E.T.S. INGENIERÍA INFORMÁTICA
GRADO EN GRADO EN INGENIERÍA INFORMÁTICA

Arquitectura flexible para la creación de suscriptores RabbitMQ
Flexible architecture for the creation of RabbitMQ subscriptors

Realizado por
Rafael López Gómez
Tutorizado por
María del Mar Gallardo Melgarejo
Cotutorizado por
Laura Panizo Jaime
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, 25 de junio de 2022

Fecha defensa:
El Secretario del Tribunal

Agradecimientos

En primer lugar quiero agradecer a mis padres, no hubiera podido llegar hasta aquí sin su comprensión y apoyo incondicional.

En segundo lugar quiero agradecer a mis tutoras María del Mar y Laura por brindarme la oportunidad de realizar este proyecto, por guiarme y por darme un gran apoyo en el proceso de realización del trabajo.

Por último, quiero agradecer a todos mis compañeros, amigos y a mi familia, por apoyarme en momentos difíciles y alentarme a seguir.

Muchas gracias a todos.

Resumen: El objetivo de este proyecto es realizar un acercamiento al uso de las nuevas tecnologías en el mundo del Internet de las Cosas ("Internet of Things", IoT). En muchos casos, las aplicaciones IoT comparten características comunes como, por ejemplo, el uso de una plataforma que recoge y gestiona el tráfico de datos entre los extremos. Por otro lado, el procesamiento de los datos es, en general, muy dependiente del caso de uso (sensores que envían datos de temperatura, drones que mandan imágenes, etc.).

En este proyecto, se propone el diseño e implementación de una herramienta que facilite el desarrollo de aplicaciones IoT que se encargan de recibir y procesar datos. A esta herramienta la llamaremos *Fabricante*. Por otro lado, las aplicaciones IoT, producidas por la herramienta *Fabricante*, hacen uso de una entidad intermedia, como un servidor de mensajería (en nuestro caso RabbitMQ), que se encarga de la distribución y gestión del tráfico de mensajes, a la que se conectan las aplicaciones IoT para procesar y recibir los datos. A este tipo de aplicaciones las llamaremos *Consumidores*. En el proyecto, se profundiza en la automatización de la lógica que hay detrás de estas conexiones.

La herramienta *Fabricante* es capaz de construir de forma automática aplicaciones de procesamiento de datos que leen desde una fuente remota utilizando la configuración de entrada proporcionados por el usuario (cómo son los datos a procesar, y de qué forma hay que procesarlos). De esta forma, el desarrollador solo tiene que concentrarse en la creación del algoritmo que trata los datos. No tiene que preocuparse en programar si el algoritmo se repite o bajo que condiciones se ejecuta. En definitiva, el uso de la herramienta *Fabricante* dará como resultado aplicaciones *Consumidor* que se pueden conectar a un servidor de gestión de datos y procesar los mensajes recibidos, cuyo comportamiento es configurable.

Palabras claves: Internet de las Cosas, RabbitMQ, Arquitectura flexible, AMQP, MQTT

Abstract:

The objective of this project is to make an approach to the use of new technologies in the context of the Internet of Things ("Internet of Things", IoT). In many cases, IoT applications share common features, such as the use of a platform that collects and manages data traffic between endpoints. On the other hand, data processing is, in general, very dependent on the use case (sensors that send temperature data, drones that send images, etc.).

In this project, we propose the design and implementation of a tool that facilitates the development of IoT applications that are responsible for receiving and processing data. This tool is called the *Factory* tool. Additionally, the IoT applications, generated by the *Factory* tool, make use of an intermediate entity, like a messaging server (for example RabbitMQ), which is responsible for the distribution and management of message traffic, to which the IoT applications connect to process and receive the data. These applications are called the *Consumer* apps. In the project, we will delve in to the automation of the logic behind these connections.

The *Factory* tool can automatically build a data processing application that read from a remote source using the input configuration provided by the user (what the data to be processed is like, and how it is to be processed). In this way, the developer only has to concentrate on creating the algorithm that processes the data. The developer does not have to worry about programming whether the algorithm repeats itself or under what conditions it runs. In short, the use of the *Fatory* tool will result in *Consumer* apps that can connect to a data management server and process the received messages, whose behavior is configurable.

Keywords: Internet Of Things, rabbitMQ, Flexible Architecture, AMQP, MQTT

Índice de contenidos

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos 3
1.3. Metodología y Fases de Trabajo 4
1.4. Estructuración de la memoria 5

2. Tecnologías **7**
2.1. Lenguajes y entornos de desarrollo 7
2.1.1. Java 7
Maven 8
2.1.2. Formatos y especificaciones 8
JSON 8
JSON Schema 9
GSON 9
Jsonschema2pojo 10
Apache Commons CLI 10
2.1.3. Eclipse: Entorno de desarrollo para Java 10
2.2. Bróker de mensajería 10
2.2.1. Protocolos de comunicación 11
MQTT 14
AMQP 14
Interoperabilidad 15
2.2.2. RabbitMQ: Clientes en Java 16
Eclipse Paho 16
AMQP client 17

3. Diseño **19**
3.1. Descripción general 19
3.2. Análisis de requisitos 20
3.2.1. Requisitos del Consumidor 20
3.2.2. Requisitos del Fabricante 21
3.3. Diseño del Consumidor 22
3.3.1. Diseño de la estructura de consumidor 22
3.3.2. Diseño de las funcionalidades del consumidor 23
3.4. Diseño del Fabricante 24
3.4.1. Diseño de la estructura del fabricante 25
3.4.2. Diseño de las funcionalidades del fabricante 26

4. Desarrollo **29**
4.1. Implementación del *Fabricante* . . . 29
4.1.1. Especificación de la estructura del *Fabricante* . . . 29
4.1.2. Descripción de la funcionalidad del *Fabricante* . . . 31
4.2. Implementación del Consumidor . . . 36
4.2.1. Estructura del consumidor generado . . . 36
4.2.2. Descripción de la funcionalidad del consumidor . . . 38
Módulo de interacción con el usuario . . . 38
Módulo de gestión de las conexiones . . . 38
Módulo de procesamiento de datos . . . 40
Restricciones sobre las opciones de configuración . . . 42

5. Pruebas y casos de estudio **47**
5.1. Pruebas Generales . . . 47
5.2. Casos de estudio . . . 49
5.2.1. Caso de estudio 1: Dataset Iris . . . 49
5.2.2. Caso de estudio 2: Filogenética computacional . . . 50
5.2.3. Caso de estudio 3: Criptografía . . . 52

6. Conclusiones y líneas futuras **55**
6.1. Satisfacción de requisitos . . . 55
6.2. Trabajos Futuros . . . 56

Bibliografía **59**

Apéndice A. Manual del fabricante **63**
A.1. Instalación . . . 63
A.2. Funcionamiento . . . 63
A.2.1. Paso 1: Elegir el nombre del consumidor . . . 63
A.2.2. Paso 2: Seleccionar el esquema JSON . . . 64
A.2.3. Paso 3: Guardar la plantilla del algoritmo . . . 65
A.2.4. Paso 4: Subir la clase del algoritmo completada . . . 65
A.2.5. Paso 5: Elegir o no JARS externos . . . 65
A.2.6. Paso 6: Elegir el protocolo de comunicación . . . 66
A.2.7. Paso 7: Seleccionar donde guardar el consumidor . . . 66

Apéndice B. Manual del consumidor **69**
B.1. Requisitos previos . . . 69
B.2. Funcionamiento . . . 69
B.2.1. Recepción de mensajes . . . 71

Índice de figuras

1.1. Introducción al concepto del Fabricante . . . 3
1.2. Introducción al concepto del Consumidor . . . 3
1.3. Esquema de la Metodología Incremental . . . 4

2.1. Esquema de la portabilidad de Java . . . 8
2.2. Ejemplo del formato JSON . . . 9
2.3. Esquema de la arquitectura de los brókers de mensajería . . . 11
2.4. Esquema del patrón Publish/Subscibe . . . 12
2.5. Esquema del patrón RPC . . . 12
2.6. Fundamentos de AMQP . . . 15

3.1. Visión general del proyecto . . . 20
3.2. Diagrama de componentes del consumidor . . . 23
3.3. Esquema del patrón Modelo-Vista-Controlador . . . 25
3.4. Diagrama de actividades del fabricante . . . 26

4.1. Estructura de la interfaz gráfica del *Fabricante* . . . 30
4.2. Funcionamiento de la interfaz gráfica . . . 30
4.3. Diagrama de clases del *Fabricante* . . . 31
4.4. Diagrama de actividades del *Fabricante* . . . 32
4.5. Ejemplo de Esquema JSON . . . 33
4.6. Ejemplo de clase -*Schema* . . . 34
4.7. Ejemplo de Plantilla del algoritmo . . . 34
4.8. Ejemplo de clase Algoritmo completado . . . 35
4.9. Estructura del consumidor . . . 37
4.10. Diagrama de clases del Consumidor . . . 38
4.11. Ejemplo de la estructura de los mensajes recibidos por el consumidor . . . 40
4.12. Implementación del tiempo de espera entre mensajes . . . 41
4.13. Restricciones si se usan todos los mensajes . . . 44
4.14. Restricciones si se desechan los mensajes procesados . . . 44
4.15. Diagrama de secuencia del consumidor . . . 45

5.1. Esquema JSON para el dataset Iris . . . 50
5.2. Clase del algoritmo para el caso de uso con el dataset Iris . . . 51
5.3. Ejemplo de resultado del algoritmo para el dataset Iris . . . 51
5.4. Esquema JSON para la definición de los datos en el caso de estudio del algoritmo de Sankoff . . . 52
5.5. Ejemplo de resultado del algoritmo de Sankoff . . . 52
5.6. Esquema JSON del algoritmo verificador de contraseñas . . . 53

5.7. Ejemplo de resultado del algoritmo verificador de contraseñas 54

A.1. Mensajes de error paso 1 del *Fabricante* 64
A.2. Paso 2 del *Fabricante* 64
A.3. Ejemplo de Esquema JSON 64
A.4. Ejemplo de clase -*Schema* 66
A.5. Paso 3 del *Fabricante* 67
A.6. Ejemplo de Plantilla del algoritmo 67
A.7. Paso 5 del *Fabricante* 67
A.8. Paso 5* del *Fabricante* 68
A.9. Paso 6 del *Fabricante* 68
A.10. Paso 7 del *Fabricante* 68

B.1. Ayuda del consumidor (Versión AMQP) 70
B.2. Restricciones entre opciones de configuración 72
B.3. Ejemplo de la estructura de los mensajes recibidos por el consumidor . . . 73

Índice de tablas

2.1. Tabla comparativa entre protocolos IoT 13

3.1. Requisitos funcionales del consumidor 21
3.2. Requisitos no funcionales del consumidor 21
3.3. Requisitos funcionales del fabricante 22
3.4. Requisitos no funcionales del fabricante 22

4.1. Clasificación de las opciones de configuración 43

6.1. Requisitos funcionales del *Fabricante* logrados 56
6.2. Requisitos no funcionales del *Fabricante* logrados 56
6.3. Grado de satisfacción de los requisitos funcionales del consumidor 57
6.4. Grado de satisfacción de los requisitos no funcionales del consumidor . . . 57

CAPÍTULO 1

Introducción

1.1. Motivación

El Internet de las Cosas (“Internet Of Things” - IoT) puede definirse como la agrupación e interconexión de dispositivos de cualquier índole a través de una red. Normalmente, la comunicación en los entornos IoT se suele externalizar mediante el uso de un servicio de notificaciones centralizado. Este servicio tiene que proporcionar fiabilidad y escalabilidad. Los dispositivos no saben si hay otros dispositivos conectados, y tampoco son conscientes del destino u origen de los datos, ya que esta tarea de distribución suele quedar en manos del ya mencionado servicio. Esta estructura busca un bajo acoplamiento y una mayor escalabilidad entre los dispositivos porque al no depender unos de otros, pueden conectarse y desconectarse del sistema sin que esto suponga un problema para la arquitectura general.

El propósito de partida de este proyecto ha sido el estudio de nuevas tecnologías usadas en el mundo de IoT. Tras un análisis inicial quedó patente la heterogeneidad de las aplicaciones que pueden desarrollarse y las posibilidades que ofrecía en diferentes entornos:

- **Domótica:** incluye todo tipo de aplicaciones y dispositivos que permiten automatizar una vivienda. Por ejemplo, dispositivos conectados a internet que a través de comandos de voz pueden acceder a repositorios remotos para reproducir canciones o vídeos; sistemas de alarma que se conectan con centrales de seguridad; sensores conectados al sistema de calefacción de la casa que monitorizan constantemente información acerca de la temperatura ambiente para que regularla, etc.

- **Sector ganadero:** por ejemplo, la monitorización biométrica y la geolocalización que mantiene a los animales siempre controlados.

- **Sector agrícola:** por ejemplo, el uso de drones y sensores que envían datos sobre las condiciones de los cultivos y del terreno para que sean analizados con el fin de hacer predicciones, estudios o ejecutar sistemas de riegos.

- **“Smart Cities” y “Smart Buildings”:** por ejemplo, el uso de dispositivos para gestionar el tráfico, el control del suministro del agua, o la afluencia del transporte público.

Éstas son sólo algunas de las muchas opciones que entran dentro del ámbito de IoT.

Pese a la gran disparidad de utilidades, las aplicaciones IoT comparten ciertas similitudes como, por ejemplo, el uso de una plataforma que hace de intermediario en la distribución de datos entre extremos. Por ello, se planteó como objetivo explotar estos puntos en común para diseñar e implementar una herramienta que facilitara el desarrollo de aplicaciones IoT que se encargan de recibir y procesar datos.

La herramienta ha sido nombrada como *Fabricante*, y su función consiste en generar aplicaciones IoT de comportamiento configurable y que llamamos *consumidores*. Los consumidores tienen la capacidad de conectarse a un servidor de mensajería RabbitMQ [1], que se encarga de la distribución y gestión del tráfico de mensajes, para recibir y procesar los datos.

La Figura 1.1 muestra una idea general la funcionalidad del *Fabricante*. Un usuario interacciona con la herramienta *Fabricante*, la cual le pedirá que introduzca diversas especificaciones del consumidor que tiene que generar. Entre los datos que el usuario ha de proporcionar tenemos el nombre del consumidor, el protocolo de comunicación que desea que el consumidor utilice en las conexiones con el servidor de mensajería RabbitMQ; el algoritmo que ha de ejecutar el consumidor, y una descripción de los datos que el consumidor espera recibir.

El uso concreto de RabbitMQ se debe a que dentro del catálogo de brókers (también denominado *Router* o *servidor de mensajería*) es la opción más extendida y la que mayor soporte ofrece. Además, es compatible con dos de los protocolos más usados en este ámbito: AMQP y MQTT.

Un consumidor es una aplicación receptora de datos procedentes de fuentes específicas y para procesarlos siguiendo un algoritmo predefinido, dependiente de la aplicación. Los consumidores presentan un comportamiento configurable por el usuario en cada ejecución. La Figura 1.2 ofrece una visión general de un consumidor. En primer lugar, un usuario puede configurar aspectos como: la dirección IP del servidor RabbitMQ, el puerto de escucha del servidor, diversas opciones de autenticación, la condición de parada de ejecución del consumidor, la condición de ejecución del algoritmo, filtrar los mensajes que recibe el consumidor y que van a ser procesados en el algoritmo, etc. Una vez que el usuario introduce como argumentos las opciones de configuración que considere oportunas, el consumidor establecerá una conexión con el servidor RabbitMQ y procederá a esperar la recepción de mensajes. Por otro lado, hay una entidad llamada Publisher, de cuya existencia el consumidor desconoce, que es la encargada de enviar esos datos (fuente anónima de datos). El servidor RabbitMQ es el mediador en la distribución de los mensajes y es el único que es consciente de la identidad de los extremos. Por último, el consumidor ejecutará el algoritmo de procesado que le fue integrado en el momento de su creación en el Fabricante cuando se cumplan las condiciones introducidas por el usuario por argumento. Además, el consumidor podrá terminar su ejecución siguiendo las configuraciones que también ha introducido el usuario.

Una vez finalizado el proceso de diseño e implementación, se ha hecho una aproximación del uso de la herramienta en diversos casos de estudio reales, entre ellos dos relacionados con el campo de la filogenética computacional y la criptografía. La disparidad en la elección de ámbitos de uso demuestra la utilidad de la herramienta en un amplio espectro de entornos realistas.

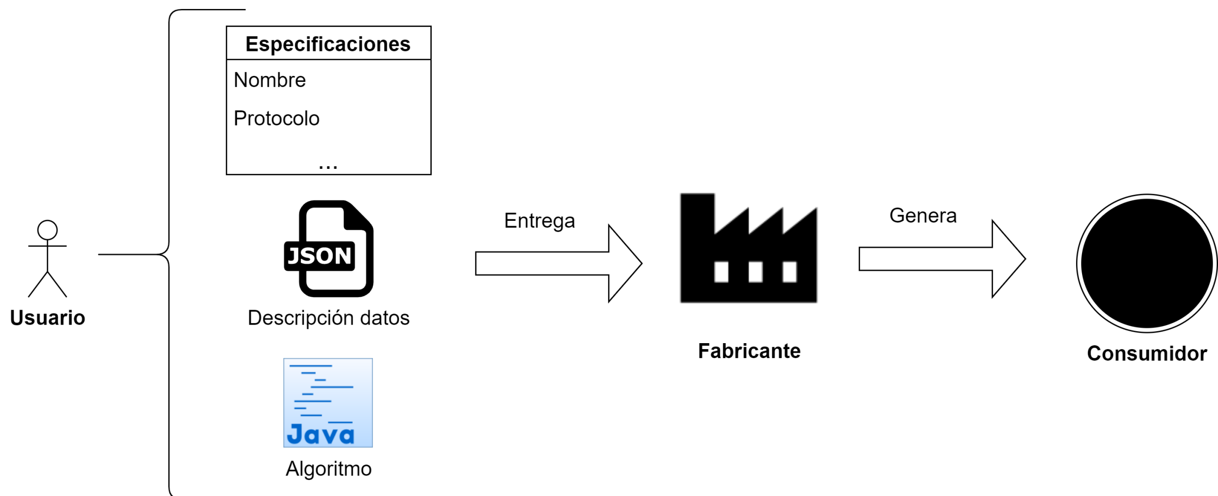

Figura 1.1: Introducción al concepto del Fabricante

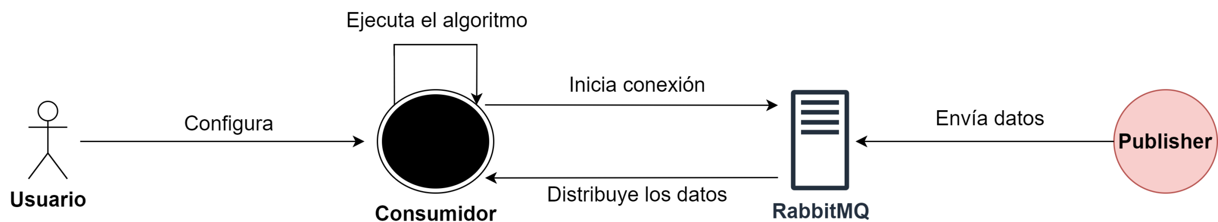

Figura 1.2: Introducción al concepto del Consumidor

1.2. Objetivos

El objetivo del proyecto es el diseño e implementación de una herramienta (*Fabricante*) capaz de generar nodos de procesamiento de datos (consumidores) que utilizan RabbitMQ como sistema de interconexión y paso de mensajes. Dichos nodos generados tendrán un comportamiento configurable en cada ejecución, por lo que podrán adaptarse a las necesidades del usuario en cada momento.

En particular, durante la realización del proyecto se han abordado principalmente las siguientes tareas:

1. Estudio y selección de las tecnologías y protocolos disponibles en el mundo de IoT.
2. Diseño e implementación de una arquitectura genérica para aplicaciones configurables que procesen datos, llamadas como consumidores.
3. Implementación del *Fabricante*, herramienta que pueda generar los consumidores previamente definidos.
4. Selección e integración de diversos casos de estudio. Para ello será necesario adaptar aplicaciones ya existentes con la funcionalidad de la plataforma RabbitMQ.

Los resultados del proyecto son los siguientes:

1. Arquitectura de las aplicaciones que constituyen los nodos IoT que procesan datos y tienen un comportamiento configurable (consumidores).

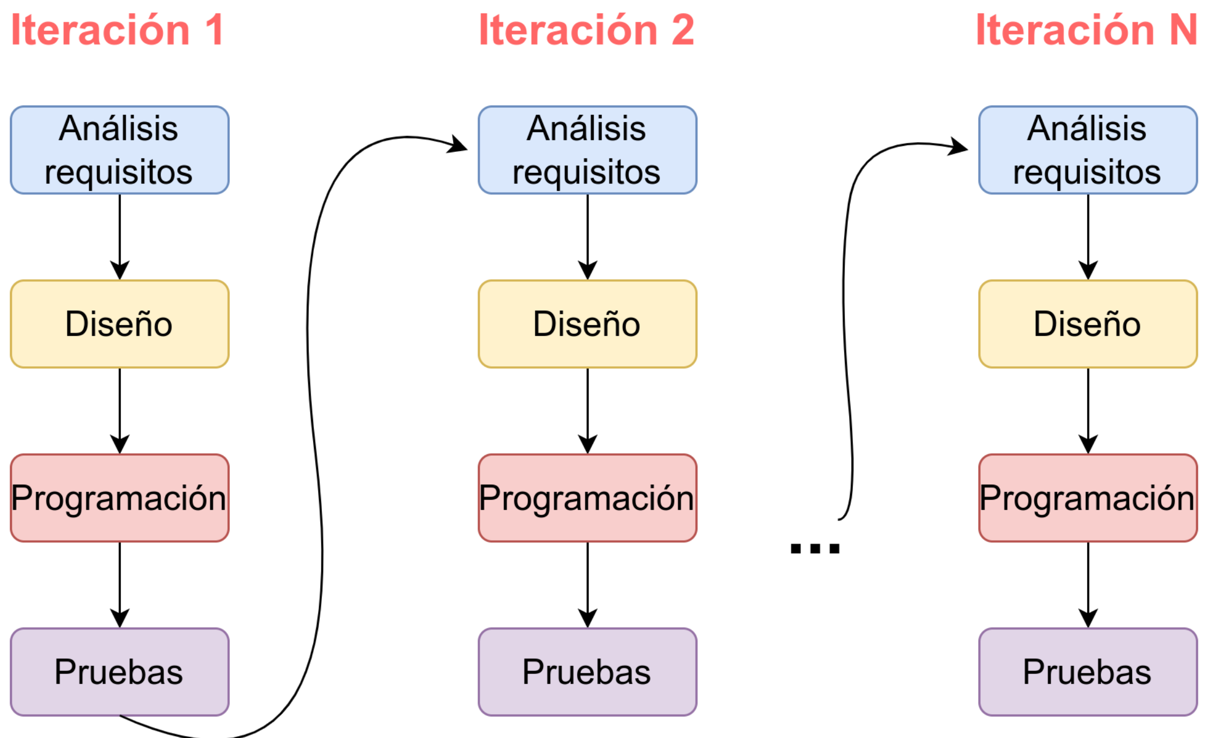

Figura 1.3: Esquema de la Metodología Incremental

2. Herramienta llamada *Fabricante*, que genera automáticamente a los consumidores.
3. Código fuente del proyecto.
4. Memoria del proyecto junto con un manual de usuario para el uso y las reglas de estilo.

