Informe

Roger Fuentes Rodríguez
Kevin Manzano Rodríguez

Índice

1. Arquitectura

1.1. Organización del sistema distribuido

Implementaremos una arquitectura de microservicios, con los principales servicios:

- Autenticación (gestión de usuarios, autenticación e identificación)
- Grupos (gestión de los grupos de los usuarios, incluyendo roles y jerarquías)
- Agendas (gestión de eventos y reuniones)

- Notificaciones (envío de notificaciones en tiempo real a los usuarios)
- Base de datos distribuida (almacena la información)

1.2. Distribución de servicios en ambas redes Docker

Tendremos dos redes de Docker separadas:

- **Cliente (frontend)**
- **Servidor (backend y base de datos)**

El servidor alberga los servicios anteriormente mencionados.

2. Procesos

2.1. Tipos

1. Autenticación
   a) Gestión de usuarios
   b) Autenticación y autorización
2. Grupos
   a) Gestión de grupos
3. Agenda
   a) Gestión de agenda y reuniones
4. Notificaciones
   a) Envío de notificaciones en tiempo real
5. Base de datos
   a) Gestión de datos

2.2. Organización en una instancia

Los procesos estarán en instancias separadas para cada microservicio, cada servicio tiene su propia instancia de contenedor Docker.

2.3. Tipo de patrón de diseño

El sistema utilizará peticiones asíncronas (`async`) para las llamadas HTTP. Para optimizar el envío de notificaciones, se emplearán hilos que permitan paralelizar ese proceso.

3. Comunicación

Tendremos comunicación cliente-servidor a través de APIs RESTful. Entre servidores utilizaremos `ZMQ` y, entre procesos, `RPC`.

4. Coordinación

Para la coordinación entre servidores utilizaremos `PUB-SUB` que nos brinda `ZMQ`, y para el acceso exclusivo a recursos emplearemos `locks`. Con respecto a la toma de decisiones, implementaremos un algoritmo de consenso de quórum.

5. Nombrado y localización

Utilizaremos `CHORD` como se vio en clases. Tendremos un anillo de tamaño 2^m y asignaremos los recursos a una clave utilizando la función de `hash SHA-256`. Para acceder a un recurso, emplearemos las *finger tables* almacenadas en los nodos.

6. Consistencia y replicación

Dado que utilizaremos `CHORD`, para garantizar nivel 2 de tolerancia a fallos, implementaremos replicación múltiple (K-réplicas) para distribuir las copias de los datos entre varios nodos (por ejemplo, consecutivos). Usaremos consistencia eventual combinada con reparación automática de réplicas.

7. Tolerancia a fallas

Utilizaremos mecanismos de *health checks* a los nodos vecinos de manera periódica para verificar si están activos. En caso de que un nodo falle, se asegura que las claves asignadas a ese nodo sean rápidamente reasignadas (gracias a la replicación). Para la recuperación de datos, las réplicas almacenadas en otros nodos sincronizan su información una vez que el nodo se recupere.

8. Seguridad

Durante el tránsito, utilizaremos `TLS` para cifrar la comunicación entre los nodos de `CHORD`. Los nodos deben verificar la autenticidad de otros nodos, por lo que implementaremos una autenticación basada en certificados. Para la autenticación de los usuarios, usaremos `JWT`.