Sistema de Mensajería Distribuido

MongoDB · Redis · Neo4j

Índice

1. Introducción

El objetivo de esta práctica es el diseño e implementación de un sistema de mensajería distribuido empleando tres bases de datos NoSQL con distintos modelos y responsabilidades: Neo4j, Redis y MongoDB.

El sistema simula una red de usuarios conectados mediante relaciones de mensajería, donde cada usuario puede enviar mensajes a otros usuarios con los que mantiene una relación directa. Cada mensaje enviado provoca una actualización coordinada en las tres bases de datos, de forma que cada una cumple un rol específico dentro del sistema.

Neo4j se utiliza para modelar la red de usuarios como un grafo dirigido, almacenando tanto los usuarios como las relaciones de mensajería entre ellos, junto con un contador del número de mensajes intercambiados. Redis se emplea como sistema de colas, manteniendo una cola independiente por cada relación entre usuarios, lo que permite desacoplar el envío y el consumo de mensajes. Finalmente, MongoDB se utiliza como almacenamiento persistente del histórico de mensajes y como repositorio de snapshots del sistema.

Además del envío y consumo de mensajes, el sistema implementa consultas avanzadas sobre el grafo y sobre el histórico de mensajes, así como un mecanismo de snapshots que permite capturar y restaurar el estado completo del sistema en un momento dado.

El diseño de la práctica prioriza la modularidad, la separación de responsabilidades y la coherencia entre las distintas bases de datos, utilizando una arquitectura basada en APIs independientes y servicios de orquestación.

2. Arquitectura general del sistema

El sistema de mensajería se ha diseñado siguiendo una arquitectura modular basada en la separación de responsabilidades. Cada base de datos cumple un rol específico y se accede a ella a través de una API independiente, lo que permite desacoplar la lógica de acceso a datos del resto del sistema.

La arquitectura se compone de tres niveles principales:

- **Capa de persistencia**, formada por Neo4j, Redis y MongoDB.

- **Capa de acceso a datos (APIs)**, que encapsula las operaciones sobre cada base de datos.
- **Capa de servicios**, responsable de coordinar las operaciones entre las distintas APIs.
- **Capa de pruebas**, implementada mediante notebooks Jupyter.

2.1. Capas del sistema

2.1.1. Capa de persistencia

- **Neo4j** almacena el grafo de usuarios y las relaciones de mensajería (`CAN_MESSAGE`). Cada relación incluye un contador del número de mensajes enviados, que se utiliza posteriormente en consultas sobre el grafo.
- **Redis** actúa como sistema de colas, manteniendo una cola independiente por cada relación de mensajería. Esto permite desacoplar el envío de mensajes de su consumo.
- **MongoDB** se utiliza como almacenamiento persistente del histórico de mensajes y como repositorio de snapshots del sistema.

2.1.2. Capa de acceso a datos (APIs)

Cada base de datos dispone de una API propia que encapsula todas las operaciones posibles sobre ella:

- `Neo4jAPI`: gestión de usuarios, relaciones y consultas sobre el grafo.
- `RedisAPI`: gestión de colas de mensajes y operaciones de snapshot relacionadas con Redis.
- `MongoAPI`: gestión del histórico de mensajes y almacenamiento de snapshots.

Esta capa evita que el resto del sistema dependa directamente de las librerías de acceso a las bases de datos, facilitando la mantenibilidad y el aislamiento de cambios.

2.1.3. Capa de servicios

Sobre las APIs se implementan servicios que coordinan la interacción entre las distintas bases de datos:

- `MessagingService`: orquesta el envío y consumo de mensajes, garantizando la coherencia entre Neo4j, Redis y MongoDB.
- `SnapshotService`: se encarga de crear y restaurar snapshots, capturando y reconstruyendo el estado completo del sistema.

Esta capa contiene la lógica principal del sistema y define los flujos de operación de alto nivel.

2.1.4. Capa de pruebas

La validación del sistema se realiza mediante notebooks Jupyter, que permiten ejecutar de forma interactiva los distintos flujos y consultas. Existen notebooks específicos para probar cada API de forma aislada, así como un notebook principal que integra todo el sistema.

2.2. Flujo de envío de un mensaje

El envío de un mensaje implica una actualización coordinada en las tres bases de datos. El flujo completo es el siguiente:

1. El usuario emisor solicita el envío de un mensaje a través del `MessagingService`.
2. Se valida la existencia de la relación de mensajería en Neo4j y se incrementa el contador de mensajes (`msg_count`) de dicha relación.
3. El mensaje se serializa y se inserta en la cola correspondiente de Redis, identificada por la relación entre usuarios.
4. El mensaje queda pendiente de consumo hasta que es recuperado de la cola.

