPU mediante el uso de otras optimizaciones como la vectorización y paralelismo. Para ello, se ha usado OpenMP [37], una librería que facilita la tarea de añadir mecanismos de concurrencia en programas de C++.

La simulación se ha implementado en los ficheros *world_simularion.cu* y *weather_simulation.cpp*, cada uno cumple una función distinta.

world_simulation.cu

Es el fichero encargado de inicializar la configuración de la simulación. Introduce la clase `WorldSimulation` que actualiza los datos generados del invernadero (sensores y sistema HVAC). Los hilos de la tarjeta gráfica son asignados dinámicamente para que cada uno se encargue de calcular humedades para una planta del invernadero.

La simulación hace uso de una multitud de variables esenciales que se listan en la cabecera *world_simulation.hpp*. El fichero utiliza dh_VALVE_STATES, un arreglo de enteros, para gestionar el estado de las válvulas de riego (1 si abierto, 0 si cerrado). La clase WorldSimulation contiene otros parámetros necesarios en la lógica. La variable d_Humidity contiene una matriz de *n* dobles, siendo *n* el número de plantas en el invernadero. Cada doble se mantiene en un rango de 0 a 100, que representa el porcentaje de humedad en el sustrato. Esta matriz de humedades se vuelca en la variable *lastHumidityObtained* cada vez que se realiza una captura de los sensores. Por otro lado, dh_blockDistribution y dh_sensorAccumulation contienen números enteros que representan bloques y el número de sensores para cada módulo y permiten la distribución dinámica de los hilos de la tarjeta gráfica. Los índices de evapotranspiración, se han guardado en dh_EVAPOTRANSPIRATION_INDEXES, arreglo de dobles que una vez inicializados, solo se accede a ellos para lectura. La variable d_TEMPERATURE_INDEXES guarda la temperatura en cada módulo de producción. Para simular el clima y el tiempo de la simulación, se guarda una instancia de la clase `WeatherSimulation` en la variable *weatherSimulation*.

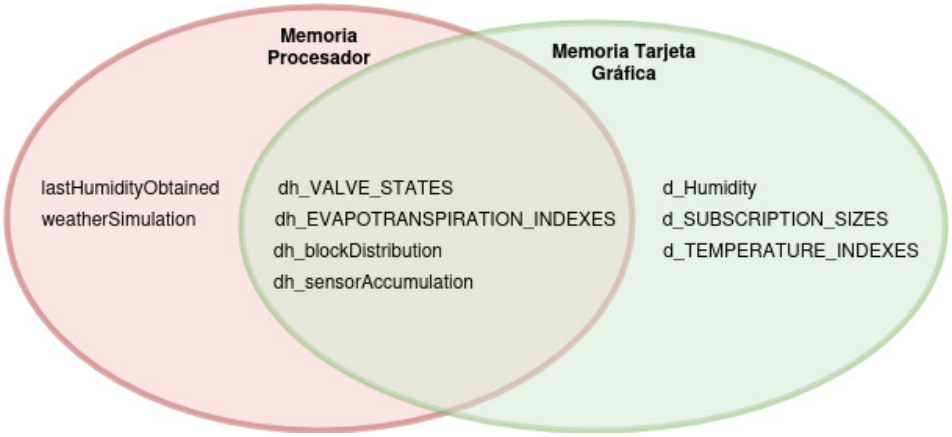

Figura 9: Ubicación en memoria de las variables

En la imagen se pueden ver todos los atributos que forman la clase `WorldSimulation` y la memoria en que residen. Tal y como se menciona previamente en el apartado de la arquitectura, las variables nombradas con el prefijo “d_”, pertenecen a la GPU, las que comienzan por “dh_” son compartidas entre ambos (procesador y tarjeta gráfica), mientras que el resto pertenecen exclusivamente a la CPU.

Para simular la adición o pérdida de humedad en la tierra, se ha hecho referencia a dos de las principales causas de la pérdida de humedad en el sustrato: la evapotranspiración y la percolación. La primera hace referencia a la pérdida de agua en forma de vapor de la superficie del suelo y de las hojas de las plantas. Factores ambientales como la temperatura o la humedad del aire afectan directamente a este proceso. Por otro lado, la percolación, se refiere al proceso en el que el agua se infiltra en la tierra dirigiéndose hacia capas inferiores. Estos efectos naturales que se mencionan, son configurables, y permiten crear escenarios con distintos sustratos y plantas.

Por norma general, la retención de la humedad depende en su mayor parte del sustrato. Por ejemplo, no se conserva de la misma manera en un sustrato de tierra que en uno de arcilla. Para simular el comportamiento de la pérdida de humedad, se han replicado los gráficos de mediciones con sensores de humedad basados en el estudio de [38]. El cálculo de la humedad para una iteración *n*, se realiza en la función `simulationIterate`, que recurre a `humidityCalculations` para aplicar las funciones que se muestran a continuación:

$$\Delta H = |Ti \cdot (Hi + Ei)|$$

Símbolo	Descripción
`ΔH`	Pérdida de humedad en la iteración.
`Ti`	Temperatura interna del módulo de producción.
`Hi`	Índice constante de percolación del agua por iteración.
`Ei`	Índice constante de evapotranspiración del agua por iteración.

$$H_{n+1} = V \cdot W + H_n \cdot e^{-\Delta H}$$

Símbolo	Descripción
`Hx`	Humedad en la iteración x.
`V`	Estado de la válvula (valor 0 si cerrada, 1 si abierta).
`W`	Índice constante de agua desprendida por iteración.
`ΔH`	Pérdida de humedad en la iteración.

Mediante estas formulas, se generan escenarios como el de la siguiente gráfica:

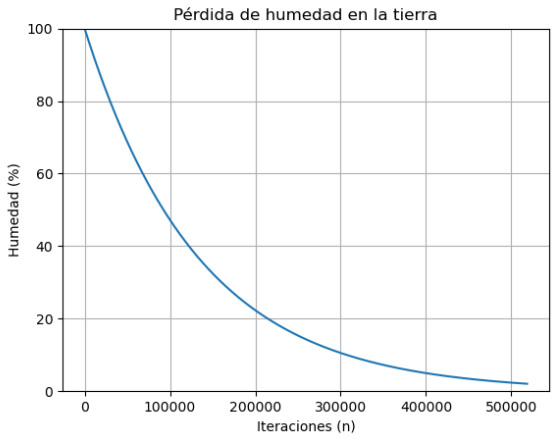

Figura 10: Caso de pérdida de humedad

La gráfica muestra la evolución del porcentaje de humedad en el sustrato de un módulo de producción a 25 ºC durante 6 dias. Los índices de evapotranspiración y percolación que se han usado son de 0.0000001 y 0.0000002 respectivamente. En el ejemplo se ha realizado una iteración por segundo, equivalente a hacer 86.400 iteraciones diarias.

Nótese como inicialmente se tiene una humedad del 100%, que conforme va incrementando el eje x, se va reduciendo la pendiente progresivamente, respetando el límite inferior de la ecuación en 0.

La lógica de la simulación, así como la del productor, está definida dentro de la función `initiateServer`, de la clase `WorldSimulation`. En esta función se genera el entorno, calculando los bloques necesarios de la GPU e iniciando los contadores y temporizadores. De seguido, se cargan las propiedades del productor Kafka y se calcula el tiempo en que se ejecutará la próxima adquisición de datos, resultado de multiplicar el número de la iteración por el tiempo real que transcurre entre cada captura. Ya dentro del bucle, primero se computa el tiempo en que se deberá iniciar la próxima iteración. Para ello, se sigue la siguiente fórmula:

$$T_n = (n+1) \times \left(\frac{S}{V}\right)$$

Símbolo	Descripción
`Tn`	Tiempo de ejecución que corresponde a la iteración *n*.
`n`	Número de iteración en la simulación.
`S`	Segundos transcurridos entre cada iteración.
`V`	Multiplicador de la velocidad de simulación.

Una vez calculado, se entra al bucle de las iteraciones, donde cada iteración de la simulación es equivalente a SECONDS_PER_ITERATION segundos reales.

Lo que sigue es la sección crítica, una región de código sensible a la concurrencia que puede dar problemas de escritura y lectura si no se controla. La interacción de la API con la simulación, a la vez que se modifican sus datos, obliga a establecer estos mecanismos de sincronización de exclusión mútua. Para ello, al introducirse a la sección crítica, se bloquea el *mutex*.

Al entrar en esta sección, primero se actualiza el clima y el tiempo a través de la clase `WeatherSimulation`, explicada en profundidad más adelante. Una vez calculado el clima, se trasladan sus datos a la memoria de la GPU, para posteriormente calcular los nuevos valores de humedad.

Se comprueba la condición de sensado mediante una serie de temporizadores que determinan si se debe capturar el estado de los sensores. En caso afirmativo, se copian los datos al procesador, aumenta el contador de sensados realizados y calcula el siguiente tiempo de sensado. A continuación, el productor procede a hacer el envío de los datos tras recibir los mensajes formateados a JSON de los microcontroladores (simulado mediante la función `cookJSONbyPlan`), para finalmente desbloquear el *mutex* y volver a comenzar el ciclo tras transcurrir el tiempo de espera para la siguiente iteración.

La lógica de la simulación explicada se ve representada en el siguiente diagrama de flujos:

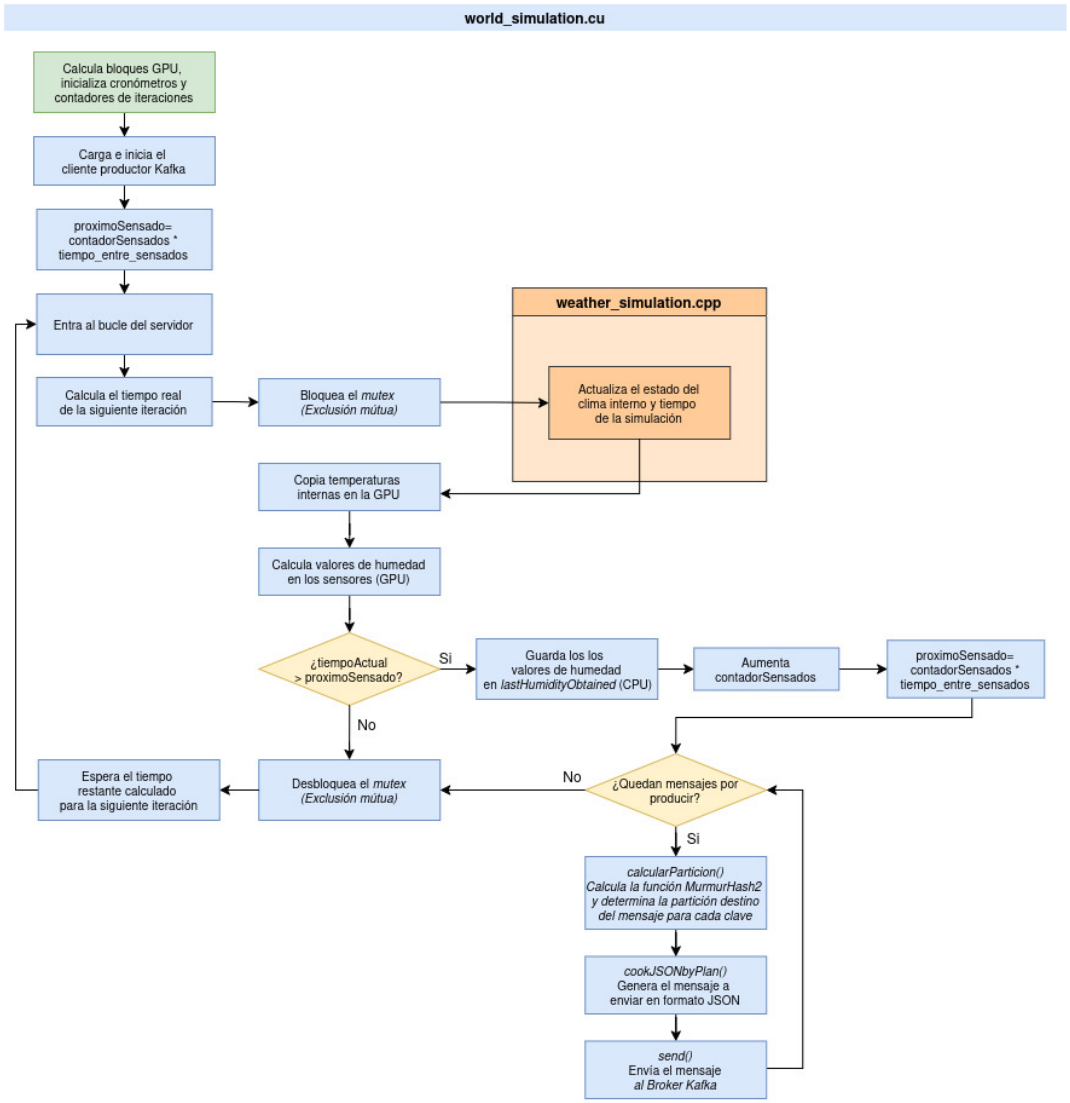

Figura 11: Funcionamiento del productor

La imagen muestra la lógica a bajo nivel que sigue la ejecución de la simulación gestionada por la clase `WorldSimulation`: desde la inicialización de las variables y el uso de temporizadores para controlar la simulación, hasta la interacción entre la memoria CPU-GPU y la incorporación del productor dentro de la simulación.

weather_simulation.cu

Simula el clima y gestiona el estado temporal de la simulación mediante la clase `WeatherSimulation`, definida en la cabecera *weather_simulation.hpp*.

