Solución Publicador-Suscriptor para Edge/Fog Stream Processing usando Network Pipelining

—

Publisher-Subscriber Solution for Edge/Fog Stream Processing using Network Pipelining

TRABAJO FIN DE MÁSTER
Manual Técnico

Máster Universitario en Inteligencia Computacional
e Internet de las Cosas

Autor: Antonio Gómez Giménez

Directores: Fernando León García
José Manuel Palomares Muñoz

Córdoba

UNIVERSIDAD D CÓRDOBA

Índice:

1. Abstract / Resumen: 1
2. Introducción y motivación científica: 3
2.1. Hipótesis científica: 5
3. Revisión Bibliográfica 7
3.1. Fog/Edge Computing. 7
3.2. Stream Processing. 8
3.2.1. Edge Stream Processing. 8
3.3. Pipelining distribuido. 9
4. Objetivos e hipótesis: 11
5. Métodos y materiales 14
5.1. Paradigma publicador-suscriptor 14
5.1.1. MQTT 15
5.2. Plataformas hardware en Edge 16
5.2.1. Placas Espressif 16
5.3. Stream Processing Engines orientados a Edge 18
5.4. Network Pipelining 18
5.5. Lenguajes de programación y librerías para comunicaciones, toma de datos, exportación, etc. 19
5.5.1. Python 19
5.6. Plataformas de integración de IoT en la nube 20
5.6.1 Ubidots 20
5.7. Elección final de métodos y herramientas 21
6. Metodología: 24
6.1. Diseño: 24
6.1.1. Diseño general del sistema: 24
6.1.2. Explicación casos de uso: 29
6.1.2. Explicación árbol de topics: 30
6.2. Implementación: 33
6.2.1. Implementación simulada. 33
6.2.1.1. Servidor MQTT. 33
6.2.1.2. Librería STRPLibrary. 34
6.2.1.2.1. Master. 34
6.2.1.2.2. Slave. 46
6.2.1.3. Script python MasterControl. 52
6.2.1.4. Script python SlaveXControl. 57
6.2.1.5. Script NodeConexCloud. 64
6.2.1.6. Script auxiliar. 65
6.3. Aportación a la comunidad científica. 66

7. Resultados y discusión: **68**
7.1. Ejecución implementación simulada. 68
7.2. Pruebas. 75
7.3. Resultados: 85
7.3.1. Delay. 85
7.3.2. Throughput. 86
7.3.3. Discusión sobre los resultados. 87
8. Conclusiones y trabajo futuro: **93**
Bibliografía **97**
Agradecimientos **99**

1. Abstract / Resumen:

En la actualidad, existe un gran desarrollo en la tecnología, en concreto en IoT (Internet of Things), y en todo lo que esto conlleva, como Big Data. Este gran desarrollo conlleva también problemas técnicos, siendo el más importante el procesamiento de los datos. Por ello, en este proyecto se pretende plantear una posible solución al problema del procesamiento de datos aplicando el concepto de paradigma publicador-suscriptor, para permitir el desarrollo de sistemas distribuidos, reconfigurables y escalables, como es el caso de los *Stream Processing Engines.* Dicho paradigma, permite crear taxonomías descriptivas, permitiendo la distribución de los datos de la red de manera selectiva permitiendo mayor simplicidad.

At present, there is a great development in technology, specifically in IoT (Internet of Things), and all that this entails, such as Big Data. This great development also entails technical problems, being the most important the processing of the data. Therefore, this project aims to propose a possible solution to the problem of data processing by applying the concept of paradigm Publisher-Subscriber, to enable the development of distributed, reconfigurable and scalable systems, such as *Stream Processing Engines.*This paradigm allows the creation of descriptive taxonomies, allowing the distribution of network data selectively allowing greater simplicity.

Palabras Clave / Keywords:
Stream Processing, IoT, MQTT, Network Pipelining

2. Introducción y motivación científica:

Hoy en día, en pleno siglo XXI, podemos apreciar la continua evolución de las tecnologías que nos rodean. El desarrollo del concepto de Internet de las Cosas (IoT, Internet of Things) ha obtenido bastante relevancia debido a las mejoras que ha introducido, haciendo su uso imprescindible para el despegue real del Big Data.

IoT no es un concepto definido como tal, pero se puede llegar a entender cómo la digitalización de cosas y su respectiva conexión a internet. Por ejemplo una serie de sensores que reciben información y la traspasan entre otros dispositivos o internet para tomar una serie de decisiones automatizadas y sin intervención humana, todo ello teniendo en cuenta que los datos se generan en tiempo real.

Como respuesta a los desafíos técnicos que trae consigo la multiplicidad e inmediatez de las fuentes de información, no solo desde los dispositivos IoT, sino también de los servicios en la nube, cada vez más demandantes en cuanto a volumen de datos y tiempos de respuesta, surge el concepto de Stream Processing, con el objetivo de procesar flujos de datos de manera continua e ininterrumpida.

Existen proyectos software actuales, conocidos como Stream Processing Engines, que aprovechan la elasticidad de los recursos en los servidores Cloud para responder adaptativamente a las demandas de los flujos de datos entrantes, incorporando, entre otras, estrategias de tolerancia a fallos, protocolos de recuperación, etc.

Por otra parte, las crecientes capacidades de cómputo y red que presentan los dispositivos desplegados en los entornos (el borde de la red) no han pasado desapercibidas por la comunidad científica o la industria. Propuestas como Fog / Edge Computing parten de las características de este escenario para acercar, en la medida de lo posible, la computación a las fuentes de los datos. Es decir, delegar computación de la nube (Cloud) al borde (Edge) o la infraestructura (Fog) para conseguir menores latencias e incrementar la eficiencia.

Para dejar algo más claro cómo sería la estructura donde se encuentran estos tres conceptos (Cloud/Fog/Edge), se realizó la siguiente imagen:

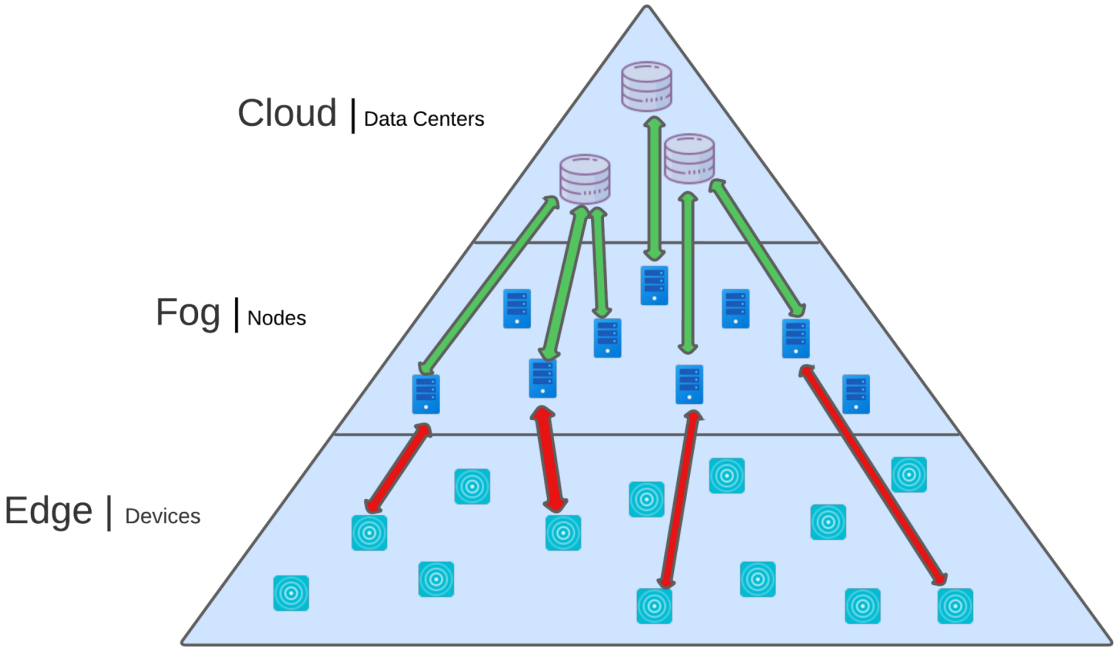

Imagen 2.1: ejemplo de estructura de la red con capas de Cloud, Fog y Edge.

Donde en dicha estructura de red, la capa de Cloud es la más pequeña y donde se suele realizar todo el procesamiento. La capa de Fog, es la intermediaria entre la capa de cloud y Edge, y se realiza algo de procesamiento, pero suele ser en menor medida que en el cloud. Y la capa de Edge, siendo estos los sensores que adquieren la información, es la capa de adquisición de datos. A medida que más se asciende en la pirámide, más se reduce la cantidad de dispositivos.

Como combinación de ambos conceptos surge el tópico de vanguardia científica "Edge Stream Processing", o procesamiento cercano y continuo de los flujos de datos generados.

IoT se aplica en distintos ámbitos como transporte, minería de datos, domótica, automatización, etc. En nuestro caso, dentro del amplio campo de aplicación de IoT, nos centraremos en **desarrollar una metodología para stream processing reconfigurable en entornos locales distribuidos con dispositivos de baja capacidad de cómputo.**

Para poder llevar a cabo la metodología anterior, será necesario **determinar la viabilidad de utilizar tecnologías de comunicación local basadas en el paradigma publicador-suscriptor para dar soporte al control de un Stream Processing Engine,** como puede ser el caso de MQTT.

Para comprobar la metodología creada y la viabilidad de la misma, se buscará **evaluar el rendimiento del desarrollo propuesto como caso de estudio,** siendo en nuestro caso, un contexto de **aplicación sintética**. Aun así, se podría llegar a realizar casos de estudios en entornos más reales, como podría ser en la agricultura, ya que es un campo donde IoT permite recolectar información para mejorar la productividad (mejorando calidad y cantidad) y mejorar la eficiencia, pudiendo ser un contexto de aplicación bastante interesante.

Respecto a la motivación científica, tras realizar un estudio de la revisión bibliográfica, se puede observar que es un tema interesante de investigar, en el sentido de que es un campo emergente. Ya que parece que la cantidad de artículos donde su caso de estudio se centra en el uso de MQTT para comprobar la estructura del sistema IoT (metadatos) al realizar Stream Processing en entornos distribuidos entre Edge/Fog, evitando sobrecargar la red y la nube, es limitado.
Por ello, se ha considerado un proyecto interesante con bastante flexibilidad a la hora de aplicar dicha metodología que se pretende crear en diferentes contextos de aplicación.

2.1. Hipótesis científica:

La hipótesis en la que se basa este trabajo y que se pretende resolver en este proyecto, es la siguiente:

“Creemos que el paradigma publicador-suscriptor es idóneo como base tecnológica para desarrollar un sistema distribuido reconfigurable y escalable, como es el caso de los Stream Processing Engines. Esto se debe fundamentalmente a la facilidad que introduce este paradigma para crear taxonomías descriptivas; los mecanismos que aporta para distribuir la información entre los nodos de la red de manera selectiva; y la simplicidad de los mecanismos de suscripción, de-suscripción y publicación”.

3. Revisión Bibliográfica

En este apartado se realizará una revisión bibliográfica de aquellos conceptos que se utilizan en este proyecto, permitiendo así entender el estado de la vanguardia y, comprobar así, desde que punto parte este proyecto.

3.1. Fog/Edge Computing.

Cuando nos referimos al concepto de **Fog/Edge Computing**, nos referimos al procesamiento que se realiza fuera de la nube, en las capas de Fog y Edge. Tanto Edge computing como Fog computing buscan procesar los datos lo más próximo al lugar de origen de los mismos. La principal diferencia entre ambos conceptos es el lugar donde se ubica dicho procesamiento de los datos.

Cabe destacar que, cuando nos referimos a nube, nos referimos a la infraestructura de servidores que albergan software y que se puede acceder a través de Internet. Siendo el **Cloud Computing**, el procesamiento que se realiza en dicha capa.

En el **Edge Computing**, el procesamiento se realiza en los dispositivos conectados, mientras que, en el **Fog Computing**, se coloca la inteligencia del procesamiento en la red de área local.

Por lo tanto, se podría decir que Edge Computing se usa para realizar cálculos sencillos e inmediatos, mientras que, Fog Computing busca realizar operaciones más complejas basándose en centros de microdatos.

Aunque sean conceptos similares con objetivos diferentes, estos pueden convivir en la misma infraestructura de red.

De hecho, podemos encontrar proyectos que trabajan sobre dichos conceptos, como puede ser en la siguiente conferencia **[1]**, donde se comentan distintos tipos de estructuras de red para poder mejorar temas como el ancho de banda, mejorando en general la latencia de comunicación.

3.2. Stream Processing.

El concepto de **Stream Processing** aparece por la idea de procesar datos de forma continua, realizando el procesamiento se forma secuencial sobre los datos que se van obteniendo. Este flujo de datos debe ser continuo e infinito, de tal forma que, continuamente se generen resultados de dichos datos.

Este concepto surge por la necesidad de aumentar la velocidad de procesamiento de los datos, ya que, la manera típica de procesar los datos es por batches, siendo los batches conjuntos de datos agrupados en grandes lotes.

Este concepto se ha aplicado en muchos ámbitos, como puede ser en el procesamiento de la nube **[2]** o incluso como se verá en el siguiente punto en la capa de Edge.

Incluso cabe destacar que este concepto no se limita únicamente al procesamiento en las capas de Edge, Fog o Cloud, ya que se utiliza en cualquier ámbito de procesamiento, como puede ser en proyectos donde se busca mejorar el rendimiento de un computador aplicando este concepto en el uso de cores. En el siguiente artículo se comenta dicho proyecto donde se basa en la idea anteriormente comentada **[3]**.

3.2.1. Edge Stream Processing.

Como se comentó en el apartado anterior, el concepto de Stream Processing queda ya explicado, pues el concepto de **Edge Stream Processing**, se basa en dicho concepto pero aplicándolo en concreto a la capa de Edge, siendo el procesamiento en Edge explicado en el apartado 3.1. Fog/Edge Computing.

Este concepto, es el que más relación podría tener con el proyecto que se plantea en este trabajo, ya que trata sobre Stream Processing aplicado en concreto al Edge.

En este caso, nos encontramos una serie de proyectos, que buscan dar soluciones al procesamiento de datos en el Edge donde aplican diversas herramientas o métodos para lograr dichos objetivos. Algunos de ellos, se basan en el uso de apache para lograr dichos objetivos usando herramientas como **Samza [4]**, **Flink [5]** o **Spark [6]** entre otros.

En el siguiente proyecto **[7]**, se aplica **Spark** para llevar a cabo el concepto de stream processing en el Edge y, mejorar así, el procesamiento de los datos para la conducción autónoma, ya que esta necesita trabajar en tiempo real.

En el proyecto **[8]**, se aplica **Samza** y **Flink** para el procesamiento del flujo de datos continuo, aplicando así Stream Processing. En dicho proyecto se busca crear un almacén de valor clave para sistemas basados en eventos, para garantizar un acceso de baja latencia a datos relevantes con sólidas garantías de consistencia, al mismo tiempo que brinda tolerancia frente a fallas geográficamente correlacionadas.

Como se puede observar, dicho concepto se puede aplicar en gran cantidad de proyectos que pretendan mejorar la latencia, mejorando la eficiencia del procesamiento de los datos.

3.3. Pipelining distribuido.

El concepto de pipelining distribuido, se puede comprender mejor, explicando el concepto de pipeline. **Pipeline** es un conjunto de procesos que se encuentran automatizados y que permiten transferir datos desde un origen hasta un destino en concreto. Este proceso puede también contener distintas extracciones de datos diferentes o el procesamiento (transformación) de los datos, para que al llegar al destino, se encuentren adaptados a él.

Los pipelines se utilizan en muchos ámbitos, utilizados por ejemplo, en procesamiento de volúmenes grandes de datos utilizados para Big Data o en aplicaciones de inteligencia artificial y aprendizaje automático.

Si el concepto anterior lo adaptamos a un entorno distribuido, obtenemos el concepto de **pipelining distribuido**.

Respecto a este concepto, se pueden encontrar artículos científicos donde se utiliza, en el artículo siguiente **[9]**, se comenta la creación de un framework basado en la programación de pipelining distribuido (DPPF), donde se busca aprovechar la estructura intuitiva de un pipeline para diseñar un patrón para la programación paralela y distribuida.

4. Objetivos e hipótesis:

La hipótesis en la que se basa este trabajo y que se pretende resolver, se comentó en el apartado de introducción siendo la siguiente:

“Creemos que el paradigma publicador-suscriptor es idóneo como base tecnológica para desarrollar un sistema distribuido reconfigurable y escalable, como es el caso de los Stream Processing Engines. Esto se debe fundamentalmente a la facilidad que introduce este paradigma para crear taxonomías descriptivas; los mecanismos que aporta para distribuir la información entre los nodos de la red de manera selectiva; y la simplicidad de los mecanismos de suscripción, de-suscripción y publicación.”

A partir de dicha hipótesis, surgen una serie de objetivos a cumplir, siendo los siguientes objetivos los objetivos a alcanzar con este proyecto:

Identificador	Objetivo
OB-01	Desarrollar una metodología para Stream Processing reconfigurable en entornos locales distribuidos con dispositivos de baja capacidad de cómputo.
OB-02	Determinar la viabilidad de utilizar tecnologías de comunicación local basadas en el paradigma publicador-suscriptor para dar soporte al control de un Stream Processing Engine.
OB-03	Evaluar el rendimiento del desarrollo propuesto en un caso de estudio real.

Los objetivos presentados con anterioridad se le van a asignar una prioridad alta a la hora de realizar este **TFM**, se podrían incluir otros objetivos y metas adicionales para mejorar el resultado de este **TFM**, pero se prefiere incluir una cantidad de objetivos reducida para completar la mayoría de los mismos e incluso todos.

Si se consigue realizar algún avance o mejora en este **TFM** que no se haya comentado en esta lista, se comentará en el apartado de conclusiones.

Cabe destacar que también se pretenden lograr otro tipos de objetivos, denominados **objetivos secundarios**, que se plantean como hitos para lograr los objetivos principales anteriormente comentados.

Algunos de ellos son:

- **Diseñar una interfaz de programación para configurar una cadena de procesos en cascada distribuidos (Pipeline),** permitiendo de una forma más sencilla, la gestión de distintos procesos.

- **Diseñar un mecanismo auto-adaptativo para reasignar los procesos encadenados a nodos disponibles,** permitiendo la adaptación de la red frente a diversos posibles fallos, evitando la caída de la red completa.

- **Diseñar una librería para apoyar el desarrollo de la metodología para Stream Processing reconfigurable,** permitiendo así su uso en un formato sencillo y simple.

- **Diseñar un contexto de aplicación de ejemplo**, para así comprobar el funcionamiento de la red y la librería previamente creadas.

Dichos objetivos secundarios son muy interesantes, ya que son considerados hitos a lograr dentro del proyecto, permitiendo así tener un mejor control sobre los objetivos que se van logrando a lo largo del proyecto y el avance del mismo.

5. Métodos y materiales

En este apartado se comentarán los métodos y materiales existentes que podrían ser utilizados para la realización de dicho proyecto, y más concretamente, para lograr los objetivos anteriormente comentados. Además de nombrar y explicar algunos posibles materiales y métodos, se explicarán aquellos que se han escogido para la realización del proyecto.

5.1. Paradigma publicador-suscriptor

Para poder realizar la comunicación entre dispositivos, es necesario la existencia de un paradigma para que lo permitan, existen diferentes métodos de comunicación, pero para nuestro proyecto se eligió el **paradigma publicador-suscriptor**, ya que está centrado principalmente para situaciones donde se necesita comunicación en grupo, es decir, situaciones donde un mensaje es enviado por un dispositivo y es requerido por uno o varios dispositivos.

Además, este paradigma es muy utilizado cuando se desea dividir la información de manera más cómoda y eficiente entre distintos elementos de un grupo, ya que, al contrario que el modelo de comunicación multicast, los clientes del modelo publicador-suscriptor pueden describir los eventos en los que están interesados de una forma más específica (especificando por ejemplo los topics de los que desea información).

El paradigma publicador-suscriptor consiste, por lo menos, en un broker que redirige las notificaciones de los clientes del sistema hacia otros clientes mostrando así su interés en recibir dichas notificaciones. Esto permite gran escalabilidad y distribución de la red permitiendo gran número de clientes y comunicaciones entre ellos, además, se pueden añadir más brokers permitiendo aún más escalabilidad.

Cabe destacar que la entrega de las notificaciones entre los clientes puede ser de manera síncrona o asíncrona, dependerá del protocolo utilizado.

Los clientes pueden publicar notificaciones o suscribirse a los filtros (topics por ejemplo) que vinculan a dicho cliente con las notificaciones proporcionadas por los brokers, ya que estos proporcionarán dichas notificaciones a los clientes cumplan con las condiciones de dichos filtros.

De tal forma que, cuando un broker recibe una nueva notificación, comprueba sus suscripciones locales, y en caso de que se cumpla el filtro para alguno de ellos, entregar dicha notificación a cada uno de ellos. Además el propio broker puede reenviar dicha notificación a otros brokers que conviven en el mismo grupo.

En la actualidad existen distintos protocolos que se basan en el paradigma publicador-suscriptor para su funcionamiento, siendo de los más conocidos por ejemplo **MQTT** (Message Queue Telemetry Transport) o **COAP** (Constrained Application Protocol).

5.1.1. MQTT

MQTT (Message Queue Telemetry Transport) es un protocolo basado en el paradigma publicador-suscriptor. Se considera un protocolo de transporte M2M (machine to machine) de tipo message queue. Está basado en la pila TCP/IP para realizar dicha comunicación. A diferencia de otros protocolos de transporte como HTTP 1.0, donde para cada transmisión se realiza una conexión, MQTT mantiene cada conexión abierta y esta se reutiliza para cada comunicación.

El protocolo MQTT fue creado por **Dr. Andy Stanford-Clark [10]** de IBM y **Arlen Nipper de Arcom** (ahora Eurotech) en 1999 como un mecanismo para conectar dispositivos empleados en la industria petrolera. Ya que los ingenieros necesitaban un protocolo que permitiera un ancho de banda mínimo y una pérdida de batería mínima, para poder así supervisar los oleoductos vía satélite.

