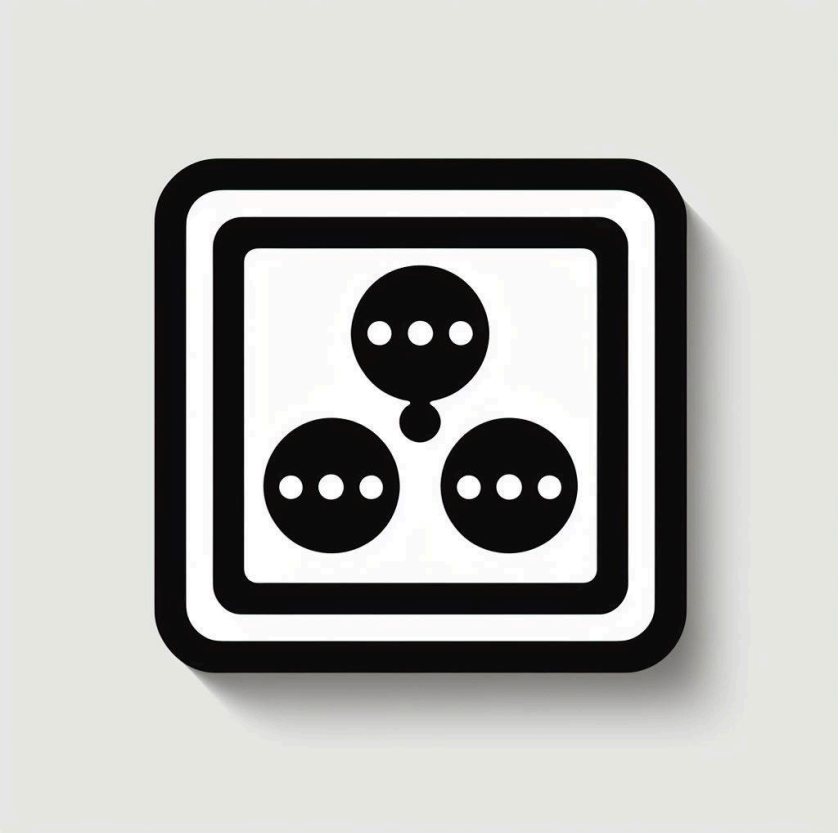
uc3m | Universidad Carlos III de Madrid

Ingeniería Informática

Año Académico (2023-2024)



Sistemas Distribuidos

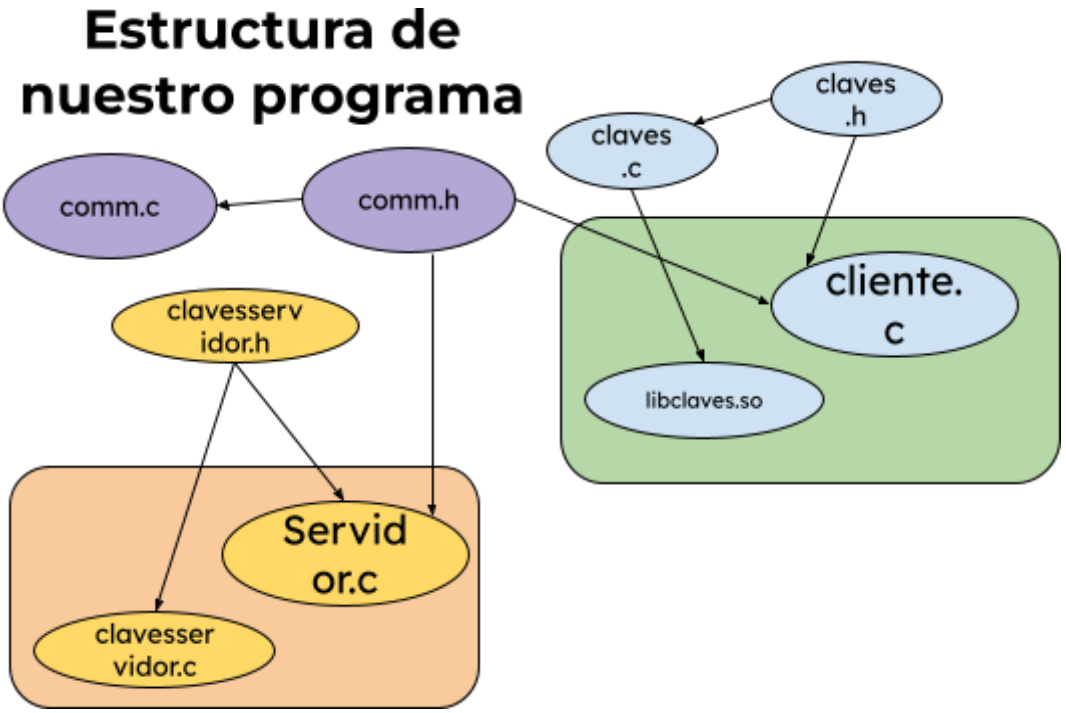
"P2"

Enrique Alcocer Díaz (100472118@alumnos.uc3m.es)

Iván Fernández Martín-Gil (100472263@alumnos.uc3m.es)

1. Estructura del código.

Para el desarrollo de esta práctica se han realizado 6 ficheros .c llamados clientes.c, claves.c clavescliente.c,comm.c y servidor.c.
Además de los ficheros mencionados anteriormente también se han añadido claves.h, claves cliente.h y comm.h que nos permite realizar las comunicaciones a través de sockets entre cliente y servidor.