Este diseño permite que el envío de mensajes sea una operación rápida, delegando el almacenamiento persistente al proceso de consumo.

2.3. Flujo de consumo de mensajes

El consumo de mensajes se realiza de forma desacoplada mediante Redis:

1. El consumidor solicita mensajes de una cola concreta utilizando una operación bloqueante con tiempo de espera.
2. Al recuperarse un mensaje, este se elimina de la cola de Redis.
3. El mensaje se deserializa a un objeto de dominio (`MessageData`).
4. El mensaje se almacena de forma persistente en MongoDB como parte del histórico.

De esta forma, Redis actúa como un buffer temporal, mientras que MongoDB garantiza la persistencia a largo plazo.

2.4. Flujo de snapshots y restauración

El sistema implementa un mecanismo de snapshots que permite capturar el estado completo del sistema en un momento dado:

- El estado del grafo en Neo4j (usuarios y relaciones).
- El contenido de todas las colas de Redis.
- El histórico de mensajes almacenado en MongoDB.

La restauración de un snapshot implica la eliminación del estado actual y la reconstrucción completa del sistema a partir de la información almacenada, respetando el orden necesario para mantener la coherencia entre las bases de datos.

Este mecanismo facilita la recuperación ante errores y permite analizar la evolución del sistema entre distintos estados.

---

3. Diseño y uso de Neo4j

Neo4j se utiliza en el sistema como base de datos orientada a grafos para modelar la red de usuarios y las relaciones de mensajería entre ellos. Este modelo resulta especialmente adecuado para representar conexiones directas entre usuarios y para realizar consultas sobre caminos y vecindades dentro de la red.

3.1. Modelo de datos

El modelo de datos en Neo4j se compone de los siguientes elementos:

- **Nodos `User`**: representan a los usuarios del sistema.
  Cada nodo contiene al menos un identificador único (`user_id`) y puede almacenar propiedades adicionales como el nombre del usuario.

- **Relaciones `CAN_MESSAGE`**: representan la posibilidad de que un usuario envíe mensajes a otro.
  Estas relaciones son dirigidas y almacenan información adicional relevante para el sistema.

Cada relación `CAN_MESSAGE` incluye las siguientes propiedades:

- `rel_id`: identificador único de la relación, construido a partir de los identificadores de los usuarios.
- `msg_count`: contador del número total de mensajes enviados a través de dicha relación.
- Propiedades adicionales opcionales que permiten extender el modelo sin modificar su estructura básica.

3.2. Operaciones básicas

Sobre el modelo descrito se implementan operaciones básicas a través de la API de Neo4j:

- Creación, actualización y obtención de usuarios.
- Creación y obtención de relaciones de mensajería.
- Incremento del contador de mensajes asociado a una relación.
- Listado de usuarios y obtención de vecinos dentro del grafo.

Todas estas operaciones se encapsulan en la clase `Neo4jAPI`, evitando el uso directo de consultas Cypher en el resto del sistema.

3.3. Consultas sobre el grafo

Además de las operaciones básicas, se implementan consultas más avanzadas sobre el modelo de grafo.

3.3.1. Vecinos de un usuario

Se implementa una consulta que permite obtener los usuarios conectados a un usuario dado mediante relaciones de mensajería. Esta consulta no distingue entre relaciones entrantes o salientes, ya que para el sistema únicamente es relevante la existencia de una conexión.

Los resultados pueden ordenarse según distintos criterios, como el número total de mensajes intercambiados, lo que permite identificar las conexiones más activas de un usuario.

3.3.2. Caminos ponderados entre usuarios

El sistema incluye una consulta que permite obtener el camino óptimo entre dos usuarios teniendo en cuenta no solo la longitud del camino, sino también el volumen de mensajes intercambiados.

Dado que no se dispone del plugin Graph Data Science (GDS), se define un coste para cada relación de mensajería basado en el número de mensajes enviados. Concretamente, el coste de una relación se define como:

$\text{coste} = \frac{1}{\text{msg_count} + 1}$

De este modo, las relaciones con mayor número de mensajes tienen un coste menor. El camino óptimo se obtiene minimizando el coste total acumulado a lo largo del camino, priorizando así rutas con mayor volumen de comunicación.

Esta aproximación permite implementar un criterio de selección de caminos coherente con los objetivos del sistema sin depender de extensiones externas de Neo4j.

