ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

[DEIM]

Sistemas distribuidos

Práctica 1

Curso 23/24

Miguel Robledo Kusz
Víctor Fosch Tena

ÍNDEX

Resumen del proyecto

El sistema que hemos implementado para la realización del proyecto contempla todas las funcionalidades básicas de un sistema de chat. Tiene dos componentes principales: el cliente y el servidor.

El servidor es la máquina donde se aloja la base de datos redis (que usamos como nameserver para los chats privados) y un contenedor de rabbitmq que ejecuta el servicio de broker de mensajes MOM (Message Oriented Middleware) que da soporte a las funcionalidades de chat grupal, descubrimiento de chats y canal de insultos.

Los clientes no podrán hacer uso del sistema de chat si no conocen la dirección IP del servidor, ya que en el momento en que se inicia el cliente se pide introducir el nombre de usuario (que será su identificador de chat) para acto seguido enviarlo al servidor de redis y así registrar la sesión del usuario para que otros usuarios puedan comunicarse con él mediante un chat privado. Bien, los clientes son capaces de descubrir la dirección IP del servidor de manera dinámica, con un código que envía paquetes broadcast a través de la red local con un mensaje determinado que el servidor estará esperando para contestar con su dirección IP.

Una vez los usuarios conocen la dirección IP del servidor ya pueden hacer uso de las funcionalidades del sistema de chat.

Los usuarios pueden enviarse mensajes mediante chats privados peer-to-peer implementados con gRPC, el hecho de que sean peer-to-peer implica que se está utilizando comunicación directa (menos para descubrir la dirección IP y puerto del otro cliente para lo que necesitarán hacer una consulta a la base de datos redis del servidor).
Además, también pueden suscribirse a diferentes grupos y conectarse a ellos para visualizar y enviar mensajes. En este caso la comunicación es indirecta ya que toda la gestión de los mensajes va a cargo del servidor (a.k.a broker) de rabbitmq.
Por último, tenemos las funcionalidades de descubrimiento de chats, gracias a la cual un cliente puede conocer qué usuarios están conectados en cada momento, y el canal de insultos, donde a medida que se van conectando clientes pueden enviar insultos que llegarán a solamente uno de los clientes que esté conectado en ese momento al canal. Estas dos funcionalidades también están basadas en rabbitmq.

Diseño del sistema

Cliente

Las funcionalidades del cliente están contempladas en el fichero client.py y server_discovery.py.
El client.py contiene el código necesario para representar la interfaz del cliente, esto es, las opciones disponibles en el sistema y los chats. También se realiza un manejo de errores exhaustivo para controlar el flujo del programa, en cuanto a esto, la combinación de teclas Ctrl+C se usa en todas las partes del programa para poder salir al menú principal, y si el usuario ya se encuentra en el menú principal se finalizará el programa.
Desde el cliente se usa el server_discovery.py ya que tiene una función que permite al cliente enviar mensajes broadcast a la red local para descubrir la dirección IP del servidor. El cliente estará bloqueado esperando recibir esta información al iniciar el programa.

Ahora, en cuanto a los aspectos más importantes del diseño del cliente:

- Solamente podrá haber un usuario registrado con un determinado nombre en cada momento. Ejemplo: Si nos registramos como miguel en el sistema y intentamos registrarnos de nuevo como miguel mientras la otra sesión sigue activa, obtendremos un error.
- Una vez finalice el programa (recordamos, con Ctrl+c), el código del cliente envía una petición al servidor de redis para eliminar su información. De esta manera, si intentamos iniciar sesión como miguel después de haberla cerrado, sí que podremos acceder.
- Este diseño del cliente es simple pero eficaz. Permite a los demás usuarios enviar mensajes a su chat privado si su sesión está activa (se encuentra en el servidor redis), o de lo contrario lanzar un mensaje de error avisando de que el usuario miguel no está activo (y por lo tanto tampoco lo estará su servidor gRPC). El hecho de que solamente pueda haber un usuario “miguel” en el sistema en cada momento, permite que los otros usuarios que se están comunicando con él estén seguros de que es él y no otro usuario suplantando su identidad. Este diseño también implica que cualquier usuario se pueda registrar como miguel después de que el original haya cerrado su sesión, de manera que para evitar “robos de sesión” (un usuario accediendo a la información de otro, como los grupos a los que estaba suscrito y los mensajes enviados), cada vez que un cliente cierre sesión borraremos toda la información que pueda ser persistente (en este caso solamente son las suscripciones a grupos las que puede suponer un problema).