1.3. Metodología y Fases de Trabajo

El modelo o metodología incremental [2] es un proceso del desarrollo de software donde los requisitos se dividen en varios módulos diferenciados e individuales en el ciclo del desarrollo del software. Cada módulo puede incluir las siguientes fases: fase de análisis de requisitos, diseño, implementación y pruebas; el conjunto de fases se denomina iteración. Cada salida subsecuente de una iteración incluye los resultados de ciclos anteriores. Este proceso continúa hasta el el sistema está completo, tal y como se muestra en la Figura 1.3.

Esta metodología ha permitido tener varios prototipos funcionales desde etapas tempranas del desarrollo. Dichos prototipos han estado centrados en probar funcionalidades concretas para decidir si incluirlas en el resultado final.

El proceso de desarrollo ha constado de siete fases de longitud variable. Los objetivos y tareas de cada una de ellas han sido los siguientes:

- **Fase 1:** Esta iteración consistió en una exploración de las tecnologías existentes en el mundo de IoT para delimitar los objetivos concreto del proyecto.

- **Fase 2:** Con las tecnologías seleccionadas en la fase anterior, esta iteración tenía como objetivo familiarizarse con los recursos elegidos.
- **Fase 3:** En esta fase se realizaron prototipos de aplicaciones consumidoras tanto para el protocolo AMQP como para el protocolo MQTT.
- **Fase 4:** La siguiente iteración consistió en desarrollar el *Fabricante*.
- **Fase 5:** Esta iteración consistió en desarrollar la parte de la interfaz gráfica del *Fabricante*. También se añadieron opciones de configuración a los consumidores.
- **Fase 6:** En esta fase se integró toda la lógica detrás de los consumidores configurables en el *Fabricante*, para que éste pudiera generarlos.
- **Fase 7:** Esta iteración tenía como meta mejorar la interfaz gráfica del *Fabricante* que para que fuera más agradable visualmente y añadir opciones de accesibilidad para que su uso fuera más cómodo. Además, se hizo al *Fabricante* compatible con sistemas operativos Linux.

En definitiva, el modelo incremental ha permitido un mejor flujo de trabajo gracias a que, en cada iteración, se han ido definiendo y desarrollando nuevos requisitos y funcionalidades.

1.4. Estructuración de la memoria

Capítulo 2. Tecnologías

Este capítulo estará centrado en detallar las tecnologías utilizadas en el desarrollo de este proyecto, así como los protocolos de comunicación que se usan en la interconexión de mensajes.

Capítulo 3. Diseño

En este capítulo se explicará el funcionamiento tanto del *Fabricante*, aplicación que genera nodos de procesamiento flexibles, como de los nodos consumidores de datos generados por el *Fabricante*.

Capítulo 4. Desarrollo

Este capítulo profundizará en el proceso de implementación de las funcionalidades descritas en el capítulo anterior. Además, se abordarán decisiones de implementación tomadas en diferentes puntos del desarrollo.

Capítulo 5. Pruebas y Casos de estudio

El capítulo consistirá en mostrar pruebas de carácter general para después enseñar el potencial del proyecto usando casos de estudio reales e implementados.

Capítulo 6. Conclusiones y líneas futuras

Para finalizar, se expondrán las conclusiones para posteriormente realizar un análisis del estado final de los requisitos planteados al principio. Y finalmente se indicarán posibles mejoras o ideas para explorar en trabajos futuros.

CAPÍTULO 2

Tecnologías

Este capítulo introduce las tecnologías utilizadas en el proyecto, los lenguajes de programación y sus respectivos IDE, las librerías e interfaces, y el concepto de bróker de mensajes. En este último caso, se explica con más detalle RabbitMQ que es el que se ha usado en este trabajo, así como los protocolos de comunicación más usados por los brókers y los que se han seleccionado para la arquitectura creada en este proyecto.

2.1. Lenguajes y entornos de desarrollo

Como mención especial, se ha utilizado LaTeX para la redacción del presente documento. LaTex es un sistema de composición de textos, enfocado en la creación de libros, artículos académicos, tesis, y en general todo tipo de documentos relacionados con el ámbito científico y técnico. Es software libre con licencia LPPL [3].

Existen múltiples editores para LaTex, pero en concreto en este proyecto se ha usado Overleaf, una herramienta de publicación y redacción colaborativa en línea que facilita el proceso de creación de documentos [4]. Entre las principales características de Overleaf que lo han hecho popular se encuentran la posibilidad de edición y colaboración de varios autores en tiempo real y la gestión de versiones.

2.1.1. Java

Java es un lenguaje de programación desarrollado por Sun Microsystems (en la actualidad Oracle) y actualmente uno de los más populares [5]. Se trata de un lenguaje de alto nivel, orientado a objetos y fuertemente tipado.
La característica principal por la que es tan popular es su portabilidad, es decir, si un programa desarrollado en Java es ejecutado en una plataforma específica, no necesita ser recompilado para ejecutarse en otra. Esta propiedad se debe a que las aplicaciones de Java son compiladas a Bytecode, un lenguaje de bajo nivel que puede ser interpretado en cualquier máquina virtual Java (JVM) independientemente del sistema donde se encuentre, como se muestra en la Figura 2.1. Actualmente existen JVM para varias arquitecturas de procesador (Intel y ARM las más populares) y sistemas operativos como Windows, Mac OS o Linux.

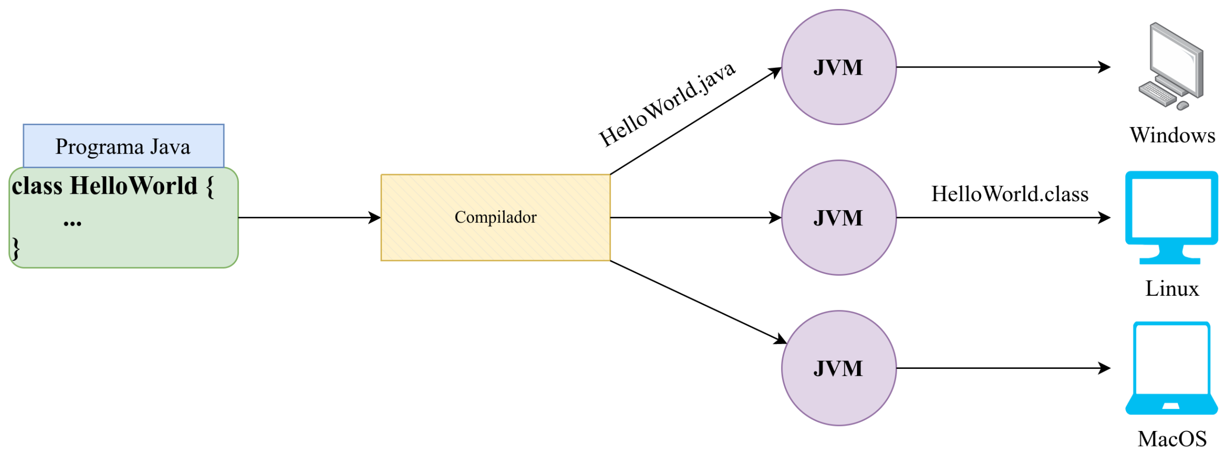

Figura 2.1: Esquema de la portabilidad de Java

Otras características de Java son que es un lenguaje de propósito general y multi-hilo; que es simple y fácil de aprender y que tiene una enorme comunidad de desarrolladores que lo apoyan, por lo que existen muchas librerías e interfaces para el desarrollo de aplicaciones Java desarrolladas por terceros. A continuación, presentamos algunos de los plugins y librerías Java que se han utilizando en este proyecto.

Maven

Maven es una herramienta usada para la construcción y gestión de cualquier proyecto basado en Java. [6]

Su característica principal es el facilitar el proceso de construcción de proyectos, mediante una plantilla uniforme.

El archivo clave de los proyectos hechos en Maven es el POM (Project Object Model), que se utiliza para describir el proyecto en construcción, sus dependencias de módulos y componentes externos como plugins.

Maven posee un repositorio [7] desde donde se descarga tanto los plugins como las dependencias necesarias automáticamente y de forma dinámica. Dicho repositorio proporciona acceso a diferentes versiones de diversos proyectos Open Source de Java.

Para este trabajo se ha usado Maven integrado en el entorno de desarrollo Eclipse.

2.1.2. Formatos y especificaciones

Este fragmento de la memoria está dedicado a introducir el formato JSON y la especificación JSON Schema, conceptos clave para entender toda la lógica que hay detrás del proyecto.

JSON

JSON (“JavaScript Object Notation”, Notación de Objetos de JavaScript) [8] es un estándar [9] que define un formato ligero de texto para el intercambio de datos indepen-

```
{
  "array": [
    {"clave": "valor"},
    3
  ],
  "boolean": true,
  "color": "azul",
  "null": null,
  "numbero": 123,
  "objeto": {
    "a": "b",
    "c": "d"
  },
  "string": "Hola mundo"
}
```

Figura 2.2: Ejemplo del formato JSON

diente del lenguaje. Es fácil de escribir y de entender por los humanos; para las máquinas es sencillo de interpretar y generar. JSON está constituido por dos estructuras:

A grandes rasgos, el formato JSON se presenta como un objeto, que es una colección desordenada de pares nombre/valor (o clave/valor). Un objeto comienza con "{" y termina con "}". Cada nombre es seguido por ":" y después por el valor. Los pares nombre/valor están separados por ",". Los nombres son siempre cadenas de texto y los valores pueden ser una cadena de texto, un número, null, un booleano, otro objeto o un array (son estructuras que puede ir anidadas). La Figura 2.2 muestra un ejemplo de una cadena en formato JSON.

JSON Schema

JSON Schema (Esquema JSON) [10] es una especificación del formato JSON que se usa para definir la estructura que van a tener los datos en JSON.

El esquema se describe con el mismo formato que un objeto JSON. Se podría considerar como datos en sí, es una forma de declarar la estructura de otros datos.

GSON

Gson(Google Gson) [11] es una librería de código abierto para Java que se usa para convertir objetos Java en una representación en notación JSON (serializar). La misma librería permite realizar el proceso inverso (deserializar) y transformar una cadena JSON en un objeto Java equivalente.

La librería GSON está disponible tanto en el repositorio Maven, como en Gradle (otro gestor de proyectos para Java), y además también está disponible una versión Jar para añadir manualmente a los proyectos.

Jsonschema2pojo

Jsonschema2pojo [12] es una librería Java que permite la creación de clases de Java a partir de un esquema JSON.

Hay dos formas de usarlo como plugin de Maven. Por un lado, se puede definir su comportamiento en el POM de Maven (no hay que introducir ningún comando dentro del código de Java), con lo que crea las clases en el momento de la compilación a partir de los esquemas dentro de un directorio especificado. Por otro lado, se puede utilizar mediante comandos dentro del código Java. Esta última es la opción elegida ya que permite un uso más extendido, pues pueden crearse las clases en el momento de ejecución.

Apache Commons CLI

Apache Commons CLI es una API que facilita el desarrollo de programas que admitan argumentos por línea de comandos. Además, también permite generar fácilmente mensajes personalizados de ayuda y alertas de errores para facilitar el uso de las opciones disponibles por línea de comandos.

Se ha usado el plugin disponible en el repositorio de Maven.

2.1.3. Eclipse: Entorno de desarrollo para Java

Eclipse en un entorno de desarrollo integrado (IDE) libre y open source conocido por su versión de Java [13], aunque también está disponible para dar soporte al desarrollo en otros lenguajes de programación como C/C++, PHP y JavaScript/TypeScript, entre otros. Incluso se puede utilizar la misma versión para dar soporte a diferentes lenguajes. En el proyecto se ha utilizado este entorno de desarrollo, en el que se ha instalado el plugin para Maven.

2.2. Bróker de mensajería

Un *bróker* de mensajería [14] (message broker o servidor de mensajería) es un programa que sirve de mediador (middleware orientado mensajes) entre aplicaciones o dispositivos, tanto receptores como emisores, que intercambian mensajes.

En el mundo del IoT, un bróker de mensajería consiste en un servicio de notificaciones centralizado. Dicho servicio presenta una arquitectura que consta de un servidor central que se encarga de recibir los mensajes de todos los dispositivos emisores y distribuirlos a los receptores. Este servidor tiene una dirección fija y conocida por todos los dispositivos. Todos los dispositivos que están conectados al bróker solo son conscientes de ellos mismos y no saben que otros dispositivos están conectados ni cual es el verdadero origen de los mensajes que reciben; esto es gracias a que el bróker se encarga de registrar todas las conexiones, recibir los mensajes y distribuirlos, por lo que proporciona escalabilidad y bajo acoplamiento. Esta estructura puede contemplarse en la Figura 2.3

Actualmente, existe una gran variedad de brókers disponibles, tanto específicos de un protocolo como más versátiles. Entre los más populares están:

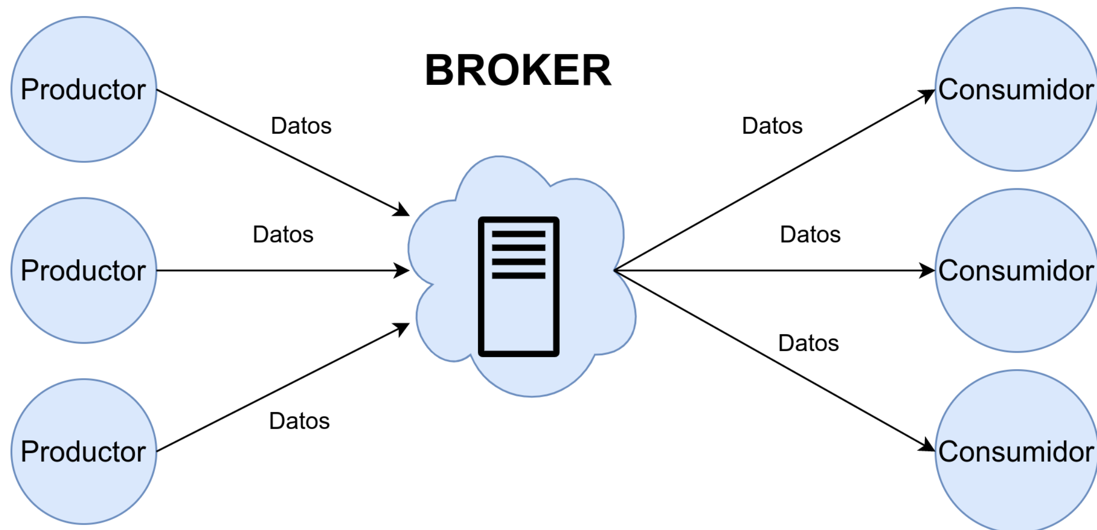

Figura 2.3: Esquema de la arquitectura de los brókers de mensajería

- **RabbitMQ:** Popular bróker de mensajería Open Source centrado en el protocolo AMQP [1], que además soporta el protocolo MQTT a través de un plugin. Es el bróker empleado en este trabajo.
- **Apache Kafka:** Es una plataforma de transmisión de datos Open Source [15] que está diseñada para administrar flujos de gran cantidad de datos en tiempo real desde varias fuentes y distribuirlos a diversos usuarios. Posee un protocolo basado en TCP que está optimizado para ser eficiente. Comenzó siendo un sistema interno de Linkedin.
- **Mosquitto:** Bróker específico de MQTT pensado para ser ligero y adecuado para todo tipo de dispositivos [16]. Es un bróker Open Source desarrollado por la fundación Eclipse.
- **Amazon Simple Queue Service (Amazon SQS):** Es un sistema de colas distribuido desarrollado por Amazon. La ventaja es que evita la complejidad de instalar y administrar cualquier otro tipo de middleware orientado a mensajes. Es muy escalable ya que utiliza AWS (Amazon Web Services) para ajustar su capacidad en función de la demanda.

Como se ha mencionado en apartados anteriores, para este proyecto se hará uso de RabbitMQ. En la siguiente sección, se hará una introducción general a los protocolos de comunicación de IoT para luego dar paso a la explicación de dos de los protocolos soportados por RabbitMQ que se han utilizado en este trabajo.

2.2.1. Protocolos de comunicación

Los protocolos de comunicación especifican una serie de normas para que diferentes dispositivos puedan comunicarse, así como el formato que deben presentar los mensajes intercambiados. En entornos IoT, los protocolos han de cubrir aspectos adicionales propios de la naturaleza del Internet de las Cosas [17]. Primero, hay que tener en cuenta la gran cantidad dispositivos que pueden llegar a estar conectados y la diferencia de tareas que desempeñan así como la disparidad en la potencia. Segundo, un requisito que se espera

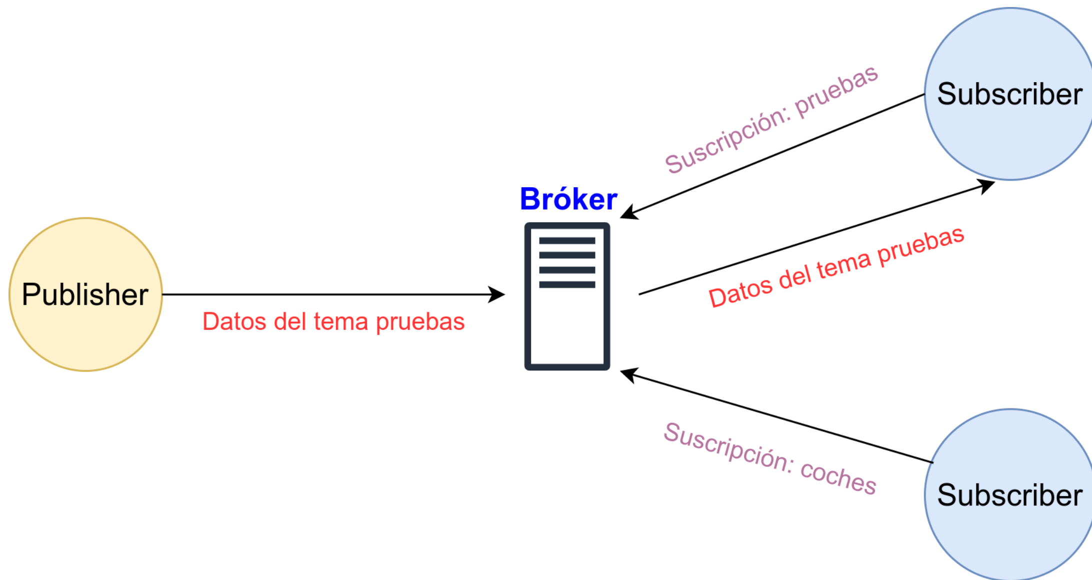

Figura 2.4: Esquema del patrón Publish/Subscibe

es que la arquitectura sea escalable, pudiendo añadirse o dar de baja dispositivos en el sistema sin que se perciba diferencia alguna. Esto conlleva adoptar un bajo acoplamiento entre dispositivos. Finalmente, debe asegurarse la seguridad de las comunicaciones.

Los protocolos de IoT se basan en diferentes patrones de mensajería, entre los más comunes encontramos:

- Publish/Subscribe: Se basa en que una entidad *Subscriber* informa a un bróker sobre el tipo de mensajes que quiere recibir. Por otro lado, la entidad *Publisher* es la encargada de enviar datos al bróker sobre un tema determinado. Como muestra la Figura 2.4, el bróker es el encargado de distribuir los mensajes a los *Subscribers* suscritos a cada tema.

- Remote Procedure Calls (RPC): Este patrón, que se muestra en la Figura 2.5, consiste en que un “Callee” hace saber al bróker que puede realizar un procedimiento en concreto. Por otro lado, el “Caller”, es quien hace uso de este procedimiento a través de llamadas al bróker, que es el que invoca al procedimiento del Callee, recoge el resultado y lo reenvía al Caller que lo ha solicitado.

Las infraestructuras de servicio consisten en un middleware que ayuda al envío y recepción de mensajes entre sistemas distribuidos. Dichas infraestructuras pueden estar orientadas a mensajes (Message-oriented middleware MOM), en cuyo caso se centran en implementar patrones como PubSub; u estar orientadas a RPC (RPC-based middleware).

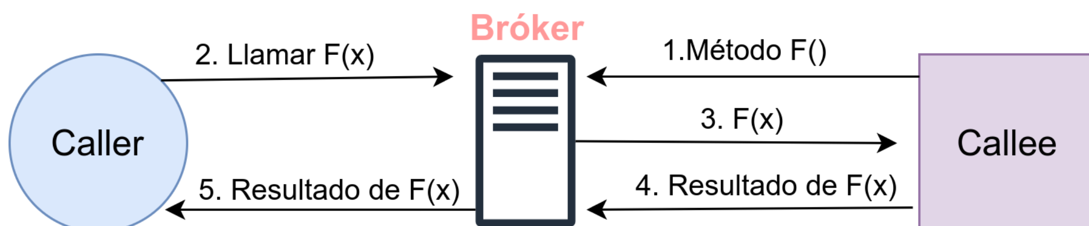

Figura 2.5: Esquema del patrón RPC

Nombre	Patrón	Protocolo Transporte	Seguridad
MQTT	Pub/Sub	TCP	SSL, TLS
AMQP	Pub/Sub, RCP	TCP	TLS
WAMP	Pub/Sub, RPC	TCP	TLS
CoAP	REST	UDP	DTLS
XMPP	Pub/Sub	TCP	SSL, TLS

Tabla 2.1: Tabla comparativa entre protocolos IoT

Puede darse el caso de infraestructuras mixtas que implementan más de un patrón de mensajería o soluciones híbridas. En cualquier caso proveen, entre otras cosas, elementos de software (servicios) que implementan las características de los diferentes patrones de mensajería. A continuación se explicarán dos de las más empleadas en cuanto al patrón Publish/Subscribe (patrón elegido como forma de intercambio de mensajes en este proyecto):

- **Message Queue:** Es un servicio de mensajería en el que el bróker genera una cola de mensajes única para cada cliente que ha comenzado una subscripción. El bróker es el encargado de filtrar mensajes para que al cliente solo le lleguen los especificados. En este caso, se mantienen los mensajes en la cola hasta que son entregados al cliente una vez que se conecte.