3.4. Decisiones de diseño

Las principales decisiones de diseño relacionadas con Neo4j son:

- Uso de un modelo de grafo dirigido para representar relaciones de mensajería.
- Almacenamiento del contador de mensajes en la relación, evitando cálculos costosos sobre el histórico.
- Definición explícita de un identificador de relación (`rel_id`) para facilitar la integración con Redis.
- Implementación de consultas avanzadas sin depender del plugin GDS, garantizando la portabilidad del sistema.

Estas decisiones permiten un uso eficiente de Neo4j y una integración clara con el resto de componentes del sistema.

---

4. Diseño y uso de Redis

Redis se utiliza en el sistema como un sistema de colas en memoria que permite desacoplar el envío de mensajes de su consumo y almacenamiento persistente. Su baja latencia y su soporte para estructuras de datos como listas lo convierten en una opción adecuada para este propósito.

4.1. Modelo de datos en Redis

Redis almacena los mensajes en **listas**, utilizando una cola independiente por cada relación de mensajería entre usuarios. Cada cola se identifica de forma única mediante el identificador de la relación (`rel_id`), lo que permite mantener separadas las comunicaciones entre distintos pares de usuarios.

La clave de cada cola sigue el formato:

queue:<rel_id>

donde `rel_id` coincide con el identificador de la relación almacenada en Neo4j. Este diseño garantiza la coherencia entre el grafo y el sistema de colas.

Los mensajes se almacenan en formato JSON, tras serializar la estructura canónica del mensaje definida en el sistema.

4.2. Estructura de los mensajes

Todos los mensajes que se almacenan en Redis siguen una estructura uniforme, representada por la clase `MessageData`. Esta estructura incluye, entre otros, los siguientes campos:

- Identificador único del mensaje.
- Marca temporal (`timestamp`) en formato UTC.
- Usuario emisor y receptor.
- Identificador de la relación de mensajería.
- Contenido del mensaje.
- Metadatos opcionales.

El uso de una estructura común facilita la interoperabilidad entre Redis, MongoDB y el sistema de snapshots.

4.3. Envío de mensajes

Cuando se envía un mensaje, este se inserta al final de la cola correspondiente mediante una operación `RPUSH`. Esta operación es rápida y no bloqueante, lo que permite que el sistema acepte mensajes de forma eficiente incluso bajo alta carga.

Redis actúa en este punto como un buffer temporal, manteniendo los mensajes en memoria hasta que son consumidos.

4.4. Consumo de mensajes

El consumo de mensajes se realiza mediante una operación bloqueante (`BLPOP`) con un tiempo de espera configurable. Este enfoque permite:

- Esperar de forma eficiente a la llegada de nuevos mensajes.
- Evitar bucles activos de sondeo.
- Liberar el control si no hay mensajes disponibles.

Una vez consumido, el mensaje se elimina de la cola y se deserializa a un objeto `MessageData`, que posteriormente se almacena de forma persistente en MongoDB.

Este mecanismo garantiza que cada mensaje se procese una única vez y evita duplicaciones.

4.5. Desacoplamiento y tolerancia a fallos

El uso de Redis como sistema de colas introduce un desacoplamiento claro entre el envío y el procesamiento de mensajes. En caso de que el consumidor no esté disponible temporalmente, los mensajes permanecen en Redis hasta que puedan ser procesados.

Este diseño mejora la tolerancia a fallos del sistema y permite una mayor flexibilidad en la gestión de los consumidores.

4.6. Redis y snapshots

Redis forma parte del mecanismo de snapshots del sistema. En el momento de crear un snapshot, se exporta el estado completo de todas las colas, incluyendo el orden de los mensajes pendientes.

Durante la restauración de un snapshot, las colas se reconstruyen insertando nuevamente los mensajes en el mismo orden en que fueron almacenados originalmente, garantizando así la coherencia del sistema tras la

recuperación.

4.7. Decisiones de diseño

Las principales decisiones de diseño relacionadas con Redis son:

- Uso de listas como estructura de datos para implementar colas FIFO.
- Una cola independiente por cada relación de mensajería.
- Consumo bloqueante de mensajes para mejorar la eficiencia.
- Serialización de mensajes en formato JSON.
- Integración explícita con el mecanismo de snapshots.

Estas decisiones permiten un uso eficiente de Redis y una integración clara con el resto del sistema.

---

5. Diseño y uso de MongoDB

MongoDB se utiliza en el sistema como base de datos documental para el almacenamiento persistente del histórico de mensajes y como repositorio de snapshots. Su modelo flexible basado en documentos JSON resulta especialmente adecuado para almacenar mensajes con estructura variable y metadatos adicionales.

5.1. Modelo de datos en MongoDB

MongoDB almacena dos tipos principales de documentos:

- **Mensajes**: representan el histórico completo de mensajes consumidos desde Redis.
- **Snapshots**: representan una captura completa del estado del sistema en un momento dado.