Para ejecutar un cliente lo único que debemos hacer es ejecutar el comando: bash start_client.sh, el cual se encargará de crear el entorno virtual y instalar las dependencias en caso de que sea necesario, y después ejecutará el client.py.

Servidor

En el servidor usamos redis para la implementación del nameserver y rabbitmq para la gestión de mensajes indirectos. Solamente necesitamos que estos dos servicios estén instalados y en ejecución, aunque para rabbit usamos docker con un contenedor público de rabbitmq para ofrecer el servicio. Una vez los dos servicios estén activos, se ejecuta el script server_daemon.py que es el encargado de responder con su dirección IP a las peticiones de descubrimiento que lancen los clientes en la red local.

Para ejecutar el servidor solamente debemos ejecutar el comando: bash start_server.sh, este ya se encarga de comprobar que redis esté instalado y en ejecución, de la misma manera que para rabbitmq (los instala o ejecuta en caso de que sea necesario), y después ejecuta el server_daemon.py.

Chat privado

En el chat privado usamos gRPC para la ejecución de procedimientos remotos. La definición de la interfaz .proto es bastante simple:

```
message Message {
  string sender = 1;
  string receiver = 2;
  string message = 3;
}

service PrivateChatService {
  rpc AddMessage (Message) returns (google.protobuf.Empty);
}
```

El método de la interfaz está implementado en el fichero grpc_server.py:

```
class PrivateChatServicer(private_chat_pb2_grpc.PrivateChatServiceServicer):
    def __init__(self, client):
        self.client = client

    def AddMessage(self, message, context):
        self.client.store_message(message)
        response = private_chat_pb2.google_dot_protobuf_dot_empty__pb2.Empty()
        return response
```

Las funcionalidades básicas del chat privado así como del cliente están en grpc_client.py.
Una vez instanciado el cliente inicia un hilo al que se le pasa la instancia del cliente y espera peticiones gRPC.

Cada vez que se ejecuta AddMessage en el servidor del cliente, ejecuta el método store_message del grpc_client.py. En este método se añade el mensaje a la cola del chat y también a la lista de mensajes del chat. El método store_message también se ejecuta desde el método add_message cuando un cliente quiere enviar un mensaje a otro, es decir, al enviar un mensaje se ejecutará el store_message tanto en un cliente como en el otro. La diferencia radica en que cuando el store_message se ejecuta desde el método AddMessage del servidor, se añade el mensaje a la cola del chat y cuando el store_message se ejecuta desde el add_message no se añade a la cola. Esto es importante ya que a la cola del chat solamente deben llegar los mensajes enviados por el otro cliente, ya que en el client.py se ejecutará un código que esperará nuevos mensajes en la cola mediante el método .wait(), así cada vez que se recibe un nuevo mensaje se ejecuta la lógica necesaria para representarlo en la interfaz.

En el client.py tenemos dos funciones: connect_to_private_chat() y chat_interface(). En el connect solamente se pide el identificador de chat al que queremos conectarnos (otro usuario) y se manejan los errores que puedan suceder. Si todo va bien, ejecutará el chat_interface().
En el chat interface usamos dos variables de tipo threading.Event() para señalar eventos a los threads de la función. Tenemos dos threads, el que ejecuta refresh_messages() y el que ejecuta display_chat(). El thread que ejecuta refresh_messages() estará bloqueado en client.get_new_message(chat_id) ya que este método ejecuta el método get() sobre la cola del chat, el cual se bloquea hasta que en la cola hay un nuevo mensaje. Cuando se reciba un nuevo mensaje se llamará a set() sobre el evento new_messages_event para que así, el otro

