Programación de Objetos Distribuidos

Trabajo Práctico Especial

Integrantes:

Bendayan, Alberto (Legajo: 62786)

Boullosa Gutierrez, Juan Cruz (Legajo: 63414)

Deyheralde, Ben (Legajo: 63559)

Profesores:

Meola, Franco Román

Turrin, Marcelo Emiliano

Para implementar las funcionalidades de los Servants del lado del servidor, se crearon 5 repositorios que almacenan la información durante la ejecución del mismo. Estos repositorios son:

Attention Repository: es el repositorio encargado de almacenar las atenciones. Se decidió implementarlo con las siguientes colecciones:

- HashMap para las atenciones comenzadas cuya clave es el número de habitación y el valor es la atención. La idea es poder buscar de forma eficiente si alguien está siendo atendido en la habitación.
- ArrayList para las atenciones terminadas ya que deben estar en orden de finalización y como se agregan siempre al final nos sirve para esto.

Sincronización: para sincronizar este repositorio, al ser dos colecciones distintas, se decidió utilizar un *ReentrantReadWriteLock*. Las funciones de escritura como por ejemplo *startAttention* o *finishAttention* las encerramos con *lock.writeLock()* y las de lectura como por ejemplo *existAttention* con *readLock()*. De esta manera se asegura que si alguien escribe sobre estas colecciones, nadie más pueda acceder ni para escribirlas ni para leerlas. Mientras que si alguien está leyendo, pueden entrar más lectores pero no escritores que modifiquen lo que están leyendo los lectores. De esta forma evitamos bloqueos globales.

Doctor Repository: es el repositorio encargado de almacenar los doctores. Se decidió implementarlo con una única colección. Esta colección en un principio era un TreeMap con clave nombre del doctor y como valor el doctor en sí mismo. Se eligió un TreeMap para tener a los doctores ordenados alfabéticamente respetando la consigna.
Sincronización: como en este repositorio solo se maneja una colección, luego de investigar las colecciones concurrentes que provee Java, se modificó el *TreeMap* por un *ConcurrentSkipListMap* ya que esta clase mantiene las claves ordenadas naturalmente, es thread-safe por ende a la hora de escribir el código no hay que hacerlo de forma manual, lo cual puede generar problemas si no se hace de la manera correcta y debido a su arquitectura, las lecturas pueden ocurrir sin bloqueos y a su vez, no tiene bloqueos globales.

Patient Repository: es el repositorio donde se guardan los pacientes en la sala de espera y los pacientes históricos con atenciones finalizadas o en curso. Para esto se decidió utilizar dos estructuras:

- Vector de SortedSet: un vector de 5 posiciones donde cada una corresponde con el nivel de emergencia del paciente en espera. Dentro de cada nivel hay un SortedSet que ordena los pacientes en orden de llegada, de este modo se atiende dependiendo del nivel al paciente que llegó primero.

- HashMap para todos los pacientes que ingresaron al hospital para poder verificar ante una nueva entrada si el paciente ya fue ingresado y en ese caso rechazarlo.

Sincronización: al ser dos colecciones distintas, para sincronizarlas se utilizó la misma estrategia que en el Attention Repository, es decir, un *ReentrantReadWriteLock*.

Room Repository: es el repositorio donde se guardan las habitaciones y está modelado con un ArrayList. Por un lado nos permite encontrar una habitación de forma muy sencilla ya que cada una se almacena en la posición de su número menos uno. Por ende la búsqueda es O(1) y además el crecimiento de la lista es dinámico.
Sincronización: se utilizó *ReentrantReadWriteLock* ya que no se encontró una implementación de Java ideal que sea concurrente y que funcione de la manera esperada. Se había utilizado Collections.synchronizedList pero según la documentación es necesario manejar la concurrencia manualmente en algunos casos asique se optó por usar el ArrayList clásico con locks.

NotificationRepository: En este repositorio se encuentran todas las notificaciones de los doctores que están suscritos. En este caso, se utiliza un concurrentHashmap cuya clave es el nombre del doctor registrado y el valor guardado es una queue de notificaciones. Esta queue es en particular un *ConcurrentLinkedQueue*. Entonces buscar la lista de notificaciones es O(1), agregar notificaciones es O(1) y también obtener la notificación más antigua es O(1). Esto es muy bueno en términos de eficiencia y es el motivo por el cual se eligió esta arquitectura..
Sincronización: Los motivos por los cuales se utilizan estas clases es porque permiten realizar operaciones concurrentes sin tener que realizar bloqueos globales y a su vez ambas clases garantizan la concurrencia en su implementación ya que son thread safe.

En resumen, se buscó utilizar clases que sean lo más eficientes posibles en función de las necesidades de cada servicio y en los casos donde fue posible se usaron clases cuya implementación ya sea thread-safe evitando en la mayoría de los casos de tener que garantizar esto de forma manual. De todas formas, siempre se buscó que los bloqueos no sean globales.

Salidas: De acuerdo con la consigna establecida, se ha decidido implementar un mecanismo de manejo de excepciones para las condiciones definidas como "falla si". En estos casos, se lanzarán excepciones que serán capturadas y gestionadas en el cliente, donde se procederá a imprimir el mensaje de error enviado por el servidor. En contraste, para todas las operaciones que se realicen de manera exitosa, los mensajes

correspondientes se enviarán y procesarán de manera habitual, sin intervención del manejo de excepciones.

Observación: Se modificó el puerto del servidor de 50051 a 50052 porque uno de los miembros del grupo usa Mac y no era posible conectarse al puerto 50051.