Informe Comparativo Tecnologías RabbitMQ y gRPC

Por Sebastián Alvarado y Felipe González.

La mensajería instantánea (conocida en ingles con las siglas IM) es una forma de comunicación en "tiempo real" entre 2 o mas personas basada en textos, los cuales son enviados a través de distintos sistemas de redes como por ejemplo el internet.

En el año de 1988, aparece el primer chat (considerado como el pionero en la mensajería instantánea), otorgando la posibilidad de interactuar con diferentes personas en tiempo real, su nombre era "Internet Relay Chat (IRC)", el cual fue creado y programado por Jarkk Oikarinen o mejor conocido en la como "Wiz". Este funcionaba mediante el uso de "Chat rooms o canales" conectados por internet, permitiendo a todos los integrantes de la sala leer y escribir mensajes en tiempo real con todos los presentes, los cuales se identificaban utilizando "Nick names". En la actualidad existen diversas aplicaciones de comunicación, citando a algunos de los más conocidos como lo son "Whatsapp", "Instagram", "Skype", entre otros.

Si bien las tecnologías de la información han avanzado a pasos agigantados, aún existen problemáticas para generar estructuras de mensajería que sean totalmente funcionales y enfocados en los usuarios, por este motivo en este informe se comparará la utilización del bróker "RabbitMQ" (al cual nombraremos Rabbit de ahora en adelante) y el framework "gRCP"haciendo énfasis en su implementación por parte del proceso de desarrollo mas que por el desempeño que podrían generar. Finalizando con una recomendación técnica relacionada a la implementación realizada con "Python".

1. Problema

El "problema" a resolver en general es la construcción de una arquitectura de mensajería tipo *chat* que sea capaz de tener **n** clientes (usuarios) coordinados mediante un servidor central de coordinación de mensajes utilizando el protocolo RPC (Remote Procedure Call).

Esta arquitectura debe de cumplir con los siguientes requisitos de funcionamiento:

a. Tener un servicio de recepción y envío de mensajes.
b. Tener un servicio que pueda mostrarle a un cliente la lista de todos los que hay.
c. Tener un servicio para mostrarle al cliente todos los mensajes que ha enviado.
d. Mantener un log con todos los mensajes que pasen por el servidor.
e. Cualquier cliente debe poder enviar y recibir mensajes a través del servidor a otros clientes de forma *asíncrona* (que sea capaz de utilizar el servicio descrito en **a.**).
f. Cualquier cliente debe poder utilizar cualquiera de los servicios mencionados anteriormente.

Se dejaron de lado los requisitos específicos de implementación en este listado ya que explicarlos y enumerarlos haría más difícil la lectura de esta y tampoco se encuentra dentro de los objetivos de este informe el llegar a tal nivel de detalle.

2. RabbitMQ

RabbitMQ[1] es un bróker de mensajería que implementa un sistema de comunicación asíncrono mediante colas o “queues”, esto quiere decir que a través del uso de estas entrega un *soporte* para que las aplicaciones puedan comunicarse entre sí. El uso de colas implica que habrá un **productor** y un **consumidor**, el primero creará mensajes y los depositará en la **cola** para que luego el consumidor los lea de esta. La idea anterior se representa en la siguiente figura:

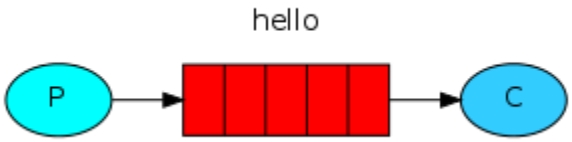

Figura 1: Esquema Productor – Cola – Consumidor

Notar que este modelo de mensajería cambia el esquema típico que se tiene en algunas aplicaciones en donde hay un servidor central que recibe los mensajes enviados y los re-envía al lugar especificado, en este caso podríamos pensar en la cola y su gestor, rabbit, como el servidor que distribuye la mensajería. En la práctica rabbit es un servidor que efectivamente gestiona las colas y su mensajería, con la salvedad de que es el desarrollador el que le tiene que pasar los mensajes de la manera adecuada para que este llegue a su destino de forma correcta.

2.1 Implementación

Para la implementación realizada en Python de la arquitectura se utilizó el ***Pika Python Client***, una librería que permitió el uso de rabbit en este lenguaje.

Para poder implementar la arquitectura y *utilizar* el protocolo RPC con rabbit se tuvo que cambiar un poco el modelo de la figura 1, ya que ahora en vez de tener solamente un consumidor y un productor, se tendrá un cliente y un servidor de coordinación que se comunicarán entre sí. Dado lo anterior y tratando de mantener el vocabulario empleado anteriormente, el cliente y el servidor se convertirán en productores y consumidores.

