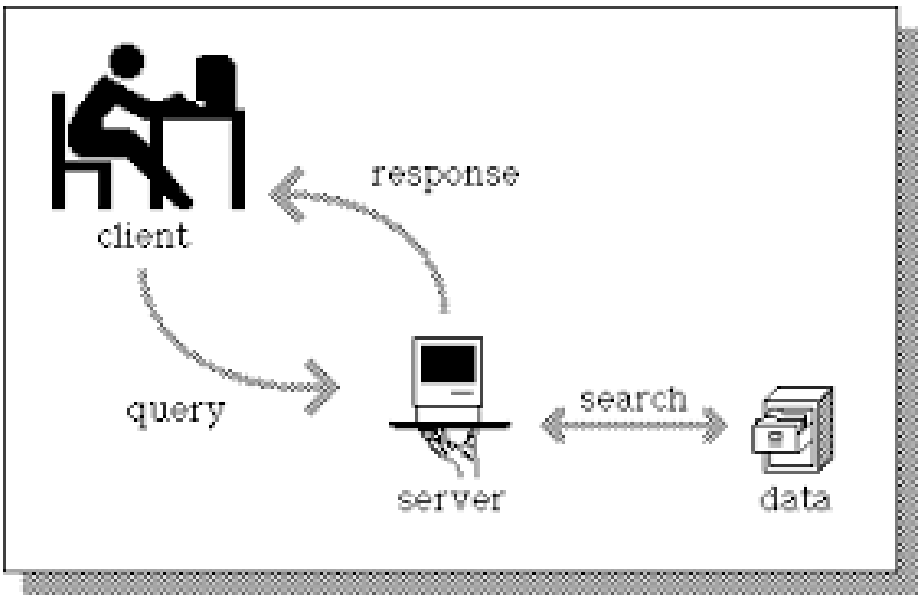
Práctica IV: Uso de Sockets



La finalidad de este ejercicio por parejas es crear una aplicación de escritorio que, siguiendo el modelo de arquitectura cliente- servidor, permita la comunicación mediante sockets a diversos clientes, los cuales accederán a un servidor para realizar operaciones contra una base de datos.

Hay que desarrollar en Java, todas aquellas clases que permitan el acceso simultáneo de varios clientes al servidor.

Cada pareja ha de elegir una base de datos y desarrollar un CRUD para dos de las tablas de la base de datos, permitiendo, también, añadir/eliminar relaciones entre registros de la tabla. Se puede usar una BBDD ya sea creada desde cero o que hayáis obtenido de internet. Como se ha de garantizar que la información es estable e integra, el acceso a la misma ha de ser en ***exclusión mutua.***

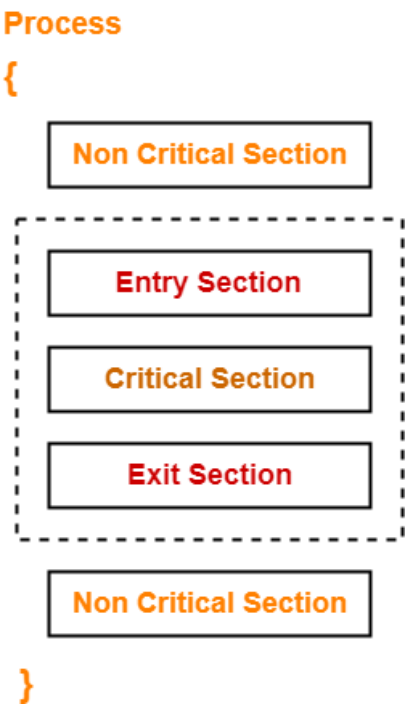
En más detalle, las tareas a realizar es:

- La implementación del cliente y servidor. El servidor ha de poder gestionar varias peticiones a la vez, con un máximo de 3 cliente simultáneos**. (1p)**
- El cliente podrá, consultar los registros de todas las tablas, y realizar un CRUD de una de ellas. La que seleccionéis para el CRUD tiene que estar relacionada con otra. **(1p)**
- Permite que el cliente pueda realizar una búsqueda filtrando por un campo (alguno que consideréis útil). El tipo de filtrado dependerá del tipo del campo asociado, y de la operación que queráis implementar. Por ejemplo, si tengo una base de datos de campeones del LOL, puedo filtrar por nombre, o puedo filtrar por aquellos campeones que tengan más HP que un valor en concreto… esto como queráis. **(1p)**
- Permite que el cliente pueda realizar 4 consultas útiles (usando GROUP BY, subconsultas…) **(1p)**

- Diseñar una interfaz para el cliente, el cual podrá realizar todas las funcionalidades especificadas arriba. El diseño corre a vuestro cargo **(2p)**
- Para gestionar esa concurrencia, hay que usar la clase **ExecutorService**, el cual gestiona una cola FIFO de clientes conectados. Quiere decir que, si hay 3 clientes conectados, el 4 y el 5 se quedarán esperando hasta que el servidor quede libre y los pueda atender. De esta forma los gestiona esta clase y no lo tenéis que hacer vosotros. **(2p)**
- Mejorar la sincronización entre hilos permitiendo que varios hilos puedan leer a la vez, pero la inserción/borrado sea en exclusión mutua.**(2p)**

Consideraciones:

- Recordad que para evitar bloqueos, el lock se tiene que coger para entrar en la SC y luego soltarlo.



- Para implementar la comunicación con la BBDD hay que usar Hibernate, siguiendo algún patrón de desarrollo que hayáis visto, ya sea el patrón DAO, el patrón de Services…
- Depende de la base de datos y tablas que uséis, os podréis enfrentar a tablas que tienen 5+ campos. La inserción en estos casos, que sea únicamente de 2 campos, el resto lo podéis dejar nulo…
- Para sincronizar los hilos hay que usar la clase ReentrantLock.
- Podéis diseñar la interfaz como queráis, pero se tendrá en cuenta la usabilidad: feedback, uso de componentes que faciliten la interacción, diseño y uso de colores, distribución de elementos…
- La clase ExecutorService funciona instanciandola estableciendo un número máximo de conexiones:

```java
// Permitimos 3 conexiones
ExecutorService threadPool = Executors.newFixedThreadPool( nThreads: 3);
```

Cuando se conecta, lo añadimos a la cola…

```java
Socket cliente = serverSocket.accept();
System.out.println("Se conexta el colega " + cliente.getInetAddress());

// Lo añadimos a la cola
Thread t = new Thread(new HiloServidor(cliente));
threadPool.submit(t);
```

Y si el servidor ya está trabajando con 3 sockets, el 4 se quedará esperando, simulando una cola FIFO. Cuando uno de los 3 clientes activos terminen, el servidor dará conexión al siguiente de la cola. Os dejo más info aquí y aquí (hasta la diapo 35)