A diferencia de la clase `WorldSimulation`, `WeatherSimulation` se ejecuta solamente en el procesador. En el constructor de la clase, se establece la simulación en el año 2000, mientras que el día, viene definido por la variable de configuración INITIAL_DAY, a partir de la cual se obtiene la estación del año. Para simplificar la simulación, se ha supuesto que cada año está conformado por 12 meses de 30 días cada uno.

En el constructor también se inicializan las tres matrices principales del fichero: PLAN_TEMPERATURES, encargada de guardar la temperatura interna de cada módulo en tiempo real, FAN_STATES, que almacena el estado del sistema de ventilación (1 si abierto, 0 si cerrado), y HEATING_COOLING, que indica si la calefacción o el aire acondicionado están encendidos (1 y -1 respectivamente) o apagados (0).

Tanto FAN_STATES y HEATING_COOLING, como dh_VALVE_STATES (del fichero *world_simulation.cpp*), guardan el estado del sistema HVAC en la simulación. Cualquier cambio en su estado, se refleja en estas variables.

El cálculo de los valores del clima y tiempo en cada cada día de la simulación viene dado por el método `calculateNextDay`. Inicialmente, se actualizan las variables temporales: año, mes, día del año, día del mes y dia de la semana. Estos parámetros residen en una estructura `tm`, de la librería *ctime*, que ofrece funciones para transformar los datos a formato *aaaa/mm/dd hh:mm:ss*. El siguiente paso es actualizar la estación del año, para lo que se usa el arreglo de enteros ENDING_STATION_DAY (definido en *server_utilities.hpp*), que guarda el último día de cada estación del año.

A continuación se procede a calcular la temperatura promedio del día. Para ello se han usado funciones sinusoidales que replican climas estacionarios y permiten generar la temperatura promedio de cada día del año. La funcion en cuestión es la siguiente:

$$T_n = V \times \sin\left(\frac{n \cdot \pi}{180} + A\right) + Tp$$

Símbolo	Descripción
`Tn`	Temperatura promedio en el dia *n*.
`V`	Varianza climática, amplitud de la función, equivalente a la diferencia entre la temperatura promedio y la temperatura máxima en un año.
`n`	Día del año (entre 1 y 359).
`A`	Índice constante de ajuste estacionario.
`Tp`	Temperatura promedio anual.

A continuación se muestra la temperatura promedio para cada día del año en la configuración por defecto:

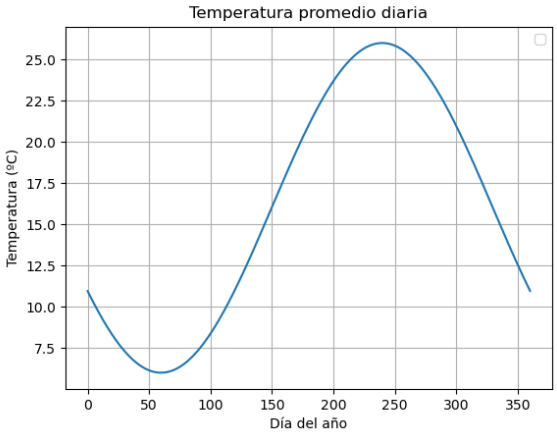

Figura 12: Caso de temperaturas promedios en un año

En el gráfico se muestra el caso de ejemplo para la configuración predeterminada del fichero, donde se intenta replicar el clima de Madrid usando los valores siguientes: *V=10.0, A=179.6, Tp=16*.

Para el cálculo de la temperatura máxima y mínima diaria, se usa la temperatura promedio resultante de la función previa. Mediante el uso de la librería *stdlib.h*, se añade un componente de pseudoaleatoriedad que junto al arreglo SESGO_ESTACIONAL, se obtiene la temperatura máxima y mínima del día. El cálculo de la tempertura para cada hora se calculará usando la mitad de la diferencia entre estas temperaturas.

Acto seguido, se procede a generar los datos a precisión horaria. Para ello, se ha usado la siguiente ecuación:

$$T_n = D \times \sin\left(\frac{h \times 2 \cdot \pi}{24} + 9.3\right) + D + Tm_n$$

Símbolo	Descripción
`Tn`	Temperatura promedio del día *n*.
`D`	Diferencia entre la temperatura máxima y mínima diaria dividida entre dos.
`h`	Hora del día (entre 0 y 23).
`Tm`	Temperatura mínima del día *n*.

La siguiente gráfica muestra un caso de ejemplo de esta ecuación.

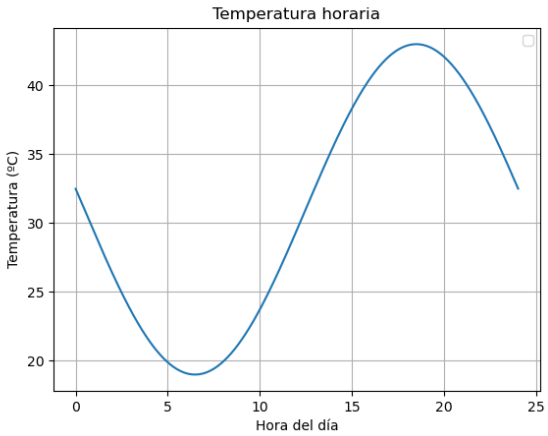

Figura 13: Caso de temperaturas en un día

En el gráfico se muestra el caso en el que: *D=12, Tm=19*. Nótese como la temperatura mínima del día es 19 y la máxima 43 (equivalente a 2*D + Tm).

El último escenario que se simula es el de la iluminación. Para evitar cambios espontáneos del clima, se calcula para cada 6 horas.

El modelo se basa en probabilidades configuradas por el usuario para cada estación, y sigue la siguiente fórmula:

$$L_h = \left(\frac{Lmax}{2} \cdot \sin\left(\pi \cdot \frac{2 \cdot h}{24} + 10.4\right) + \frac{Lmax}{2}\right) \cdot \frac{100 - Lp}{100}$$

Símbolo	Descripción
`Lh`	Iluminación en una hora *h*.
`Lmax`	Iluminación máxima estacional (%).
`h`	Hora del día (entre 0 y 23).
`Lp`	Iluminación en el periodo de 6 horas actual (calculado previamente con pseudoaleatoriedad).

A continuación se muestra un caso de ejemplo aplicando esta ecuación:

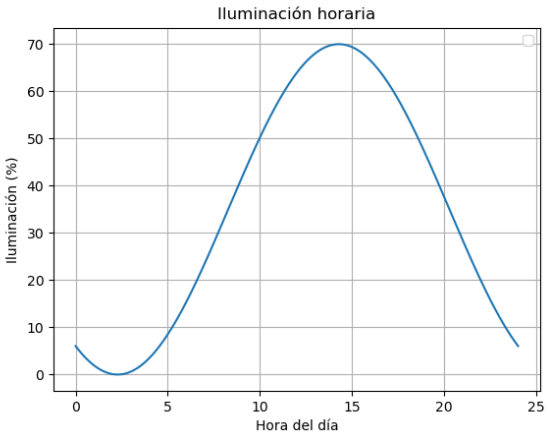

Figura 14: Caso de iluminación en un día

El gráfico muestra cómo evoluciona la iluminación en un día de invierno con cielos despejados. Nótese cómo durante el periodo nocturno, la iluminación se mantiene en mínimos, mientras que sobre las seis de la mañana empieza a aumentar progresivamente.

Dentro de la lógica de la simulación, se han configurado algunos límites de diseño para mantener el entorno en valores controlados. El primer ajuste afecta al rango de la temperatura interna en los módulos, que se mantendrá entre -10 y 50 ºC. Esta restricción es necesaria, ya que al tratarse de fórmulas lineales las que simulan la temperatura, no cuentan con límites, por lo que se ha restringido al rango mencionado. El otro límite se ha establecido manualmente en los valores de las humedades, que deben permanecer entre 0 y 100. Al igual que sucedía en la temperatura, la función de riego afecta de forma lineal a la ecuación, ocasionando que las humedades superaran el 100, por lo que se ha añadido una condición para que no los supere en ningún caso. Cabe recalcar que en este caso, ajustar el mínimo no ha sido necesario puesto que la ecuación que se sigue en la pérdida de humedad, tiene $\lim_{x\to\infty} f(x) = 0$.

Para optimizar el código del fichero, se ha hecho uso de mecanismos de vectorización y palalelismo facilitados la librería OpenMP. Los valores de las 24 horas del día se han calculado a través de la creación de una región paralela, a través de la directiva *#pragma omp parallel* y la planificación de hilos estática en el bucle, mediante *#pragma omp for schedule(static)*.

Por otro lado, también se han usado mecanismos de vectorización para actualizar la temperatura de cada módulo de producción. Para ello se ha usado la opción *#pragma omp simd*. Esta optimización puede ser de ayuda en casos en los que hay una gran cantidad de módulos de producción, reduciendo ligeramente los ciclos de procesador consumidos en la tarea.

Configuración de la simulación

El comportamiento de la simulación puede ser configurado desde el archivo *server_utilities.h*. En esta cabecera, se pueden modificar una multitud de parámetros que controlan el clima, la velocidad de la simulación, o la influencia del entorno en la simulación.

Estos parámetros se encuentran ordenados en función del ámbito de la simulación al que afectan. A continuación, se procede a explicar brevemente estos grupos:

- **Ambientales**

Cuenta con variables que permiten establecer constantes que afectan a todo el sistema. Desde estas variables se puede asignar la humedad inicial en las plantas del invernadero, los índices de percolación del sustrato e incluso modificar la velocidad a la que se humedece el sustrato y a la que influye calefacción y el aire acondicionado.

- **Temporales**

Permite alterar el comportamiento de la simulación al margen del tiempo real. Se especifica el tiempo real transcurrido entre iteración (SECONDS_PER_ITERATION), los segundos que pasan entre capturas de los sensores (TIME_SIMULATED_BETWEEN_SENSORANALIS), y la velocidad de reproducción de la simulación respecto al tiempo real (TIME_SPEED), que actúa como un multiplicador que modifica el tiempo simulado respecto al real.

Uno aspecto a destacar es que TIME_SIMULATED_BETWEEN_SENSORANALIS, debe ser múltiplo de SECONDS_PER_ITERATION. De este modo se asegura que la captura de los sensores muestra los datos más actualizados y se permiten mejoras de rendimiento. Inicialmente, el simulador y la captura de sensores se diseñaron para ejecutarse en hilos diferentes. Una consecuencia de esto, era la necesidad de recurrir a mecanismos de sincronización que evitara problemas de escritura o lectura, comunes en sistemas concurrentes. Para mediar con ello, se decidió establecer esta restricción, ya que permitió convertir el código en secuencial, eludiendo el uso de mecanismos de sincronización para la captura de los sensores. A pesar de ello, los mecanismos de sincronización han sido necesarios igualmente, ya que la simulación se ejecuta en paralelo con la API.

- **Parametros de módulo**

Son variables que afectan a cada módulo por separado. Para cada uno se definen el número de plantas existentes, el índice de evapotranspiración, la temperatura interna, y el estado inicial del sistema HVAC.

De este modo, la simulación permite crear escenarios con plantas de diferente consumo hídrico, y ofrece pleno control del estado inicial de cada módulo de producción.