- **Message Service:** Este planteamiento es similar al anterior, con la única diferencia que los mensajes se distribuyen inmediatamente a los clientes conectados, perdiéndose los mensajes si el cliente está desconectado. Este servicio es el que se utilizará para el trabajo.

A partir de las tecnologías presentadas, surgen una gran cantidad de protocolos para IoT, entre los que encontramos:

- MQTT (MQ Telemetry Transport): Es un protocolo PubSub de Message Service. Es uno de los más usados y se detallará en una sección posterior.

- AMQP (Advanced Message Queueing Protocol): Protocolo PubSub de Message Queue. Al igual que MQTT, es uno de los más populares y también se explicará en detalle más adelante.

- WAMP (Web Application Messaging Protocol): Es un protocolo que se ejecuta sobre WebSockets, adapta tanto los patrones de PubSub como RPC.

- CoAP (Constrained Application Protocol): Protocolo enfocado a dispositivos de baja capacidad de cómputo.

- XMPP (Extensible Messaging and Presence Protocol): Es un protocolo basado en XML que suele ser usado en aplicaciones de mensajería instantánea.

La siguiente Tabla 2.1 recoge las principales características de los protocolos introducidos anteriormente. A continuación, presentaremos con más detalles los protocolos MQTT y AMQP que serán los que consideraremos en este trabajo.

MQTT

MQTT o MQ Telemetry Transport es un estándar abierto de OASIS tanto en su versión 3 [18] como en su versión 5 [19] que define un protocolo de comunicación Máquina a Máquina (M2M) y entornos IoT. Está diseñado para el transporte de mensajes mediante publicación/subscripción (publish/subscribe) y tiene como base la pila TCP/IP para la comunicación [20]. Los clientes MQTT requieren pocos recursos y son muy livianos ya que están enfocados a la conexión de dispositivos con poca capacidad como microcontroladores. Además, las cabeceras de los mensajes son pequeñas para adaptarse a redes con bajo ancho de banda [21]. Los datos transportados son binarios.

MQTT soporta sesiones con persistencia para reducir el tiempo de reconexión del cliente con el bróker. Además, este protocolo asegura el envío fiable de mensajes con tres niveles de calidad de servicio diferentes. Con el nivel 0 un mensaje llega a lo sumo una vez (sin ninguna certeza), con el nivel 1 el mensaje llega al menos una vez (pueden llegar duplicados) y con el nivel 2 se asegura que el mensaje llega exactamente una vez (máxima certeza).

MQTT permite la comunicación bidireccional entre dispositivos y servidores, y también permite el broadcasting.

Por último, con respecto a la seguridad, hay diferentes mecanismos de seguridad disponibles como, por ejemplo, cifrar las conexiones mediante SSL/TLS o usar autenticación con usuario y contraseña.

AMQP

Advanced Message Queuing Protocol (AMQP) es un estándar de protocolo abierto, también estandarizado por OASIS [22], que funciona sobre la capa de aplicación y permite la interoperabilidad entre sistemas, independientemente del bróker de mensajería o la plataforma usada [23]. Es un protocolo binario, es decir, que todo lo enviado a través de él son datos binarios. Por defecto actúa como message queue 2.2.1, aunque se puede configurar para que los mensajes no sean almacenados (message service). En cuanto al modelo de este protocolo, tenemos cuatro componentes claves:

- El mensaje
- El productor (publisher) que crea el mensaje y lo envía al bróker o servidor de mensajería.
- El bróker de mensajes, que distribuye el mensaje teniendo en cuenta las reglas especificadas en el mensaje (distinto al contenido del mensaje) en diferentes colas.
- El consumidor (consumer), que obtiene el mensaje de la cola a la que tiene acceso y finalmente lo procesa.

La característica diferenciadora de este protocolo es el llamado intercambiador (*exchange*) que se encuentra dentro del bróker y cuyo cometido es gestionar la distribución de los mensajes en las colas [24], tal y como se muestra en la Figura 2.6. Realmente el publisher envía el mensaje al intercambiador y no tiene porqué saber a qué cola está destinada el mensaje. Las reglas de vinculación entre el intercambiador y las colas están definidas por

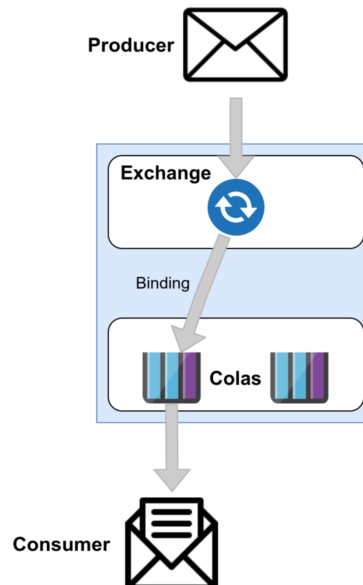

Figura 2.6: Fundamentos de AMQP

el tipo de intercambio. Al igual que los mensajes, el intercambiador puede ser persistente o borrarse una vez se pierde el canal de conexión.

En AMQP se pueden intercambiar mensajes de cuatro maneras diferentes:

1. Intercambio directo: Existe un parámetro denominado clave de vinculación ("binding key") que vincula una cola y un intercambiador. Por otro lado, cada mensaje tiene un valor llamado clave de enrutamiento ("routing key"). Si la clave de enrutamiento de un mensaje que llega al intercambiador coincide con la clave de vinculación de una cola, ésta última recibe ese mensaje.

2. Intercambio por temas: Es parecido al directo solo que las clave de vinculación y enrutamiento no tienen porqué coincidir exactamente. En este caso, la clave de vinculación es una *expresión regular* y si el valor de la clave de enrutamiento está dentro de su patrón ese mensaje es recogido por la cola con dicha vinculación. Este intercambio se usa para implementar variaciones del patrón Publish/Subscribe.

3. Intercambio por cabecera: similar al intercambio por temas. La diferencia radica en que la vinculación depende del valor de la cabecera del mensaje.

4. Intercambio en abanico: en este caso, el intercambiador envía el mensaje a todas las colas que conoce.

Aunque su funcionamiento está definido en base al intercambiador, este protocolo puede adaptarse para replicar el funcionamiento del patrón de publish/subscribe y el del Remote Procedure Call (RPC).

Interoperabilidad

Teóricamente el uso de los protocolos es independiente del resto de la arquitectura. Sin embargo, tras estudiar la tecnología y los protocolos detrás de los bróker de mensajería, se han descubierto problemas a la hora de comunicar algunas implementaciones de clientes

(publisher o consumer) con brókers que admitan el mismo protocolo debido a que dichos servidores pueden tener implementadas parcialmente especificaciones de los protocolos o porque la implementación del cliente no soporte algunas de las características ofrecidas por el bróker.

Sin entrar en demasiados detalles, uno de los artículos consultados [25] y realizado por Rabbit Technologies Ltd., la empresa encargada del desarrollo de RabbitMQ, menciona que han realizado pruebas enviando mensajes a distintos brókers de mensajería usando diferentes clientes. El resultado es que para el protocolo AMQP no hay una interoperabilidad total, ya sea por bugs, porque las entidades que se comunican implementan diferentes versiones del mismo protocolo o porque no implementan todas las características descritas por la especificación del protocolo, llegando al punto en que en ciertos casos los brókers no implementan gran parte de las especificaciones del protocolo (como es el caso del bróker OpenAMQ).

En cuanto a MQTT, no hay artículos recientes relacionados con la interoperabilidad, aunque no se descarta que pueda presentar problemas parecidos.

2.2.2. RabbitMQ: Clientes en Java

Existen librerías que evitan la tarea de crear un cliente que utilice un protocolo de mensajería desde cero, aplicando una capa de abstracción con métodos para conectarse a un bróker, y para el envío y recepción de mensajes. Para el proyecto se va a hacer uso de dos librerías, cada una de ellas da soporte a uno de los protocolos escogidos.

Para el envío de mensajes al bróker de mensajería RabbitMQ se han usado AMQP client, que es la solución para desarrollar aplicaciones cliente que utilicen el protocolo AMQP para el envío y recepción de mensajes, además de conectar al servidor RabbitMQ; es desarrollada por la empresa de RabbitMQ. Eclipse Paho es la librería usada para el desarrollo de clientes Java que implementen toda la lógica del uso del protocolo MQTT, ha sido desarrollada por la Eclipse Foundation.

Eclipse Paho

Eclipse Paho [26] es una librería para implementar, en aplicaciones Java, la conexión, envío y recepción de mensajes con un bróker compatible con el protocolo MQTT.

A la hora de programar usando esta librería hay que tener en cuenta que:

1. Posee una serie de clases que actúan como clientes y que hay que instanciar para conectarse a un bróker. Todas necesitan un identificador único para conectarse y la dirección del servidor.

2. Se puede seleccionar si mantener la configuración de las conexiones de una sesión a otra o no.

3. Los clientes se pueden suscribir a topics absolutos o a expresiones regulares.

4. Hay dos clases clientes. Por un lado, *MqttAsyncClient* implementa una interfaz de métodos no bloqueantes y cuyo flujo hay que controlar. Por otro lado, en la clase *MqttClient* donde todos los métodos se bloquean hasta que la operación que realizan es completada. Ambas clases tienen en común métodos cuyas operaciones son

asíncronas: *messageArrived* que notifica cuando llega un mensaje, *connectionLost* que notifica cuando se ha perdido la conexión con el servidor y *deliveryComplete* que notifica cuando un mensaje se ha enviado al servidor.

5. La clase *MqttConnectOptions* se utiliza para modificar las opciones de conexión por defecto, hay que indicarlo antes de establecer la conexión. Por ejemplo, se puede decidir si borrar las configuraciones de una sesión anterior, introducir varias direcciones alternativas del servidor y modificar las credenciales de seguridad.

AMQP client

La librería AMQP client para Java permite el desarrollo de aplicaciones cliente capaces de interactuar con RabbitMQ [27] usando el protocolo AMQP.

La librería permite una abstracción de una conexión socket, negociando automáticamente la versión del protocolo y la autenticación. Solo hace falta indicar parámetros como la dirección del servidor y las credenciales si es necesario. Tras abrir la conexión, la librería posee métodos para crear canales dentro de esa conexión. También ofrece métodos para configurar el modo en el que se van a comportar la distribución de mensajes, pudiéndose crear intercambiadores, colas y enlaces entre ellos. Además, también se pueden configurar diferentes formas de publicar y recibir mensajes.

CAPÍTULO 3

Diseño

En este capítulo se desgranará todo el proceso correspondiente al diseño del proyecto, empezando por una descripción general, pasando por la especificación de requisitos tanto funcionales como no funcionales y finalizando por una descripción detallada de la estructura y comportamiento de cada componente.

3.1. Descripción general

El objetivo de este proyecto consiste en diseñar e implementar la herramienta *Fabricante* para facilitar el desarrollo de aplicaciones de procesamiento de datos (consumidores) en una arquitectura IoT con un servidor de mensajería centralizado (bróker).

El uso de la herramienta tendrá como resultado consumidores cuya configuración es flexible. Para ello, se diseñarán plantillas de consumidores que permitirán configurar parámetros como la condición de parada, el tiempo de espera o, incluso, el evento que hace que se ejecute el algoritmo que procesa los mensajes obtenidos. Estas plantillas estarán integradas en la herramienta.

La Figura 3.1 muestra una visión general de la propuesta desarrollada en el presente proyecto, en la que tenemos los siguientes componentes:

- El publisher: Son elementos secundarios usados para hacer pruebas de los consumidores creados. A partir de ellos se podrán elegir los mensajes para enviar. Hay uno para cada protocolo.
- RabbitMQ: Es el bróker de mensajería elegido para usar en el proyecto. Es ampliamente usado y tiene una gran cantidad de manuales y ejemplos sobre su uso. Soporta también los dos protocolos seleccionados, MQTT Y AMQP.
- El consumidor: Nodo de procesamiento independiente que será capaz de conectarse al servidor RabbitMQ que se le especifique, y realizar ciertas acciones dependiendo de los parámetros introducidos cuando se llama.
- El *Fabricante*: Esta es la herramienta que es desarrollada en el proyecto. Es la encargada de facilitar el desarrollo de un consumidor flexible a través de parámetros que se especificarán más adelante.

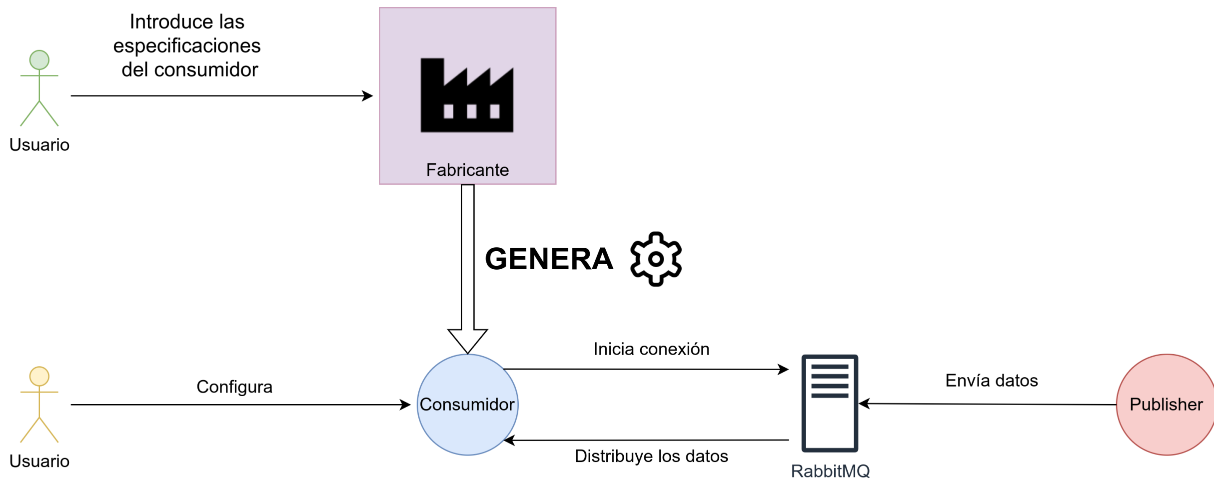

Figura 3.1: Visión general del proyecto

3.2. Análisis de requisitos

Como se ha mencionado en el apartado anterior, hay realmente dos entidades a desarrollar, por lo que la especificación de requisitos ha de diferenciar entre los requisitos del *Fabricante y los del consumidor.*

Dentro de cada sección de requisitos, se hará una división entre los requisitos funcionales y no funcionales. A modo de breve descripción de ambos conceptos:

Requisitos funcionales: *son descripciones y restricciones explícitas del comportamiento y de las funcionalidades que debe tener un determinado software.*

Requisitos no funcionales: *se tratan de especificaciones sobre los estándares de calidad, usabilidad, seguridad, portabilidad, y cualquier otro que no entre dentro del ámbito del comportamiento o funcionalidad de un sistema software.*

Para poder dar un orden de prioridad a los requisitos se ha utilizado la técnica denominada MoSCoW (Must Have, Should have, Could have, and Would like but won´t get). Se trata de una técnica de priorización de requisitos que no se basa en valores numéricos si no que se enfoca en darles un valor semántico. De esta manera, los requisitos se dividirán en cuatro grupos:

- ***M – Must:*** *Requisitos fundamentales y obligatorios.*
- ***S – Should:*** *Requisitos que deberían ser cumplidos en la medida de lo posible.*
- ***C – Could:*** *Requisitos que son interesantes y que es positivo cumplirlos pero pueden ser descartados si las condiciones dificultan su implementación.*
- ***W – Won't****: Requisitos que no se cumplirán en la fase actual pero que podrían desarrollarse en el futuro.*

3.2.1. Requisitos del Consumidor

Tras una primera fase de análisis de los requisitos (funcionales y no funcionales) se han identificado los requisitos que se muestran en las Tablas 3.1 y 3.2. Cada requisito

Id	Descripción	Prioridad
RF-1	Debe poder aceptar la dirección del servidor de mensajería RabbitMQ especificado por parámetro	M
RF-2	Debe poder aceptar por parámetro el nombre de la cola de la que recibe los mensajes	M
RF-3	Su comportamiento ha de ser configurable a través de argumentos en el momento en el que es llamado para su ejecución	M
RF-4	Dadas todas las especificaciones y direcciones necesarias, ha de ser capaz de conectarse a RabbitMQ, esperar mensajes y procesarlos con el comportamiento seleccionado	M
RF-5	Debe tener un sistema de aviso de errores cuando se introduzcan opciones en la configuración incompatibles o cuando falten datos de configuración necesarios	C
RF-6	El consumidor debería tener un comando que imprima una ayuda que especifique las opciones disponibles	S
RF-7	El consumidor está preparado para ser reutilizable con diferentes configuraciones	S

Tabla 3.1: Requisitos funcionales del consumidor

Id	Descripción	Prioridad
RNF-1	Su uso debe ser sencillo y amigable para todo tipo de usuarios	S
RNF-2	El consumidor es un archivo JAR cuyas dependencias están contenidas en su interior	W
RNF-3	La aplicación tiene un manual de uso completo e ilustrativo	S
RNF-4	Debe poder ser ejecutada en cualquier sistema que posea el Java Runtime Environment (Java JRE)	M

Tabla 3.2: Requisitos no funcionales del consumidor

tiene un identificado (RF-x si es funcional y RNF-y para los no funcionales), una breve descripción y la prioridad que se les ha asignado.

3.2.2. Requisitos del Fabricante

El diseño del fabricante es un poco especial ya que por su condición es dependiente de los requisitos que poseen los consumidores que crea. Por ello, primero se han definido éstos antes que los del fabricante. Las Tablas 3.3 y 3.4 presentan los requisitos funcionales y no funcionales del fabricante.

Id	Descripción	Prioridad
RF-1	El fabricante es una aplicación que tendrá una serie de pasos en los que recogerá las especificaciones necesarias para crear un consumidor	M
RF-2	En el fabricante se podrá decidir el nombre de la clase del consumidor a crear	M
RF-3	Habrá alguna manera de introducir el algoritmo que el consumidor ejecutará para procesar los datos	M
RF-4	Se podrá describir en el fabricante el tipo de datos que el consumidor recibirá y procesará	M
RF-5	En el fabricante se podrá elegir el protocolo que usará el consumidor a crear, MQTT o AMQP, pero no ambos	M

Tabla 3.3: Requisitos funcionales del fabricante

Id	Descripción	Prioridad
RNF-1	Debe poseer una interfaz gráfica sencilla de usar e intuitiva	S
RNF-2	El fabricante es un archivo JAR con dependencias en su interior	S
RNF-3	La aplicación tiene un manual de uso completo e ilustrativo	S
RNF-4	Debe ser compatible con varios Sistemas Operativos	C

Tabla 3.4: Requisitos no funcionales del fabricante

3.3. Diseño del Consumidor

Cuando hablamos del consumidor puede parecer que se trata de una única entidad, pero realmente son dos modelos de aplicaciones consumidoras de datos, una adaptada al protocolo AMQP y otra al protocolo MQTT.

3.3.1. Diseño de la estructura de consumidor

Como se puede observar en la Figura 3.2, el consumidor es una entidad que presenta tres módulos diferenciados:

1. Módulo de interacción con el usuario: *a partir de él se interpretan los argumentos introducidos por el usuario cuando éste quiere configurar el comportamiento del consumidor. El usuario podrá indicar aspectos como la dirección IP del servidor RabbitMQ a la que el consumidor tiene que conectarse, el nombre de la cola que tiene que crear; bajo qué condiciones se ha de ejecutar el algoritmo, etc.*

2. Módulo de gestión de las conexiones: *es el encargado de abrir un canal de conexión con el servidor RabbitMQ, y de esperar y guardar los mensajes recibidos. La forma en que se conecta con el bróker variará si se trata del modelo del consumidor que usa MQTT o AMQP.*

3. Módulo de procesamiento de datos: *este módulo es responsable de controlar*

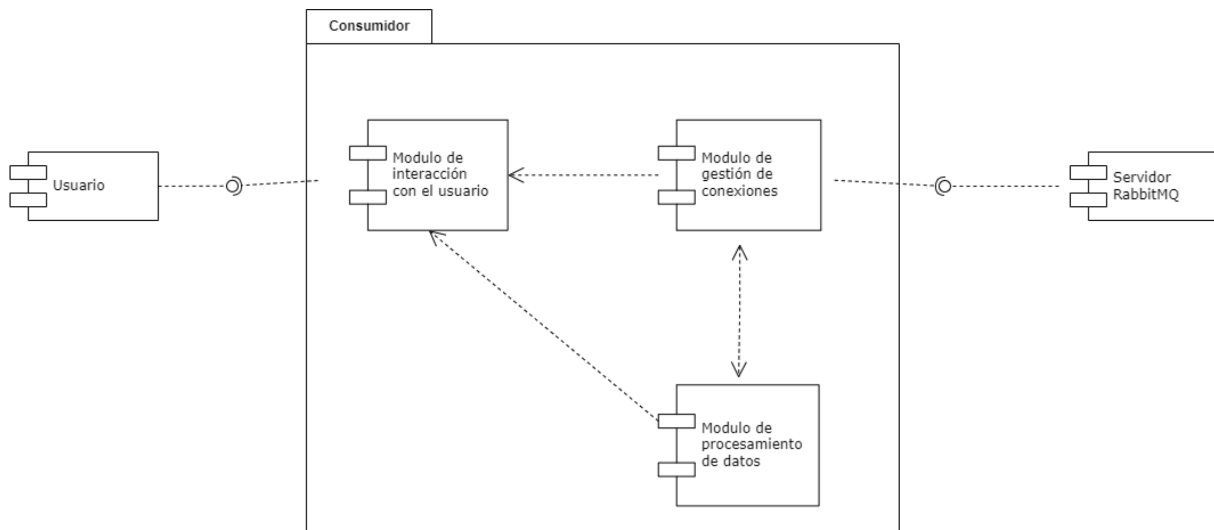

Figura 3.2: Diagrama de componentes del consumidor

cuándo y cómo se ejecuta el algoritmo que procesa los mensajes. Además, este módulo transforma el formato de los datos recibidos por el consumidor al que acepta el algoritmo.

El módulo de interacción con el usuario le ofrece al módulo de gestión de las conexiones la información necesaria para establecer una dirección con el servidor RabbitMQ. También le proporciona al módulo de procesamiento de datos información acerca de cómo configurar el comportamiento a la hora de gestionar los datos y ejecutar el algoritmo que le ha proporcionado el usuario.