El protocolo MQTT se ha convertido en un estándar para la transmisión de datos, sobre todo en IoT, ya que dispone de los siguientes beneficios.

- **Ligero y eficiente**. Requiere recursos mínimos, pudiéndose usar incluso en pequeños microcontroladores.
- **Escalable**. Optimizado para funcionar con gran cantidad de dispositivos IoT
- **Fiable**. Funciones que reducen el tiempo de reconexión con la nube e implementación de niveles de calidad de servicio.
- **Seguro**. Facilita a los desarrolladores el cifrado de mensajes y autenticidad de usuarios y dispositivos.
- **Admitido**. Varios lenguajes, como python por ejemplo, están adaptados a este protocolo existiendo librerías con la implementación de MQTT

5.2. Plataformas hardware en Edge

La computación en el Edge, es aquella computación que se realiza en la ubicación física del usuario, en concreto, cerca de la fuente de datos o muy cerca a la misma, esto permite que los servicios sean más rápidos y confiables, permitiendo por ejemplo, el concepto de cloud computing híbrido, de tal forma que se realiza parte de la computación en el edge y en la nube.

Esto ha hecho que en los últimos años haya existido un auge de la tecnología denominada “Internet de las cosas” (IoT, Internet of Things), consistiendo en el uso de dispositivos que mantienen conexión permanente a internet con el objetivo de transmitir información que monitorizan sensores o incluso, realizar tareas de control de manera remota.

Para poder lograr dichos objetivos, existen muchos módulos y componentes, pero los más utilizados son las placas Espressif.

5.2.1. Placas Espressif

Las placas Espressif, reconocidas también como ESP, son las placas más utilizadas en IoT por su bajo costo, simplicidad y su variedad.

Espressif es una empresa China situada en Shangai que se dedica al diseño de chips electrónicos denominados SoC (System On Chips). Estos chips integran distintos bloques con distintas funciones en un circuito único integrado, reduciendo así el tamaño de la propia placa y por lo tanto, reduciendo así el costo.

Por ejemplo, el Soc ESP contiene un microcontrolador, un conversor A/D, un stack TCP/IP que contiene el software o firmware necesario para resolver la mayor parte de comunicación por internet , distintas entradas y salidas, GPIO y memoria flash.

Actualmente existen tres tipos de SoC, siendo **ESP32**, **ESP8266** y **ESP32-S**. Las principales características de cada tipo son las siguientes:

ESP32:

- Una o dos CPU LX6 de 32 bit con velocidades entre 80 MHz y 240 MHz.
- Soporte para Bluetooth y Bluetooth Low Energy (BLE).
- Consumo de corriente menor a 5 µA en modo de bajo consumo.
- Gran variedad de periféricos incluyendo sensores táctiles capacitivos, de efecto Hall, interface a tarjetas SD, Ethernet, SPI, UART, I2S e I2C.
- 520 KB de memoria de datos e instrucciones
- Memoria Flash incluida hasta 4 MB (según el modelo), soporte hasta 16 MB de Flash externa.
- Soporte para WiFi b/g/n a 2.4 GHz.

ESP8266:

- CPU de 32 bit LX6 de Tensilica corriendo a 160 MHz
- Modo de bajo consumo que permite su funcionamiento alimentado por baterías.
- Incluye una amplia variedad de periféricos como UART, GPIO, I2C, I2S, SDIO, PWM, ADC y SPI.
- 32 KB de memoria para instrucciones.
- 80 KB de memoria para datos.
- Soporte para WiFi b/g/n a 2.4 GHz.

ESP32-S:

- CPU de 32 bit LX7 de Tensilica corriendo a 240 MHz.
- Coprocesador RISC-V.
- 320 KB de memoria de datos e instrucciones.
- No incluye memoria Flash pero soporta hasta 1 GB de Flash externa.
- Características de seguridad: eFuse, encriptación de memoria Flash, algoritmos AES, SHA y RSA integrados.
- Periféricos incluidos: 43 GPIOs, interface USB OTG, SPI, I2S, UART, I2C, LED PWM, LCD, interface a cámara, ADC, DAC, sensor táctil, sensor de temperatura.
- Soporte para WiFi b/g/n a 2.4 GHz.

5.3. Stream Processing Engines orientados a Edge

La computación en el edge, como comentamos anteriormente, es aquella computación que se realiza en la ubicación física del usuario, en concreto, cerca de la fuente de datos o muy cerca a la misma.

El concepto de stream processing surge de la idea de procesar los datos de forma continua, de tal forma que, en cuanto los datos se obtienen se procesan de forma secuencial. Para poder conseguir este objetivo, el flujo de datos debe ser infinito y sin límites de tiempo.

La manera típica de procesar los datos es a través del Batch Processing, agrupando los datos en grandes lotes, llamados batches.

La unión de ambos conceptos genera el concepto de Stream Processing Engines pero con la orientación a Edge. De tal forma, que consiste en el procesamiento del flujo de datos pero orientado a los dispositivos del Edge, realizando el procesamiento mientras el flujo de datos se dirige a la nube.

Existen algunas herramientas que se basan en el concepto de Stream Processing Engines como puede ser kafka o spark, ya que actualmente los servicios de stream processing tienen más demanda y este tipo de herramientas, permiten acelerar la velocidad con las que se obtienen los datos y generar interacciones con los mismos.

Cabe destacar que es un tema en auge donde existen proyectos basados tanto en Kafka **[11]** como con Spark **[12]**.

5.4. Network Pipelining

El concepto de Network Pipelining viene de la agrupación de distintos pipelines, siendo un pipeline el procesamiento que se le aplica a un flujo de datos. Un pipeline, contiene diferentes stages (etapas de procesamiento). De tal forma que, un Network Pipelining permite la paralelización del procesamiento, existiendo distintos pipelines que realizan el procesamiento sobre etapas.

5.5. Lenguajes de programación y librerías para comunicaciones, toma de datos, exportación, etc.

Respecto a los lenguajes de programación, podemos encontrar mucha diversidad como por ejemplo, **Python, C, C++, C#, Java**, entre otros.

Es recomendable elegir el lenguaje de programación que más ventajas nos ofrece para el problema a resolver, para ello, hay que tener en cuenta elementos como:

- Facilidad de uso.
- Soporte y documentación abundante.
- Librerías existentes.
- Optimización del código.
- Etc

De entre todos los lenguajes posibles para este proyecto se escogerá Python, en el siguiente subapartado se entiende las ventajas que este lenguaje nos ofrece frente a otros para este proyecto en concreto.

5.5.1. Python

Python es un lenguaje de programación creado a finales de los ochenta por Guido van Rossum. Python es un lenguaje interpretado, dinámico y multiplataforma de alto nivel, cuya filosofía se centra en la legibilidad del código y su facilidad de uso.

Actualmente, Python es un lenguaje muy usado donde su comunidad ha creado gran cantidad de librerías para permitir que su uso sea aún más sencillo.

Aparte, también existen lenguajes originados a partir de python, como puede ser **micropython**, que está fuertemente ligado a las placas Espressif, anteriormente explicadas. Esto se debe a que **micropython** es una implementación del lenguaje de programación Python orientada a aquellos dispositivos que cuentan con una cantidad de recursos limitada. Refiriéndonos a dispositivos como los ESP32, los ESP32-S o los ESP8266.

Respecto al tema de las librerías, como Python dispone de una gran comunidad, existen ya librerías que implementan el protocolo MQTT, facilitando el desarrollo del sistema. Un ejemplo de librería de python para implementar dicho protocolo es **paho-mqtt**.

En el caso de que se utilice una ESP-32, como su capacidad es reducida, dicha librería no podría utilizarse, por ello existen otras librerías con tamaño reducido para poder lograr dicho objetivo. Un ejemplo de librería sería **umqttsimple.**

5.6. Plataformas de integración de IoT en la nube

Una plataforma de integración de IoT en la nube es una infraestructura y las aplicaciones necesarias que reúnen, almacenan y ofrecen inteligencia a partir de los datos que generan los sensores de los dispositivos conectados a internet. Dicha plataforma unificada, está alojada en la nube y centraliza la recopilación de datos y el uso e intercambio de dicha información.

Este tipo de plataformas son muy útiles para reducir el tiempo necesario en gestionar y crear una plataforma IoT propia para la recolección y gestión de los datos.

Existen diversa variedad de plataformas de integración de Iot en la nube, como por ejemplo pueden ser Ubidots, Fracttal, Akenza, Google Cloud, etc

5.6.1 Ubidots

Ubidots es una plataforma de IoT que permite la toma de decisiones para empresas de integración de sistemas a nivel global.
En concreto permite enviar datos de los dispositivos sensores (Edge/Fog) a la nube, permitiendo a su vez configurar alertas o dashboard. También permite conexión con otras plataformas, usar herramientas analíticas o incluso mostrar mapas de datos en tiempo real. Dispone de una versión gratuita de prueba donde se pueden usar algunas de estas funciones.

5.7. Elección final de métodos y herramientas

En este apartado se explican los métodos y herramientas escogidas para el proyecto, aclarando el porqué de su elección. Para poder llevar a cabo dichas elecciones, se tuvo en cuenta el problema real y técnico, para intentar alcanzar los objetivos del proyecto.

Como bien se ha explicado, el problema real que se abarca en este proyecto, viene heredado de los conceptos de Edge Computing, Fog Computing y Stream Processing: teniendo en cuenta la saturación de los nodos centrales de cálculo, necesidad de procesar cerca de los generadores de información, etc etc. Todos estos conceptos son propuestas que se han desarrollado para dar solución a esto.
Pero las soluciones basadas en estos conceptos siguen en desarrollo, teniendo un margen de mejora, por ejemplo carecen de flexibilidad (son ad-hoc). Este proyecto busca adaptarse a dichos problemas y crear una solución a ellos.

El problema científico parte de las ideas de estas soluciones propuestas, aportando la flexibilidad y dinamización de recursos que consideramos que estas redes necesitan. Centrándose en el uso de Stream processing pero respecto al Edge/Fog Computing, alejándose del procesamiento realizado en la nube. Por ello, surge la hipótesis de que el paradigma publicador suscriptor puede facilitar bastante el desafío de manejar la red de cómputo distribuido de forma abstracta, simple, escalable, etc. Este es el núcleo del trabajo y por ello, se crea el código necesario para demostrar y probar esta hipótesis.

Para poder llevar a cabo la solución de dichos problemas y teniendo en cuenta los objetivos del proyecto, se escogieron las siguientes herramientas y métodos:

Respecto al **protocolo utilizado para el proyecto** nos encontramos con los dos siguientes: MQTT y COAP. Para nuestro proyecto se escogió MQTT, aunque se podría haber escogido COAP.

Principalmente se escogió el protocolo MQTT por su implementación y por la comunicación de los nodos, ya que es fácilmente implementable y su comunicación es M:N, mientras que en COAP, la comunicación de los nodos es 1:1 y existen pocas librerías y soporte.

Aun así COAP no sería mala elección ya que ofrece otras muchas ventajas frente a MQTT, como podrían ser, menor consumo de energía, posibilidad de mensajes asíncrono y síncrono, etc.

Respecto **a las plataformas hardware en Edge usadas** nos encontramos con las placas Espressif, pudiendo ser: ESP32, ESP8266 y ESP32-S. Entre sí todas ellas son similares, aún así hay distintas ventajas y desventajas entre ellas, dependiendo del problema donde se vayan a aplicar dichas placas. Para nuestro proyecto se escogió la **ESP32**, principalmente por la disponibilidad de la misma aparte de su bajo consumo, memoria flash de gran capacidad frente a las otras familias, etc.

Respecto **al lenguaje de programación usado** nos encontramos diversas opciones y para este problema en concreto, se escogió Python. Las principales causas por las que se escogió este lenguaje de programación frente a otros, aparte de por su facilidad de uso, fue por las librerías ya existentes que permitían acelerar el proceso de desarrollo, aparte de que para el uso de la ESP32, es prácticamente necesario el uso de micropython, de tal forma que es más lógico usar para la versión simulada código Python, ya que es código con un formato similar.

Respecto a **Stream Processing Engines orientados a Edge y Network Pipelining,** se utilizaron los conceptos para la creación del sistema, pero no se utilizó ningún tipo de herramienta como Kafka o Spark, ya que se prefería mayor control de la red pudiendo gestionar en conjunto el procesamiento, los stages y los pipelines.

Respecto **a la plataforma de integración de IoT en la nube** nos encontramos muchas opciones, en nuestro caso se escogió ubidots. Para este proyecto la plataforma de integración no es algo relevante. Se escogió principalmente Ubidots por su facilidad de uso y porque su concepto sencillo de enviar los datos a un punto central donde poder consultarlos, parece bastante interesante como un apoyo al proyecto, permitiendo mayor organización de los datos finales.

6. Metodología:

En este apartado se comentará la metodología llevada a cabo para la realización del proyecto, esta constará de un apartado de diseño y un apartado de implementación. Primero se especificará el diseño para crear el sistema y posteriormente, se entrará en detalle de cómo se implementa dicho diseño.

6.1. Diseño:

Teniendo en cuenta los objetivos que el sistema debe satisfacer, se puede empezar a realizar la etapa de diseño. Para ello, se explicará el diseño general del sistema y se especificará el diseño específico de ciertos apartados que se consideran de relevancia.

6.1.1. Diseño general del sistema:

Para poder mostrar con mayor claridad el diseño general del sistema, se mostrará el problema global al que nos enfrentamos de forma gráfica, para adaptar así el sistema en la medida de lo posible a dicho problema. Antes de mostrar gráficamente el problema global, se realizará la gráfica para una versión más sencilla del problema, para mayor entendimiento.

La gráfica sería la siguiente:

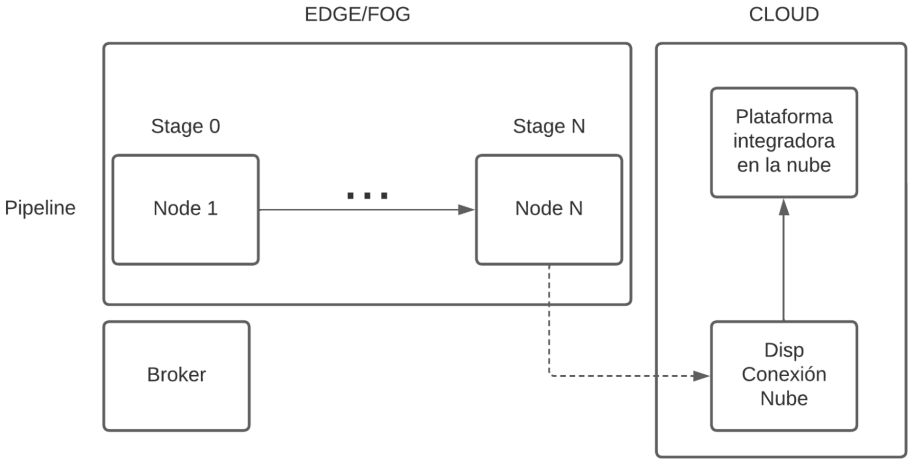

Imagen 6.1.1.1: diagrama del problema en una versión sencilla

Como se puede apreciar en la imagen anterior, se creó un diagrama con una versión sencilla del problema, es decir, si se pretende resolver un problema aplicando stream processing y usando MQTT como protocolo publicador-suscriptor, esta sería la solución en el caso de que no se aplique un sistema de control y teniendo en cuenta que es una versión reducida.

Como se puede apreciar, en el problema existen los **pipelines** y los **stage**. Siendo los stages las etapas en las que se divide el procesamiento, y los pipelines, las líneas de trabajo, desde la introducción de uno o varios datos hasta la solución obtenida tras su procesamiento (podrían existir varias, pero para el caso más sencillo se crea solo una).

Cada nodo representa una máquina, en nuestro caso es una esp32, siendo los nodos capaces de realizar el procesamiento necesario. Estos nodos se encuentran en las posibles capas de EDGE o FOG, ya que el Stream Processing se realiza en estas capas obviando el procesamiento en la nube. En este caso EDGE y FOG, se pueden llegar a fusionar, ya que un nodo que se encuentra en EDGE generando datos, puede tener la capacidad de también procesarlos, por ese motivo estas dos capas no se encuentran completamente separadas. Entonces en dichas capas se encuentran los nodos que realizan todo lo relacionado con el procesamiento, y aparte, en la capa que se encuentra cercana al cloud, nos encontramos con el nodo que envía los datos a la nube una vez se disponga del resultado a enviar.

Por último, encontramos un nodo llamado broker, dicho nodo es necesario para MQTT, ya que este nodo gestiona el árbol de topics y las comunicaciones entre suscriptores y publicadores.

Una vez explicado el diagrama anterior para un problema sencillo, se procede con la creación del diagrama para un problema más complejo, en el que se centra dicho proyecto. El diagrama es el siguiente:

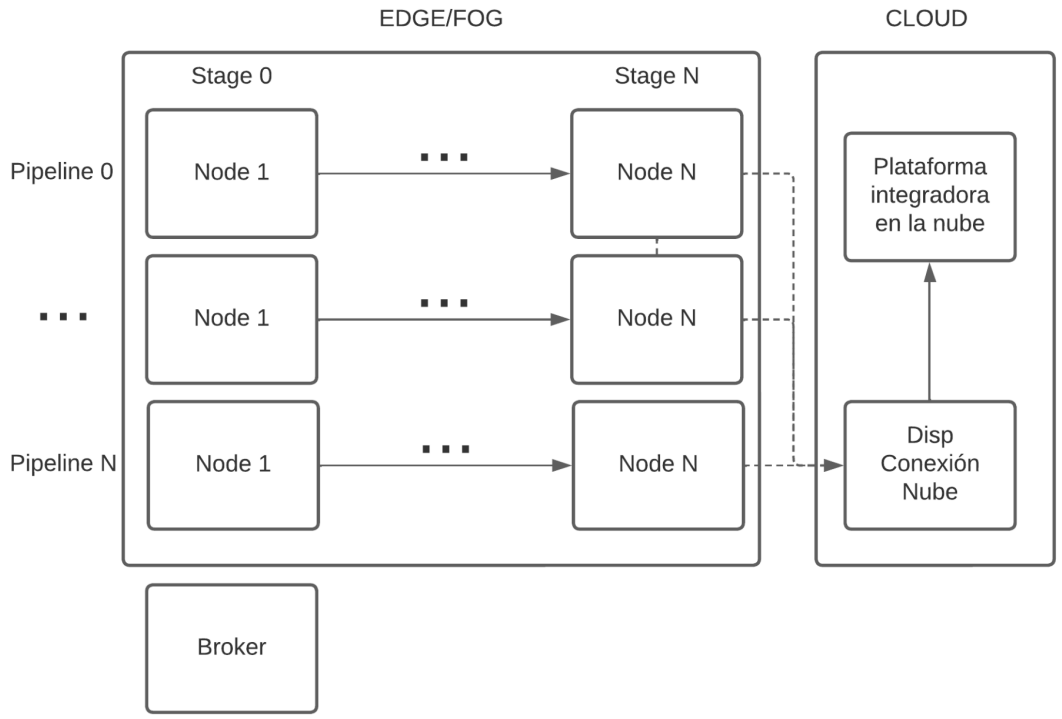

Imagen 6.1.1.2: diagrama del problema en una versión compleja

Como se puede observar en el diagrama anterior, la principal diferencia es la cantidad de pipelines, ya que esta puede ser variable. Esto implica que la creación de cada pipeline debería realizarse a mano junto a cada stage. Todo esto puede ser muy laborioso, aparte de que la red no dispondría de una mínima robustez.

Por lo tanto, lo que se busca para solucionar estos problemas, es añadir una capa de control que se abstraiga del plano de datos, es decir, el contexto de la aplicación (pudiendo ser agricultura entre muchas otras).

Para poder lograr la creación de la capa de control, se creará una librería con dos clases, maestro y esclavo, de tal forma, que los esclavos se limitarán a realizar el procesamiento del contexto de aplicación llamando a esta clase y el máster, realizará la gestión de la red.

Para que sea más sencillo de entender, se realiza el siguiente diagrama:

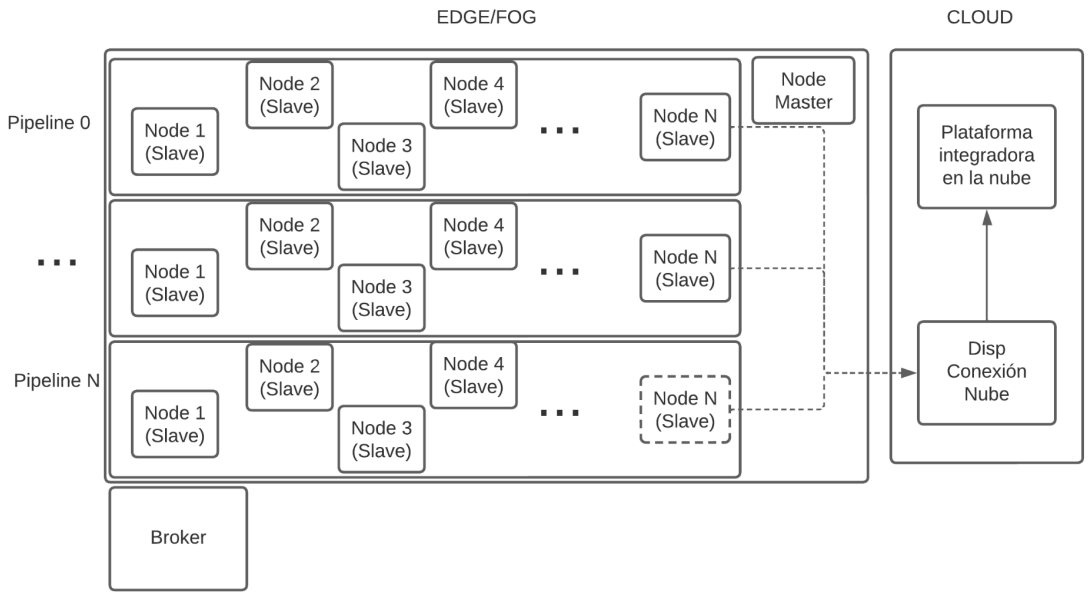

Imagen 6.1.1.3 diagrama que representa la solución dada por la librería

Como se puede apreciar, cada pipeline dispondrá de una serie de esclavos, independientes entre sí. De esta forma se abstraen los nodos de ejecución de los stage a procesar. El máster estará comunicado con todos los nodos y se encargará de asignar cada stage o varios de ellos a los esclavos que sean necesarios y las comunicaciones entre los distintos nodos, permitiendo así gran flexibilidad en la red. De esta forma se puede separar completamente la parte de control de la parte de contexto de la aplicación.

