INFORME TRABAJO PRÁCTICO Nº4

Nivel y Área:

72.11 – Sistemas Operativos

1º Cuatrimestre 2018

Comisión: S - **Carrera:** Informática

Fecha: 24 de junio de 2018

Alumnos Expositores:

AQUILI, Alejo Ezequiel (57432)

BASSANI, Santiago (57435)

RITORTO, Bianca (57082)

SANGUINETI ARENA, Francisco Javier (57565)

Informe: Trabajo Práctico Nº4 - Programación de Sockets

Objetivo

El objetivo de este trabajo práctico consiste en crear un programa para la gestión de la reserva de asientos de avión utilizando un sistema con arquitectura cliente-servidor. El servidor se implementará mediante la creación de procesos que atiendan las consultas de los clientes. Se utilizarán Free BSD sockets como mecanismo de IPC (Inter Proccess Comunication) para sincronizar y proveer comuncación a los procesos clientes con el proceso servidor.

Desarrollo e implementación

En lo que respecta a la creación y administración de la base de datos utilizada por el proceso servidor, se utilizó la librería SQLITE propuesta por la cátedra para contar con un DBMS que nos permitió realizar Querys en lenguaje C. Para no acoplar esta libreria con el resto de nuestro desarrollo, se optó por crear un tipo abstracto de dato que encapsula el comportamiento de esta librería permitiendo cambiar de servicio de DBMS sin tener que alterar el resto de nuestro sistema. Es decir basta con respetar el contrato del TAD con el nuevo DBMS (Data Base Managment System) sin necesidad de un refactor global del sistema.

Se implementó un servidor concurrente multi-proceso, el cual hace uso de los sockets bajo el esquema BLAB (Bind Listen Accept Begin-to-listen-again) Funciona de la siguiente forma: se abre un nuevo socket y se lo asocia con un puerto específico conocido tanto por el servidor como por los clientes. Luego, el socket comienza a escuchar en el puerto y al recibir alguna request del cliente un segundo socket entra en en juego para ser atendido por un nuevo proceso hijo del servidor mientras que el proceso padre (el servidor) seguirá escuchando en el puerto. Se utilizó un semáforo para el control del máximo numero de procesos hijos que atiendan a los clientes. En caso de que el número de procesos hijos del servidor que atienden concurrentemente a los clientes sea igual al maximo de los hijos permitidos, los nuevos clientes deberán esperar por el recurso. Esta sincronización es la que se logra gracias al uso del mecanismo de IPC de POSIX Semaphores.

Si dos clientes se conectan en simultaneo al servidor, sus request se encolan en el socket que escucha en el puerto.

El siguiente esquema expone el comportamiento entre servidor y clientes a nivel operaciones sobre los IPC de Sockets:

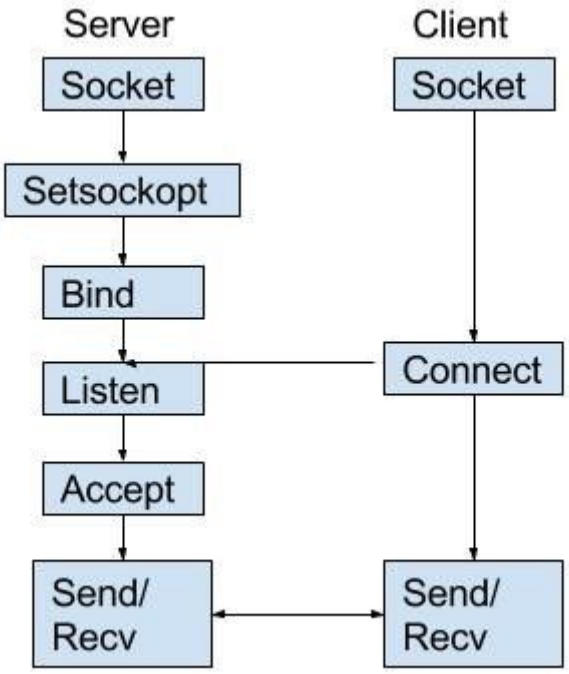

Figura 1:comportamiento entre servidor y clientes a nivel operaciones sobre Sockets

Para el envio de información por sockets se implementó un sistema de serializacion de tipos de datos con una tendencia o leve orientación a objetos. El sistema de serialización se basa en gran medida en el estándar JSON (JavaScript Object Notation).

En lo que respecta a los clientes, se utilizó un intérprete de comandos que le provee al usurio funciones para poder ver todos los vuelos disponibles, agregar o eliminar un vuelo, ver el DAV (Distribución de Asientos del Vuelo) de un vuelo dado, reservar un asiento en un vuelo y cancelar una reservación. El proceso cliente se conceta al socket y envía la *request* correspondiente a cada comando ejectuado y recibe y deserializa la informacíon que fue serializada y enviada por el servidor.

