Sistema de Mensajería Distribuído con gRPC y RabbitMQ

Informe Comparativo

Gonzalo Fernández C. Sebastian Godinez S. M.

1. Introducción

El presente documento explica las ventajas y desventajas encontradas luego de desarrollar un sistema de mensajería simple a través de 2 tecnologías distintas con el fin de compararlas, la primera es una implementación a través de llamadas a procedimientos remotos (RPC) mediante un servidor gRPC[1], la segunda en cambio, es una implementación que se sirve de un servicio de colas de mensajería RabbitMQ[2], generando en base a los resultados una completa recomendación técnica. La implementación tiene 2 modos de trabajo, uno en que los programas actúan de manera autónoma (0) y otro en que requieren un input manual del usuario (1), este último es el modo que se habla en este informe

2. Implementación en gRPC

La implementación del sistema de chat a través de gRPC se compone de 2 programas: un cliente y un servidor, los cuales se detallan a continuación:

2.1. Servidor:

- Al ejecutarse, inicia un servidor gRPC en la dirección entregada por consola, una vez que el servidor de escucha está listo, crea una estructura de datos que estará encargada de almacenar todas las conexiones gRPC de los clientes, así como también, los mensajes que envían y reciben.
- Una vez inicializado, agrega un cliente *Broadcast*, el cual, si alguien le envía un mensaje, el servidor se lo reenviará a todos.
- Cuando un cliente intenta logear, busca si su usuario se encuentra conectado para determinar si puede o no acceder, en caso de que no esté, le asigna una id única y procede a atender el resto de solicitudes.
- Cuando recibe un mensaje, le asigna un id único y luego revisa revisa a qué usuario lo debe entregar, para así agregarlo a lista de mensajes enviados del emisor y a la lista de mensajes recibidos de ese cliente, el cual está permanentemente escuchando por nuevos mensajes, en caso de ser la id del usuario *Broadcast*, el servidor repite este proceso para todos los usuarios. Luego, guarda en un archivo de texto `log.txt` cada mensaje del chat.
- Para las otras solicitudes, el servidor revisa en la estructura de datos donde guarda todo los datos que debe entregar y la retorna al cliente, a través de *Remote Procedure Calls*.

2.2. Cliente:

- Al ejecutarse, el cliente se conecta con el servidor en la dirección ingresada por la consola, luego cuando la conexión es establecida, procede a insertar un nombre de usuario con el fin de logear.

- Una vez logeado, el cliente guarda su id único y entonces ya está habilitado para hacer solicitudes al servidor
- Para enviar un mensaje, necesita el id del cliente a quién le escribirá, ésto se logra solicitando la lista de clientes al servidor con el comando `clientes`, el cual retorna sus ids y nombres y luego con estos datos el cliente sabe a quién dirigir su mensaje con el comando `msg [id] [mensaje]`
- El cliente siempre está revisando una lista de mensajes en paralelo, cada vez que hay un mensaje nuevo, se muestra en pantalla sin interrumpir lo que esté haciendo
- Cuando el cliente envía un mensaje, no debe esperar a que el servidor le responda, pues es asíncrono.
- Para ver el historial de mensajes, debe ingresar el comando `historial`
- Al ingresar `salir`, el cliente se desconecta.

3. Implementación en RabbitMQ

La implementación con RabbitMQ también se compone de 2 programas, un cliente y un servidor, donde este último no es un servidor como tal si no más bien un *worker* encargado de extraer los mensajes de los clientes para así procesar sus solicitudes.
Por el lado de RabbitMQ, la comunicación se lleva a cabo a través de mensajes en formato JSON[4] por medio de 3 *exchanges*:

- **login:** Cola encargada de manejar la parte de logeo de cada cliente
- **chat:** Cola encargada de manejar los mensajes de chat del sistema
- **services:** Cola encargada de los otros servicios del sistema (lista de clientes, historial)

A continuación, se detalla como los 2 programas realizan las acciones del sistema de chat (centrándose principalmente en la parte de RabbitMQ, dado que el resto del comportamiento es igual a la implementación anterior):

3.1. Servidor:

- Al iniciar, se construye la misma estructura de datos de la implementación anterior, no obstante, esta vez en vez de almacenar una conexión gRPC por cliente, se almacena una id única de cada uno, la cual será usada para escribirle a un cliente en particular usando su id como *routing_key*, usando el *exchange* que corresponda.
- Una vez inicializado, se crean 3 hebras, cada una establece una nueva conexión con RabbitMQ, declara el *exchange* correspondiente, crea una cola y se la asocia para así, proceder a mantenerse consumiendo los mensajes que lleguen a esa cola. Una hebra está encargada de manejar el login de los usuarios, otra de manejar el chat y la última de manejarlos otros servicios del sistema de chat.

