[1 - Características y objetivos del sistema 2](#_Toc1962196)

[2 - Implementación y decisiones de diseño 2](#_Toc1171452827)

[2.1 - Comunicación directa 2](#_Toc615851535)

[2.2 - Comunicación indirecta 3](#_Toc1199383370)

[3 - Versión final y metas cumplidas 3](#_Toc2003740058)

[4 - Preguntas teóricas 3](#_Toc391340015)

# 1 - Características y objetivos del sistema

Hemos diseñado dos sistemas de comunicación distribuida, uno de estos basados en una comunicación directa y el otro en una comunicación indirecta. Los principales objetivos de estos sistemas son recolectar datos de los diversos sensores disponibles, que detectan parámetros de la calidad del aire y de la contaminación (entre 2 y N sensores), y enviarlos a un conjunto de servidores que los procesan, generan una serie de coeficientes y los almacenan en una base de datos. Posteriormente estos datos se redirigen a cada uno de los terminales de los clientes del sistema y se muestran en tiempo real los datos obtenidos mediante una gráfica.

Las metas de ambos sistemas es poder llevar a cabo una comunicación entre diferentes máquinas mediante una comunicación directa (los componentes del sistema se conocen entre sí) y mediante una comunicación indirecta (los componentes del sistema se comunican a través de un intermediario) y que además estos sean lo más eficiente posible.

# 2 - Implementación y decisiones de diseño

## 2.1 - Comunicación directa

La comunicación directa entre los diferentes componentes del sistema se realiza mediante llamadas gRPC, las cuales se configuran en un archivo .proto que incluye los diferentes tipos de mensajes empleados en la comunicación y las diversas llamadas entre los clientes y los servidores.

**1 - Sensor -> LoadBalancer**

En primer lugar, abrimos un canal de comunicación con gRPC en un puerto determinado y creamos un stub con el cual nos comunicamos con el servidor (LoadBalancer) mediante un mensaje y la llamada implementada previamente. La frecuencia de recogida de datos que hemos establecido en los sensores es de 1 segundo.

**2 - LoadBalancer -> Servidor**

En primer lugar, abrimos tantos canales de comunicación como se hayan indicado por parámetro. Para evitar retrasos en la recogida de los datos hemos implementado la librería futures para poder convertir la comunicación en asíncrona, de tal manera que vamos atendiendo a todas las llamadas sin quedarnos bloqueados en ningún momento. En cuanto a la distribución de los datos a los múltiples servidores del sistema hemos implementado un mecanismo de Round Robin, que reparte equitativamente los datos.

**3 - Servidor -> Redis**

En primera instancia creamos tantos servidores como se hayan indicado por parámetro (creamos un nuevo proceso por cada servidor) y esperamos indefinidamente a su finalización. En cuanto a Redis, hemos decidido que la estructura de datos óptima es una tabla de hash, donde la clave es el timestamp de los datos recogidos y el valor es el coeficiente generado por el servidor. Cabe aclarar que tenemos dos tablas de Hash, una por cada tipo de coeficiente. Esto nos permite un acceso a los datos de tiempo O(1), ya que las consultas son inmediatas, por cada timestamp habrá tan solo uno o unos pocos valores asociados. Así pues, según la llamada recibida almacenamos los datos en una tabla o en la otra. Por último, destacar que hemos decidido que al principio de la ejecución del servidor se eliminen los datos de Redis, para evitar guardar valores históricos y solo almacenar los valores generades durante la sesión (así evitamos problemas con valores residuales de otras sesiones que pueden afectar al correcto funcionamiento del tumbling window).

**4 - Proxy -> Terminal**

Inicialmente abrimos tantos canales de comunicación como se hayan indicado por parámetro, que corresponde al número de terminales del sistema. Además, se le pasa un segundo parámetro que corresponde al tamaño del intervalo de los datos a tratar en el tumbling window.