Esta práctica parte como base de la anterior, en la que ya hicimos una explicación de las correspondientes funciones que hace *clavesservidor,* que es el que realmente realiza las funcionalidades que se piden, ya que *claves.c* únicamente realiza llamadas al servidor, no implementa las funciones.

La implementación de las funciones ya fue detallada en la entrega anterior del proyecto. En particular, en la explicación previa se abordó cómo `claves.c` funciona como un intermediario entre `clientes.c` y el procesamiento de los datos, verificando la validez de los datos de entrada y manejando la comunicación mediante colas de mensajes POSIX. Por lo tanto, esta descripción es una recapitulación de lo discutido previamente en trabajos anteriores.

Para adaptar el código de colas de mensaje a sockets se sustituyeron todas las creaciones de colas y su uso por la creación de sockets y su uso, esto lo realiza *claves.c* en cada una de sus funciones y *servidor.c.* Todo esto usó como base el archivo *comm.c* y *comm.h*, proporcionados en clase, cuya finalidad es realizar la implementación para la creación de sockets.

En *claves.c* se utilizó lo siguiente:

- La creación del socket del cliente se realizaba pasando como parámetros las variables de entorno de la ip y el puerto cuya implementación era de la siguiente manera:

```
sock = clientSocket(IP_TUPLAS, PORT_TUPLAS);
if (sock < 0) {
    perror("Client socket failed");
    exit(EXIT_FAILURE);
}
```

- El envío se basaba en *sendMessage* y *readLine*, la primera se utilizaba para mandar la cadena, la cuál es generada por una función estática auxiliar que recoge como parámetros los valores a usar y dónde guardarlos, *request* en este caso, y genera la cadena de carácteres la cuál tiene formato op_code/key/value1/N_value2/V_value2 (cada elemento de V_value2 está separado por guiones, 1.0-2.0-3.0). Respecto a *readLine*, espera una respuesta, y reserva el tamaño máximo del buffer establecido.

```
if (readLine(sock, (char *)&response, MAXSIZE) < 0) {
    perror("Receive response failed");
    exit(EXIT_FAILURE);
}
```

- Por último se cierra el socket.

Este es el proceso que se realiza en todas las funciones de *claves.c.*

Respecto al servidor, el envío y recepción de mensajes se realiza de la misma manera, llamando a serverSocket en este caso, pero el procedimiento es el mismo.

Por otra parte está *servidor.c* que es el encargado de recibir los mensajes y mediante threads crear un subproceso que trata el mensaje, de esta forma el servidor puede recibir varias peticiones a la vez de distintos clientes y tratarlas todas correctamente, por lo que se realiza un servidor concurrente que pueda administrar peticiones simultáneas.

Esto último se ha logrado recibiendo en sc la aceptación de comunicación que realiza el socket y tras ello creando el thread y enviando este valor como argumento, con esto logramos que cada thread sea capaz de recibir el mensaje correspondiente al valor guardado en sc y no colapsar entre distintas peticiones de clientes, a la vez que conseguimos un trato de mensajes concurrente.

```
while (1) {
    sc = serverAccept(sd);
    if (sc < 0) {
        printf("Error en serverAccept\n");
        continue;
    }

    if (pthread_create(&thid, &t_attr, (void *)tratar_mensaje, (void *)&sc) == 0) {
        pthread_mutex_lock(&mutex_mensaje);
        while (mensaje_no_copiado)
            pthread_cond_wait(&cond_mensaje, &mutex_mensaje);
        mensaje_no_copiado = true;
        pthread_mutex_unlock(&mutex_mensaje);
    }
}
```

Finalmente, es importante mencionar el archivo cliente.c, el cual es un código de prueba que describe varios clientes utilizando la llamada al sistema fork(). Esto simula múltiples clientes realizando solicitudes simultáneas al servidor, manejadas de manera concurrente mediante el uso de hilos (threads), mutex y variables de condición, exactamente igual al proyecto anterior.

En cuanto al Makefile, generamos los ejecutables de la misma manera que en el proyecto anterior, aunque ahora para la ejecución del proyecto realizamos *make run-servidor* y *make run-cliente*, que ya tienen los parámetros correspondientes a los puertos y la ip del cliente. Estos parámetros se han declarado a través de variables de entorno.

2. Conclusiones.

En esta práctica, comparando con la anterior, la cantidad de trabajo se ha visto reducida debido a que no teníamos que rehacer de nuevo las funciones, sino emplear sockets para modificar la comunicación entre cliente y servidor. También es notable decir que a pesar de esto, hemos revisado nuestro código anterior para corregir posibles fallos e implementar mejoras.

En cuanto a problemas encontrados durante la práctica podemos encontrar el uso de variables de entorno, el cual no entendíamos muy bien cómo implementar y no sabíamos dónde debía ser declarado, gracias a las resoluciones en clase solucionamos este problema. Otro error a comentar fue la utilización de *malloc* para la creación de la cadena del mensaje, es un error ya que teníamos una cadena de caracteres de 256 y el *malloc* en este caso no tenía ningún tipo de utilidad, además que nos suponía problemas a la hora de realizar el *free.*

Además se ha optado por permitir únicamente strings de una sola palabra para *value1* como por ejemplo “Mensaje” con el objetivo de facilitar el procesado de los valores y evitar errores no deseados en el código.

Los sockets nos ha parecido una manera bastante cómoda de trabajar en este tipo de aplicaciones distribuidas, y creemos que es una buena práctica para acercarnos a la realidad de la situación a la hora de implementar este tipo de sistemas.