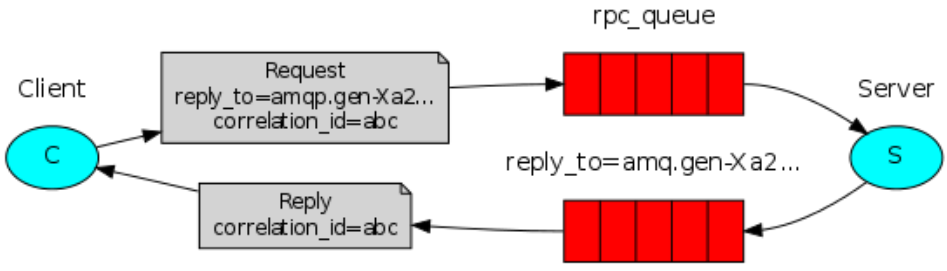

Figura 2: Esquema de la interacción ente cliente-servidor empleando colas.

La figura anterior muestra como fue la implementación realizada, se creó una cola por la cual el servidor está constantemente escuchando, y el cliente también tiene la suya por la cual le deben de ser respondidos los mensajes que este envía.

La manera de enviar y responder mensajes es mediante colas, pero para que la coordinación de estas funcione es necesario configurarlos de la siguiente manera:

- Para el cliente: tras realizar la conexión con rabbit, estos deben de crear una cola propia para que puedan recibir las respuestas por allí, entonces la manera de comunicarle al servidor de que mediante esta cola el cliente quiere recibir sus respuestas es enviándole en cada mensaje precisamente un parámetro que indique esto, en particular este se llama “reply_to” (responder a).

Es necesario que para mantener una buena comunicación entre ambas aplicaciones el uso de algo que confirme que los mensajes enviados fueron recibidos, es por esto por lo que se utilizan los “**acks**”, que es la abreviación para el inglés “acknowledgement”, confirmación o reconocimiento. Así, ante algún problema de pérdida del mensaje, se sabrá si es que se debe re-enviar o no el mensaje.

Otro detalle de implementación tiene que ver con la gestión de los mensajes recibidos por parte del cliente ya que algunos de los servicios que ofrece el servidor envía, para una solicitud, varias respuestas. Un ejemplo de lo anterior es lo que sucede cuando el cliente quiere ver sus mensajes enviados, es lógico que él reciba **n** mensajes del servidor donde n es la cantidad de mensajes enviados por el cliente a través de este:

- Para poder gestionar lo anterior se hizo uso de los acks de modo que al terminar de recibir el flujo total de mensajes, se envíe una confirmación de que todos se enviaron de manera correcta ya que si se realiza un ack por cada mensaje, con la implementación actual, podría llegar a suceder de que se confirma un mensaje aún no procesado y por lo tanto perderlo, ese y los que sigan.

3. gRPC

gRPC[2] es un framework para la implementación y utilización del protocolo RPC. Este utiliza los llamados “**protocol buffers**”[3] para poder definir los servicios y los métodos que se utilizarán en las aplicaciones cliente y servidor. Los ya mencionados *protocol buffers* son un tipo de archivo para serializar datos estructurados, básicamente lo mismo que hacen los famosos archivos .xml pero, según Google[3], de manera más compacta, simple y rápida.

3.1 Implementación

Gracias a los archivos **.proto** es que se pueden generar 2 códigos, a los cuales llamaremos pb2 de ahora en adelante, con los mensajes, servicios y métodos definidos en estos. El primero de estos es el Project_pb2.py (donde Project es el nombre del archivo .proto) que contiene las estructuras definidas en el .proto, el otro archivo es el Project_pb2_grpc.py que contiene las clases que deben ser extendidas para realizar las aplicaciones. No se dará un mayor énfasis en estos archivos dado que se generan automáticamente del archivo .proto.

A grandes rasgos las aplicaciones para ser implementadas y realizar su cometido, extienden las clases definidas en los *pb2*, utilizando los métodos que están en estas entregándoles como parámetros los *mensajes* con la estructura definida en el proto y proveída por los pb2, y retornando *mensajes* con la misma o distinta estructura dependiendo de como se haya definido en el archivo .proto.

Más específicamente, cada servicio (representado como clase) dentro del servidor tiene sus métodos, y si se desea que estos estén disponibles es necesario añadirlos al servidor con el método “Project_pb2_grpc.add_*ServiceName*Servicer_to_server”, donde *ServiceName* es el nombre del servicio. Ahora, para poder retornar los mensajes es necesario hacer un “return” del tipo de mensaje requerido por el

método según el archivo .proto definido, pero si se quisiera retornar una serie de mensajes seguidos, es decir, un "**stream**" de mensajes, es necesario hacer un "yield" dentro de un ciclo (while o for).

Otros detalles para el servidor incluyen que es necesario entregarle un threadpool cuando este es inicializado, y que se debe abrir un puerto manualmente.

