Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial 1 - Sala de Emergencias

72.42 - Programación de Objetos Distribuidos

Informe Técnico: Diseño e Implementación de Servicios Concurrentes.

Integrantes

Grupo Número 5

Madero Torres, Eduardo Federico - 59494

Limachi, Desiree Melisa - 59463

Comisión S - Segundo Cuatrimestre del 2024

Introducción: Estructura del Sistema

El sistema de servicios concurrentes para la gestión de emergencias médicas se ha diseñado utilizando un enfoque modular y orientado a servicios. La arquitectura se basa en el principio de "una clase por servicio", lo que permite una clara separación de responsabilidades y facilita el mantenimiento.

Los principales servicios implementados son:

1. **EmergencyAdminService**: Gestiona la administración de doctores y salas de emergencia.
2. **EmergencyCareService**: Maneja el proceso de atención de emergencias, incluyendo la asignación de pacientes a doctores y salas.
3. **WaitingRoomService**: Administra la lista de espera de pacientes y sus niveles de emergencia.
4. **QueryService**: Proporciona funcionalidades para consultar el estado del sistema, incluyendo el estado de las salas y la lista de espera.

Cada uno de estos servicios se implementa como una clase separada, lo que permite una clara delimitación de responsabilidades y facilita la implementación de la lógica específica de cada servicio. Además, se han creado repositorios correspondientes (DoctorRepository, PatientRepository, RoomRepository) para manejar el almacenamiento y recuperación de datos de manera eficiente y thread-safe.

Esta estructura modular no solo mejora la organización del código, sino que también facilita la implementación de mecanismos de concurrencia específicos para cada servicio, permitiendo un manejo más granular y eficiente de los recursos compartidos.

Decisiones de diseño e implementación de los servicios

1. Uso de Blocking Stub para servicios sincronizados

El sistema utiliza `BlockingStub` para manejar servicios sincronizados. Esta decisión se basa en la necesidad de operaciones síncronas y secuenciales en el manejo de emergencias médicas. Por ejemplo, en `EmergencyAdminClient.java`:

```
emergencyAdminServiceGrpc.emergencyAdminServiceBlockingStub blockingStub =
emergencyAdminServiceGrpc.newBlockingStub(channel);
```

El uso de Blocking Stub permite un flujo de control más sencillo y predecible, donde el orden de las operaciones es importante, asegura que las operaciones se ejecuten de manera síncrona, lo que garantiza que cada paso en el flujo del sistema se complete antes de iniciar el siguiente. Esto es esencial en un entorno de emergencias donde el orden de las acciones, como la asignación de médicos o la disponibilidad de consultorios, es crucial para la integridad y consistencia del sistema.

2. Uso de readWriteLock en operaciones de adición y actualización

Se utiliza `ReentrantReadWriteLock` en los repositorios (por ejemplo, `DoctorRepository.java`) para proteger las operaciones de escritura mientras permite múltiples lecturas simultáneas.

Se eligió utilizar `ReentrantReadWriteLock` en los repositorios debido a su capacidad para mejorar el rendimiento y garantizar la integridad de los datos en un entorno concurrente. La razón principal es que este tipo de bloqueo permite que múltiples hilos accedan simultáneamente a los datos para lecturas, pero restringe el acceso a un solo hilo durante las escrituras.

```
private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();

public Doctor addDoctor(String doctorName, Integer level) {
    rwLock.writeLock().lock();
    try {
        // Operación de escritura
    } finally {
        rwLock.writeLock().unlock();
    }
}
```

En las operaciones de lectura, como `getDoctor`, `getPatient`, o `getRoom`, no utilizamos explícitamente el read lock del `ReentrantReadWriteLock`. La decisión de no utilizar `ReentrantReadWriteLock` en operaciones de lectura se basa en varios factores clave. Primero, las lecturas son generalmente más frecuentes que las escrituras, por lo que evitar el overhead de adquirir y liberar locks mejora significativamente el rendimiento. Además, el uso de `ConcurrentHashMap` para almacenar datos proporciona thread-safety en operaciones de lectura sin necesidad de locks adicionales. Este enfoque se alinea con el principio de consistencia eventual, aceptable en sistemas de alta concurrencia, donde las lecturas pueden ocasionalmente devolver datos desactualizados en favor de un mejor rendimiento. Finalmente, `ConcurrentHashMap` garantiza que operaciones individuales como 'get' son atómicas, lo que significa que, incluso sin locks explícitos, no se obtendrán datos en estados intermedios o corruptos. Esta estrategia equilibra eficazmente la necesidad de consistencia con un alto rendimiento en las operaciones de lectura.