- **Climáticos**

Conjunto de variables que alteran el clima y cómo interactúa la simulación con este. Cuenta con variables temporales, como INITIAL_DAY, que define el día del año y permite conocer la estación del año. También se definen variables que afectan a las funciones seno vistas previamente, tales como la temperatura promedio anual, varianza climática o el sesgo estacional.

Los parámetros disponibles en la configuración del escenario se muestran a continuación.

Parámetro	**Descripción**
HUMEDAD_INICIAL	Porcentaje de humedad de los sensores al inicio de la simulación.
HUMIDITY_LOSS_PER_SECOND	Constante de la pérdida de humedad del sustrato por segundo.
WATERING_INDEX_PER_SECOND	Constante de cantidad de agua que desprenden las válvulas por segundo.
HEATING_COOLING_INDEX_PER_SECOND	Constante de variación de la temperatura en la calefacción y refrigeración.
SECONDS_PER_ITERATION	Segundos transcurridos entre cada iteración.
TIME_SIMULATED_BETWEEN_SENSORANALIS	Segundos transcurridos entre cada captura de los sensores (múltiplo de SECONDS_PER_ITERATION).
TIME_SPEED	Velocidad de simulación, cuantas veces se acelera el tiempo.
SUBSCRIPTION_SIZES	Arreglo con el número de sensores que se quiere simular para cada módulo de producción.
EVAPOTRANSPIRATION_INDEXES_PER_SECOND	Arreglo con las constantes de pérdida de humedad de las plantas de cada módulo de producción por segundo.
INITIAL_FAN_STATES	Arreglo con el estado inicial de la ventilación para cada módulo de producción (1 encendida, 0 apagada).
INITIAL_VALVE_STATES	Arreglo con el estado inicial de las válvulas para cada módulo de producción (1 encendida, 0 apagada).
INITIAL_TEMPERATURE_PLAN_INDEXES	Arreglo con la temperatura inicial en cada módulo de producción
INITIAL_DAY	Número del día dentro del año (entre 1 y 359).
TEMPERATURA_PROMEDIO_ANUAL	Media de la temperatura en grados celsius durante un año.
VARIANZA_CLIMATICA	Variación deseada entre temperaturas máxima y mínimas diarias.
AJUSTE_ESTACIONARIO	Ajusta las funciones seno al estado estacionario del 1 de enero.
TEMPERATURE_INFLUENCE_INDEX	Influencia de la temperatura exterior a la interior.
SESGO_ESTACIONAL	Variación entre la temperatura máxima y mínima en cada estación.
ILUMINACION_MAXIMA_ESTACIONAL	Porcentaje de luminosidad máxima para cada estación.
REDUCCION_POR_NUBOSIDAD	Porcentajes de reducción de la nubosidad (usado para determinar la luminosidad diaria).
PROBABILIDAD_NUBES_ESTACIONARIO	Porcentaje de nubes diarias promedio para cada estación del año.
ENDING_STATION_DAY	Define los días frontera entre cada estación del año.

Cuadro 1: Parámetros configurables de la simulación

5.2.3 API del productor

Para comunicar la ejecución del servidor local con aplicaciones externas, se ha recurrido a una API REST. En esta, se exponen los recursos del invernadero y permite modificaciones del estado interno mediante los recursos que expone. Se ha hecho uso de *libmicrohttpd*, una librería de código libre para C optimizada para incorporar servidores HTTP dentro de una aplicación.

La API es el punto de entrada del exterior al invernadero. De este modo, los servidores consumidores pueden modificar el estado de la simulación desde una red remota, además de permitir que usuarios en la red local puedan conectarse a la interfaz gráfica y visualizar el estado del invernadero en tiempo real. Los recursos expuestos en la API son los siguientes:

Recurso	**Método**	**Descripción**
/plan/<planId>	GET	Devuelve la última captura de los sensores dentro de un módulo de producción *planId*.
/plan/<planId>	POST	Modifica el estado interno del invernadero. El formato de los datos debe ser *application/x-www-form-urlencoded*. Se disponen de las siguientes opciones: • Activación/desactivación del sistema de temperatura: `COOLHEAT=1` (calefacción), `-1` (aire acondicionado), `0` (apagado). • Activación/desactivación del sistema de riego: `openValve=1` (activar), `0` (desactivar). • Activación/desactivación del sistema de ventilación: `openFan=1` (activar), `0` (desactivar). • Asignación manual de la humedad para una planta del sistema: `humidityGoal=<humedadEstimada>` (en formato flotante) && `sensorID=<sensorId>`

Cuadro 2: Recursos expuestos por la API

A través de estos recursos, la API ejecuta su lógica interna para interactuar con la simulación. Concretamente, hace uso de cuatro métodos protegidos por mecanismos con para modificar el estado interno del sistema. Estos métodos son `openValve`, para los mecanismos de riego, `openFan`, para el sistema de ventilación, `modifyHeatingCooling`, para la calefacción y aire acondicionado, y también `waterIndividual`, a través del cual se puede modificar la humedad para una planta especificada. La obtención de datos de cada módulo viene dada por una versión asíncrona de la función `cookJSONbyPlan` llamada `asyncCookJSONbyPlan`. Todas estas funciones mencionadas, aseguran la exclusión mútua con *mutex*.

Para construir el entorno, la URI de la API debe ser configurada en el archivo *config.yaml* mediante el *tag* `API_URI` y un valor con formato *ip(o nombre de dominio)*:*puerto*, tal y como se muestra en el siguiente ejemplo.

```
API_URI: "192.168.1.55:8888"
```

5.2.4 Broker

El *broker* es el corazón del sistema, la herramienta intermedia entre productor y consumidor. Almacena los mensajes por tópico y partición, para que sean procesados y mediante un sistema de confirmaciones o *commits*, lleva el seguimiento de los mensajes que se consumen. Para instanciar el servidor, se ha hecho uso de la imagen *docker* oficial de Apache Kafka, *apache/kafka:4.0.0* [39], que permite configuración del servidor desde las variables de entorno.

Al entrar en funcionamiento el *broker*, un contenedor con la imagen de *confluent*, *confluentinc/confluent-local:7.9.0* [40] actúa de cliente Kafka e inicializa el tópico "riego". En este tópico se van a guardar los mensajes de los módulos de producción en el *broker*. Inicialmente, se trató de crear un tópico por módulo, pero era contraproducente y dificultaba el diseño de un sistema con redistribucion de la computación. La solución que se encontró, fue dividir los módulos mediante particiones. Apache Kafka cuenta con un sistema de redistribución de particiones que puede beneficiar a un sistema escalable basándose en los *consumer-groups*. Los *consumer-groups* son grupos de servidores que trabajan juntos para leer datos de un tópico con la restricción de que cada partición, solo puede ser asignada a una sola instancia del *consumer-group*. Gracias a esta propiedad, se ha distribuido la responsabilidad de cada módulo de producción entre las instancias consumidoras del sistema de forma única.

En Kafka, las particiones se asignan aplicando funciones hash a la clave de un mensaje, junto a una operación modular que usa el número de particiones de un tópico como divisor. Esto se aprovecha por los consumidores, mediante el uso de diferentes claves para cada módulo de producción. Todos los mensajes con la misma clave pertenecerán a la misma partición durante todo el tiempo de ejecución, por lo que al redistribuir las particiones, se está distribuyendo también la responsabilidad de los módulos de producción.

Para para lograr una distribución compensada de los módulos de producción, se ha configurado el tópico "riego" con tantas particiones como módulos existentes. Esta medida busca mejorar la distribución de las particiones, especialmente en el caso en que haya tantas instancias consumiendo como módulos de producción, donde en el caso ideal, cada consumidor se encargará de un solo módulo.

El funcionamiento interrumpido o ralentizado, incapaz de procesar los datos en el periodo de tiempo entre cada envío de datos, puede introducir una acumulación de mensajes que comprometa el funcionamiento del sistema. El apalancamiento de los mensajes en el *broker* constituye un problema mayor que debe resolverse para asegurar el procesamiento en tiempo real. Para garantizar la eliminación de mensajes obsoletos, en el tópico "riego" se ha establecido un tiempo de vida límite de 20 segundos para los mensajes, una vez transcurrido este periodo, el mensaje será eliminado automáticamente.

En el archivo *config.yaml*, la URI del *broker* debe especificarse para generar la configuración adecuada. Para ello, se debe especificar el *tag* `KAFKA_URI`, seguido de su URI en formato *ip(o nombre de dominio)*:*puerto*, tal y como se muestra a continuación.

```
KAFKA_URI: "kafka-broker:9092"
```

El *broker* se ha configurado para que anuncie diferentes puertos según si el origen de los paquetes. Para las solicitudes internas como los consumidores, se anuncia el puerto 19092, expuesto solamente dentro del clúster, mientras que para las externas, se ha expuesto el 9092. En el clúster de Kubernetes, se define su recurso dentro del archivo *DataStorage.yaml*. Para permitir el acceso del exterior, se ha implementado un recurso *service* de tipo *NodePort*, que permite exponer un puerto al exterior. Concretamente, se ha expuesto el puerto 30.000, por lo que todas las peticiones entrantes a este puerto, serán redirigidas al servicio del *broker*. Además de esta ventaja, el recurso *service* permite la resolución DNS dentro del clúster, donde todos los contenedores pueden interactuar con servicios haciendo uso del nombre del servicio creado. En el caso del clúster que se ha implementado, esto se aprovecha por los consumidores para solicitar los mensajes del *broker*, expuesto al clúster bajo el nombre *kafka-service*.

En la siguiente imagen, se puede visualizar la exposición de los puertos y las conexiones que permite:

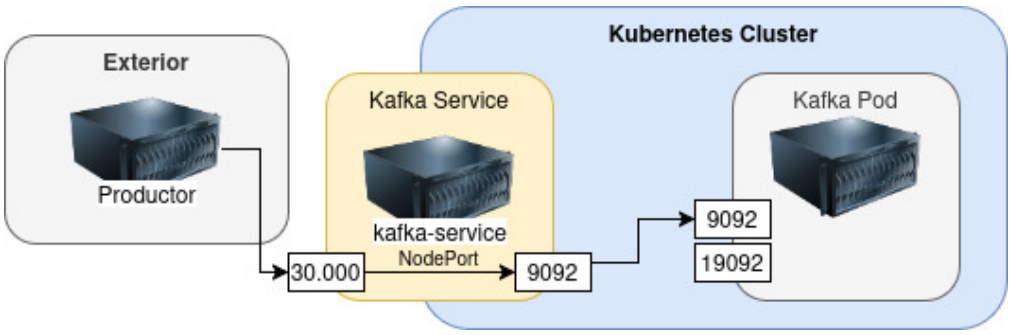

Figura 15: Exposición de los puertos del broker en el clúster

En la imagen anterior se puede cómo queda expuesto el puerto 9092 al exterior mientras el 19092 queda solo visible en el interior del clúster de Kubernetes. Nótese como el *broker* está desplegado sobre un recurso de tipo *pod*, la unidad mínima de despliegue, cuyo comportamiento por defecto en casos de error, es de reinicio.

También se muestra cómo un productor externo puede acceder al clúster mediante el *NodePort* definido, que realiza la redirección de los paquetes hacia el *pod* del *broker*.

5.2.5 Consumidor

Los servidores consumidores son el cerebro del sistema. Su función es gestionar grandes volúmenes de datos y procesarlos con los algoritmos configurados. Una vez procesados, se encargan de comunicar las decisiones directamente al servidor local del invernadero. Estos servidores se gestionan desde un *deployment*, un recurso sin estado que se ha configurado para escalar según el consumo de CPU. Para evitar pérdidas de datos, se ha establecido el número mínimo de réplicas del despliegue a dos, de modo que en caso de errores, la instancia activa pueda recuperar la información. En este *deployment*, al variar el número de instancias, comienza un proceso de redistribución de particiones, donde cada consumidor guarda su estado en la base de datos con la finalidad de ser replicado por otras instancias tras la repartición.

Para generar el ecosistema que permita la creación de algoritmos sin límites, se ha decidido usar el lenguaje de programación Java, uno de los lenguajes más utilizados en entornos empresariales dada su facilidad, seguridad y compatibilidad. Gracias a la máquina virtual de java (JVM), los proyectos Java puede ejecutarse prácticamente en cualquier sistema, siendo uno de lenguajes más sencillos de desplegar en la actualidad [41]. Asimismo, las funcionalidades del consumidor se han realizado con la API oficial de Apache Kafka para Java [42], que ofrece las clases necesarias para interactuar con el *broker*.

