Sistemas Distribuidos (INF-343)

Actividad 3: Informe comparativo

Sebastián Torrico - 201330061-8
Juan Escalona - 201373551-7

3 de enero 2020

1 Resumen

En este informe se analizarán las diferencias técnicas entre las dos maneras de implementar el sistema de mensajería distribuido basado en los micro-servicios definidos en la tarea. El objetivo de esto es determinar cuál es la mejor implementación y tecnología para desarrollar la arquitectura de un Chat.

2 Introducción

En las arquitecturas existe un estilo que es representado por micro-servicios, cuales también son un modo de programar software. Los micro-servicios permiten que las aplicaciones sean independientes entre sí y sean divisibles en componentes más pequeños. Estos son independientes, pero pueden funcionar en conjunto para realizar las mismas tareas, es decir, representan una arquitectura distribuida. Normalmente se usan APIs a través de HTTP. [1]

Existen varios tipos de tecnologías para realizar una infraestructura de micro-servicios. Este informe se enfocará en dos tipos: *procedimientos remotos* y *mensajería asíncrona*. Para ello, se desarrolló una tarea que consistía en construir un Chat en el cual pueden ingresar multiples clientes y debe existir un servidor central de coordinación de mensajes. Además de los requerimientos específicos del Chat, este debía ser construido con los dos tipos de tecnologías mencionados anteriormente. Para ello, en el desarrollo de procedimientos remotos se utilizó la tecnología *gRPC*; mientras que para la mensajería asíncrona se utilizó *RabbitMQ* como broker de mensajería.

El objetivo principal de este informe es comparar ambas soluciones mediante el análisis de las diferencias técnicas de cada sistema de mensajería distribuida basado en microservicios. Finalmente, se concluirá cuál es la mejor tecnología para implementar este problema.

3 Tecnologías, características e implementación

3.1 gRPC

3.1.1 Características

gRPC es un *framework open source* moderno de alto rendimiento para realizar llamadas a procedimientos remotos; está desarrollado inicialmente en Google y es utilizado en proyectos como Dropbox. Esta tecnología es capaz de generar enlaces multiplataforma entre clientes y

servidores en una gran variedad de lenguajes; en este caso se utilizó proto (versión 3) y python. Su principal uso fue conectar los servicios de la arquitectura (Cliente-Servidor) mediante el traspaso de mensajes. [2]

3.1.2 Implementación

Para ilustrar de mejor manera las características que brinda gRPC es conveniente analizar la implementación. La primera solución se desarrolló un archivo proto que especifica los principales servicios y forma de los mensajes de la arquitectura. Los servicios contienen a los métodos que son llamados remotamente con sus parámetros y retornos.

```
syntax = "proto3";

package grpc;

import "google/protobuf/timestamp.proto";

/* Servicio de Chat para en envIo y recepciOn de mensajes. */
service Chat {
    rpc SendMsg (Msg) returns (Empty);
    rpc Channel (Empty) returns (stream Msg);
}

/* Servicio de Usuarios para el administracion de estos.
 * Permite la conexion de clientes al chat, obtencion del listado
 * de usuarios y la desconexion. */
service Users {
    rpc Join (User) returns (Response);
    rpc GetUsers (Empty) returns (UsersListResponse);
    rpc Disconnect (User) returns (Empty);
}

/* Servicio de Almacenamiento de mensajes. Permite guardar temporalmente
 * los mensajes y entregar una lista de estos al cliente que lo solicite.
 */
service MessagesService {
    rpc SaveMessage (Msg) returns (Empty);
    rpc GetAllMessages (User) returns (UserMessages);
    rpc DeleteMessages (User) returns (Empty);
}

/* Mensaje auxiliar para la asignacion de ID para clientes. */
message Response {
    bool opt = 1;
}

/* Mensaje vacio auxiliar para funciones de servicios que no exijan argu-
 * mentos o retornos. */
message Empty {}

/* Mensaje de envio para usuarios. Contiene los elementos que identifican
 * al cliente, el texto enviado y un timestamp para registrar la hora de
 * env o. */
message Msg {
    string id = 1;
    string message = 2;
    google.protobuf.Timestamp timestamp = 3;
}

/* Mensaje de usuario que contiene el id de este. Utilizado para funciones
 * que requieran al usuario como argumento. */
message User {
```

```
    string user_id = 1;
}

/* Mensaje de tipo lista que contiene usuarios. Utilizado para aquellos
 * servicios que retornan listas de usuarios. */
message UsersListResponse {
    repeated User users = 1;
}

/* Mensaje de tipo lista que contiene mensajes. Utilizado en servicios que
 * retornar listas de mensajes. */
message UserMessages {
    repeated Msg msgs = 1;
}
```