Para implementar esto, obtenemos el primer timestamp almacenado en alguna de las tablas de hash. De esta manera nuestro intervalo es el siguiente [timestamp, timestamp + num\_sec]. Posteriormente obtenemos los coeficientes que se encuentran dentro de nuestro intervalo, calculamos la media de estos y la enviamos a los diferentes terminales.

Para evitar que el Proxy adelante a los sensores y los servidores en cuanto a la generación de todos los datos de un intervalo, hemos considerado esperar a obtener un coeficiente con un timestamp mayor que el intervalo actual. De esta manera, aseguramos que siempre calculamos la media una vez se han almacenado todos los coeficientes del intervalo.

En cuanto a los terminales, pasamos por parametro el número de terminal, para saber el puerto del canal que hay que abrir. Posteriormente creamos la gráfica correspondiente y la actualizamos a medida que vamos recibiendo las llamadas gRPC. Cada vez que ejecutamos una llamada actualizamos el tiempo de la gráfica y el valor correspondiente del coeficiente.

## 2.2 - Comunicación indirecta

**1 - Sensor -> RabbitMQ**

En primer lugar, se realiza la conexión con la cola de RabitMQ.

Posteriormente se empiezan a mandar los datos que se recogen indefinidamente cada 1 segundo.

Para saber si los mensajes que se acumulan en la cola son de un sensor o de otro al inicio del mensaje añadimos un identificador. Realizando esto el servidor podrá diferenciar los tipos de datos y analizar cada uno como se debe.

**2 - RabbitMQ -> Servidor**

En un inicio se realiza la conexión con la cola de RabitMQ donde los sensores están depositando los datos.

Seguidamente cuando se recibe algún dato, este se analiza y se guarda el resultado en Redis. Este se guarda en una tabla de hash paro poder tener el acceso a los datos de una manera más rápida.

En el código de servidor se implementan varios servidores a la vez con múltiples procesos. De esta manera se le pasa por parámetro cual es el número de servidores que se desea tener en una misma ejecución. De tal manera que no haya que ejecutar múltiples instancias del código.

**3 - Servidor -> Redis**

En el proxy la primera cosa que se realiza es la conexión con la cola de RabitMQ. Posteriormente de igual forma que en la comunicación directa espera a tener todos los datos del intervalo en el que se esté para poder continuar. Cuando ya se tienen los datos, se calcula la media del intervalo y se pone el mensaje en la cola de RabitMQ para que le llegue a todos los terminales que estén conectados.

El mensaje que envía el proxy a la cola contiene un identificador al principio para diferenciar los tipos de dato que contiene.

Los parámetros que hay que pasarle al proxy en este caso serian solo el intervalo que se desea no hace falta indicar cuantos terminales hay.

**4 - Proxy -> Terminal**

El terminal es muy similar al de la comunicación directa, el cambio que hay seria en los parámetros, ya que no hace falta indicarle el número de puerto en el que esta, el terminal simplemente al principio se conecta a la cola de RabitMQ para empezar a recibir mensajes.

# 3 - Versión final y metas cumplidas

Respecto a las metas propuestas inicialmente, consideramos que hemos cumplido cada una de ellas. Hemos conseguido implementar un sistema de comunicación directa con llamadas gRPC para establecer la comunicación entre los diversos componentes del sistema. Además, hemos creado un LoadBalancer que distribuye de forma equitativa a los servidores los datos recogidos por los diferentes sensores. También destacar que hemos creado un sistema de almacenamiento de datos persisente con Redis en el que utilizamos diccionarios para guardar los datos.

En cuanto a la comunicación indirecta, hemos conseguido implementar una comunicación entre sensores y servidores mediante una cola de trabajo con RabbitMQ. Por lo que al Proxy y los terminales se refiere, hemos realizado una comunicación mediante RabbitMQ implementado un sistema basado en el patrón publicador/suscriptor que distribuye los datos de la cola a todos los terminales suscritos a ella.