Para resumir el funcionamiento a grandes rasgos de todo el sistema a nivel general se muestra el siguiente grafo donde se incluye al usuario/administrador que ejecuta el sistema:

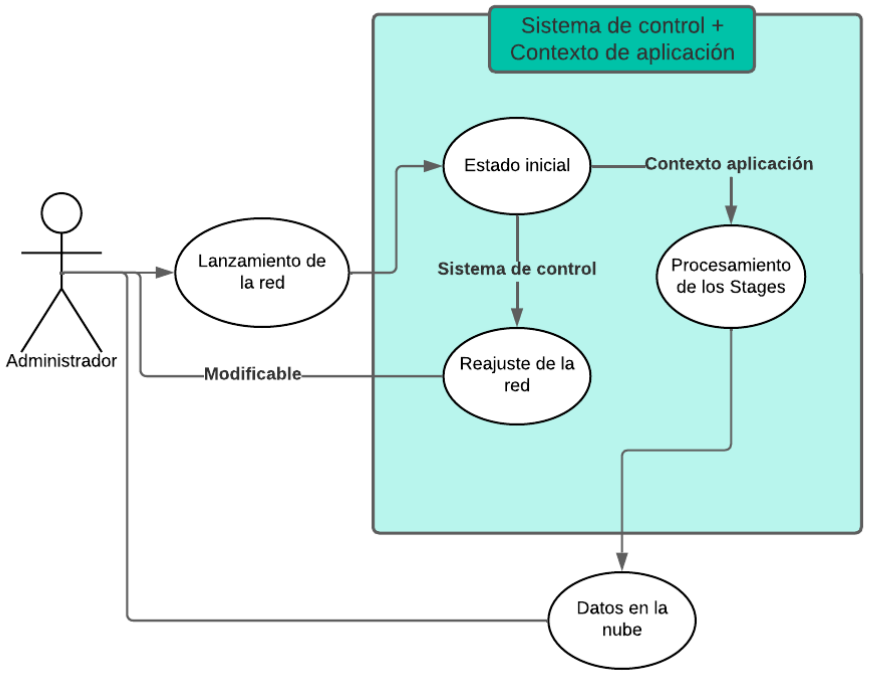

Imagen 6.1.1.4 diagrama del funcionamiento a grandes rasgos de todo el sistema

Como podemos ver en la imagen previa, se pone como ejemplo de usuario el **administrador**.

El **administrador**, tendrá pleno control de todo el sistema, pudiendo implementar la lógica de reajuste de la red, ya que dispondrá de métodos de la librería master permitiendo implementar la lógica necesaria el control del sistema. El administrador será el usuario que arranque el sistema y que podrá consultar en la nube los resultados de los datos.

6.1.2. Explicación casos de uso:

Una vez explicado de forma general el funcionamiento del sistema y su actuador, se explicarán todos los casos de uso, entrando en mayor profundidad sobre su uso y funcionamiento.

CU-Lanzamiento de la red. En este caso de uso, se realiza el lanzamiento de la red, es decir, la puesta a punto de todos los nodos que trabajan tanto los esclavos como el master. Para el correcto funcionamiento del sistema, se deberán lanzar los script de python que contendrán en el caso de los esclavos, el contexto de aplicación junto con la respectiva llamada a la clase slave de la librería, que gestionará el control de las llamadas entre nodos y el procesamiento que realizará dicho nodo.

Respecto al script que contendrá el master, contendrá la llamada a dicha clase y el código de control de la red, es decir, las reglas y gestión sobre como deberá de funcionar la red (juntar stages o separarlos, gestionar caídas de nodos, etc) apoyándose en la librería, concretamente en los métodos de la clase master.

Una vez los scripts se encuentren preparados y el nodo broker arrancado (por ejemplo configuración por defecto), se lanzarán en el caso de ser una simulación, en hilos de un procesador, y en el caso de una esp32 se arrancará directamente en la máquina.

CU-Estado inicial. Este caso de uso se podría considerar transitorio, esto se debe a que carece de sentido realizar ajustes sobre la red como juntar stages si no existe un número mínimo de nodos. Durante este caso de uso, los esclavos pedirán un identificador al master y este, les generará un id y un stage que procesar en el caso de que sea posible.
El master, generará una baliza para pedir información del estado de los esclavos para así posteriormente reestructurar la red en caso de que sea necesario.

Una vez exista un número mínimo de esclavos con id, se comienza la reestructuración en el caso de ser necesario, en paralelo al procesamiento de los datos en el contexto de aplicación.

CU-Procesamiento de los stages. En este caso de uso, se realiza el procesamiento de cada esclavo, teniendo en cuenta los stages que le haya asignado el master. Este caso de uso es paralelo al caso de uso de Reajuste de la red. Este caso de uso se limita a realizar el procesamiento indicado por el máster y enviarlo al nodo que indique el máster. Dicho nodo puede llegar a procesar varias etapas antes de enviar el resultado.

CU-Reajuste de la red. En este caso de uso, es paralelo al caso de uso de procesamiento de los stages. Se limita a reajustar la red, ya sea introduciendo nuevos nodos, parando nodos (un nodo deja de funcionar), re-asignando stages, agrupando stages en un esclavo o dividiendo los stages en diferentes, etc. Cómo se tratan de llamadas a métodos de la clase master, el usuario puede controlar el funcionamiento de la red, como por ejemplo, cada cuánto envía la baliza la llamada para que los esclavos informen de su estado.

CU-Datos en la nube. Este caso de uso hace referencia a cómo se muestran los datos resultantes. El usuario podrá comprobar el resultado obtenido de un pipeline tras su procesamiento. Para ello, cuando se obtenga un resultado, un nodo deberá capturarlo y enviarlo a una nube, donde finalmente se mostrará.

6.1.2. Explicación árbol de topics:

Una vez explicado el problema planteado, el diseño general para resolverlo y los casos de uso con mayor profundidad, se explicará de una forma más detallada el **árbol de topics de mqtt** que se pretende utilizar.

La parte de control y la parte de contexto de aplicación se encuentran totalmente separadas, por ello, se podrían considerar dos árboles de topics distintos aunque en este proyecto, para mayor sencillez se han realizado juntos.

Para la **parte de contexto** se utilizan los siguientes topics:

App
/APPLICATION_CONTEXT/ID-X
/APPLICATION_CONTEXT/PIPELINE-W/RESULT

La X simboliza el id del esclavo, de esta forma, los esclavos pueden comunicarse para el procesamiento de los datos.

En el segundo caso la w indica de donde proviene el pipeline y junto la etiqueta RESULT, se envía a dicho topic el resultado del pipeline.

Para la **parte de control**, situada en la librería con las clases master y slave, nos encontramos los siguientes topics:

Master
/CONTROL/MASTER/GET_MY_ID
/CONTROL/MASTER/ID-X/GET_CONNECT_NODES
/CONTROL/MASTER/ID-X/GIVE_INFO

Estos topics son los topics que utiliza la clase máster de la librería, se utiliza el mismo root **/CONTROL/MASTER/** para referenciar al master.

La etiqueta **GET_MY_ID** es una llamada que realizan los esclavos cuando quieren obtener una id.

La x hace referencia al id que realizó la petición, de tal forma que, para **ID-X/GET_CONNECT_NODES** se informa al esclavo x de cuales son los esclavos que tiene delante (a quien le envía) y detrás (de quien recibe).

Por último, para **ID-X/GIVE_INFO**, el esclavo especificado da su información de estado al master.

Slave
/CONTROL/SLAVE/SET_MY_ID/X
/CONTROL/SLAVE/ID-X/NEW_SUSCRIBER
/CONTROL/SLAVE/ID-X/NEW_PUBLISHER
/CONTROL/SLAVE/REQUEST_INFO
/CONTROL/SLAVE/ID-X/UPDATE_STAGE

Estos topics son los topics que utiliza la clase slave de la librería, se utiliza el mismo root **/CONTROL/SLAVE/** para referenciar al slave.

La etiqueta **SET_MY_ID/X** es una llamada que recibirán los esclavos cuando quieran obtener una id, la x representa un código que tienen los esclavos antes de tener asignada una id.

La x cuando se encuentra en ID-X hace referencia al id que realizó la petición, de tal forma que, para **ID-X/NEW_SUSCRIBER** se informa al esclavo x de cuál es su nuevo suscriptor.

Para **ID-X/NEW_PUBLISHER** se informa al esclavo x de cuál es su nuevo publicador (a quién publica).

Para **REQUEST_INFO** el máster pide el estado del esclavo, por lo tanto, el esclavo da su información.

Por último, para **ID-X/UPDATE_STAGE**, el esclavo especificado actualiza su stage.

Cabe destacar que **el nodo que se conecta a la nube** dispone del siguiente topic para enviarlo:

/v1.6/devices/prueba-stream-processing

Se necesita previamente conexión mqtt con el broker de ubidots.

6.2. Implementación:

En este apartado, se comentará la implementación realizada sobre el diseño explicado en el apartado anterior, en concreto sobre la implementación simulada.

6.2.1. Implementación simulada.

En este apartado se explicará detalladamente todos los pasos realizados para llevar a cabo la simulación del sistema que se planteó en el apartado de diseño.

6.2.1.1. Servidor MQTT.

Para poder trabajar con mosquito, es necesario tener un servidor que haga de broker, para ello se usó un portátil Toshiba Satellite L50D-C-13P de 8G de ram, procesador AMD A10-8700p radeon r6 y una gráfica AMD Radeon r6 graphics para dicho propósito. El sistema operativo utilizado es ubuntu 20.04 y los pasos para su instalación fueron los siguientes:

- sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
- sudo apt-get update
- sudo apt-get install mosquitto
- sudo apt-get install mosquitto-clients
- sudo apt clean

Como se utilizó la configuración por defecto, ya se podía utilizar el servidor y arrancarlo o pararlo según se desee. Los comandos para parar y arrancar mqtt broker son los siguientes:

```
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/iot/TrabajoFinal/SimulacionFin
al$ sudo service mosquitto stop
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/iot/TrabajoFinal/SimulacionFin
al$ sudo mosquitto -v -c new_mosquitto.conf
```

Imagen 6.2.1.1.1: comandos para parar y arrancar mqtt

6.2.1.2. Librería STRPLibrary.

Para la implementación de las clases master y slave, se diseñaron dos clases que se encuentran almacenadas en el script de python STRPLibrary, siendo una librería propia.

En este apartado se explicará más en profundidad el funcionamiento de cada clase y sus diferentes métodos.

6.2.1.2.1. Master.

En este apartado se explicará el funcionamiento y la implementación de la clase master, para ellos se muestra el siguiente diagrama con el funcionamiento que debe realizar dicha clase.

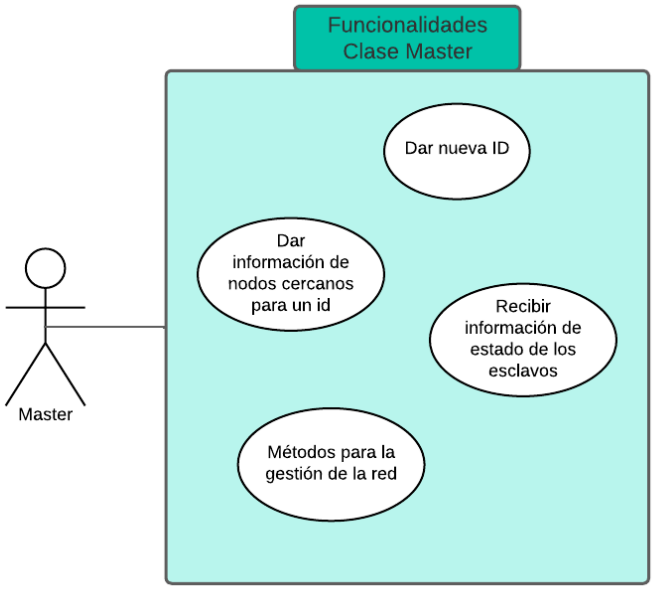

Imagen 6.2.1.2.1.1: diagrama de las funcionalidades de la clase master

Antes de explicar las funcionalidades de la clase master, cabe destacar que es una clase con el objetivo de crear la estructura de la red para un contexto de aplicación, es decir, se centra exclusivamente en la parte de control para la gestión de la red.

De hecho, cuando se crea una clase master, se ejecuta el código del constructor, que es el siguiente:

```
# ==============================
# Builder
# ==============================
# This funcion create the obj master
# Param in ->
# pipelines: Nº pipelines
# stages: Nº stages
# IP_SERVER: Ip server
# PORT: Nº port
# Param out ->
# None
def __init__(self, pipelines, stages, IP_SERVER, PORT):
    #create events for eventsProcessor
    self.__events()

    # Create a thread with the func __eventsProcessor
    start_new_thread(self.__eventsProcessor,())

    # Assign initial values to private variables
    self._Stages = stages
    self._Pipelines = pipelines
    for i in range(pipelines):
        self._Structure.append([])

    self._IP_SERVER = IP_SERVER
    self._PORT      = PORT
    self.__connect_to_server()# try to connect to the server
```

En dicho código, cuando realizamos la creación de dicha clase, estamos creando una conexión mqtt al broker para realizar la gestión de la parte de control y además creamos la estructura donde se definen los pipelines a utilizar y la cantidad de stages que se usarán en el contexto de aplicación.

Una vez explicado la inicialización de la clase, está ya puede realizar las funcionalidades vistas en el diagrama de la imagen 6.1.1.2.1.1

Dar nueva id. Como la clase master está inicializada y tiene conexión al broker de mqtt, esta ya puede recibir mensajes de topics a los que se haya suscrito.
Por lo tanto, cada vez que reciba un topic con la siguiente etiqueta **/CONTROL/MASTER/GET_MY_ID,** debe generar un nuevo id y enviarle esta información al slave que lo haya pedido, además debe guardar en la estructura de la red el nuevo id y si se le ha asignado algún tipo de stage.

Para poder lograr dicho propósito, se realizó el siguiente código:

```
# if the topic match with the Regex Get Id, give id to this slave
if re.search(RegEx_Get_Id, self.message.topic):
    print (f"DATA_RECEIVED: {self.message.topic}={self.message.payload}")
    code = self.message.topic[-29:]# get code to return the new id
    data_in=json.loads(self.message.payload)
    # take pipeline where it will work
    Number_Pipeline = data_in
    # assign number
    new_Id = 0
    for i in range(len(self.get_Structure())):
        new_Id += len(self.get_Structure()[i])
    # assign stage
    if len(self.get_Structure()[Number_Pipeline]) != 0:# check if the pipeline is empty
        if len(self.get_Structure()[Number_Pipeline][-1]['List_Stages']) != 0:# check if List_Stages
its empty
            if self.get_Structure()[Number_Pipeline][-1]['List_Stages'][-1] != -1:# check if Last
node its not working
                Number_Stage = [self.get_Structure()[Number_Pipeline][-1]['List_Stages'][-1] + 1]
#take new stage inside the pipeline
                if Number_Stage[-1] >= self.get_Stages():# check if new stage is out of range
                    Number_Stage = []
            else:
                Number_Stage = []
        else:
            Number_Stage = []
    else:
        Number_Stage = [0]
    # create dicc for new slave
    dicc = {
        "ID":new_Id,
        "List_Stages":Number_Stage
    }
    dicc_tmp = {}# dicc empty
    self._Status_IDS.insert(new_Id,[0,dicc_tmp])# id with counter + diccionary with status
    self.get_Structure()[Number_Pipeline].append(dicc)
    data_out=json.dumps(dicc)
    self.conex_to_Server.publish(ROOTSLAVE +"SET_MY_ID/" + code, data_out)
```

Como se puede apreciar en el mensaje recibido, existe un código único del slave que solicita el id, ya que la primera vez que un slave envía un mensaje este no tiene id y se necesita poder identificarlo de forma unívoca para poder asignarle dicho id.

Cada vez que se genera un nuevo id para un slave, se almacena en una lista la estructura de la red con el nuevo id y el stage, en el caso de que se le haya asignado uno. Cuando se asigna un id, se comprueba los stages, en el caso de que haya stages libres se van asignando conforme llegan nuevos id, y en el caso de que no queden stages libres, dichos id no tendrán ningún stage y por tanto, serán ids ociosos.

De esta forma, si en algún momento se necesita asignar algún id por cualquier motivo, se dispondrán de estos slaves ociosos.

Finalmente, el master pública en el topic **/CONTROL/SLAVE/SET_MY_ID/X**, para que el nodo slave que esté suscrito a dicho topic pueda saber cuál es su id.

Dar información de nodos cercanos para un id. Esta funcionalidad es similar a la anterior, y se activa de la misma forma, cuando un slave la solicita a través de mqtt. Por lo tanto, cada vez que se reciba un topic con la siguiente etiqueta **/CONTROL/MASTER/ID-X/GET_CONNECT_NODES,** debe proporcionar al slave la información de cuál es el nodo que se encuentra delante y detrás, haciendo referencia a quien publica y quien le publica.
Esta función se realiza solo de forma informativa para que el slave sepa quien tiene delante y detrás, pero para el contexto de aplicación no es relevante para su funcionamiento.

Se realizó el siguiente código:

```
# if the topic match with the Get Connect Nodes, for this id get the next and previous nodes
elif re.search(RegEx_Get_Connect_Nodes, self.message.topic):
    print (f"DATA_RECEIVED: {self.message.topic}={self.message.payload}")
    data_in=json.loads(self.message.payload)
    dicc = data_in
    Number_Pipeline = self.__Get_Pipeline(dicc["ID"])
    #Get connect nodes
    if len(dicc["List_Stages"]) != 0:# check if List_Stages is empty
        Previous_ID = self.ID_Previous_Stage(Number_Pipeline, dicc["List_Stages"][0])
        if  Previous_ID != -1:# check if exist previous id
            data_out=json.dumps(Previous_ID)
            self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(dicc["ID"]) + "/NEW_SUSCRIBER",
data_out)# update previous nodes for actual id
            data_out=json.dumps(dicc["ID"])
            self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Previous_ID) + "/NEW_PUBLISHER",
data_out)# update next nodes for before id
```

Como se puede apreciar en dicho código, el master extrae la información necesaria y pública en los topics del slave que haya pedido dicha información, cual es el nodo anterior y cual es el siguiente. Pública en los topics:

/CONTROL/SLAVE/ID-X/NEW_SUSCRIBER
/CONTROL/SLAVE/ID-X/NEW_PUBLISHER.

Recibir información de estado de los esclavos. Esta funcionalidad es similar a la anterior, y se activa de la misma forma, cuando un slave da su información a través de mqtt.
Por lo tanto, cada vez que se reciba un topic con la siguiente etiqueta **/CONTROL/MASTER/ID-X/GIVE_INFO,** el master actualizará una lista, que se va creando a medida que se introducen nuevos id, con los estados de cada slave y un contador de actualización, que permite gestionar si un nodo slave deja de enviar actualizaciones, y por tanto, se da por muerto.

Se realizó el siguiente código:

```
# if the topic match with the Give Info, update _Status_IDS with new status info and reset the
counter
elif re.search(Regex_Give_Info, self.message.topic):
    print (f"DATA_RECEIVED: {self.message.topic}")
    data_in=json.loads(self.message.payload)
    dicc = data_in
    Iterator_ID = dicc["ID"]
    self._Status_IDS[Iterator_ID] = [0, dicc]
```

En dicho código, se puede observar cómo se almacena para la id que haya recibido el estado del slave, y además, se pone el contador 0, ya que dicho id ha enviado su estado.

Métodos para la gestión de la red. Esta funcionalidad es distinta a las anteriores, ya que hace referencia a métodos de la clase, en vez de activarse con eventos de mqtt. Esta funcionalidad hace referencia a las gestiones que realizará el usuario sobre la red, como puede ser la gestión de los status o la reestructuración de la red con la caída de un nodo slave.

En resumen, se comentan los métodos de la clase que permiten aplicar la lógica necesaria para el control de la red. Los métodos creados son los siguientes:

get_Stages(), **get_Pipelines()**, **get_Structure()**, **get_Status_IDS()**, **get_IP_SERVER()** y **get_PORT()**, son getters sencillos para consultar las distintas variables, como por ejemplo, el número de pipelines, la estructura, en número de stages, el estado de los ids, etc.

get_stages_from_Id(). Este método devuelve una lista de con los stage asignados a una id.

El código es el siguiente:

```
# ============================
# get_stages_from_Id
# ============================
# This funcion return stages from id
# Param in ->
# Id: Nº id
# Param out ->
# list_stages: list stages
def get_stages_from_Id(self, Id):
    structure = self.get_Structure()
    list_stages = []
    for i in range(len(structure)):
        for w in structure[i]:
            if w["ID"] == Id:
                if len(w["List_Stages"]) != 0:# check if List_Stages is empty
                    list_stages = w["List_Stages"]
    return list_stages
```

Como podemos observar, en este método, se devuelve una lista con los stages para el id que se especifique, para ello recorre la estructura y devuelve los stages del id que coincida.

set_stages_from_Id(). Este método es similar al anterior, pero en este caso se actualizan los stages del id proporcionado con una lista de nuevos ids.