Listing 1: Servicios y mensajes.

Se definieron 3 servicios: *Chat*, *Users* y *MessagesServices*. El servicio Chat es aquel que coordina el envío y recepción de mensajes en el servidor. Para ello se definen 2 métodos (RPC), donde *SendMsg* recibe un mensaje de un cliente y lo almacena; mientras que *Channel* es un método que reenvía los mensajes a los destinatarios, en este caso mediante un flujo de estos. Como es un chat, el mensaje enviado por un cliente debe ser enviado a todos los presentes en la sala. El servicio Users se encarga de controlar los usuarios conectados al chat mediante una función *Join*, que recibe un usuario y responde si es posible asignarle un nombre, el método *GetUsers* que recibe un mensaje vacío y retorna una lista con todos los usuarios conectados y el método *Disconnect* que borra al usuario del Chat. Por último, el servicio MessagesServices se encarga de almacenar los mensajes mediante sus tres llamadas remotas: *SaveMessage*, almacena un mensaje enviado; *GetAllMessages* que retorna todos los mensajes del usuario que llama al método y DeleteMessages, cual borra todos los mensajes del usuarios que se desconecte.

Por otra parte se definen los mensajes a ser utilizados (y enviados) por los servicios, donde los principales son *Msg*, *UserListResponse* y *UserMessages*; el primero es la estructura principal de los mensajes que se envían los usuarios, mientras los otros son colecciones de mensajes. La definición de cada uno se encuentra en el código de arriba.

Con el uso de Docker se desplegaron contenedores de servidor y clientes, donde estos últimos crean un *channel* único con el servidor. El contenedor del servidor dispone de los 3 servicios definidos, además de anotar en un log todos los mensajes enviados. Por otra parte, los clientes envían y reciben los mensajes de forma asíncrona, esencialmente con el servicio Chat.

3.1.3 Resultados

Con esta implementación se logró un sistema capaz de ejecutar métodos de manera síncrona y asíncrona. Un ejemplo es el rpc que se utiliza para obtener la lista de los mensajes enviados por el cliente, ya que el cliente se bloquea hasta recibir la respuesta del servidor. Por otra parte, el envío de mensajes hacia el chat funciona de manera asíncrona, ya que los usuarios no tienen que esperar a recibir una confirmación para volver a enviar. La ventaja de esto es que permite una mensajería instantánea y a tiempo real entre los usuarios; sin embargo, su principal desventaja es que los métodos síncronos ralentizan el uso del chat, donde las listas muy extensas se mostraban lentamente.

