ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

[DΣIM]

Distributed Systems

Pràctica 1

curs 2023-24

Estudiantes: Germán Puerto y Montserrat Palazón

Profesor/a: Pedro García

Fecha de entrega: 10/05/2024

Abstract

*This work describes the implementation of a chat application in Python. The app allows private chats between two users and group chats with multiple participants. For communication, different technologies are used, the first of which is the **Client**, whose interface is executed by the user and which offers options to connect to existing chats (private or group), subscribe to new groups, discover active chats and enter an insult channel. The **Server** runs in a separate process and consists of two components a Name Server which uses Redis (or RabbitMQ) to store the relationship between user/group names and connection information and the Message Broker - uses RabbitMQ for group chats, discovery of chats and messages from the insult channel. Regarding **Private Chats**, the connection and exchange of messages between two users is done using gRPC. While **Group Chats** use RabbitMQ Exchanges for collective communication. Customers can subscribe, connect and message groups. Regarding **Chat Discovery**, this is implemented with RabbitMQ to find active chats through shared events, rather than a shared repository. Finally, there is also an **Insult Channel** that uses a RabbitMQ queue (rather than an Exchange) to demonstrate the difference and send insults to random clients.*

Decisiones de diseño

Private chats

Nuestro servicio proporciona dos métodos remotos rpc uno para enviar mensajes y otro para recibir mensajes definidos en el .proto junto con la estructura de los 4 mensajes a intercambiar. EnviarMissatge recibe por parámetro una petición de mensaje que contiene el propio mensaje y los identificadores del remitente, del destinatario y del thread y devuelve un RespostaMissatge que únicamente contiene un identificador del mensaje correspondiente al número de mensajes presentes en la cola en ese momento. Por otro lado, RebreMissatges recibe por parámetro al Cliente mensaje el cual solo contiene su identificador y devuelve un flujo de Mensajes cada uno con información del identificador del mensaje en la cola, los identificadores del remitente, del destinatario y del thread así como el propio mensaje recibido. Al devolver un *stream* esto permite que el servidor pueda comunicar mas de un mensaje a través de una única conexión, el cliente leerá mensajes de este flujo hasta que no queden más, además, gRPC garantiza que los mensajes se leeran en el orden que se han enviado.

Con el .proto generamos los ficheros chatPrivat_pb2.py, chatPrivat_pb2.pyi y chatPrivat_pb2_grpc.py y una vez obtenidos implementamos el servidor y el cliente. Antes de explicar el funcionamiento de estos, indicar que hemos decidido implementar una cola asíncrona la cual consiste en un Diccionario que tiene como clave el identificador del cliente y como valor una cola asíncrona de asyncio. De esta manera el servidor guarda un Diccionario que contiene las colas de todos los clientes connectados a chats privados. Una decisión de diseño importante es utilizar esta cola de asyncio, hemos decidido utilizarla debido a que nos permite gestionar operaciones de E/S de manera eficiente y dado que estas tienden a ser la principal fuente de bloqueo utilizando esta cola y no la convencional conseguimos que se agreguen elementos a la cola o se extraigan sin bloquear el flujo de ejecución. Esta cola tiene 4 métodos uno que devuelve el nombre de mensajes que hay en la cola de un determinado cliente, otro que comprueba si existe o no la cola de un cliente y en caso de que no, la crea. Y los dos principales métodos que son put_missatge y get_missatge que añade u obtiene respectivamente un mensaje a la cola del destinatario indicado. Cuando se añade un nuevo mensaje se retorna el identificador del mensaje en la cola (que se corresponde al nombre de mensajes que había en la cola) y cuando se obtiene un mensaje el retorno del método es el propio mensaje.

En relación con el servidor, lo primero a indicar es que hemos decidido implementar sus funciones como métodos asíncronos utilizando asyncio. De esta manera el servidor puede hacer uso de la cola implementada la cual está diseñada para ser utilizada en código asíncrono. Además, el motivo principal es conseguir concurrencia en las funcionalidades del servidor y creemos que en este caso asyncio es una muy buena solución porque sus funciones asíncronas van ejecutándose concurrentemente y cambiándose en momentos indicados que suelen ser cuando se producen lecturas o escrituras específicas. El servidor declara su cola asíncrona y implementa sus dos funciones EnviarMissatge i RebreMissatges. Por un lado, para enviar mensajes en primer lugar se comprueba que el mensaje enviado no sea “EXIT” en caso de serlo el destinatario de ese mensaje pasaría a ser el propio remitente indicando que quiere abandonar el chat. Mientras que si el mensaje no se corresponde con “EXIT” introducimos el mensaje en la cola asíncrona permitiendo con *await* que se espere a que se complete la operación pudiendo mover el flujo a otras tareas mientras tanto. Habrá que pasarle un mensaje con la estructura marcada en el proto i por tanto, indicamos todo lo necesario. Por otro lado, para recibir mensajes se entra en un bucle el cual va obteniendo mensajes de la cola y retornándolos. Cuando el mensaje recibido sea “EXIT” será la indicación para finalizar el bucle y abandonar el chat. Por último, el main principal inicia el servidor el cual configura inicialmente el puerto y