El código es el siguiente:

```
# ============================
# set_stages_from_Id
# ============================
# This funcion return stages from id
# Param in ->
# Id: Nº id
# list_stages: list stages
# Param out ->
# None
def set_stages_from_Id(self, Id, list_stages):
    structure = self.get_Structure()
    for i in range(len(structure)):
        for w in structure[i]:
            if w["ID"] == Id:
                w["List_Stages"] = list_stages
```

request_info(). Este método incrementa todos los contadores de los estados de los ids y realiza una publicación a los slave para solicitar la información del estado, es una función muy útil si se utiliza dentro de una baliza, permitiendo gestionar las llamadas a los esclavos para solicitar información (cada cuanto se realizan las llamadas, por ejemplo).

```
# ==============================
# request_info
# ==============================
# Func that increase all counters for status counters and publish REQUEST_INFO
# Param in ->
# None
# Param out ->
# None
def request_info(self):
    for i in self.get_Status_IDS():
        if i[0] < 999: #999 is the counter limit
            i[0] = i[0] + 1
    self.conex_to_Server.publish(ROOTSLAVE + "REQUEST_INFO", 0)
```

Como podemos observar, en este método, también se asegura que el contador no sobrepase los 999, ya que si un nodo cae y nunca se recupera, este número llegaría hasta el infinito.

ID_Previous_Stage(). Este método devuelve el ID del stage previo al solicitado, es decir, para el pipeline seleccionado y el stage elegido, devolverá en el caso de ser posible, cual es el id del stage anterior.

```
# ==============================
    # ID_Previous_Stage
    # ==============================
    # Func get the previous stage for stage given (inside pipeline)
    # Param in ->
    # pipeline: Nº pipeline
    # stage: Nº stage
    # Param out ->
    # ID_Previous_Node: ID for previous stage to the stage passed
    def ID_Previous_Stage(self, pipeline, stage):
        structure = self.get_Structure()
        ID_Previous_Node = -1
        if stage != 0:# if stage is 0, we dont need to check
            for i in range(len(structure)):
                if i == pipeline:
                    for w in structure[i]:
                        for x in w["List_Stages"]:
                            if x == stage-1:
                                ID_Previous_Node = w["ID"]
        return ID_Previous_Node
```

Como podemos observar, en este método, se recorre el pipeline seleccionado en busca del stage anterior al especificado, devolviendo el id del mismo.

ID_Next_Stage(). Este método devuelve el ID del stage posterior al solicitado, es decir, para el pipeline seleccionado y el stage elegido, devolverá en el caso de ser posible, cual es el id del siguiente stage.

```
# ==============================
# ID_Next_Stage
# ==============================
# Func get the next stage for stage given (inside pipeline)
# Param in ->
# pipeline: Nº pipeline
# stage: Nº stage
# Param out ->
# ID_Next_Node: ID for next stage to the stage passed
def ID_Next_Stage(self, pipeline, stage):
    structure = self.get_Structure()
    ID_Next_Node = -1
    for i in range(len(structure)):
        if i == pipeline:
            for w in structure[i]:
                for x in w["List_Stages"]:
                    if x == stage+1:
                        ID_Next_Node = w["ID"]
    return ID_Next_Node
```

Como podemos observar, en este método, se recorre el pipeline seleccionado en busca del stage siguiente al especificado, devolviendo el id del mismo.

delete_Id_stage(). Este método, es un método complejo que elimina todos los stages de un id dado, pasando a ser un nodo no funcional, se considera que dicho nodo se ha caído.

```
# ==============================
# delete_Id_stage
# ==============================
# Func that delete stage for id given, update neighbors stage, this id dont work
# Param in ->
# Id: Nº id
# Param out ->
# None
```

```
def delete_Id_stage(self, Id):
    structure = self.get_Structure()
    for i in range(len(structure)):
        for w in structure[i]:
            if w["ID"] == Id:
                if len(w["List_Stages"]) != 0:# check if List_Stages is empty
                    empty_stages = w["List_Stages"]# save stages for update neighbors
                    w["List_Stages"] = [-1]# this id dont work
                    dicc = {
                        "ID":Id,
                        "List_Stages": w["List_Stages"]
                    }
                    # update id given
                    data_out=json.dumps(dicc)
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Id) + "/UPDATE_STAGE",
data_out)
                    data_out=json.dumps(-1)
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Id) + "/NEW_SUSCRIBER",
data_out)
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Id) + "/NEW_PUBLISHER",
data_out)
                    # update id for next stage
                    Previous_ID = self.ID_Previous_Stage(i, empty_stages[0])
                    if Previous_ID != -1:
                        self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Previous_ID) +
"/NEW_PUBLISHER", data_out)
                    # update id for previous stage
                    Post_ID = self.ID_Next_Stage(i, empty_stages[-1])
                    if Post_ID != -1:
                        self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Post_ID) +
"/NEW_SUSCRIBER", data_out)
```

Como podemos observar, aparte de actualizar la estructura de la red del master eliminando el stage del id proporcionado, actualiza al propio id con la nueva información (pasar a ser un nodo no funcional), y además, actualiza el id siguiente y anterior que tenían relación con el stage eliminado, evitando así llamadas recursivas al master.

get_Stages_Without_ID(). Este método, es un get que devuelve una lista con aquellos stages que no se encuentran asignados a ningún id. Esto es muy útil si se desea gestionar aquellos stages que no están asignados a ningún id, permitiendo gestionarlos, ya sea asignándoles a un id ocioso o juntando dicho stage con un id que disponga de un stage próximo, por ejemplo.

El código es el siguiente:

```
# ==============================
# get_Stages_Without_ID
# ==============================
# Func get Stages that dont have ID
# Param in ->
# pipeline: Nº pipeline
# Param out ->
# Stage_Without_ID: return list stage without ID
def get_Stages_Without_ID(self, pipeline)  :
    structure = self.get_Structure()
    list_Stages_With_ID = []
    list_Stages_tmp = []
    for i in range(self.get_Stages()):
        list_Stages_tmp.append(i)# save all stages
        for w in structure[pipeline]:
            if len(w) != 0:# check if pipeline is empty
                if i in w["List_Stages"]:# check if status i is inside in w["List_Stages"]
                    list_Stages_With_ID.append(i)
    Stages_Without_ID = list(set(list_Stages_tmp) - set(list_Stages_With_ID))
    return Stages_Without_ID
```

Como podemos observar, se recorre la estructura de la red, en el pipeline especificado, en busca de aquellos stages que no estén asignados. Para ello, como se sabe los stages totales, se buscan dentro del pipeline todos los stage asignados y finalmente, se busca la diferencia entre los stage totales y los stage encontrados, siendo este resultado los stage sin asignar.

get_Free_Slaves(). Este método, es un get que devuelve una lista con aquellos ids que no tienen asignado stages, por lo tanto son nodos ociosos.

El código es el siguiente:

```
# ==============================
# get_Free_Slaves
# ==============================
# Func get free slaves
# Param in ->
# pipeline: Nº pipeline
# Param out ->
# list_free_slaves: return list free slaves
def get_Free_Slaves(self, pipeline):
    structure = self.get_Structure()
    list_free_slaves = []
    for i in structure[pipeline]:
        if len(i) != 0:# check if pipeline is empty
            if len(i["List_Stages"]) == 0:# check if ["List_Stages"] is empty for each id in
pipeline
                list_free_slaves.append(i["ID"])
    return list_free_slaves
```

Como podemos observar, se recorre la estructura de la red, en el pipeline especificado, en busca de aquellos ids que no tienen ningún stage.

set_Stage_To_Id(). Este método, es un método complejo que dado un stage y un id, asigna a dicho id el stage especificado.

El código es el siguiente:

```
# ============================
# set_Stage_To_Id
# ============================
# Func get free slaves
# Param in ->
# stage: Nº stage
# Id: Nº Id
# Param out ->
# None
def set_Stage_To_Id(self, stage, Id):
    structure = self.get_Structure()
    for i in range(len(structure)):
        for w in structure[i]:
            if w["ID"] == Id:
                # update id
                w["List_Stages"].append(stage)
                tmp_list = w["List_Stages"]
                w["List_Stages"] = list(set(tmp_list))
                dicc = {
                    "ID":Id,
                    "List_Stages": w["List_Stages"]
                }
                data_out=json.dumps(dicc)
                self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Id) + "/UPDATE_STAGE",
data_out)
                # update id for previous stage
                Previous_ID = self.ID_Previous_Stage(i, w["List_Stages"][0])
                if  Previous_ID != -1:
                    data_out=json.dumps(Previous_ID)
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(dicc["ID"]) +
"/NEW_SUSCRIBER", data_out)
                    data_out=json.dumps(dicc["ID"])
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Previous_ID) +
"/NEW_PUBLISHER", data_out)
                # update id for next stage
                Post_ID = self.ID_Next_Stage(i, w["List_Stages"][-1])
                if Post_ID != -1:
                    data_out=json.dumps(Post_ID)
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(dicc["ID"]) +
"/NEW_PUBLISHER", data_out)
                    data_out=json.dumps(dicc["ID"])
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Post_ID) +
"/NEW_SUSCRIBER", data_out)
```

Como podemos observar, se actualiza la estructura de la red asignando el nuevo stage, además se actualiza dicho nodo slave para que actualice su información.
Por último, se actualizan los nodos slaves con stages vecinos, para que tengan en cuenta la actualización de la estructura realizada.

delete_Stage_To_Id(). Este método, a diferencia del método **delete_Id_stage(),** es un método complejo que dado un stage y un id, elimina el stage de dicho id.

El código es el siguiente:

```
# ============================
# delete_Stage_To_Id
# ============================
# Func delete stage from id
# Param in ->
# stage: Nº stage
# Id: Nº Id
# Param out ->
# None
def delete_Stage_To_Id(self, stage, Id):
    structure = self.get_Structure()
    for i in range(len(structure)):
        for w in structure[i]:
            if w["ID"] == Id:
                # update id
                if stage in w["List_Stages"]:
                    list_stages_update = w["List_Stages"]
                    while(stage in list_stages_update):# remove the stage
                        list_stages_update.remove(stage)
                    # update id
                    w["List_Stages"] = list(set(list_stages_update))
                    dicc = {
                        "ID":Id,
                        "List_Stages": w["List_Stages"]
                    }
                    data_out=json.dumps(dicc)
                    self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Id) + "/UPDATE_STAGE",
data_out)
                    # update id for previous stage
                    Previous_ID = self.ID_Previous_Stage(i, w["List_Stages"][0])
                    if  Previous_ID != -1:
                        data_out=json.dumps(Previous_ID)
                        self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(dicc["ID"]) +
"/NEW_SUSCRIBER", data_out)
                        data_out=json.dumps(dicc["ID"])
                        self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Previous_ID) +
"/NEW_PUBLISHER", data_out)
                    # update id for next stage
                    Post_ID = self.ID_Next_Stage(i, w["List_Stages"][-1])
                    if Post_ID != -1:
                        data_out=json.dumps(Post_ID)
```

```
                        self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(dicc["ID"]) +
"/NEW_PUBLISHER", data_out)
                        data_out=json.dumps(dicc["ID"])
                        self.conex_to_Server.publish(ROOTSLAVE + "ID-" + str(Post_ID) +
"/NEW_SUSCRIBER", data_out)
                else:
                    print("Can't delete stage, that stage no exist in this id")
```

Como podemos observar, se actualiza la estructura de la red eliminando el stage de dicho id (si se eliminan todo sus stage pasaría a ser un nodo ocioso, no es un nodo caído), además se actualiza dicho nodo slave para que actualice su información. Por último, se actualizan los nodos slaves con stages vecinos, para que tengan en cuenta la actualización de la estructura realizada.

6.2.1.2.2. Slave.

En este apartado se explicará el funcionamiento y la implementación de la clase slave, para ello se muestra el siguiente diagrama con el funcionamiento que debe realizar dicha clase.

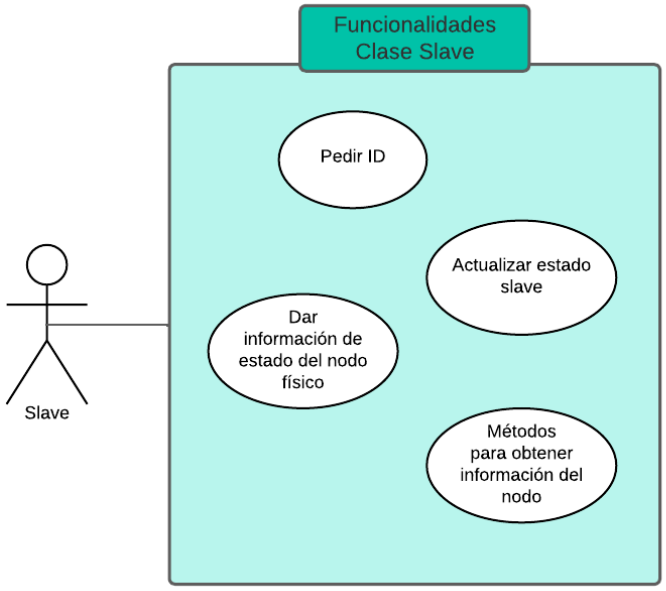

Imagen 6.2.1.2.2.1: diagrama de las funcionalidades de la clase slave

Antes de explicar las funcionalidades de la clase slave, cabe destacar que es una clase con el objetivo de crear cada nodo slave de la red donde se almacenarán los topics del contexto de aplicación, es decir, se centra exclusivamente en la identificación del propio id nodo y de las conexiones cercanas, para así almacenar donde publica. Por último, también cada nodo informa del estado de la máquina y el estado del slave.
De hecho, cuando se crea una clase slave, se ejecuta el código del constructor, que es el siguiente:

```
# ==============================
# Builder
# ==============================
# This funcion create the obj slave
# Param in ->
# pipeline: Nº pipeline
# IP_SERVER: Ip server
# PORT: Nº port
# Param out ->
# None
def __init__(self, pipeline, IP_SERVER, PORT):
    #create events for eventsProcessor
    self.__events()

    # Create a thread with the func __eventsProcessor
    start_new_thread(self.__eventsProcessor,())

    # Assign initial values to private variables
    self._Pipeline = pipeline

    self._IP_SERVER = IP_SERVER
    self._PORT      = PORT
    self.__connect_to_server()# try to connect to the server

    # Stop code when this slave doesn't have ID
    self.__TakeNumber()
    while(self.get_MY_ID() == -1):
        print("Taking a ID from Master... ")
        sleep(10)

    # This slave subscribes to its own id
    MQTT_TOPIC = [(ROOTSLAVE + "ID-" + str(self.get_MY_ID()) + "/" +"#",0)]
    self.conex_to_Server.subscribe(MQTT_TOPIC)
    # The slave requests his neighbors nodes
    dicc = {
        "ID":self.get_MY_ID(),
        "List_Stages":self.get_Stages()
    }
    data_out=json.dumps(dicc)
    self.conex_to_Server.publish(ROOTMASTER + "ID-" + str(self.get_MY_ID()) + "/GET_CONNECT_NODES",
data_out)

    # The slave is ready to give his info
    MQTT_TOPIC = [(ROOTSLAVE + "REQUEST_INFO",0)]
    self.conex_to_Server.subscribe(MQTT_TOPIC)
```

En dicho código, cuando realizamos la creación de dicha clase, estamos creando una conexión mqtt al broker para realizar la gestión de la parte de control y además se crean las variables globales para la gestión de estado de dicho slave, por ejemplo el pipeline donde se encuentra o el id cuando lo pida.

En la propia inicialización de la clase, encontramos una de las funcionalidades vistas en el diagrama de la imagen 6.2.1.2.2.1

Pedir id. Para que un slave comience su correcto funcionamiento en la red este debe tener un id proporcionado por el master, para ello se creó el método privado *__TakeNumber()*, donde continuamente se pide un id al máster hasta que este se lo proporciona. No se realizan otras funcionalidades del slave hasta que este no dispone de un id.

La función *__TakeNumber()*, se puede apreciar en la siguiente imagen:

```
# ============================
# __TakeNumber
# ============================
# This funcion take Id from master to this slave
# Param in ->
# None
# Param out ->
# None
def __TakeNumber(self):
    MQTT_TOPIC = [(ROOTSLAVE + "SET_MY_ID/" + str(self._Mac) + str(self._Time),0)]# subscribe to his
set id
    self.conex_to_Server.subscribe(MQTT_TOPIC)
    data_out=json.dumps(self.get_Pipeline())
    self.conex_to_Server.publish(ROOTMASTER + "GET_MY_ID/" + str(self._Mac) + str(self._Time),
data_out)
```

Como se puede observar, se pide al master, a través de topics de mqtt, que le proporcione un id. Para poder recibir el id a este slave es necesario especificar un identificador único, por ello se utiliza la mac del dispositivo.

En este caso concreto, como todos los slave son hilos de una misma máquina al ser simulado, el mac no es suficiente como identificador. Se propuso la idea de añadir al código mac el tiempo de la petición para que así fuera un código único para cada slave.

Si nos fijamos de nuevo en el builder de la clase, como podemos observar, tras obtener el id, dicho slave se suscribe a los topics que hacen referencia a su id, además se actualiza el estado del slave con su id nuevo, pide al máster cuales son los nodos vecinos, y por último, se suscribe al topic que sirve de baliza para enviar el estado de la máquina donde se encuentra el slave.

Actualizar estado slave(). Como la clase slave está inicializada y tiene conexión al broker de mqtt, esta ya puede recibir mensajes de topics a los que se haya suscrito. Por lo tanto, cada vez que reciba un topic al que se encuentra suscrito, realizará en algún tipo de acción, en este caso que hace referencia a actualizar el estado del slave, los topics son los siguientes:

/CONTROL/SLAVE/ID-X/NEW_SUSCRIBER
/CONTROL/SLAVE/ID-X/NEW_PUBLISHER
/CONTROL/SLAVE/ID-X/UPDATE_STAGE

Cada vez que se activa un evento de recibir un mensaje de los topics anteriores, se realiza una acción. Aunque el código sea diferente para cada evento, las acciones como tal son muy similares, se limitan a actualizar el nodo anterior (topic **NEW_SUSCRIBER**), actualizar el nodo siguiente (topic **NEW_PUBLISHER**) y actualizar los stage que procesa dicho slave(topic **UPDATE_STAGE**).

Un ejemplo de código sería el siguiente:

```
# if the topic match with the RegEx Update Stage, update the stage for this slave
elif re.search(RegEx_Update_Stage, self.message.topic):
    data_in=json.loads(self.message.payload)
    self._Stages = data_in['List_Stages']
```

Dar información de estado del nodo físico. Esta funcionalidad es similar a la anterior, y se activa de la misma forma, cuando el master la solicita a través de mqtt.

Por lo tanto, cada vez que se reciba un topic con la siguiente etiqueta **/CONTROL/SLAVE/REQUEST_INFO,** debe proporcionar el slave la información del estado de la máquina donde se encuentra y publicarlo en el topic que esté suscrito el master siendo el topic **/CONTROL/MASTER/ID-X/GIVE_INFO**.

Para poder lograr dicho propósito, se realizó el siguiente código:

```
# if the topic match with the RegeX Request Info, send the status info to master
elif re.search(RegeX_Request_Info, self.message.topic):
    # Get info from this slave
    dicc = self.__get_Info_From_Slave()
    data_out=json.dumps(dicc)
    self.conex_to_Server.publish(ROOTMASTER + "ID-" + str(self.get_MY_ID()) + "/GIVE_INFO",
data_out)
```

```
# ==============================
# __get_Info_From_Slave
# ==============================
# This funcion get all status info from slave
# Param in ->
# None
# Param out ->
# dicc: dicc with all status info from slave
def __get_Info_From_Slave(self):
    # Stats Frecuency
    cpufreq = psutil.cpu_freq()
    # Stats Mem
    svmem = psutil.virtual_memory()
    # Stats Network
    NetSpeed = psutil.net_if_stats()
    InfoAllNIC = []
    for nic, addrs in psutil.net_if_addrs().items():
        if nic in NetSpeed:
            st = NetSpeed[nic]
            diccNic ={
                "NIC": nic,
                "Speed": st.speed
            }
            InfoAllNIC.append(diccNic)

    # Stats Battery
    battery = psutil.sensors_battery()
    dicc = {
                    "ID":self.get_MY_ID(),
                    "MAX_FREQUENCY(MHZ)": round(cpufreq.max, 2),
                    "MIN_FREQUENCY(MHZ)": round(cpufreq.min, 2),
                    "CURRENT_FREQUENCY(MHZ)": round(cpufreq.current, 2),
                    "TOTAL_CPU_USAGE(%)": psutil.cpu_percent(),
                    "MEMORY_PERCENTAGE(%)": svmem.percent,
                    "NETWORK_SPEED(MB)": InfoAllNIC,
                    "BATTERY_PERCENTAGE(%)": battery.percent
                }
    return dicc
```

Como se puede apreciar en dicho código, el slave obtiene la información de la función *__get_Info_From_Slave()* y la pública en el topic especificado anteriormente. Entrando en un poco más de detalle en la función *__get_Info_From_Slave()*, en dicha función se busca dar la información que se desee de la máquina específica. En este caso se escogieron datos como el uso de la cpu, batería de la máquina, etc

Métodos para obtener información del nodo. Esta funcionalidad es distinta a las anteriores, ya que hace referencia a métodos de la clase, en vez de activarse con eventos de mqtt. Esta funcionalidad hace referencia a las gestiones que realizará el usuario sobre el slave, principalmente para ver el estado del slave y ver cuales son los topics de suscripción y publicación del contexto de aplicación.
Los métodos creados son los siguientes:

get_Stages(), **get_Pipelines()**, **get_MY_ID**, **get_IP_SERVER()**, **get_PORT()**, **get_Previous_Node()** y **get_Previous_Node()**, son getters sencillos para consultar las distintas variables, como por ejemplo, el número de pipelines, el id, los distintos stages, los nodos adyacentes, etc.

get_Topics_Subscribe() y get_Topics_Publish(). Estos métodos especifican a quien se suscribe en el contexto de aplicación y a quien publica, el primer método es irrelevante ya que un nodo slave, en el contexto de aplicación, se suscribirá siempre a sí mismo, el código del método **get_Topics_Publish()** es el siguiente:

```
# ============================
# get_Topics_Publish
# ============================
# This funcion return the topics to which the slave publishes
# Param in ->
# None
# Param out ->
# _Topics_Subscribe: List with Topics Publish
def get_Topics_Publish(self):
    _Topics_Publish = []
    for i in self.get_Next_Node():
        _Topics_Publish.append("/APPLICATION_CONTEXT/ID-" + str(i))
    return _Topics_Publish
```

Como podemos observar, en este método, se devolverá un lista con el/los topics donde dicho slave, en el contexto de aplicación, deba publicar. Esto es posible, gracias a la existencia del método *get_Next_Node()* donde encontramos el estado de cual es el siguiente nodo.