También se ha usado Maven [43] como herramienta de gestión de ficheros. Esta herramienta facilita la construcción del programa mediante un fichero de gestión de dependencias (*pom.xml*). Cualquier librería externa a Java usada en los algoritmos, debe ser especificada en este archivo para que se construya correctamente. Para la conversión de los mensajes de kafka a objetos, se ha incorporado la librería `ObjectMapper`, que permite transformar los mensajes JSON residentes en el *broker*, en un objeto `SensorData`, con los valores del invernadero necesarios para ejecutar los algoritmos.

El servidor consumidor es el único componente del sistema para el que se requieren conocimientos técnicos. El usuario querrá añadir sus propios algoritmos, una vez programados. Para ello, lo primero que se debe hacer es añadir de las clases dentro de la carpeta `algorithms`, que es el único directorio que debe ser modificado por el usuario. Posteriormente, se tienen que añadir de las dependencias en el *pom.xml*, fichero usado por Maven para la construcción del entorno. Por último, el usuario deberá configurar el archivo *config.yaml*. En este archivo de configuración, se debe incluir el arreglo *plans*, con cada uno de los algoritmos usados, especificando el nombre que se le quiere dar en el *tag name*, seguido del nombre de la clase del algoritmo, en el *tag algorithm*. A continuación, se muestra un ejemplo de configuración.

```yaml
plans:
  - name: "Mediterráneo"
    algorithm: "PlanSeco"
  - name: "Tropical"
    algorithm: "PlanHumedo"
  - name: "Desiértico"
    algorithm: "PlanÁrido"
```

Asimismo, también deben especificarse el número de planes que se han incorporado, especificando el *tag* `NUM_PLANS`, que se usará por el archivo constructor del entorno para generar la configuración de los consumidores y la del productor. A continuación se muestra un ejemplo:

```
NUM_PLANS: 6
```

Para su implementación, se ha hecho uso de tres clases principales: `GreenhouseConsumer`, `Task` y `BasicPlan`. Estas clases se detallan en mayor profundidad a continuación:

GreenhouseConsumer

Es la encargada de realizar todas las comunicaciones necesarias con el *broker*. Desde esta clase se gestiona la solicitud de registros al *broker*, su distribución en tareas, así como las confirmaciones finales de los desplazamientos.

Dentro del constructor de la clase se configuran los parámetros del servidor mediante un objeto `Properties`, a partir del cual, se instancia el `KafkaConsumer`, necesario para empezar a consumir mensajes.

En el objeto properties se ha especificado el uso de `StringDeserializer` para que los mensajes se reciban en una cadena `String`, que será transformada más adelante. También se han desactivado las confirmaciones automáticas (*auto commit*), realizadas manualmente desde el código. Tal y como se ha explicado en el subapartado anterior, los grupos de consumidores tienen una función crucial en la distribución de las particiones. Para aprovechar que las particiones solamente se asignan a una instancia del grupo, se ha configurado bajo el grupo *irrigategroup*, *consumer-group* al que pertenecerán todos los consumidores.

Otro aspecto clave a configurar es la estrategia de particionado. La estrategia por defecto es *RangeAssignor*, cuyo criterio para asignar las particiones a los consumidores es el reparto equitativo de las particiones entre las instancias existentes. Sin embargo, esta estrategia no es óptima en casos como el que se plantea, donde interesa que un consumidor mantenga el máximo de distribuciones para conservar su estado. Al asignarse una lista de nuevas particiones en cada distribución, en cada rebalanceo, los consumidores se verían obligados a migrar los datos de todos los módulos. Para reducir estas migraciones, se ha recurrido a la estrategia *CooperativeStickyAssignor*, que además de permitir la cooperación entre consumidores, permite una redistribución balanceada, a la vez que busca la conservación del máximo número de particiones posible. Tamién se ha modificado el *auto.offset.reset*, propiedad que define el comportamiento del consumidor cuando el grupo de consumidores al que pertenece, no cuenta con un *offset* al que adherirse, afectando principalmente al inicio de la ejecución. Para que desde el inicio se lean los mensajes en tiempo real, se ha configurado a “latest”, de modo que comenzará a leer por el último mensaje del *broker*. La última de las configuraciones modificadas, es *fetch.max.wait.ms*, responsable de definir el tiempo máximo de bloqueo del *broker* antes de enviar un paquete de datos. Se ha establecido a un segundo.

Para su implementación, se han usado una serie de objetos que permiten el funcionamiento del sistema. Dentro del constructor, se inicializa *keysPerPartition*, un mapa que contiene todas las claves de módulo que pertenecen a cada partición. Para su determinación, se ejecuta el algoritmo *MurmurHash2* con todas las claves posibles en el escenario, guardando la partición resultante dentro del mapa.

La clase `GreenhouseConsumer`, se ocupa de guardar todo objeto en memoria que procese algoritmos. Todos estos objetos se guardan como valor dentro del mapa *plans* bajo la clave de módulo respectiva. Además, para instanciar estos objetos, se ha usado el mapa *constructors* que contiene los constructores de cada una de las clases que procesan algoritmos y se han configurado en el fichero *config.yaml*. Para inciar este mapa, se ha requerido un preprocesamiento que añade los constructores vinculados a cada clave de módulo. Dentro del código, esto se añade en la macro @MACRO_CONSTRUCTORS.

Para llevar un seguimiento de las tareas activas, se ha hecho uso del mapa *activeTasks*. Y la confirmación de los desplazamientos, viene dada por el mapa *offsetsToCommit*.

Una vez se ha inicializado el consumidor, la clase procede a ejecutar tres funciones en bucle: `handleFetchedRecords`, `checkFinishedTasks` y `commitOffsets`.

La primera función se encarga de agrupar los mensajes perteneciente a la misma partición, y ejecutarlos en *Tasks*, desde la creación de la tarea hasta su finalización, se pausa la obtención de mensajes para la partición. Para realizar un seguimiento a estas tareas, se añaden al mapa *activeTasks*. Una vez añadidas, se ejecutan en un *pool* de hilos en segundo plano, basado en la clase de Java `ExecutorService`.

A continuación, la función `checkFinishedTasks` revisa los estados de las tareas activas con la finalidad de registrar todos los *offsets* para que sean confirmados posteriormente. Además al finalizar estas tareas, se permite la solicitud de mensajes para la partición finalmente procesada.

En la última fase se confirman los offsets necesarios registrados en el mapa *offsetsToCommit* para posteriormente vaciarlo.

En la siguiente imagen se muestra el diagrama de flujos resultante:

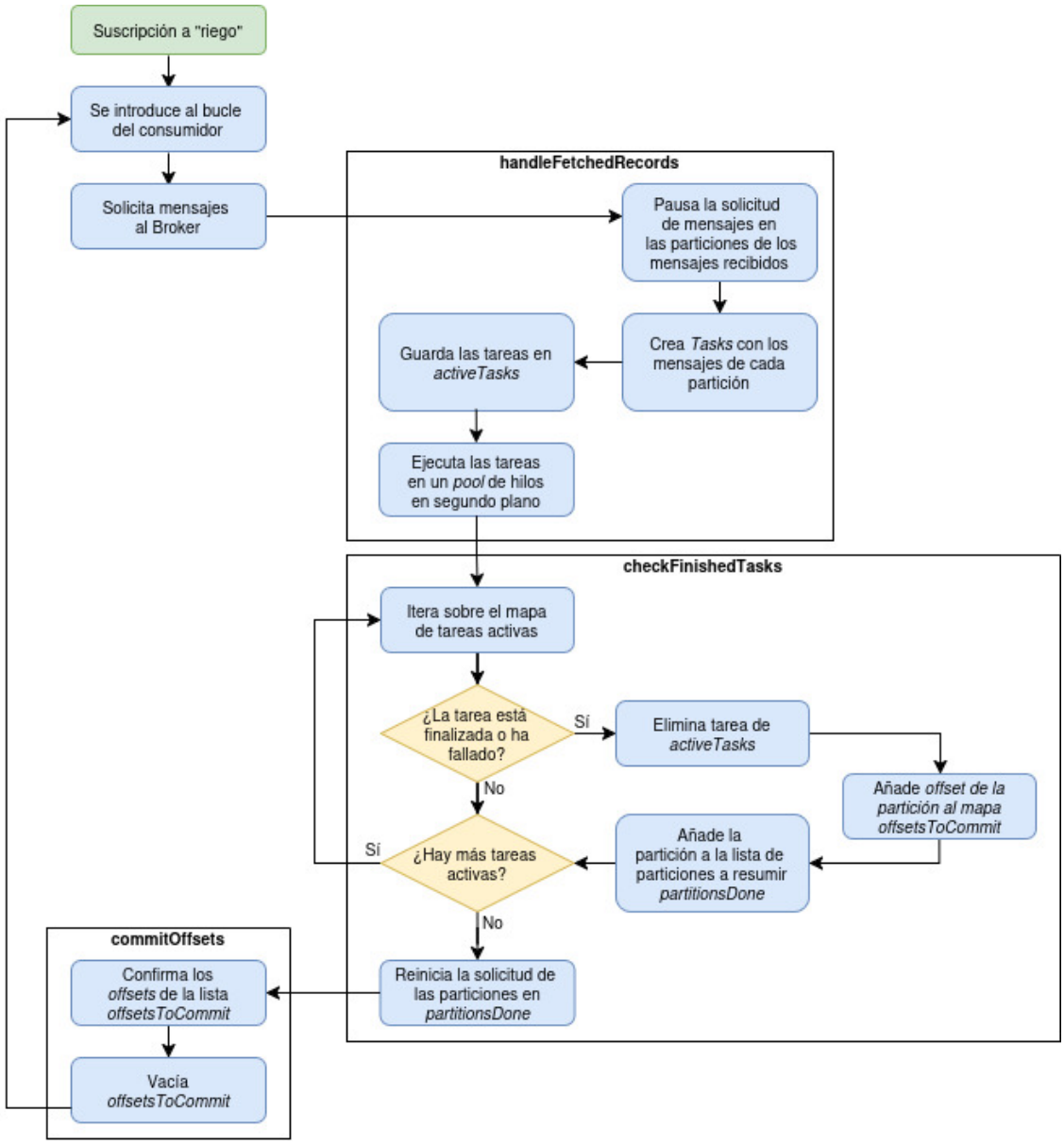

Figura 16: Diagrama de flujo GreenhouseConsumer

La imagen insertada contiene la lógica que sigue la clase `GreenhouseConsumer` basada en las tres funciones principales comentadas previamente.

Una vez explicada la lógica que sigue el consumidor para generar el flujo de procesamiento de los datos, se procede a explicar cómo se ha logrado la migración de las particiones entre las instancias.

Al variar el número de instancias, los consumidores comienzan un proceso coordinado de redistribución de las particiones. Para estas situaciones, la API de Kafka ofrece la interfaz `ConsumerRebalanceListener`, que permite controlar el programa en estos escenarios mediante tres funciones principales que se explican a continuación:

- onPartitionsRevoked

Se ejecuta inmediatamente en los consumidores al solicitar mensajes al *broker* durante el proceso de redistribución de las particiones. Recibe como argumento de entrada una colección de las particiones que va a dejar de gestionar tras la distribución.

En esta función se espera a que finalicen las tareas activas y se confirman los nuevos desplazamientos. Posteriormente, se procede a guardar toda la información persistente del algoritmo a la base de datos para finalmente eliminar los objetos del mapa *plans*, eliminando su única referencia existente en memoria y permitiendo que el recolector de basura de Java libere su espacio.

- onPartitionsAssigned

Llamada de manera coordinada por todos los consumidores una vez que todos han finalizado la ejecución de `onPartitionsRevoked`. Incluye como argumento una colección con las nuevas particiones asignadas que no tenía previamente.

Para cada una de las particiones, recorre el mapa *keysPerPartition* e inicializa los objetos de los módulos de producción asignados a cada partición. Actualiza los objetos con los últimos valores guardados en la base de datos.

- onPartitionsLost

Las redistribuciones, rara vez pueden dar lugar a errores. Un ejemplo puede ser la denegación de una partición no anunciada. Esto puede dar lugar a que un consumidor trate de solicitar mensajes de una partición que realmente no es de su responsabilidad. En estos casos, de pérdidas inesperadas, se ejecuta esta función, donde al igual que en la función `onPartitionsAssigned`, se va a eliminar cualquier referencia a los objetos de las particiones eliminadas.

