Ejercicio Evaluable 1: Colas de mensajes

Leganés

Ingeniería Informática

Sistemas Distribuidos

Alejandro Prieto Macías NIA 100383428 Gr. 81

Laura Sánchez Cerro NIA 100383419 Gr. 81

Índice

Introducción

El motivo de este ejercicio se basa en la idea de poder aprender las técnicas para poder crear una aplicación cliente-servidor de forma concurrente. Además, se pretende aprender a administrar sistema de comunicación para sistemas distribuidos, es decir, para ordenadores que no comparten el espacio de memoria.

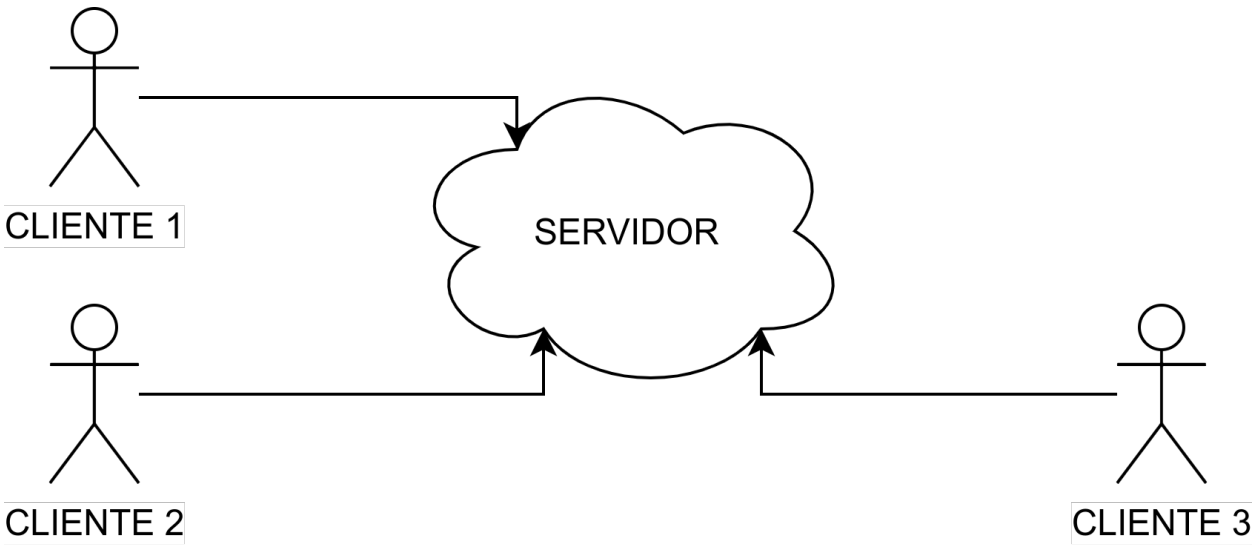

Figura 1: Esquema de servicio cliente-servidor concurrente

Modelo

Este software pretende que el cliente sea capaz de crear vectores de tamaño N, con un nombre, y poder administrarlos a partir de una aplicación. Estos vectores serán almacenados en el servidor con el que se comunica el cliente.

Estructura del código

Los archivos más importantes son cliente.c y servidor.c (bajo demanda) que son las que representan al cliente y al servidor y contienen los respectivos main. En cliente.c se ha añadido unos métodos para imprimir un mensaje con el resultado de las operaciones. Además, se incluyen sus respectivas cabeceras en los archivos cliente.h y servidor.h. El desarrollo de las peticiones del cliente de las operaciones init, get, set y destroy se realizan en los ficheros array.c en la parte cliente y imple.c en la parte servidor. Para la comunicación entre cliente y servidor hemos creado el fichero comm.h que contiene la estructuras de la petición y la respuesta.

Compilación

Para poder compilar el ejercicio se puede hacer de forma sencilla con *make all*, esto generará todos los ejecutables.

Nota: antes de compilar el cliente manualmente (make cliente/cliente2), será necesario haber generado la biblioteca dinámica haciendo en primer lugar *make array*.

Cliente

La función del cliente en el sistema es la de utilizar la interfaz gráfica, en este caso el main del archivo *cliente.c* para administrar los datos que el cliente quiere utilizar, es decir, los vectores mencionados anteriormente. Tendrá la capacidades de realizar varias opciones.

- **Crear un vector**: Usará la función *init* para crear el vector especificando el nombre y tamaño a través de los argumentos.
- **Modificar un vector**: Podrá modificar elementos de un vector de tipo número a través de la función *set* indicando la posición en el vector, el nombre del vector y el nuevo valor.
- **Obtener de un vector**: Podrá obtener elementos de tipo número entero del vector a través de la función *get* indicando la posición en el vector, el nombre del vector y la variable valor donde se guardará el dato.
- **Eliminar un vector**: El cliente podrá eliminar con la función *destroy* el vector que desee indicando su nombre.

