Informe Trabajo Práctico

72.42 - Programación de Objetos Distribuidos

2° Cuatrimestre 2019

Lund, Marcos (57159)

Ferrer, Tomás (57207)

Katan, Jonathan (56653)

Atar, Marcos Ariel (57352)

Introducción

A modo de introducción se describe la estructura general del proyecto. El proyecto consiste en tres carpetas principales: ***api***, ***client*** y ***server***.

- La carpeta ***api*** contiene las clases e interfaces generales que se usan tanto en el cliente como en el servidor. Contiene clases que contienen distintos tipos de *excepciones*, *modelos*, interfaces con *servicios* (administración, fiscalización, consulta y voto), distintos archivos *enum* (estados de la elección, partidos políticos, modos de consulta y estados) y una interfaz llamada `RemoteMonitoringClient` que representa a un fiscal cliente. Esta última contiene el método `notifyVote(Vote)`, mediante el cual se notifica al fiscal cada vez que hay un voto en su mesa para su partido político.
- La carpeta ***client*** contiene cuatro *packages* distintos, cada uno correspondiente a un tipo de cliente distinto. Cada uno de los clientes se encarga de llamar a los servicios correspondientes, ofrecidos por el server.
- Para la carpeta ***server*** las dos clases principales son:
  - *Servant*: Esta clase implementa los cuatro servicios que están disponibles para los clientes. Contiene todas las estructuras de almacenamiento necesarias para el correcto funcionamiento de los servicios implementados.
  - *Server*: Registra los cuatro servicios disponibles para poder ser invocados por el cliente, creando una instancia de *Servant*.

Implementación

A continuación se describe la implementación de cada uno de los servicios que se ofrece a los clientes.

- *Administration Service*: Se implementan los métodos *`startElection()`*, *`endElection()`* y *`queryElection()`*. Este último método devuelve el estado actual de la elección, el cual puede ser: `NOT_STARTED`, `FINISHED` o `IN_PROGRESS`. Se utilizan dos `AtomicBoolean` llamados `electionStarted` y `electionFinished`, y métodos atómicos para cambiar su valor, con el fin de evitar problemas de concurrencia.
- *Monitoring Service*: Se implementa el método *r*`egisterFiscal(RemoteMonitoringClient,Integer,PoliticalParty)`. Cabe aclarar que el argumento `Integer` es el número de mesa. Para esta implementación se utiliza el mapa `fiscalsMap` que contiene como clave el número de mesa, y como valor un mapa que tiene como clave el partido político y como valor un `RemoteMonitoringClient`, es decir un fiscal. Para evitar condiciones de

carrera, se optó por que tanto `fiscalsMap` como el mapa interno sean `ConcurrentHashMap`. En primer lugar se verifica que la elección no haya comenzado ni finalizado, y en caso de no cumplirse la condición se lanza una excepción. A continuación se verifica que no haya otro fiscal que pertenezca a esa mesa y al mismo partido político que el fiscal que se quiere registrar, y en caso de no existir tal fiscal se agrega el nuevo fiscal a `fiscalsMap`. Si ya existe, lanza una excepción.

- ***Query Service***: se implementaron métodos para consulta de resultados a nivel nacional, provincial y de mesa. Además, se implementaron métodos auxiliares en clases helper para evitar inundar el Servant de contenido no estrictamente relacionado.
    - En el primer caso, se implementó el sistema de votación **Alternative Vote (AV)**, mapeando en primer lugar los votos a sus respectivas primeras opciones y creando un mapa *votes*, procurando evitar errores de sincronización mediante el uso de la palabra reservada `synchronized`. A continuación se trabaja sobre dicha estructura creada localmente por lo que no es necesario validar la sincronización. Se crea un set de resultados ordenado según el porcentaje obtenido y el nombre de cada partido, y luego se cicla eliminando a los peores candidatos hasta que un candidato supere el 50% de los votos o quede solo uno en carrera, actualizando los resultados en cada paso.
    - En el caso de la query por provincia, se tuvo que modificar el modelo de los datos debido a problemas que surgían de la repartición de votos característico del método **Single Transferable Vote (STV)**, al tener que trabajar con porcentajes en lugar de votos físicos. Se optó por implementar grupos de votos como objetos `PercentageChunk`, con un porcentaje correspondiente del total de votos y una colección de `VoteProportions`. Estos últimos objetos permiten observar la proporción de votos para ese candidato, en función de las opciones siguientes elegidas. Por ejemplo, si la Tortuga obtuviera seis (6) votos (primera opción), y dos de ellos tuvieran como segunda opción al Búho, sobre un total de 10 votos en la elección, el objeto `PercentageChunk` contabilizaría el 60% de los votos, y contendría dos objetos `VoteProportion`: uno representando votos que sólamente tuvieron como 1ra opción a Tortuga (el 66.67% de los votos totales que obtuvo) y otro representando aquellos que además eligieron al Búho como segunda opción (el 33.33% de los votos). De esta manera se diferencian todos los distintos tipos de votos que se registran a un candidato. Cuando se redistribuyen porcentajes (debido a que se supera el mínimo requerido), cada objeto `VoteProportion` contabilizará votos para la siguiente opción elegida, manteniendo una referencia de qué opción se utiliza. Los porcentajes correspondientes a la misma siguiente opción irán al mismo destino. Si el límite de votos es de 50%, un 10% de la Tortuga deberá ser reprocesado. De ese porcentaje, un 33.33% se descarta (al no tener siguiente opción) y un 66.67% se pasa al Búho, indicando que dichos votos no tienen siguiente opción. Entonces, el Búho obtendrá un 6.67% de nuevos votos, sobre el total, que se guardarán como un nuevo objeto

`PercentageChunk`. Dicho objeto contendrá un solo `VoteProportion` que abarcará el 100% de esos votos (los votos que tuvieron como primera opción a Tortuga y como segunda al Búho). Si además se hubieran transferido votos con una tercera opción Serpiente (y segunda opción Búho), habrían dos objetos `VoteProportion` para diferenciarlos. De esta manera el porcentaje final de un candidato se compone de distintos `PercentageChunk` que provienen de distintos candidatos, cada uno con un número de opción asociado a él. También se reprocesan porcentajes para los candidatos menos votados que deben ser eliminados, y se cicla hasta alcanzar el número de representantes requerido.

- Por último, se procedió de manera similar en la query por tabla utilizando una estructura local obtenida mediante un uso sincronizado del mapa de votos global. Al igual que la query nacional, se contabilizaron los resultados utilizando los objetos *Vote* y según su primera opción, siguiendo lo establecido por el método de votación **First-Past-The-Post (FPTP)**.

➢ *Voting Service*: Se implementa el método *`vote(List<Vote>)`*. Hace uso del mapa `stateVotes` que tiene como clave el estado y como valor un mapa que contiene como clave el número de mesa y como valor una lista de votos. En primer lugar se verifica que la elección haya arrancado y no finalizado, y para lo que sigue del método se hace un `synchronized` sobre `stateVotes`. Se itera sobre los votos que se pasan por parámetro, y se van agregando uno a uno al mapa. Luego, en esa misma iteración, se itera sobre `fiscalsMap` y por cada fiscal que pertenezca al mismo partido político y mesa que el voto, se lo notifica mediante el método remoto `notifyVote(Vote)`.