Cada mensaje se almacena como un documento independiente siguiendo la estructura definida por la clase `MessageData`. MongoDB añade automáticamente un identificador interno (`_id`), que no se utiliza como identificador lógico del mensaje.

5.2. Histórico de mensajes

El histórico de mensajes contiene todos los mensajes que han sido consumidos desde Redis y procesados correctamente. Cada documento de mensaje incluye, entre otros, los siguientes campos:

- `msg_id`: identificador lógico único del mensaje.
- `timestamp`: marca temporal del envío del mensaje.
- `sender_id` y `receiver_id`: usuarios implicados.
- `rel_id`: identificador de la relación de mensajería.
- `content`: contenido del mensaje.
- `metadata`: metadatos opcionales.

Este diseño permite conservar un registro completo y persistente de la actividad del sistema, independientemente del estado de las colas en Redis.

5.3. Consultas sobre el histórico

Sobre el histórico de mensajes se implementan diversas consultas que permiten analizar la actividad del sistema:

- Obtención de los últimos mensajes enviados.
- Conteo de mensajes asociados a un usuario, ya sea como emisor, receptor o ambos.
- Agregaciones temporales que permiten contar mensajes por intervalos de tiempo (por ejemplo, por día).
- Filtrado de mensajes por relación de mensajería.

Estas consultas se implementan utilizando el framework de agregación de MongoDB, aprovechando su capacidad para procesar grandes volúmenes de documentos de forma eficiente.

5.4. MongoDB como soporte de snapshots

Además del histórico de mensajes, MongoDB almacena los snapshots del sistema. Cada snapshot contiene:

- El estado del grafo de Neo4j (usuarios y relaciones).
- El estado de las colas de Redis.
- El histórico de mensajes en MongoDB en el momento de la captura.
- Estadísticas agregadas del sistema (número de usuarios, relaciones, mensajes, etc.).

Los snapshots se almacenan como documentos independientes en una colección específica, lo que permite mantener múltiples versiones del estado del sistema.

5.5. Restauración desde snapshots

La restauración de un snapshot implica la eliminación completa del estado actual del sistema y la reconstrucción de todos los componentes a partir de la información almacenada en MongoDB.

Durante este proceso:

- Se reconstruyen los nodos y relaciones en Neo4j.
- Se restauran las colas de Redis manteniendo el orden original de los mensajes.
- Se reinsertan los mensajes históricos en MongoDB.

Para evitar conflictos, los identificadores internos (`_id`) de MongoDB no se conservan durante el proceso de snapshot, permitiendo una restauración limpia del sistema.

5.6. Decisiones de diseño

Las principales decisiones de diseño relacionadas con MongoDB son:

- Uso de MongoDB como almacenamiento persistente del histórico de mensajes.
- Separación clara entre mensajes y snapshots en colecciones distintas.
- Uso de identificadores lógicos (`msg_id`) independientes del identificador interno de MongoDB.
- Implementación de consultas mediante agregaciones para análisis temporal y estadístico.
- Uso de MongoDB como punto central para la gestión de snapshots.

Estas decisiones permiten combinar flexibilidad, persistencia y capacidad de análisis dentro del sistema.

6. Snapshots y restauración del sistema

El sistema implementa un mecanismo de snapshots que permite capturar y restaurar el estado completo de todas las bases de datos involucradas. Un snapshot representa una imagen consistente del sistema en un instante concreto, facilitando la recuperación ante errores y el análisis de la evolución del sistema.

6.1. Contenido de un snapshot

Cada snapshot almacena de forma estructurada la siguiente información:

- **Neo4j**:
  - Lista de nodos de usuarios con sus propiedades.
  - Lista de relaciones de mensajería con sus propiedades, incluyendo el contador de mensajes.
- **Redis**:
  - Estado completo de todas las colas de mensajes.
  - Orden de los mensajes pendientes en cada cola.
- **MongoDB**:
  - Histórico de mensajes existente en el momento del snapshot.
  - Información adicional del snapshot, como nombre, fecha de creación y estadísticas agregadas.

Este enfoque garantiza que el snapshot contiene toda la información necesaria para reconstruir el sistema sin depender del estado previo de ninguna base de datos.

6.2. Creación de snapshots

La creación de un snapshot se realiza mediante el servicio `SnapshotService`, que coordina la exportación del estado de cada base de datos:

1. Se exporta el estado del grafo desde Neo4j, incluyendo usuarios y relaciones.
2. Se exporta el estado de todas las colas de Redis, preservando el orden de los mensajes.
3. Se obtiene el histórico de mensajes almacenado en MongoDB.
4. Se calculan estadísticas agregadas del sistema, como el número de usuarios, relaciones y mensajes.
5. Toda esta información se almacena como un documento en la colección de snapshots de MongoDB.