thread (display_chat) que estará bloqueado en la línea new_messages_event.wait() pueda ejecutarse y representar el mensaje en la interfaz.
Si en algún momento durante el chat se acontece un error, establecemos una variable booleana a True, así cuando el usuario quiera salir del chat (Ctrl+c) se eliminará tanto la cola como la lista de mensajes de ese chat. Esto es una decisión de diseño que se ha llevado a cabo para procurar que si en algún momento el cliente1 se está comunicando con el cliente2, y este último se desconecta, que al cliente1 se le elimine este chat para que al volver a conectarse al cliente2 sea como una nueva sesión y no tenga ningún mensaje en ese chat, ya que como hemos dicho antes, aunque se vuelva a iniciar una nueva sesión como cliente2, se considera como una sesión completamente diferente.

Un aspecto a tener en cuenta es que en ningún momento se hace una llamada a nombre_thread.join(). Este método se usa para que el hilo principal espere a que acaben de ejecutarse los otros hilos antes de continuar. En este caso para detener la ejecución de los hilos al salir del chat utilizamos los eventos de sincronización del módulo threading. Cuando se establece el evento stop_event con set() los hilos dejarán de ejecutarse automáticamente.

RabbitMQ

Hemos implementado una clase llamada RabbitMQ que nos sirve para inicializar todas las colas y exchanges necesarios en las siguientes funcionalidades (chat grupal, descubrimiento de chats y canal de insultos), y también tiene un método que establece una conexión con el servidor de rabbit. Esta clase se instancia después de obtener el nombre de usuario y dirección IP del servidor, y la instancia se pasará como argumento a los constructores de las clases GroupChat, InsultChannel y ChatDiscovery.

Chat grupal

Antes de conectarse a la interfaz de un chat grupal el cliente tiene que suscribirse.
Es importante que se suscriba porque es cuando se crea la cola del cliente que se vinculará al exchange para poder recibir mensajes dirigidos a ese grupo.
Para suscribirse debe ejecutar el método subscribe_to_group_chat() del client.py, donde se realiza un control de errores y se comprueba si el usuario ya estaba suscrito.
Una vez suscrito ya tiene la cola creada y vinculada al exchange group_chat que se ha creado en el método initialize() de la clase RabbitMQ.
El exchange es de tipo "topic". Cuando se vincula la cola al exchange se establece la routing_key como *.*nombre_grupo*.*. Así, cuando un usuario quiere enviar un mensaje a todos los miembros de ese grupo, debe enviar el mensaje con la routing_key establecida como, en nuestro caso, nombre_usuario.nombre_grupo.msg, y el exchange se encargará de distribuir ese mensaje a todas las colas vinculadas al exchange que tengan la routing_key indicada.

Es importante reconocer que esta funcionalidad se podría haber implementado declarando un exchange de tipo "fanout" para cada chat grupal, y cada vez que un usuario se suscriba, que cree su cola y la vincule al exchange del chat grupal en cuestión, de esta manera cada vez que un usuario quisiera enviar un mensaje a ese grupo solamente debería enviar el mensaje al exchange de ese grupo y este enviaría el mensaje a todas las colas vinculadas a él. Podría ser que esta implementación fuera más eficiente ya que en la que hemos usado nosotros, el exchange debe comparar la routing key del mensaje enviado con la de las colas vinculadas, y ahí se podría perder cierta velocidad, en cambio el de tipo "fanout" envía el mensaje que le llega a todas las colas vinculadas.

Las colas de suscripción a los grupos de cada cliente son durables, esto quiere decir que si el servidor de rabbitmq se reinicia las colas seguirán existiendo. Los mensajes que se envían a estas colas son persistentes, de manera que si el servidor se reinicia seguirán existiendo tanto las colas como los mensajes.
Lo de las colas durables lo hemos hecho para procurar que si durante la sesión del usuario se reinicia el servidor por lo que sea, que el cliente no deba volver a suscribirse a los diferentes grupos, ya que las colas seguirán existiendo.
Por otro lado, lo de los mensajes persistentes lo hemos hecho por lo siguiente: si un cliente se suscribe a un grupo empieza a recibir todos los mensajes. Si el servidor de rabbit se reinicia antes de que el cliente pueda conectarse a la interfaz del grupo, no tendrá los mensajes del grupo almacenados en local, pero gracias a que los mensajes son persistentes, cuando el

servidor vuelva a estar disponible podrá conectarse al chat y recuperar todos los mensajes desde el momento de su suscripción.

