Ejercicio 3

Sistemas Distribuidos

Alberto Penas Díaz:
NIA: 100471939
Correo: 100471939@alumnos.uc3m.es
Titulación: Ingeniería informática

Índice

Instrucciones de Compilación y Ejecución

Para compilar y ejecutar el programa se ha usado la herramienta CMake (disponible en Guernika). Se ha creado un script en bash para automatizar la compilación. Este script se llama **compile.sh** y se encarga de crear todas las librerías y ejecutables del proyecto. Cabe destacar que en el fichero CMakeLists.txt se vincula directamente la librería libclaves.s con el cliente. Para ejecutar este script de compilación, primero de todo hay que otorgarle permisos de ejecución con: `chmod +x ./compile.sh` Posteriormente, se puede ejecutar el script de compilación con `./compile.sh` Este script[1] generará el directorio *cmake-build-release/* en el que se encuentran todos los ejecutables. Al contrario que las prácticas anteriores, en este caso, para ejecutar los programas no se necesita ningún argumento adicional en el caso del servidor. A diferencia del cliente, en el que sigue siendo necesario determinar la ip mediante una variable de entorno. Estos programas se pueden ejecutar de la siguiente manera:

```
>> ./cmake-build-release/servidor-mq > server.txt &[2]
>> env IP_TUPLAS=localhost ./cmake-build-release/app-cliente
```

NOTA IMPORTANTE: Los ficheros generados por el fichero rpc_api.x se generan mediante rpcgen. La llamada a esta herramienta se realiza dentro del script de compilado `compile.sh` Estos ficheros no se han modificado en absoluto (se usa el código tal cual lo genera rpcgen) por simplicidad.

Para la batería de pruebas, simplemente es necesario ejecutar el fichero de python test.py desde terminal, especificando como argumento la ip del servidor. Antes de ejecutarlo, es muy importante tener el proceso del servidor ejecutándose al mismo tiempo, si no, los test no pasarán.

```
>> ./cmake-build-release/servidor-mq &
>> python3 test.py localhost
```

Diseño del Programa

Diseño General

La arquitectura principal es idéntica a la de la práctica anterior pero el proceso de comunicación se establece, esta vez a mediante ONC-RPC. Este tipo de arquitectura compromete la “libertad” que generan los sockets además de perder cierta parte de rendimiento. No obstante, simplifica significativamente la gestión de peticiones, liberando prácticamente toda la carga de concurrencia y gestión de hilos al programador, únicamente teniendo que preocuparse por el diseño de la API en el fichero *api_rpc.x*

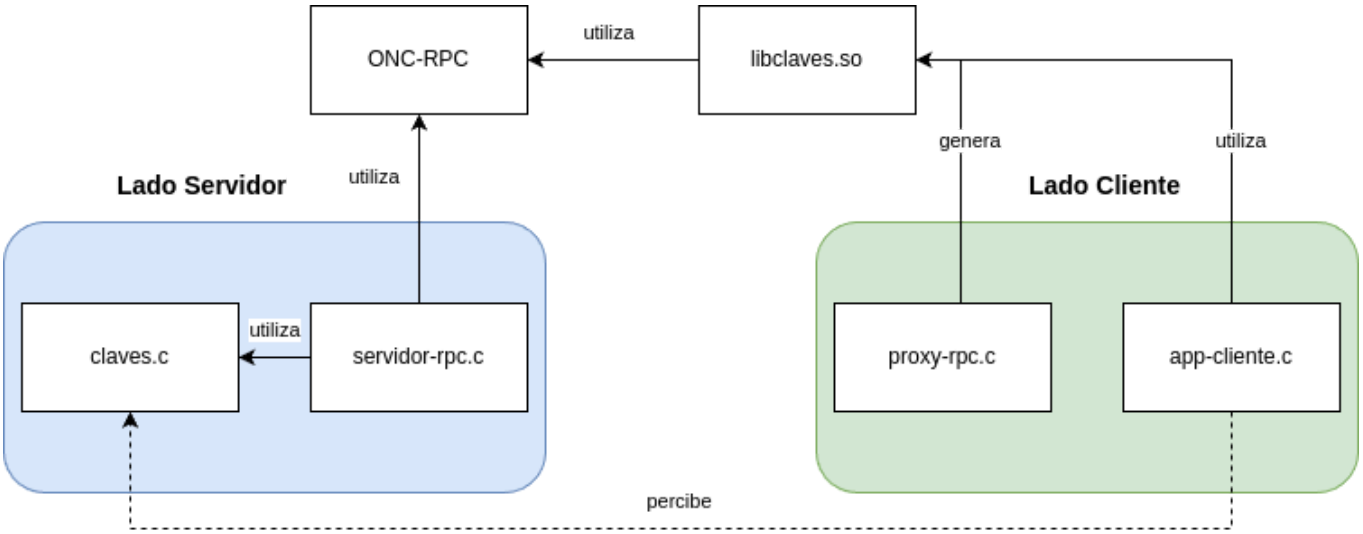
En este fichero, se devuelven siempre unas estructuras concretas (excepto en destroy, que por ser la función más simple, únicamente devuelve un número entero). De esta forma se simplifica muchísimo la legibilidad del código, convirtiéndolo en un código más trazable y transparente. Es importante mencionar que esta implementación **no permite el uso de la librería de encriptado** que se ha estado

