Práctica 3

Ejercicio Evaluable 3:
RCP

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

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

Grupo 82 - Ingeniería Informática

1. Introducción

La tercera entrega del proyecto de Sistemas Distribuidos tiene como objetivo la implementación de un servicio distribuido para la gestión de tuplas de la forma `<key, value1, value2, value3>`, haciendo uso de la tecnología **ONC RPC (Open Network Computing Remote Procedure Call)**. Esta evolución sustituye el uso de sockets TCP y colas de mensaje de entregas anteriores por un sistema de invocación remota de procedimientos, lo cual permite una **mayor abstracción**, **menor gestión manual de la comunicación** y una implementación más declarativa a través del uso de interfaces `.x`.

El sistema mantiene la lógica original de operaciones sobre tuplas implementada en `real.h` y `real.c`, y se reestructura ahora en torno a tres componentes principales:

- El archivo de definición de interfaz **`claves_rpc.x`**, que describe los tipos de datos y funciones expuestas por el servidor.
- El servidor `servidor-rpc.c`, que implementa las funciones declaradas en la interfaz y se comunica con la lógica de tuplas.
- El cliente `proxy-rpc.c`, que actúa como una biblioteca intermedia (`libclaves.so`) para invocar procedimientos remotos de forma transparente.

2. Diseño General del Sistema

El sistema mantiene la arquitectura cliente-servidor tradicional, pero ahora delega en el middleware de ONC RPC la serialización de datos, el manejo de conexiones y la ejecución remota. La **comunicación se realiza mediante llamadas a funciones como si fueran locales**, aunque en realidad son llamadas remotas gestionadas por las funciones generadas automáticamente por `rpcgen`.

El archivo clave del diseño es `claves_rpc.x`, que define tanto los **tipos de datos** utilizados por las tuplas como los **procedimientos remotos** disponibles. Este archivo es el punto central que conecta el cliente con el servidor y permite que el sistema sea distribuido de forma natural.

---

3. Análisis del archivo claves_rpc.x

El archivo `claves_rpc.x` es el núcleo del sistema RPC. Contiene:

- Definiciones de estructuras personalizadas: `Coord`, `Entrada`, `SoloClave`, `RespuestaGet`.
- Declaración del programa RPC: `CLAVES_PROG`, con versión `CLAVES_VERS`.

Estructuras definidas

```C/C++
struct Coord {
    int x;
    int y;
};
```

Esta estructura representa el campo `value3` de la tupla, codificando una coordenada bidimensional.

```C/C++
struct Entrada {
    int key;
    string value1<256>;
    int N_value2;
    struct {
        u_int V_value2_len;
        double *V_value2_val;
    } V_value2;
    Coord value3;
};
```

Es la estructura que encapsula **todos los valores** de la tupla. Se utiliza en las operaciones que requieren establecer inserciones como (`set_value`) o modificaciones (`modify_value`).

```C/C++
struct SoloClave {
    int key;
};
```

Una estructura auxiliar usada en operaciones donde solo se necesita la clave (`get_value`, `delete_key`, `exist`).

```C/C++
struct RespuestaGet {
    int resultado;
    string value1<256>;
    int N_value2;
    struct {
        u_int V_value2_len;
        double *V_value2_val;
    } V_value2;
    Coord value3;
};
```

Permite devolver una tupla completa junto con el código de resultado de la operación. Se utiliza exclusivamente en `get_value`.

Declaración de procedimientos remotos

```C/C++
program CLAVES_PROG {
    version CLAVES_VERS {
        int set_value(Entrada) = 1;
        RespuestaGet get_value(SoloClave) = 2;
        int modify_value(Entrada) = 3;
        int delete_key(SoloClave) = 4;
        int exist(SoloClave) = 5;
        int destroy(void) = 6;
    } = 1;
} = 0x20000001;
```

El programa `CLAVES_PROG` define **seis funciones remotas**, cada una asociada a un identificador único:

- `set_value`: Inserta una nueva tupla.
- `get_value`: Recupera una tupla por clave.
- `modify_value`: Modifica una tupla existente.
- `delete_key`: Elimina una tupla.
- `exist`: Verifica la existencia de una clave.
- `destroy`: Borra todos los elementos del sistema (privilegiada).

Los identificadores de programa (`0x20000001`) y versión (`1`) son convenciones de ONC RPC y deben mantenerse únicos dentro del sistema.

4. Implementación del servidor (servidor-rpc.c)

El archivo `servidor-rpc.c` contiene la lógica de **implementación de los servicios** descritos en el `.x`. Cada función implementa el sufijo `_svc`, necesario para ONC RPC.

