Sistemas distribuidos G81: Segunda Práctica

József Iván Gafo - 100456709@alumnos.uc3m.es

Pablo Moreno González - 100451061@alumnos.uc3m.es

1. Introducción

El objetivo de la segunda parte de esta práctica, es de reutilizar los clientes de la parte 1 e implementar un nuevo proxy y servidor usando programación de sockets usando TCP en C.

Para esto, el sistema se va a componer de tres elementos fundamentales.

Cliente: va a ser el que envía las peticiones al servidor mediante una API (app-cliente).

Proxy: va a actuar como intermediario entre el cliente y el servidor.

Servidor: va a ser el encargado de procesar las peticiones de los clientes y de almacenar las duplas en una lista enlazada que previamente hemos implementado.

Entre la elección entre threads bajo demanda y un pool de threads, nos hemos decantado por la segunda opción; el Pool de Threads, ya que permite mejorar la eficiencia en el manejo de las peticiones concurrentes así evita la creación y la destrucción constante de hilos.

2. Diseño

2.1 Arquitectura del Sistema

1. Se arranca el servidor especificando en sus parámetros el puerto de este. (e.g ./servidor-sock 4500). Y también el servidor inicializa el pool de threads.
2. El cliente debe primero establecer las variables del entorno en el cual define la IP y el puerto del servidor. (e.g env IP TUPLA=localhost PORT_TUPLAS=4500.)
3. El flujo del sistema va a ser que primero el cliente realiza una petición mediante la Api (*claves.h)*.
4. El proxy va a convertir la petición de tal manera que el servidor lo pueda recibir correctamente sin errores.
5. El servidor recibe la petición, le asigna un thread y la función servicio se encargará de gestionar la petición del cliente.
6. Una vez que el servidor haya gestionado el cliente, le envía la respuesta y cierran su comunicación entre ellos.
7. El proxy recibe la respuesta y se lo envía al cliente.

2.2 Estructura de los Archivos

```
P2_sistemas_distribuidos
.
├── Makefile
├── memoria_p2_sistemas_distribuidos.pdf
├── README.md
├── apps
│   ├── app-cliente-1.c #Pruebas (insertar, modificar, eliminar, existir, destroy)
│   ├── app-cliente-2.c #Pruebas de errores, límites y validaciones
│   └── app-cliente-3.c #Pruebas de concurrencia en múltiples procesos
├── common
│   ├── claves.c #Implementación del almacenamiento en memoria (listas enlazadas)
│   └── comm.c #Funciones auxs para manejo de sockets
├── include
│   ├── claves.h #API del sistema clave-valor
│   └── comm.h #Declaraciones para el uso de sockets y comunicación
├── proxy
│   └── proxy-sock.c #Implementación(TCP) que traduce llamadas claves.h a operaciones de red
└── servidor
    ├── common
    │   ├── pool_of_threads.c #Pool de hilos que procesa clientes concurrentemente
    │   └── servicio.c #Lógica de gestión de operaciones recibidas
    ├── include
    │   ├── pool_of_threads.h #Interfaz del pool de hilos
    │   └── servicio.h  #Declaración de función servicio
    └── servidor-sockt.c
```

Comparándolo con la práctica 1, hemos decidido reestructurar el código para tener una mejor estructura y poder diferenciar los diferentes componentes.

En la carpeta de apps tendremos todos los app clientes, que no han sido modificados de la práctica 1.
En common tendremos archivos de C comunes como ejemplo comm.c que tiene funciones para poder conectarse cliente-servidor, y luego claves.c que solo lo usa el servidor.
En include tendremos las cabeceras de las funciones que se usarán en los clientes y servidor

En la carpeta proxy tendremos el proxy del cliente encargado de establecer comunicación con el servidor , y enviar la respuesta del servidor al cliente.

Por último en servidor tendremos todo el código relacionado al servidor. Una mejora que hicimos comparado a la práctica 1, es que hemos hecho el código modular y poniéndolo en las carpetas common e include para tener el código más ordenado.
En el archivo pool of threads tenemos todas las funciones relacionadas con la creación y gestión del pool de threads, en el cual cada cliente se conecta a un thread.
En el archivo de servicio.c está encargado de procesar la petición del cliente
Y el archivo servidor-sock.c está encargado de encender el servidor, encender el pool de threads y añadir conexiones por cada cliente que se conecte.

2.3 Sección de Pruebas

Para verificar el correcto funcionamiento de nuestro nuevo proxy y nuevo servidor usando programación de sockets, hemos reutilizado los mismos app clientes de la práctica 1. En el cual el primer app cliente está encargado de verificar el correcto funcionamiento de los distintos componentes, el 2 encargado de mirar los límites y el último encargado de verificar que nuestro sistema puede funcionar correctamente de manera distribuida.

