# Ejercicio Evaluable 1

Sistemas Distribuidos

Raúl Aguilar Arroyo

Grupo 80

17 mar 2024

# Diseño

## Diseño general

En el diseño general, he buscado prioritariamente la simplicidad y la claridad sobre la eficiencia. Por lo tanto, en la implementación de las colas y mensajes, se han empleado dos estructuras:

### Struct request

```C/C++
struct request
{
  int queue_id;   // id of the client queue to send the response
  int op;       //  0: init, 1: set, 2: get, 3: modify_value, 4: delete_key, 5: exist
  int key;
  char value1[255];
  int N_value2;
  double V_value2[31];
};
```

Esta estructura es generada por el cliente y enviada al servidor. Se utiliza independientemente de la función que se quiera ejecutar, ya que tiene espacio reservado para los parámetros de cualquier función.

### Struct response

```C/C++
struct response
{
  char value1[255];
  int N_value2;
  double V_value2[31];
  int error;
};
```

Esta estructura es generada por el servidor y enviada al cliente, independientemente de la función ejecutada. Es responsabilidad del cliente buscar en los atributos aquellos que corresponden a la función ejecutada. El argumento 'error' se reutiliza en las funciones 'exist' y 'delete' para conocer el resultado de la ejecución.

Esta gestión de las colas y la comunicación, aunque ineficiente en términos de espacio desperdiciado en la mayoría de los casos, simplifica enormemente el proceso, eliminando la necesidad de "handshakes" y permitiendo que la comunicación se realice únicamente a través de dos colas: la del cliente y la del servidor.

Una alternativa sería implementar un tipo de "handshake" en el que el cliente informa al servidor sobre la función que va a ejecutar y el tamaño de los parámetros. Entonces, el

servidor responderá con el identificador de la cola personalizada que se utilizará exclusivamente para esa comunicación. El cliente podría realizar la solicitud con los parámetros de la función y, a su vez, proporcionar al servidor el identificador de una nueva cola donde recibiría la salida de la función. Aunque esta implementación es más eficiente, considero que la complejidad adicional que conlleva no justifica su uso.

# Implementación de la biblioteca 'libclaves'

El archivo 'claves.c', además de las funciones 'públicas' que proporciona, incluye otras funciones internas que ayudan a su funcionamiento. En cada ejecución de una función, se verifica si la cola del cliente ya ha sido inicializada; de lo contrario, se crea. El identificador de la cola está asociado al identificador del proceso. Reconozco que esto podría causar conflictos en máquinas con muchos procesos o en situaciones donde los procesos se crean y destruyen rápidamente, ya que podría haber colisiones. Sin embargo, considero que en este caso, este riesgo no es significativo. En caso de conflicto, lo sustituirá con algún tipo de algoritmo de HASH/UUID.

Las solicitudes que la biblioteca hace al servidor son bloqueantes; es decir, cada vez que se realiza una solicitud, el programa se bloquea esperando la respuesta. Se implementa un temporizador de 5 segundos; si no se recibe respuesta en este tiempo, la función devuelve -1, asumiendo que ha ocurrido un error en las comunicaciones.

# Cliente

Con el fin de simplificar las pruebas y aumentar la versatilidad del cliente, he optado por implementarlo como una suerte de intérprete de comandos. Al ejecutarlo, se debe proporcionar como parámetro la ruta de un archivo "script" que contiene la secuencia de comandos que el cliente ejecutará, invocando a la biblioteca libclaves.
El formato de este archivo script está incluido en Uso del ejecutable cliente.

# Servidor

Los elementos se almacenan en memoria, utilizando un array que se relocaliza cuando se añaden o eliminan elementos. Esta implementación es menos eficiente que otras alternativas, como un árbol binario de búsqueda por clave o un mapa de memoria con hashes funcionando como diccionario. Sin embargo, es mucho más sencilla de implementar. Además, el hecho de que las solicitudes puedan llegar a ser relativamente lentas me permite realizar pruebas más exhaustivas en cuanto a la competencia de datos.

