****

**U.B.A. FACULTAD DE INGENIERÍA**

**Departamento de Informática**

**75.59 – Técnicas de Programación Concurrentes I**

**Segundo Proyecto: Cola de mensajes**

**Curso: 2013 – 1er Cuatrimestre**

|  |  |
| --- | --- |
| **APELLIDO, Nombres** | **N° PADRÓN** |
| Rodriguez, Sebastian | 90202 |
| Schenkelman, Damián | 90728 |
| Servetto, Matías | 91363 |
| **Fecha de Aprobación :** |  |
| **Calificación :** |  |
| **Firma de Aprobación :** |  |

**Observaciones:**

Contenido

[Análisis del problema 2](#_Toc359880313)

[Hipótesis 2](#_Toc359880314)

[Casos de uso 2](#_Toc359880315)

[Resolución del problema 4](#_Toc359880316)

[Programas 4](#_Toc359880317)

[Concurrencia 5](#_Toc359880318)

[Ventajas 7](#_Toc359880319)

[Desventajas 7](#_Toc359880320)

[ClientUI 7](#_Toc359880321)

[Protocolo de mensajes 7](#_Toc359880322)

[Diagrama de clases 8](#_Toc359880323)

# Análisis del problema

Se nos presento la situación de resolver un problema que se modela como el caso de un cliente-servidor.

En él se pide la construcción de dos programas:

* Uno encargado de la gestión de una base de datos.
* El otro encargado de realizar peticiones a la base de datos con las operaciones que se desean realizar.

La base de datos consta de registros con la forma:

|  |  |
| --- | --- |
| **Nombre de campo** | **Tipo del campo** |
| Name | Char[61] |
| Address | Char[120] |
| Phone | Char[13] |

Restricciones

* La comunicación debe darse a través de las colas de mensajes.

# Hipótesis

* Las peticiones que se piden son:
  + Alta de registros.
  + Baja de registros.
  + Modificación de registros.
* El identificador univoco de los registros es el campo “name”.
* Al modificar un registro especificando el campo “name” (el id del registro al que hay que modificar), si no se encuentra un registro con dicho identificador se da de alta un registro nuevo.
* Si al gestor se le envía la orden de finalizar el servicio, el servicio se finaliza y no se procesa la consulta de los clientes.
* No hay prioridades entre clientes.

# Casos de uso

Programa **servidor**

* Ejecutar servidor.
* Finalizar servidor.
* **Logguear** operaciones realizadas.

Para ejecutar el servidor se debe ejecutar por consola la línea:

|  |
| --- |
| * ./server |

Para finalizar el servidor se debe ingresar por la consola donde se ejecuto el programa una tecla y enviarla con **“enter”**.

El **loggeo** de las operaciones es automático, se genera un archivo llamado **“ServerSession<date>.txt”** en el directorio donde se encuentra el programa.

Programa cliente

* Realizar consulta de registros.
* Realizar alta de registros.
* Realizar baja de registros.
* Realizar modificación de registros.
* Realizar finalización del servicio.

Todos los casos de uso se ejecutan con la línea de comando:

|  |
| --- |
| * ./client <id> <name> <address> <phone> [<name id>] |

El campo **<id>** representa el identificador de la operación a realizar. Las operaciones son:

Para la **consulta**:

El identificador es “**3**”

En los campos **<name>**, **<address>** y **<phone>** se ingresa un patrón que debe estar contenido en el campo del registro correspondiente. Distingue mayúsculas de minúsculas.

Si no se quiere filtrar por alguno de los campos, ingresar “**-a**”.

El quinto campo es ignorado.

Para el **alta**:

El identificador es “**1**”

En los campos **<name>**, **<address>** y **<phone>** se ingresa el valor que tendrán los campos.

El quinto campo es ignorado.

Para la **baja**:

El identificador es “**4**”.

En el campo **<name>** ingresar el nombre completo del registro a eliminar.

En los campos **<address>** y **<phone>** ingresar cualquier dato.

El quito campo es ignorado.

Para la **modificación**:

El identificador es “**2**”.

En los campos **<name>**, **<address>** y **<phone>** se ingresa el valor que tendrán los campos.

En el campo **<name id>** ingresar el nombre completo del registro a modificar.

El campo **<name>** no debe repetirse en la base de datos.

Para la **finalización del servicio**:

El identificador es “**5**”.