3. Implementación

3.1 Gestión de comunicación tcp: comm.c

Este archivo fue inspirado en el caso ejemplo proporcionado por los profesores por github.

- ***ServerSocket***: Inicializa el socket del servidor con el puerto obtenido en el parámetro. y pone el servidor en modo escucha.
- ***ServerAccept***: Acepta una conexión entrante
- ***ClientSocket***: Inicializa el socket del cliente
- ***CloseSocket***: Cierra el socket
- ***SendMessage***: Envía un mensaje a través del socket
- ***recvMessage***: Recibe el mensaje a través del socket
- ***writeLine***: Escribe una línea en el socket
- ***readLine***: Lee una línea del socket
- ***connectServer***: Esta función no estaba en el github, y está encargada a extraer las variables de entorno para sacar el puerto y la ip para poder conectarse al servidor y a la vez crea un socket al cliente
- ***disconnectServer***: Función encargada a desconectarse del servidor

3.2 Intermediario en proxy-sock.c

El próximo va a actuar como un intermediario entre el cliente y el servidor.

1) Recibe una petición de la API
2) Se conecta con el servidor, usando las variables de entorno para conectarse
3) Cabe destacar que para enviar o recibir mensajes el cliente deberá transformar los valores network to host o de host to network. Para poderlo leer correctamente.
4) Envía primero el código de operación usando sendMessage (implementado en comm.c)
5) Luego en mensajes individuales le envía uno a uno los parámetros necesarios para ejecutar la función.
6) A continuación se bloquea en receive message, y cuando lo recibe cierra el socket con el servidor y envía el resultado al cliente.

3.3 Servidor en servidor-sock.c

El servidor usa un pool de threads para manejar las múltiples peticiones concurrentes. Una nueva adición comparado al servidor de la práctica 1, es que hemos añadido unos signal handler para cuando detecte el ctrl+C, el servidor pueda cerrarse correctamente.

1) Se verifica el puerto que sea correcto
2) Se crea el server socket
3) Se inicializa el pool of threads
4) Se define el signal handler para gestionar la señal de ctrl+C
5) Creamos un loop en el cual servidor estará escuchando a clientes nuevos
6) Cuando detecte una nueva conexión, este lo aceptara y le asignará un thread ejecutando la función servicio.
7) Cuando detecte que el servidor se cierra, entonces se libera la memoria de los threads y se cierra el socket del servidor.

3.4 función servicio

1. Cuando un cliente se conecta al servidor, se inicializa la función de servicio.
2. Cabe destacar que para cada vez que reciba o desea enviar un mensaje usando tcp el servidor tiene que traducir dicho mensaje usando funciones como host to network o network to host. Usando la librería de arpa/inet.
3. En él, el servidor recibe el primer byte encargado de decir el tipo de operación.
4. Después se usa un switch para determinar el tipo de operación que el cliente desea hacer, y determinar qué acciones hacer.
5. Nosotros hemos decidido que para por ejemplo enviar 3 parámetros el cliente deberá sus tres parámetros en 3 mensajes distintos. Por lo que el servidor dependiendo del tipo de operacion tendra un numero determinado de receive message,
6. Cuando haya recibido todas sus parámetros, ejecuta la función localmente y le envía al cliente la respuesta.
7. Por último retorna un 0, diciendo al servidor que se ha finalizado correctamente.

4. Compilación y Ejecución

Usamos un Makefile para compilar y ejecutar todo el sistema:

```
# Limpiar archivos antiguos
make clean

# Compilar todo
make
```

```
# Ejecutar el servidor
./servidor-sock 4500

# Variables de entorno para proxy
export IP_TUPLAS=localhost
export PORT_TUPLAS=4500

# En otra terminal, ejecutar el cliente
./app-cliente-1
./app-cliente-2
./app-cliente-3
```

Primero, compilamos todo el proyecto con el Make, esto compilará todos los archivos que necesita el servidor, el próximo, las librerías comunes y los tres clientes de prueba que hemos implementado.

Segundo, una vez compilado todo el proyecto pasamos a la ejecución del sistema donde iniciaremos el servidor. En este caso, el servidor requiere que le indiquemos un puerto de escucha como argumento (en nuestro caso para levantarlo en el puerto 4500). (si activamos el DEBUG, eso nos mostrará mensajes informativos sobre las conexiones y las operaciones).

Tercero, pasamos a ejecutar los clientes. Antes de ejecutar cada cliente, es necesario configurar las variables de entorno para que nuestro PROXY sepa el puerto y la dirección IP al que ha de conectarse. Antes de ejecutar cada cliente es necesario configurar las variables de entorno para que nuestro próximo sepa el puerto y la IP al que ha de conectarse. (el cliente 1 va a ser para funcionalidades básicas, el cliente 2 para los errores y límites, y el cliente 3 para la concurrencia.)