El módulo de gestión de conexiones depende del módulo de interacción con el usuario porque necesita información crucial como la dirección IP del servidor RabbitMQ y el nombre de la cola o topic (dependiendo de si es la versión MQTT o AMQP) para comenzar la conexión. Es también dependiente del módulo de procesamiento de datos porque es el que le indica cuándo ha de terminar la conexión. Además, interacciona con el servidor RabbitMQ para recibir los datos.

El módulo de procesamiento de datos es dependiente del módulo de interacción con el usuario porque necesita saber cómo y bajo qué condiciones ejecutar el algoritmo para tratar los datos. También depende del módulo de gestión de conexiones porque es el que recibe y guarda los mensajes que hay que tratar.

3.3.2. Diseño de las funcionalidades del consumidor

En la fase de diseño del consumidor se han identificado una serie de opciones configurables debían estar disponibles para que una vez generado un consumidor se pueda utilizar de forma flexible. A continuación, se introducen estas opciones de configuración:

- ***Dirección del servidor RabbitMQ:*** *El usuario podrá introducir la dirección IP del servidor RabbitMQ.*
- ***Nombre de la cola/topic:*** *En el caso de la versión AMQP el usuario podrá introducir el nombre de la cola que se creará en la conexión. Para MQTT será el topic al que el consumidor se suscribirá.*

- ***Puerto de escucha del servidor RabbitMQ:*** *En el caso de que el servidor RabbitMQ esté configurado para escuchar en un puerto diferente del predeterminado, el usuario tendrá disponible esta opción para especificar a que puerto se tiene que conectar el consumidor.*

- ***Nombre de usuario:*** *Si la conexión necesita unas credenciales de autenticación, esta opción permite introducir el nombre de usuario.*

- ***Contraseña:*** *Al igual que el nombre de usuario, esta opción permite introducir la contraseña necesaria en una conexión con credenciales.*

- ***Tiempo de espera entre mensajes:*** *Se trata del tiempo máximo que el consumidor esperará para que llegue un mensaje. Este tiempo se restablece cada vez que llega un mensaje. Si se sobrepasa y no llega ningún mensaje, se cerrarán las conexiones y el consumidor terminará su ejecución.*

- ***Número máximo de mensajes:*** *Valor que indicará el número máximo de mensajes que el consumidor va a esperar. Una vez llega a esa cifra, el consumidor cerrará las conexiones y terminará su ejecución.*

- ***Intervalo de repetición temporal del algoritmo:*** *El usuario será capaz de indicar cada cuanto tiempo se ha de repetir la ejecución del algoritmo.*

- ***Repetición del algoritmo por número de mensajes recibidos:*** *Opción alternativa a la repetición del algoritmo por tiempo. En este caso, se repite el algoritmo tras la llegada de un determinado número de mensajes.*

- ***Desechar mensajes usados:*** *Opción que se usa para definir si se quieren reutilizar los mensajes que ya han sido usados por el algoritmo o si, por el contrario, se han de borrar para solo procesar los que no hayan sido tratados.*

- ***Ejecución final:*** *Opción que se usa para definir si ha de realizarse una última ejecución del algoritmo tras el cierre de conexiones. Pueden darse casos en los que queden mensajes sin tratar con ciertas configuraciones y se cumpla la condición de cierre y finalización del consumidor. Esta opción permite que esos mensajes puedan ser tratados justo antes de que la ejecución del consumidor termine.*

En el Capítulo 4 de implementación se detallarán las posibles incompatibilidades que se han identificado entre estas opciones.

3.4. Diseño del Fabricante

El fabricante consiste en una aplicación (o herramienta) para facilitar la creación de los consumidores cuyo diseño se ha descrito en la sección anterior. En esta herramienta se podrán especificar aspectos del consumidor como el protocolo de comunicación que utilizará (a elegir entre AMQP o MQTT), su nombre, el algoritmo que va a utilizar para procesar los mensajes o el formato de los datos que va a recibir a través del bróker.

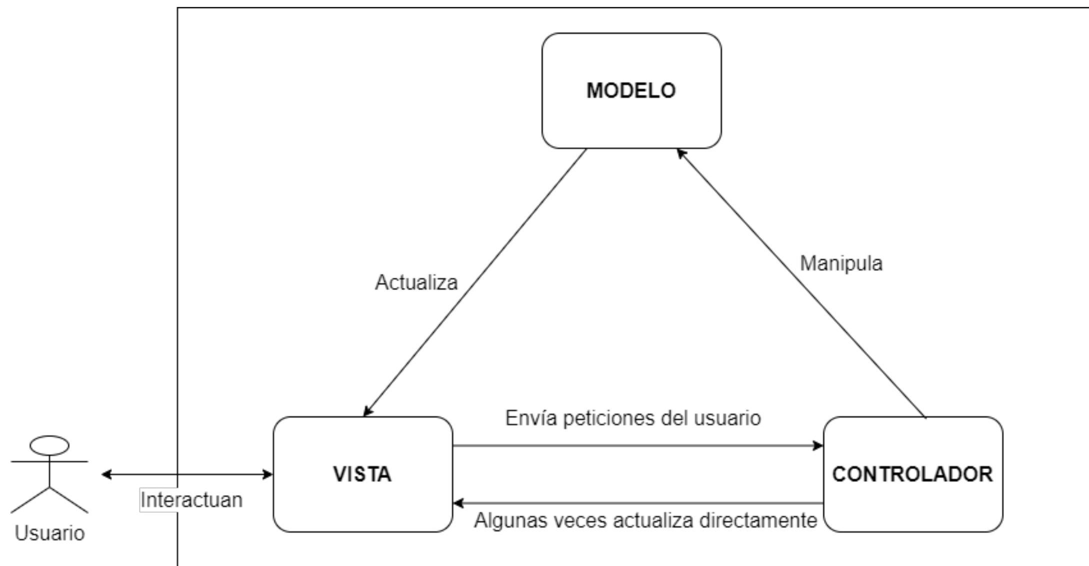

Figura 3.3: Esquema del patrón Modelo-Vista-Controlador

3.4.1. Diseño de la estructura del fabricante

El fabricante es una aplicación cuyo diseño e implementación sigue el patrón Modelo-Vista-Controlador.

El Modelo-Vista-Controlador (MVC) es un patrón de arquitectura de software que, utilizando tres capas diferentes llamadas Vista, Modelo y Controlador, separa la lógica de la aplicación de la lógica de la parte gráfica [28]. La razón por la que es tan utilizado es porque permite separar los componentes de una aplicación según la responsabilidad que tengan, es decir, cuando se hace una modificación en una parte del código, esto no debería afectar a otras partes del mismo (a esto se le llama principio de responsabilidad única). Como se puede observar en la Figura 3.3, los componentes del modelo, la vista y el controlador interactúan entre sí y tienen las siguientes funciones:

Modelo: *Es la capa encargada de los datos, define que datos debe contener la aplicación, presenta mecanismos para acceder a ellos y para actualizar su estado. El modelo del fabricante consiste en una serie de métodos y objetos destinados a la creación de los consumidores.*

Controlador: *Es el código encargado de enlazar la vista y el modelo. El controlador contiene una lógica que actualiza el modelo y/o vista en respuesta a las entradas de los usuarios de la aplicación. El controlador del fabricante consiste en una clase encargada de cambiar los paneles de la parte gráfica cuando se requiera, así como de utilizar los métodos disponibles en el modelo para recoger y tratar los datos introducidos por el usuario.*

Vista: *Se trata del código de la aplicación que se encarga de la representación visual de los datos y de la interacción con el usuario. Aunque la vista trabaja con los datos, no realiza ningún acceso directo a estos, simplemente muestra la salida que genera el modelo cuando la aplicación lo requiere. El fabricante tendrá como vista una interfaz gráfica compuesta por diversos paneles (o clases).*

Es necesario mencionar que la aplicación del fabricante se ofrecerá como un archivo JAR autosuficiente en su versión final.

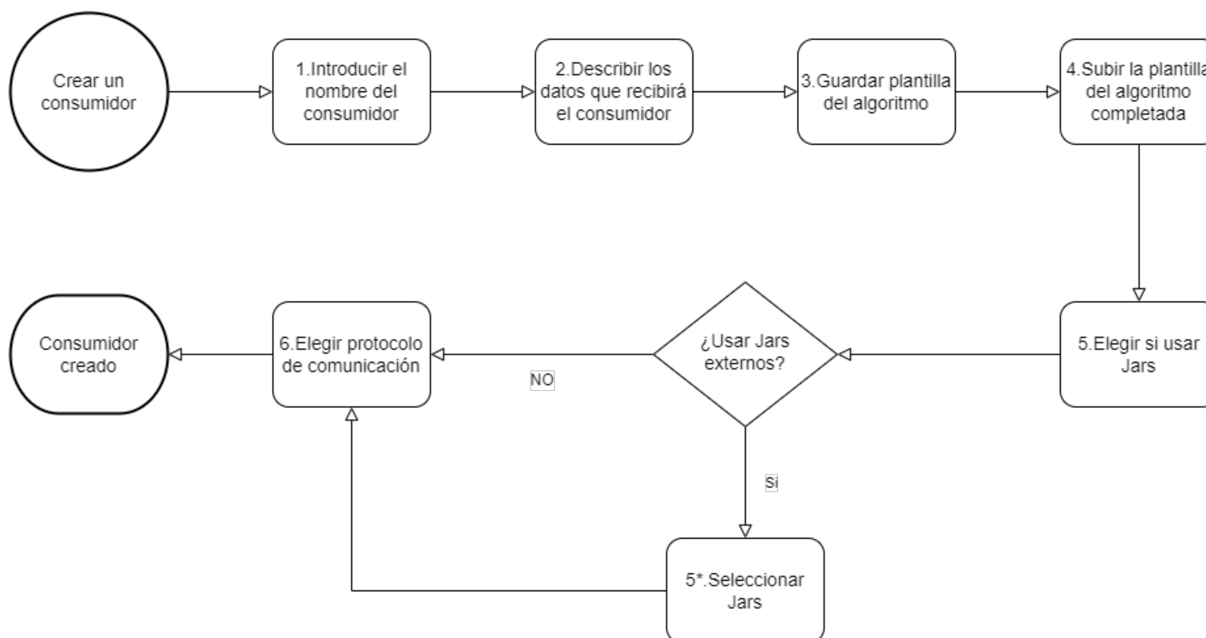

Figura 3.4: Diagrama de actividades del fabricante

3.4.2. Diseño de las funcionalidades del fabricante

Esta sección está dedicada al diseño de la funcionalidad que el fabricante presentará a la hora de la creación de un programa consumidor. Se ha optado por el uso del diagrama de actividades como medio de explicación de sus fases y la relación entre ellas.

La figura 3.4 muestra el diagrama de actividades de la aplicación conocida como fabricante, y que describe su flujo de ejecución normal. A continuación, se detalla en qué consisten cada uno de los pasos.

1.Introducir el nombre del consumidor: *En este paso, el usuario introduce el nombre que recibirá el consumidor. En el capítulo de implementación, se detallarán algunos aspectos relacionados con esta fase como, por ejemplo, las normas de nombrado del consumidor.*

2.Describir los datos que recibirá el consumidor: *El usuario tendrá que proporcionar el formato de los datos que el consumidor va recibir a través del bróker. Es necesario que el usuario especifique el formato para que el consumidor pueda procesar los mensajes recibidos. Los detalles de cómo será este proceso se darán en el capítulo de implementación.*

3.Guardar la plantilla del algoritmo: *En este paso la herramienta le ofrecerá al usuario una plantilla que deberá completar con el algoritmo que desea que el consumidor utilice para procesar los mensajes. Para la correcta integración del algoritmo en el consumidor que genera el fabricante, el usuario debe seguir unas pautas o reglas. Estos detalles se explicarán tanto en el capítulo de implementación como en el manual de usuario del fabricante.*

4.Subir la plantilla del algoritmo completada: *Una vez que el usuario complete la plantilla de la clase del algoritmo, éste tendrá que devolvérsela a la herramienta para que se pueda generar el consumidor.*

5.Elegir si usar JARs: *El algoritmo para procesar los datos puede ser bastante complejo como para que pueda especificarse por completo en una clase. Por ello, el usuario puede proporcionar bibliotecas adicionales utilizadas por el algoritmo. A partir de esta decisión tomada por el usuario, hay dos opciones:*

1. Si el usuario indica que el algoritmo no usa JARs externos, se procederá directamente a elegir el protocolo de comunicación que usará el consumidor final (paso 6).

2. Si se indica que serán necesarios JARs externos para una correcta compilación del algoritmo, se pasará al punto 5.*

5*. Seleccionar JARs *En este paso el usuario seleccionar los JARs externos que sean necesarios para la compilación y uso del algoritmo. Se podrán seleccionar varios JARs. Una vez finalizado el proceso, el flujo convergerá en el paso 6.*

6.Elegir protocolo de comunicación: *Este es el paso en el que el usuario decide el protocolo de comunicación que usará el consumidor. El consumidor, como se ha detallado anteriormente, presenta dos modelos, uno para la comunicación mediante el protocolo AMQP y otro para el protocolo MQTT. Ambos modelos están adaptados para que su uso sea prácticamente idéntico.*

Estos son los pasos esenciales que el fabricante debe ofrecer para una creación de un consumidor de forma satisfactoria. En realidad, hay más pasos intermedios pero a la hora del diseño se ha hecho una abstracción evitando entrar en detalles más apegados a la ejecución real de la herramienta.

CAPÍTULO 4

Desarrollo

Este capítulo trata de ayudar al lector a entender cómo se ha desarrollado tanto el consumidor como el Fabricante*, explicando su estructura interna, las funcionalidades que presentan y cómo se han implementado.*

4.1. Implementación del *Fabricante*

El Fabricante *es una herramienta cuyo fin es facilitar la creación de aplicaciones que puedan hacer uso de protocolos para la recepción de mensajes y que puedan procesar los datos recibidos. Estas aplicaciones además presentan un comportamiento configurable en cada ejecución.*

Las próximas secciones describirán la implementación de la herramienta así como las funcionalidades que posee.

4.1.1. Especificación de la estructura del *Fabricante*

El Fabricante *es una aplicación que posee una interfaz gráfica de usuario (GUI - "Graphical User Interface") que se ha desarrollado usando Java Swing; y que está contenida en un archivo JAR autocontenido.*

Tal y como muestra la Figura 4.1, la parte gráfica del Fabricante *consiste en un Marco (o Frame) en cuyo interior se haya un panel (color blanco en la figura) denominado como bg que hace de fondo y sobre el que se superponen la imagen con el logotipo y dos paneles:*

- *Panel azul: Es un panel denominado content en el código y sobre él se irán alternando los diferentes paneles para interactuar con el usuario. El controlador es el encargado de cambiar los paneles de su interior y de extraer la información que ofrece el usuario en ellos.*
- *Panel amarillo: Panel denominado como menu en el código, posee varias etiquetas usadas para indicar el paso en el que se encuentra el usuario en el proceso de creación de un consumidor. También es modificado por el controlador.*

En la Figura 4.2 se puede observar como interactúan los diferentes componentes que participan en la interfaz gráfica.

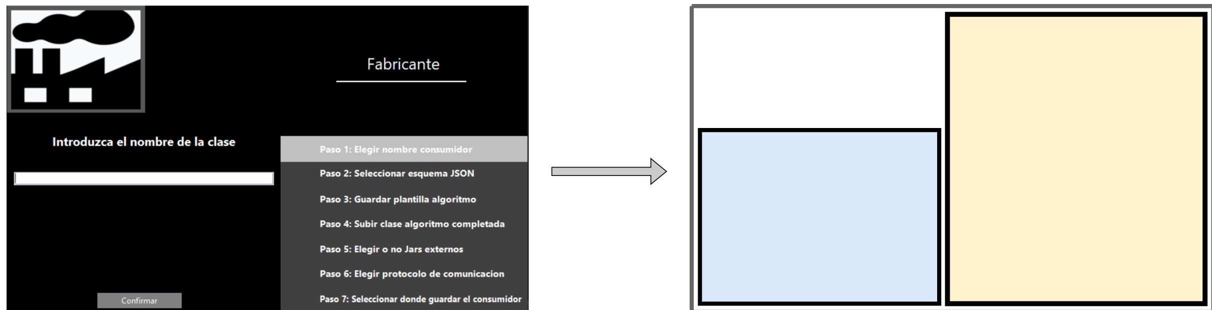

Figura 4.1: Estructura de la interfaz gráfica del *Fabricante*

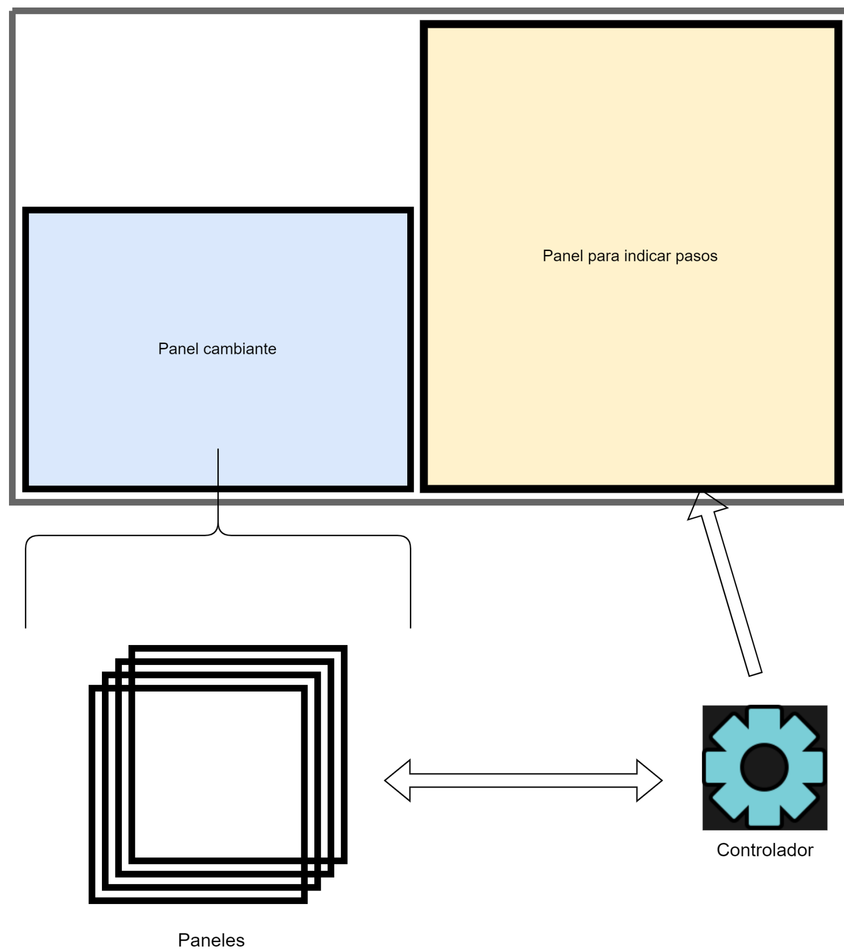

Figura 4.2: Funcionamiento de la interfaz gráfica

Finalmente, la Figura 4.3 muestra todas las clases Java que conforman el Fabricante. *La clase Controlador presenta el comportamiento del controlador, es decir que proporciona la implementación de todos los manejadores de eventos, la clase* **`Fabricante`** *se corresponde con la clase principal y posee tanto el marco (o frame) como el panel menu y el panel content. Y por último están todas las clases que comienzan por* Panel- *que son las que van cambiando dentro del panel content. La vista está conformada por la clase* **`Fabricante`** *y las clases* Panel-. *El modelo se encuentra en el controlador.*

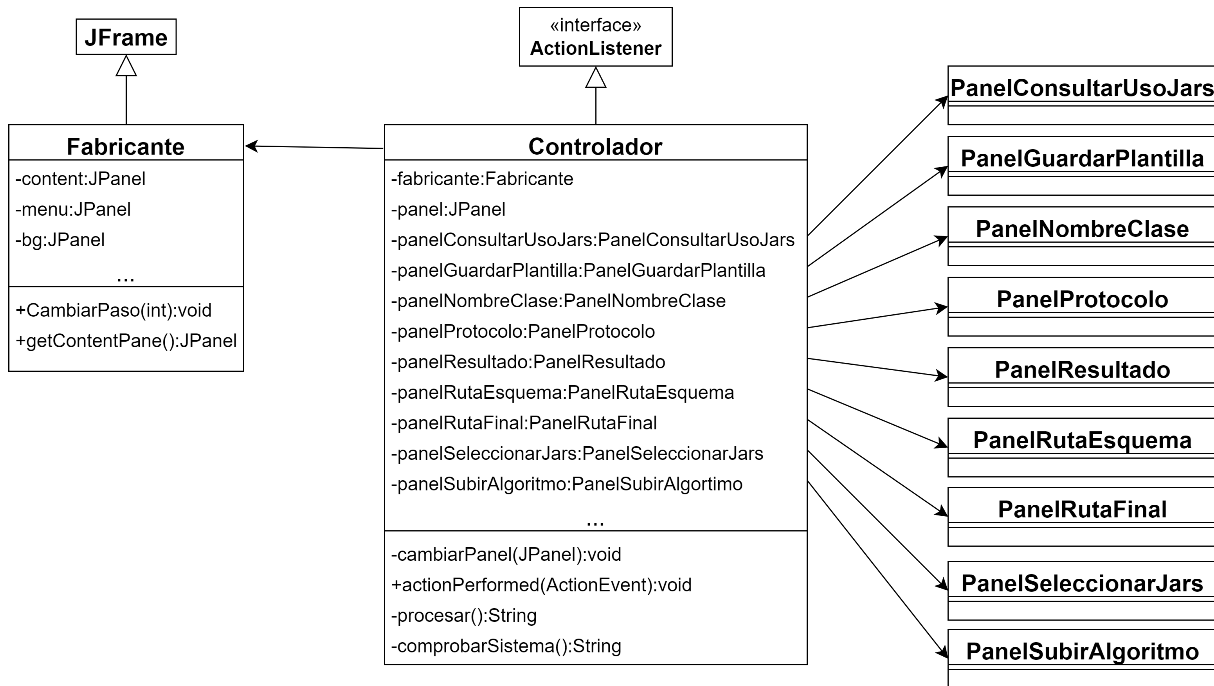

Figura 4.3: Diagrama de clases del *Fabricante*

4.1.2. Descripción de la funcionalidad del *Fabricante*

En esta sección vamos a explicar como se ha implementado la funcionalidad de la aplicación Fabricante, *explicando los diferentes pasos que sigue para generar un consumidor. Para facilitar la explicación, vamos a utilizar el diagrama de actividades que se muestra en la Figura 4.4. Como se puede observar en la figura, la generación de un consumidor consiste en 7 pasos:*

