Memoria Ejercicio Evaluable 2
Sistemas Distribuidos

Esteban Gómez Buitrago
100485446
Nicolás Alejandro Cuesta García
100495966

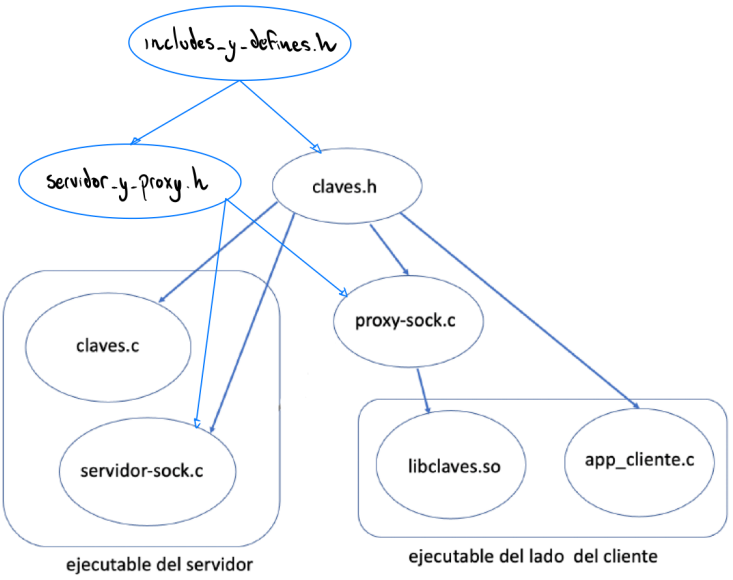
Marzo 2025

Universidad Carlos III de Madrid
Leganés, Madrid

Diseño de las Funciones

Archivos .h adicionales

A lo largo de la realización de este ejercicio, nos dimos cuenta de que tanto *servidor-sock.c* como *proxy-sock.c* requerían de funciones para mandar y recibir mensajes, convertir strings a números, leer números de un socket… Por ello, definimos un nuevo archivo header *servidor_y_proxy.h*. Como, a su vez, las funciones implementadas en *servidor_y_proxy.h* requerían de otros archivos header (“includes”) y de ciertas constantes (“defines”) definidas en *claves.h*, decidimos extraerlos a otro archivo header *includes_y_defines.h*, incluido en *servidor_y_proxy.h* y *claves.h*. Aquí adjuntamos un esquema para que quede claro:



Servidor-sock.c

Si bien este *servidor-sock.c* y el anterior *servidor-mq.c* presentan una estructura similar, hay que señalar que el reemplazo de las colas de mensaje por sockets ha conllevado un muy importante rediseño del código. A continuación, nos centraremos en cada una de las áreas que han sido modificadas:

- Configuración del socket: A grandes rasgos, se basa en la ejecución de las funciones *socket()*, *setsockopt()*, *bind()* y *listen()* en ese orden.
- Creación de hilos: Es fundadamente un bucle infinito en donde el servidor se prepara para aceptar conexiones entrantes al socket. Una vez una acepta una, crea un hilo para que gestione la petición. Aquí hay que hacer mención al uso de un mutex, una variable condicional y una variable global de sincronización para garantizar que cada hilo se conecte al socket adecuado y no a uno que no le corresponda.
- Por otro lado, dentro de la función *sendResponse* que ejecuta cada hilo, también encontramos una serie de líneas dedicadas a la sincronización. El grueso de la función será descrito en detalle en el siguiente apartado *Diseño del Protocolo de Aplicación.*

Proxy-sock.c

De forma análoga a la función main dentro de *servidor-sock.c*, el proxy contiene la función *connectToServer*. Esta intenta conectar un socket TCP a un servidor usando la IP y el puerto definidos en las variables de entorno. Para ello, hace uso de las funciones *socket* y *connect*.

Luego encontramos a la función *sendRequestToServer*, que una vez ha hecho uso de *connectToServer*, se encarga de mandar la petición al servidor. Esto será descrito más en detalle en el siguiente apartado *Diseño del Protocolo de Aplicación*.

Más abajo se encuentran las llamadas a sendRequestToServer que hacen cada una de las funciones que llama el cliente. La función *getValue* será discutida en el siguiente apartado *Diseño del Protocolo de Aplicación*.