[1] En el propio script de compilación se generan los archivos .c llamando al programa rpcgen. El programa **no funcionará** si rpcgen no está instalado.
[2] Es recomendable ejecutar el proceso del servidor en background con & para así poder ejecutar también el proceso del cliente en la misma terminal. También se redirige la salida estándar a server.txt para que no moleste.

utilizando, ya que el propio rpcgen utiliza un sistema de serialización propio generado en base a la estructura definida en el fichero *api_rpc.x*

Hubiera sido posible enviar información encriptada si el fichero .x se hubiera definido que todo mensaje entre cliente y servidor tuviera que estar compuesto de secuencias de Bytes aparentemente aleatorias y de longitud arbitraria, lo que no solo hubiera complicado de manera tremenda su implementación, si no que el código habría perdido gran parte de su trazabilidad y coherencia. En pos de mantener el código limpio y legible, se ha descartado esta opción.

De manera esquemática, el proceso de diagrama de comunicación y llamadas entre ficheros es el siguiente:



El bloque “ONC-RPC” representa los ficheros generados por *rpcgen* mediante el archivo rpc_api.x:

- rpc_api.h - Utilizado tanto por el cliente como por el servidor (cabeceras)
- rpc_api_xdr - Utilizado tanto por el cliente como por el servidor (serialización)
- rpc_api_clnt - Utilizado por el cliente
- rpc_api_svc - Utilizado por el servidor

No tener control sobre el puerto sobre el que opera la aplicación también ha supuesto un reto para comunicar dos programas. RPC utiliza el puerto 135 para escuchar una conexión entrante y redirigir la petición a un puerto (entre el 5000 y 6000) que se asigna de manera dinámica en función del puerto de escucha escogido por el programa servidor.

Por último, con respecto a la concurrencia del servidor, esta viene implementada en la propia herramienta de ONC-RPC, por lo que no se ha tenido en cuenta a la hora de gestionar peticiones entrantes de nuevos clientes. No obstante, esto no pasa en la librería de claves.c, que sigue requiriendo mecanismos para evitar escrituras de memoria simultáneas en memoria.

Batería de Pruebas

Para la batería de pruebas se ha utilizado Unitest en Python. Aunque las pruebas funcionales están explicadas en el propio fichero de python en el que están programadas, de forma resumida, son las siguientes:

1. **test_set_get**: verifica la inserción y obtención de una tupla tras reiniciar el servidor, comprobando los mensajes de éxito.

2. **test_modify**: tras reiniciar el servidor, inserta y modifica una tupla, verificando el mensaje de modificación correcta.

3. **test_delete_exist**: reinicia el servidor, inserta una tupla, comprueba su existencia, la elimina y verifica que ya no existe.

4. **test_destroy**: prueba la función destroy y comprueba el mensaje “Servidor destruido”.

5. **test_stress_1**: inserta 100 tuplas de forma secuencial y comprueba que todas se insertan correctamente.

6. **test_zzz_concurrency_stress**: simula 5 clientes concurrentes que insertan y obtienen tuplas, y luego verifica globalmente que todos los valores estén correctos. (El nombre zzz se debe a que unittest ejecuta los test en orden alfabético y por razones de depuración de código me convenía que ésta prueba fuera la última en ejecutarse para comprobar el estado del servidor al finalizar)

7. **test_nok_1**: intenta un set con vector de 1000 elementos para comprobar el error “N debe estar entre 1 y 32”.

8. **test_nok2**: intenta modificar una tupla inexistente y comprueba “Error al modificar la tupla”.