1.Introducir el nombre del consumidor: *En este paso el usuario introduce el nombre que recibirá la clase principal del consumidor, es decir, la clase que incluirá el código de gestión de las conexiones con el bróker de mensajería.*

El nombre que proporciona el usuario en este paso debe cumplir un par de reglas: 1) tiene que empezar en mayúsculas (tal y como los nombres de una clase de Java) y 2) el nombre no debe incluir la extensión .java. *Si el usuario introduce un nombre incorrecto se le notificará en la interfaz del error en cuestión y no le será posible avanzar al siguiente paso hasta que elija un nombre correcto.*

2.Describir los datos que recibirá el consumidor: *En el paso 2 el usuario seleccionará la ruta a un archivo que describirá como son los datos que el consumidor espera recibir. Este archivo será un esquema JSON como el que se puede apreciar en la Figura 4.5. Con a esta descripción se crea una clase de Java (o una principal y otras auxiliares) que se utilizarán para instanciar los datos que reciba el consumidor con la conexión establecida con el servidor RabbitmQ. Su nombre será formará a partir del nombre introducido en el paso 1 y el sufijo* Schema. *Por ejemplo, si en el paso 1 se introduce el nombre Iris para la clase principal, en este paso se generará una clase llamada IrisSchema.*

Para generar de forma automática la clase dado un JSON esquema se utiliza la librería

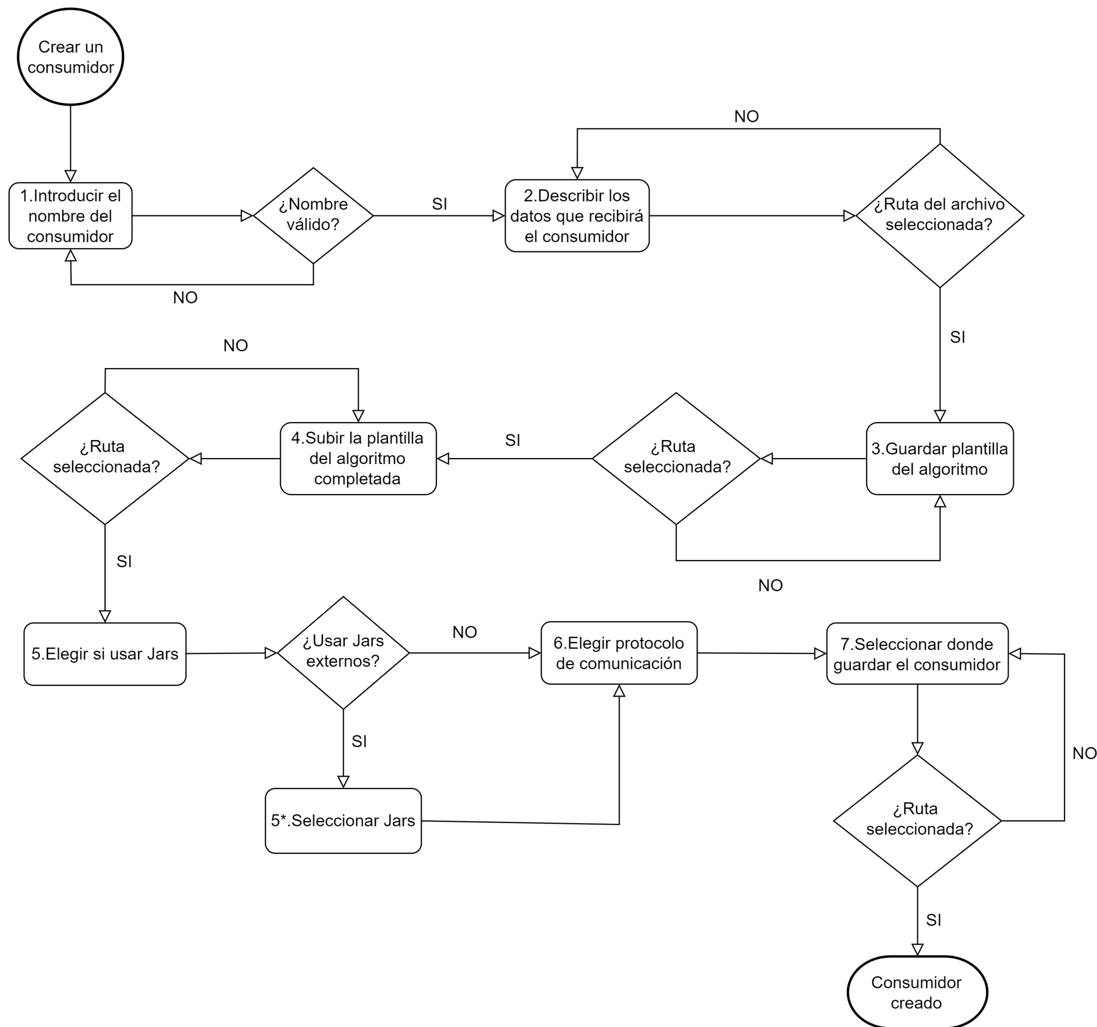

Figura 4.4: Diagrama de actividades del *Fabricante*

Jsonschema2pojo, *que usa un esquema JSON como referencia para crear una clase POJO ("Plain Old Java Objects") principal. Si se definen otras clases dentro de los atributos del esquema de la clase principal, se crearán otras clases auxiliares para ajustarse a la descripción.*

Como se muestra en el esquema de la Figura 4.5, los atributos presentes en la descripción han de tener el estilo "Lower case with underscores". Esto se debe ha una decisión de diseño ya que la librería GSON que deserializa los mensajes en formato JSON en instancias de la clase creada (proceso que se realiza en el consumidor) está configurada para que acepte el estilo "Lower case with underscores" en los campos del formato JSON. Si les llega algún mensaje con los campos con estilo "lowerCamelCase" los transformará al otro. Y es necesario que la clase POJO posea el mismo estilo que el que maneja GSON.

Por cada campo presente en la definición de propiedades "properties" de un objeto en el esquema JSON se añade a la clase Java un atributo privado y a sus métodos de acceso (getter y setter), cuyos nombres siguen el estilo "lowerCamelCase" y son de la forma getAtributo() y setAtributo() (también se aplica a las clases auxiliares que aparecen como atributos de la principal). La Figura 4.6 muestra la clase -Schema *generada para el esquema de la Figura 4.5.*

```
{
    "title": "Flor Iris",
    "description": "Conjuntos de datos de la flor Iris",
    "type": "object",
    "properties": {
        "sepal_length": {
            "type": "number"
        },
        "sepal_width": {
            "type": "number"
        },
        "petal_length": {
            "type": "number"
        },
        "petal_width": {
            "type": "number"
        },
        "species": {
            "type": "string"
        }
    }
}
```

Figura 4.5: Ejemplo de Esquema JSON

Si se ha seleccionado la ubicación del archivo en el buscador se procederá al paso 3. De lo contrario aparecerá un mensaje de error hasta que se seleccione una ruta.

3.Guardar la plantilla del algoritmo: *En este paso el usuario seleccionará con un buscador donde guardar un archivo java denominado algoritmo que tendrá que completar.*

La Figura 4.7 ilustra como es la plantilla que se le ofrece al usuario. La clase algoritmo implementa la interfaz `Runnable` *(por si es necesario que herede de otra clase para el correcto funcionamiento del algoritmo) y el usuario debe completar obligatoriamente el método* `run()` *adaptando su algoritmo para acceder a los datos instanciados en la clase* -Schema *usando los métodos de acceso detallados en el paso anterior. La información de los mensajes se guardan en la variable llamada datos y que consiste en un* `ArrayList` *de la clase* -Schema. *El usuario tiene la posibilidad de manipular el resto de la clase a voluntad (siempre y cuando no se modifique el contenido que viene por defecto), importando las librerías necesarias y declarando nuevas variables.*

Como ocurría en el paso anterior, aparecerá un mensaje de error si se intenta pasar al siguiente paso sin especificar una ruta.

4.Subir la plantilla del algoritmo completada: *Una vez que el usuario complete la plantilla de la clase del algoritmo, éste tendrá que indicar su ubicación en un buscador del* Fabricante. *La Figura 4.8 muestra un ejemplo de como se completaría la clase con los datos introducidos en pasos anteriores, este algoritmo calcula el porcentaje de flores de cada especie. Una vez más, aparecerá un mensaje de error si se intenta pasar al siguiente paso sin elegir la ruta.*

```java
public class IrisSchema {

    @JsonProperty("sepal_length")
    private Double sepalLength;
    @JsonProperty("sepal_width")
    private Double sepalWidth;

    //...

    @JsonProperty("sepal_length")
    public Double getSepalLength() {
        return sepalLength;
    }

    @JsonProperty("sepal_length")
    public void setSepalLength(Double sepalLength) {
        this.sepalLength = sepalLength;
    }

    @JsonProperty("sepal_width")
    public Double getSepalWidth() {
        return sepalWidth;
    }

    @JsonProperty("sepal_width")
    public void setSepalWidth(Double sepalWidth) {
        this.sepalWidth = sepalWidth;
    }

    //...
}
```

Figura 4.6: Ejemplo de clase -*Schema*

```java
   public class Algoritmo implements Runnable {
    ArrayList<IrisSchema> datos;

    public Algoritmo(ArrayList<IrisSchema> datos){
        this.datos = datos;
    }

    @Override
    public void run() {
     //A completar por el desarrollador
    }
}
```

Figura 4.7: Ejemplo de Plantilla del algoritmo

5.Elegir si usar JARs: *En este paso el usuario tiene que decidir si su algoritmo (la plantilla que ha subido en el paso anterior) necesita de alguna librería JAR externa, pueden ocurrir dos opciones:*

- *Si se pulsa en la opción de que el algoritmo no usa JARs externos, se procederá*

```java
public class Algoritmo implements Runnable {
    ArrayList<IrisSchema> datos;

    public Algoritmo(ArrayList<IrisSchema> datos){
        this.datos = datos;
    }

    @Override
    public void run() {
        float virginica = 0, versicolor = 0, setosa = 0,
              nulo = 0;
        for (IrisSchema flor : datos) {
            switch (flor.getSpecies()) {
            case "setosa":
                setosa++;
                break;
            case "versicolor":
                versicolor++;
                break;
            case "virginica":
                virginica++;
                break;
            default:
                nulo++;
            }
        }
        System.out.println("Virginica: " + virginica / datos.size()
           * 100 + "%");
        System.out.println("Versicolor: " + versicolor /
           datos.size() * 100 + "%");
        System.out.println("Setosa: " + setosa / datos.size() * 100
           + "%");
        System.out.println("Nulos: " + nulo / datos.size() * 100 +
           "%");
    }
}
```

Figura 4.8: Ejemplo de clase Algoritmo completado

directamente a elegir el protocolo de comunicación que usará el consumidor final (paso 6).

- *En otro caso, serán necesarios JARs externos para una correcta compilación del algoritmo y se pasará al paso 5*.*

5*. Seleccionar JARs *En este paso, se desplegará un panel donde el usuario podrá ir seleccionando las librerías en formato JAR. Los ficheros JAR seleccionados se mostrarán en la interfaz. También será posible cancelar la última selección y terminar el proceso de selección el cualquier momento. Una vez finalizado el proceso, el flujo convergerá en el paso 6.*

6.Elegir protocolo de comunicación: *Este es el paso en el que el usuario decide el protocolo de comunicación que usará el consumidor. El consumidor que se genera usa un*

protocolo en específico de comunicación específico pero el Fabricante *permite seleccionar cuál será. Las dos opciones disponibles son el protocolo AMQP y otro el protocolo MQTT.*

7.Seleccionar donde guardar el consumidor: *El último paso consiste en la selección de una ruta para guardar el consumidor creado. Como en pasos anteriores, si se intenta finalizar sin seleccionar una ruta aparecerá un mensaje de error.*

Tras unos segundos después de la selección aparecerá una ventana con un log que mostrará lo que ha sucedido durante el proceso de creación y compilación del consumidor, mostrando si ha habido algún error. De esta manera el usuario puede intentar subsanar los errores en la implementación en la plantilla del algoritmo, en el formato del JSON schema o en la selección de los JARs.

Para la creación del consumidor el Fabricante *hace uso de unas plantillas que consisten en cadenas de texto con todo el código necesario para crear la case principal del consumidor y que durante el proceso de uso del* Fabricante *son completadas, posteriormente transformadas en una clase de Java y finalmente compiladas junto a las otras clases también creadas en el tiempo de ejecución del* Fabricante*, como la clase del algoritmo o las clases usadas para instanciar los datos, con el fin de formar la verdadera entidad del consumidor.*

Durante el proceso de desarrollo de las plantillas se crearon diversas aplicaciones procesadoras de datos, estas aplicaciones eran una única clase de Java que contenían el algoritmo en su interior, sólo trataban con cadenas de texto simples que provenían de los mensajes que recibían del Publisher a través de RabbitMQ, y que usaban las librerías de AMQP client Y Eclipse Paho para crear y gestionar las conexiones con el servidor de RabbitMQ. A estas aplicaciones se les implementó las opciones configurables que serían trasladadas a las plantillas.

4.2. Implementación del Consumidor

Tal y como se ha descrito en el capítulo de diseño, los consumidores son aplicaciones que se generan utilizando unas plantillas que difieren según el protocolo de comunicación entre el consumidor y el bróker de mensajería (versión para el protocolo AMQP y MQTT). El uso de las plantillas permite construir aplicaciones capaces de procesar los datos de forma flexible.

En los próximos apartados se detallarán en que consiste realmente la estructura de un consumidor creado por un Fabricante *y como se ha desarrollado, desgranando sus funciones, las restricciones entre ellas y el flujo de su ejecución.*

4.2.1. Estructura del consumidor generado

El consumidor es una aplicación Java (sin GUI) que se ejecuta a través de línea de comando. En su invocación puede recibir argumentos para especificar su comportamiento. Como se puede apreciar en la Figura 4.9, el consumidor consta de las siguientes partes:

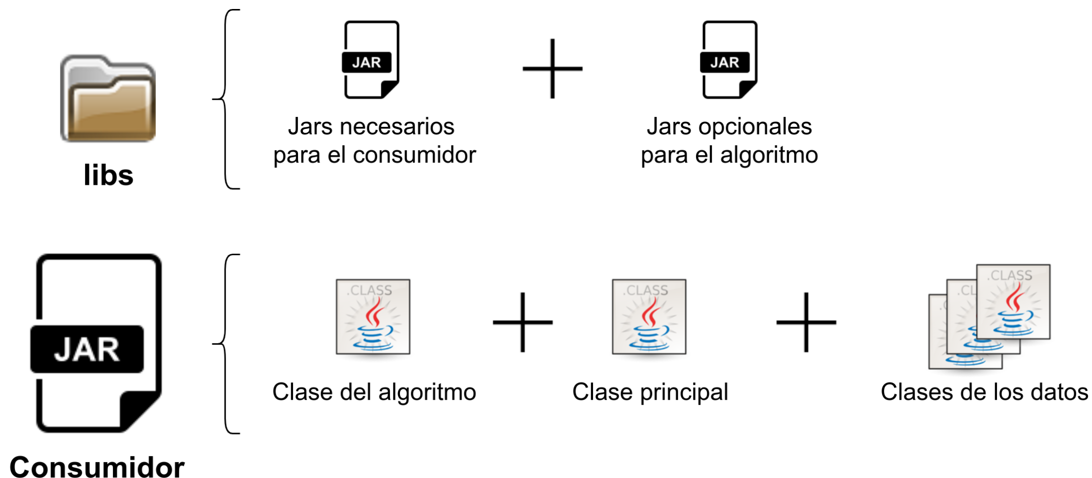

Figura 4.9: Estructura del consumidor

Archivo JAR Consumidor: *Este archivo es el eje central del consumidor. En su interior se encuentran archivos java compilados (extensión .class) que son esenciales para su correcto funcionamiento:*

- *La clase del algoritmo: Se trata de la clase donde el usuario introducirá el algoritmo que quiere que el consumidor ejecute.*
- *La clase principal: Esta es la clase que contiene toda la lógica detrás del consumidor configurable. Existen dos versiones de clientes consumidores, uno para AMQP, usando la librería de AMQP client; y otro para el protocolo MQTT, usando la librería Eclipse Paho. Además, posee métodos que procesan los argumentos que introduce un usuario.*
- *Las clase de los datos: Pueden ser una o varias clases dependiendo de como se hayan definido los datos.*

  Estas clases tienen como objetivo servir de instancia a los datos que le llegan al consumidor en formato JSON y que así puedan ser utilizados por el algoritmo.

  Su creación se realiza a partir de un esquema JSON (JSON-Schema) que introduce el usuario en el Fabricante. *Este esquema especifica la estructura de los datos que va a recibir el consumidor, y a partir de él se puede generar una clase Java que GSON puede utilizar para transformar (deserializar) el mensaje en formato JSON en una instancia de dicha clase.*

  La clase que siempre va a estar presente tiene el nombre del consumidor + Schema, por ejemplo, si el consumidor tiene el nombre "Iris" esta clase aparecerá nombrada como `IrisSchema`.

  Las clases opcionales tendrán el nombre de la estructura que representen en el Esquema, por ejemplo si hay una lista de coches, la clase se denominará `Coche` y será llamada desde la clase -Schema.

Directorio libs: *Este directorio contiene los JARS imprescindibles para el correcto funcionamiento del consumidor así como los JARS que el usuario elija en el* Fabricante *para el correcto funcionamiento del algoritmo.*

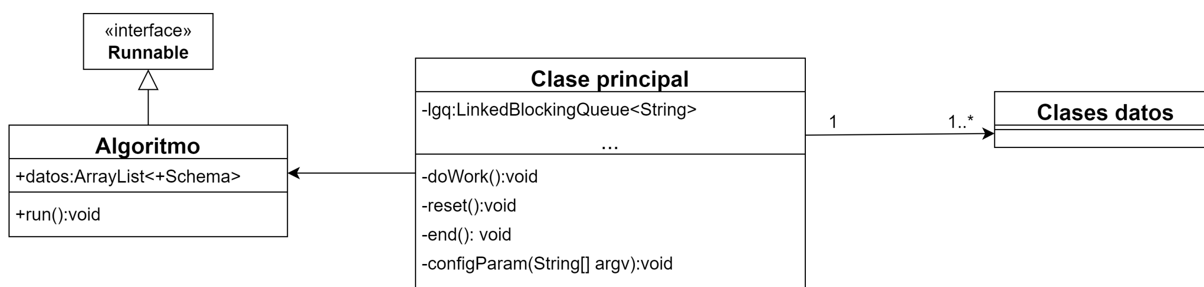

Figura 4.10: Diagrama de clases del Consumidor

4.2.2. Descripción de la funcionalidad del consumidor

En los siguientes apartado se detallará como se han implementado los diferentes módulos mencionados en el capítulo de diseño y que conforman al consumidor.

La Figura 4.10 muestra el diagrama de clases general de un consumidor. Todos los consumidores creados por el Fabricante *tienen una clase principal, la cual contiene toda la lógica tanto del módulo de interacción con el usuario como del módulo de gestión de las conexiones. También están tanto la clase* `Algoritmo` *como las clases necesarias para instanciar los datos (según la descripción del esquema JSON). El módulo de procesamiento de datos está también contenido en la clase principal en su mayoría aunque depende tanto de la clase* `Algoritmo` *como las clases que instancian los datos.*

Módulo de interacción con el usuario

Para realizar el módulo de interacción con el usuario se ha utilizado la herramienta Apache Commons CLI para interpretar los argumentos de línea de comandos. El consumidor al iniciarse pasa todos los argumentos a un método denominado `configParam` *que se encarga de interpretar las opciones y realizar las transformaciones necesarias antes de dar comienzo al resto del comportamiento de la aplicación. El método se encarga además de comprobar que están todos los argumentos obligatorios y que no hay incompatibilidad entre las opciones seleccionadas por el usuario. En caso de detectar alguna incompatibilidad, este módulo avisa al usuario mostrando un mensaje de error por consola.*

Módulo de gestión de las conexiones

Este módulo es el encargado de establecer y gestionar las conexiones con el bróker de mensajería. Este módulo varía según el protocolo que que utilice el consumidor (MQTT o AMQP). Para ambas versiones se ha desactivado la opción de persistencia de mensajes porque se ha decidido que los mensajes no permanezcan guardados de una conexión a otra.

Versión MQTT: *Como se ha explicado anteriormente, se ha utilizado la librería de Eclipse Paho para la construcción del módulo de gestión de conexiones en la versión de MQTT.*

Se ha implementado la versión 3 de MQTT porque no hay documentación suficiente sobre el uso de la librería bajo la versión 5.

Se ha usado la clase `MqttClient` *para la gestión de las conexiones. Para gestionar la llegada de mensajes el consumidor además implementa la interfaz* `MqttCallback`*, que notifica a la aplicación cuando un evento asíncrono relacionado con el cliente ocurre (como ya se mencionó en el capítulo de tecnologías, los dos clientes que ofrece la librería comparten algunos métodos asíncronos, entre ellos el que procesa la llegada de mensajes). Esta interfaz contiene tres métodos* `connectionLost`, `deliveryComplete` *y* `messageArrived` *que son llamados cuando ocurren dichos eventos asíncronos (cada uno está relacionado con un evento en particular) y cuyo comportamiento puede ser reescrito. En particular, se ha desarrollado el comportamiento del método* `messageArrived` *para que cuando sea invocado el módulo de procesamiento de datos se encargue de tratar los mensajes que lleguen.*

Para configurar las opciones de conexión se ha utilizado la clase `MqttConnectOptions` *de Eclipse Paho. En concreto se ha configurado para intentar una reconexión automática en caso de error en la conexión (con el método* `setAutomaticReconnect`*); iniciar siempre una sesión nueva usando el método (*`setCleanSession`*) y se ha desactivado el tiempo máximo que la aplicación va a esperar para establecer una conexión con el servidor (mediante el método* `setConnectionTimeout`*.*

Versión AMQP: *Para la gestión de conexiones usando el protocolo AMQP se ha usado la librería AMQP client.*

El modo de distribución de mensajes ha sido el denominado Default Exchange *que consiste en el intercambio directo pero sin un intercambiador definido. Esta característica hace que cada cola que se cree se vincule a un intercambiador por defecto (cadena vacía) y la binding key será la misma que el nombre de la cola. Con lo cual es una adaptación idéntica al patrón Publish/Subscribe, y permite que el comportamiento del consumidor sea igual usando ambos protocolos. Para configurarlo el módulo hace uso de métodos que proporciona AMQP client.*

Para la recepción de mensajes se ha utilizado una clase de la librería denominada `DeliverCallback` *y en ella se ha realizado toda la lógica de gestión del mensaje una vez que llega.*

Cinco de las opciones de configuración definidas en el capítulo de diseño pertenecen a este módulo, éstas son la Dirección del servidor RabbitMQ, el Nombre de la cola/topic, el Puerto de escucha del servidor RabbitMQ, el Nombre de usuario y la Contraseña.