una vez indicado el server.start() se espera a su finalización con wait_for_termination() para poder cerrarlo de una manera limpia se ejecuta con la función de asyncio: loop.run_until_complete() que interrumpimos con un CTRL+C.

Por último, indicar que con la implementación y las colas para cada cliente que almacena el servidor se ha conseguido que el chat privado sea persistente de manera que cuando un cliente abandona el chat los mensajes que recibe se almacenan en su cola y no se leen hasta que vuelve a conectarse.

Group chats

El chat grupal implementado a través del patrón Publisher/Subscriber de RabbitMQ consiste en groupChat.py el cual recibe por parámetro el nombre del grupo al que el cliente desea conectarse. Hemos decidido implementarlo mediante la creación de un thread que se encargará del consumo de mensajes (Consumer) mientras que el hilo principal funcionará de Publisher. En primer lugar, se inicia el ConsumerThread pasándole por parámetro el nombre del grupo a continuación, se establece la conexión con el servidor RabbitMQ y se declara el Exchange que llevará el nombre del grupo ya que consiste en el topic al que se conectaran todos los clientes que quieran estar en un mismo grupo. Una vez configurado todo, el hilo principal entra en un bucle infinito que espera inputs que corresponderían a los mensajes enviados por el cliente. Captura estos inputs i los publica en el canal en el grupo concreto a través del exchange especificado. Este bucle infinito finaliza con la recepción de un CTRL+C que lo interrumpe. Por otro lado, el thread del Consumer al crearse inicializa el nombre del grupo y un booleano para detectar interrupciones. La función que ejecuta el thread empieza configurando todo lo necesario para la conexión y uniendo su cola al exchange concreto. Finalmente va consumiendo los mensajes de su cola hasta que es interrumpido por el stop() que le hace el hilo principal al detectar un CTRL+C.

Además, la implementación persistente (muy similar a la transitoria) consiste en tres ficheros: subscribeGroupChat.py, connectGroupChat.py y ConsumerThreadGroupChat.py que se diferencian de la versión persistente en que se ha dividido la configuración de RabbitMQ dejando en la parte de subscripción toda la declaración del Exchange y la unión y creación de la cola mientras que al conectarse únicamente se consume de esta cola ya creada antes y la cual al ser Durable y los mensajes ser DeliveryMode.Persistent mantiene los mensajes enviados mientras el cliente se desconecta y se los muestra al volver a conectarse. En cada desconexión se le pregunta al cliente si quiere abandonar definitivamente el grupo para eliminar su cola.

Chat discovery

Para el chat discovery tenemos dos implementaciones: estática (redis) y dinámica (rabbitmq). Redis es estática ya que somos nosotros los que modificamos los valores (con hset) a la hora de que un cliente entra en un chat, en cambio rabbitmq, en nuestra implementación, consulta dinámicamente mediante su servicio quien está conectado.

Redis

Para redis, la idea es sencilla. La librería redis de python, permite que los diccionarios de python sean fácilmente añadibles a un hashset de Redis. Entonces para añadir, simplemente estableciendo la conexión a redis y a este objeto conexión, llamar la función hset, pasando por parámetro el nombre del hashset (private_chat o group_chat), la clave y el valor.

Desde tanto los chats privados y los grupales, cuando un usuario entra en el chat, se añadirá el identificador (id_client o group_name) y la IP, y cuando se desconecte se borrará el identificador.

RabbitMQ

Para rabbitmq, hemos hecho una solución alternativa a lo que se proponía principalmente. La funcionalidad de chat Discovery está implementada, pero no con las colas que se describen ya que la idea sería muy similar al group_chat. Entonces, tanto por simplificarlo como por ver una metodología distinta, hemos decidido que la consulta de los clientes conectados a un chat grupal se hará mediante la API WEB de rabbitmq. Esto lo podemos hacer ya que estamos haciendo una simulación en localhost, pero si fuese un caso real, no sería la mejor opción, y si la usásemos, tendríamos que poner algún tipo de autenticación y/o seguridad.