Por otro lado, el cliente necesita abrir un canal con la dirección y el puerto del servidor, además requiere de la creación de "**stubs**", un sinónimo de cliente específico para un servicio, entonces si se necesitan utilizar los métodos de 3 servicios distintos dentro del servidor, se deben crear 3 stubs específicos para cada uno. Ahora, cuando se quiere realizar una solicitud al servidor, se realiza un proceso descrito más generalmente arriba, se realiza una petición de la forma "respuesta = Project_pb2_grpc.*MethodName*(mensajeRequerido)", entonces tras *formatear* el mensaje para hacer la solicitud, este se envía solamente al utilizar el nombre del método, y la respuesta será guardada en la variable *respuesta*.

4. Comparación de Implementación

Si bien ambas tecnologías se pueden implementar y se implementaron para resolver el problema planteado, si se vuelve a repasar la implementación de ambas se podrán apreciar claramente las diferencias específicas de cada una.

Rabbit utiliza mensajería mediante colas y gRPC utiliza los archivos pb2 para poder realizar el intercambio entre las aplicaciones. Lo que hace Rabbit es estructurar mensajes de una manera que se parece a los protocolos de la Capa de Transporte del internet, principalmente debido a que se utilizan *acks* de manera explícita, y al uso de una configuración que obliga al desarrollador a indicar a qué lugar debe ir el mensaje para que se pueda realizar el "*routing*". Por otro lado, gracias a lo que hace gRPC el desarrollador solo debe llamar a los métodos que requiera con una sintaxis que hace parecer que solo tuvo que importar una librería local para acceder a ellos, cuando en realidad estos pueden estar en máquinas en otro sitio del mundo ofreciéndolos de manera continua (online).

Una importante característica que se concluye de la implementación con Rabbit, es que las dos aplicaciones (cliente y servidor) son muy parecidas debido a que ambas terminan realizando las mismas operaciones de envío y recepción de mensajes, mientras que por su parte gRPC hace una clara distinción de lo que es un cliente y lo que es un servidor en todo sentido.

Para Rabbit el servidor solo necesita declarar su única cola de recepción de mensajes y a través de ella escuchará de a uno (ya que solo se tiene un programa servidor corriendo), y cuando le quiera responder al cliente deberá enviarle el mensaje mediante la cola privada de este. gRPC hace las cosas de una distinta manera, se tiene un solo canal de comunicación por donde se realiza la conexión directa, además el servidor necesita de un threadpool con el cual poder atender a distintos clientes para recién comenzar su ejecución.

Finalmente, ambas utilizan métodos o funciones para la recepción de mensajes, específicamente mediante los parámetros que se les entregan a estas. Rabbit utiliza una sola función para todos los mensajes, mientras que gRPC utiliza cada uno de los métodos definidos en el servidor para poder responder a los mensajes.

5. Comparaciones finales, Conclusión y Recomendación

Tras haber realizado toda una caracterización de las implementaciones y las diferencias entre las tecnologías, hay algunas conclusiones que se pueden obtener de todo ellas:

Rabbit parece ser la opción más flexible de las dos ya que esta permite la comunicación mediante mensajes directos que tienen un gran set de opciones y configuraciones disponibles, por lo que si se quisiera llegar a trabajar a un nivel más "bajo" con el manejo de mensajería, esta sería la mejor opción para ello. El problema que surge con esta tecnología es que necesita de una constante supervisión de la estructura y el contenido de los mensajes intercambiados, cosa que no ocurre con gRPC.

gRPC por su parte requiere un poco más de preparación inicial, pero tras haber superado esa barrera parece ser la opción más cómoda ya que el uso de las llamadas remotas mediante este framework es muy cómodo para un desarrollador. El problema que viene con esto es que habrá ocasiones en que no se sabrá si un método o una función estará implementada de manera local o global, lo que podría llevar a confusión y código spaghetti.

En general, en términos de escalabilidad ambos tienen problemas, Rabbit por una parte tiene el problema de manejor de una gran cantidad de colas y tipos de mensajes, y gRPC por su lado también tendrá problemas con el tipo de mensajes que se describan en su archivo .proto y se verá incrementado el nivel de posible desconocimiento de las cosas implementadas local o remotamente.

En conclusión, para implementaciones que requieren de un complejo manejo de mensajes casi a nivel de capa de transporte Rabbit es la solución, pero para todo lo demás es el framework gRPC el que parece ser el indicado para las tareas más genéricas a realizar, así que considerando todo lo relatado a lo largo de este informe, la recomendación es que gRPC es la mejor y más cómoda opción de las dos.

6. Referencias

1. Tutoriales y conceptos de Rabbit.

- https://www.rabbitmq.com/getstarted.html Visitado por última vez en 8.1.2020

2. Tutoriales y conceptos de gRPC en Python.

- https://grpc.io/docs/tutorials/basic/python/ Visitado por última vez en 8.1.2020

3. Guía para el desarrollador de protocol buffers.

- https://developers.google.com/protocol-buffers/docs/overview Visitado por última vez en 8.1.2020