# 4 - Preguntas teóricas

1. Frame all communication steps of the system based on the four types of communication (synchronous/asynchronous, pull/push, transient/persistent, stateless/stateful). Also include their pattern and cardinality (one-to-one, one-to-all...).

Comunicación directa:

**Sensor -> LoadBalancer**

**·** synchronous, push, transient, stateless

**LoadBalancer -> Servidor**

**·** asynchronous, push, transient, stateless

**Servidor -> Redis**

**·** synchronous push, persistent, statefull

**Redis -> Proxy**

**·** asynchronous, pull, transient, stateless

**Proxy -> Terminal**

**·** synchronous, push, transient, stateless

Comunicación indirecta:

**Sensor -> RabbitMQ**

**·** asynchronous, push, transient, stateless

**RabbitMQ -> Servidor**

**·** asynchronous, pull, transient, stateless

**Servidor -> Redis**

**·** synchronous push, persistent, statefull

**Redis -> Proxy**

**·** asynchronous, pull, transient, stateless

**Proxy -> RabbitMQ**

**·** asynchronous, push, transient, stateless

**RabbitMQ -> Terminal**

**·** asynchronous, pull, transient, stateless

2. Mention which communication type is more appropriate for each step and justify your decision in terms of scalability and fault tolerance.

**Sensor -> Servidor**

Para la comunicación entre los sensores i el servidor, sería mejor realizar la comunicación indirecta debido a que para ampliar tanto sensores como servidores, sería más fácil. Con la comunicación directa se podría realizar, pero puede llegar a ser más costoso, ya que habría que añadir el puerto del nuevo servidor en el load balancer, en cambio con la indirecta abres uno nuevo y ya tendrías la conexión realizada.

**Proxy -> Terminal**

Para la comunicación entre el proxy y los terminales, también sería más efectiva con la indirecta por el mismo caso. Si se añadiera un nuevo terminal con la directa tendrías que añadirlo a la lista que contiene el proxy, pero con la comunicación indirecta conectas el terminal a la cola y ya se podrá poner a leer los mensajes

3. Are there single points of failure in the system? How could you resolve them?

Un punto de falla serían las colas de RabbitMQ, ya que si estas fallan el sistema no podría mantener la comunicación entre los componentes.

Para solucionar esto se podrían tener varios nodos de RabbitMQ y balancear la carga entre estos, de tal forma que si falla un nodo habría varios más.

Otro punto de falla sería el servidor Redis, de tal forma que si este fallara se perderían todos los datos almacenados, y en consecuencia no se podrían leer para mostrarlo en los terminales.

Esto se podría solucionar teniendo varias instancias de Redis donde hubiera réplicas de datos para que si fallase uno hubiese otro servidor con los mismos datos, para realizar esto sería necesario tener un balanceo en la carga de los múltiples servidores de Redis.

4. Regarding system decoupling, what does a Message Oriented Middleware (MOM) such as RabbitMQ provide?

RabbitMQ ofrece un desacoplamiento temporal y espacial. Esto es porque los componentes del sistema se comunican entre sí mediante mensajes a través de una cola centralizada. Gracias a esto los componentes no tienen que conocer la ubicación exacta de los demás, tan solo tienen que saber la dirección de la cola centralizada donde se envían y reciben los mensajes. Por otra parte, no necesitan estar conectados al mismo tiempo para comunicarse. Estos envían los mensajes a la cola y los reciben cuando deseen.

5. Briefly describe Redis’ utility as a storage system in this architecture.

La utilidad de Redis en el sistema es que nos permite conseguir una persistencia de los datos y tolerancia a fallos. Además, al ser un sistema de almacenamiento en memoria de clave-valor nos posibilita estructurar los datos (recogidos por los diferentes sensores y computados por los servidores) de una forma óptima. Esto se debe a que utilizamos los timestamp de los datos como clave, la cual cosa nos permite emplear una tumbling window para calcular la media de los valores almacenados durante un intervalo determinado.