Task

La clase `Task` añade una capa intermedia que realiza el procesamiento en segundo plano. Su principal tarea es informar a la clase `GreenhouseConsumer` del estado de procesamiento de las particiones que gestionan.

`Task` implementa la interfaz `Runnable`, que permite la ejecución concurrente de su método *run*. En esté método, se ejecuta la función `proccessRecord`, seguida de un aumento del *offset* al finalizar. La función `processRecord`, se encarga de ejecutar en el orden requerido las funciones de ejecución de los algoritmos, funciones de la clase abstracta `BasicPlan`. Gracias a esta clase abstracta, los usuarios pueden incluir sus propios algoritmos personalizados que se adapten a la plataforma.

Tal y como se ha visto, la clase `GreenhouseConsumer`, consulta asíncronamente el estado de las tareas. Para ello, `Task` cuenta con las funciones `isFinished` e `isFailed`, a través de las cuales, se comprueba el estado de las tareas, así como la naturaleza de su finalización. Finalmente se ha introducido la función `getCurrentOffset`, a la que recurre el consumidor para conocer el desplazamiento en tiempo real.

A continuación se muestra un diagrama con su funcionamiento:

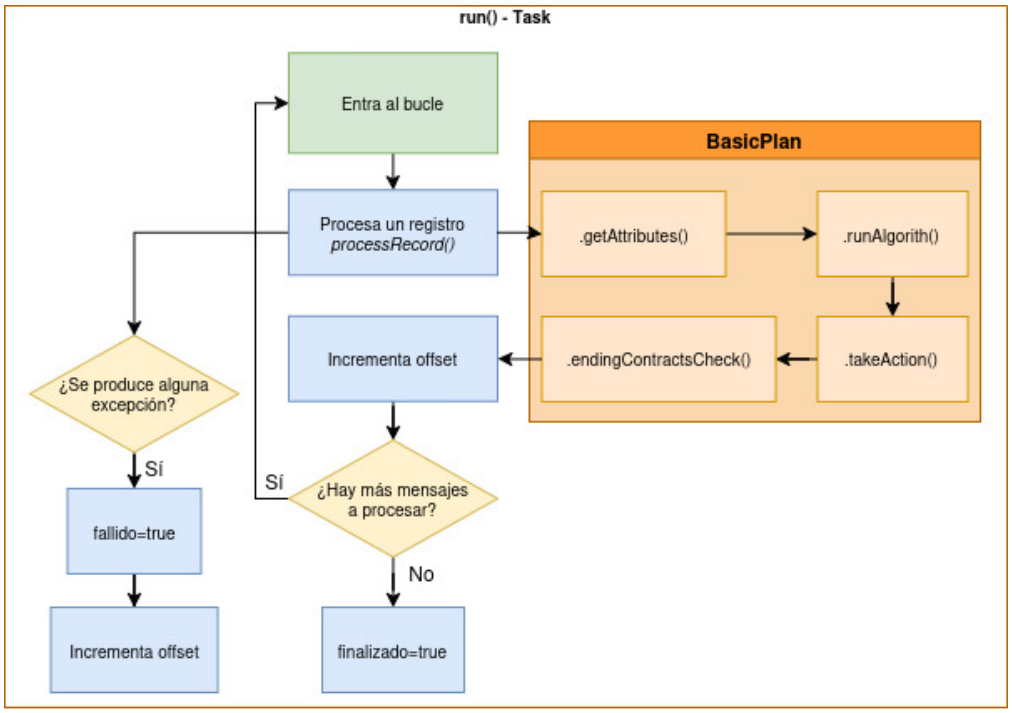

Figura 17: Diagrama de flujo Task

En este diagrama de flujos se puede seguir la lógica que ejecuta cada `Task` desde su creación hasta su finalización. Nótese como a pesar de darse excepciones, la ejecución continúa su funcionamiento aumentando el desplazamiento. También se muestra el orden en el que se ejecutan los algoritmos mediante la clase `BasicPlan`.

BasicPlan

Es una clase abstracta, que permite encapsular los algoritmos del usuario para que puedan ejecutarse en el entorno implementado. Cuenta con una serie de atributos para facilitar la creación de los algoritmos. Con visibilidad *protected*, estos atributos son visibles para las clases que la extienden, y constituyen las bases a partir de las cuales se diseñan estos algoritmos.

El atributo *sensorData* pertenece a la clase `SensorData`. Mediante este atributo se pretende que los algoritmos puedan mantener actualizados todos los datos de un módulo de producción. Las `Task` invocan a los objetos `BasicObject` introduciendo como argumento un objeto `SensorData`. La clase `SensorData` contiene la información de los mensajes de los módulos de producción deserializada a un objeto. Estos atributos son el identificador del módulo (*planId*), la fecha y hora del mensaje (*datetime*), el estado de las válvulas (*openValve*), el estado del sistema de temperatura (*coolHeat*), la temperatura interna (*temp*) y una colección de humedades (*humidity*). Todas están encapsuladas y pueden accederse mediante *getters*.

Para interactuar con la API, se ha introducido la clase `GreenhouseAction` Para modificar sus valores, la clase cuenta con los siguientes *setters*: `setOpenValve, setOpenFan y setCoolHeat`. A través de ellos, los usuarios pueden usar sus propias acciones dentro de las clases. Un caso de uso es en los contratos, que deben devolver acciones.

Para mayor claridad en el código, se recomienda hacer uso de la variable *action*, un objeto `SensorData` que se ejecuta una vez procesado el algoritmo. Tras esto, la acción se restablece en valores nulos para poder ser reusada.

La estructura también ofrece la posibilidad de establecer condiciones que se mantengan en el tiempo. Estas condiciones, se llaman contratos, y se ejecutan tras cada procesamiento de los datos para comprobar cuales se cumplen. Estos contratos, se programan en forma de *callback* mediante la clase abstracta `DataCallback`, y permiten crear lógicas que se revisan constantemente. Los contratos deben devolver una acción, en caso de haberse cumplido, o un objeto nulo en el caso contrario. Una vez cumplido, se ejecuta la acción, y se elimina el contrato, dejando la posibilidad de activarlo posteriormente.

Estos contratos deben registrarse en el mapa *contractsRegistered* con un identificador como clave, de modo que para activar un contrato específico, se debe añadir su identificador a *contractsActive*.

La Base de Datos es un componente fundamental para la migración de datos. Para interactuar con ella, se ha facilitado la clase `DBHashMap`, que permite introducir datos a una base de datos compartida entre todas las instancias. Esta clase hace la función de `HashMap` para los tipos `String`, `Integer`, `Double` y `Boolean`. También ofrece soporte para listas, no obstante la compatibilidad de los tipos contenidos con el serializador de la API de MongoDB, queda a responsabilidad del usuario. Desde esta clase se ofrece la opción de realizar las operaciones en local (`getFromLocal, putOnLocal`) , o con la base de datos (`getFromDB, putOnDB`). Además, hace posible la migración de los datos, mediante las funciones `getBackup` y `setBackup`, que pueden gestionarse automaticamente si las clases de los algoritmos establecen el atributo *autoMigration* a *true* en el iniciado del objeto (función `onStart` o constructor). Para mayor control de la base de datos, también se puede hacer uso del atributo *collection*, no obstante se requieren un mayor conocimiento de la API de MongoDB.

Esta clase encapsula los algoritmos del usuario mediante cuatro funciones primordiales que se eje-

cutan secuencialmente en el siguiente orden: `getAttributes, runAlgorithm, takeAction y endingContractsCheck`.

El procesamiento de los mensajes comienza en la función `getAttributes`, donde se recibe como argumento un objeto *capturedData* (de tipo `SensorData`) con los contenidos del mensaje recibido. En esta función, el usuario debe preparar el entorno para la ejecución del algoritmo.

Posteriormente, en `runAlgorithm`, entra en acción el algoritmo. Las decisiones que se concreten, pueden ejecutarse manualmente mediante un objeto `GreenhouseAction` y su posterior ejecución con `execute`, no obstante, se recomienda usar en el atributo *action* visto previamente, puesto que al finalizar `runAlgorithm`, se ejecutará su contenido desde `takeAction`.

Por último se computa `endingContractsCheck`, donde se comprueban todos los contratos existentes. En el caso de que se cumplan, automáticamente son eliminados de la lista `contractsActive`, y se introducen en la base de datos para permitir su migración entre las instancias.

5.2.6 Base de Datos

La persistencia de datos en el sistema viene dada por una base de datos no relacional implementada con MongoDB. En el despliegue se ha usado la imagen oficial *mongodb/mongodb-community-server:latest* [44]. En el clúster de Kubernetes está definido su recurso en el archivo *DataStorage.yaml*. La base de datos, se despliega en un *deployment* definido con el mínimo número de réplicas de una instancia, que asegura la existencia permanente de la base de datos incluso en casos de error. Además, se expone al clúster a través de un *service* bajo el nombre *mongodb-server*, a la que los consumidores acceden por el puerto 27017. En el escenario, los datos se almacenan en la base de datos “riego”.

La base de datos es un componente clave en la migración de los contratos (*callbacks*), puesto que en caso de errores dentro de los consumidores, estos proceden a guardar sus datos, que pueden ser recuperados de inmediato por otras instancias.

La base de datos de MongoDB se basa en colecciones, similar a las tablas en bases de datos relacionales con la diferencia de que en lugar de datos estructurados, contiene objetos JSON (llamados “documentos”). A través de estos documentos, el usuario puede introducir objetos serializados y recuperarlos desde otras instancias.

Dentro de los algoritmos, se facilita el uso de la clase `DBHashMap` que facilita interactuar con la base de datos. Además el usuario también puede hacer uso del objeto de la colección, a través de la cual se pueden realizar operaciones más personalizadas con la base de datos. Cada algoritmo, gestiona exclusivamente su colección, bajo el nombre equivalente a la clave de módulo, creando un entorno aislado del resto de módulos.

Para evitar conflictos de nombrado entre los documentos de una colección, la clase `DBHashMap` utiliza dos *tags* reservadas: *_key* y *_value*. Además, se ofrece la opción de incluir los datos en el sistema de migrado, para ello, es necesario introducir el *tag* `isBackup`. A continuación se muestra un ejemplo de cómo se guardaría un objeto persistente en la base de datos.

```
{
"_key": "nombreObjeto",
"_content": "valorObjeto"
"isBackup": true
}
```

Asimismo, los callbacks se guardan bajo la clave *_contracts*, seguido de un arreglo con los contratos activos, tal y como se muestra en el siguiente ejemplo:

```
{
"_key": "_contracts",
"_content": ["cotractId1","cotractId2","cotractId3","cotractId4"],
"isBackup": true
}
```

5.2.7 Interfaz de Usuario

La aplicación web para de monitorizar el invernadero se ha realizado en Node-RED, mediante el módulo dashboard [45], que introduce un catálogo de nodos para construir la UI. A partir de la información recibida a través de la API, se genera dinámicamente el estado de cada móudlo en formato visual. Además, de permitir la visualización, también se ofrecen facilidades para modificar manualmente el estado del invernadero.

El funcionamiento de Node-RED es sencillo. se basa en nodos interconectados que responden a señales. Generalmente, cada uno cuenta con una entrada y una salida mediante la cual permiten transformaciones de datos. Cada nodo ejecuta tareas distintas, por ejemplo el nodo "request", realiza peticiones a APIs externas. Según como se redirija su salida, estas peticiones pueden ser aprovechadas por una multitud de nodos. En nuestra configuración, se ha hecho uso de múltiples "function", que mediante código en JavaScript, permiten modificar sus contenidos así como redirigirlos a múltiples salidas del nodo.

Para la implementación se ha hecho uso de dos flujos de Node-RED: `MenuPrincipal` y `PlanN`. En `MenuPrincipal`, se generan los datos a través de un nodo "injection", que envía cada segundo una señal a los nodos conectados y llega al nodo "http" (llamado Server), que solicita la información del módulo de producción seleccionado a la API, mediante la siguiente URL: *http:///plan/{{topic}}*. La URL se genera dinámicamente, según el módulo que se haya seleccionado en la interfaz gráfica. La selección de este módulo se realiza desde un panel desplegable (nodo "dropdown") que incluye los nombres de los módulos configurados en el fichero de configuración. Este nodo produce como salida el número identificador del módulo seleccionado, de modo que al enviarse al nodo "change", se modifica la variable global "topic", usada en la generación dinámica de la URL. Adicionalmente, para su modificación, a través del nodo "link" al que está conectado, se envía un objeto nulo a las gráficas del flujo `PlanN`, para que se eliminen sus contenidos.