Tanto la Dirección del servidor RabbitMQ como el Nombre de la cola/topic son Strings que se introducen en los métodos correspondientes para cada protocolo a la hora de definir la dirección del Host y para definir el topic en el caso de MQTT y el nombre de la cola en AMQP. Son parámetros obligatorios para ambas versiones del consumidor. El argumento de la opción para definir la dirección del servidor RabbitMQ (Host) es -ip (dirección del Host) o --host (dirección del Host) en su versión completa. Para introducir el topic en el consumidor MQTT es -t (nombre del topic) o --topic (nombre del topic); y para introducir el nombre de la cola en el consumidor AMQP es -q (nombre de la cola) o --queue (nombre de la cola).

Los parámetros del Puerto de escucha, el Nombre de usuario y la Contraseña son opcionales. El puerto de escucha del servidor RabbitMQ consiste en un número entero mayor que cero cuyo valor se introduce con el argumento -p (número del puerto) o --port (número

```
[
    {
        "algorithm": "MD5" ,
        "password": "asdqwen21nassdfw",
        "hash": "NGQ2NTRmNjcyZTQ1MWU0Mzk3MGNlMDI1Njg3ZGJhNWY="
    },
    {
        "algorithm": "MD5" ,
        "password": "sdfnwjzxfer",
        "hash": "NGQ2NTRmNjcyZTQ1MWU0Mzk3MGNlMDI1Njg3ZGJhNWY="
    }
]
```

Figura 4.11: Ejemplo de la estructura de los mensajes recibidos por el consumidor

del puerto). El nombre de usuario y la contraseña consisten en Strings que se introducen con los argumentos -u (nombre de usuario) o --username (nombre de usuario) y -pass (contraseña) o --password (contraseña) respectivamente.

Módulo de procesamiento de datos

Este módulo es el que realiza el procesado de los mensajes que recibe el consumidor del bróker, es decir, es el encargado de ejecutar el algoritmo de acuerdo a la configuración que ha seleccionado el usuario.

Para hacer segura las acciones sobre los mensajes recibidos los consumidores poseen una `LinkedBlockingQueue`*. Esta estructura implementa la interfaz de* `BlockingQueue` *[29] que contiene métodos* thread-safe *para evitar accesos concurrentes a los mensajes que provoquen incongruencias.*

Para invocar al algoritmo se ha creado un método denominado `doWork()` *cuyo funcionamiento es el siguiente:*

Primero se tratan los mensajes almacenados en la `LinkedBlockingQueue` *que están en formato JSON, la librería GSON se encarga de transformar la cadena JSON en una instancia de la clase que es generada en el* Fabricante*. Esta cadena de texto en formato JSON que recibe el consumidor tiene que presentar la estructura de JSON Array(estructura que soporta GSON para listas de objetos), en la Figura 4.11 se puede ver un ejemplo de como tiene que ser la estructura del JSON, empiezan y terminan por corchetes y cada objeto está encerrado entre llaves; cada objeto está separado de otro por una coma.*

Posteriormente se envía un `ArrayList` *con los datos transformados en instancias de su clase de Java (creada en el* Fabricante*) a una hebra que ejecuta el algoritmo (Como se ha dicho en la implementación del* Fabricante*, la clase* `Algoritmo` *implementa la interfaz* `Runnable`*).*

Las diferentes opciones definidas en el capítulo de diseño relacionadas con la configuración del comportamiento del consumidor a la hora de procesar los datos se han implementado de la siguiente manera:

```java
if (milisec > 0) {
    timerWaitTime = new Timer();
    TimerTask task = new TimerTask() {
        @Override
        public void run() {
            try {
                end();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
    timerWaitTime.schedule(task, milisec);
}
```

Figura 4.12: Implementación del tiempo de espera entre mensajes

Tiempo de espera entre mensajes: *Debido a que no existen métodos nativos en ninguna de las librerías para fijar un tiempo límite de espera entre mensajes se ha utilizado las clases* `Timer` *y* `TimerTask` *de Java para poder implementar la funcionalidad. Estas clases se usan para programar tareas que se realizan con un Thread en background. En concreto, en* `TimerTask` *se define la tarea y en* `Timer` *se programa cuando va a ejecutarse.*

Como se puede observar en el fragmento de código 4.12 se instancia un timer denominado "timerWaitTime" que se encarga de la gestión del tiempo de espera de mensaje, posteriormente se crea la tarea "task" de la clase `TimerTask` *y en ella se le introduce un método denominado* `end()`*, este método lo que hace es finalizar las conexiones, cancelar los timers que haya activos y comprobar si hay que realizar una última ejecución, en caso afirmativo llamaría al método* `doWork()`*. Por último se programa la ejecución de la tarea con el método* `schedule(tarea, tiempo en milisec)` *de la clase* `Timer`*.*

Para reiniciar el timer cuando llegan nuevos mensajes se ha creado un método denominado `reset()` *que cancela el timer y vuelve a programar otro con la misma tarea (No hay métodos disponibles en la clase* `Timer` *que permitan resetearlos).*

Como se puede apreciar, el tiempo introducido en los timers es en milisegundos, mientras que el usuario lo introduce en minutos; esto es debido a que se ha buscado facilitar la forma en que se configuran los parámetros para el usuario, por ello en el módulo de interacción con el usuario se hacen las conversiones necesarias.

Esta opción tiene como argumento -w (tiempo en minutos) o --wait (tiempo en minutos).

Intervalo de repetición temporal del algoritmo: *Esta opción se ha implementado de manera similar a la anterior, se usa un timer y se programa para que se ejecute la llamada al método* `doWork()` *dado un intervalo de repetición. La diferencia radica en el método usado de la clase* `Timer` *para programar la tarea,* `scheduleAtFixedRate(tarea, retraso de la primera ejecución en milisec, intervalo de repetición en milisec)` *permite la repetición de una tarea dado un retraso de tiempo para la primera ejecución y un intervalo de tiempo entre ejecuciones.*

El argumento de esta opción es -tr (tiempo en minutos) o --timerep (tiempo en minu-

tos).

Repetición del algoritmo por número de mensajes recibidos: *Opción de repetición del algoritmo que emplea por condición el número de mensajes recibidos. Consiste en una variable que contabiliza los mensajes recibidos y que llama al método* `doWork()` *cuando se alcanza la condición.*

El argumento de esta opción es -mr (número de mensajes) o --messagerep (número de mensajes).

Número máximo de mensajes: *Se trata de una variable que va contabilizando el número de mensajes recibidos y que cuando se alcanza el máximo establecido por el usuario llama al método* `end()`*.*

El argumento para esta opción es -max (número de mensajes) o --maxmessages (número de mensajes).

Desechar mensajes usados: *Es una variable booleana que borra todos los mensajes de la* `LinkedBlockingQueue` *si el usuario a especificado que se desechen los mensajes usados.*

Su argumento asociado es un flag: -d o --delused.

Ejecución final: *Es una variable booleana que se encuentra en el método* `end()` *y llama al método* `doWork()` *si el usuario ha especificado que se realice una ejecución final antes de acabar.*

El argumento que le corresponde es un flag: -fe o --finalexec.

Restricciones sobre las opciones de configuración

A la hora de desarrollar las opciones de configuración pertenecientes al módulo de procesamiento de datos se problemas de compatibilidad entre las diferentes configuraciones que un consumidor podía presentar, por lo que se determinaron e implementaron restricciones a la hora de usarlas.

Para desarrollar las restricciones primero se han clasificado todos las opciones disponibles. La Tabla 4.1 muestra el resultado de la clasificación, los grupos que intervienen en el procesamiento de los datos son el de condición de repetición del algoritmo, el de configuración de parada del consumidor, el de selección de mensajes y el de ejecución final del algoritmo.

Las Figuras que se han usado para realizar las restricciones son árboles de decisión que se leen de izquierda a derecha y que están divididos por niveles, cada nivel se corresponde con un grupo de las opciones que intervienen en el procesamiento de datos. Los nodos que aparecen entre paréntesis es lo que sucedería si no se introduce ningún argumento de ese grupo.

Grupo	Configuraciones pertenecientes
Configuración de conexiones	Dirección del servidor RabbitMQ, Nombre de la cola/topic, Puerto de escucha del servidor RabbitMQ, Nombre de usuario y Contraseña
Condición de repetición del algoritmo	Intervalo de repetición temporal y Repetición por número de mensajes
Condición de parada del consumidor	Tiempo de espera entre mensajes y Número máximo de mensajes
Selección de mensajes	Desechar mensajes usados
Ejecución final del algoritmo	Ejecución final

Tabla 4.1: Clasificación de las opciones de configuración

La Figura 4.13 muestra las opciones de configuración compatibles si no se usa la opción de solo seleccionar los mensajes no usados. Por defecto se usarían siempre todos los mensajes para la ejecución del algoritmo. Tras ello se presentan tres opciones en la condición de repetición del algoritmo:

1. Intervalo de repetición temporal que a su vez deriva en tres posibilidades para la condición de parada del consumidor: introducir la opción del Tiempo de espera entre mensajes(El valor del Tiempo de espera ha de ser mayor que el del Intervalo de repetición temporal para que tenga sentido que se ejecute el algoritmo por repetición), introducir el Número máximo de mensajes o que no termine (si no se introduce por argumento ninguna condición de parada). Cuando se cumple la condición de parada del Tiempo de espera entre mensajes o el Número máximo de mensajes puede dar lugar (o no) a una Ejecución final.

2. Repetición por número de mensajes, las posibilidades de configuración son idénticas al del Intervalo de repetición temporal, con la única diferencia que si se elige en la Condición de parada del consumidor el Número máximo de mensajes, el valor de mensajes que se esperan ha de ser mayor del número de mensajes necesarios para que se repita el algoritmo.

3. Puede darse la posibilidad de que no se indique una Condición de repetición del algoritmo y que por tanto el algoritmo solo se ejecute una vez, esta es una posibilidad especial ya que se tiene que indicar la Condición de parada (seleccionar alguno de los dos argumentos) y siempre se va a realizar una Ejecución final (la única vez que se va a ejecutar el algoritmo).

En el caso de la Figura 4.14 se muestran el árbol de posibilidades disponibles en el caso de que se use el argumento de solo utilizar los Mensajes no usados. La diferencia con el árbol anterior radica en que en este caso no puede haber no repetición ya que no tiene sentido usar el argumento de Mensajes no usados cuando solo se va a ejecutar una vez el algoritmo.

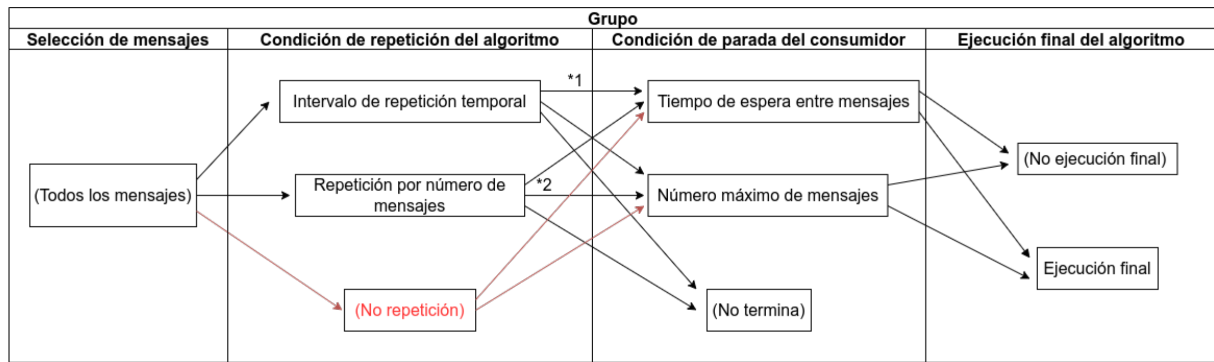

Figura 4.13: Restricciones si se usan todos los mensajes

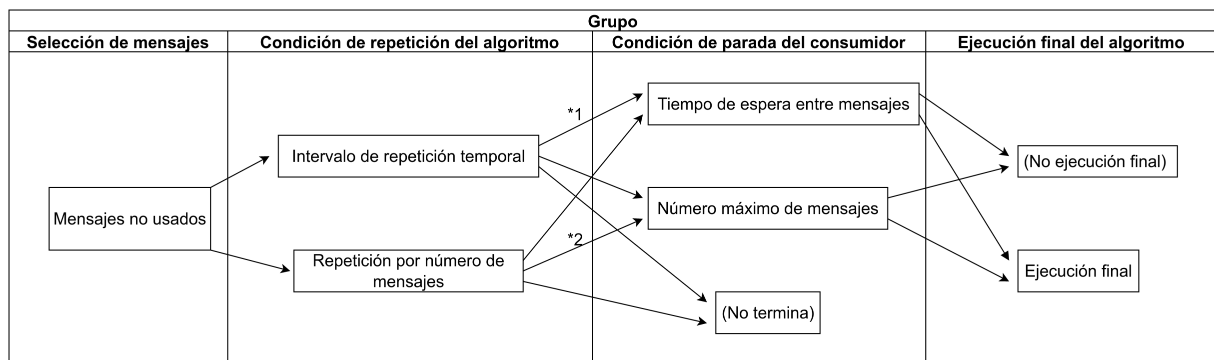

Figura 4.14: Restricciones si se desechan los mensajes procesados

Tras definir las restricciones que han de aplicarse a las configuraciones se puede visualizar como sería el curso de ejecución normal de un consumidor, como se puede observar en la figura 4.15 el usuario comienza introduciendo los parámetros al consumidor en el momento que ejecuta la instrucción para su invocación. Si los argumentos introducidos son los correctos, el consumidor inicia una conexión con el servidor RabbitMQ y procede a esperar mensajes. En este caso hay dos alternativas, que el consumidor esté configurado para no repetir el algoritmo y por el contrario, para que repita el algoritmo cuando se cumpla cierta condición:

1. Si no se repite el algoritmo, el consumidor almacenará los mensajes hasta que se cumpla la condición de parada, entonces cerrará las conexiones con el servidor, ejecutará el algoritmo con los mensajes acumulados y devolverá el resultado al usuario.

2. Si hay una condición de repetición, durante el transcurso de recepción de mensajes puede cumplirse dicha condición de repetición, en cuyo caso se ejecutará el algoritmo y el resultado se devolverá al usuario. Cuando se cumpla la condición de cese del consumidor, puede suceder el escenario en el que el consumidor esté configurado para una ejecución final, en este caso se ejecutará el algoritmo tras cerrar la conexión con el servidor RabbitMQ y después terminará la ejecución del consumidor.

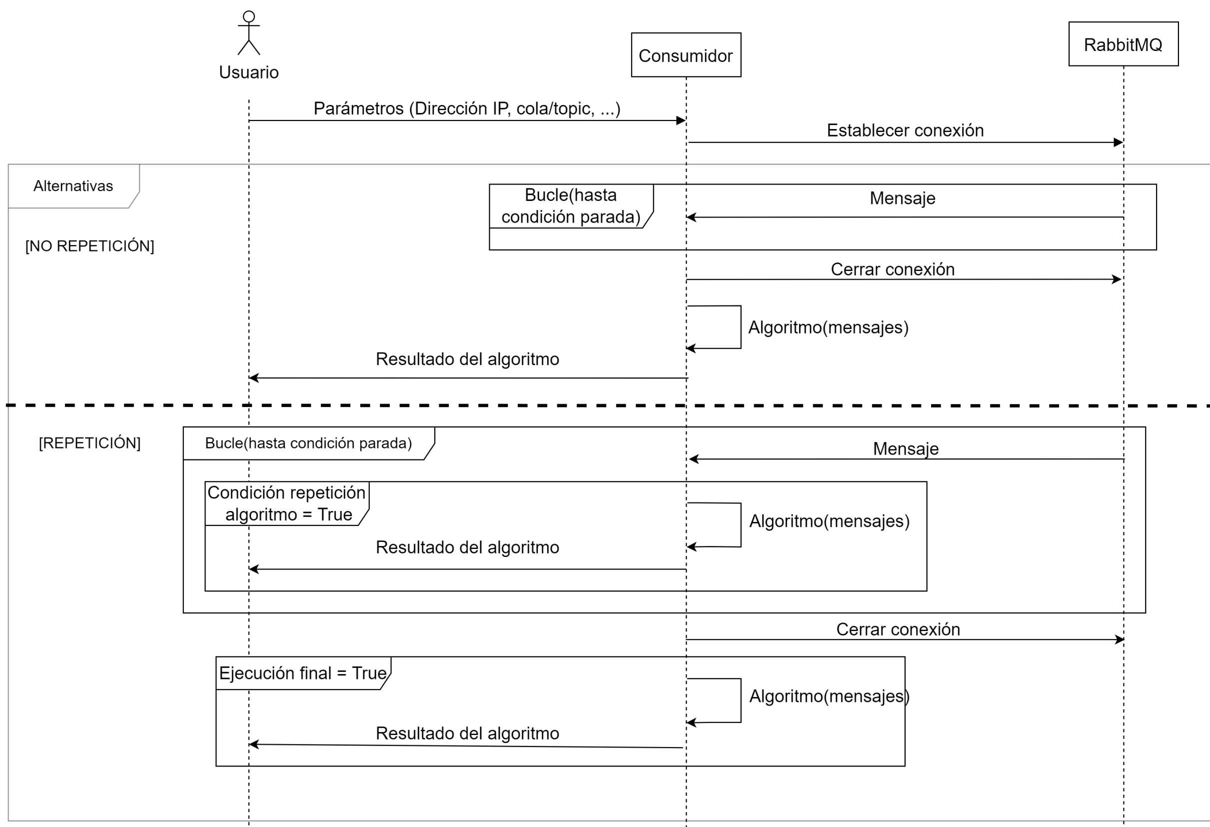

Figura 4.15: Diagrama de secuencia del consumidor

CAPÍTULO 5

Pruebas y casos de estudio

Este capítulo presenta las diferentes pruebas realizadas tanto al Fabricante *como a los consumidores que genera. El capítulo está dividido en dos secciones, una centrada en pruebas generales del funcionamiento; la segunda sección presenta varios casos de estudio realistas para resaltar la utilidad de nuestra propuesta.*

5.1. Pruebas Generales

Las pruebas generales permiten comprobar el correcto funcionamiento tanto del Fabricante *como de los consumidores, sin entrar en detalles de su aplicación en un caso de estudio concreto.*

Las pruebas realizadas sobre el fabricante están enfocadas en comprobar que la GUI permite una correcta interacción con el usuario, todas ellas se han realizado manualmente. Las pruebas más destacables han sido:

1. *Comprobar que si no se introduce ningún nombre para la clase del consumidor en el paso 1 el* Fabricante *no permite avanzar al paso 2: El resultado ha sido el esperado, el* Fabricante *devuelve un error y no permite avanzar hasta que se introduzca un nombre.*

2. *Comprobar que si se introduce un nombre incorrecto en el paso 1 el* Fabricante *mostrará un error y no será posible avanzar al siguiente paso hasta introducir un nombre válido: Prueba superada, el* Fabricante *notifica del error junto al motivo de porqué el nombre es inválido.*

3. *Intentar avanzar al siguiente paso sin elegir un archivo o ruta en los pasos donde sea necesario seleccionarlos: El comportamiento del* Fabricante *es el deseado, si se intenta avanzar sin seleccionar una ruta o archivo requerido muestra un mensaje de error y no permite avanzar.*

4. *Comprobar que se realiza correctamente el proceso de selección de JARs: Se ha comprobado que se seleccionan correctamente tanto uno como múltiples JARs en un mismo directorio, se ha comprobado que también se pueden seleccionar en diferentes usos del buscador y que el botón de cancelar la última selección funciona correctamente.*

5. *Comprobar que el panel del resultado muestra los errores correctamente: Se han realizado varias iteraciones sobre esta prueba introduciendo causas de error tanto en el algoritmo como no seleccionando JARs necesarios y en todas se han mostrado los errores que se han producido en el proceso de creación del consumidor.*

En el caso del consumidor, se han realizado pruebas centradas en comprobar si su funcionamiento es correcto dadas diferentes configuraciones. Las pruebas realizadas más reseñables son:

1. *Comprobar que si no se introducen los parámetros obligatorios, que son la dirección IP del servidor RabbitMQ y el nombre de la cola/topic, el consumidor devuelve un mensaje de error con los motivos y no continua su ejecución: Prueba superada, el consumidor tiene el comportamiento esperado.*

2. *Comprobar que no se pueden introducir por argumento configuraciones inválidas: Se han realizado diversas pruebas introduciendo argumentos incompatibles (ya explicados en el capítulo de desarrollo) y en todos los casos el consumidor devuelve un mensaje de error.*

3. *Comprobar que si se introducen valores negativos en argumentos como el tiempo de espera entre mensajes aparece un mensaje de error: La prueba se ha superado con éxito, el consumidor devuelve un mensaje de error cuando un valor introducido por argumento es negativo.*

4. *Comprobar que dada una dirección IP del servidor RabbitMQ, el consumidor puede conectarse correctamente: Se han introducido tanto una dirección a un servidor RabbitMQ que se ejecuta en la misma máquina donde se ha usado el consumidor (localhost) como una dirección IP a un servidor RabbitMQ remoto y en ambos casos ha logrado conectarse correctamente.*

5. *Comprobar que funcionan correctamente las conexiones con autenticación y con elección de puerto: Se ha comprobado satisfactoriamente que funcionan las opciones de introducción de usuario y contraseña, así como la de introducción de un puerto de escucha del servidor diferente al predeterminado. Para comprobarlo se han introducido los datos necesarios para conectar el consumidor con un servidor RabbitMQ remoto que escucha en un puerto específico y que exige credenciales para autenticar la conexión, el consumidor ha establecido la conexión exitosamente y ha ocurrido lo mismo con la recepción de datos y ejecución del algoritmo.*

6. *Configurar el consumidor para mantener una conexión prolongada con el servidor RabbitMQ: Se han hecho diversas pruebas con diferentes configuraciones que mantengan una conexión de larga duración o indefinida, en cualquier caso el consumidor mantiene la conexión sin problemas.*

7. *Comprobar que el consumidor es capaz de recibir y tratar mensajes con múltiples datos (array de JSON): Prueba superada, el consumidor los trata e introduce cada dato individualmente en una lista interna que le pasa al algoritmo.*

8. *Comprobar que el argumento de ayuda funciona correctamente: Se ha introducido el argumento (-h) que muestra una ayuda de uso del consumidor por pantalla. Funciona correctamente, el consumidor devuelve un mensaje de ayuda con las opciones disponibles y una breve descripción.*

5.2. Casos de estudio

Esta sección presenta 3 casos de estudio de diferentes dominios de aplicación. El objetivo es demostrar la utilidad en una aplicación consumidora creada por el Fabricante.*Para simplificar el desarrollo de creación de un entorno para las pruebas se ha emulado el envío de datos de prueba manualmente desde unos Publishers más genéricos ya que el Publisher no entra dentro de la propuesta de este trabajo, en cualquier caso el resultado sería el mismo dado que esto no influye en las prestaciones de un consumidor.*

En todas las pruebas realizadas el publisher manda diferentes mensajes con uno o varios datos en cada uno de ellos. El tiempo de envío exacto entre cada mensaje no se ha tenido en cuenta y ha variado según el objetivo de cada prueba.

