Fiuba
Facultad de Ingeniería
Universidad de Buenos Aires

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

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.
  - Lectura 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.
- Realizar un **Log** de las 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 (READ)**:

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 (CREATE)**:

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 (DELETE)**:

El identificador es "**4**".

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

El resto de los argumentos es ignorado

Para la **modificación (UPDATE)**:

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 (GRACEFUL_QUIT)**:

El identificador es "**5**".

El resto de los argumentos es ignorado

Resolución del problema

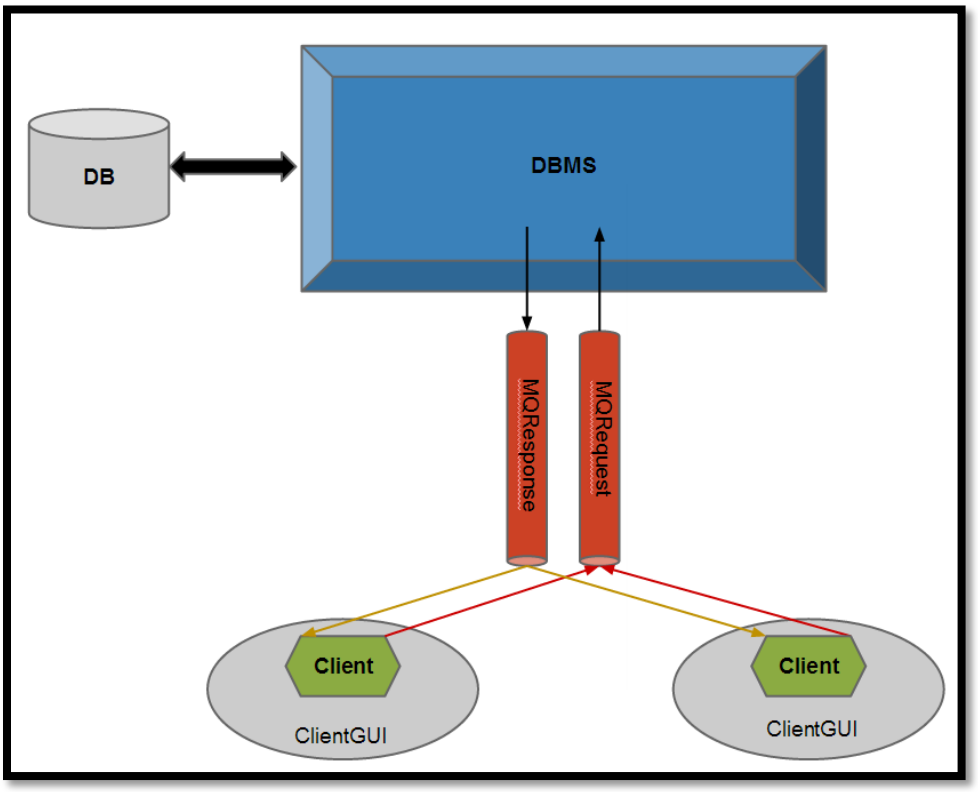
Programas

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.
- **ClientGUI**: 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:



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** (Al no haber prioridad entre clientes, se lee siempre el próximo mensaje en la cola.) 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**. Se lee de manera no bloqueante ya que se quiere dar prioridad a esta petición, que es la del cierre del servidor. Si no se recibió ninguna petición se sigue como se explico previamente de manera normal.

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. Este 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

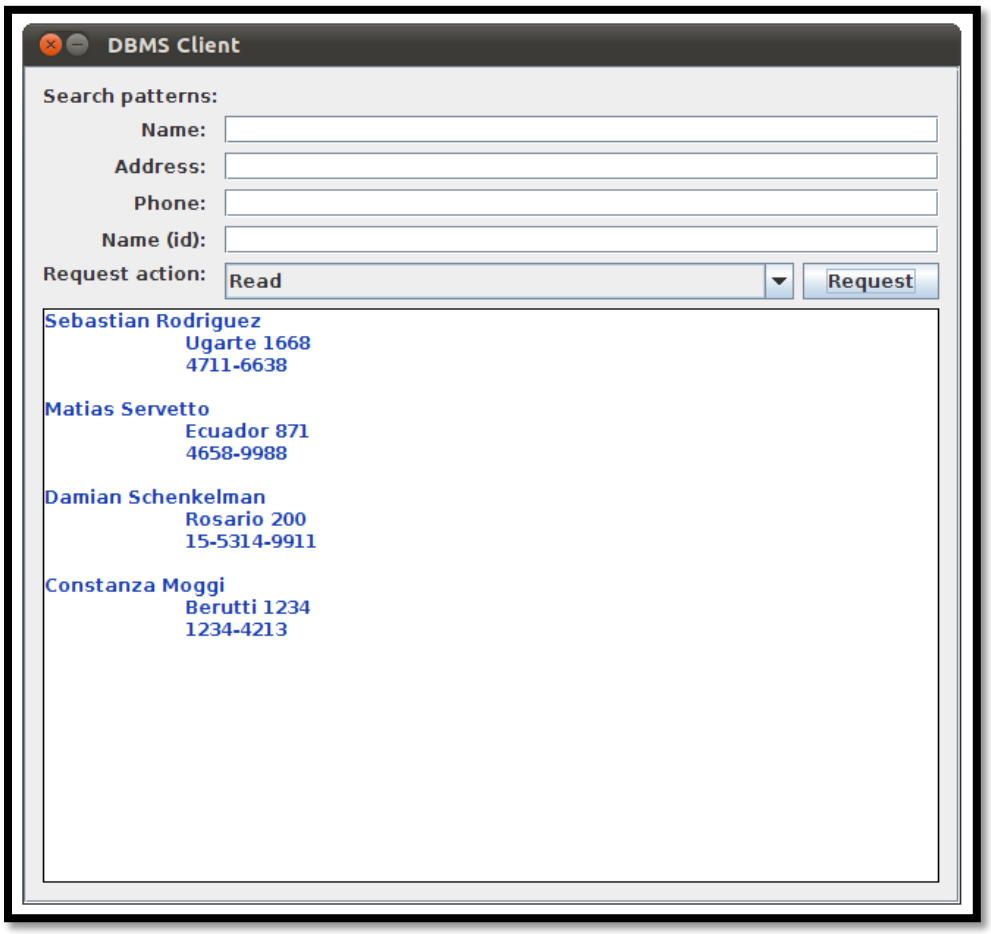
- Como todas las respuestas se dan a través de una cola, es posible que si los clientes no leen de ella su respuesta la cola se llene y se comiencen a perder **structs**. Sin embargo se decidió seguir con este diseño dado que se especifica que la cola soporta una cantidad de mensajes bastante superior a la esperada en el alcance de este proyecto. (En "/proc/sys/fs/mqueue/msg_max" se puede modificar la cantidad de mensajes máximos que pueden existir en una cola de mensajes. El límite máximo [HARD_MAX] es de (131072 / sizeof(void *)), que según el *man* de *mq_overview* es de **32768** en un Linux/86)
- 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.