El proceso de creación de snapshots no interrumpe el funcionamiento normal del sistema.

6.3. Restauración del sistema

La restauración de un snapshot implica la reconstrucción completa del sistema a partir de la información almacenada. Para garantizar la coherencia, el proceso sigue un orden estricto:

1. Eliminación del estado actual de todas las bases de datos.
2. Restauración de los nodos y relaciones en Neo4j.
3. Restauración de las colas de Redis, reinsertando los mensajes en el orden original.
4. Restauración del histórico de mensajes en MongoDB.

Este orden evita inconsistencias, como relaciones sin nodos asociados o colas sin correspondencia en el grafo.

6.4. Comparación entre snapshots

Además de la creación y restauración, el sistema permite comparar dos snapshots diferentes. Esta comparación se basa en las estadísticas almacenadas en cada snapshot e incluye:

- Número total de mensajes.
- Número de usuarios.
- Número de relaciones de mensajería.
- Número de mensajes pendientes en colas Redis.

La comparación se realiza calculando la diferencia entre los valores de dos snapshots seleccionados, lo que permite analizar de forma cuantitativa la evolución del sistema entre distintos estados.

6.5. Ventajas del enfoque

El mecanismo de snapshots proporciona varias ventajas:

- Permite recuperar el sistema ante fallos o errores.
- Facilita la validación del correcto funcionamiento del sistema.
- Permite analizar la evolución de la red de mensajería y del tráfico de mensajes.
- Proporciona una base sólida para posibles extensiones futuras, como auditorías o análisis históricos.

6.6. Decisiones de diseño

Las principales decisiones de diseño relacionadas con los snapshots son:

- Uso de MongoDB como almacenamiento centralizado de snapshots.
- Captura explícita del estado de todas las bases de datos.
- Restauración completa del sistema en lugar de restauraciones parciales.
- Inclusión de estadísticas agregadas para facilitar la comparación entre snapshots.

Estas decisiones garantizan un mecanismo de snapshots robusto y coherente con la arquitectura del sistema.

---

7. Decisiones de diseño y conclusiones

7.1. Decisiones de diseño

A lo largo del desarrollo del sistema se han tomado diversas decisiones de diseño con el objetivo de garantizar coherencia, modularidad y facilidad de mantenimiento. Las más relevantes son las siguientes:

- **Separación de responsabilidades**: cada base de datos cumple un rol específico dentro del sistema (Neo4j para el grafo, Redis para colas y MongoDB para persistencia), evitando solapamientos innecesarios.
- **Uso de APIs independientes**: todas las operaciones sobre las bases de datos se encapsulan en APIs específicas, lo que desacopla la lógica de acceso a datos del resto del sistema.
- **Modelo de mensajes unificado**: se define una estructura canónica de mensaje (`MessageData`) que se utiliza de forma consistente en Redis, MongoDB y durante los snapshots.

- **Colas por relación**: Redis mantiene una cola independiente por cada relación de mensajería, lo que simplifica el consumo y garantiza el orden de los mensajes.
- **Contador de mensajes en Neo4j**: el número de mensajes se almacena directamente en la relación de mensajería, evitando cálculos costosos sobre el histórico.
- **Consultas de caminos sin dependencias externas**: al no disponer del plugin Graph Data Science, se implementa una solución alternativa basada en un coste inverso al número de mensajes, manteniendo la portabilidad del sistema.
- **Snapshots completos**: los snapshots capturan el estado completo del sistema y se restauran de forma integral, evitando inconsistencias parciales.

Estas decisiones permiten un sistema claro, extensible y coherente con los objetivos de la práctica.

7.2. Conclusiones

La práctica ha permitido diseñar e implementar un sistema de mensajería distribuido que integra tres bases de datos NoSQL con modelos y finalidades diferentes. El uso combinado de Neo4j, Redis y MongoDB demuestra cómo distintas tecnologías pueden complementarse para resolver un problema común de forma eficiente.

El sistema resultante soporta el envío y consumo de mensajes, consultas avanzadas sobre grafos y datos históricos, así como mecanismos de snapshot y restauración del estado completo. La arquitectura modular facilita la comprensión del sistema y permite futuras extensiones, como la incorporación de nuevos tipos de consultas o mecanismos de análisis más avanzados.

En conjunto, la práctica cumple con los objetivos propuestos y proporciona una visión práctica del diseño de sistemas distribuidos apoyados en múltiples bases de datos.