El servidor utiliza una única cola para recibir las solicitudes. Cada vez que llega una solicitud, se crea un nuevo hilo que la procesa, copiando la solicitud antes de que el servidor pueda procesar la siguiente.

Se ha optado por implementar un modelo de threads bajo demanda por las siguientes razones:

- **Mayor eficiencia**: Si asumimos que la carga del servidor es esporádica, es decir, que

hay largos períodos de espera entre solicitudes, este enfoque permite que el sistema no reserve recursos para threads que no se están utilizando. Además, evita que el servidor se bloquee procesando peticiones y facilita una gran escalabilidad en caso de que se produzcan muchas solicitudes simultáneas en algún momento.

- **Mayor rendimiento**: Si consideramos que las operaciones de búsqueda y modificación de datos en el array de elementos pueden ser bastante lentas, el tiempo de creación del thread en comparación con el tiempo de ejecución de las solicitudes es poco relevante.

# Compilación y ejecución

## Uso del ejecutable cliente

Se debe generar un archivo de texto que contenga una secuencia de instrucciones. Las instrucciones disponibles son las siguientes (no voy a explicar su funcionamiento, ya que corresponde al enunciado):

1. init
2. set <key> <value1> <N_value2> <value2[0] value2[1], ... >
3. get <key>
4. modify <key> <value1> <N_value2> <value2[0] value2[1], ... >delete <key>
5. exist <key>

Los parámetros de las funciones deben ir separados por espacios, y cada función debe estar en una línea independiente, como se muestra en el ejemplo:

```
Unset
init
set 001 "Texto al azar 1" 3 8.2 3.6 5.1
set 002 "Texto al azar 2" 1 12.34532
get 001
delete 002
modify 001 "Texto modificado" 2 8.3 5.2
get 001
exist 002
```

**Ejemplo de compilación y ejecución** de cliente/servidor:

```
1. make
2. # export LD_LIBRARY_PATH=/ruta/a/libclaves.so:$LD_LIBRARY_PATH
3. ./servidor
4. ./cliente ruta/al/fichero/script.txt
```

# Batería de pruebas

**Prueba de funcionamiento básico**
Archivo: `valid.txt`

Este archivo prueba todas las funciones básicas de manera general, es útil para verificar si se ha compilado correctamente.

**Prueba de casos inválidos (rangos, valores, etc)**
Archivo: `invalid.txt`

Incluye pruebas de las funciones considerando sus límites de valores, rangos, etc.

**Servidor no se ha ejecutado**
Salida esperada/obtenida: Error opening server queue, is the server running?: No such file or directory

**Servidor se detiene en mitad del proceso y no responde a un request que ha recibido correctamente**
Salida esperada/obtenida: Error receiving message Tiempo de espera agotado

**Prueba de concurrencia:**
Esta prueba la he realizado insertando 100.000 elementos y después en dos procesos paralelos uno elimina y otro consulta, de esta forma se puede observar como los procesos se ejecutan simultáneamente, alternando aciertos y errores (dependiendo de la planificación del proceso).
Ejecución:

1. `ejecutar create_set_file.py`
2. `ejecutar create_get_file.py`
3. `duplicar get.txt y reemplazar todos los get por delete`
4. `ejecutar ./servidor &`
5. `ejecutar ./cliente set.txt`
6. `ejecutar ./cliente tests/get.txt > get.out & ./cliente tests/delete.txt > del.out &`
7. `Comparar leer get.out y ver cómo se van alternando los aciertos y fallos cada cierto tiempo.`

Nota: los archivos de entrada no se han incluido, en su lugar se proporcionan scripts para crearlos ya que eran demasiado pesados.

**Prueba de punteros null**
Esta prueba no se ha podido incluir como un archivo, ya que se implementa directamente en el cliente. Consiste en pasar punteros null como argumentos a las funciones.