SISTEMAS OPERATIVOS

TRABAJO PRÁCTICO Nº 1
INTER-PROCESS COMMUNICATION

ALUMNOS

- 52056 – Juan Marcos Bellini
- 55821 – Diego María De Grimaudet De Rochebouët
- 55824 – Juan Li Puma
- 58241 – Ishbir Singh

TABLA DE CONTENIDOS

TABLA DE CONTENIDOS 2
INTRODUCCIÓN 3
ALCANCE DEL PROYECTO 4
CAPA DE COMUNICACIÓN 5

INTRODUCCIÓN

El presente informe trata acerca de la entrega preliminar del primer trabajo práctico de Sistemas Operativos. La misma consta de de la idea general del proyecto, el diseño conceptual, y la capa de comunicación entre el cliente y el servidor. El trabajo consiste en aprender a utilizar los distintos mecanismos de IPC (*inter-process communication*) presentes en un sistema POSIX.

Dentro de las características del proyecto se encuentran la existencia de más de un proceso cliente (todos ellos concurrentes), conectándose a un proceso servidor (que también es concurrente). La comunicación entre los clientes y el servidor es a través de una capa de comunicación con dos implementaciones posibles, *Named Pipes* (o *FIFOs*) y *Sockets TPC*. Además, el servidor se comunica con servicios de datos y de *logging* independientes.

A continuación se puede ver un breve esquema de la arquitectura general del *software*.

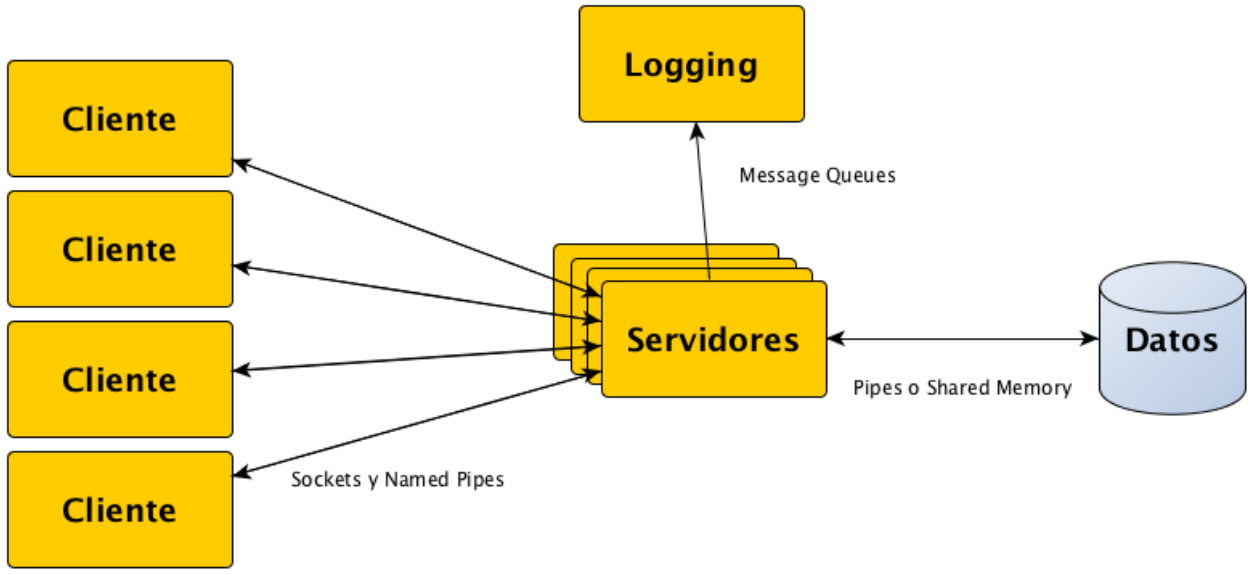

FIG. 1: ESQUEMA GENERAL DEL *SOFTWARE*

ALCANCE DEL PROYECTO

El *software* a desarrollar tiene la finalidad de crear una herramienta que pueda ayudar en la comercialización de bebidas alcohólicas en una modalidad de *delivery*. Los clientes de la empresa pueden utilizar la aplicación para consultar precios y/o *stock*, y realizar compras. A su vez, los dueños del comercio, pueden actualizar sus productos ofrecidos, sus precios, y sus depósitos, y consultar y actualizar los pedidos pendientes y pasados.

DISEÑO CONCEPTUAL

El *software* cuenta con distintas partes, cada una con un rol específico. Por un lado, se cuenta con la aplicación cliente, la cual es utilizada por los usuarios del sistema. A través de ella se pueden realizar acciones como pedir información de productos, o realizar compras. Cuando se instancia un cliente (o sea, cuando se inicia dicha aplicación), se realiza una conexión a un servidor “principal”, que tiene la función de esperar conexiones entrantes. Cuando llega alguna solicitud de conexión, éste instancia otro servidor – mediante la *syscall fork()* – el cual tiene la función de comunicarse con el cliente que previamente se conectó al servidor principal. De esta manera, se logra la concurrencia de clientes. Cada uno de ellos tiene su servidor “dedicado”.

