Sistemas distribuidos

Ejercicio evaluable 2: sockets de mensajes

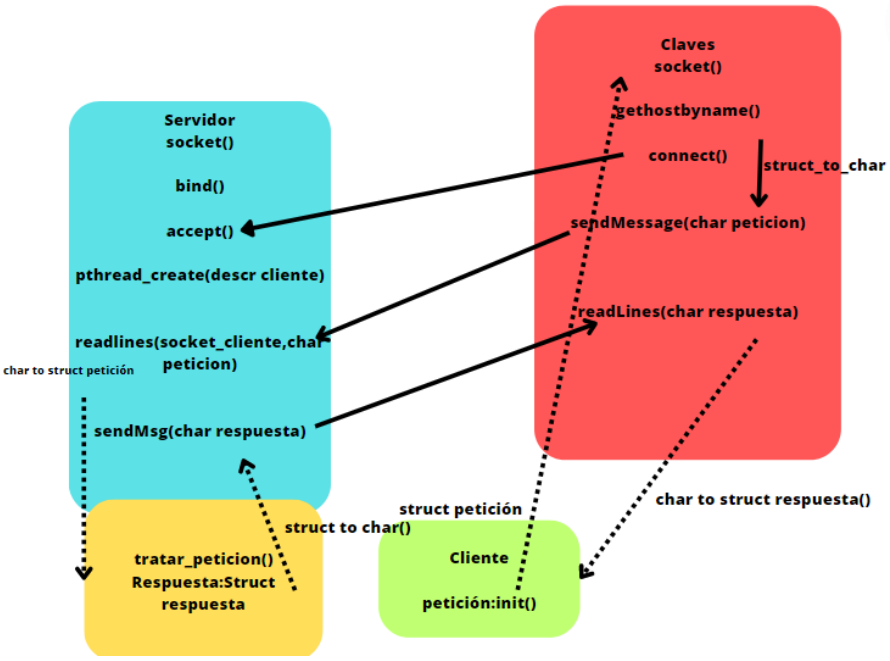
Grupo 82, grupo N

Elisa Utrilla Arroyo (100451242)

Luis González Perea (100451116)

Diseño

En general,sigue el mismo diseño que el ejercicio 1,sin embargo la forma de enviar datos con socket varía con la forma de enviar con colas. Aquí se ilustra gráficamente como funciona



1. El cliente manda una petición a claves a través de la llamada a una función(como pasaba en el anterior ejercicio)
2. Claves crea un socket cliente y **busca establecer conexión con el servidor.** Esto lo consigue gracias a las variables de entorno que introducimos a la hora de ejecutar tanto el ejecutable del cliente como el del servidor(La Ip del servidor es el localhost y el puerto de conexión 8080).Obtiene el "host" mediante **gethostbyname(IP_TUPLAS)** y,una vez establecido,configuramos el socket_in server_addr al puerto introducido anteriormente
3. Una vez hecho esto, y como forma de reutilizar el máximo código posible ,la petición creada anteriormente al llamar la función desde cliente la transformamos a char mediante la función **petición_to_char()** que convierte el struct definido en mensajes.h a un char de la forma **operación;clave;valor1;valor2;valor2** (todos los valores del struct separados por ;) y mandamos este char al socket servidor mediante **sendMessage.**
4. En el código del servidor,indicamos que esa dirección pueda ser usada incluso después de desconectar,inicializamos el servidor y lo ponemos a escuchar su socket.En este momento cuando ya se ha conectado cliente y servidor,creamos los hilos **bajo demanda según las peticiones que reciba el socket mediante accept.** Todos estos hilos serán detached y accederán al tratamiento de la petición uno a uno mediante mutex.Las peticiones se recibirán en formato char mediante **readLines(de lines.c)** serán pasadas de char a struct petición mediante la función **char_to_petición()**

5. El tratamiento de peticiones es exactamente igual que en el proyecto anterior,usando **un sistema de ficheros** para almacenar las tuplas.
6. Después del tratamiento de la petición,el struct respuesta que genera pasa a forma de char y,de nuevo,se envía al socket cliente mediante sendMessage con la estructura antes dada(todos los elementos del struct separados entre comas).