6.2.1.3. Script python MasterControl.

Para poder usar y comprobar el funcionamiento de la librería es necesario realizar un pequeño programa que la utilice, por eso, se creó el script de python **MasterControl** donde se utiliza dicha librería creando una clase **Master** y utilizando **sus métodos** para controlar el funcionamiento de la red. En el siguiente diagrama se presenta qué funcionalidades debe cumplir dicho script:

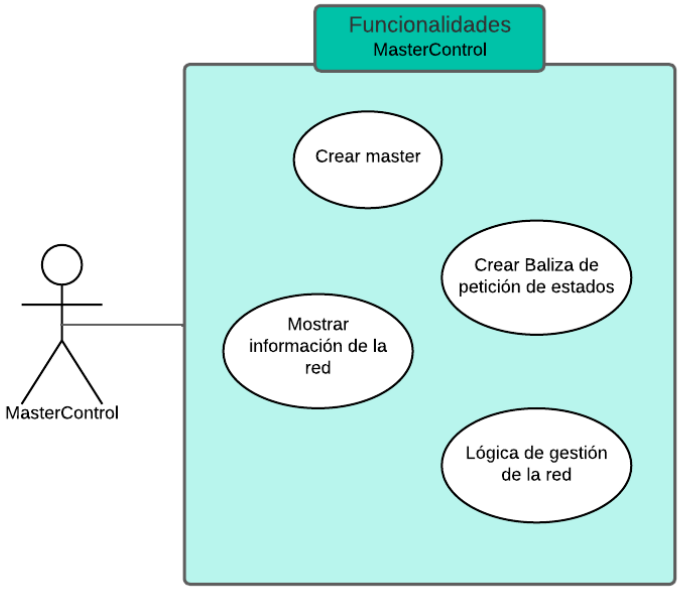

Imagen 6.2.1.3.1: diagrama de las funcionalidades de MasterControl

A continuación se explicará por partes el código utilizado para alcanzar las funcionalidades vistas en el diagrama anterior.

En el siguiente código se pueden apreciar las librerías utilizadas, como por ejemplo la llamada a la librería **STRPLibrary**, donde se encuentra la clase máster que posteriormente se utiliza.

También se pueden encontrar variables globales que se utilizan durante todo el programa como por ejemplo los pipelines que se pretenden utilizar y cuantos stages se utilizan en el contexto de la aplicación.

Aparte de la **creación de la clase**, también se muestra algo de información inicial.

```
#################################################################################################################
#
#                       Code Example to use the Master Class (STRPLibrary)
#
#################################################################################################################
#

# Libraries
import paho.mqtt.client as libmqtt
from time import sleep
from _thread import start_new_thread
from random import randint
import STRPLibrary


# Global variables
IP_SERVER = "localhost"
PORT      = 1883
PIPELINES = 3
STAGES = 3
MAX_LIMIT_STAGE_DEAD = 3
N_NODES_TO_START_CHECK = 4

# Call the master class
test = STRPLibrary.Master(PIPELINES, STAGES, IP_SERVER, PORT)

# Initial print with some info
print("="*40, "Master, Initial Info", "="*40)
print("Number of Stages->" + str(test.get_Stages()))
print("Number of Pipelines->" + str(test.get_Pipelines()))
print("Structure->" + str(test.get_Structure()))
print("IP_SERVER->" + str(test.get_IP_SERVER()))
print("Port->" + str(test.get_PORT()) + "\n")
print("="*80)
print()
```

Para cumplir la funcionalidad de **Crear Baliza de petición de estados,** se creó el siguiente hilo que es paralelo a la ejecución principal del programa, que se limita a la invocación del método explicado anteriormente *request_info()*. Esto permite que podamos ajustar la cantidad de llamadas a los slave para pedir información del estado de cada máquina.

```
# Create a thread with a beacon for request info from slaves
def beacon_request_info():
    while(True):
        test.request_info()
        sleep(15)

start_new_thread(beacon_request_info,())
```

Dentro del loop main principal, encontramos la **muestra de la información de la red** y la **lógica de gestión de la red**.

En la siguiente imagen encontramos el código que muestra información de la estructura de control de red, mostrando los pipelines con sus ids y sus respectivos stages.

```
# Print with some info (structure, status, etc)
print("="*40, "Master, I'm living", "="*40)
structure = test.get_Structure()
for i in range(len(structure)):
    print("Pipeline:" + str(i))
    for w in structure[i]:
        print("ID " + str(w['ID']) + " has the following stages " + str(w['List_Stages']))
    print()
```

En el siguiente código se muestra también la información recibida del estado de cada máquina (slave), con su respectivo id. Se controla también si cada slave está vivo, comprobando sus intentos de conexión. Esto quiere decir que si el contador está a menos de 5, indica que ha respondido desde hace al menos 5 peticiones, y por lo tanto se da por vivo. Si el contador está a 5 o más, indica que no ha respondido desde hace 5 peticiones y por lo tanto se da por muerto (5 es un ejemplo, se puede asignar lo que se desee a partir de las variables globales).

```
if len(test.get_Status_IDS()) != 0:# check if get_Status_IDS is empty
    for i in test.get_Status_IDS():
        if bool(i[1]):# check if status (inside ID) is empty
            print("*"*20)
            print("Status ID -> " + str(i[1]["ID"]))
            print("Connection attempt counter -> " + str(i[0]))# higher number means worse
            print("MAX_FREQUENCY -> " + str(i[1]["MAX_FREQUENCY(MHZ)"]) + "MHZ")
            print("MIN_FREQUENCY -> " + str(i[1]["MIN_FREQUENCY(MHZ)"]) + "MHZ")
            print("CURRENT_FREQUENCY -> " + str(i[1]["CURRENT_FREQUENCY(MHZ)"]) + "MHZ")
            print("TOTAL_CPU_USAGE -> " + str(i[1]["TOTAL_CPU_USAGE(%)"]) + "%")
            print("MEMORY_PERCENTAGE -> " + str(i[1]["MEMORY_PERCENTAGE(%)"]) + "%")
            print("NETWORK_SPEED ->")
            for w in i[1]["NETWORK_SPEED(MB)"]:# print info of all NIC
                print("NIC -> " + str(w["NIC"]))
                print("Speed -> " + str(w["Speed"]))
            print("BATTERY_PERCENTAGE -> " + str(i[1]["BATTERY_PERCENTAGE(%)"]) + "%")
            print("*"*20)
            print()
            if i[0] <= MAX_LIMIT_STAGE_DEAD: # id is live, check if it is working
                if len(test.get_stages_from_Id(i[1]["ID"])) != 0:# check if status is empty
                    if -1 in test.get_stages_from_Id(i[1]["ID"]):
                        new_stages = []
```

```
                        test.set_stages_from_Id(i[1]["ID"], new_stages)# id is ready to work again

            # Here would go the code based on the decision rules system
            # If slave dont response or we can improve the performance of the network
            else:# check if slave is dead, if true, the status for that id is deleted (5 trys)
                if len(test.get_stages_from_Id(i[1]["ID"])) != 0:# check if status is empty
                    if (-1 in test.get_stages_from_Id(i[1]["ID"])) == False:# check if status is
empty or working
                        test.delete_Id_stage(i[1]["ID"])
```

El siguiente código se centra en la lógica de control, añadiendo el control anterior del estado (vivo/muerto), de los slaves.

El código es el siguiente:

```
# Check if some status is not assigned, trying to assign to some id
for i in range(len(structure)):
    list_Stages_Without_Id = test.get_Stages_Without_ID(i)
    print("Status without Id for Pipeline " + str(i) + " -> " + str(list_Stages_Without_Id))
    if len(list_Stages_Without_Id) != 0:#try to assign status to empty id or merge with others
        for w in list_Stages_Without_Id:
            list_free_slaves = test.get_Free_Slaves(i)# check the free slaves
            print("Trying to assign stage to id -> " + str(w))
            print("There are the following slaves free -> " + str(list_free_slaves))
            n_free_slaves = len(list_free_slaves)
            if n_free_slaves > 0: # if there are some slaves, status is assigned to a free id
                test.set_Stage_To_Id(w, list_free_slaves[randint(0, n_free_slaves - 1)])
            else:
                # Decision to merge with other id with status
                # Example code
                if len(test.get_Status_IDS()) >= N_NODES_TO_START_CHECK - 1:
                    next_id = test.ID_Next_Stage(i,w)
                    if next_id != -1:
                        test.set_Stage_To_Id(w, next_id)
                    elif test.ID_Previous_Stage(i,w) != -1:
                        previous_id = test.ID_Previous_Stage(i,w)
                        test.set_Stage_To_Id(w, previous_id)
                    else:
                       print("Can't merge stages")
            print()
    else:# try to split status in differents id
        list_id_with_differents_id = []
        for w in structure[i]:# check if there are some id with different stages
            if len(w["List_Stages"]) > 1:
                list_id_with_differents_id.append(w)
        if len(list_id_with_differents_id) != 0:# check if it's empty
            list_free_slaves = test.get_Free_Slaves(i)# check the free slaves
            n_free_slaves = len(list_free_slaves)
            for id_value in list_id_with_differents_id:
                if n_free_slaves != 0:
                    while(len(id_value["List_Stages"]) > 1):
                        list_free_slaves = test.get_Free_Slaves(i)# check the free slaves
                        print("There are the following slaves free -> " + str(list_free_slaves))
```

```
                        n_free_slaves = len(list_free_slaves)
                        if n_free_slaves > 0: # if there are some slaves, status is assigned to a
free id
                            n_stage = id_value["List_Stages"][-1]
                            test.delete_Stage_To_Id(n_stage, id_value["ID"])# delete stage from id
                            test.set_Stage_To_Id(n_stage, list_free_slaves[randint(0, n_free_slaves
- 1)])# put stage to free id
                        else:
                            break
                else:
                    print("Don't exists slaves free -> " + str(list_free_slaves))
        else:
            print("Can't split stages")
```

Resumiendo el código anterior, se están realizando comprobaciones en la red, de tal forma que, a partir de un número de slaves que consideremos suficientes, la red se comenzará a regular. Esto lo que significa, es que en el caso de no tener suficientes slaves para asignar un stage a cada slave, se podrán fusionar stages que sean vecinos para cumplir que se realice todo el procesamiento.

También puede darse el caso contrario, si existen nodos slaves con distintos stage, el sistema intentará asignarle un nodo ocioso en el caso de que exista.

Con estas dos simples reglas se pueden controlar las caídas de los nodos, de tal forma que, si un nodo se cae, intentá asignarle los stages de dicho nodo a un nodo ocioso, y en el caso de que no exista, juntar los stages de dicho nodo a un nodo cercano.

Este es un ejemplo de reglas sencillas para controlar la red, pero aprovechando los estados de las máquinas recibidos por parte de los slave, se podría hacer un sistema mucho más complejo.

6.2.1.4. Script python SlaveXControl.

Para poder usar y comprobar el funcionamiento de la librería es necesario realizar un pequeño programa que la utilice, por eso, se creó el script de python **SlaveXControl** donde x representa los distintos scripts de cada pipeline, será necesario lanzar una mínima cantidad de slaves para el funcionamiento de la red, siendo este mínimo configurable en el master. Aparte en dicho código, se utilizan **sus métodos** para controlar el estado de dicho nodo (el id que tiene). En el siguiente diagrama se presenta qué funcionalidades debe cumplir dicho script:

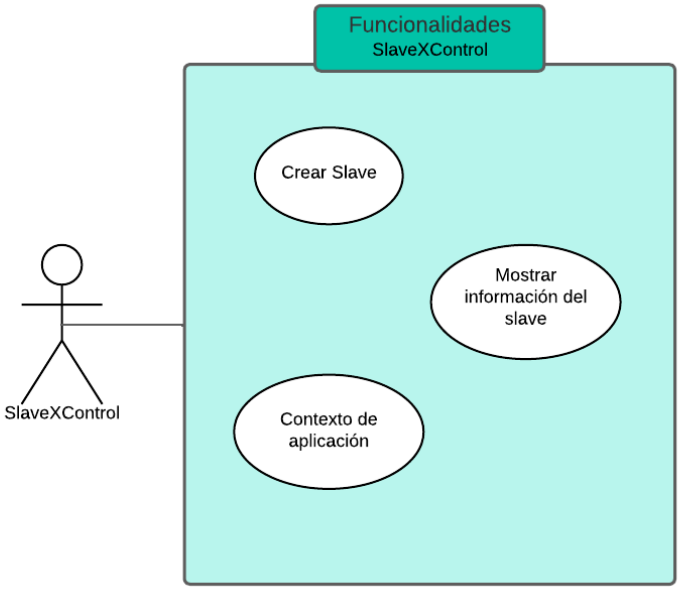

Imagen 6.2.1.4.1: diagrama de las funcionalidades de SlaveXControl

A continuación se explicará por partes el código utilizado para alcanzar las funcionalidades vistas en el diagrama anterior.

Se explicará el código en general, ya que para cada número que representa X (Slave0Control, Slave1Control, etc), la única diferencia es el pipeline al que pertenecen.

En el siguiente código se pueden apreciar las librerías utilizadas, como por ejemplo la llamada a la librería **STRPLibrary**, donde se encuentra la clase slave que posteriormente se utiliza.

También se pueden encontrar variables globales que se utilizan durante todo el programa como por ejemplo el pipeline al que pertenece y cuantos stages se utilizan en el contexto de la aplicación, entre otras variables globales.

```
#########################################################################################################
#                                                                                                       #
#                        Code Example to use the Slave Class (STRPLibrary)                              #
#########################################################################################################

# Libraries
import paho.mqtt.client as libmqtt
from _thread import allocate_lock, start_new_thread
from time import sleep
from random import randint
import re
import json
import math
import STRPLibrary
import time

# Global variables
# Server
IP_SERVER = "localhost"
PORT      = 1883
conex_to_Server= libmqtt.Client()
conex = False

# Lock
lock = allocate_lock()
lock.acquire()

# Message
message = None

# Event
event = 0
ESTABLISHED_CONEX = 1
FAIL_CONEX = 2
PUBLICATION_MADE = 3
ACTIVE_SUBSCRIPTION = 4
DATA_RECEIVED = 5

# Stage Number
N_STAGE = 3

# N Pipeline
N_PIPELINE = 0
```

Para cumplir la funcionalidad de **creación de la clase**, se realiza la siguiente llamada, también se añade algo de información inicial:

```
# Call the slave class
test = STRPLibrary.Slave(N_PIPELINE, IP_SERVER, PORT) # 0 is pipeline 0

# Initial print with some info
print("="*40, "Slave, Initial Info", "="*40)
print("My Id->" + str(test.get_MY_ID()))
print("My Pipeline->" + str(test.get_Pipeline()))
print("My Stages->" + str(test.get_Stages()))
print("IP_SERVER->" + str(test.get_IP_SERVER()))
print("Port->" + str(test.get_PORT()) + "\n")
print("="*80)
```

Hay que destacar que en la función de creación del slave, el programa se detendrá hasta obtener una id como se pudo apreciar dentro del constructor de la clase slave. Aparte se da algo de información como puede ser el id o el pipeline al que pertenece dicho slave.

Dentro del loop main principal, encontramos la **muestra de la información del slave**.

En la siguiente imagen encontramos el código que muestra información del slave, mostrando datos como el id, el pipeline, los stages y los nodos vecinos, aparte muestra el topic al que se suscribe y al que publica.

El código es el siguiente:

```
# Loop with the main program
while(True):
    # Print with some info (current stage, next id, before id, etc)
    print("="*40, "Slave, I'm living", "="*40)
    print("ID->" + str(test.get_MY_ID()))
    print("Pipeline->" + str(test.get_Pipeline()))
    print("Stages->" + str(test.get_Stages()))
    print("Previous_Node->" + str(test.get_Previous_Node()))
    print("Next_Node->" + str(test.get_Next_Node()))
    Topic = STR_APPLICATION_CONTEXT + "ID-" + str(test.get_MY_ID())

    print("Suscribe to ->")# print suscribers
    print(Topic)
    MQTT_TOPIC = [(Topic,0)]
    conex_to_Server.subscribe(MQTT_TOPIC)
    print()

    print("Current Publishers ->")# print publisher
    for i in test.get_Topics_Publish():
        print(i)
    print()
```

```
    # Check if i'm the first stage
    data_out=json.dumps([])
    if len(test.get_Stages()) != 0:#check if there are stages in this slave
        matches = [value for value in n_stage if value in test.get_Stages()]# Check the stage with
matches
        if len(matches) != 0:# check if its empty
            if 0 in matches:# check if 0 is inside the matches
                for stage in matches:# process all stage with matches
                    print("Processing Stage " + str(stage) + "...")
                    data_out = Stages(stage, data_out)
                if matches[-1] != N_STAGE - 1:#check if it's the last stage
                    if len(test.get_Topics_Publish()) != 0:# check if there are some topic to
publish
                        conex_to_Server.publish(test.get_Topics_Publish()[-1], data_out)#the final
result is published
                else:
                    STR_N_PIPELINE = "PIPELINE-" + str(N_PIPELINE)
                    topic = STR_APPLICATION_CONTEXT + STR_N_PIPELINE + "/RESULT"
                    conex_to_Server.publish(topic, data_out)#the final result is published

    print("="*80)
```

Aparte de la información se añade un pequeño código que comprueba si este slave contiene el stage 0, ya que el funcionamiento es algo distinto, ya que no se suscribe a alguien. Si es el caso, dicho slave genera datos que publica cada cierto tiempo, para ello se comprueba si dicho slave contiene el stage 0, y si es el caso, realiza el contexto de aplicación de dicho stage.
Aparte se comprueba si contiene algún otro stage vecino, para procesarlos todos sin enviar peticiones intermediarias que aumentan la carga de control de la red.

Finalmente, el SlaveXControl también debe tener un contexto de aplicación. Dicha aplicación hace referencia a el código donde se realiza el stream processing concreto sobre un problema, es decir, sería el código usado sin tener una parte de control.

Para ello, el Slave realiza una conexión Mqtt donde se gestionan los topics de dicho contexto de aplicación, la parte de control simplemente proporcionará a cada slave el topic donde debe publicar.

Para poder lograrlo, y obviando todo el código relacionado con la parte de conexión Mqtt, el código que hace referencia al contexto de aplicación es el siguiente:

```
elif event == DATA_RECEIVED:
    print (f"DATA_RECEIVED: {message.topic}={message.payload}")
    RegEx_Its_Me = "^"+ STR_APPLICATION_CONTEXT + "ID-" + str(test.get_MY_ID())
    if re.search(RegEx_Its_Me, message.topic): #if the topic match with the Regex Its me, check
stages and process
        n_stage_without_firts_stage = n_stage[1:]
        data_out = message.payload
        matches = [value for value in n_stage_without_firts_stage if value in test.get_Stages()]#
Check the stage with matches
        if matches != 0:# check if its empty
            for stage in matches:# process all stage with matches
                print("Processing Stage " + str(stage) + "...")
                data_out = Stages(stage, data_out)
            if matches[-1] != N_STAGE - 1:#check if it's the last stage
                if len(test.get_Topics_Publish()) != 0:# check if there are some topic to publish
                    conex_to_Server.publish(test.get_Topics_Publish()[-1], data_out)#the final
result is published
            else:
                STR_N_PIPELINE = "PIPELINE-" + str(N_PIPELINE)
                topic = STR_APPLICATION_CONTEXT + STR_N_PIPELINE + "/RESULT"
                conex_to_Server.publish(topic, data_out)#the final result is published
```

Cuando un slave recibe un conjunto de datos haciendo referencia al contexto de aplicación, dicho slave comprueba los stage que debe procesar, procesando un stage o varios. Si es el último stage, en vez de publicar su resultado al siguiente esclavo especificado, se envía al nodo encargado de realizar la conexión con la nube.

Para poder realizar el procesamiento de cada stage, se realiza un función Stage donde se encuentran todos los stage a procesar, de tal forma que especificando el número del stage, se puede realizar el stage concreto.
Dentro de dicha función nos podemos encontrar el siguiente código:

```
# Stages examples
def Stages(stage, message):
    global N_PACKAGE
    # Stage 0, create 100 rand numbers
    if stage == 0:
        start_time = time.time()
        buffer_length = 100
        buffer = []
        #Nº Package
        buffer.append(N_PACKAGE)
        #Start count
        buffer.append(start_time)
        N_PACKAGE = N_PACKAGE + 1
        for i in range(0, buffer_length):
            buffer.append(randint(0, 1000) / 100.0)
        data_out = json.dumps(buffer)
        return data_out
```

En el código anterior se muestra el **procesamiento del stage 0**, en este contexto de aplicación de prueba se realizó la generación de 100 números aleatorios, para su posterior procesamiento.

```
# Stage 1, create mean with the 100 before numbers
elif stage == 1:
    data_in = json.loads(message)
    mean = 0.0
    for n in data_in[2:]:
        mean += n
    mean = mean/(len(data_in)-2)
    print("N package")
    print(data_in[0])
    print("Mean")
    print(round(mean, 2))
    data_in.append(round(mean, 2))
    data_out = json.dumps(data_in)
    return data_out
```

En el código anterior se muestra el **procesamiento del stage 1**, en este caso se realizó la media de los datos obtenidos del stage anterior.

```
# Stage 2, create variance with the mean and the 100 before numbers
    elif stage == 2:
        data_in=json.loads(message)
        mean = data_in[-1]
        data_in.pop()
        n_package_last = data_in[0]
        data_in.pop(0)
        n_time = data_in[0]
        data_in.pop(0)
        variance = 0
        for data in data_in:
            variance += math.pow((data - mean), 2)
        variance = variance/(len(data_in))
        data_in.clear()
        print("N package")
        print(n_package_last)
        data_in.append(n_package_last)
        print("Time elapsed")
        time_elapsed = round(time.time() - n_time, 4)
        print(time_elapsed)
        data_in.append(time_elapsed)
        print("Mean")
        print(mean)
        data_in.append(mean)
        print("Variance")
        print (round(variance, 2))
        data_in.append(round(variance, 2))
        data_out = json.dumps(data_in)
        return data_out
```

En el código anterior se muestra el **procesamiento del stage 2**, en este caso se realiza la varianza de los datos recibidos del stage anterior, aparte se eliminan los 100 datos ya que solo se almacena la media y la varianza, aparte de ciertos valores de control como el número del paquete.

