**Caso 1 Infraestructura computacional**

**Nombre:** Stephania Otalora **Código**: 201714191

**Nombre:** Cristhian Forigua **Código:** 201713023

1. **Modelo conceptual del mundo:**

En la figura 1 se puede observar el modelo conceptual del mundo implementado para la solución del caso 1 de sincronización. El modelo conceptual consta de 5 clases principales: Cliente, Mensaje, Servidor, Buffer y Main.

Captura de pantalla de un celular con letras

Descripción generada automáticamente

Figura 1. Modelo conceptual del mundo para el caso 1 de sincronización.

* Mensaje: Esta clase es la encargada de modelar las consultas que puede realizar el usuario. Tiene como atributos tanto el contenido de la consulta, el cual es un entero, como el id del cliente que realiza la consulta.
* Cliente: Esta clase representa a un cliente que quiere realizar un número de consultas definido por *numMensajes*. Así mismo, tiene un id único que lo diferencia de los otros threads cliente.
* Servidor: Esta clase representa un thread del servidor, el cual está encargado de responder las consultas que realicen los clientes. Tiene como único atributo id único para cada thread servidor.
* Buffer: Esta clase es la encargada de recibir los mensajes de los clientes y mantenerlos en espera hasta que un thread servidor decida retirar un mensaje para resolverlo. Sobre está clase está la mayor parte del trabajo de sincronización realizado. Como atributos cuenta con la capacidad máxima de consultas que puede recibir, el número de clientes totales que tienen consultas, el número de clientes actuales que tienen mensajes en el buffer y por último un atributo booleano que indica si hay o no clientes con mensajes en el buffer.
* Main: Esta clase es la encargada de inicializar la aplicación. Crea los threads de los clientes y los servidores. Así mismo se encarga de inicializar el buffer compartido entre las clases mencionadas anteriormente. Todo es inicializado con respecto al archivo de propiedades “configuración.properties”.

1. **Funcionamiento del programa (Manejo de la aplicación):**

Como se mencionó anteriormente, la gran parte del manejo de la sincronización fue realizada en la clase Buffer. A continuación, se expone la implementación de la sincronización de la clase Buffer tanto para la clase cliente como para la clase servidor.

* **Buffer – Servidor:**

Al empezar el thread servidor se ejecuta el método run, mostrado a continuación:

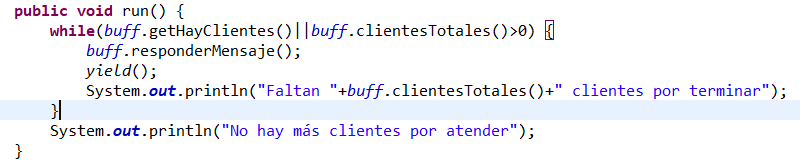


Figura 2. Método run de la clase Servidor.

Como se muestra anteriormente, el servidor intentará responder mensajes mientras haya clientes que estén esperando el buffer por la respuesta a sus consultas. Los métodos *getHayClientes()* y *clientesTotales()* son métodos synchronized sobre el buffer.

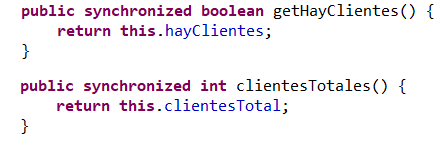


Figura 3. Métodos *getHayClientes()*  y *clentesTotales()*

Lo anterior se realiza para poder proteger el valor de las variables *hayClientes* y *clientesTotal* ya que hay acceso recurrente hacía el objeto buffer por parte de múltiples threads de clientes y servidores.

Dentro del método run, los thread servidor llaman al método *responderMensaje(),* el cual se muestra en la figura 3. En primera instancia se verifica el tamaño del buffer, cuando el buffer está vacío los servidores salen del método sin realizar ninguna otra operación. En el caso contrario, cuando el buffer no está vacío, se saca un mensaje del arreglo, se llama al método responder del mensaje removido. Todo lo anterior se encuentra dentro de una sección crítica establecida sobre el buffer, pues se quiere proteger el acceso a la variable *buff* del buffer. Una vez removido y respondido la consulta, se debe notificar al cliente que estaba esperando por dicho mensaje, por lo anterior se hace el *notify()* sobre el objeto mensaje, el cual pertenece al cliente que está esperando. Con el objetivo de proteger el acceso a dicho mensaje está en una sección crítica sobre el mismo objeto mensaje.

