Trabajo Práctico Especial Programación de Objetos Distribuidos

Marcelo Lynch - 56287
Juan Godfrid - 56609
Pablo Radnic - 57013
Joaquín Ormachea - 57034

En este documento, se va a estar haciendo una breve descripción de cómo funciona el sistema. Además daremos nuestras explicaciones acerca de algunas decisiones cruciales tomadas en el diseño del mismo.

En el *readme* del proyecto se encuentran las instrucciones para correr el sistema. Esto se hará a partir de diferentes *bash scripts* que *wrapean* la ejecución de las distintas partes.

El proyecto consiste en 3 partes:

- **“api”** donde se encuentran las interfaces para la comunicación de los servicios y los clientes
- **“client”** donde se desarrollan los clientes que luego van a ser corridos para conectarse con la tercera parte del proyecto
- **“server”** donde se encuentra la lógica del sistema de votación y el manejo de la interacción con los clientes y resolución de sincronización

En el servidor se implementa y se exporta una clase por cada servicio, pero la mayor parte de la lógica y (más importantemente) la sincronización se delega en la clase **ElectionManager**, con la que todos los servicios se comunican. Allí es donde se guarda el estado de la elección y se procesan los pedidos.

En cuanto al cliente, se implementan cuatro aplicaciones distintas. Cabe mencionar que para el cliente de fiscalización se implementó un modelo de callbacks, donde se exporta un objeto remoto desde el cliente que luego invocará el servidor.

Consideraciones de sincronización y concurrencia

ElectionManager implementa un esquema de lectores/escritores para la sincronización. Para esto utilizamos un **ReentrantReadWriteLock,** que provee un lock para escritura (exclusivo) y para lectura (que pueden compartir muchos lectores). Se utiliza el modo *fair* de este lock, que evita problemas de de *starvation*. Es de notar que el beneficio de usar un esquema de lectores y escritores no es muy grande en el caso de nuestro sistema (frente a una exclusión mutua más estricta), porque se espera que la mayoría de las operaciones concurrentes sean de escritura (al ser una elección), pero semánticamente tiene sentido.

El estado del sistema que puede sufrir escrituras está en

- el estado de la elección (`electionState`), y
- la lista de votos

No implementamos locks distintos para estos dos recursos: solo existe un lock de escritura que protege al mismo tiempo los dos. Esto nos asegura consistencia frente a casos como:

1. Si alguien está consultando la lista de votos para computar resultados parciales (leyendo la lista de votos) entonces no se le "cierra" la elección en el medio (ante una escritura de `electionState`).

2. Sí hay votos todavía pendientes de emisión cuando llega un pedido de cerrar la elección, van a estar en la cola de este único lock y van a entrar antes de que cierre la elección.

Algo a notar es que no adquirimos el lock de lectura al consultar `electionState`. Esto podría llevar a condiciones de carrera desde un punto de vista técnico, por ejemplo: si alguien está intentando registrar un fiscal en la votación puede obtenga luz verde porque el estado se lee como "no comenzada". Mientras se procesa ese pedido puede llegar otro que pasa la elección a "abierta", lo cual prohibiría el registro del fiscal. Decidimos no proteger contra esto pues no afecta la consistencia del sistema ni las expectativas de comportamiento por parte de los usuarios (pues es incomprobable que se haya producido esa condición de carrera o simplemente hayan llegado en el orden correcto). Esto permite pasar un poco de lógica al servicio en lugar de hacerlo todo desde ElectionManager.

Una vez que cierra la elección, los resultados no cambiarán, por lo que los cacheamos luego de la primera vez que los piden. Así, el método que calcula los resultados nacionales es synchronized para asegurar que el cálculo se realiza una vez, el que calcula los provinciales lo hace dentro de un bloque sincronizado contra el enum de las provincias, y también se crean locks para cada mesa para lograr ese mismo objetivo. Para garantizar thread-safety se guardan los resultados de las mesas en un array fijo, y los de las provincias en un EnumMap (que internamente utiliza un array).

En cuanto a uso de threads, los utilizamos en dos lugares: en el cliente de votación se utiliza `parallelStream` para enviar los votos, aprovechando que la lista de votos completa está en memoria cuando cargamos el CSV. Del lado del servidor, utilizamos un **CachedThreadPoolExecutorService** para llamar a los callbacks de los clientes de fiscalización: esto es porque la llamada al callback es bloqueante, y tenemos una potencial denegación de servicio si esto se hace en un mismo thread siempre.

Mejoras futuras

Con respecto a las mejoras, creemos que es técnicamente posible mejorar un poco la concurrencia en algunas ocasiones liberando antes el lock de lectura: esto es porque la mayoría de las operaciones de cálculo de resultados copian rápidamente la lista (con un collect), y luego no están más consultando el recurso que se debe proteger. No se hizo esto porque no se consideró crítico y además porque perderíamos modularidad (la lógica del cálculo estaría mezclada con la de sincronización).

De la misma forma, es posible que se puedan paralelizar más las operaciones de cálculo de votos (por ejemplo, utilizando más parallel streams en donde usamos streams). No lo consideramos crítico y decidimos no implementarlo a ciegas (siguiendo al ítem 48 de Effective Java, 3a edición: *Use caution when making streams parallel*).