Instituto Tecnológico de Buenos Aires

Informe

72.42 - Programación de Objetos Distribuidos

Tepedino, Cristian - 62830
Mendonca, Juana - 61185
Bloise, Luca - 63004

Decisiones de diseño e implementación de los servicios

Para el desarrollo del sistema, se decidió separar la lógica de la aplicación de la siguiente forma:

Los clientes se encargan de tomar los parámetros, verificar que no falten los requeridos y que sus valores sean válidos, y enviar el pedido al servidor para procesarlo.

El servidor tiene 3 principales categorías de clases: Los servicios, los repositorios, y los modelos.

Los servicios se encargan de traducir los tipos de datos recibidos de los clientes a los usados por los repositorios, pedir el procesado de la acción requerida, y enviar la respuesta al cliente.

Los repositorios contienen la lógica de negocio del servidor. Se encargan de guardar el estado y de dar los resultados de las operaciones pedidas por los clientes

Los modelos son clases que usan los repositorios para abstraer el estado de la aplicación. Al crear clases específicas para esto, se pueden usar como clases parametrizadas en las colecciones que usan los repositorios

Desde el lado de los clientes, al ser en su mayoría requests simples que solamente esperan al resultado del servidor para luego imprimirlo, se decidió optar por el uso de BlockingStubs, dado que son más simples de implementar y no se tenía nada que realizar mientras se esperaba la respuesta del servidor. La única excepción es con el servicio de notificación al personal, para el cual se usa server-side streaming a fin de enviar la notificación de cada evento al momento en que ocurra. Esto llevó a que desde el lado del cliente se use un Stub, con un Observer que imprime las respuestas del servidor a medida que van llegando.

Criterios aplicados para el trabajo concurrente

Se utilizaron clases de java.util.concurrent para la sincronización de las colecciones de datos en los repositorios. A su vez, se utilizaron bloques synchronized para evitar que algunas partes del código puedan ejecutarse simultáneamente, previniendo problemas de race conditions cuando se requiere utilizar el estado de múltiples colecciones en conjunto.

También se usó un ReentrantLock en la clase Doctor, el cual es utilizado para mantener fijo el estado de un médico durante el tiempo en que se elige como candidato para atender una emergencia hasta que se decida si será atendida por ese doctor o no. Usar un lock en este caso nos permitió escribir este algoritmo de forma más sencilla que de haber tratado de reemplazarlo por bloques synchronized.

Potenciales puntos de mejora y/o expansión

Un primer punto de expansión sería aumentar las capacidades de los clientes, de forma tal que permitan indicar secuencias de operaciones en una única operación o realizar operaciones más complejas con las respuestas en lugar de solo imprimirlas, y en tal caso se podrían usar Stubs no bloqueantes para realizar operaciones mientras se espera la respuesta del servidor.

Siguiendo con la idea de expandir en los clientes, otra mejora podría ser la creación de una aplicación de interfaz visual que, usando los clientes y el servidor creado, puedan presentar la información e interactuar con el usuario de forma más amigable.

Por limitaciones de tiempo, no fuimos capaces de implementarlas para esta entrega, pero se consideraron algunas ideas que podrían simplificar el manejo de la la concurrencia en el servidor. Una de estas es hacer a los modelos inmutables, para evitar tener que tener en cuenta cambios de status en los doctores o los consultorios, por ejemplo. Otro posible cambio que se nos ocurrió es mantener múltiples colecciones de doctores, en vez de usar un enum para su estado. Tendría la desventaja de tener que realizar operaciones de agregado y remoción para cada vez que se cambie el estado, pero podría resultar en una búsqueda más sencilla de doctores disponibles.