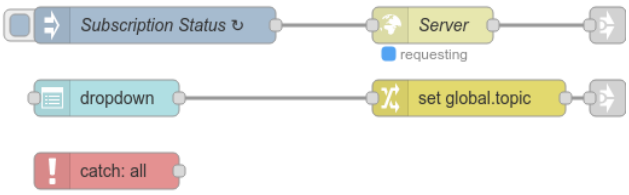

Figura 18: Node-RED: MenuPrincipal

La imagen muestra cómo estan conectados por un lado el nodo “injection” con el nodo que realiza las solicitudes HTTP y redirige su salida al flujo `PlanN`. Por el otro lado, el nodo “dropdown”, conecta con el nodo “change”, que envía una señal nula al flujo `PlanN` a través del nodo “link”.

Desde el flujo `PlanN`, se realiza la generación visual de la interfaz. Tal y como se ha visto, los nodos “link” introducen las respuestas de la API en forma de mensajes JSON con todos los contenidos del interior del módulo de producción seleccionado. Estos mensajes se manipulan dentro de los nodos “function”, que seleccionan la información de interés y la retransmiten por sus salidas respectivas. Tal y como se muestra en la siguiente imágen, el estado de la ventilación y de las válvulas se han representado mediante nodos LED de color verde (si abierto), o gris (si cerrado). Para el sistema de temperatura, los colores que se han usado son rojo (si calefacción), azul (si aire acondicionado) y gris (si cerrado).

Para representar elementos numéricos como la temperatura interna, se ha hecho uso de un nodo “gauge”, mientras que las humedades promedio y la temperatura interna se representan en un gráfico cada una. Para representar la multitud de humedades, se ha hecho uso de un nodo template, que computa un preprocesado a la lista de humedades con CSS.

La interfaz también cuenta con nodos “switch” y “slider” que conectados a un “function” y a un “httprequest”, permiten modificar el estado del invernadero desde la aplicación web.

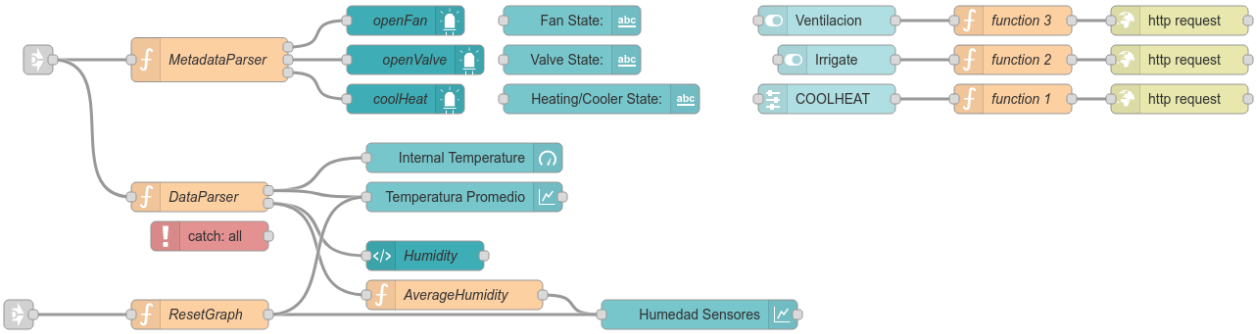

Figura 19: Node-RED: PlanN

La imagen muestra cómo se han conectado los nodos para renderizar los datos y ofrecer un formato más agradable para el usuario final. *MetadataParser* distribuye los contenidos del mensaje entre nodos LED para su representación. *DataParser* distribuye la temperatura en un nodo “gauge” y las humedades en un “template”, y en el “function” *AverageHumidity*, que envía la temperatura promedio a la gráfica. En la parte superior derecha se pueden ver los mecanismos de estado: dos

nodos “switch” para activar/desactivar la ventilación y el riego, así como un “slider”, para modificar el sistema de temperatura.

Los flujos de Node-RED están formados por objetos JSON que identifican a los nodos, sus conexiones y sus atributos configurados. Para la generación del fichero JSON de la interfaz desde los archivos generadores, se preprocesa el fichero *flows.json*, que contiene la plantilla para generar el entorno. Las macros que se han editado son @MACRO_API_URI, donde se añade la `API_URI` configurada, y @MACRO_BORROW_OPTIONS, que añade los nombres establecidos en los módulos de producción, para que se muestren en el módulo “borrow”. El fichero *flows.json* final, puede encontrarse dentro del entorno configurado en el directorio `sensorSimulation/nodeREDClient/`. Mediante este fichero, se puede cargar la UI en la aplicación Node-RED desde el navegador web.

El resultado de esta configuración es una interfaz compuesta de por 5 paneles principales: *Metadata* contiene la informacióndel sistema HVAC y su temperatura interna. *Metrics* muestra en un gráfico cómo evolucionan con el tiempo la humedad promedio de los sensores y la temperatura interna. *Sensors* representa un mapa con todas las humedades capturadas por los sensores, numeradas por su sensorId. Desde *AdminFunctions* se permite modificar el estado del sistema HVAC, y finalmente *SelectedPlan* ofrece un desplegable a partir del cual se pueden consultar las páginas de cada uno de los módulos.

Cada uno de estos paneles se puede ver en la siguiente imagen:

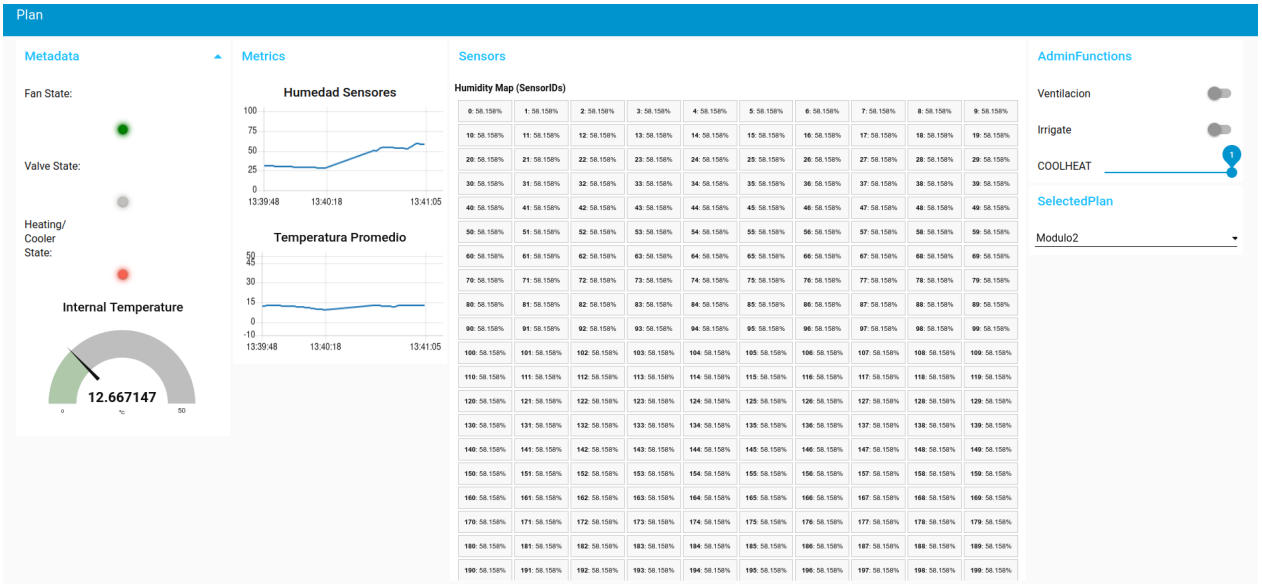

Figura 20: Interfaz gráfica del sistema

En la interfaz gráfica mostrada en la imagen se pueden ver los paneles mencionados (de derecha a izquierda): *Metadata, Metrics, Sensors, AdminFunctions* y *SelectedPlan* (debajo de *AdminFunctions*)

5.2.8 Configuración del entorno

En el proyecto se definen dos ficheros principales que realizan la tarea de desplegar el sistema. Por un lado, el fichero *composesetup.sh*, está dedicado a construir el entorno dentro de la carpeta `ComposeEnvironment`, mientras que *k8ssetup.sh*, lo construye en `K8sEnvironment`. Estos archivos de configuración, usan el fichero de configuración *config.yaml* para una vez copiados los archivos al directorio del entorno, preprocesarlos. En el fichero de configuración, se debe incluir en formato YAML una serie de *tags* vistos a lo largo de este documento. Estos *tags* se listan a continuación:

- ghousename

Incorpora el nombre del entorno del invernadero, no tiene ningún otro efecto práctico.

- `API_URI`

Debe especificar la IP y el puerto de la API. Se usa para preprocesar la macro @MACRO_API_URI en el fichero de la interfaz gráfica para el envío de las peticiones, además de en el fichero de despliegue para el consumidor en Docker (*docker-compose.yaml*) y el de su recurso en Kubernetes (*ConsumerRedundancy.yaml*).

- `KAFKA_URI`

Debe especificar la IP y el puerto de la API. se usa para procesar la macro @MACRO_KAFKA_URI en las variables de entorno del archivo *docker-compose.yaml* y en el *Dockerfile* de la imagen kafka-brokerk8s.

- `NUM_PLANS`

Asigna el número de módulos existentes en el sistema. Es usado para modificar la macro @MACRO_NUM_PLANS en la construcción del simulador, y para establecer la variable de entorno `NUM_PLANS` en docker (*docker-compose.yaml*) y Kubernetes (*ConsumerRedundancy.yaml*).

- plans

Introduce un array con los `NUM_PLANS` módulos de producción existentes. Se definen bajo un nombre (*tag name*), que los identifica dentro de la interfaz gráfica, y un algoritmo (*tag algorithm*), que referencia al nombre de la clase de los algoritmos usados (sin el sufijo .java). Se usan para generar dinámicamente el código de las macros @MACRO_BORROW_OPTIONS, en la interfaz gráfica y en la clase `GreenhouseConsumer`.

A continuación se muestra un ejemplo de archivo de configuración:

```yaml
ghousename: "SampleGHouse"
API_URI: "192.168.1.55:8888"
KAFKA_URI: "kafka-broker:9092"
NUM_PLANS: 4
plans:
  - name: "Mediterraneo"
    algorithm: "PlanArido"
  - name: "Tropical"
```

```
    algorithm: "PlanHumedo"
  - name: "Selva"
    algorithm: "PlanLluvioso"
  - name: "Desierto"
    algorithm: "PlanDesiertico"
```

Dentro de los directorios de cada entorno se llevan a cabo la construcción de las imagenes necesarias, y en el caso de implementar el entorno en Kubernetes, su introducción a la base de datos de Minikube.

Se recomienda guardar los algoritmos del usuario dentro de la carpeta `algorithms`, para que sea visible desde la clase `GreenhouseConsumer` y se pueda compilar.

En esta carpeta, se han incluido una serie de algoritmos de prueba que permiten ejecutar el sistema. El `Plan0` mantiene la temperatura entre 25 y 30 ºC, activa el riego en 50% de humedad y lo desactiva al superar el 99%. El `PlanSierra`, activa el riego cada cuatro horas durante una hora, resultando en la obtención de una gráfica en forma de sierra. También se ofrecen el `PlanNULL`, un algoritmo que que no hace nada, y el `PrintPlan`, que imprime por la salida estándar los contenidos en formato JSON del módulo en tiempo real.

6 Resultados y conclusiones

Al final del proyecto, se ha logrado desarrollar un ecosistema de gestión de invernaderos personalizable que incorpora herramientas para facilitar y automatizar su despliegue.

El programa se ha enfocado en el procesamiento de datos en tiempo real, incorporando mecanismos de tolerancia a errores, escalabilidad y alta disponibilidad, a través de los recursos proporcionados por Kubernetes. El sistema diseñado es capaz de procesar datos actualizados incluso en escenarios con cómputo limitado gracias a la escalabilidad horizontal y el borrado de mensajes antiguos. Mediante el sistema de procesamiento y los mecanismos de construcción del entorno, se ha logrado una plataforma de procesamiento que permite integrar de forma nativa los algoritmos del usuario, sin restricciones de diseño ni tener que recurrir a APIs externas.