Por otro lado, los datos – listado de bebidas y las compras realizadas – se almacenan en una base de datos SQLite. Este es otro proceso servidor independiente el cual tiene como función tanto almacenar datos en disco, como también ofrecer un servicio de consulta organizada. SQLite tiene como característica resolver la concurrencia por sí mismo. Las consultas (y modificaciones) a los datos son realizadas por cada servidor instancia (llamado servidor *forkeado*). Si el usuario desea, por ejemplo, consultar el listado de productos en venta, ingresa el comando correspondiente en la aplicación cliente, ésta realiza la petición a su servidor dedicado, y éste último consulta la base de datos. Luego SQLite responde (al servidor *forkeado*), y él le transmite la información al cliente (ver **Fig. 1**).

Finalmente, para que los distintos servidores puedan comunicar su estado al administrador, se cuenta con un servicio independiente de *logging*. Los distintos servidores (principal y *forkeados*) envían mensajes de distintos niveles a través de él, dando a conocer el estado en que se encuentran, logrando así que el administrador del sistema pueda monitorear el *software*.

CAPA DE COMUNICACIÓN

Existen distintas formas de comunicación en el *software*. Para empezar, la comunicación entre los servidores y los clientes son a través de *named pipes*, y a través de *sockets TCP*. La elección de cada método es en el momento de compilación del sistema.

Un *named pipe* (o *FIFO*), a diferencia de un *pipe* común, vive en el *file system* de la computadora. Al momento de instalar el sistema, se genera un archivo de configuración en el cual se especifica la ruta de acceso del *FIFO*. Cuando el servidor principal inicia, crea un *named pipe* y comienza a escuchar a través de él, esperando conexiones entrantes. Cuando un cliente inicia, crea dos *named pipes* (uno de entrada, y otro de salida), y escribe sus direcciones en el *FIFO* del server principal. Éste lee dichas direcciones y se *forkea*, para que, de esta manera, sea el servidor *forkeado* quien se comunique con el cliente.

Por otro lado, si la forma de comunicación es través de *sockets TCP*, cuando el servidor principal inicia, éste crea un *socket,* configurándolo de manera que pueda escuchar a través de todas las interfaces de red disponibles (*Ethernet*, *Wi-Fi*, *Loopback*, *Bridge*, etc.), y espera a que lleguen conexiones entrantes. Cuando un cliente inicia, éste crea su propio *socket* especificando la dirección del servidor (ya sea por línea de comandos o por archivos de configuración), y se conecta con él. En ese momento, el servidor acepta la conexión, se *forkea*, y crea un servidor dedicado (con su propia conexión con el cliente) para que se puedan comunicar independientemente.

Respecto a la comunicación con la base de datos, se decidió hacer uso de *pipes* debido a que la implementación de este tipo de comunicación es más simple que a través de *shared memory*.

Finalmente, para el sistema de *logging* se usarán colas de mensajes (*message queues*) entre los servidores y dicho servicio.

INTERFAZ DE COMUNICACIÓN CLIENTE-SERVIDOR

A continuación se expone la interfaz de comunicación entre los procesos cliente y servidor. Se listan los prototipos de función y su correspondiente documentación.

```
#ifndef comm_h
#define comm_h

#include <sys/types.h>

/*
 * Structure for storing connection information. The definition of the structure
 * is dependent on the communication method used (FIFOs or Sockets), so it's
 * defined in more than one file. When compiling the system, the user decides
 * which communication method to use, so only one of the definitions gets
 * compiled for any given system run.
 */
typedef struct connection_t* Connection;


/*
 * Standard messages to communicate different statuses or actions.
 */
#define MESSAGE_OK "KCOOL"
#define MESSAGE_ERROR "OHNOES"
#define MESSAGE_CLOSE "KTHXBAI"
```

```c
/*
 * Creates a connection between the current process and a different one.
 * Depending on what the current process is acting at, the other endpoint can be
 * the main server, a forked server or a client.
 *
 * @param const char *address A formatted address of the other endpoint of the
 * connection to establish.
 * @return Connection The connection, ready to transmit and receive data, or
 * NULL on error.
 */
Connection conn_open(const char* address);


/*
 * Sends MESSAGE_CLOSE to the other endpoint of this connection and frees up the
 * resources used by this process to maintain the specified connection.
 *
 * @return int 1 on success, 0 on error.
 * @see MESSAGE_CLOSE
 */
int conn_close(Connection connection);


/*
 * Sends the specified message to the other endpoint of the specified
connection.
 * To expect a response, call <i>conn_receive()</i> afterwards.
 *
 * @param const Connection c The connection through which to send data.
 * @param const void* data The data to send.
 * @param const size_t length The length (in bytes) of the data to send. This is
 * sent before sending the data, so the receiving end knows how many bytes to
 * read.
 * @return int 1 on success, 0 on error.
 */
int conn_send(const Connection connection, const void* data, const size_t
length);


/*
 * Reads data from the specified connection.
 *
 * @param const Connection c The connection from which to read data.
 * @param void** data Where to store the received data.
 * @param size_t* length Where to store the length of the received data.
 * This is read first, in order to allocate just enough memory for the received
 * data.
 * @return int 1 on success, 0 on error.
 */
int conn_receive(const Connection conn, void** data, size_t* length);


#endif /* comm_h */
```