Como hemos mencionado antes, cuando el cliente cierre la sesión se eliminarán todas las colas de suscripción para no vincular una sesión con otra.

El método group_chat_interface(chat_id) es parecido al del chat privado, pero con algunas diferencias. Habrá un thread que estará consumiendo constantemente mensajes de la cola, hasta que se establezca el stop_event con set() al pulsar Ctrl+c. Por cada mensaje consumido se ejecuta el wrapper_callback del método receive_messages de la clase GroupChat. Si está el evento de stop establecido, se rechaza el mensaje recibido y se vuelve a poner en la cola, para que la próxima vez que se conecte el cliente al chat grupal pueda consumir ese mensaje y no se pierda. Si el evento no está establecido se hace ack y se ejecuta el callback. El callback es el receive_callback del group_chat_interface, el cual añade el mensaje a la lista de mensajes temporal y representa todos los mensajes de nuevo.

De nuevo, para el hilo no hace falta hacer join() ya que este finalizará automáticamente al establecer el stop_event.

Descubrimiento de chats

La idea aquí es que cada vez que un cliente se conecte, que se ponga a escuchar peticiones de descubrimiento de chats y a responderlas. Cuando un cliente quiera descubrir chats, tiene que enviar una petición de descubrimiento a la que responderán los demas clientes que están escuchando.

Esto lo hemos conseguido de la siguiente forma:

- Declaramos dos exchanges: discovery_requests de tipo “fanout”
- y discovery_responses de tipo “direct”.
- Creamos dos colas durables: nombre_usuario:discovery_requests y nombre_usuario_discovery:responses.
- Vinculamos las colas a los correspondientes exchanges.

Ahora, al iniciar un nuevo cliente se ejecuta un thread que consume mensajes activamente de la cola requests. Antes de empezar, se hace un purge de la cola para evitar responder a peticiones que hubieran en la cola de alguna otra sesión con ese mismo nombre. Cuando se consume una request, le responde, publicando un mensaje en el exchange responses con la routing key establecida con el nombre del requester. Como el exchange es de tipo “direct”, lo que hace es redirigir ese mensaje a la cola responses del requester.
El requester solamente podrá enviar peticiones de descubrimiento y escuchar las respuestas en caso de que se ejecute el método del cliente discover_chats(). En este método se usan dos threads adicionales, donde uno envía peticiones de descubrimiento cada 3 segundos y el otro consume activamente mensajes de la cola de respuestas, y para cada respuesta que se recibe se actualiza la información disponible en la interfaz.
Cuando el cliente pulse cualquier tecla, se establecerá el stop_event haciendo que dejen de ejecutarse los dos threads. Realmente el thread que escucha respuestas solamente finalizará cuando reciba un último mensaje, el cual envíamos nosotros manualmente a la cola con la instrucción chat_discovery.stop_consuming(), así evitamos que en la próxima ejecución hayan 2 threads consumiendo respuestas, lo que puede llevar a problemas de sincronización entre hilos.

Las colas vuelven a ser durables por la misma razón que en los chats grupales, para evitar que se pierdan en caso de reinicio del servidor.

Canal de insultos

En este caso se declara un nuevo exchange llamado “insults” de tipo “fanout”, junto con una cola llamada “insults_queue” que es durable y se vincula al exchange. Todos los usuarios que quieran acceder al canal de insultos consumirán y enviarán mensajes a esta misma cola.