La clase abstracta `BasicPlan`, sirve de plantilla para el desarrollo de estos algoritmos, ofreciendo una multitud de funciones imprescindibles y atributos para interactuar con el sistema. Asimismo, también cuenta con una serie de clases que añaden funcionalidad al sistema, desde la migración de los datos hasta el establecimiento de contratos temporales.

Todo esto en un entorno altamente configurable, desde la simulación hasta los algoritmos. Además, el usuario también dispone de la posibilidad de incorporar el sistema a través de dos entornos diferentes: Docker y Kubernetes.

A continuación se muestran casos de uso del sistema en funcionamiento:

Docker

Se ha usado la siguiente configuración:

```yaml
ghousename: "SampleGHouse"
API_URI: "192.168.1.43:8888"
KAFKA_URI: "kafka-broker:9092"
NUM_PLANS: 6
plans:
  - name: "Modulo0"
    algorithm: "Plan0"
  - name: "Modulo1"
    algorithm: "Plan0"
  - name: "Modulo2"
    algorithm: "PlanSierra"
  - name: "Modulo3"
    algorithm: "PrintPlan"
  - name: "Modulo4"
    algorithm: "PlanSierra"
  - name: "Modulo5"
    algorithm: "PlanNULL"
```

La simulación se ha configurado para que cada módulo genere los datos para 300 plantas.

```
const int SUBSCRIPTION_SIZES[] = {300,300,300,300,300,300};
```

Una vez cofigurados los ficheros, se procede a construir el entorno con el archivo *composesetup.sh*, que prepara en el directorio `ComposeEnvironment` todos los archivos necesarios. Una vez preparado, se construye e inicia el entorno con `docker compose up --build`.

Tras iniciar la simulación y el entorno, se puede visualizar el comportamiento del sistema e incluso interactuar con él mediante el uso de las funciones de administrador mostradas en interfaz gráfica. En la siguiente imagen se puede ver cómo el algoritmo modifica el estado del invernadero:

Figura 21: Resultado módulo de producción con algoritmo PlanSierra

Concretamente, se ha aplicado el algoritmo PlanSierra, que consiste en activar el riego durante una hora en cada cuatro, formando con el tiempo una figura similar a dientes de sierra. Al inicio de la simulación se ha aumentado al máximo la temperatura para reducir la humedad más rápidamente. Además, se ha hecho uso del recurso de riego individual para la planta con *id=0*, por lo que se puede ver cómo su porcentaje es mucho mayor al del resto.

También se puede visualizar la salida estándar producida por el algoritmo PrintPlan, que se limita a imprimir los contenidos de los mensajes recibidos:

```
kafka-consumer      | {
kafka-consumer      | planId: 3
kafka-consumer      | datetime: 2000-01-04T18:45
kafka-consumer      | openFan: 0
kafka-consumer      | openValve: 0
kafka-consumer      | coolHeat: 0
kafka-consumer      | temp: 11.0
kafka-consumer      | humidity: [[37.568347, 37.568448...
kafka-consumer      | }
```

Así como otros logs informativos producidos por el propio sistema, que no solo se limita a anunciar las tareas que van finalizando ni las modificaciones del invernadero, sino que también incorpora el nivel de error:

```
kafka-consumer      | [pool-1-thread-8] INFO greenhouseutils.GreenhouseAction -
Valve set to 1
kafka-consumer      | [pool-1-thread-8] INFO multithreading.Task -
Task multithreading.Task@6580cb0d ended succesfully
```

```
kafka-consumer      | [pool-1-thread-6] INFO multithreading.Task -
Task multithreading.Task@181f7621 ended succesfully
```

Kubernetes

Este escenario se enfoca en demostrar la redistribución de las particiones, la escalabilidad del sistema, y cómo se realiza la migración de datos. Se ha mantenido la configuración del caso anterior, con la diferencia de que el puerto del servidor kafka se ha establecido a 30.000. Una vez construido el entorno ejecutando *k8ssetup.sh*, se procede a iniciar los recursos de Kubernetes mediante `kubectl apply -f DataStorage.yaml` y `kubectl apply -f ConsumerRedundancy.yaml`. A continuación, se ven todos los recursos activos:

```
NAME                                  READY   STATUS      RESTARTS   AGE
broker-initializer                    0/1     Completed   0          36s
consumer-deployment-67766b78bc-h2mnq  1/1     Running     0          17s
consumer-deployment-67766b78bc-j5zpf  1/1     Running     0          2s
kafka-deployment-d549d9456-5m2x2      1/1     Running     0          36s
mongodb-deployment-5d49888b4f-n9vml   1/1     Running     0          36s
```

Se puede ver como se instancian dos servidores consumidores. A continuación se muestra la distribución de las particiones entre estos dos consumidores:

Consumidor 1

```
Assigned partitions:                       [riego-3, riego-4, riego-5]
Current owned partitions:                  []
Added partitions (assigned - owned):       [riego-3, riego-4, riego-5]
Revoked partitions (owned - assigned):     []
```

Consumidor 2

```
Assigned partitions:                       [riego-0, riego-1, riego-2]
Current owned partitions:                  [riego-0, riego-1, riego-2]
Added partitions (assigned - owned):       []
Revoked partitions (owned - assigned):     []
```

Nótese como cada una gestiona particiones de manera única. Para poner a prueba el escalado de las instancias, se ha rebajado el consumo promedio de las instancias de la CPU al 5% (por defecto, 30%). Es importante tener activado el *addon metrics-server* para Minikube, de lo contrario, el sistema no será capaz de escalar. A continuación se procede a ejecutar la simulación. Para ello, se debe configurar la URI del servicio kafka como variable de entorno. Para ver la dirección IP del servicio de Kafka, se puede usar `minikube service kafka-service --url`.

El servicio ha escalado hasta 6 instancias (máximo permitido en la configuración) y las particiones se han asignado de la siguiente manera:

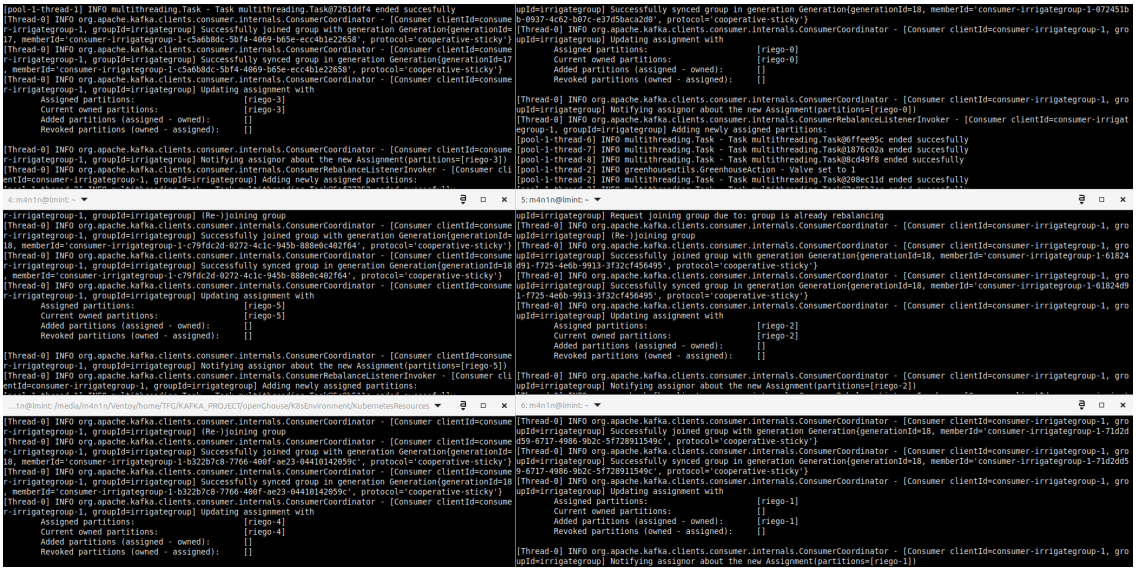

Figura 22: Distribución de las particiones entre cada consumidor

La imagen muestra un escenario con 6 consumidores que ha dado lugar al escenario idílico en que cada instancia se encarga de una sola partición.

A continuación, se puede ver cómo se ha guardado un contrato en la base de datos dentro del algoritmo del módulo 3. Gracias a ello, estos contratos pueden ser recuperados desde otras instancias en caso de error.

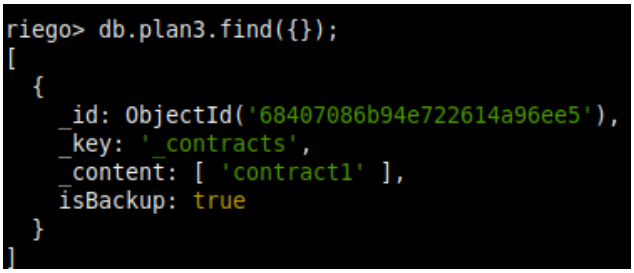

Figura 23: Ejemplo de Objeto guardado en la base de datos

En este caso, se trata de la lista de callbacks para el módulo 3. Las instancias pueden migrar sus contratos a través del objeto con clave *_contracts* en la base de datos.

Para comprobar la migración de datos, ha ejecutado un escenario con dos instancias. Se procede a cerrar una de ellas:

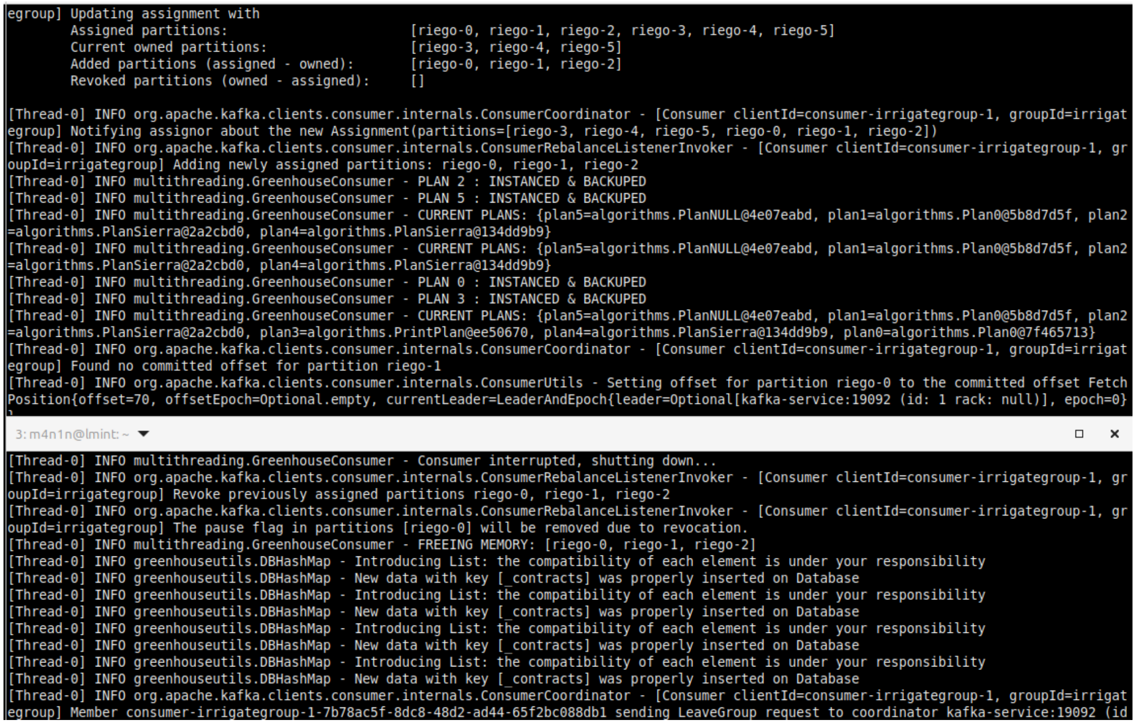

Figura 24: Caso de migración de datos entre consumidores

Como se puede ver en la imagen, el servidor inferior ha sido cerrado forzosamente, de modo que antes de liberar sus recursos, se guardan los objetos existentes en la base de datos. Estos datos son recuperados por el servidor consumidor, que crea y actualiza sus instancias con los datos existentes en la base de datos (imprimiendo mensajes informativos de que se ha creado el objeto y establecido el backup).

