72.42 Programación de Objetos Distribuidos

Juan Ignacio Matilla, 60459
jmatilla@itba.edu.ar
Felipe Hiba, 61219
fhiba@itba.edu.ar
Patricio Escudeiro, 61156
pescudeiro@itba.edu.ar

26 de Septiembre de 2024

Decisiones de Diseño

Para la implementación del sistema se tomaron una serie de decisiones de diseño enfocadas en la modularización y la separación clara de responsabilidades. Estas decisiones incluyen:

Estructura del Servidor:

Se dividió el servidor en cuatro módulos clave:

- **Models:** Aquí se encapsulan las entidades del sistema: Doctor, Patient, Room y CareHistory. Estas clases representan la información básica manejada en el hospital, como los médicos, los pacientes y las atenciones médicas.
- **Services:** Este módulo maneja la lógica de negocio. Todas las reglas relacionadas con el proceso de atención de emergencias, asignación de médicos y consultorios, así como la gestión de la disponibilidad de los médicos, se encuentran aquí.
- **Servants:** Esta capa se encarga de la lógica de comunicación de gRPC, asegurando la correcta implementación de los servicios remotos especificados en el enunciado.
- **Repositories:** Los repositorios manejan la persistencia en memoria de los datos.

Manejo de Errores en los Servicios:

Se decidió manejar los errores potenciales directamente en los archivos .proto, mediante la inclusión de un campo booleano success y mensajes de error personalizados. Esta aproximación simplifica el manejo de errores, facilitando tanto el desarrollo como la interpretación de los resultados por parte del cliente.

Pruebas:

Para validar las funcionalidades desarrolladas, se crearon pruebas automatizadas utilizando scripts en bash, que ejecutan los diferentes clientes y validan los comportamientos esperados de cada servicio según lo especificado en la consigna.

Estructura de Datos:

Se empleó una estructura TreeSet tanto para médicos como para patients y los rooms en el módulo de repositorios. Esta decisión fue clave para garantizar que no hubiera duplicados y que los elementos estuvieran ordenados de la manera pedida, a través de la implementación de Comparable en los modelos, lo que optimiza el manejo de estos datos según lo especificado en el trabajo. Los repositorios específicos son:

- **PatientRepository:** Un conjunto de pacientes manejado por un synchronized set que fue inicializado con un TreeSet de la clase collections.synchronizedSet.
- **DoctorRepository:** Un conjunto de doctores también manejado por un synchronized set, inicializado con un TreeSet de la clase collections.synchronizedSet.
- **CareHistoryRepository:** Una cola de CareHistory, instanciada como una ConcurrentLinkedQueue.
- **RoomRepository:** Un conjunto de habitaciones, manejado por un synchronized set inicializado con un TreeSet de la clase collections.synchronizedSet.

Estructura del Cliente:

El cliente fue diseñado con una clase abstracta Action y una clase Client que implementa Closable. Esto asegura que cualquier conexión abierta sea correctamente cerrada al finalizar una operación. Posteriormente, se implementaron clases Action específicas para cada servicio, extendiendo las funcionalidades requeridas en el documento.

Estas decisiones permitieron desarrollar un sistema modular, fácil de mantener y extensible, asegurando la separación de las responsabilidades y facilitando el agregamiento de nuevas funcionalidades.

Decisiones de Concurrencia

Para garantizar la seguridad en el acceso a las estructuras de datos compartidas, se tomaron varias decisiones relacionadas con la concurrencia:

1. **Uso de Estructuras Synchronized:**
    - Primero, revisamos la documentación de Java para identificar los métodos que garantizan que sean **thread-safe** en los `SynchronizedSet` utilizados. Notamos que era necesario utilizar una sección `synchronized` al iterar sobre las colecciones y al aplicar operaciones de stream complejas.
    - Por ello, añadimos `synchronized` en los lugares donde se iteraba sobre estas colecciones y donde se realizaban operaciones mediante streams. Por ejemplo, en la función `CareAllPatient`, decidimos que la función entera fuera `synchronized` para evitar problemas de concurrencia en caso de que múltiples hilos intentaran ejecutar el método simultáneamente.
2. **Uso de CountDownLatch:**
    - Implementamos un `CountDownLatch` para manejar la funcionalidad de `register` y `unregister` del `doctorPager`. Esta implementación permite que el sistema se quede esperando eventos para loguear, asegurando que las operaciones de registro y anulación se manejen de manera ordenada y controlada.

Posibles Mejoras

- **Extensión de modelos generados por gRPC**: Sería recomendable que los modelos del sistema extiendan directamente las clases generadas por los archivos `.proto`. Esto permitiría una integración más transparente con gRPC, eliminando la necesidad de mapear manualmente los objetos de entrada y salida en cada servicio. Este cambio no solo simplificará la implementación, sino que también reduciría la posibilidad de errores de conversión y mejoraría la mantenibilidad del código.
- **Uso de estructuras de datos concurrentes**: Actualmente, el sistema utiliza `SynchronizedSet` para garantizar la sincronización en la lectura y escritura de datos críticos como los doctores y pacientes. Sin embargo, una mejora sería reemplazar estos `SynchronizedSet` con `ConcurrentSet` utilizando `ConcurrentHashMap`, lo que permitiría un manejo más eficiente en escenarios de alta concurrencia, mejorando el rendimiento general del sistema, al mismo tiempo se debería tener en cuenta nuevos casos para aplicar el synchronized.
- **Pruebas unitarias adicionales**: Aunque el sistema ya cuenta con pruebas automatizadas en Bash, sería beneficioso incorporar pruebas unitarias más granulares utilizando JUnit. Esto permitiría un mayor control sobre la validación de cada componente individual, mejorando la capacidad de detectar errores específicos en las funcionalidades críticas antes de realizar pruebas integrales.
- **Reutilización de strings:** Para las notificaciones utilizamos un switch para decidir qué mensaje imprimir dependiendo de la notificación, pero al ser siempre un String, este se podría reutilizar y reducir la lógica en el cliente. Al margen de esta mejora, una forma mejor de manejar estos mensajes sería crear una clase para cada mensaje específico dentro de los protos, y así hacer que el cliente genere el mensaje y no el servidor.