6.2.1.5. Script NodeConexCloud.

Finalmente, se creó este script que permite la conexión del contexto de aplicación con una plataforma en la nube para la visualización de los datos, en este caso se utilizó ubidots.
Para poder lograr dicho propósito, se realizó la conexión de mqtt con el contexto de aplicación y la conexión de mqtt con ubidots. Obviando la conexión con mqtt, se realizó la subida de los datos procesados a la nube y para lograrlo se realizó el siguiente código:

```
elif event == DATA_RECEIVED:
            print (f"DATA_RECEIVED: {message.topic}={message.payload}")
            data_in=json.loads(message.payload)
            # message processing to send to the cloud
            for i in range(0, N_PIPELINES):
                STR_N_PIPELINE = "PIPELINE-" + str(i)
                topic = STR_APPLICATION_CONTEXT + STR_N_PIPELINE + "/RESULT"
                if message.topic == topic:

                    n_package = int(data_in[0])
                    time = data_in[1]
                    mean = data_in[2]
                    variance = data_in[3]

                    key_N_PACKAGE = STR_N_PIPELINE + "N_PACKAGE"
                    key_MEAN = STR_N_PIPELINE + "MEAN"
                    key_VARIANCE = STR_N_PIPELINE + "VARIANCE"

                    Dict = {
                        key_N_PACKAGE: {"value": n_package, "context": {"time_elapsed": time}},
                        key_MEAN: mean,
                        key_VARIANCE: variance
                    }

                    # publish results on ubidots
                    main(mqtt_ubidots_client, Dict)
```

Cuando dicho nodo recibe un mensaje del contexto de aplicación, lo procesa, dividiendo el paquete en media y varianza, se añadieron otros datos de control como el número del paquete o el tiempo que este ha tardado desde que empezó el proceso. Cuando se obtienen dichos datos, estos son transformados en un diccionario json, para así dentro de la función main creada, publicarlos dentro del mqtt de ubidots.

6.2.1.6. Script auxiliar.

Este es un script sencillo que se creó por comodidad, para poder lanzar todos los *archivos.py* del sistema en terminales, para evitar tener que lanzar cada script de python a mano en una nueva terminal. El nombre de dicho script es *start.py*.

El código es el siguiente:

```
# 1 MASTER
gnome-terminal --execute python3.8 MasterControl.py
sleep 1

# 4 SLAVES PIPELINE 0
gnome-terminal --execute python3.8 Slave0Control.py
sleep 1
gnome-terminal --execute python3.8 Slave0Control.py
sleep 1
gnome-terminal --execute python3.8 Slave0Control.py
sleep 1
gnome-terminal --execute python3.8 Slave0Control.py
sleep 1

# 2 SLAVES PIPELINE 1
gnome-terminal --execute python3.8 Slave1Control.py
sleep 1


# 3 SLAVES PIPELINE 2
gnome-terminal --execute python3.8 Slave2Control.py
sleep 1
gnome-terminal --execute python3.8 Slave2Control.py
sleep 1
gnome-terminal --execute python3.8 Slave2Control.py
sleep 1

# 1 NODECONEXCLOUD
gnome-terminal --execute python3.8 NodeConexCloud.py
sleep 1
```

6.3. Aportación a la comunidad científica.

Tras la realización del diseño y del código propuesto para este proyecto, en el siguiente enlace se presenta el repositorio donde se encuentra dicho código, para su libre uso y mejora del mismo.

El enlace al github donde se encuentra alojado todo el código del proyecto, es el siguiente:

https://github.com/Witiza99/TFM

7. Resultados y discusión:

En este apartado se comentan los resultados obtenidos frente a la metodología utilizada para solventar el problema. Se mostrarán a continuación un ejemplo de la implementación simulada y unas pruebas, para comprobar si realmente funciona la solución propuesta y los resultados obtenidos.

7.1. Ejecución implementación simulada.

Para poder entender mejor el funcionamiento del sistema se muestra un ejemplo de ejecución del mismo. Para poder llevarlo a cabo, primeramente se arranca el sistema, teniendo en cuenta que previamente se ha arrancado mqtt broker en nuestro dispositivo como se comentó anteriormente. Se ejecuta el siguiente comando para arrancar el sistema:

```
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/Proyecto_TFM/TFM_Solucion_Publ
icador_Suscritor_para_Edge_Fog_Stream_Processing_usando_Network_Pipelining/codig
o$ ./start.sh
```

Imagen 7.1.1: comando para arrancar el sistema

Una vez ejecutada dicha línea, ya se creará todo el sistema y no será necesario realizar ninguna acción más. Con la configuración propuesta de la red, se creará un master y un nodo de conexión a la nube, aparte se crearán 4 nodos slave para el pipeline 0, 2 slave para el pipeline 1 y 3 slave para el pipeline 2, por lo tanto en nuestro sistema habrá 3 pipelines, siendo el pipeline 0, el pipeline 1 y el pipeline 2.

Cabe destacar, que por el contexto de aplicación propuesto, se dispondrán de 3 stage, siendo la generación de datos, la media de los mismos y la varianza de dichos datos.

Cuando se ejecute dicho script, aparecerán distintas terminales, que comentan distinta clase de información, en las siguientes imágenes se podrán ir apreciando dichas salidas.

```
connecting...
ESTABLISHED_CONEX
ACTIVE_SUBSCRIPTION
======================================== Master, Initial Info ========================================
Number of Stages->3
Number of Pipelines->3
Structure->[[], [], []]
IP_SERVER->localhost
Port->1883

================================================================================
```

Imagen 7.1.2: inicialización del master

En la terminal del máster se puede apreciar la inicialización de la clase master

```
======================================== Master, I'm living ========================================
Pipeline:0

Pipeline:1

Pipeline:2

Status without Id for Pipeline 0 -> [0, 1, 2]
PUBLICATION_MADE
Trying to assign id to status -> 0
There are the following slaves free -> []

Trying to assign id to status -> 1
There are the following slaves free -> []

Trying to assign id to status -> 2
There are the following slaves free -> []
```

Imagen 7.1.3: estructura de la red mostrando el pipeline 0, terminal del master

Posteriormente, comentará en loop la estructura de la red, mostrando los pipelines y los status que faltan por asignar de cada pipeline.

El master mostrará continuamente la estructura, por lo tanto, hasta que no se introduzcan los distintos slaves, la salida será siempre la misma, en la siguiente imagen se mostrará el resultado obtenido, una vez se han inicializado los slaves.

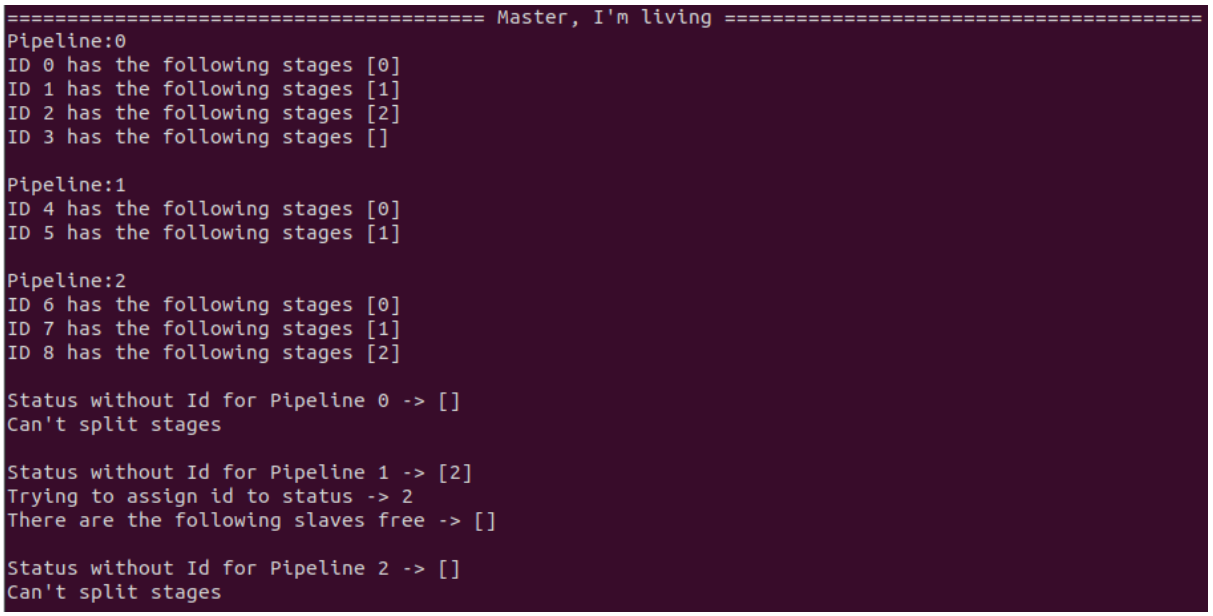

Imagen 7.1.4: estructura de la red tras la inicialización de los slaves

De la imagen anterior se puede extraer mucha información. Como se puede observar, se pueden apreciar todos los nodos slaves que se han inicializado y a que pipeline pertenece, aparte se muestran el stage que procesa cada slave.
También se puede apreciar que el Pipeline 0 tiene un nodo de reserva y que además todos sus slaves contienen su stage propio, esto no ocurre con el pipeline 1, ya que este le falta por asignar el stage 2. Cuando haya un número mínimo de nodos en la red, esta se reestructurará y dicho stage se asignará al nodo más cercano que pueda procesarlo. Por lo tanto, el master mostrará continuamente la estructura de la red, los nodos libres y los stage a asignar.

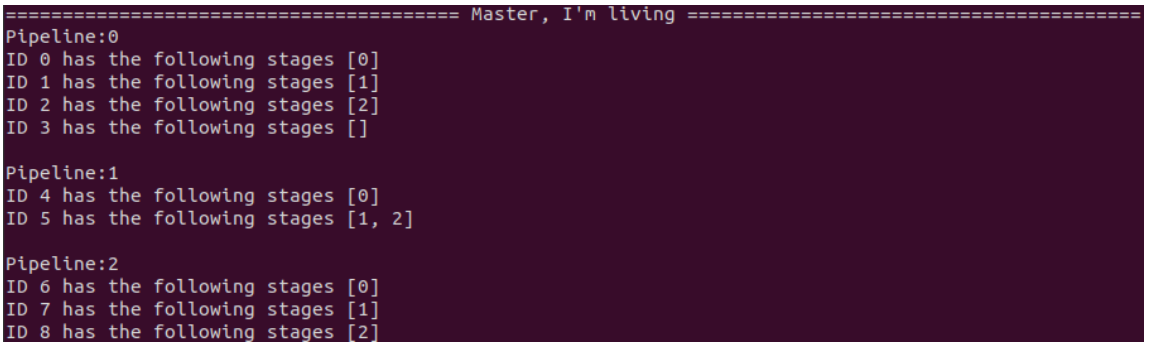

Imagen 7.1.5: estructura de la red tras la reestructuración

En la imagen anterior se puede apreciar como finalmente quedaría la red en el caso de no existir ningún cambio (como por ejemplo una caída de un nodo).

Cabe destacar que también aparecen mensajes de recepción y envío de mensajes de control, pero estos no son tan relevantes. Por último, el master muestra el estado de los slaves, en la siguiente imagen se muestra la información del nodo con id 0:

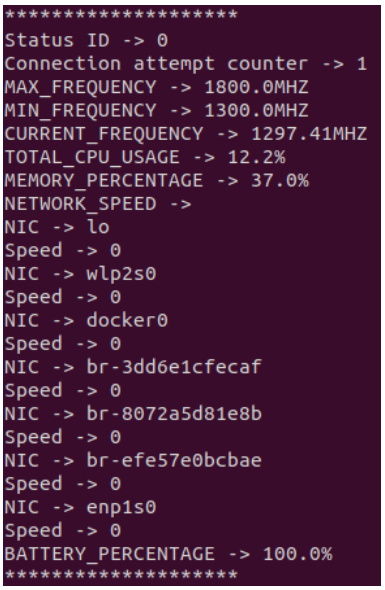

Imagen 7.1.6: estado del nodo con id 0

En dicha imagen se puede apreciar el id del nodo con el contador de intentos, es decir, las solicitudes que donde el master se intentó comunicar con él y este no le respondió, si el valor está entre 0 o 1, se puede considerar que el nodo funciona sin ningún tipo de problema. También encontramos otros datos relacionados con el estado de la máquina como puede ser la batería, frecuencia, datos relacionados con la red, etc.

Una vez visto las salidas de la consola del master, se pretende mostrar las salidas de los slaves, cómo son similares, se utilizará como ejemplo el nodo con id 0.

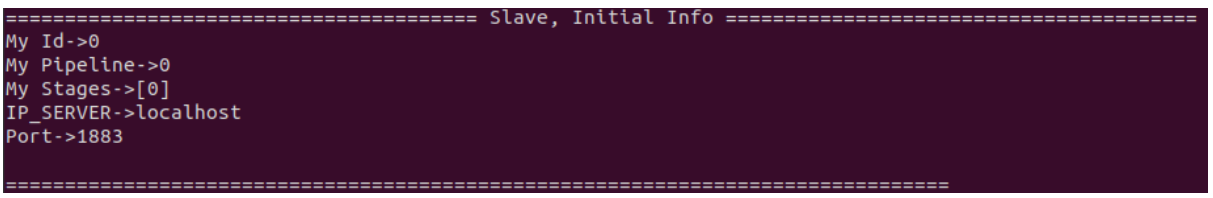

Imagen 7.1.7: estado inicial nodo slave

Tras realizar la conexión con mqtt e iniciar la clase slave, se muestran los datos iniciales, que en este caso será la id 0 que pertenece al pipeline0, y cuyo stage que tiene asignado es el 0.

Cuando esté slave se inicia, pide al master la información de los nodos conexos, cuando el máster le da dicha información, el slave se actualiza. Además como tiene asignado el stage 0 comenzará a realizar el procesamiento del mismo. En la siguiente imagen se puede apreciar:

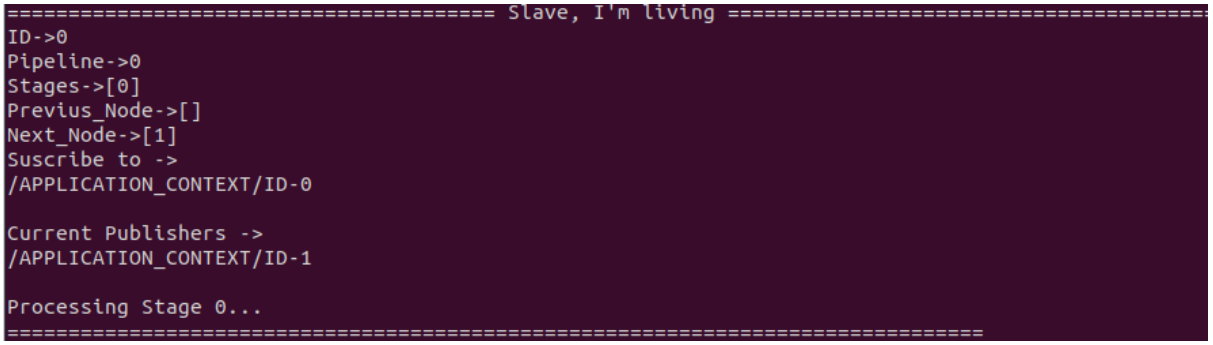

Imagen 7.1.8: estado final del slave con valores actualizados y procesando sus stages

Como se puede apreciar, dicho slave ya dispone de los nodos conexos, de tal forma que sabe a quién debe enviar el procesamiento de la etapa que realiza.

Por último, falta comentar la salida de la consola que hace referencia al Nodo de conexión a la nube. Este simplemente muestra información de cuando recibe un paquete y cuando lo publica, es decir, solo contiene mensajes de información, por lo tanto, para el usuario no es relevante.

Si se quieren observar los resultados de la ejecución, en ubidots se encuentran en un formato más legible.

Un ejemplo de resultados son los siguientes:

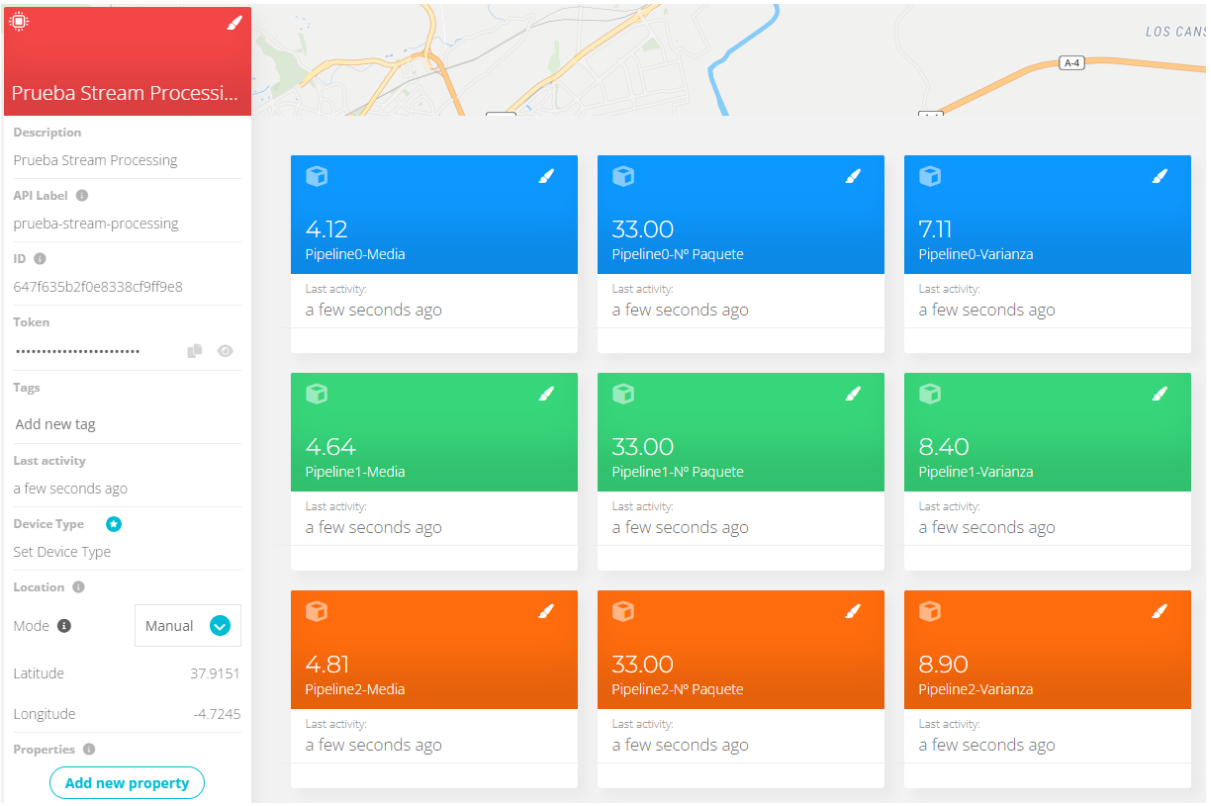

Imagen 7.1.9: resultados vistos desde ubidots

Como se puede apreciar, desde ubidots se pueden observar los datos de una forma mucho más clara.

En este caso, para los resultados de este contexto de aplicación, como se disponen de tres pipelines, se han escogido tres colores para representarlos, por lo tanto, el color azul representa las salidas del pipeline 0, el verde el pipeline 1, y el naranja, el pipeline 2.

Si nos centramos únicamente en el pipeline 0 (color azul), podemos encontrar tres tipos de datos, siendo los siguientes:

- **Pipeline0-Nº Paquete.** Hace referencia al número del paquete que ha llegado desde el pipeline 0.

- **Pipeline0-Media.** Hace referencia al número del paquete que ha llegado desde el pipeline 0.

- **Pipeline0-Varianza.** Hace referencia al número del paquete que ha llegado desde el pipeline 0.

De esta forma podemos observar la media y varianza calculadas durante el stream processing, de una manera sencilla, y ver a qué paquete de salida está asociado.
Cuando nos referimos al número del paquete, cuando se genera en el stage 0 los datos, a dichos datos se le asigna un número de paquete, por ejemplo, si es el segundo caso se le asigna el número de paquete 1. Cuando dicho paquete realice la media en el stage 1, y posteriormente la varianza en el stage 2, mantendrá dicho número, de esta forma podemos entender la procedencia de la media y la varianza de los datos obtenidos. Con el siguiente diagrama se puede entender de forma más sencilla.

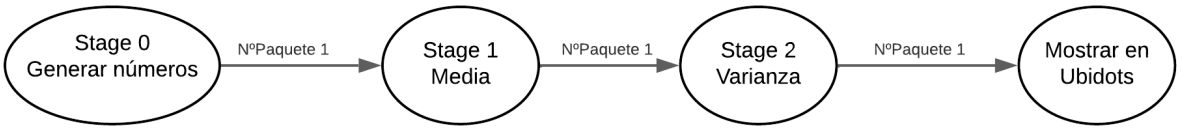

Imagen 7.1.10: desplazamiento del paquete 1 a través del stream processing

Este diagrama es un caso sencillo donde el paquete va de stage en stage, pero puede ocurrir que mientras el paquete 1 se encuentra procesando en el stage 2, el paquete 2 se encuentra ya procesando en el stage 1. Este diagrama no es un fiel reflejo de lo que sucede en un caso real, es más ilustrativo para el entendimiento de cómo se desplaza un paquete a través de los distintos stages hasta llegar a mostrarse en ubidots.

7.2. Pruebas.

En este subapartado se realizan pruebas para comprobar el correcto funcionamiento del sistema.
Primeramente se comprobará si la solución propuesta funciona correctamente, aplicándola incluso en otro contexto de ejecución más sencillo que permita comprobar el correcto funcionamiento del apartado de control con el uso de la librería, y la mínima robustez que debe tener el sistema frente a las caídas de nodos, pudiendo reestructurar la red.
Posteriormente, se le aplicarán otra serie de pruebas para comprobar la eficiencia de la solución y si es rentable u óptima para ser aplicada.