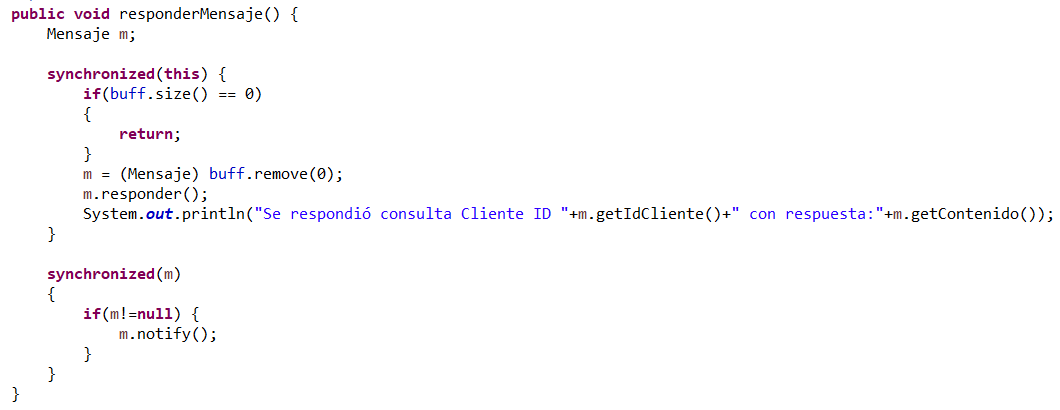


Figura 4. Método *responderMensaje()* de la clase buffer.

Por último, una vez el thread sale del método *responderMensaje(),* el servidor cede el procesador para permitir que otros threads sigan su ejecución. Para lo anterior, después del llamado al método se implementa un *yield().*

* **Buffer – Cliente:**

Al correr el thread se implementa el método *run()*. Con la siguiente configuración:

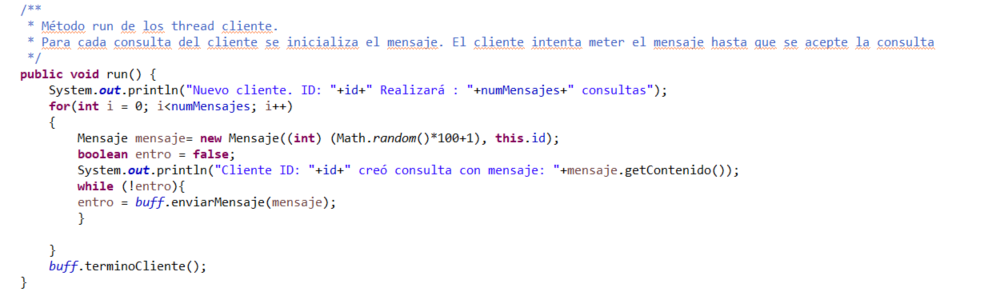


Figura 5. Método *run()* de la clase Cliente.

En ese caso el cliente crea la cantidad de mensajes según el atributo numMensajes e intenta depositarlo en el buffer. Sin embargo, si el buffer esta ocupado este lo sigue intentando hasta que lo pueda mandar. Y el contenido del mensaje es un numero aleatorio entre 1 y 100. Y finalmente cuando el cliente termina de enviar todos sus mensajes notifica al buffer y termina su ejecución.

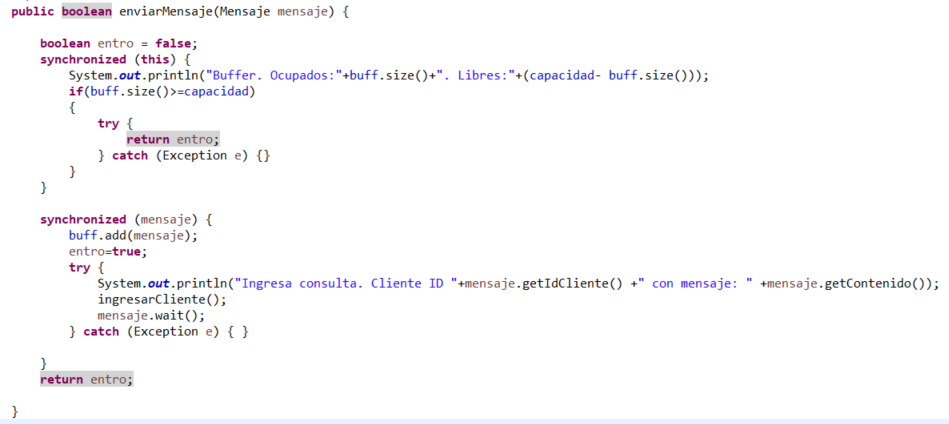


Figura 6. Método *enviarMensaje(Mensaje mensaje)* de la clase buffer.

En este punto se realiza la sincronización de los Threads Clientes. En donde primero se verifica si el buffer tiene capacidad, sino la tiene se retorna esto, y esta parte esta sincronizada sobre el buffer ya que el atributo buff es compartido por los Threads y puede generar problemas de concurrencia. Luego si el buffer si tiene capacidad se sincroniza sobre el mensaje (sobre el thread de cliente), se agrega al arreglo de mensajes a responder, se aumenta la cantidad de clientes actuales con el método *ingresarCliente() y* se duerme al thread mientras que espera la respuesta del mensaje.

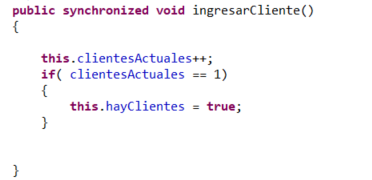


Figura 7. Método *ingresarCliente()* de la clase buffer.

Por otro lado, el método ingresar cliente esta sincronizado y este aumenta en 1 a los clientes actuales. Además, realiza la revisión de que si esta variable toma el valor de 1 se cambia el atributo de *hayClientes* a true para que los servidores comiencen a responder los mensajes.