Biblioteca Dinámica

En la biblioteca dinámica se han implementado las funciones que se comunicarán con el sistema servidor. La inclusión de estructuras de este estilo facilita la posible mejora de los sistemas de comunicación sin afectar a la interfaz del usuario, incluso pudiendo tener varias interfaces.

Servidor

El servidor será el encargado de recibir las peticiones para que, con ayuda de hilos bajo demanda, se puedan ir ejecutando las acciones que requiere el cliente para administrar los vectores. El servidor debe estar configurado

de tal manera que se evite que se pierdan peticiones debido a condiciones de carrera y, para ello, se emplearán herramientas de POSIX. Hemos optado por implementarlo mediante un cerrojo y una variable condicional que hacen que no se pueda atender a una nueva petición si no se atendió a la anterior. El término "atender" no se refiere a procesar la petición, simplemente al hecho de copiar la petición de entrada en una variable local y evitar que se pierda.

Modelo de datos

Para administrar la estructura de datos que almacena los vectores se ha decidido establecer una lista enlazada implementada a través de un *struct* del lenguaje de programación **C**. Cada elemento de la lista está formado por un vector de enteros de tamaño N, una cadena de caracteres que representa el nombre del vector y un puntero al siguiente elemento. Además, las peticiones se han podido realizar mediante *structs* debido a la tecnología de comunicación que son las colas de mensajes del sistema UNIX.

Implementación

Las funciones que manejan la lista enlazada se encuentran en un archivo distinto al del propio servidor para facilitar los cambios en el sistema, dando prioridad a la modularización del sistema. Las funciones que realizan los clientes tienen su correspondencia en el servidor. Todas ellas devuelven los valores indicados en el enunciado.

- **Crear un vector**: La función *INIT* será la encargada de crear un nuevo elemento en la lista. Para ello, se debe reservar en memoria el tamaño necesario para almacenar un vector de N enteros, la cadena de caracteres (nombre del vector) de tamaño **MAX** (que se establece en 100 caracteres) y el puntero. Además, se deberá enlazar al final de la lista con el elemento correspondiente.
- **Modificar un vector**: La función *SET* se encargará de encontrar el vector en la lista a través del nombre y, posteriormente, se encargará de cambiar el valor que se almacenaba en la posición indicada por el valor que se desea introducir.
- **Obtener de un vector**: La función *GET* se encargará de encontrar el vector en la lista a través del nombre y, posteriormente, guardará el valor que se almacenaba en la posición indicada en la variable que se pasa como parámetro.
- **Eliminar un vector**: La función *DESTROY* se encarga de eliminar el espacio de memoria que ocupaba el vector en la lista y, además, enlazar correctamente los elementos de lista para evitar que queden elementos sin apuntar.

Pruebas

Para verificar el correcto funcionamiento de las operaciones en el servidor hemos creado el método *show* que muestra la lista con el contenido de todos los vectores para ver las modificaciones. Además, hemos creado unos métodos en el cliente que muestra el resultado de los métodos: ErrorMsgInit, ErrorMsgSet, ErrorMsgGet y ErrorMsgDestroy que devuelven "OK" si ha sido correcta o "Something went wrong" si no lo es. A continuación, verificamos el correcto funcionamiento de las operaciones de manera secuencial.

- **init**
  Para probar el funcionamiento del método init primero introducimos un vector llamado "vector1" de tamaño 1000. La salida en este caso es: 1. Después, introducimos otro vector con el mismo nombre y mismo tamaño. La salida es: 0 y no se enlaza en la lista. Finalmente, introducimos otro vector del mismo nombre

y tamaño distinto y la salida es -1. Añadimos otro vector de distinto nombre y tamaño y se añade a la lista sin problemas.

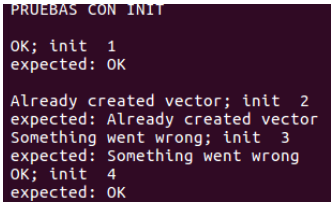

Figura 2: Pruebas de función init

También se ha probado a crear un vector con un nombre y una dimensión desde un cliente e intentar volver a crearlo igual desde otro cliente cuando el primero ya ha terminado. Se comprueba que el primero lo crea correctamente, pero al segundo cliente le da un mensaje de que el vector ya existe.