La idea es que en el método access_insult_channel() del client.py, se ejecuta un thread que consume mensajes de la cola global (donde acceden todos los clientes). Por cada insulto consumido de la cola se añadirá a la lista temporal de insultos y se representará en la interfaz.
En el momento en que el cliente salga de la interfaz mediante Ctrl+c, se volverá a establecer un stop_event con set(), como en los casos anteriores, de esta manera el thread que consumía insultos rechazará el próximo insulto que consuma y lo volverá a poner en la cola (requeue=True), y acto seguido dejará de consumir. Mediante la instrucción `channel.basic_qos(prefetch_count=1)` del método consume_insults() de InsultChannel, hacemos que los insultos se repartan de manera equitativa (siguiendo un mecanismo RoundRobin).

Preguntas

1. **Are private chats persistent? If not, how could we give them persistency?**
   Los chats privados son persistentes en el sentido que si el cliente no está conectado al chat en un momento determinado no se perderán los mensajes que envíe el otro usuario, sino que se representarán la próxima vez que se conecte. Eso sí, una vez se cierra el programa del cliente, se eliminan todos los chats privados ya que se encuentran sólo en memoria principal. Si quisieramos que la próxima vez que el cliente se conecte con el mismo nombre los chats privados sigan existiendo, podríamos guardar todos los mensajes del chat en ficheros de texto o en una base de datos. Pero tampoco tendría mucho sentido ya que los mensajes que haya querido enviar el otro usuario mientras el primero estaba desconectado se habrán perdido. Una mejor solución podría ser usar el servidor rabbitmq donde se usen colas para los chats privados, así cuando se desconecte el usuario que guarde todos los mensajes de los chats privados en ficheros de texto o en cualquier otro sitio persistente y cuando se vuelva a conectar podrá recuperar los mensajes que se hayan ido enviando a su cola y representarlos todos en la interfaz.
2. **Are there stateful communication patterns in your system?**
   Un sistema o aplicación stateful es aquel que mantiene un estado interno que registra información sobre sesiones, transacciones o cualquier otra interacción con los usuarios.
   La gestión de las suscripciones y mensajes que se realiza mediante diccionarios de python en cada cliente, podría considerarse un patrón stateful, ya que de esta manera cada vez que te conectas a un chat grupal se recuerda si estabas ya suscrito, así como también se recuerdan los mensajes que se habían enviado a ese grupo.

3. **What kind of pattern do group chats rely on? In terms of functionality, compare transient and persistent communication in group chats using RabbitMQ.**

   Los chats grupales siguen un patrón de comunicación indirecto, el cual proporciona desacoplamiento espacial y temporal.

   En nuestro caso, consideraremos que nuestros chats grupales son persistentes (se guardan los mensajes y se puede acceder a ellos aunque no estuvieramos conectados al chat en ese momento, aunque en el momento que se cierra el cliente se eliminan las suscripciones).

   Gracias a la persistencia que hemos conseguido en los chats grupales de nuestra implementación, los clientes pueden estar haciendo cualquier otra cosa en la aplicación, que cuando se conecten al chat grupal recuperarán todos los mensajes que se hayan enviado.

   Con una implementación no persistente o "transient", este comportamiento no sería posible y cada vez que se saliera de la interfaz del chat grupal y volviera a conectarse perdería todos los mensajes anteriores.

4. **Redis can also implement Queues and pubsub patterns. Would you prefer to use Redis than RabbitMQ for events ? Could you have transient and persistent group chats? Compare both approaches.**

   En nuestro caso preferimos RabbitMQ a Redis para los eventos. Como hemos ido mencionando en la documentación, con RabbitMQ se puede conseguir que las colas sean durables, lo que significa que pueden sobrevivir a un reinicio del servidor, y así evitar que se pierdan las colas de suscripción, de descubrimiento de chats y insultos para poder seguir recibiendo eventos. En el caso de Redis, como es una base de datos naturalmente "transient" ya que la información reside únicamente en memoria principal, un reinicio del servidor supondría la pérdida de toda la información necesaria para la gestión de eventos. De esta manera, con Redis sería imposible implementar chats grupales persistentes en el sentido estricto de la palabra, ya que la información se pierde cada vez que se reinicia el servidor, en cambio esto sí que se puede conseguir con RabbitMQ.