Diseño del Protocolo de Aplicación

Empecemos hablando de la función *sendRequestToServer* del proxy, pues es la primera que se ejecuta. En esencia, manda los parámetros que recibe en orden (action, key, value1, N_value2, V_value2, value3) por medio de la función *sendMessage* y luego se bloquea esperando por la respuesta del servidor (función *recvMessage*).

```C/C++
//La función es solo accesible dentro este archivo (solo para el cliente).
static int sendRequestToServer( //...
                                int * key) {
    //...
    // Ejemplo de mensaje enviado
    if (key != NULL){
        snprintf(buffer, sizeof(buffer), "%d", *key);
        if ((ret= sendMessage(sc, buffer, strlen(buffer) + 1)) != 0) //ERROR
    }
    // ...
    // Recibir la respuesta
    if ((ret = recvMessage(sc, buffer, 2)) != 0) // ERROR
    if ((strtol_handling(buffer, &ret)) != 0) // ERROR
    // ...
}
```

En consecuencia, la función *SendResponse* del servidor hace las acciones opuestas en el mismo orden. Dependiendo de la acción recibida, se prepara para recibir unos parámetros u otros. Por ejemplo, la función *destroy* no requiere de ningún parámetro, mientras que set_value necesita todos los parámetros. Una vez ha procesado la petición, manda un mensaje de vuelta al proxy.

```C/C++
void * SendResponse(void * sc){
    // ...
    // Recibir la acción a realizar
    if ((ret = recvMessage(s_local, &action, 1)) != 0) // ERROR
    // Procesar la solicitud
    switch (action) {
        case DESTROY:
            ret = destroy();
        //...
        case SET_VALUE:
            // Recibir la key
            // Recibir value1
            // Recibir número de doubles contenidos en value2
            // Ejemplo: Recibir double a double de value2
            // ...
            for (int i = 0; i < N_value2; i++){
                if ((ret = readLine(s_local, buffer, 256)) < 0) // ERROR
                if ((ret = strtod_handling(buffer, &V_value2[i])) != 0)
                                                                // ERROR
            }
            // Recibir value3
            // Llamada a set_value, en claves.c
            ret = set_value(key, value1, N_value2, V_value2, value3);
            // ...
    }
    snprintf(buffer, sizeof(buffer), "%d", ret);
    if ((ret = sendMessage(s_local, buffer, strlen(buffer) + 1)) != 0)
       // ...
}
```

Adicionalmente, debemos comentar un caso especial –la función *getValue* del proxy. Esta sigue una lógica separada al resto debido a que requiere de un protocolo de mensajes muy distinto al resto: envía la key, recibe un mensaje acerca de si existe dicha key y, en caso afirmativo, lee los valores uno a uno y los guarda en las variables que recibió como argumentos. Esto último es importante, *getValue* no devuelve un struct con los valores, sino que debe guardarlos en la memoria que ya reservó el cliente antes de llamar a la función.

```C/C++
int get_value(int key, char *value1, int *N_value2, double *V_value2, struct
Coord *value3) {
    //...
    // Conectarse al servidor
    // Enviar acción a realizar
    // Enviar key
    snprintf(buffer, sizeof(buffer), "%d", key);
```

```
    if ((ret = sendMessage(s, buffer, strlen(buffer) + 1)) != 0) // ERROR
    int result;
    // Leer el resultado
    if ((ret = read_num_from_socket(s, buffer, &result)) != 0) // ERROR
    ret = result;
    // Si es correcto, obtener los datos
    if (result == 0) {
        // Ejemplo: Leer value1
        if ((ret = readLine(s, value1, 256)) < 0) // ERROR
        // Leer resto de valores
    }
    // ...
}
```

De nuevo, el servidor contiene una sección dedicada a *getValue*:

```
C/C++
     //...
     ret = get_value(key, value1, &N_value2, V_value2, &value3);
     int ret_copy = ret;
     // Enviar el resultado
     snprintf(buffer, sizeof(buffer), "%d", ret);
     if ((ret = sendMessage(s_local, buffer, strlen(buffer) + 1)) != 0)
                                                              // ERROR
     if (ret_copy == 0){
           // Enviar value1
           if ((ret = sendMessage(s_local, value1, strlen(value1) + 1)) != 0)
           // Enviar resto de valores
```

Compilación y Generación de Ejecutable

El archivo *Makefile* de este ejercicio es prácticamente idéntico al anterior. La única diferencia significativa es la adición de los nuevos archivos header:

```
Unset
$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c $(HEADER_SRC) $(INC_DEF_SRC) $(SERV_PROX_SRC)
    @mkdir -p $(OBJ_DIR)
    $(CC) $(CFLAGS) -c $< -o $@
```

iniciar_clientes.sh

De forma adicional, este programa ejecuta los clientes en una misma sesión de la terminal.