El código de "cliente.c" utiliza la biblioteca compartida "claves" para realizar peticiones al servidor. Desde el main llama a todas las funciones a modo de prueba.

El código de "claves.c" se comunica con un servidor a través de sockets. Las peticiones que se pueden realizar:

1. init(): Inicializa el servidor.
2. set_value(): Establece un valor asociado a una clave determinada en el servidor.
3. get_value(): Obtiene el valor asociado a una clave determinada del servidor.
4. modify_value(): Modifica el valor asociado a una clave determinada en el servidor.
5. exist(): Verifica si una clave determinada existe en el servidor.
6. delete_key(): Elimina una clave determinada del servidor.
7. copy_key(): Copia el valor asociado a una clave a otra clave en el servidor.

Forma de compilar

Para compilar nuestro proyecto y para mayor comocdidad,hemos utilizado el mismo makefile que en el trabajo anterior,pero añadiendo las funciones de **separar_mensaje.c**(pasa los struct a char y viceversa para el envío de peticiones) y **lines.c**(código proporcionado en Aula Global para enviar y recibir los mensajes del socket con control de errores):

Antes de nada, posicionamos qué elementos irán enlazados para qué ejecutables y cuáles objetos irán dentro de la librería dinámica,de esta forma:

```
# Archivos objeto
CLIENTE_OBJ_FILES = cliente.o
SERVIDOR_OBJ_FILES = servidor.o tratamiento_serv.o lines.o separar_mensaje.o
LIB_OBJ_FILES = claves.o
```

1. Creamos la **biblioteca dinámica libclaves.so** que será compartida con clientes.c y,en este caso, también añadimos lines.c a esta biblioteca dinámica(para el envío por sockets) Para ello primero compilamos claves.c como una librería dinámica

```
# Crear biblioteca compartida de objetos
libclaves.so: claves.o lines.o separar_mensaje.o
	$(CC) -shared -o $@ $^ $(LDFLAGS) $(LDLIBS)
```

2. Al crear el ejecutable de clientes,enlazamos el archivo .so a la compilación.

```
# Crear ejecutables
cliente: $(CLIENTE_OBJ_FILES) libclaves.so
	$(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@ $(LDLIBS)
```

3. Y luego compilamos el servidor.
4. En este caso para el servidor utilizamos 3 programas:servidor.c ,tratamiento_serv.c(tiene las funcionalidades que haría el servidor tras recibir el mensaje del cliente por su cola) **y lines.c**(que contiene las funciones de enviar y recibir mensajes por el socket)

```
servidor: $(SERVIDOR_OBJ_FILES)
	$(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@ $(LDLIBS)
```

5.
6. No hace falta meter en el makefile mensaje.h porque ya está incluido en todos los ficheros.

Y asi se nos generan los ficheros necesarios

Que se pueden ejecutar en dos terminales distintas usando ./cliente y ./servidor. En este caso,como usamos variables de entorno,ejecutamos los ejecutables de la siguiente manera:

SERVIDOR: PORT_TUPLAS=8080 ./servidor

CLIENTE: IP_TUPLAS=localhost PORT_TUPLAS=8080 ./cliente

Hay que destacar que en este caso,hay que exportar la ruta de donde estará la biblioteca dinámica,asi que previamente usaremos este comando en la terminal

export LD_LIBRARY_PATH=/home/elisa/Escritorio/distribuidos-ejercicio-2

El ejecutable clientes.c es solo un archivo de pruebas que ejecuta todas las posibles funciones del servidor en el que además comprobamos la concurrencia de este.En nuestro caso,los hilos se crean **bajo demanda en el servidor ,**es decir,no se crea otro hilo hasta que no se envía la petición. Podemos comprobar que funciona correctamente