Sistemas distribuidos G81: Primera Práctica

József Iván Gafo - 100456709@alumnos.uc3m.es
Pablo Moreno González - 100451061@alumnos.uc3m.es

1. Introducción

El objetivo de esta práctica va a ser tratar de diseñar, implementar un sistema distribuido que esté basado en una cola de mensajes POSIX. Este sistema nos va a permitir, almacenar y gestionar tuplas clave-valor gracias a un servidor concurrente que nos va a manejar las peticiones de los clientes por medio de un Pool de threads.

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

El flujo del sistema va a ser que primero el cliente realiza una petición mediante la Api (claves.h). lo segundo, es que el proxy va a convertir la petición en un mensaje POSIX y lo va a enviar al servidor. Lo tercero, es que el servidor va a recibir la petición y la va a colocar en una cola de tareas compartida. Y lo último va a ser que un hilo de Pool de threads, va a procesarla y devolver la respuesta al cliente.

- El cliente llama a funciones como set_value, get_value, exist, etc.
- El proxy recibe la petición y la manda al servidor usando colas de mensajes.
- El servidor recibe la petición, la procesa y devuelve el resultado al proxy, que luego se lo envía al cliente.
- Las tuplas se almacenan en una lista enlazada dentro del servidor.

2.2 Estructura de los Archivos

Archivo	Descripción
claves.h	Define la API del sistema. No se puede modificar.
claves.c	Implementa la gestión de las tuplas en el servidor.
proxy-mq.c	Se encarga de la comunicación cliente-servidor con colas de mensajes.
servidor-mq.c	Recibe las peticiones del cliente y ejecuta las funciones de claves.c (Servidor concurrente con pool de threads)
app-cliente-1.c	Validar operaciones básicas secuenciales.
app-cliente-2.c	Evaluar límites, errores y manejo de excepciones.
app-cliente-3.c	Comprobar la concurrencia con múltiples procesos.

2.3 Sección de Pruebas

Para verificar el correcto funcionamiento del sistema hemos implementado tres clientes de prueba. Cada uno de ellos va a estar diseñado para probar diferentes aspectos de nuestra implementación como las operaciones básicas, los manejos de los errores y el manejo de la concurrencia.

2.3.1 app-cliente-1

Aquí el primer cliente realizará una serie de operaciones secuenciales para comprobar que funciona de manera correcta la API claves.h, verificando que las tuplas se gestionen de manera correcta 3

– Insertar una tupla (set_value) con clave 5.
– Verificar existencia (exist) de la clave insertada.
– Obtener la tupla (get_value) y comprobar los valores almacenados.
– Modificar la tupla (modify_value) y confirmar que los cambios se reflejan.
– Obtener la tupla modificada (get_value) y verificar los nuevos valores.
– Eliminar la tupla (delete_key) y asegurarse de que ya no exista.
– Insertar nuevamente la tupla (set_value) tras haberla eliminado.
– Destruir todas las tuplas (destroy) y confirmar que se eliminan correctamente.

i.e, el cliente prueba que todas las operaciones básicas funcionan de manera secuencial en un entorno controlado.

2.3.2 app-cliente-2

Aquí el segundo cliente evaluará casos más extremos y evaluará los errores forzados, garantizando que nuestra implementación maneje correctamente dichas entradas inválidas y dichas excepciones.

– Intentar insertar una tupla con N_value2 = 0 (valor fuera del rango permitido).
– Intentar insertar una tupla con N_value2 = 33 (superando el máximo permitido de 32).
– Intentar insertar una tupla con clave negativa (-1).
– Insertar una tupla válida y verificar get_value con distintos tamaños de N_value2.
– Intentar acceder a una clave inexistente (get_value con clave 999).
– Insertar y recuperar una tupla con el tamaño máximo de V_value2 = 32.
– Destruir todas las tuplas (destroy) y confirmar que se eliminan.

i.e; con esta prueba, confirmamos que nuestra implementación maneja correctamente los valores que están fuera de rango, las claves inexistentes, el tamaño máximo de los datos

2.3.3 app-cliente-3

Aquí el tercer cliente va a simular un escenario de concurrencia con múltiples procesos, hijos, realizando operaciones, simultáneamente, y con esto podremos verificar la robustez del sistema.

– Se crean 20 procesos hijos (NUM_HIJOS = 20).
– Cada hijo ejecuta 1.000 operaciones (OPERACIONES_POR_HIJO =1.000).
– Cada hijo repite las operaciones que se hacen en app-cliente-1 sin el último insert y destroy

> Ahora el destroy se ejecuta una vez que todos los hijos hayan acabado, asegurando que la lista de tuplas se limpia completamente después de la prueba.
>
> También se puede deducir si en un futuro despliegue del sistema, habría que definir permisos para ver quién puede ejecutar el destroy, porque afecta a todos los clientes que lo estén usando.

i.e; este cliente va a verificar que el servidor puede manejar múltiples clientes, concurrentes, sin errores, ni inconsistencias en el almacenamiento

3. Implementación

3.1 Gestión de tuplas en claves.c

Las tuplas se almacenan en una lista enlazada. Las operaciones que hemos implementado son:

- set_value: Inserta una tupla si la clave no existe.

- get_value: Recupera los valores de una tupla.
- modify_value: Modifica una tupla existente.
- delete_key: Borra una tupla del almacenamiento.
- exist: Comprueba si una clave existe y está registrada.
- destroy: Elimina todas las tuplas almacenadas.

3.2 Intermediario en proxy-mq.c

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

1) Recibe una petición de la API
2) LA TRANSFORMA EN UN MENSAJE POSIX y lo envía al servidor
3) Espera la respuesta y la devuelve al cliente

3.3 Servidor en servidor-mq.c

El servidor usa un pool de threads para manejar las múltiples peticiones concurrentes.

1) Se inicializa una cola de tareas compartida
2) Se crean y los trabajadores que van a permanecer en espera de peticiones
3) Entonces cuando llega una petición, se va a colocar en la cola y se despierta un hilo disponible
4) Por último, el hilo extrae la petición, la procesa y envía la respuesta al cliente

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-mq

# En otra terminal, ejecutar el cliente
LD_LIBRARY_PATH=. ./app-cliente
```

Si todo está bien, el cliente 1 mostrará:

```
Tupla insertada correctamente
La clave 5 existe en el sistema
Tupla obtenida:
Value1: ejemplo de valor 1
Value2: {2.30, 0.50, 23.45}
Value3: {x: 10, y: 5} ...
```

El servidor mostrará en caso que esté el DEBUG activado:

```
[Servidor] Recibida solicitud de cliente PID 12345
[Servidor] Operación recibida: SET (key=5, value1=ejemplo, N_value2=3, value3={x=10, y=5})
[Servidor] Resultado de la operación: 0
...
```

– Si todo funciona correctamente dentro de app-cliente 2; entonces mostrará un CORRECTO en todos los chequeos de errores.

– Si todo funciona correctamente dentro de app-cliente-3, no debería imprimir ningún error, ni tampoco debiera el servidor hacer un crash , o una corrupción de memoria.