Práctica 2

Ejercicio Evaluable 2:
Sockets TCP

Miguel Jimeno Casas - 100495932 - 100495932@alumnos.uc3m.es

Alberto Menchén Montero - 100495692 - 100495692@alumnos.uc3m.es

Grupo 82 - Ingeniería informática

1. Introducción

Esta práctica tiene como objetivo la implementación de un servicio distribuido de almacenamiento y gestión de tuplas <key, value1, value2, value3> mediante sockets TCP en C. Partiendo de la base implementada en el Ejercicio 1 (usando colas POSIX), se desarrolla un nuevo modelo de comunicación cliente-servidor utilizando sockets TCP y un protocolo de aplicación basado en mensajes codificados en JSON.

El sistema está compuesto por:

- Biblioteca del cliente: `proxy-sock.c`
- Servidor: `servidor-sock.c`
- Lógica de tuplas: `claves.c`, `claves.h`, `real.h`
- Aplicaciones cliente: `app-cliente-1.c`, `app-cliente-2.c`

---

2. Diseño General del Sistema

El diseño se basa en un modelo cliente-servidor. El cliente, mediante la biblioteca `libclaves.so`, se comunica con el servidor para solicitar operaciones sobre tuplas. La comunicación se realiza a través de TCP sockets y los mensajes intercambiados están codificados en formato JSON, asegurando independencia del lenguaje.

Cada operación del cliente se transforma en un mensaje JSON que se envía al servidor, el cual interpreta, ejecuta la acción solicitada, y devuelve una respuesta también en JSON.

---

3. Protocolo de Comunicación

El protocolo de aplicación definido es simple y se basa en la estructura de objetos JSON con los siguientes campos:

```C/C++
{
  "operation": <código_de_operacion>,
  "key": <clave_entera>,
  "value1": <string>,
  "N_value2": <entero>,
  "V_value2": [<float>, ...],
```

```
  "value3": { "x": <int>, "y": <int> },
  "resultado": <resultado>
}
```

Los códigos de operación son:

- 1: set_value
- 2: get_value
- 3: modify_value
- 4: delete_key
- 5: exist
- 6: destroy

Cada operación tiene requerimientos específicos sobre los campos presentes.

---

4. Implementación del Servidor (`servidor-sock.c`)

El servidor se inicia con `./servidor <puerto>` y queda escuchando conexiones entrantes. Por cada conexión, se crea un hilo que:

1. Recibe el mensaje JSON.
2. Lo interpreta según el campo `operation`.
3. Llama a las funciones internas (prefijadas con `_`) para ejecutar la operación.
4. Devuelve una respuesta JSON al cliente con el campo `resultado`. Si el resultado se mantiene con 'resultado=-2', se considera que se ha producido un error con la comunicación entre cliente-servidor.

Ejemplo de fragmento para `get_value`:

```C/C++
case 2:
```

```
    key = cJSON_GetObjectItem(json, "key")->valueint;
    result = _get_value(key, value1, &N_value2, V_value2,
&value3);
    if (result == 0) {
        cJSON_AddStringToObject(json, "value1", value1);
        // ... agregar resto de datos al JSON
    }
    break;
```

La concurrencia se maneja con `pthread_create`, y el acceso a la lista de tuplas está protegido por un mutex (`lista_mutex`).

---

5. Implementación del Proxy del Cliente (`proxy-sock.c`)

El proxy implementa la biblioteca `libclaves.so` expuesta al cliente. Cada función de la API (como `set_value`, `get_value`, etc.):

1. Crea un objeto JSON con los datos de la operación.

2. Lo convierte a cadena y lo envía al servidor usando `send`. Conectándose con el servidor mediante la función `connect_to_server.` Que realizar el conect() del sd creado con el servidor en función. Dentro de la funcion se usna las variables de entorno `IP_TUPLAS` y `PORT_TUPLAS`, cumpliendo así el requisito de configuración flexible sin hardcoding.

3. Espera la respuesta y la analiza usando `cJSON`.

Por ejemplo, en `set_value`:

```C/C++
cJSON *json = cJSON_CreateObject();
cJSON_AddNumberToObject(json, "operation", 1);
cJSON_AddNumberToObject(json, "key", key);
...
```

---

7. Clientes de Prueba

- `app-cliente-1.c`: realiza una destrucción del sistema invocando `destroy()`. Debido a que Destroy es una función que solo debe poder ejecutar el administrador, hemos modificado el archivo makefile para ejecutar make clean_tuplas en la que se ejecuta el app-cliente-1 limpiando las tuplas del sistema para después correr los app-cliente-2 en concurrencia.

- `app-cliente-2.c`: ejecuta una batería completa de pruebas (inserción, consulta, modificación, borrado, etc.) y muestra los resultados en formato JSON. Podemos comprobar que todas las funciones retornan los valores que deben devolver en cada una de las operaciones y cómo está compuesto el JSON en ese instante, visualizando los valores de entrada y comprobando la salida correcta.

  En el `makefile` tenemos declarado "Lanzar_pruebas" para ejecutar 5 app-cliente-2 en concurrencia. A parte, para no estar metiendo constantemente en la terminal la línea de código, antes de ejecutar los clientes:

```
C/C++

env IP_TUPLAS=localhost PORT_TUPLAS=4500
```

Quedándonos así el código del `makefile`.

```
C/C++
clean_tuplas:
	@echo "Limpieza de tuplas con Cliente 1 (destroy)..."
	@IP_TUPLAS=127.0.0.1 PORT_TUPLAS=4500 ./app-cliente-1

lanzar_pruebas:
	@echo "Ejecutando 5 instancias de Cliente 2 en paralelo
tras limpiar tuplas..."
	@IP_TUPLAS=127.0.0.1 PORT_TUPLAS=4500 ./app-cliente-2 & \
	IP_TUPLAS=127.0.0.1 PORT_TUPLAS=4500 ./app-cliente-2 & \
	IP_TUPLAS=127.0.0.1 PORT_TUPLAS=4500 ./app-cliente-2 & \
	IP_TUPLAS=127.0.0.1 PORT_TUPLAS=4500 ./app-cliente-2 & \
	IP_TUPLAS=127.0.0.1 PORT_TUPLAS=4500 ./app-cliente-2
```

---

8. Compilación y Ejecución

La compilación se gestiona mediante `makefile`. Este genera:

- La biblioteca compartida `libclaves.so`
- Ejecutables `servidor`, `app-cliente-1` y `app-cliente-2`

Comandos:

En una terminal ejecutamos el servidor que va a estar recibiendo operaciones y escuchando constantemente.

```C/C++
$ make
$ ./servidor 4500
```

Mientras que en una terminal en paralelo, debemos de:

```C/C++
$ make clean_tuplas //Que ejecutara el app-cliente-1
$ make lanzar_pruebas //Que ejecutara el app-cliente-2
```

---

9. Conclusión

La solución propuesta cumple con los requisitos funcionales y de diseño establecidos. Se ha implementado un sistema robusto y concurrente basado en sockets TCP y JSON.

La decisión de utilizar JSON permite independencia del lenguaje y facilita la depuración. El uso de sockets TCP garantiza comunicaciones fiables y orientadas a conexión, adecuadas para un sistema distribuido robusto.