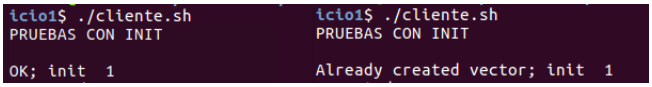

Figura 3: Pruebas de función init

- **set**
  Modificamos el valor de un vector que está en la lista y en una posición correcta, la salida es 0. Después, modificamos el valor de un vector cuyo nombre no se encuentra en la lista, la salida es -1 . Finalmente, intentamos modificar el valor de una posición de un vector de la lista fuera de rango (de tamaño mayor del vector), la salida es -1.

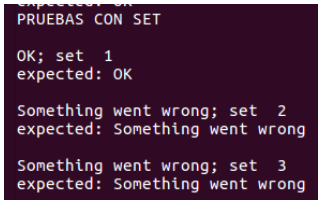

Figura 4: Pruebas de función set

- **get**
  Obtenemos el valor de un vector que está en la lista y en una posición correcta, la salida es 0. Después, obtenemos el valor de un vector cuyo nombre no se encuentra en la lista, la salida es -1 . Finalmente, intentamos obtener el valor de una posición de un vector de la lista fuera de rango (de tamaño mayor del vector), la salida es -1.

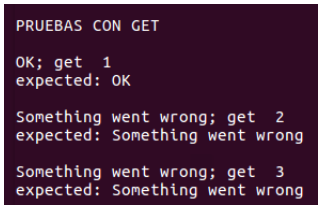

Figura 5: Pruebas de función get

- **destroy**
  Eliminamos un vector que está en la lista, la salida es 1. Después, eliminamos un vector cuyo nombre no se encuentra en la lista, la salida es -1 .

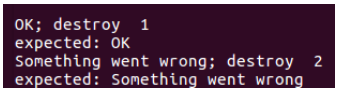

Figura 6: Pruebas de función destroy

Una vez probado la ejecución con un cliente de manera secuencial, mostramos algunos ejemplos en la siguiente imagen para comprobar el comportamiento con varios clientes de manera concurrente. Para ello, hemos utilizado sleep() para que uno de ellos se duerma en mitad de la ejecución. En la prueba que mostramos como ejemplo el primer cliente crea un vector vector1 de tamaño 1000, muestra el valor de la posición 80 con el get (resultado 0 al estar recién creado) y se duerme. Por otra parte, el segundo cliente crea el mismo vector (el resultado es que ya existe) y modifica los datos del vector con un set en un bucle de 1000 llamadas. A continuación, el primer cliente se despierta y muestra de nuevo con un get el valor de la posición 80 que modificó el segundo cliente. Por último, el segundo cliente borra otro vector vector2 que el primero creó antes de dormir. Comprobamos que lo hace correctamente.

Figura 7: Pruebas concurrencia

Mostramos otro ejemplo con el fin de comprobar que el servidor es capaz de borrar a un vector vector2 situado entre dos vectores vector1 y vector3. Para ello, hacemos que el primer cliente cree el vector1, se duerma, el cliente2 crea el vector2, se duerma y, finalmente, el cliente primero crea el vector3 y destruya el vector2. Por último, el cliente2 intenta borra el vector2 y ya no existe.

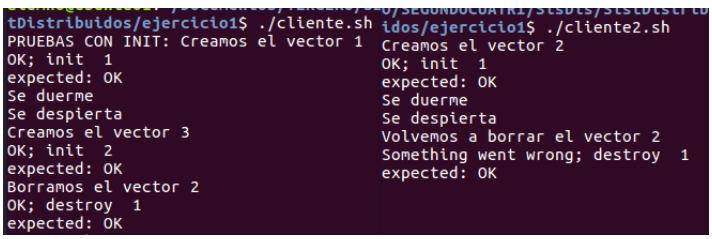

Figura 8: Pruebas concurrencia

Conclusiones

Esta práctica nos ha permitido aprender la estructura básica de un sistema de tipo cliente-servidor, pudiendo descubrir algunas de sus ventajas y desventajas.

- **Ventajas**
  - Modularidad.
  - Menor cómputo en los dispositivos de los clientes.
- **Desventajas**
  - Mucha dependencia de los servidores.
  - Puede llegar a colapsar el sevicio si llegan demasiadas peticiones.

Por otro lado, hemos tenido ciertas dificultades a la hora de realizar la concurrencia puesto que en la implementación con hilos *bajo demanda* se lanzan muchos más hilos llegando incluso a poder empeorar la calidad del servicio, mientras que con una implementación de *pool de hilos* el sistema es mucho más eficaz. Decidimos empezar la tarea con *hilos bajo demanda* y resultó menos complejo, tras completar esta versión pudimos realizar la versión con el *pool de hilos*. Finalmente la versión de pool de hilos dejó de funcionar y decidididimos no continuar la depuración de esta implementación.