Para poder llevar a cabo los objetivos anteriores, se realizaron las siguientes pruebas:

P1 - Uso de otro contexto de aplicación. El contexto explicado anteriormente, realiza la media y la varianza de 100 datos, este contexto puede ser complicado a la hora de comprobar si funciona correctamente, por ende, se creó una copia del código donde se creó un nuevo contexto de aplicación sencillo, cuyo objetivo es comprobar el correcto funcionamiento del apartado de control, simplificando en la medida de los posible el contexto de aplicación, para ello se realizó el siguiente código:

```
# Stages examples
def Stages(stage, message):
    global N_PACKAGE
    # Stage 0, create iterator
    if stage == 0:
        buffer = []
        #Nº Package
        buffer.append(N_PACKAGE)
        N_PACKAGE = N_PACKAGE + 1
        buffer.append(1)
        data_out = json.dumps(buffer)
        return data_out

    # Stage 1, iterator + 1
    elif stage == 1:
        data_in = json.loads(message)
        iterator = data_in[-1]
        iterator = iterator + 1
        print("Iterator")
        print(iterator)
```

```
        data_in.pop()
        data_in.append(iterator)
        data_out = json.dumps(data_in)
        return data_out

    # Stage 2, iterator + 1
    elif stage == 2:
        data_in = json.loads(message)
        iterator = data_in[-1]
        iterator = iterator + 1
        print("Iterator")
        print(iterator)
        data_in.pop()
        data_in.append(iterator)
        data_out = json.dumps(data_in)
        return data_out

    # Stage 3, iterator + 1
    elif stage == 3:
        data_in = json.loads(message)
        iterator = data_in[-1]
        iterator = iterator + 1
        print("Iterator")
        print(iterator)
        data_in.pop()
        data_in.append(iterator)
        data_out = json.dumps(data_in)
        return data_out

    # Stage 4, iterator + 1
    elif stage == 4:
        data_in = json.loads(message)
        iterator = data_in[-1]
        iterator = iterator + 1
        print("Iterator")
        print(iterator)
        data_in.pop()
        data_in.append(iterator)
        data_out = json.dumps(data_in)
        return data_out
```

Como se puede observar en el código anterior, se han creado 5 stages donde, en cada uno de ellos, se incrementa el valor en uno. Esto permite comprobar si cada stage funciona correctamente, ya que el primer stage generará un 1, el siguiente stage añadirá 1 más y así hasta el stage 5, donde el resultado final deberá ser 5.

Como se puede apreciar, es un contexto de aplicación muy sencillo donde se incrementa por cada stage el valor inicial, además al disponer del número de paquete, se puede controlar que el paquete llegue como resultado.

Para comprobar el resultado obtenido, utilizamos ubidots como se explicó anteriormente. Antes de mostrar los resultados, la red creada y funcional es la siguiente:

```
Pipeline:0
ID 0 has the following stages [0]
ID 1 has the following stages [1]
ID 2 has the following stages [2]
ID 3 has the following stages [3]
ID 4 has the following stages [4]
ID 5 has the following stages []

Pipeline:1
ID 6 has the following stages [0]
ID 7 has the following stages [1]
ID 8 has the following stages [2, 3, 4]

Pipeline:2
ID 9 has the following stages [0]
ID 10 has the following stages [1]
ID 11 has the following stages [2]
ID 12 has the following stages [3]
ID 13 has the following stages [4]
```

Imagen 7.2.1: estructura de la red del testing

En este caso la red está formada por tres pipelines con 5 stage por pipeline, el pipeline 0 dispone de 6 slaves, pipeline 1 dispone de 3 slaves, y pipeline 2 dispone de 5 slaves.

El resultado obtenido en ubidots es el siguiente:

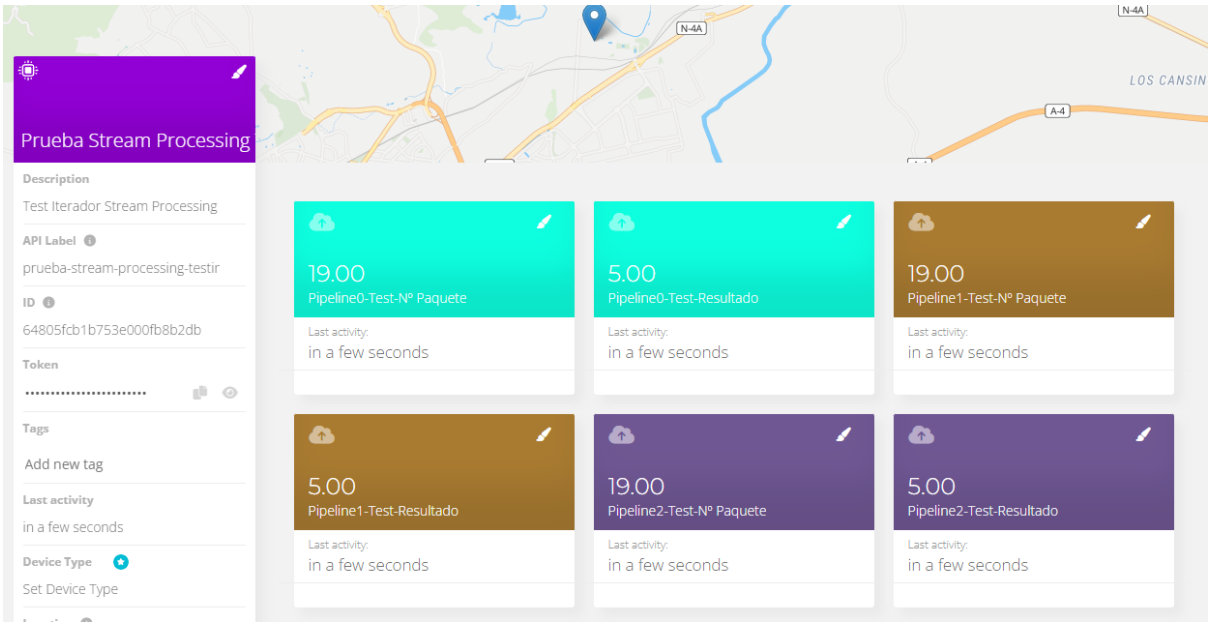

Imagen 7.2.2: resultados prueba de contexto de aplicación sencillo (1)

Como se puede observar en la imagen anterior, podemos ver que hay tres pipelines donde cada uno se puede apreciar con un color diferente, siendo el azul cian el pipeline 0, el marrón el pipeline 1, y el morado el pipeline 2.

Cada uno de ellos proporciona dos resultados, siendo estos, el número del paquete y el resultado obtenido, como se puede apreciar, siempre los resultados son 5, lo que quiere decir que el funcionamiento del sistema ha sido correcto. Continuamente se va incrementado los valores del número de paquete, ya que cada 30 segundos se envía un nuevo paquete al que se realiza el procesamiento.

En la siguiente imagen se puede apreciar como el número del paquete ha aumentado:

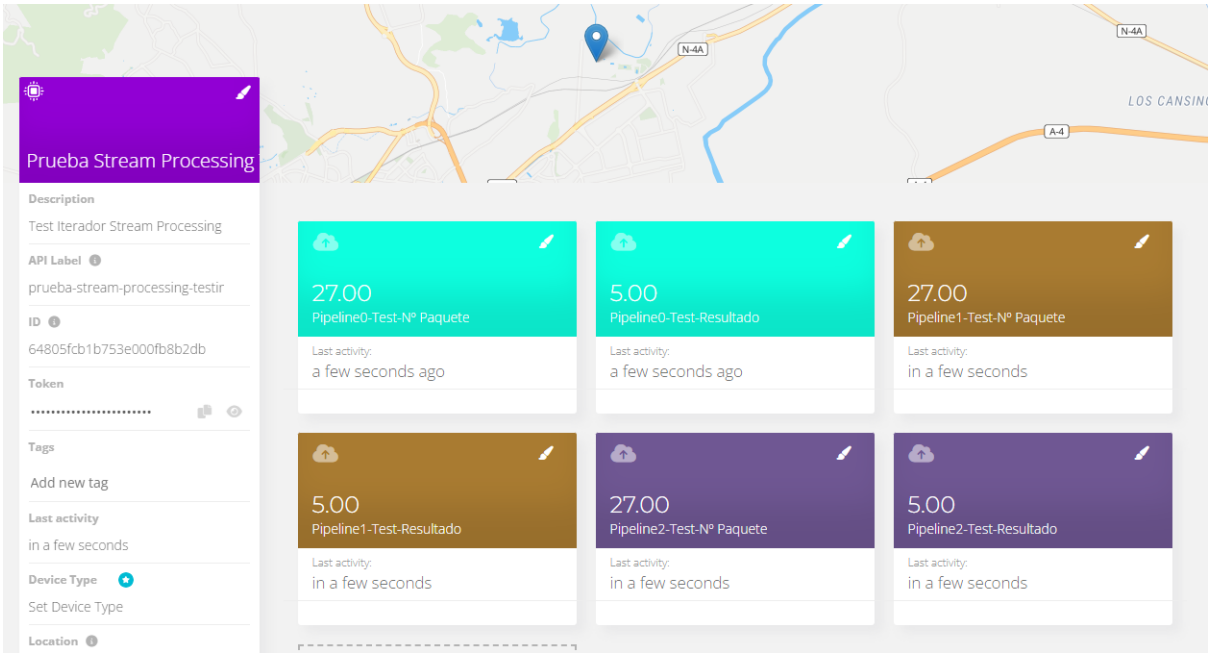

Imagen 7.2.3: resultados prueba de contexto de aplicación sencillo (2)

Como se puede apreciar el número del paquete aumenta, pero el valor resultado siempre se mantiene constante en 5, por lo tanto, el sistema está funcionando correctamente.

P2 - Robustez frente a caídas de nodos. Esta prueba se considera implícita dentro de la lógica que se haya implementado por parte del usuario, en el ejemplo realizado de master donde se usa la librería, se realizó una lógica sencilla donde el sistema reestructura la red, para que esta siga pudiendo realizar el stream processing.

Para ello, cuando un nodo deja de dar su estado, tras 5 intentos, se da por muerto. Es decir, se le avisa de que no va a seguir procesando y a los nodos vecinos se les avisa de la caída de dicho nodo.

Cuando un stage se encuentra libre, se le asigna primeramente, si es posible, el slave que se encuentre ocioso, en el caso de que no exista, se adjunta dicho stage al stage siguiente vecino, y en el caso de no ser posible, al stage previo.

De esta manera, siempre existe la posibilidad de que un slave realice todos los stage si no existe ningún otro disponible. Como esto es una sobrecarga para dicho slave, si existen a posterior un o varios slaves ociosos, se le van delegando los últimos stages a los siguientes slaves ociosos. Disminuyendo la sobrecarga de dicho slave.

Para comprobar que lo explicado anteriormente se cumple, sobre la red explicada en implementación simulada, se van a ir eliminando hilos de ejecución (slaves) para ver cómo se adapta la red, hasta que un slave contenga todos los stages. Finalmente se crearán nuevos nodos que permitan apreciar cómo se divide la carga de dicho slave.

- Estado inicial de la red:

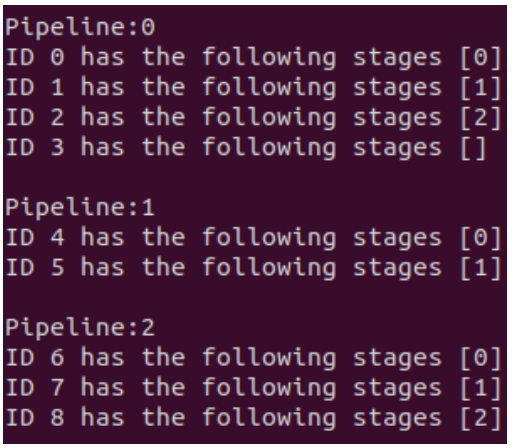

Imagen 7.2.4: estado inicial de la red de la P2

- Eliminación de primer slave, slave que contiene el stage 1:

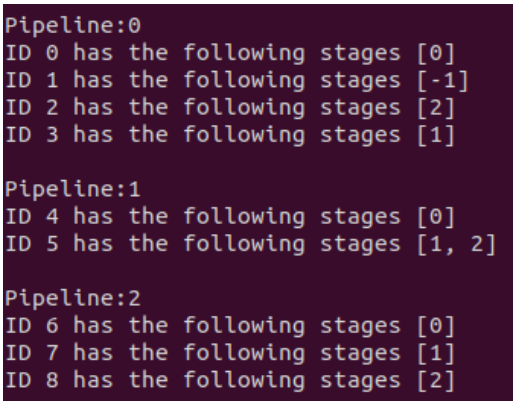

Imagen 7.2.5: estado de la red tras eliminar nodo 1 de la P2

De hecho se puede observar que los nodos vecinos se han ajustado al nuevo id apreciando las siguiente dos imágenes de los nodos vecinos:

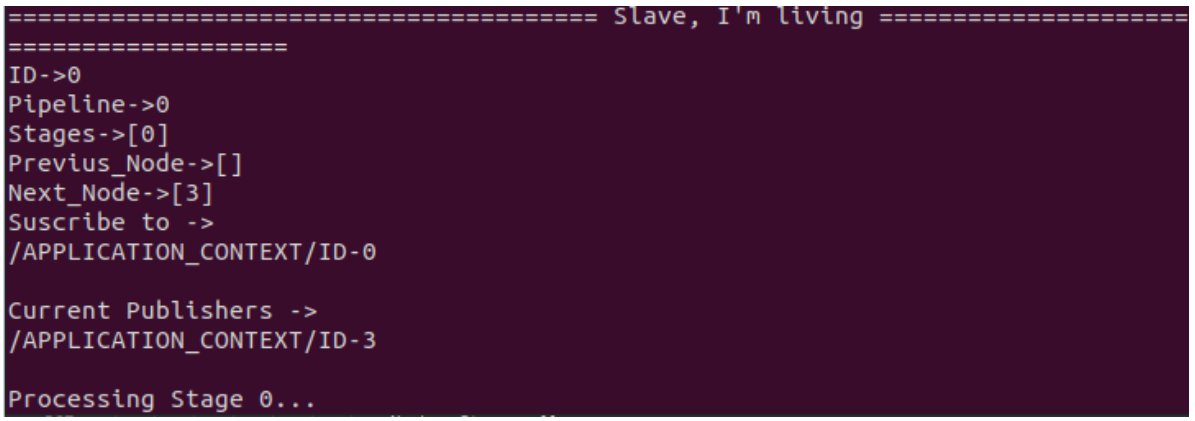

Imagen 7.2.6: vecinos del id 0 tras eliminación del id 1

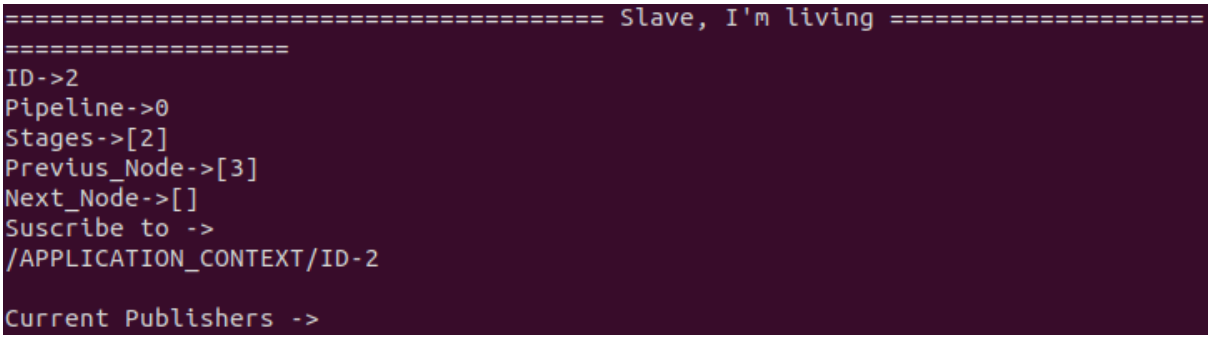

Imagen 7.2.7: vecinos del id 2 tras eliminación del id 1

- Eliminación de segundo slave, slave que contiene el stage 2:

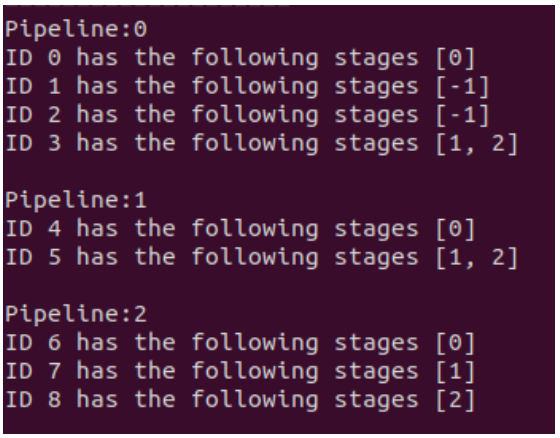

Imagen 7.2.8: estado de la red tras eliminación del id 2

De hecho se puede observar que los nodos vecinos se han ajustado al nuevo id apreciando, para ello se muestran las siguientes imagenes de los vecinos:

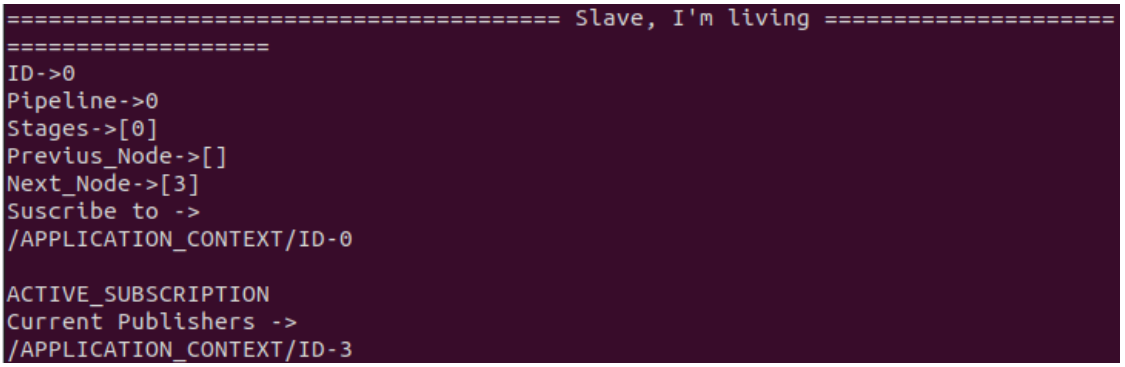

Imagen 7.2.9: vecinos del id 0 tras eliminación del id 2

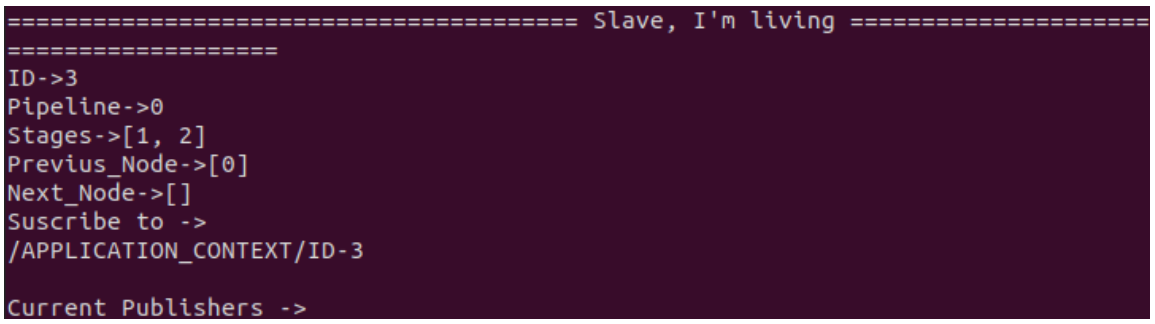

Imagen 7.2.10: vecinos del id 3 tras eliminación del id 2

De hecho, ya se puede apreciar cómo realiza el procesamiento de dos stages en el mismo slave, en la siguiente imagen se puede apreciar:

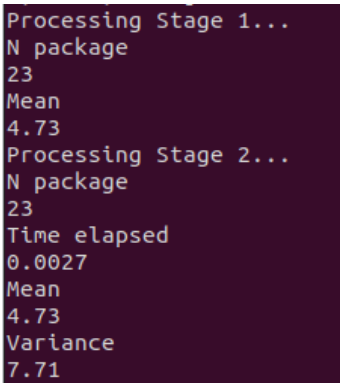

Imagen 7.2.11: procesamiento slave id 3

- Eliminación de tercer slave, slave que contiene el stage 3:

```
========================================= Master, I'm living ====================
====================
Pipeline:0
ID 0 has the following stages [0, 1, 2]
ID 1 has the following stages [-1]
ID 2 has the following stages [-1]
ID 3 has the following stages [-1]

Pipeline:1
ID 4 has the following stages [0]
ID 5 has the following stages [1, 2]

Pipeline:2
ID 6 has the following stages [0]
ID 7 has the following stages [1]
ID 8 has the following stages [2]
```

Imagen 7.2.12: estado de la red tras eliminación del id 3

El estado del slave con id 0 es el siguiente:

```
========================================= Slave, I'm living =====================
====================
ID->0
Pipeline->0
Stages->[0, 1, 2]
Previus_Node->[]
Next_Node->[]
Suscribe to ->
/APPLICATION_CONTEXT/ID-0

Current Publishers ->
```

Imagen 7.2.13: vecinos del id 0 tras eliminación del id 3

Como se puede observar, el slave 0 está realizando todo el procesamiento el solo, es decir, está realizando el stage 0, el stage 1 y el stage 2. Como lo que se busca con el stream processing es la repartición del procesamiento entre los distintos recursos, en cuanto exista un slave libre en el pipeline 0, se le asignará el último stage, para liberar carga del procesamiento al slave 0.

- Inserción de nuevo slave.