Insult Channel

A parte de los private chats y los group chats, la aplicación también implementa un servicio de canal de insultos. El funcionamiento de este canal será similar al de un chat grupal, pero con diferencias. El canal se ha implementado sando una cola en lugar de un intercambio. En este grupo, los clientes pueden escribir y recibir insultos. Cada insulto llegará a un cliente. Cada mensaje debe llegar a uno, y sólo un cliente indefinido. Idealmente, los mensajes deberían distribuirse equitativamente entre todos los clientes conectados. Al implementar colas en rabbitmq, la distribución de mensajes por defecto es round robin, es decir va rotando de uno a uno cíclicamente la lista de los clientes conectados a esa cola; por lo tanto, llegará cada mensaje únicamente a un cliente conectado y se irá alternando.
La implementación se ha dividido en 3 partes, la clase Server (Consumer) y la clase Client (Producer) y un main que fusiona estas dos, convirtiendo al usuario en consumidor y productor cuando entra al chat. El usuario puede alternar entre tener ambos papeles o simplemente el de Consumidor al pulsar CTRL+C. Si pulsa un segundo CTRL+C ya se sale de la cola de insultos.

Questions

Q1. ¿Los chats privados son persistentes? Si no, ¿cómo podríamos darles persistencia?

Sí, hemos decidido hacer los chats privados persistentes. Si un usuario (1) ha establecido un chat privado con otro usuario (2) mientras los dos están conectados van recibiendo los mensajes correctamente a medida que estos se envían. Pero además, en el supuesto caso que el usuario (1) abandonara el chat y el usuario (2) durante su ausencia le enviara un mensaje al ser el chat persistente cuando el usuario (1) vuelva a entrar en el chat recibirá aquel mensaje enviado. Esto se ha conseguido gracias a que el servidor gRPC tiene un Diccionario el cual tiene por cada usuario una cola de manera que cada vez que el servidor recibe un mensaje mira el destinatario y añade ese mensaje en su correspondiente cola. Solo se sacan los mensajes de esa cola y se le muestran al usuario que los recibe cuando este está activo. Por tanto, cuando el usuario vuelve a conectarse se pone en marcha su lectura de mensajes y recibe todos los que tenía pendientes.

Q2. ¿Existen *stateful communication patterns* en vuestro sistema?

Tenemos patrones de comunicación stateful en nuestro sistema, en los chats privados como en los grupales, el envío de un mensaje puede depender del anterior, es decir, se almacena un estado que influirá en la acción de los siguientes mensajes.

- Private chats: gRPC lo consideramos stateless, lo que significa que cualquier información necesaria para mantener la comunicación se retransmite con cada intercambio de solicitud/respuesta. Podríamos implementar en nuestro código alguna manera de guardar el estado entre mensajes; lo que tenemos es persistencia de mensajes para guardarlos en las colas de los clientes hasta que se lean, pero esa persistencia no la consideramos como que cambia el estado del envío de mensajes, por eso consideramos gRPC stateless.
- Publish/Suscribe: el chat grupal si que es stateful, ya que para empezar, a comparación de gRPC, rabbitmq suele ser stateful. En este caso lo es, ya que almacena que clientes están suscritos a un exchange, y recibirán ese mensaje solo los suscritos, por lo tanto el envío de mensajes si que depende del estado.
- Insult channel: la cola de insultos no es stateful, ya que no hay nada que modifique el estado del envío de mensajes.

Q3. ¿En qué tipo de patrón se basan los chats grupales? En términos de funcionalidad, compara la comunicación transitoria y persistente en chats grupales utilizando RabbitMQ.

Los chats grupales se basan en el patrón Publish/Subscribe el cual consiste en publishers que corresponden a los remitentes que envían mensajes a topics. En estos topics puede haber diversos subscribers de los cuales no sabrá el Publisher. Por tanto, los subscribers pueden estar en diversos topics por interés al Publisher el cual enviará mensajes y estos serán recibidos por todos los subscribers.