ClientGUI

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++**.

La interfaz permite un acercamiento y usabilidad mayor al usuario final con el programa. Posee ciertas facilidades, como por ejemplo el hecho de que se pueden dejar los campos vacíos y luego estos serán interpretados como si se hubiese ingresado –a (Osea *, "todo"). Esto hace que el uso de filtros para la lectura sea más ameno, así como también el uso de los otros casos de uso que precisaban que tengan los primeros cuatro argumentos necesariamente se ve simplificado con esta sustitución automática que genera la GUI.



El resto de la funcionalidad de la GUI consta en lanzar el programa **Client** con los argumentos que el usuario ingreso en la interfaz, y luego leer la salida del programa **Client** e imprimirlo por pantalla.

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

Protocolo de comunicación

Para llevar a cabo la comunicación entre procesos, se implemento un protocolo de comunicación. Dicho protocolo permite informar tanto una petición como una respuesta con su debido detalle. Dentro de cada mensaje enviado, por ambas colas, existe un campo de "*tipo de acción*"; este campo contiene un identificador de una situación que o bien se quiere llevar a cabo o bien se ha realizado y posee dicha respuesta. Los identificadores de acción son los siguientes definidos:

Identificador	Acción
1	CREATE
2	UPDATE
3	READ
4	DELETE
5	GRACEFUL_QUIT
6	NULL_ACTION_TYPE
7	HEAD
8	BODY
9	OPERATION_FAILED
10	ENDOFCONNECTION
11	OPERATION_CREATE_SUCCESS
12	OPERATION_CREATE_FAIL_PERSON_EXISTS
13	OPERATION_UPDATE_SUCCESS
14	OPERATION_UPDATE_FAIL_PERSON_EXISTS
15	OPERATION_UPDATE_FAIL_PERSON_NOT_EXISTS
16	OPERATION_DELETE_SUCCESS
17	OPERATION_DELETE_FAIL_PERSON_NOT_EXISTS
18	OPERATION_UNKNOWN

Cada mensaje enviado posee un identificador de acción. De acuerdo al identificador, el proceso responde de forma diferente.

En el caso del **Server**, los posibles identificadores que le llegaran serán del 1-5; y actuará de manera distinta de acuerdo al caso que sea. Estos identificadores son los propios del ABM, consulta de registros y finalización del servidor.

En el caso del **Client**, el resto de los identificadores son posibles respuestas a su previa petición. De acuerdo a cada identificador, posee una estructura "***map***" que permite imprimir el detalle de información correspondiente.

Existe un caso en particular, que es la consulta de registros; es particular por el hecho de que no llega un solo mensaje del servidor, sino que llegaran tantos mensajes como registros hayan matcheado con los parámetros de la consulta. En este caso, se decidió agregar un campo mas en el mensaje de respuesta que indique la cantidad de registros encontrados. Cuando se hace la consulta, se guarda el número de matcheados que existe, y se lo pone en un primer mensaje con identificador HEAD. El **Client** al leer este tipo de identificador sabe que es producto de un READ previo, y se dispone a recibir la cantidad de registros indicados por HEAD. Los registros siguientes poseen el identificador BODY. En el caso donde no haya ningún registro que matchee los parámetros, al leer el 0 el **Client** guarda el registro HEAD para interpretarlo luego. En el resto de los casos, el campo cantidad de registros matcheados se pone en 0.

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 )
      a. PrepareDataBase();
      b. CreateMessageQueues();
      c. While( isWorking )
            i.  Request = CheckForAdministratorRequest();
           ii.  if( NULL_REQUEST == Request.type )
                   1. Request = ReadNextRequest();
          iii.  Responses = ProcessRequest(Request);
           iv.  SendResponse(Responses);
      d. SaveDataBase();
      e. ReleaseMessageQueues();
3. Else
      a. Fgetc(stdin);
      b. CreateAdminClient();
      c. Wait( NULL );
```

Clases utilitarias