En el campo **<name>**, **<address>** y **<phone>** ingresar cualquier dato.

El quito campo es ignorado.

Aclaraciones

El **parseo** de argumentos del cliente es simple, se ignoran los argumentos mayores al 5to.

Se piden siempre al menos 4 argumentos aunque no sean utilizados.

Esto fue pensado de esta manera ya que se debe utilizar por un usuario final a través de la interfaz grafica.

# Resolución del problema

## Programas

Al problema se lo dividió en los siguientes programas:

* **Server**: programa implementado en c++, que consta de dos procesos. Uno se encarga de gestionar la base de datos, recibir peticiones, procesar ABM en la base de datos y enviar resultados (de forma iterativa). El otro se encarga de lanzar el proceso servidor, esperar por una acción del usuario para finalizar el servidor y finalizar el programa entero.
* **Client**: programa implementado en c++, que consta de un proceso. Se encarga de generar una petición, enviarla y esperar la respuesta del servidor.
* **ClientUI**: programa implementado en java, que sirve como interfaz grafica entre el usuario y el programa **client**. Su función es la de generar una GUI más amigable que una consola, presentar las operaciones posibles al usuario y realizar la consulta y posterior muestra de los resultados.

## Concurrencia

El modelo de comunicación con cola de mensajes:

**//Actualizar diagrama (mostrando solo un proceso del lado del cliente y los nombres correspondientes a las MQ) y ponerlo acá.**

Se decidió implementar la comunicación a través de dos colas de mensajes.

* **mqRequest**: es la cola en la que todos los procesos insertan la petición.
* **mqResponse**: es la cola en que todos los procesos esperan la respuesta de la operación enviada.

Los **structs** que se envían tienen el siguiente formato:

|  |  |  |
| --- | --- | --- |
| **Struct** que viaja por **mqRequest** | | |
| **Nombre** | **Tipo** | **Descripción** |
| ClientId | long | Identificador del procesos que envía la petición |
| requestActionType | int | Identificador de la acción a realizar |
| Name | Char[61] | Campo **<name>** explicado en los casos de uso |
| Address | Char[120] | Campo **<address>** explicado en los casos de uso |
| Phone | Char[13] | Campo **<phone>** explicado en los casos de uso |
| Name id | Char[61] | Campo **<name id>** explicado en los casos de uso |

|  |  |  |
| --- | --- | --- |
| **Struct** que viaja por **mqResponse** | | |
| **Nombre** | **Tipo** | **Descripción** |
| ClientId | long | Identificador del procesos que recibirá la respuesta |
| requestActionType | int | Identificador de la respuesta recibida |
| numberOfRegisters | int | Indica la cantidad de **structs** que se enviaran luego del **struct** actual, si este es del tipo HEAD |
| Name | Char[61] | Campo del registro **name** de la base de datos, solo valido si el **struct** es del tipo BODY |
| Address | Char[120] | Campo del registro **address** de la base de datos, solo valido si el **struct** es del tipo BODY |
| Phone | Char[13] | Campo del registro **phone** de la base de datos, solo valido si el **struct** es del tipo BODY |

¿Cómo se crean las colas?

Desde el lado del **server**:

Al momento de iniciar este programa, se generan dos archivos temporales en la carpeta **/tmp**. Estos son utilizados para generar el identificador de la cola. Acto seguido, este procesos genera las colas, prepara la base de datos y entra en su ciclo de vida.

Desde el lado del **client**:

Este cuando inicia verifica que existan los archivos temporales en **/tmp**. Si estos existen se generan las colas. Si no existen, se toma como indicador de la no existencia de servicio y se finaliza el proceso indicando al usuario que no hay servicio disponible.

¿Cómo se produce la comunicación a través de las colas?

Desde el lado del **server**:

En cada iteración del ciclo de vida, se lee de **mqRequest** la siguiente petición (con el id 0) de manera bloqueante. Luego se procesa la consulta y se generan los **structs** de respuesta, para enviarlos por **mqResponse**.

Si bien se lee con el id 0, las **structs** de peticiones vienen firmadas con un identificador único para cada proceso (para asegurarnos que el identificador es único, se usa el **process id**). Esto se hace para, luego, marcar las **structs** de respuesta con este identificador.

Del lado del **client**:

El cliente genera la petición (ingresando como id del **struct** que modela la petición su **process id**), envía la petición por el **mqRequest** y se queda leyendo, de manera bloqueante, por **mqResponse** con el identificador propio.