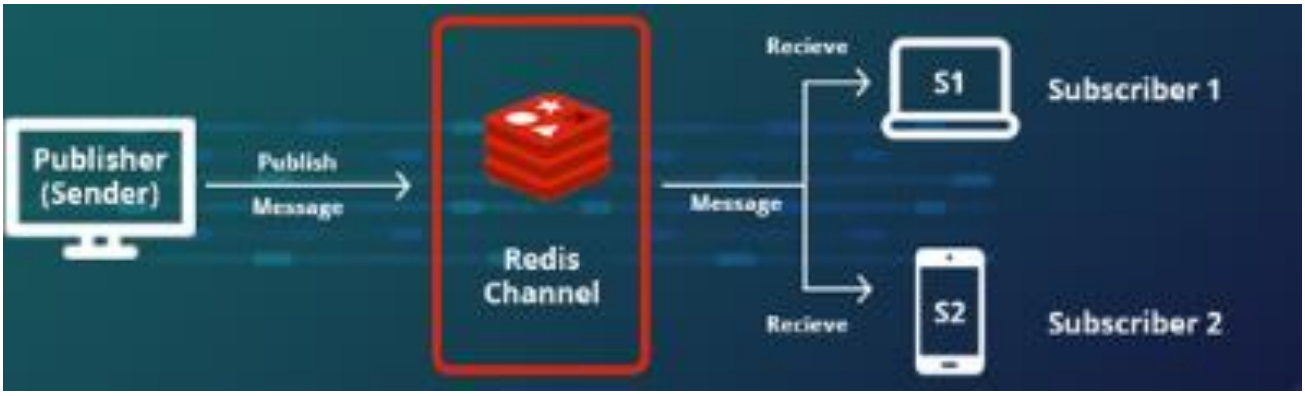
Para el chat grupal esto se ha utilizado de manera que todos nuestros usuarios son a la su vez publishers y subscribers ya que queremos que puedan leer y escribir mensajes en el grupo al que estén conectados. El chat grupal al que están conectados viene determinado por el Exchange del que hace uso RabbitMQ.

Hemos decidido utilizar la comunicación transitoria para los chats grupales y la persistente para los chats privados. Creemos que, así como para chats privados vemos esencial tener una comunicación persistente para no perder ningún mensaje en los chats grupales en algunos casos puede tener sentido la comunicación transitoria ya que puedes desconectar-te del grupo y darse el caso que todo el resto de los participantes (que pueden no ser pocos) seguir manteniendo una conversación de una gran cantidad de mensajes que el

cliente cuando se conecte posteriormente no quiera leer. Por tanto, en la versión completa hemos mantenido la versión transitoria de los chats grupales mientras que la versión persistente se ha dejado a un lado implementada para poder probar su funcionalidad, pero decidiendo integrar la persistente teniendo de esta manera chats privados persistentes y grupales transitorios.

En términos de funcionalidad, RabbitMQ implementa el patrón de manera que cada subscriber a un topic tiene su propia cola y los mensajes de los publishers se mandan a todas estas colas por tanto, la diferencia entre transitoria y persistente consistiría en no eliminar la cola al desconectarte y en establecer las propiedades de cola Durable i de DeliveryMode persistent así, la cola continuaría almacenando todos los mensajes y al volver a conectarse el cliente al grupo esta no se volvería a crear sino que la mantendría y se mostrarían todos aquellos mensajes recibidos durante su ausencia.

Q4. Redis también puede implementar colas y patrones de pubsub. ¿Preferiríais utilizar Redis en lugar de RabbitMQ para eventos? ¿Podríais tener chats grupales transitorios y persistentes? Comparar ambos enfoques.



RabbitMQ se destaca como la herramienta perfecta para sistemas que manejan un alto volumen de mensajes, ofreciendo una escalabilidad robusta que te permite adaptarte sin problemas al ritmo creciente de la demanda. Además, garantiza una entrega segura de mensajes incluso en los momentos de mayor tráfico, lo que la convierte en una opción confiable para aplicaciones críticas.

También, va más allá de la mensajería básica, proporcionando un conjunto completo de características que incluyen enrutamiento complejo, priorización de mensajes, reintentos y manejo de mensajes fallidos. Estas capacidades la convierten en la herramienta ideal para escenarios de mensajería exigentes que requieren un control preciso y una gestión robusta del flujo de mensajes.

Redis, por otro lado, brilla en aplicaciones simples y en tiempo real, donde la velocidad y la facilidad de uso son primordiales. Su interfaz intuitiva y su rendimiento excepcional para operaciones básicas de pub/sub la convierten en una opción ideal para implementaciones rápidas y fáciles. Si tu aplicación no maneja un volumen de mensajes extremadamente alto y la complejidad no es un factor crítico, Redis puede ser una alternativa atractiva por su simplicidad y rapidez.

En resumen:

- **Rendimiento, la escalabilidad y las funciones avanzadas:** RabbitMQ.
- **Simplicidad, velocidad y menor volumen de mensajes:** Redis.