5.2.1. Caso de estudio 1: Dataset Iris

El primer caso de uso trata del manejo de los datos del conocido dataset Iris [30], el conjunto de datos contiene muestras de cada una de tres especies de la la flor de Iris (Iris setosa, Iris virginica e Iris versicolor). Cada muestra presenta cuatro rasgos que permiten diferenciar a cada especie: el largo y ancho del sépalo y del pétalo. Existe un total de 150 muestras.

Como se ha presentado en los capítulos anteriores, para generar un consumidor es necesario proporcionarle al Fabricante *el formato de los mensajes que tiene que procesar el consumidor. La Figura 5.1 muestra el esquema JSON para este ejemplo. El esquema tiene cinco propiedades que incluyen los cuatros rasgos de la flor y la especie a la que pertenece.*

Además del formato de los datos, hay que proporcionarle al Fabricante *el algoritmo que va a procesar los mensajes. En este caso los datos se van a procesar con un algoritmo sencillo que calcula el porcentaje de Iris de cada especie que hay en el conjunto de datos. La Figura 5.2 muestra como se puede implementar el algoritmo en cuestión utilizando la plantilla de la clase Algoritmo.*

Una vez generado el consumidor se puede lanzar su ejecución con diferentes configuraciones. En este caso de estudio vamos a procesar todos los mensajes que se hayan recibido hasta el momento. Los resultados del algoritmo se muestran por pantalla, la Figura 5.3 es un ejemplo de como se visualiza el resultado de una ejecución del algoritmo.

Las pruebas realizadas sobre este caso de estudio tienen como objetivo comprobar que el funcionamiento del consumidor es correcto tanto en su versión MQTT como en su versión AMQP, y que el comportamiento de ellos es idéntico. Para ello se han creado con el Fabricante *dos consumidores, uno usa el protocolo MQTT y el otro el protocolo AMQP.*

Es interesante que siempre se mantengan todos los datos recibidos a la hora de ejecutar el algoritmo aunque éstos ya hayan sido procesador, por ello no se utilizará el argumento para desechar mensajes usados (-del) en ninguna configuración.

Algunas configuraciones han sido:

- *Repetición cada mensaje, sin ejecución final y con un máximo de cinco mensajes (-mr 1 -max 5).*

```
{
    "title": "Flor Iris",
    "description": "Conjuntos de datos de la flor Iris",
    "type": "object",
    "properties": {
        "sepal_length": {
            "type": "number"
        },
        "sepal_width": {
            "type": "number"
        },
        "petal_length": {
            "type": "number"
        },
        "petal_width": {
            "type": "number"
        },
        "species": {
            "type": "string"
        }
    }
}
```

Figura 5.1: Esquema JSON para el dataset Iris

- *Repetición cada dos minutos, con ejecución final y con una espera de mensajes de tres minutos (-tr 2 -w 3).*
- *Ejecución final sin repetición, con cinco mensajes máximos por condición de parada (-max 5).*
- *Ejecución final sin repetición, con tres minutos de espera máximos para recibir un mensaje por condición de parada (-w 3).*

5.2.2. Caso de estudio 2: Filogenética computacional

La filogenética es una rama de la biología evolutiva que trata de trazar la relación ancestro-descendiente de los organismos a diferentes niveles taxonómicos. La filogenética computacional consiste en la aplicación de algoritmos y programas computacionales a los análisis filogenéticos. El objetivo de ellos es ensamblar un árbol filogenético que represente una hipótesis sobre la ascendencia evolutiva de un conjunto de genes, especies u otros taxones(grupo de clasificación).

Este caso de estudio se enmarca en el campo de la filogenética computacional, el objetivo es generar un consumidor que procese que contienen árboles filogenéticos utilizando el algoritmo de Sankoff [31]. Dado un árbol cuyos nodos hoja incluyen secuencias de ADN, las cuales representan características de diferentes especies, este algoritmo calcula una secuencia de ADN que representa al ancestro común de todas esas especies que requiere el mínimo número de cambios en las secuencias de ADN de los nodos hoja.

En este caso, los mensajes que procesa el consumidor contienen un árbol filogenético en notación de Newick [32]. El esquema JSON usado para este caso es el que presenta la Figura 5.4, en el se ha definido que cada mensaje contendrá un identificador denominado

```
public class Algoritmo implements Runnable {
    ArrayList<IrisSchema> datos;

    public Algoritmo(ArrayList<IrisSchema> datos){
        this.datos = datos;
    }

    @Override
    public void run() {
        float virginica = 0, versicolor = 0, setosa = 0,
              nulo = 0;
        for (IrisSchema flor : datos) {
            switch (flor.getSpecies()) {
            case "setosa":
                setosa++;
                break;
            case "versicolor":
                versicolor++;
                break;
            case "virginica":
                virginica++;
                break;
            default:
                nulo++;
            }
        }
        System.out.println("Virginica: " + virginica / datos.size()
           * 100 + "%");
        System.out.println("Versicolor: " + versicolor /
           datos.size() * 100 + "%");
        System.out.println("Setosa: " + setosa / datos.size() * 100
           + "%");
        System.out.println("Nulos: " + nulo / datos.size() * 100 +
           "%");
    }
}
```

Figura 5.2: Clase del algoritmo para el caso de uso con el dataset Iris

```
Virginica: 21.73913%
Versicolor: 35.217392%
Setosa: 43.04348%
Nulos: 0.0%
```

Figura 5.3: Ejemplo de resultado del algoritmo para el dataset Iris

como treeId, otro campo será la longitud de las hojas de los árboles (los nodos hoja del árbol tendrán la misma longitud) y por último el árbol filogenético en notación Newick.

El algoritmo desarrollado hace uso de un JAR externo denominado como SmallParsimony*, el cual contiene los métodos necesarios para tratar con las cadenas filogenéticas y para realizar el algoritmo de Sankoff sobre ellas. Los datos resultantes de la ejecución se vuelcan en un fichero que crea en tiempo de ejecución. La Figura 5.5 muestra como se ve el resultado del algoritmo.*

```
{
    "title": "PylogeneticTree Schema",
    "description": "Phylogenetic Tree in Newick Notation",
    "type": "object",
    "properties": {
        "treeId": {
            "description":"Identifier of the Tree",
            "type": "integer"
        },
        "labelLength":{
            "description": "Length of the tree node labels",
            "type": "integer"
        },
        "newick": {
            "description": "PhylogeneticTree in Newick Notation",
            "type": "string"
        }
    }
}
```

Figura 5.4: Esquema JSON para la definición de los datos en el caso de estudio del algoritmo de Sankoff

```
Id: 1 original tree: ((CAAATCCC,ATTGCGAC),(CTGCGCTG,ATGGACGA));
Id: 1 Parsimony Score: 16
Id: 1 full-labeled Tree: ((CAAATCCC,ATTGCGAC)ATAGCCAC,
                          (CTGCGCTG,ATGGACGA)ATGGACGA)ATAGACAA;
```

Figura 5.5: Ejemplo de resultado del algoritmo de Sankoff

Para las pruebas realizadas sobre este caso de estudio se han creado dos consumidores con el Fabricante*, uno usa el protocolo MQTT y el otro el protocolo AMQP. La razón de tener un consumidor por cada protocolo para realizar las pruebas es para comprobar que el comportamiento es correcto e idéntico en ambos casos.*

Como este algoritmo no necesita reutilizar mensajes todas las pruebas tienen el argumento para desechar mensajes usados (-del).

Algunas configuraciones han sido:

- *Repetición cada mensaje, borrando los mensajes usados, con ejecución final y con un máximo de dos mensajes (-mr 1 -del -fe -max 2).*
- *Repetición cada dos minutos, borrando los mensajes usados, con ejecución final y con una espera de mensajes de tres minutos (-tr 2 -del -w 3).*
- *Repetición cada mensaje, borrando los mensajes usados (-mr 1 -del).*

5.2.3. Caso de estudio 3: Criptografía

El tercer caso de estudio hace un acercamiento al ámbito de la criptografía, disciplina que se encarga del cifrado y codificación de los datos para hacerlos inteligibles ante personas a las que no están destinados.

El algoritmo preparado para este caso contiene diversas implementaciones de algoritmos

```
{
    "title": "Caso de uso criptografia",
    "type": "object",
    "properties": {
        "algorithm": {
            "type": "string"
        },
        "password": {
            "type": "string"
        },
        "hash":{
            "type": "string"
        }
    }
}
```

Figura 5.6: Esquema JSON del algoritmo verificador de contraseñas

de clave asimétrica, en concreto son MD5, SHA-1, SHA-256, SHA-512 y BCryipt. El este caso los mensajes incluirán una contraseña, un hash que supuestamente pertenece a esa contraseña y el tipo de algoritmo de cifrado que se ha usado para obtenerlo. El objetivo es que por cada mensaje recibido, el consumidor ejecutará un algoritmo que comprobará si realmente el hash se corresponde con la contraseña dado el algoritmo de cifrado y volcará los resultados en un fichero de texto.

El algoritmo del consumidor hace uso de dos JARS externos, uno de ellos se denomina apache-commons-codec *y contiene los métodos para cifrar con MD5 y los diferentes SHA, además de poder codificar en base64; el otro JAR se denomina jbcrypt y posee diferentes métodos para hacer uso del algoritmo de cifrado lento BCrypt.*

El esquema JSON que describe como serán los mensajes es el que puede observarse en la Figura 5.6. En la Figura 5.7 puede apreciarse un ejemplo de como es la salida del algoritmo.

Se han realizado pruebas con distintas configuraciones para los dos consumidores (el que utiliza protocolo MQTP y el que utiliza AMQP). En este caso de estudio no es necesario reutilizar mensajes ya procesados, así que en todas las pruebas se ha utilizado el argumento para desechar mensajes usados (-del). Algunas configuraciones han sido:

- *Repetición del algoritmo cada dos mensajes, borrando los mensajes ya procesados, con ejecución final y con un máximo de tres mensajes (-mr 2 -del -fe -max 3).*
- *Repetición del algoritmo cada minuto, borrando los mensajes ya procesado y con una espera de mensajes de dos minutos (-tr 1 -del -w 2).*
- *Repetición del algoritmo cada dos mensajes, borrando los mensajes usados (-mr 2 -del).*

```
Algoritmo SHA-512:
Constraseña: osF6O9+AS3
Hash: YTFjZWNmNDVlZTM4NjJkNzRkY2MyMGM4ZWJjNTE1NDIxNzU1Njg5YzcwM2IyM
TI4MzgzYmI5ZTU4ZTcyOWRhNzkwNWRmNTRkNjIxNjFlOWFkNGM3MmMzNjdkNzc0ZGRj
ODMzZTJkM2JlNTU1ZWU1ZGZlNjZjY2E2MTM0NDIwYWY=
OK
------------------------------------------------
Algoritmo Bcrypt:
Constraseña: 9tH72lsd12
Hash: $2a$10$zGdkOng6F2ZEC39Uu7r6y.ctUePfYH/2OAybsMqzI2W1tFktJ9MTy
OK
------------------------------------------------
Algoritmo MD5:
Constraseña: 8DSx35fed12
Hash: $2a$10$FL3H/l4a.xyisICUpfT1.utH5wU5owORs.0OK8OfDA14XhxmhoTam
hash no pertenece a la contraseña
------------------------------------------------
```

Figura 5.7: Ejemplo de resultado del algoritmo verificador de contraseñas

CAPÍTULO 6

Conclusiones y líneas futuras

En este proyecto se ha podido demostrar que es posible facilitar a los programadores la tarea de realizar aplicaciones que procesen datos recibidos a través de un bróker de mensajería.

Para ello se ha diseñado e implementado una aplicación Fabricante *que facilita la generación de aplicaciones consumidoras configurables. El* Fabricante *permite que los desarrolladores no tengan que preocuparse por la lógica detrás de la comunicación con el bróker de mensajería (como los protocolos de comunicación o la recepción de mensajes). Por lo que genera, esta aplicación* Fabricante *es una oportunidad para muchos usuarios que no están familiarizados con los bróker de mensajería pero que desean desarrollar sus aplicaciones centrándose únicamente en el procesado de los datos.*

Por otro lado, el diseño e implementación para la creación de consumidores configurables dentro del Fabricante *hace que la abstracción vaya más allá, pues el usuario tampoco tiene que preocuparse en programar cuando debe de ejecutarse su algoritmo.*

En definitiva, este trabajo ha servido para explorar la forma de ayudar en el desarrollo de aplicaciones pertenecientes en el campo de IoT. Una tarea que aún está en sus primeras etapas y que ahora puede abrir una veda de cara a futuras herramientas.

6.1. Satisfacción de requisitos

En esta sección revisamos los requisitos funcionales y no funcionales que identificamos en el Capítulo 3 y determinamos el grado de satisfacción de los mismos en la aplicación Fabricante *y en las aplicaciones consumidor que se generan. Para ello vamos a utilizar unas tablas similares a las utilizadas en el análisis de requisitos de la Sección 3.2 Cada tabla mostrará el identificador del requisito, el estado en el que se encuentra: Logrado (L), Parcialmente Logrado (P), No Logrado (N). Y finalmente una breve explicación de los motivos por los cuales está en ese estado.*

***Estado de los requisitos del* Fabricante**
Tal y como se muestra en la Tabla 6.1, la aplicación Fabricante *satisface todos los requisitos funcionales que se establecieron en la fase de diseño. En cuanto a los requisitos no funcionales del* Fabricante*, todos se han logrado aunque, como se observa en la Ta-*

bla 6.2, el requisito RNF-4, relativo a los sistemas operativos en los que podía ejecutarse la aplicación, solo se ha conseguido parcialmente. Este requisito tenía una prioridad baja (prioridad C, Could *dentro de la escala MSCoW), por tanto la aplicación* Fabricante *cumple con todos los requisitos más prioritarios.*

Estado de los requisitos del consumidor
Como puede verse en la Tabla 6.3, hemos conseguido satisfacer todos los requisitos funcionales de las aplicaciones consumidor. En el caso de los requisitos no funcionales, que se muestran en la Tabla 6.4, no ha sido posible satisfacer uno de ellos, en concreto el relacionado con generar una aplicación consumidor contenida en un único fichero JAR. Aunque su prioridad era muy baja (prioridad W, Won't *dentro de la escala MSCoW) y no se tenía contemplado su cumplimiento desde un principio.*

6.2. Trabajos Futuros

Tras la realización del proyecto se han identificado diferentes líneas de trabajo futuro que permitirían extender el alcance los de los resultados obtenidos.

Adaptar las funcionalidades del* Fabricante *a las propiedades Featured model

Id	Prioridad	Estado	Explicación
RF-1	M	L	El *Fabricante* recoge los datos necesarios para construir un consumidor en siete pasos
RF-2	M	L	Esta acción es realizada en el primer paso del *Fabricante*
RF-3	M	L	El *Fabricante* le ofrece una plantilla al usuario que tendrá que completar y luego devolver al *Fabricante*
RF-4	M	L	La descripción la realiza el usuario usando un Esquema JSON que entrega al *Fabricante*
RF-5	M	L	El *Fabricante* ofrece la posibilidad de elegir el protocolo del consumidor en el sexto paso

Tabla 6.1: Requisitos funcionales del *Fabricante* logrados

Id	Prioridad	Estado	Explicación
RNF-1	S	L	La interfaz gráfica desarrollada usando Java Swing facilita el uso de la herramienta dividiendo los pasos en diferentes paneles para enfocar la tarea del usuario.
RNF-2	S	L	Como es una herramienta desarrollada en Eclipse, el entorno de desarrollo posee una herramienta para crear el JAR con dependencias auto-contenidas.
RNF-3	S	L	El manual de encuentra en el anexo
RNF-4	C	P	Se ha adaptado para ser usado en sistemas operativos como Windows 10 y Ubuntu (distribución de Linux). No ha sido posible conseguir adaptarlo a sistemas macOS

Tabla 6.2: Requisitos no funcionales del *Fabricante* logrados

Id	Prioridad	Estado	Explicación
RF-1	M	L	El usuario puede introducirla mediante el parámetro -ip (dirección IP del Host)
RF-2	M	L	Existe un parámetro para introducir el topic -t (nombre del topic) en el caso de MQTT y para introducir la cola -q (nombre de la cola)
RF-3	M	L	El consumidor tiene un comportamiento configurable en cada invocación
RF-4	M	L	El consumidor inicia las conexiones con el servidor RabbitMQ, espera recibir mensajes y puede configurarse de diversas maneras para ejecutar el algoritmo
RF-5	C	L	El consumidor avisa al usuario de los parámetros incorrectos
RF-6	S	L	El consumidor tiene una opción para mostrar un mensaje de ayuda por pantalla, aparece usando el argumento -h
RF-7	S	L	El consumidor puede reconfigurarse por parámetros en cada ejecución

Tabla 6.3: Grado de satisfacción de los requisitos funcionales del consumidor

Id	Prioridad	Estado	Explicación
RNF-1	S	L	El usuario solo tiene que introducir por argumentos las opciones de comportamiento y además posee un sistema de ayuda en caso de error humano a la hora de introducir parámetros
RNF-2	W	N	El consumidor es un archivo JAR pero no ha sido posible contener las dependencias en su interior debido a problemas a la hora de cargar los archivos. Para que fuera auto-contenido tendría que modificarse la forma en que se cargan los archivos en el momento de compilación, tarea que no es trivial. Para más información consultar [33]
RNF-3	S	L	El manual se encuentra en un anexo
RNF-4	M	L	El consumidor está escrito exclusivamente en Java y no hay ninguna instrucción dependiente del sistema operativo.

Tabla 6.4: Grado de satisfacción de los requisitos no funcionales del consumidor

El software orientado a características [34] (o Featured-Oriented Sowftware*) es aquel que dadas todas sus funcionalidades que presenta, tiene que acomodarse a las necesidades de cada consumidor, descartando aquellas que el usuario no desee usar. Un ejemplo para abstraer el concepto sería las diferentes opciones que se pueden elegir en un coche, el modelo, el color, la potencia, la eficiencia, el tipo de combustible.*

Este proyecto se asemeja a las característica de una línea de producto software porque es configurable por el usuario, podría darse un paso más allá y adaptarse completamente al paradigma featured model*, ofreciendo una herramienta para configurar directamente*

las piezas de software que quieren integrarse en el consumidor final.

Explorar las posibilidades de un publisher flexible

A pesar de que en las primeras fases del proyecto se estudió la posibilidad de incluir un publisher de configuración flexible, la idea fue descartada rápidamente debido a que los dispositivos que envían datos presentan una gran diversidad de lenguajes de programación, siendo en muchos casos sensores o aparatos desarrollados con lenguajes propietarios.

Una posible rama a explorar sería la búsqueda de soluciones para diseñar e implementar una herramienta para generar aplicaciones Publisher *configurables.*

Consumidor con dependencias auto-contenidas

Como se detalló en la sección de requisitos logrados, no ha sido posible la creación en tiempo de ejecución de un consumidor con dependencias contenidas en su interior, esto es debido a como está desarrollado la forma en que el classpath de los archivos JAR encuentran las dependencias.

Como futura línea de trabajo podría desarrollarse un código que permitiera encontrar las dependencias necesarias del archivo JAR en su interior, sin necesidad de utilizar herramientas externas para su creación.

Desarrollar el proyecto en otro lenguaje de programación

Debido a que Java es un lenguaje que necesita compilación ha habido ciertas limitaciones técnicas que se han tenido que lidiar en el proyecto. Sería interesante diseñar e implementar una aplicación Fabricante *que genere consumidores en otros lenguajes, por ejemplo en Python, que además es un lenguaje muy utilizado para el procesado de datos.*

Dentro de esta línea de trabajo futuro, también se podría cambiar el diseño de la aplicación Fabricante*, ofreciéndose como un servidor web que dada unas peticiones con especificaciones devolviera el consumidor.*

Estudiar la interoperabilidad entre brókers y consumidores

Aunque se ha indagado un poco sobre la interoperabilidad entre aplicaciones consumidoras y los diferentes brókers de mensajería, no era uno de los objetivos del proyecto. Sería muy interesante estudiar más a fondo los problemas de conexión entre diferentes implementaciones de librerías y brókers que soportan los mismos protocolos.

Bibliografía

1. *RabbitMQ*, https://www.rabbitmq.com/, Último acceso: 2022-04-19.
2. *Incremental Model*, https://www.javatpoint.com/software-engineering-incremental-model, Último acceso: 2022-04-19.
3. *¿Qué es LaTeX?*, http://desarrolloweb.dlsi.ua.es/cursos/2015/herramientas-investigacion/que-es-latex, Último acceso: 2022-04-23.
4. *Overleaf*, https://en.wikipedia.org/wiki/Overleaf, Último acceso: 2022-04-23.
5. *Java Programming Language*, https://docs.oracle.com/javase/7/docs/technotes/guides/language/, Último acceso: 2022-04-23.
6. *What is Maven?*, https://maven.apache.org/what-is-maven.html, Último acceso: 2022-04-26.
7. *MVNRepository*, https://mvnrepository.com/, Último acceso: 2022-04-26.
8. *Json*, https://www.json.org/json-es.html, Último acceso: 2022-04-26.
9. *The JSON data interchange syntax*, https://www.ecma-international.org/publications-and-standards/standards/ecma-404/, Último acceso: 2022-04-26.
10. *JSON Schema*, https://json-schema.org/, Último acceso: 2022-04-28.
11. *gson*, https://github.com/google/gson, Último acceso: 2022-04-26.
12. *jsonschema2pojo*, https://www.jsonschema2pojo.org/, Último acceso: 2022-04-28.
13. *Eclipse/IDE*, https://www.eclipse.org/ide/, Último acceso: 2022-04-26.
14. *Bróker de Mensajería*, https://blog.bi-geek.com/broker-mensajeria/, Último acceso: 2022-04-28.
15. *¿Qué es Apache Kafka?*, https://www.redhat.com/es/topics/integration/what-is-apache-kafka, Último acceso: 2022-04-28.
16. *Eclipse Mosquitto*, https://mosquitto.org/, Último acceso: 2022-04-28.
17. *Protocolos de comunicación para IOT*, https://www.luisllamas.es/protocolos-de-comunicacion-para-iot/, Último acceso: 2022-04-28.
18. *MQTT Version 3.1.1*, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, Último acceso: 2022-05-14.
19. *MQTT Version 5.0*, http://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html, Último acceso: 2022-05-14.
20. *Beginners Guide To The MQTT Protocol*, http://www.steves-internet-guide.com/mqtt/, Último acceso: 2022-05-14.