- El resto se comporta del mismo modo que la implementación anterior, con la diferencia de que la comunicación se materializa a través de colas en el *exchange* correspondiente y los mensajes son derivados a su receptor a través del uso de su *routing_key*, cuya generación es responsabilidad del cliente.

3.2. Cliente:

- Al iniciar, se conecta con el servidor RabbitMQ, luego genera 2 canales: uno para logear y otro para usar los servicios aparte del chat, declara en ellos los *exchanges* correspondientes y para entonces, ya se considera "conectado con el servidor" (aunque en realidad sólo sea una conexión con RabbitMQ, pues no hay comunicación directa con el otro programa).

- Al momento de logear, dado que la única forma de que el programa que actúa de "servidor" pueda comunicarse directamente con el cliente (o bien, escribir un mensaje en la cola que va a ser leído por un cliente en particular y no el primero en leer de la cola) es a través de una *routing_key* en el *exchange* correspondiente, el cliente genera una id única combinando el nombre con el que quiere logear y un identificador único universal UUID v4[3], y procede a usarla como *routing_key* para enviar un mensaje de login utilizando el formato JSON[4].

- Una vez que envía el mensaje de login, se mantiene esperando hasta que llegue un mensaje a la cola que asoció al *exchange* de login, donde determinará si está o no correctamente logeado.

- Terminado el proceso de login, el cliente puede enviar mensajes al *exchange* de chats (de manera asíncrona) o al de servicios, siguiendo el mismo flujo que la implementación anterior, con la salvedad de que cada vez que manda un mensaje a un *exchange*, el cliente ingresa como *routing_key* la id única generada al momento de logear, esta luego es usada por el servidor para distribuir sus mensajes de chat o bien, para responderle al cliente por algún mensaje como `historial` o `clientes`.

4. Recomendación Técnica

Luego de realizar ambas implementaciones, la recomendación sería hacer uso de RabbitMQ, conforme a lo siguiente:

- Al no ser necesario mantener una conexión directa entre el cliente y el programa que hace de servidor (como en gRPC), el cliente no tiene que ser consciente de que hay un servidor al que está directamente conectado respondiendo sus solicitudes, esto facilita cosas como replicar el programa de servidor varias veces (lo que para RabbitMQ significaría conectar más *workers*) y así poder distribuir la carga de las solicitudes de los clientes.

- Ambas implementaciones tienen la problemática de que deben almacenar muchos datos en memoria, esto podría subsanarse con una base de datos en una implementación más profesional, con el fin de evitar restricciones de memoria, ahora bien, con RabbitMQ sería más sencillo mantener los datos de conversaciones distribuidos a lo largo de los distintos *workers* (sin una bd adicional).

- RabbitMQ, al ser un servicio aparte, facilita cosas como hacer un cluster, o distribuirlo geográficamente, con gRPC en cambio, sería necesario implementar esta lógica de manera manual, lo que

exigiría unir la lógica de sistema distribuido con la del programa, RabbirMQ permite abstraerse de esto.

- Ambas tecnologías permiten la comunicación entre programas en distintos lenguajes, no obstante, gRPC no ofrece la posibilidad de abstraerse de los tipos de datos de cada lenguaje (pudiendo incluso llegar a un lenguaje no soportado por gRPC), en cambio para RabbitMQ, basta con enviar mensajes de texto para traspasar la información, utilizando alguna notación como JSON[4] para comunicar 2 programas en lenguajes no soportados por gRPC
- RabbitMQ trae una serie de herramientas que permiten manejarlo de manera independiente, gRPC requeriría el desarrollo de herramientas propias orientadas al sistema que se está desarrollando con él.

No obstante, todas estas recomendaciones van de la mano de la implementación que en este informe se realizó, podrían existir mejores implementaciones donde gRPC sería una mejor alternativa, o escenarios donde esta implementación no sería adecuada para ninguna de las tecnologías, es siempre importante realizar un análisis de la situación a la que se va a enfrentar el sistema a desarrollar de modo tal de poder escoger las tecnologías a usar de la manera más adecuada.

Referencias

[1] Cloud Native Computing Foundation. (2019). *"gRPC – A high performance, open-source universal RPC framework"*. Extraído de https://grpc.io/

[2] Pivotal. (2019). *"Messaging that just works – RabbitMQ"*. Extraído de https://www.rabbitmq.com/

[3] Leach, P., Mealling, M. & Salz, R. (2005). *"A Universally Unique IDentifier (UUID) URN Namespace"*, RFC 4122, https://tools.ietf.org/html/rfc4122.

[4] T. Bray, Ed. (2017). *"The JavaScript Object Notation (JSON) Data Interchange Format"*, RFC 8259, https://tools.ietf.org/html/rfc8259.