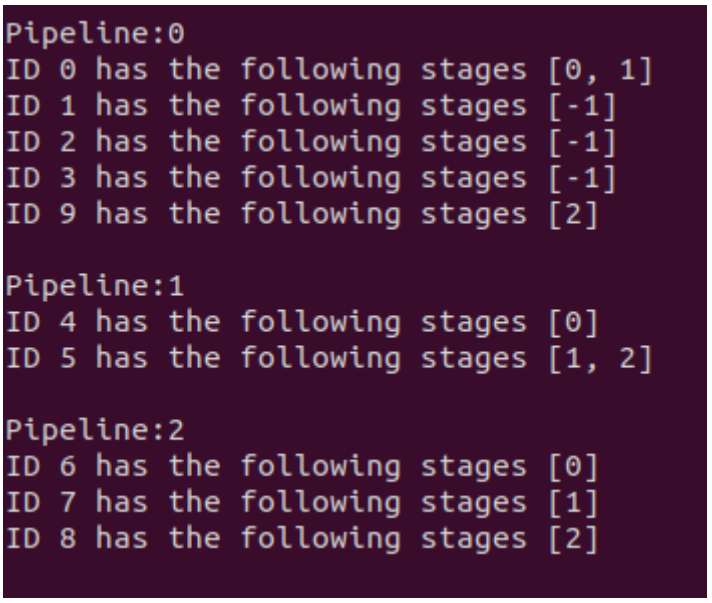

Imagen 7.2.14: estado de la red tras inserción de nuevo slave

Este proceso se repetirá continuamente hasta que se cada stage se encuentre en un único slave.

Esto no quiere decir que sea óptimo, ya que con un sistema de reglas basándose en los estados de cada máquina donde se encuentra cada slave, se podría optimizar el proceso de split y merge de stages.

Se realizó en esta prueba una lógica sencilla que permitiera cierta robustez en la red frente a la caída de nodos y como se puede ver en las imágenes anteriores, se consiguió cierta robustez.

7.3. Resultados:

Tras realizar un ejemplo de uso del sistema y comprobado la robustez del sistema frente a fallos (visto en el apartado anterior **7.2. Pruebas**), es necesario comprobar si realmente ofrece resultados notorios, es decir, es rentable la velocidad con la que trabaja el sistema y si la sobrecarga que se añade es notoria.
Para poder explicar todo este concepto de velocidad del sistema, es necesario basarse en algún tipo de métrica. En nuestro caso nos basaremos en las métricas de **Delay** y de **Throughput**.

Finalmente se comentarán los resultados obtenidos de dichas métricas para comprobar el funcionamiento del sistema y ver los resultados finales.

7.3.1. Delay.

Cuando nos referimos a la métrica de **Delay,** nos referimos al tiempo que tarda un paquete de datos desde que este se genera hasta que se envía a la nube. Este parámetro se utiliza para analizar el tiempo que tarda en procesar los datos el conjunto de stages, esto permitirá comprobar ciertos tiempos. Como por ejemplo, para distintos pipeline con distintos nodos, pudiendo tener un pipeline con un único dispositivo con todas los stages frente a un pipeline con distintos dispositivos con un único stage.

Un ejemplo visual de Delay sería el siguiente:

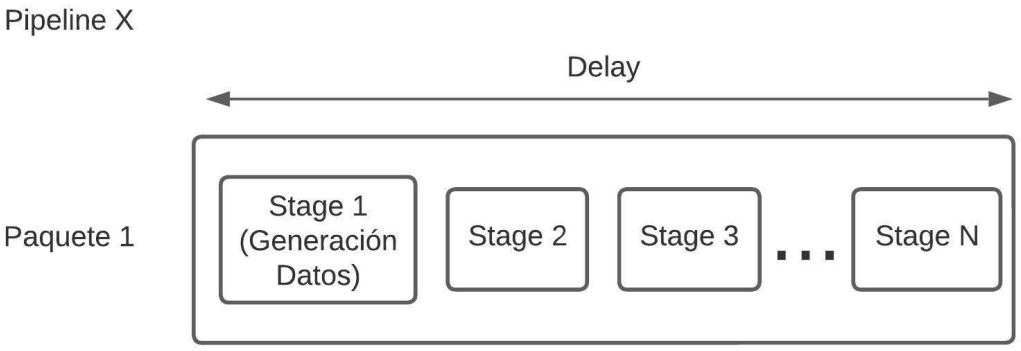

Imagen 7.3.1.1: Delay de un pipeline con sus diferentes stages de procesamiento para el paquete 1

En la imagen anterior se puede ver como para un pipeline X, se genera un paquete de datos y como este, debe procesarse por todas las etapas. Desde que se genera el paquete hasta que finalmente se envía el paquete procesado por todas las etapas a la nube, dicha franja de tiempo define el **Delay**.

7.3.2. Throughput.

Este parámetro también mide una unidad de tiempo, en este caso nos fijamos en los diferentes paquetes de datos generados. El **Throughput** se basa en la diferencia de tiempos generada entre salidas del pipeline, es decir, es el tiempo que tarda entre las salidas de los paquetes de datos datos.

Este parámetro es muy interesante para nuestro proyecto porque se basa en el concepto de stream processing y en pipeline networking, permitiendo así comprobar si existe mejoría al aplicarlos. Ya que cuando se produce una cola de procesamiento, se podría observar cierta mejora en dicho parámetro.

Para entender mejor este parámetro, se ha creado la siguiente imagen que permite un mayor entendimiento.

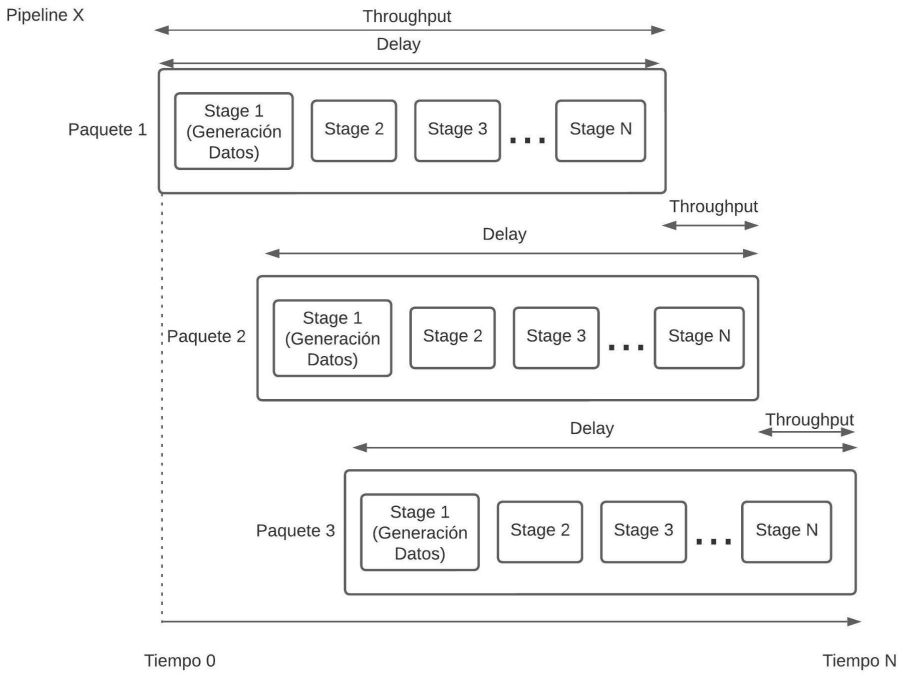

Imagen 7.3.2.1: Throughput de un pipeline junto al concepto de delay

Como se puede observar en la imagen anterior, para un pipeline determinado, el throughput es el tiempo con el que se recibe una salida, siendo esta salida un paquete de datos. Lógicamente, el primer paquete debería tener un throughput similar al delay o superior, ya que es la primera salida del pipeline y no existen paquetes anteriores de datos. Pero los siguientes paquetes, generarán throughput menor debido al concepto de procesamiento en pipeline, ya que se pueden procesar etapas de distintos paquetes de manera simultánea.

7.3.3. Discusión sobre los resultados.

Explicados los parámetros con los que se pretende trabajar, se realizan pruebas de velocidad de procesamiento sobre un pipeline (para mayor sencillez), generando la siguiente tabla con los resultados obtenidos. Para cada tabla se realizan 6 medidas, siendo los paquetes de 0 a 5, con sus respectivos delays y throughputs.

Cabe destacar que cuando se lanza el sistema por primera vez y se estructura la red, durante el tiempo que se está ajustando la red a los nodos de procesamiento disponibles, no se procesan datos, es decir, hasta que la infraestructura de red no está completa, no se comienza el procesamiento de los paquetes de datos. Dicho tiempo no influye en el Delay ni en el Throughput.

Pipeline con tres nodos y cada nodo con un stage:

Nº Paquete	Delay	Throughput
0	97.9748	97.9748
1	99.5592	57.7575
2	103.4542	61.978
3	103.8879	57.7149
4	100.9301	60.7902
5	101.8162	57.9822

Pipeline con dos nodos, el primer nodo con un stage, y el último nodo con dos stages:

Nº Paquete	Delay	Throughput
0	87.7147	87.7147
1	90.3438	57.8115
2	91.3767	59.6288
3	86.787	53.2553
4	88.694	59.9698
5	90.3777	57.8069

Pipeline con un solo nodo, dicho nodo contiene todos los stages:

Nº Paquete	Delay	Throughput
0	72.3517	72.3517
1	72.3735	72.4064
2	72.7274	72.7623
3	72.4742	72.5066
4	74.5678	74.6015
5	73.3515	73.3881

Antes de explicar los datos obtenidos, es necesario recalcar que las pruebas se han realizado sobre un solo pipeline ya que el uso de varios pipeline dificultará el entendimiento de los resultados y además, esto solo influiría en el tiempo de cómputo del máster, ya que tendría que gestionar más cantidad de llamadas entre dispositivos.

Tras observar los resultados de las pruebas realizadas, se confirma lo explicado en los apartados de Delay y Throughput. En el primer paquete de cada prueba realizada, el Delay y el Throughput son los mismos, esto se debe a lo explicado anteriormente. Durante todos los paquetes el Delay suele ser similar, mientras que el Throughput disminuye hasta una constante, siendo el punto donde se realiza procesamiento de diferentes etapas de paquetes de forma simultánea. Cabe decir que a veces estos tiempos aumentan, esto es lógico, ya que las pruebas se realizan sobre el mismo dispositivo, y el rendimiento del mismo disminuye.

Cabe destacar que para no saturar de mensajes y paquetes el sistema, se añadieron 30 segundos sintéticos de envío, es decir, que cada paquete de un 10000000 de datos se espera 30 segundos para ser enviado, dicho tiempo se añadió al tiempo que tarda la generación de los datos, y por tanto se incluye en el Delay, y también dentro del Throughput, ya que este es el tiempo entre salidas del pipeline. Recordemos que se están realizando tres stage, generación de los 10000000 datos, media de los datos, y varianza de los mismos.

Para comprobar cuánto afectan las comunicaciones, se realizaron 3 pruebas distintas, donde en cada una se limitan las comunicaciones agrupando stages.
Como se puede comprobar, a medida que se agrupan stages, los tiempos se disminuyen, hasta ser casi el mismo que el throughput. Esto es lógico, ya que estamos eliminando el tiempo que se utiliza entre las comunicaciones entre dispositivos. Pero esto limita la escalabilidad, ya que si realizáramos un mayor procesamiento con un mayor número de stages, el Delay aumentaría en gran medida, sin embargo, el throughput, primeramente sería similar al Delay, pero gracias al stream processing, reduciría los tiempos en los que se obtienen datos llegado a cierto punto.

Por ello, dependiendo del problema que vayamos a afrontar y del contexto de aplicación rentará usar o no dicha librería, no siendo recomendable si el problema es de una escala pequeña. Pero si dicho problema requiere de grandes cantidades de datos, con muchos stages y con gran cantidad de procesamiento, dicha librería puede ser muy útil para agilizar dicho procesamiento.

Tras analizar los resultados anteriores, puede no quedar del todo claro como el concepto de pipeline (y el concepto de paralelismo) beneficia el throughput. Por ello se diseñó un nuevo experimento conceptual simulado que permita entender en mejor medida dicha mejora.

El problema que se tiene al realizar un experimento en un entorno real o en el caso anterior, es la falta de control en los tiempos de los stage. Por ende, se creó el siguiente experimento donde:

- **Se crearán dispositivos dependiendo del número de stage**. Lo que significa que un stage implica un dispositivo, dos stages dos dispositivos, N stages N dispositivos.

- En total se probarán de **1 stage a 10 stages**, para comprobar si realmente aparece una mejora en el throughput, a medida que el pipeline crece.

- Se realizará una **media del Delay y throughput** de unos **10 paquetes**, para evitar fluctuaciones en los datos por algún problema con algún paquete.

- El tiempo total del procesamiento será de **10 segundos en total**, por lo tanto, cada stage dispondrá de un tiempo de **procesamiento de T/N**, siendo **T el tiempo total del procesamiento** y **N el número de stages** o dispositivos (debe ser el mismo). Como ejemplo, si disponemos de un stage, dicho stage dispone de un tiempo de procesamiento de 10 segundos, mientras que, si disponemos de 10 stages, el tiempo de procesamiento de cada uno será de un solo segundo.

Con el experimento que se pretende realizar, nos permitirá observar tanto la carga que se añade por las comunicaciones (al añadir stages) y la posible mejora del throughput, ya que al dividir el procesamiento en distintos stages, se puede procesar en paralelo distintas etapas, obteniendo un peor tiempo la primera vez, pero a partir de cierto punto, irá mejorando hasta el punto de converger por incrementar el coste de comunicación de la red.

Los resultados obtenidos de este experimentos se pueden encontrar en la siguiente tabla donde se muestra el Número de stages implementados, y la media del delay y throughput para 10 paquetes de datos:

Nº Stages	Delay	Throughput
1	10.00902	10.01010
2	12.40462	6.05360
3	12.08598	4.53731

4	13.99096	3.95452
5	14.16102	3.84481
6	12.2683	2.95212
7	13.77032	2.93390
8	14.05252	2.85523
9	17.12176	3.38337
10	16.73169	3.10566

Como se puede apreciar el la tabla con los resultados obtenidos, son esperables a lo comentado anteriormente. Al aumentar el número de stages, el delay va aumentando debido a el aumento de comunicaciones en la red, pero gracias al pipeline y al paralelismo que nos aporta, el throughput se reduce hasta converger en un mínimo donde la sobrecarga de las comunicaciones afecta, al throughput. También como vimos en el experimento anterior, cuando solo se trabaja con un stage, no existe pipeline, y por tanto, no existen comunicaciones, entonces el throughput y el delay son similares.

Por lo tanto, al realizar este experimento, podemos apreciar de una forma clara como el aumento hasta cierto punto del número de stages mejora el throughput, ya que se aplica de una manera más visual el paralelismo en el pipeline, asignando a cada stage un porcentaje de procesamiento.

Resumiendo ambos experimentos, en el primero se puede apreciar el funcionamiento de cada paquete y la ineficiencia de implementar dicha librería para una carga de procesamiento leve, ya que no es rentable por la sobrecarga en comunicación. Y en el segundo, la mejora que ofrece la librería frente a entornos con una carga de procesamiento más elevada, gracias a las ventajas que nos proporciona el paralelismo ofrecido por el pipeline y la división del procesamiento en stages.

8. Conclusiones y trabajo futuro:

Como se ha podido apreciar durante la documentación de este proyecto y más concretamente en el apartado anterior. Se podría decir que se han conseguido casi todos los objetivos marcados, de hecho, se van a revisar los objetivos de uno en uno para comprobar si se han cumplido.

En la siguiente lista se muestra un resumen de los objetivos y, de si se han conseguido lograr, añadiendo unas observaciones de los mismos.

- **OB-01, logrado.** Se ha conseguido desarrollar una metodología para Stream Processing reconfigurable en entornos locales distribuidos con dispositivos de baja capacidad de cómputo basado en Mosquitto para la interconexión entre dispositivos.

- **OB-02, logrado.** Se ha conseguido determinar la viabilidad a la hora de utilizar tecnologías de comunicación local basadas en el paradigma publicador-suscriptor para dar soporte al control de un Stream Processing Engine.

- **OB-02, parcialmente logrado.** Se ha conseguido evaluar el rendimiento del desarrollo propuesto en un caso de estudio real.

Respecto al objetivo **OB-01**, se puede decir que se ha cumplido sin ningún problema, de hecho, se puede comprobar con el apartado "**6. Metodología**", como se comenta de forma extensa todo este proceso.

Respecto al objetivo **OB-02**, se puede apreciar cómo se cumple si nos fijamos en el apartado "**7. Resultados y discusión**", donde se explica cómo se han realizado las pruebas que comentan el funcionamiento correcto y efectivo del sistema creado y una prueba del mismo. En concreto, en el apartado que trata la prueba de velocidad se comentan aspectos interesantes sobre qué se puede considerar rápido o incluso eficiente dentro de un contexto de aplicación.

Respecto al objetivo **OB-03**, podemos decir que se realizó parcialmente, ya que se consiguió a través de las pruebas, evaluar el desarrollo propuesto de la implementación simulada. Si se considera un caso de estudio real un entorno por ejemplo agrícola, se podría decir, que el contexto donde se generan 100 números y se realiza la media y varianza, hacen referencia a un sensor de temperatura, y por tanto, se podría considerar completado. Pero si se considera como caso de estudio real, una máquina real con un sensor como puede ser un esp32, no se habría cumplido este objetivo, por lo tanto, este objetivo se considera parcialmente completado por dichos motivos.

Por lo tanto, podríamos llegar a la conclusión de que se ha conseguido la solución de un Publicador-Suscriptor para Edge/Fog Stream Processing usando Network Pipelining.

Aun así cabe destacar que el sistema no es perfecto y que tiene mejoras a futuro, como pueden ser:

- Crear contextos de aplicación más variados y más complejos.
- Sistema basado en reglas para la logística del master.
- Optimización del árbol de topics e intercomunicaciones entre dispositivos.
- Mejora del dashboard de ubidots y mayor cantidad de personalización.
- Añadir más cantidad de datos de sensores distintos.
- Añadir capa de seguridad con la conexión en la nube.
- Completa división entre el apartado de control y el contexto de aplicación, usando diferentes brokers de MQTT
- Pruebas con tiempos de procesamiento diferentes en los stages, mejorando así el último experimento visto, y permitiendo unas pruebas con mayor profundidad.
- Etc.

Finalmente cabe comentar ciertas **conclusiones a nivel personal**, personalmente me encuentro bastante satisfecho con el trabajo conseguido, lógicamente soy consciente de que siempre se puede llegar a lograr un poco más, pero los objetivos personales impuestos por el proyecto han sido parcialmente completados. Pudiendo aprender de un campo que no tenía experiencia previa (el campo de MQTT), aprendiendo también del uso de Ubidots y comprendiendo la relevancia y potencia que tiene el protocolo MQTT para el intercambio de información entre dispositivos IOT.

Por último, estar satisfecho de haber sido capaz de haber creado un código más o menos sencillo, dentro de lo posible**,** de tal forma que para terceras personas su uso sea relativamente fácil, gracias a los comentarios, y la estructura en clases dentro de una librería.

Bibliografía

[1] Karagiannis, V., & Schulte, S. (2020). Comparison of Alternative Architectures in Fog Computing. In 2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC). 2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC). IEEE. https://doi.org/10.1109/icfec50348.2020.00010

[2] Klinaku, F., Zigldrum, M., Frank, M., & Becker, S. (2019). The Elastic Processing of Data Streams in Cloud Environments: A Systematic Mapping Study. In Proceedings of the 9th International Conference on Cloud Computing and Services Science. 9th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0007708503160323

[3] Garcia, A. M., Griebler, D., Schepke, C., & Fernandes, L. G. (2023). Micro-batch and data frequency for stream processing on multi-cores. In The Journal of Supercomputing (Vol. 79, Issue 8, pp. 9206–9244). Springer Science and Business Media LLC. https://doi.org/10.1007/s11227-022-05024-y

[4] Samza. (n.d.). https://samza.apache.org. https://samza.apache.org/

[5] Apache Flink® — Stateful Computations over Data Streams. (n.d.). Apache Flink. https://flink.apache.org/

[6] Apache SparkTM - Unified Engine for large-scale data analytics. (n.d.). https://spark.apache.org/

[7] Zhao, H., Yao, L., Zeng, Z., Li, D., Xie, J., Zhu, W., & Tang, J. (2020). An edge streaming data processing framework for autonomous driving. In Connection Science (Vol. 33, Issue 2, pp. 173–200). Informa UK Limited. https://doi.org/10.1080/09540091.2020.1782840

[8] Gupta, H., & Ramachandran, U. (2018). FogStore. In Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems. DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems. ACM. https://doi.org/10.1145/3210284.3210297

[9] Yin Liao, Guang-Zhong Sun, & Guoliang Chen. (2009). Distributed Pipeline Programming Framework for State-Based Pattern. In 2009 Eighth International Conference on Grid and Cooperative Computing. 2009 Eighth International Conference on Grid and Cooperative Computing (GCC). IEEE. https://doi.org/10.1109/gcc.2009.11

[10] IBM Developer. (s. f.). https://developer.ibm.com/podcasts/ibm_developer_podcast/034-mqtt-inventor-uk-ireland-cto-andy-stanford-clark/

[11] Vitorino, J. P., Simão, J., Datia, N., & Pato, M. (2023). IRONEDGE: Stream Processing Architecture for Edge Applications. In Algorithms (Vol. 16, Issue 2, p. 123). MDPI AG. https://doi.org/10.3390/a16020123

[12] Yao, L., Zhao, H., Tang, J., Liu, S., & Gaudiot, J.-L. (2021). Streaming Data Priority Scheduling Framework for Autonomous Driving by Edge. In 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE. https://doi.org/10.1109/compsac51774.2021.00017

Agradecimientos

A mi **familia** por apoyarme siempre durante esta etapa en todo lo necesario.

A **José Manuel Palomares Muñoz** por darme esta idea y la oportunidad de trabajar en un proyecto tan interesante, aparte de ayudarme con conceptos que desconocía.

A **Fernando León García** por ayudarme siempre que tenía algún problema relacionado con el código u otros recursos que necesitaba o conceptos que no entendía.