3. Uso de bloques synchronized en lugar de métodos enteros

En los servicios, se utilizan bloques `synchronized` a nivel de método para garantizar la exclusión mutua y prevenir condiciones de carrera. Esto se puede observar por ejemplo en el servicio `EmergencyCareService`.

```
private synchronized List<EmergencyCareResponse>
startEmergencyCareInFreeRooms(StreamObserver<EmergencyCareListResponse>
responseObserver)
```

Además del uso de métodos synchronized en los servicios, el sistema emplea ReentrantReadWriteLock en los repositorios para operaciones críticas de escritura. Por ejemplo, en el DoctorRepository, se utiliza este mecanismo para métodos como addDoctor y setAvailabilityDoctor. Este enfoque permite múltiples lecturas simultáneas mientras asegura la integridad de los datos durante las operaciones de escritura. La decisión de usar

ReentrantReadWriteLock en estos casos se basa en la necesidad de un control más granular sobre las operaciones concurrentes en los repositorios.

Es importante destacar la distinción entre diferentes tipos de operaciones de escritura en el sistema. Mientras que los métodos de escritura en los repositorios están protegidos por ReentrantReadWriteLock, los setters simples en las clases de modelo (como Doctor, Patient, Room) no tienen locks adicionales. Estos setters se consideran seguros en el contexto de su uso, ya que las operaciones que los invocan (como setAvailabilityDoctor en el repositorio) ya están protegidas por locks apropiados. Esta estrategia permite un equilibrio entre la seguridad de los datos y el rendimiento del sistema.

Para las operaciones de lectura en los repositorios, como getDoctor, no se utiliza explícitamente el read lock del ReentrantReadWriteLock. Esto se debe al uso de ConcurrentHashMap, que es thread-safe para operaciones de lectura. Esta decisión mejora el rendimiento al evitar el overhead de adquirir y liberar locks para cada lectura, aceptando un modelo de consistencia eventual. En este modelo, las lecturas pueden ocasionalmente devolver datos ligeramente desactualizados, lo cual se considera aceptable en favor de un mejor rendimiento en un sistema de alta concurrencia como el de gestión de emergencias médicas.

Criterios aplicados para el trabajo concurrente

A. Uso de ConcurrentHashMap para almacenamiento de datos.

Se utiliza `ConcurrentHashMap` en los repositorios para manejar colecciones compartidas de manera thread-safe:

```
private final ConcurrentHashMap<String, Doctor> doctors;
```

Como se mencionó previamente, esto proporciona operaciones atómicas para lecturas y escrituras simples sin necesidad de sincronización externa adicional.

B. Métodos synchronized en servicios:

Esto proporciona operaciones atómicas para lecturas y escrituras simples sin necesidad de sincronización externa adicional.

C. ReentrantReadWriteLock en repositorios:

Para permitir múltiples lecturas concurrentes mientras se protegen las operaciones de escritura.

Manejo de errores

El sistema utiliza un enfoque basado en excepciones para el manejo de errores. En los clientes, se emplea un método `executeHandling` para manejar las excepciones de manera uniforme:

```
DoctorResponse addResponse = ClientActionHandler.executeHandling(() ->
blockingStub.addDoctor(addRequest));
```

Este enfoque centraliza el manejo de errores y proporciona una forma consistente de informar sobre los problemas al usuario o al sistema de logging.

En el lado del servidor, se utilizan respuestas **de error gRPC** para comunicar problemas al cliente:

```
responseObserver.onError(Status.ALREADY_EXISTS .withDescription("Doctor " + request.getDoctorName() + " already exists") .asRuntimeException());
```

Este método permite una comunicación clara y estructurada de los errores entre el servidor y el cliente.

Mejoras potenciales y expansiones

Una mejora significativa sería la implementación de un sistema de notificaciones en tiempo real para médicos y personal, lo que optimizaría la comunicación en el manejo de emergencias. Esta funcionalidad el grupo no lo realizo por ser menos integrantes, pero sabemos lo que mejoraria a la comunicación de espera y atención para los pacientes. Además, para mejorar el rendimiento y evitar bloqueos innecesarios, se podría integrar el uso de FutureStub junto con listeners, lo que permitiría realizar las operaciones de notificación de manera asíncrona, liberando al sistema de esperas activas y mejorando la capacidad de respuesta.

Finalmente, se podría explorar el uso de FutureStub en lugar de BlockingStub en ciertas operaciones. Mientras que BlockingStub es apropiado para operaciones síncronas como fue para nuestro caso, FutureStub podría ofrecer mejor rendimiento en escenarios donde la asincronía es aceptable, permitiendo un mejor manejo de la concurrencia y evitando bloqueos innecesarios.

La optimización en la lectura de scripts es otra área de mejora potencial. El sistema actual utiliza un switch para manejar diferentes acciones, lo cual puede volverse ineficiente a medida que se agregan más funcionalidades. Se podría implementar un patrón Scripts o un ScriptProcessor más sofisticado para manejar las acciones de manera más escalable y mantenible.

El manejo de errores podría mejorarse significativamente. Se podría implementar un sistema de manejo de excepciones global, similar al proporcionado por la cátedra, para centralizar y estandarizar el tratamiento de errores en toda la aplicación. Esto mejoraría la robustez del sistema y facilitaría el debugging y mantenimiento.

La implementación de estas mejoras no solo aumentaría la eficiencia y robustez del sistema, sino que también lo haría más escalable y adaptable a futuras necesidades en la gestión de emergencias médicas.

.