¿Cómo se finaliza el servicio?

Ahora que se explico la comunicación y la identificación a través de los **process id**, se explicara cómo se finaliza el servicio:

El **server**, en su ciclo de vida antes de leer la siguiente petición, lee de la cola **mqRequest** con el identificador 1 de manera no bloqueante. Si no se recibió ninguna petición se sigue como se explico previamente.

En caso de recibir la petición con el identificador 1, esto significa que se creó un programa “**cliente-administrador**” que toma la decisión de terminar el proceso. Se lee de manera no bloqueante ya que se quiere dar prioridad a esta petición, pero si no se encuentra ninguna en la cola se debe seguir leyendo las peticiones de manera normal.

Este programa llamado “**cliente-administrador**” envía una petición con el identificador “1”, el cual nos aseguramos de que no se repite con un cliente común ya que es el **process id** del proceso “**init**”.

Recibida la petición de finalización, se lee de la cola todas las peticiones y se informa a los clientes del fin de la conexión para destrabarlos de la lectura de la cola. Luego se finaliza el servicio liberando las colas y borrando los archivos temporales.

El programa **server** hace uso de un programa **client**, el cual está renombrado como “**admin**” en la carpeta donde se encuentra **server**.

## Ventajas

* Simplicidad para saber si el servicio está activo, el cliente solo debe verificar que existan los archivos en **/tmp**.
* Simplicidad de conexión, el cliente debe generar las colas predeterminadas, no existe un protocolo para determinar cómo se genera una cola exclusiva para él.
* Simplicidad para la desconexión, el código del servidor no tiene muchas complicaciones para la finalización del servicio al tratar a esta acción como una petición más.

## Desventajas

* Como todas las respuestas se dan a través de una cola, es posible que si los clientes no leen de ella su respuesta la cola se llene y se comiencen a perder **structs**. Sin embargo se decidió seguir con este diseño dado que se especifica que la cola soporta hasta… **// explicar eso de que soporta muchos bytes y muchos structs y los registros son re chiquitos y pavos y el scope no da para algo complejo.**
* Si se envía la petición de finalización y el servidor se encuentra procesando una consulta muy grande, no se efectuara el cierre del mismo hasta que no se termine la consulta en curso. Dado que el sistema es pequeño, la base de datos no es compleja, se considera que para este **scope** de resolución del problema este factor es mínimo y podemos despreciarlo.

## ClientUI

Este programa fue desarrollado en java dado que es solo una interfaz grafica que agrega valor al trabajo y conocemos las librerías para llevarla a cabo y nos llevo poco esfuerzo. No se suplanto nada critico de concurrencia con código java por lo tanto no consideramos que se haya violado la restricción de que la solución deba ser resuelta en **c++**.

De todos modos es posible ignorar esta interfaz y utilizar el programa **client** desde consola.

|  |  |
| --- | --- |
| **IMPORTANTE!** | Si no se lanza el clientUI.jar darle permisos con el comando:   * chmod +x clientUI.jar |

## Protocolo de mensajes

**// describir el protocolo de head + body etc**

# Diagrama de clases

Para el desarrollo de los programas, se utilizo una mezcla de clases con programación estructurada.

El código de la programación estructurada corresponde al cliente y al servidor (en un alto panorama sin especificar chequeo de errores):

Cliente

|  |
| --- |
| 1. Request = PackageMessageRequest(args); 2. CheckConnectionWithServer(); 3. CreateMessageQueues(); 4. SendMessageRequest(Request); 5. Response = readMessageResponse(); 6. PrintResponse(Response); 7. ReleaseMessageQueues(); |

Servidor

|  |
| --- |
| 1. Id = fork(); 2. If( CHILD\_ID = Id )    1. PrepareDataBase();    2. CreateMessageQueues();    3. While( isWorking )       1. Request = CheckForAdministratorRequest();       2. if( NULL\_REQUEST == Request.type )          1. Request = ReadNextRequest();       3. Responses = ProcessRequest(Request);       4. SendResponse(Responses);    4. SaveDataBase();    5. ReleaseMessageQueues(); 3. Else    1. Fgetc(stdin);    2. CreateAdminClient();    3. Wait( NULL ); |

Clases utilitarias

**// agregar diagramas de clase de logger, messagequeue, converter y databasemanager**