Trabajo Práctico Nro. 1:

Inter Process Communication

★**Grupo:** 1

★**Integrantes:**

- Martin Leon Nagelberg - 56698
- Daniel Lobo - 51171
- Jeremias Aagaard - 53219

★**Fecha:** Lunes 16 de Mayo de 2016

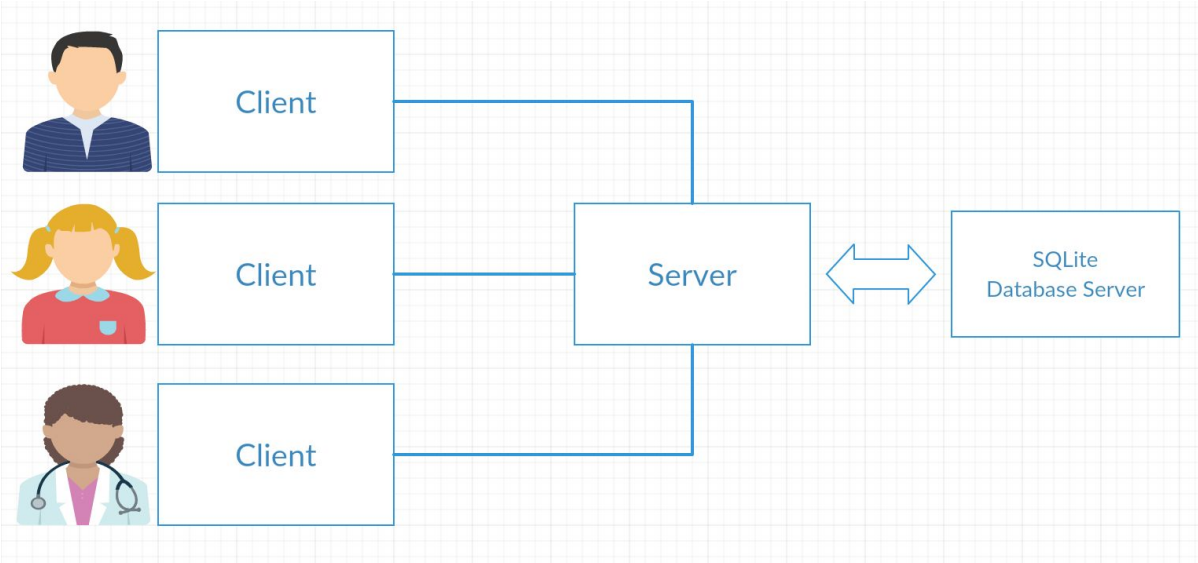
Idea del proyecto

La idea proyecto es crear una sala de chat que tiene un servidor y varios clientes que se conectan al mismo para conversar entre sí en un mismo lugar.

La idea es que los procesos cliente sean concurrentes y sean coordinados por un servidor que también es concurrente.

Inicialmente está implementada la capa de comunicación para comunicar clientes con servidor en un archivo llamado ***comm.h*** que se puede encontrar tanto en la carpeta ***sockets*** como en la carpeta ***namedPipes***.

Diseño conceptual



Diseño de la capa de comunicaciones entre el cliente y el servidor

Se decidió abstraer las siguientes operaciones en el archivo ***comm.h***:

- int connect_to(void * address);
- int disconnect(int connection_descriptor);
- int send_data(int connection_descriptor, void * message);
- int receive_data(int connection_descriptor, void *ret_buffer);
- int listen_connections(void * address, main_handler handler);

Se decidió mantener esta simple y, según nuestro criterio, poca cantidad de métodos ya que el resto de las acciones hechas por los named pipes y sockets ocurren dentro de cada uno de estos métodos. De esta manera logramos abstraernos sólo con estos cinco métodos de todo lo ocurrido entre el cliente y servidor.

Es importante también destacar que el método **listen_connections** recibe un **main_handler** que está definido de la siguiente manera:

typedef void (* main_handler) (int listener_descriptor, int new_connection_descriptor);

Concurrencia:

Este trabajo práctico se caracteriza por tener varios procesos cliente concurrentes coordinados por un servidor, el cual funciona como una sala de chat y donde cada uno de los clientes es un usuario del chat y se conecta o desconecta del mismo.

La concurrencia del servidor es manejada por con pthreads en lugar de fork, se decidió hacer esto para implementar una de las características opcionales del proyecto. Al mismo tiempo, nos pareció más sencillo de implementar, nos parecieron fáciles de iniciar y de terminarlos.

Serialización de datos:

Para serializar los datos utilizamos *bytequeues*. Éstas, nos permiten optimizar la comunicación enviando menor cantidad de datos por paquete. La estructura básica para enviar / recibir es la siguiente: **(BYTE) PACKET_ID | DATA**.

Servidor de Datos:

Se tomó la decisión de implementar el servidor de datos de este trabajo práctico usando SQLite, más que nada por la simplicidad (maneja concurrencia de lectores/escritores) que este brinda para hacer las operaciones básicas que necesitábamos de:

- Registrar un usuario,
- Modificar los permisos de un usuario,
- Eliminar un usuario por su username,
- Buscar chatlogs entre dos fechas,
- Insertar un nuevo mensaje en el chatlog,
- Modificar el password del usuario,
- Banear un usuario,
- Log in de usuario,
- Crear la base de datos.

Daemon de logging:

Se tomó la decisión de implementar con Message Queues un daemon de logging independiente que, con el comando **“make run”**, comienza a correr al mismo tiempo que el servidor. El mismo tiene la funcionalidad de loguear mensajes de 3 tipos:

1. **INFO:** sos mensajes que proveen información de lo que está haciendo el servidor.
2. **WARNING:** son mensajes sobre errores que pueden ser reparados y que ocurrieron en tiempo de ejecución.
3. **ERROR:** son mensajes sobre errores irrecuperables.

Todos estos mensajes son guardados por el logging daemon en el archivo **./logger/logs.txt**

Manual de operación del software:

Se puede encontrar en el archivo README.md en la raíz del proyecto.