9. **test_nok3**: intenta eliminar una tupla inexistente y comprueba “Error al eliminar la tupla”.

10. **test_nok4**: intenta obtener una tupla inexistente y comprueba “Error al obtener la tupla”.

11. **test_nok5**: inserta dos veces la misma clave y verifica “Error al insertar la tupla”.

12. **test_nok6**: hace set con N=0 y N=33 para comprobar dos veces el error de rango de longitud de vector.

13. **test_nok7**: usa clave no numérica para comprobar “La clave debe ser un número entero”.

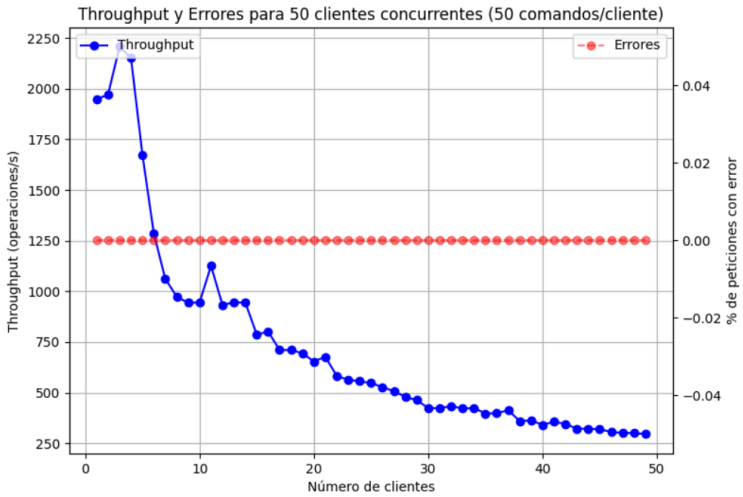
14. **test_nok8**: inserta un value1 de 256 caracteres y comprueba el límite de longitud (“máximo 255 caracteres”).

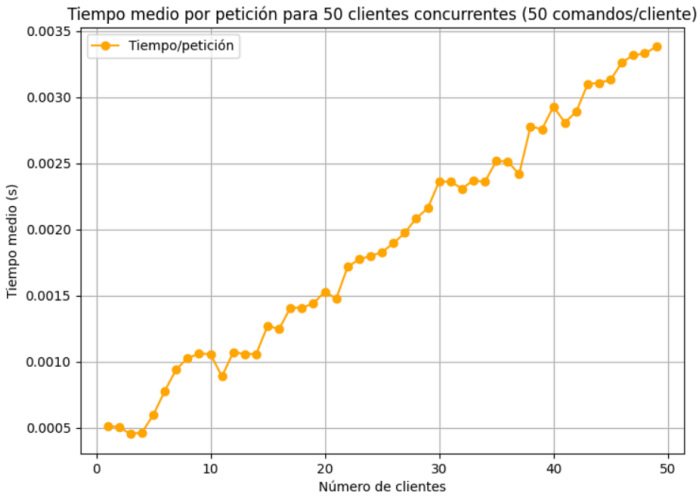
15. **test_nok9**: especifica menos valores de los indicados en el vector para comprobar “Número insuficiente de valores para v2”.

16. **test_nok10**: inserta con coordenadas finales no numéricas y verifica “Las coordenadas deben ser números enteros”.

Evaluación de rendimiento

Con respecto a la práctica anterior, en la que se usaba sockets para la comunicación entre cliente y servidor, el rendimiento es significativamente peor usando RPCs. Tal y como se puede observar en la siguiente gráfica (donde cliente y servidor se estaban ejecutando en la misma máquina):





Con respecto a los Sockets, **usar RPCs es en torno a un 75% más lento**. Teniendo en cuenta además que el baremo que se ha seguido como comparación es la gráfica propuesta en la memoria de la práctica anterior (en la que con 1 cliente se obtenía un throughput de 8000 operaciones por segundo) y en donde toda la comunicación entre cliente y servidor era serializada y encriptada mediante un algoritmo de criptografía simétrica de 64 rondas.

Contenido Extra

El uso de RPCs ha mermado de gran manera la facilidad de introducir contenido extra debido a lo explicado anteriormente. La librería de encriptado dificulta mucho el diseño de la API y por consiguiente, cambiaría totalmente toda la implementación del servidor. Además, hubiera conllevado la modificación de los archivos generados por la herramienta rpcgen. Lo que se ha tratado de evitar a toda cosa en pos de la integración total de estos archivos y de la herramienta rpcgen con el resto del código.