Por ejemplo, la función `set_value_1_svc`:

```
C/C++
int *set_value_1_svc(struct Entrada *entrada, struct svc_req *rqstp) {
    ...
    result = _set_value(entrada->key, entrada->value1, entrada->N_value2,
entrada->V_value2.V_value2_val, coord);
    ...
}
```

Internamente, las funciones llaman a la lógica ya existente (`_set_value`, `_get_value`, etc.) para mantener la separación de responsabilidades entre comunicación. También se

define una estructura estática para devolver la respuesta (`static int result`) como exige el protocolo RPC.

La función `get_value_1_svc` devuelve un puntero a una estructura `RespuestaGet` estática, rellenando todos los campos de la tupla si la operación tiene éxito.

5. Implementación del cliente (proxy-rpc.c)

El proxy actúa como una **biblioteca dinámica** que encapsula todas las llamadas remotas. Se conecta al servidor usando la IP proporcionada por la variable de entorno `IP_TUPLAS`, y usa `clnt_create(...)` para establecer la conexión RPC.

Cada operación traduce sus parámetros a las estructuras definidas en el `.x` y realiza la llamada remota. Ejemplo de `set_value`:

```
C/C++
int set_value(int key, char *value1, int N_value2, double *V_value2, struct
Coord value3) {
    struct Entrada entrada;
    ...
    return *set_value_1(&entrada, clnt);
}
```

De esta forma, el cliente simplemente invoca funciones locales que delegan la operación en el servidor remoto de forma transparente.

Además, la función `init_client()` se encarga de inicializar la conexión y evitar múltiples inicializaciones, aportando robustez.

6. Compilación y Ejecución

La compilación y gestión de la ejecución del sistema se realiza mediante un `makefile` diseñado específicamente para esta entrega. Dicho `makefile` automatiza tanto la generación de los archivos derivados de `rpcgen` como la construcción de todos los binarios necesarios.

Las principales funcionalidades del `makefile` son:

- **Generación de archivos RPC**: Usa la herramienta `rpcgen` para procesar el archivo `claves_rpc.x`, generando automáticamente:
    - `claves_rpc.h`
    - `claves_rpc_clnt.c`
    - `claves_rpc_svc.c`
    - `claves_rpc_xdr.c`
- **Compilación de componentes**:
    - **Servidor**: Compila `servidor-rpc.c` junto con `claves_rpc_svc.c`, `claves_rpc_xdr.c` y la lógica de tuplas (`real.c`) para crear el ejecutable `servidor`.
    - **Biblioteca dinámica**: Compila `proxy-rpc.c`, `claves_rpc_clnt.c`, `claves_rpc_xdr.c` y `real.c` en una biblioteca compartida llamada `libclaves.so`, que será utilizada por los clientes.
    - **Clientes de prueba**: Compila `app-cliente-1.c` y `app-cliente-2.c`, enlazándolos con la biblioteca `libclaves.so` y los objetos de RPC necesarios.
- **Objetivos especiales definidos**:
    - `make clean_tuplas`: Ejecuta el cliente `app-cliente-1` para limpiar todas las tuplas en el sistema (invoca la operación `destroy`).
    - `make lanzar_pruebas`: Ejecuta cinco instancias de `app-cliente-2` en paralelo, permitiendo realizar pruebas concurrentes de inserción, consulta, modificación y borrado de tuplas.
- **Variables de entorno**:
    - Antes de ejecutar los clientes, el `makefile` establece las variables `IP_TUPLAS` y `PORT_TUPLAS`, permitiendo una configuración flexible y dinámica de la IP y puerto del servidor sin necesidad de recompilar.

El flujo de ejecución recomendado sería:

```C/C++
make
./servidor
make clean_tuplas
make lanzar_pruebas
```

De esta forma, se inicia el servidor en segundo plano, se limpia el sistema de tuplas iniciales y, posteriormente, se lanzan las pruebas en paralelo.

7. Conclusión

La implementación mediante ONC RPC representa un avance significativo en la abstracción y modularidad del sistema distribuido. Al centralizar la definición de datos y operaciones en `claves_rpc.x`, se garantiza:

- Coherencia entre cliente y servidor.
- Reducción de errores manuales de serialización.
- Independencia del protocolo subyacente.

La transformación del sistema desde sockets TCP con JSON hacia llamadas RPC es un paso natural en la evolución hacia sistemas distribuidos más declarativos y mantenibles.

La estructura modular y reutilización de la lógica previa (`real.h`, `real.c, claves.c`) permiten que el nuevo sistema mantenga la funcionalidad ya validada, mientras mejora la legibilidad, facilidad de mantenimiento y fiabilidad de la comunicación.