3.2 RabbitMQ

3.2.1 Características

RabbitMQ es un *bróker de mensajería*, es decir, un patrón arquitectónico para la validación, la transformación y el ruteo de mensajes permitiendo establecer conexiones entre distintas aplicaciones sin que cada una de ellas conozca en detalle la arquitectura con la otra aplicación con la que se esta comunicando. Dentro de las características que permiten dicho comportamiento con un canal utilizando RabbitMQ se encuentran 3 elementos principales: Consumidores/Productores, Colas y Exchanges. Los Productores son quienes publican un mensaje con un debido ruteo hacia una cola, los Exchanges son un mecanismo para realizar transmisión de mensajes a más de una cola de destino y las Colas corresponden al mecanismo mediante el cual los Consumidores obtienen los mensajes, permitiendo mantener aislados ambos sistemas entre si. Dentro de las grandes facilidades que ofrece la arquitectura es el sistema *Publish/Subscribe* [3], el cual permite que el servidor en si mismo se encargue del ruteo de mensajes a cada una de las colas que estan suscritas a un *exchange* evitando tener que definir la lógica desde la base. También destaca el sistema de *Routing* [4] que permite al productor enviar (y a los consumidores recibir) mensajes selectivamente.

3.2.2 Implementación

Para la implementación del sistema con RabbitMQ, lo que se realizo fue, utilizando la misma arquitectura anterior de n-clientes y un servidor asignarle a cada uno de los componentes su cola de mensajes correspondientes, además de una cola adicional para el servidor por donde se realizan todas las demás solicitudes que no son mensajes estándar. Con respecto a los Exchanges, el sistema solo cuenta con uno, dado que los clientes se conectan de manera directa a las colas del servidor. Dicho Exchange se encarga de realizar *broadcast* a cada una de las colas de los clientes cuando alguno de ellos envía un mensaje nuevo.

Con respecto a las dos colas con las que cuenta el servidor, una de estas corresponde a la cola donde llegan los mensajes propiamente del chat, mientras la otra cola se encarga de identificar ciertas palabras reservadas para que el servidor registre nuevos usuarios y realice sus funcionalidades anexas, tales como entregar la lista de usuarios.

3.2.3 Resultados

Una de las ventajas de esta implementación es que permite la distribución de mensajes de manera más ordenada y rápida. Los clientes (consumidores) se suscriben al servidor (productor) quien facilita el re-envío de mensajes mediante broadcast, mientras que para las solicitudes de los clientes el servidor puede enviar directamente un mensaje a su cola mediante ruteos, generando así una arquitectura más eficiente. El problema de no utilizar llamadas a procesos remotos es que se debe definir múltiples restricciones para transformar un proceso asíncrono a algo que debiese ser síncrono. Por ejemplo, algunas solicitudes que dependen de respuestas por parte del servidor implica tener que bloquear otros procesos, como en el caso de tener que bloquear los threads (o consumidor de cola) para poder registrar a un usuario.

4 Conclusión

A pesar que la arquitectura generada por la implementación de RabbitMQ facilita la distribución de mensajes, sus características no son las más favorables para un Chat que depende del envío de éstos a tiempo real. Como el enunciado lo dice, RabbitMQ es un broker de mensajería y destaca por priorizar la confiabilidad mediante sus conexiones TCP, sin embargo, esto no es lo más adecuado o necesario en un sistema como el implementado. Algunos de los métodos

que utilizan los clientes para solicitar cosas del servidor se facilitan usando RPC, el cual estaba prohibido en la segunda actividad. El único punto negativo con el que se puede contar en gRPC es que requiere un desarrollo a un nivel mas bajo que RabbitMQ, lo cual requiere que quien lo desarolle tenga una mayor capacidad para programar los servicios. Por tanto, por simplicidad y eficiencia ante envío/recepción de mensajes instantáneos, además de considerar los beneficios de la modularidad que ofrece, la mejor opción es utilizar gRPC.

References

[1] *¿Qué son los microservicios?* URL: `https://www.redhat.com/es/topics/microservices`.

[2] *gRPC Concepts*. URL: `https://grpc.io/docs/guides/concepts/`.

[3] *Publish/Subscribe*. URL: `https://www.rabbitmq.com/tutorials/tutorial-three-python.html`.

[4] *Routing*. URL: `https://www.rabbitmq.com/tutorials/tutorial-four-python.html`.