Durante el transcurso del proyecto, se ha logrado realizar todos los requisitos listados, aunque también se han producido algunos ligeros cambios que han cambiado la forma en que se ha implementado el sistema, principalmente en el ámbito de las tecnologías usadas. Inicialmente se pretendía hacer uso de plataformas de procesamiento de datos como Kafka Streams, no obstante, se abandonó la idea debido a la pérdida de control del consumidor al usar Streams. El objetivo principal del trabajo era usar Apache Kafka, si se hubiera usado Streams, la gestión de registros y desplazamientoss se manejaría automáticamente, resultando en un programa mucho más sencillo de implementar. Además, mediante Kafka, se ha podido conocer a mayor profundidad su funcionamiento, mediante un mayor control manual sobre los registros y el comportamiento del servidor.

Otro gran cambio, ha sido la tecnología usada para crear la interfaz gráfica. En un principio se tenía previsto hacer uso de librerías como Java Swing o JavaFX, pero dada su curva de aprendizaje y el tiempo limitado para realizar el proyecto, se acabó creando la aplicación web con Node-RED, lo que permitió ahorrar una gran cantidad de tiempo. En consecuencia, se pudo realizar la interfaz gráfica en mucho menos tiempo de las dos semanas previstas inicialmente, tiempo que se aprovechó para extender al alcance del proyecto añadiendo escalabilidad y migración de datos, gracias a la implementación de un clúster de Kubernetes. La última de las tecnologías que se decidió añadir, es una base de datos MongoDB. Para mantener la persistencia, era necesario una base de datos no relacional para poder almacenar objetos de Java. Se decidió usar MongoDB dada su facilidad y su efectividad para gestionar lecturas y escrituras concurrentes.
Si bien es cierto que todos los requisitos del proyecto han podido ser realizados, algunos otros se han visto modificados por la falta de tiempo. Se ha renunciado a inclusión de un sistema de usuarios con diferentes permisos en la aplicación web del invernadero. Por último, en la simulación también se buscaba incorporar sensores de iluminación, así como mecanismos que para controlarla, no obstante, no se ha llegado a implementar.

La mayor dificultad ha sido el diseño de la simulación, que ha consumido mucho más tiempo de lo que se esperaba. Lograr un escenario de generación de tantos datos de un modo tan eficiente, ha sido complicado. En las primeras implementaciones, la simulación se retardaba respecto a la realidad. Mediante el uso de dos cerrojos, se coordinaba la GPU y la CPU. No obstante, esto llevaba en muchos casos al bloqueo de la simulación, y siempre acababa en el retardo del tiempo respecto a la realidad. Más adelante, se introdujo un sistema de temporización basado cronómetros, que junto a la reducción de los cerrojos y a la secuencialización de la simulación y la captura de los sensores, se logró un sistema increíblemente eficiente.

Otra dificultad fue comprender todo el funcionamiento de Apache Kafka. Kafka es una tecnología muy completa que ofrece una gran cantidad de facilidades. Comprender su estuctura interna fue complicado en un principio. Es imprescindible entender cómo funciona todo el sistema a bajo nivel: el consumidor, los grupos de consumidores, las particiones o los mensajes son unos de los tantos componentes que se deben aprender. Es una arquitectura tan simplificada como compleja que no es posible usarla sin antes entender todo lo que sucede por debajo.

Una limitacion a destacar es que el sistema está ideado para implementarse desde una máquina Linux. Los archivos *composesetup.sh* y *k8ssetup.sh* usan binarios de Linux para construir el entorno, por lo que probablemente no funcionará en máquinas con un sistema operativo diferente.

Como posibles mejoras se propone establecer un cifrado para todas las conexiones, implementar un

sistema de usuarios con diferentes permisos en la aplicación de monitorización, así como añadir las funcionalidades de iluminación en la simulación. También se propone implementar los recursos de Kubernetes en un proveedor de servicios en la nube.

7 Análisis de impacto

Gracias a la implementación de este sistema en invernaderos, los agricultores pueden usar un ecosistema de código abierto y de bajo coste mediante el cual gestionar sus cultivos. También ofrece la posibilidad de construir sus algoritmos de forma completamente personalizada, sin restricciones de diseño.

Se facilita la labor de su configuración a través de un fichero de configuración que permite construir el escenario listo para ser ejecutado. Es por ello, que para su generación, no se requiere de especialización técnica más que para introducir los algoritmos y sus dependencias.

Adicionalmente, también se podrá hacer un seguimiento de todos los cultivos a precisión de planta, siendo de gran utilidad para la prevención de enfermedades en el cultivo.

Bibliografía

[1] Autores de Kubernetes, «Orquestación de contenedores para producción». https://kubernetes.io/es, 2025.

[2] Apache Software Foundation, «Apache Kafka». https://kafka.apache.org, 2024.

[3] MongoDB Inc, «MongoDB: The World's Leading Modern Database | MongoDB». https://www.mongodb.com, 2025.

[4] Docker, «Docker: Accelerated Container Application Development». https://www.docker.com, 2025.

[5] Fernando Delgado, «El impacto de la Revolución Industrial en la sociedad». https://historia.ovh/el-impacto-de-la-revolucion-industrial-en-la-sociedad, 2025.

[6] Tapsa, «Beneficios de los Invernaderos Automatizados en la Agricultura». https://tapsadecv.com/beneficios-de-los-invernaderos-automatizados-en-la-agricultura, 2024.

[7] cienciaintrigante.com, «El impacto de la tecnología en la agricultura: avances y beneficios». https://cienciaintrigante.com/tecnologia/el-impacto-de-la-tecnologia-en-la-agricultura-avances-y-beneficios, 2025.

[8] John R. Pares, «Historia del riego exhudante». https://www.scribd.com/document/346388712/Historia-Del-Riego-Exudante, n.d.

[9] Rafael Fernández Gómez et al, «Manual de riego para agricultores. Riego localizado.» https://www.juntadeandalucia.es/organismos/agriculturapescaaguaydesarrollorural/servicios/publicaciones/detalle/43761.html, 2001.

[10] Eugenio Cedillo Portugal, «MANUAL DE CONTROL DE CLIMA Y RIEGO EN UN INVERNADERO». https://planificacionfesaragon.com/sites/default/files/manuales/Manual%20de%20Control%20de%20Clima.pdf, 2019.

[11] Acelera pyme, «IoT en la agricultura: Cómo mejorar la eficiencia en la producción agrícola con la tecnología IoT». https://www.acelerapyme.es/sites/acelerapyme/files/2023-11/IoT%20en%20la%20agricultura_c%C3%B3mo%20mejorar%20la%20eficiencia%20en%20la%20producci%C3%B3n%20agr%C3%ADcola%20con%20la%20tecnolog%C3%ADa%20IoT.pdf, 2023.

[12] Hortalan Med, «Greenhouses for Crops: Optimizing Agricultural Production». https://hortalan.com/ultimas-noticias/invernaderos-para-cultivos-optimizando-la-produccion-agricola, 2024.

[13] Priva, «Priva Connext: the most advanced greenhouse climate computer». https://www.priva.com/horticulture/solutions/climate-and-process-computers/priva-connext, 2022.

[14] Dusun IoT, «IoT Based Greenhouse Monitoring and Control System for Smart Agriculture». https://www.dusuniot.com/case-study/iot-greenhouse-monitoring-and-control-system-for-smart-agriculture, 2023.

[15] M. Woolley,O. Kulkarni and P. Winiarczyk, «Bluetooth® Mesh Remote Provisioning». https://www.bluetooth.com/mesh-remote-provisioning, 2023.

[16] Docker, «Swarm mode». https://docs.docker.com/engine/swarm, 2025.

[17] Amazon Web Services, «Cloud Computing - Servicios de informática en la nube». https://aws.amazon.com/es, 2024.

[18] Broadcom, «RabbitMQ. One broker to queue them all». https://www.rabbitmq.com, 2025.

[19] Fudación OpenJS y otros contribuyentes de Node-RED, «Node-RED: Low-code programming for event-driven applications». https://nodered.org, 2025.

[20] Banco Mundial, «Agricultura y alimentos». https://www.bancomundial.org/es/topic/agriculture/overview, 2024.

[21] FAO, «Peligros del cambio climático». https://www.fao.org/newsroom/detail/water-scarcity-means-less-water-for-agriculture-production-which-in-turn-means-less-food-available-threatening-food-security-and-nutrition/es, 2025.

[22] Axel Kleidon, «Droughts in Germany». https://arxiv.org/pdf/2403.16551, 2024.

[23] Sihua Lu, «The reality of climate change: evidence, impacts and engineering solutions». https://arxiv.org/pdf/2410.12412, 2024.

[24] Sergio Zabala, «Riesgo climático el mediterraneo». https://cadenaser.com/baleares/2024/12/21/los-payeses-de-mallorca-agradecen-la-llegada-del-invierno-corremos-el-riesgo-de-que-baleares-llegue-a-ser-un-desierto-radio-mallorca, 2024.

[25] Md. Galib Ishraq Emran et al, «Reasons behind the Water Crisis and its Potential Health Outcomes». https://arxiv.org/pdf/2403.07019, 2021.

[26] D. Bonazzi, «Riesgo inminente de una crisis mundial del agua». https://www.unesco.org/es/articles/riesgo-inminente-de-una-crisis-mundial-del-agua-unesco/onu-agua, 2023.

[27] Lifeng Li, «Escasez de agua, crisis climática y seguridad alimentaria mundial: un llamamiento a la acción colaborativa». https://www.un.org/es/cr%C3%B3nica-onu/escasez-de-agua-crisis-clim%C3%A1tica-y-seguridad-alimentaria-mundial-un-llamamiento-la, 2023.

[28] Water Science School, «The distribution of water on, in, and above the Earth». https://www.usgs.gov/media/images/distribution-water-and-above-earth, 2023.

[29] Christina Nunez, «La contaminación del agua constituye una crisis mundial creciente». https://www.nationalgeographic.es/medio-ambiente/contaminacion-del-agua, 2025.

[30] Jose Antonio García Mompeán, «El uso industrial del agua». https://gargil.es/el-uso-industrial-del-agua, 2024.

[31] Fernando Chávez Virreira, «Millones de litros de agua, el costo ecológico de la IA y Ghibli». https://www.vision360.bo/noticias/2025/04/14/23267-millones-de-litros-de-agua-el-costo-ecologico-de-la-ia-y-ghibli, 2025.

[32] Sebastián Salom, «openGhouse». https://github.com/Tian3030/openGhouse.git, 2025.

[33] Oracle, «Java». https://www.java.com/es, 2025.

[34] Nick Barney, «What is C++?» https://www.techtarget.com/searchdatamanagement/definition/C, 2023.

[35] NVIDIA, «CUDA C++ Programming Guide». https://docs.nvidia.com/cuda/cuda-c-programming-guide, 2025.

[36] Morgan Stanley, «modern-cpp-kafka». https://github.com/morganstanley/modern-cpp-kafka, 2024.

[37] OpenMP, «OpenMP: Home». https://www.openmp.org, 2024.

[38] Lincoln Zotarelli, Michael D. Dukes, y Kelly T. Morgan, «Interpretación del Contenido de la Humedad del Suelo para Determinar Capacidad de Campo y Evitar Riego Excesivo en Suelos Arenosos Utilizando Sensores de Humedad». https://edis.ifas.ufl.edu/publication/AE496, 2019.

[39] David Jacot, «Apache Kafka 4.0.0 Release Announcement». https://kafka.apache.org/blog#apache_kafka_400_release_announcement, 2025.

[40] Confluent, Inc, «confluentinc/confluent-local». https://hub.docker.com/layers/confluentinc/confluent-local/7.9.0/images/sha256-c499275a83c0bb21918e5fc4f463ba15c64ea61a2d427fb6abff89c749a6d516, 2025.

[41] IBM, «¿Qué es Java?» https://www.ibm.com/es-es/topics/java, 2024.

[42] Confluent, Inc, «Java Client for Apache Kafka». https://docs.confluent.io/kafka-clients/java/current/overview.html, 2025.

[43] Apache Software Foundation, «Welcome to Apache Maven». https://maven.apache.org, 2025.

[44] mongodb, «mongodb/mongodb-community-server:latest». https://hub.docker.com/layers/mongodb/mongodb-community-server/latest/images/sha256-616e86fb598ef505f83fac453ecc6b816690e0b30d078a0166a86b3d2617714b%0A, 2025.

[45] Fundación OpenJS y otros contribuyentes de Node-RED, «node-red-dashboard». https://flows.nodered.org/node/node-red-dashboard, 2024.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jun 16 11:56:13 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)