Concurrencia en la base de datos:

La exclusión mutua en la base de datos está garantizada por la librería SQLITE. Sólo se permite que haya un escritor en la base de datos al mismo tiempo, formándose una cola para las demas solicitudes de escritura. Sin embargo, se permiten múltiples lectores.

Esquema de Base de Datos:

flight		
flightNumber **PK**	origin	destination
TEXT	TEXT	TEXT

Figura 2: esquema de tabla "flight"

flightSeats			
flightNumber **FK**	colLetter	rowNumber	occupied
TEXT	CHAR(1)	INTEGER	INTEGER

Figura 3: esquema de tabla "flightSeats"

reservation			
flightNumber **FK**	userId	colLetter	rowNumber
TEXT	TEXT **NOT NULL**	CHAR(1) **NOT NULL**	INTEGER **NOT NULL**

Figura 2: esquema de tabla "reservation"

Nota: lo sombreado en naranja simboliza la clave de la tabla. PK = Primary Key. FK = Foreign Key.

Decisiones importantes en el diseño

Se decidió implementar un servidor multi-proceso en vez de multi-thread dado que se investigó que la principal ventaja de esta decisión se daba en terminos de seguridad, ya que los procesos no comparten memoria y los threads si lo hacen (comparten el heap). Se priorizó la seguridad, garantizando que si un cliente se ve afectado maliciosamente, éste no pueda acceder a memoria de otro cliente. Esta decisión se impulsó junto con el conocimiento de que el sistema correría en un sistema operativo POSIX compatible (MacOS o Linux) donde la diferencia en terminos de overhead a la hora de crear un proceso o un thread no es tanta, como sí lo es por ejemplo en un sistema operativo como Windows. Si el sistema operativo elegido para correr el servidor hubiera sido Windows, se hubiera optado por un servidor multi-thread.

Se decidió utilizar un sistema de serialización orientado en JSON dado que este formato es un formato simple de usar y que se utiliza en otros esquemas cliente-servidor como es en el caso de Navegadores Web (clientes) y servidores.

Fianlmente, la decisión primordial que rigue y riguió el desarrollo de todos los trabajos realizados en esta materia, es el uso de TDD (Test Driven Development) junto con un mecanismo de desarrollo orientado en XP (extremeProgramming). Siendo esta ultima decisión la piedra fundamental en el desarrollo de nuestro sistema.

Dificultades

Se presentaron dificultades a la hora de generar dinámicamente las querys sobre el *handler* de la base de datos construido ad-hoc para nuestro sistema, respetando nuestro esquema de base de datos diseñado y haciendo uso del tipo abstracto de datos creado como interfaz entre el DBMS y el servidor. La problématica estuvo en el calculo dinámico de la longitud de las cadenas de caracteres. Para solventar esta dificultad se

pre-armaron querys tipo “moldes” para las distintas operaciones, permitiendo una programacíon más modular y menos acoplada.

Se presentaron problemas con la utilización de la función setsockopt() con la cual se buscaba la posterior reutilización del puerto asociado al socket. El uso de esta función es completamente opcional en terminos de funcionalidad, pero se intentó utilizar en vistas de promover las buenas prácticas en la programación con el mecanismo de IPC de Sockets.

Por último, se presentaron dificultades a la hora de elegir qué tipo de socket utilizar. Unix Domain Socket VS Internet Sockets. Se optó por la utilización de los primeros, ya que se deseaba tener multiples clientes de manera local.

Instrucciones de compilación y ejecución

Para la compilación y linkedición existe un Makefile, basta con correr:

```
:~/Planeseat$ make all
```

Para correr el servidor en una terminal (luego de haber compilado y linkeditado):

```
:~/Planeseat$ ./Server/server.out
```

Para correr clientes en una terminal (luego de haber compilado y linkeditado):

```
:~/Planeseat$ ./Client/client.out
```

Testing y Debugging

Como se mencionó anteriormente, se intentó seguir la filosofía de extreme programming y llevar a cabo TDD. Se decidieron implementar test unitarios de los distintos componentes del sistema como del sistema de serialización, que se testeó de forma nativa y el módulo de base de datos para el cual se utilizó el Framework CUTest (https://github.com/aiobofh/cutest.git).

En cuanto a debbuging, se utilizaron las mismas herramientas que en las anteriores entregas. Herramientas de análisis dinámico como Valgrind y herramientas de análisis estático como Cppcheck.