21. *MQTT: The Standard for IoT Messaging*, https://mqtt.org/, Último acceso: 2022-05-14.

22. *OASIS Advanced Message Queuing Protocol (AMQP)*, https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html, Último acceso: 2022-05-14.

23. *What is AMQP and why is it used in RabbitMQ?*, https://www.cloudamqp.com/blog/what-is-amqp-and-why-is-it-used-in-rabbitmq.html, Último acceso: 2022-05-14.

24. *https://www.rabbitmq.com/getstarted.html*, https://www.rabbitmq.com/tutorials/tutorial-three-java.html, Último acceso: 2022-05-15.

25. *Interoperability*, https://www.rabbitmq.com/interoperability.html, Último acceso: 2022-05-15.

26. *Eclipse Paho Java Client*, https://github.com/eclipse/paho.mqtt.java, Último acceso: 2022-05-15.

27. *RabbitMQ Java Client Library*, https://www.rabbitmq.com/java-client.html, Último acceso: 2022-05-15.

28. *MVC (Model, View, Controller) Explicado*, https://codigofacilito.com/articulos/mvc-model-view-controller-explicado, Último acceso: 2022-05-20.

29. *BlockingQueue Interface in Java*, https://www.geeksforgeeks.org/blockingqueue-interface-in-java/, Último acceso: 2022-06-1.

30. *Iris Data Set*, https://archive.ics.uci.edu/ml/datasets/iris, Último acceso: 2022-06-16.

31. *Filogenética computacional*, https://hmong.es/wiki/Computational_phylogenetic, Último acceso: 2022-06-5.

32. *The Newick tree format*, https://evolution.genetics.washington.edu/phylip/newicktree.html, Último acceso: 2022-06-5.

33. *Adding Classes to the JAR File's Classpath*, https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html, Último acceso: 2022-06-1.

34. S. Apel, D. Batory, C. Kästner y G. Saake, *Feature-Oriented Software Product Lines Concepts and Implementation* (Springer, 2013), ISBN: 9783642375200.

35. *Overview of JDK Installation*, https://docs.oracle.com/en/java/javase/18/install/overview-jdk-installation.html, Último acceso: 2022-06-8.

36. *Get Java for desktop applications*, https://www.java.com/en/, Último acceso: 2022-06-8.

Apéndices

APÉNDICE A

Manual del fabricante

El Fabricante *es una herramienta que facilita el desarrollo de aplicaciones que procesen datos recibidos al conectarse a una arquitectura IoT con un servidor RabbitMQ como bróker de mensajería. Las aplicaciones producidas por la herramienta, denominadas* Consumidor*, presentan una característica particular, su comportamiento es configurable en cada ejecución . Para más información consultar el manual correspondiente B.*

A.1. Instalación

El Fabricante *es compatible con Windows y las distintas distribuciones de Linux. Para utilizar el* Fabricante *es necesario que en la máquina este instalado el Java Development Kit (JDK). Los pasos para instalar el JDK puede encontrarse en la página oficial de oracle [35].*

A.2. Funcionamiento

Para ejecutar la herramienta solo es necesario hacer doble click sobre su icono en el caso de Windows. Una opción alternativa es introducir el comando java -jar Fabricante.jar en una ventana de comandos.

Los siguientes apartados contienen explicaciones acerca de como hacer un uso correcto de la herramienta en cada paso.

A.2.1. Paso 1: Elegir el nombre del consumidor

En este paso ha de introducirse el nombre usarán tanto el consumidor (archivo JAR) como la clase interna que gestiona las conexiones con el bróker de mensajería.

El nombre que proporciona el usuario en este paso debe cumplir dos reglas: 1) tiene que comenzar en mayúsculas (tal y como los nombres de una clase de Java) y 2) el nombre no debe incluir la extensión .java.

Figura A.1: Mensajes de error paso 1 del *Fabricante*

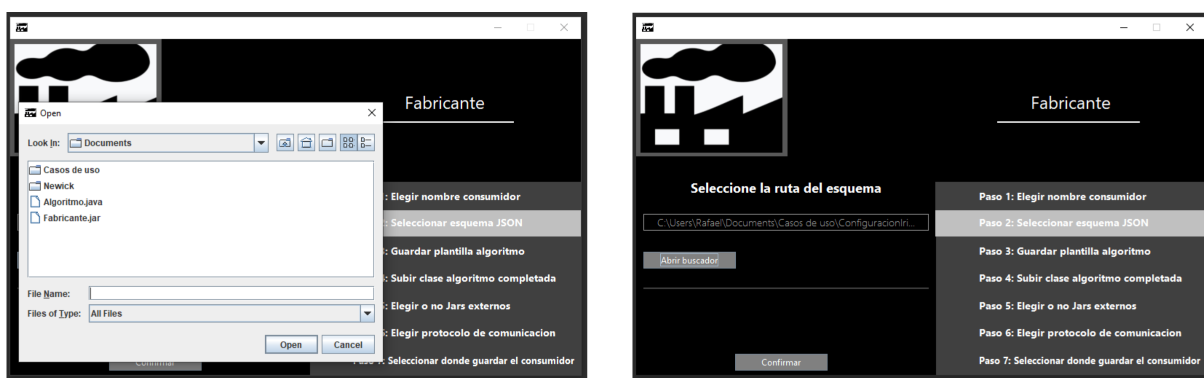

Figura A.2: Paso 2 del *Fabricante*

```
1  {   "title": "Ejemplo de esquema para una clase Ciudad que contiene
       una lista de Casas",
2      "type" : "object",
3      "properties" : {
4          "Casa" : {
5              "type" : "array",
6                  "items" : {
7                      "type" : "object",
8                      "properties" :{
9                          "puerta": {
10                             "type" : "string"
11                         }
12                     }
13                 }
14         }
15     }
16 }
```

Figura A.3: Ejemplo de Esquema JSON

Como se observa en la imagen A.1, si se introduce un nombre incorrecto aparecerá un mensaje en la interfaz informando del motivo del error, no será posible ir al siguiente paso hasta introducir un nombre correcto.

A.2.2. Paso 2: Seleccionar el esquema JSON

En la Figura A.2 se muestra el proceso de selección de la ruta a un archivo .json que describe como son los datos que el consumidor espera recibir. El archivo contendrá un esquema JSON como el que se puede apreciar en la Figura A.3.

Con la descripción el Fabricante *creará una clase de Java (cuyo nombre será el nombre introducido en el paso 1 +* Schema. *Por ejemplo, si se elige como nombre Iris la otra clase se nombrará como IrisSchema.) capaz de instanciar los mensajes recibidos por el* Fabricante *al conectarse a un servidor RabbitMQ. Junto a esta clase Java pueden generarse clases auxiliares para ajustarse a la descripción del esquema JSON, como es el caso de las listas de objetos. La Figura A.3 muestra un ejemplo de varias clases generadas, tomando como referencia el esquema JSON A.3 que se trata de la descripción de una clase Ciudad con una lista de casas, se han generado dos clases, la clase Ciudad y la Clase Casa.*

Como se muestra en el esquema de ejemplo (Figura A.4) los atributos presentes en la descripción han de tener el estilo "Lower case with underscores".

Por cada campo presente en la definición de propiedades "properties" de un objeto en el esquema JSON se añade un atributo privado a la clase Java resultante, junto a sus métodos de acceso (getter y setter) que siempre se crean siguiendo el estilo "lowerCamelCase" y son de la forma getAtributo() y setAtributo().

Si se ha seleccionado la ubicación del archivo en el buscador se procederá al paso 3. Aparecerá un mensaje de error si no se selecciona nada y se intenta ir al paso siguiente.

A.2.3. Paso 3: Guardar la plantilla del algoritmo

Como se ve en la Figura A.5, en este paso se selecciona con un buscador donde guardar un archivo Java denominado algoritmo cuyo método denominado "run()" ha de ser completado. Como se aprecia en la Figura A.6 los datos vienen dentro de una ArrayList *de la clase generada a partir del esquema JSON. Se puede acceder a los atributos que contienen cada elemento de la lista usando los métodos de acceso que detallados en el paso anterior (getter y setter).*

Como sucede en otros pasos, si no se especifica una ruta aparecerá un mensaje de error.

A.2.4. Paso 4: Subir la clase del algoritmo completada

En este paso el usuario tiene que indicar la ubicación del archivo algoritmo.java que contiene la implementación del algoritmo que procesará los datos.

Una vez más, aparecerá un mensaje de error si se intenta pasar al siguiente paso sin elegir la ruta.

A.2.5. Paso 5: Elegir o no JARS externos

Si la implementación del algoritmo de procesado de datos necesita de librerías externas, en este paso el usuario puede incluirlas, como se puede ver en la Figura A.7. En el panel desplegable se pueden seleccionando varios ficheros JAR, que se mostrarán en la interfaz, tal y como se muestra en la Figura A.8

Si no es necesario incluir ficheros JAR, se pasará directamente al paso 6.

```java
//----Casa.java----
//...
@JsonPropertyOrder({ "puerta" })
@Generated("jsonschema2pojo")
public class Casa {

  @JsonProperty("puerta")
  private String puerta;

  @JsonProperty("puerta")
  public String getPuerta() {
    return puerta;
  }

  @JsonProperty("puerta")
  public void setPuerta(String puerta) {
    this.puerta = puerta;
  }
//...
}
//----Ciudad.java----
//...
@JsonPropertyOrder({"Casa"})
@Generated("jsonschema2pojo")
public class Ciudad {

  @JsonProperty("Casa")
  private List<Casa> casa = null;

  @JsonProperty("Casa")
  public List<Casa> getCasa() {
    return casa;
  }

  @JsonProperty("Casa")
  public void setCasa(List<Casa> casa) {
    this.casa = casa;
  }
//...
}
```

Figura A.4: Ejemplo de clase -*Schema*

A.2.6. Paso 6: Elegir el protocolo de comunicación

En este paso hay que seleccionar el protocolo de comunicación que usará el consumidor, como muestra la Figura A.9. Actualmente se ofrecen dos protocolos de comunicación, Advanced Message Queuing Protocol (AMQP) y MQ Telemetry Transport (MQTT). El consumidor se genera para utilizar únicamente uno de los dos protocolos.

A.2.7. Paso 7: Seleccionar donde guardar el consumidor

El último paso consiste en la selección de una ruta para guardar la aplicación consumidor generada. Como en pasos anteriores, si se intenta finalizar sin seleccionar una ruta aparecerá un mensaje de error.

Figura A.5: Paso 3 del *Fabricante*

```java
public class Algoritmo implements Runnable {
    ArrayList<IrisSchema> datos;

    public Algoritmo(ArrayList<IrisSchema> datos){
        this.datos = datos;
    }

    @Override
    public void run() {
     //A completar por el desarrollador
    }
}
```

Figura A.6: Ejemplo de Plantilla del algoritmo

Figura A.7: Paso 5 del *Fabricante*

Tras unos segundos después de la selección aparecerá una ventana con un log que mostrará lo que ha sucedido durante el proceso de creación y compilación del consumidor, mostrando si ha habido algún error. La Figura A.10 muestra como se vería un resultado correcto.

Figura A.8: Paso 5* del *Fabricante*

Figura A.9: Paso 6 del *Fabricante*

Figura A.10: Paso 7 del *Fabricante*

APÉNDICE B

Manual del consumidor

Un consumidor es una aplicación para el procesamiento de datos recibidos al conectarse a un servidor RabbitMQ y cuyo comportamiento es configurable en cada ejecución. Cada consumidor es creado gracias al Fabricante*, para más información sobre éste consultar su manual A.*

B.1. Requisitos previos

El consumidor es independiente del sistema operativo sobre el que se use, solo necesita que en la máquina donde se vaya a ejecutar esté instalado el Java Runtime Environment (JRE). El JRE puede descargarse en este enlace [36].

B.2. Funcionamiento

Para ejecutar al consumidor primero hay que abrir una terminal de comandos y ejecutar la siguiente instrucción: java -jar [nombre del consumidor] [argumentos]

Hay dos formas de introducir los argumentos, en su versión corta, por ejemplo -ip (hostIP); o en su versión larga –host (hostIP). Los argumentos disponibles son los siguientes:

-h, –help *Argumento que imprime una ayuda de uso del consumidor.*

-ip (dirección del Host) o --host (dirección del Host) *Parámetro obligatorio que se usa para indicar la dirección IP del servidor RabbitMQ.*

-t (nombre del topic) o –topic (nombre del topic) / Solo versión MQTT *Argumento que indica el topic al que se va a suscribir el consumidor. Solo está disponible en la versión MQTT del consumidor y es obligatorio introducirlo.*

```
usage: Consumidor. Campos requeridos: Direccion del host, Nombre de la
                   cola
 -del,--delused                     Desecha los mensajes que ya han sido
                                    procesados (Por defecto: Desactivado
 -fe,--finalexec                    Flag que habilita una ultima ejecucion
                                    tras cerrar las conexiones (Por
                                    defecto: Desactivado). Este flag se
                                    ignora cuando no hay opcion de
                                    repeticion (Porque solo se puede
                                    ejecutar al final)
 -h,--help                          Imprime el mensaje de ayuda
 -ip,--hostip <hostIP>              Direccion IP del host (servidor
                                    rabbit)
 -max,--maxmessages <nummessages>   Numero maximo de mensajes esperados
                                    (Por defecto: indefinido)
 -mr,--messagerep <nummessages>     Ejecutar el algoritmo cada cierto
                                    numero de mensajes (Por defecto:
                                    desactivado)
 -p,--port <portnumber>             Puerto en el que escucha el servidor
 -pass,--password <password>        Contraseña en el caso de que la
                                    conexión necesite credenciales
 -q,--queue <queuename>             Nombre de la cola de conexion
 -tr,--timerep <interval>           Fija el intervalo de repeticion en
                                    minutos para ejecutar el algoritmo
                                    (Por defecto: desactivado)
 -u,--user <username>               Nombre de usuario en el caso de que la
                                    conexión necesite credenciales
 -w,--wait <waittime>               Tiempo que va a esperar el consumidor
                                    por un nuevo mensaje (Por defecto:
                                    Indefinido)
```

Figura B.1: Ayuda del consumidor (Versión AMQP)

-q (nombre de la cola) o --queue (nombre de la cola) / Solo versión AMQP *Argumento que indica el nombre de la cola a la que se va a suscribir el consumidor. Solo está disponible en la versión AMQP del consumidor y es un parámetro obligatorio.*

-p (número del puerto) o --port (número del puerto) *Parámetro opcional que se usa para indicar el puerto en el que escucha el servidor RabbitMQ.*

-u (nombre de usuario) o --user (nombre de usuario) *Parámetro opcional que usado para indicar el nombre de usuario en una conexión con credenciales.*

-pass (contraseña) o --password (contraseña) *Parámetro opcional que indica la contraseña en una conexión con credenciales.*

-d o --delused *Parámetro (flag) opcional usado para indicar si se quiere borrar los mensajes que ya han sido procesados por el algoritmo. Si no se indica se conservan todos por defecto.*

-fe o --finalexec *Parámetro (flag) opcional que se usa para indicar si ha de ejecutarse una última ejecución del algoritmo tras el cierre de conexiones. Si no se incluye este*

argumento no se produce la ejecución final, salvo en los casos en los que la configuración sea de no repetición.

-max (número de mensajes) o –maxmessages (número de mensajes) *Argumento que indicará el número máximo de mensajes que el consumidor va a esperar. Una vez llega a esa cifra, el consumidor cerrará las conexiones y terminará su ejecución.*

-mr (número de mensajes) o --messagerep (número de mensajes) *Parámetro que se usa para indicar cada cuantos mensajes recibidos se va a ejecutar el algoritmo.*

-tr (tiempo en minutos) o --timerep (tiempo en minutos) *Este argumento se usa para indicar cada cuanto tiempo (en minutos) el consumidor va a repetir la ejecución del algoritmo.*

-w (tiempo en minutos) o --wait (tiempo en minutos) *Se trata del tiempo que el consumidor esperará para que llegue un mensaje, este tiempo se restablece cada vez que llega uno nuevo. Si se sobrepasa este tiempo y no llega ningún mensaje, se cerrarán las conexiones y el consumidor terminará su ejecución. El tiempo hay que introducirlo en minutos.*

El uso de los argumentos presenta restricciones que pueden dar lugar a configuraciones inválidas. La siguiente Figura B.2 representa un esquema de posibles configuraciones que el usuario puede introducir. Las opciones se representan por el argumento a introducir en su versión larga.

Las líneas verticales discontinuas representan la separación entre niveles y se lee de izquierda a derecha, las opciones de un mismo nivel son excluyentes, por ejemplo si se utiliza la opción --timerep (tiempo en minutos), no se puede usar --messagerep (número de mensajes). El símbolo ____ indica la opción de no usar ningún argumento de los presentes en ese nivel. Las líneas horizontales de colores muestran la separación entre opciones y sus elecciones disponibles en niveles posteriores.

Una posible lectura sería usar el argumento delused (la otra opción en ese nivel sería no usarlo), después seleccionar timerep (la otra opción sería messagerep), tras ello seleccionar wait (las otras opciones serían maxmessages y no introducir nada en ese nivel) y por último finalexec (la alternativa sería no introducirlo).

La Figura B.2 es solo para ilustrar el árbol de opciones que se tiene al alcance a la hora de elegir una configuración, por tanto no hay que seguir el orden que se indica para introducir los argumentos.

B.2.1. Recepción de mensajes

Los datos recibidos por el consumidor han de ser cadenas de texto en formato JSON que presentan la estructura de JSON Array, en la Figura B.3 se puede ver un ejemplo de como tiene que ser la estructura de un mensaje JSON. Tienen que empezar y terminar por corchetes y cada objeto está encerrado entre llaves; cada objeto está separado de otro por una coma.

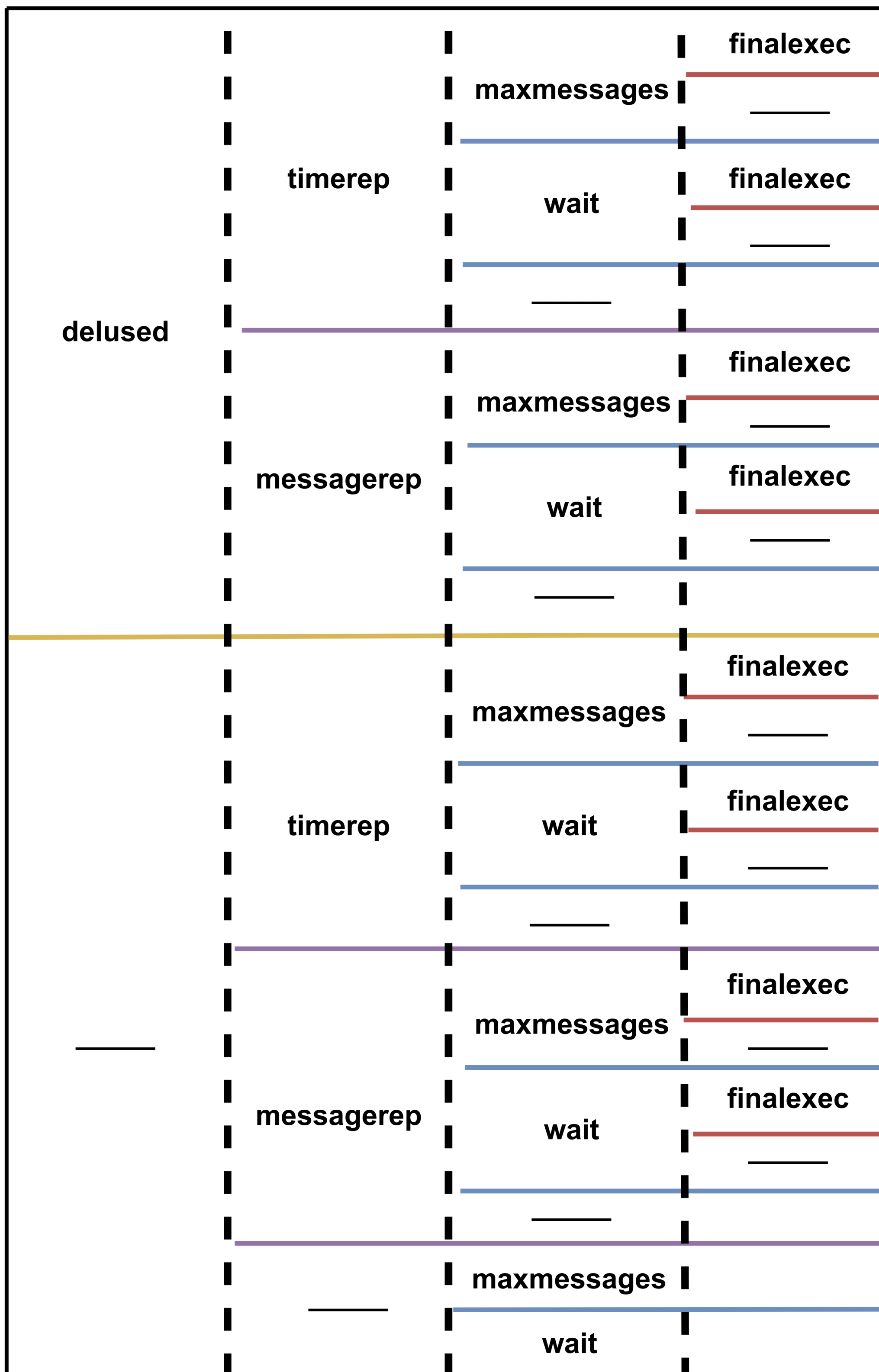

Figura B.2: Restricciones entre opciones de configuración

```
[
    {
        "sepal_length": "4.5",
        "sepal_width": "2.3",
        "petal_length": "1.3",
        "petal_width": "0.3",
        "species": "setosa"
    },
    {
        "sepal_length": "4.4",
        "sepal_width": "3.2",
        "petal_length": "1.3",
        "petal_width": "0.2",
        "species": "setosa"
    }
]
```


Figura B.3: Ejemplo de la estructura de los mensajes recibidos por el consumidor