abr 01, 19 17:07 **server_tamplate.h** Page 1/1

```h
#ifndef SERVER_TEMPLATE_H
#define SERVER_TEMPLATE_H

struct server_template {
    char* text;
    char* to_replace;
};


bool server_template_create(struct server_template *self, char* filename);
char* server_template_cat(struct server_template *self, char* replacement);
void server_template_destoy(struct server_template *self);

#endif
```

abr 01, 19 17:07 **server_tamplate.c** Page 1/1

```c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include "server_tamplate.h"

#define TO_REPLACE "{{datos}}"
#define SIZE_TO_REPLACE 9
#define MAX_LEN_REPLY 2000

bool server_template_create(struct server_template *self, char* filename) {
    self→text = malloc(MAX_LEN_REPLY);
    if ( ¬self→text ) return false;
    FILE* file = fopen(filename, "r");
    if (¬file) {
        free(self→text);
      return false;
    }

    size_t i = 0;
    while (¬feof(file) ∧ i < MAX_LEN_REPLY) {
        self→text[i] = (char) fgetc(file);
        i++;
    }
    self→text[i−1] = '\0';
    self→to_replace = strstr(self→text, TO_REPLACE);
    fclose(file);
    return true;
}

char* server_template_cat(struct server_template *self, char* replacement) {
    char* reply = malloc(MAX_LEN_REPLY);
    if ( ¬reply ) return NULL;
    char aux[MAX_LEN_REPLY];
    snprintf(aux, MAX_LEN_REPLY, "%s", self→text);

    char* to_replace = strstr(aux, TO_REPLACE);

    snprintf(to_replace, strlen(to_replace) − 1, "%s", replacement);
    snprintf(reply, MAX_LEN_REPLY − 1, "%s", aux);

    int len = strlen(&to_replace[SIZE_TO_REPLACE]);
    snprintf(&reply[strlen(reply)], len,"%s", &to_replace[SIZE_TO_REPLACE]);
    return reply;
}

void server_template_destoy(struct server_template *self) {
    free(self→text);
}
```

abr 01, 19 17:07 **server_socket.h** Page 1/1

```c
#ifndef SERVER_SOCKET_H
#define SERVER_SOCKET_H
#include <stdlib.h>
#include <stdbool.h>

struct server_socket {
    char* port;
    int skt;
    int current_peerskt;
};


/*
Crea e incializa el socket definiendo la familia, el tipo de socket y el
protocolo para poder conectarse al cliente por medio del port y host indicados
 */
bool server_socket_create(struct server_socket *self, char* _port);

/*
Almacena los parametros necesarios para la incializaciÃ³n del socket.
 */
bool server_socket_start(struct server_socket *self);


void server_socket_destroy(struct server_socket *self);
int server_socket_accept_client(struct server_socket *self);

char* server_socket_receive_message(struct server_socket *self);
int server_socket_send_message(struct server_socket *self, char* buf, int size);

/*
Desactiva las operaciones de envÃ-o y recepciÃ³n para el cliente y para si mismo
 */
void server_socket_disable_client(struct server_socket *self);

#endif
```

abr 01, 19 17:07 **server_socket.c** Page 1/3

```c
#define _POSIX_C_SOURCE 200112L
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <errno.h>
#include <stdbool.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include "server_socket.h"

#define LEN_PORT 6
#define MAX_WAITING_CLIENTS 20
#define MAX_LEN_BUF 2000


bool server_socket_create(struct server_socket *self, char* _port) {
    self→port = malloc(LEN_PORT);
    if ( ¬self→port ) return false;
    snprintf(self→port, LEN_PORT , "%s", _port);
    self→current_peerskt = 0;
    return true;
}

bool server_socket_start(struct server_socket *self) {
    int s = 0;

    struct addrinfo hints;
    struct addrinfo *ptr;

    int skt = 0;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    s = getaddrinfo(NULL, self→port, &hints, &ptr);

    if (s ≠ 0) {
        return false;
    }

    skt = socket(ptr→ai_family, ptr→ai_socktype, ptr→ai_protocol);

    if (skt ≡ −1) {
        freeaddrinfo(ptr);
        return false;
    }

    int val = 1;
    s = setsockopt(skt, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
    if (s ≡ −1) {
        close(skt);
        freeaddrinfo(ptr);
        return false;
    }

    s = bind(skt, ptr→ai_addr, ptr→ai_addrlen);
    if (s ≡ −1) {
        close(skt);
        freeaddrinfo(ptr);
        return false;
```

```
67      }
68
69      freeaddrinfo(ptr);
70      s = listen(skt, MAX_WAITING_CLIENTS);
71      if (s ≡ -1) {
72          close(skt);
73          return false;
74      }
75      self→skt = skt;
76      return true;
77  }
78
79  //-1 si falla
80  int server_socket_accept_client(struct server_socket *self){
81      int peerskt = accept(self→skt, NULL, NULL);
82      self→current_peerskt = peerskt;
83      return peerskt;
84  }
85
86  void server_socket_destroy(struct server_socket *self) {
87      free(self→port);
88      shutdown(self→skt, SHUT_RDWR);
89      close(self→skt);
90  }
91
92  char* server_socket_receive_message(struct server_socket *self) {
93      char* buf = malloc(MAX_LEN_BUF);
94      if ( ¬buf ) return NULL;
95      memset(buf, 0, MAX_LEN_BUF);
96      int received = 0;
97      int s = 0;
98      bool is_the_socket_valid = true;
99      while (received < MAX_LEN_BUF ∧ is_the_socket_valid) {
100         s = recv(self→current_peerskt, buf + received, \
101                  MAX_LEN_BUF - received, MSG_NOSIGNAL);
102         if (s ≡ 0) {
103             is_the_socket_valid = false;
104         } else if (s < 0) {
105             is_the_socket_valid = false;
106         } else {
107             received += s;
108         }
109     }
110     return buf;
111 }
112
113 int server_socket_send_message(struct server_socket *self, char* buf, \
114                                int size) {
115     int sent = 0;
116     int s = 0;
117     bool is_the_socket_valid = true;
118
119     while (sent < size ∧ is_the_socket_valid) {
120         s = send(self→current_peerskt, &buf[sent], size-sent, MSG_NOSIGNAL);
121         if (s ≤ 0) {
122             return -1;
123         } else {
124             sent += s;
125         }
126     }
127     return sent;
128 }
129
130 void server_socket_disable_client(struct server_socket *self) {
131         shutdown(self→current_peerskt, SHUT_RDWR);
132         close(self→current_peerskt);
```



```
133 }
```

abr 01, 19 17:07 **server_sensor.h** Page 1/1

```
#ifndef SERVER_SENSOR_H
#define SERVER_SENSOR_H
#include <stdbool.h>

struct server_sensor {
    FILE* file;
};

bool server_sensor_create(struct server_sensor* self, char* filename);
/*
Se ocupa de leer de del archivo binario file una temperatura
almacenada en 16 bits y formato big-endian.
La misma la interpreta de la siguiente forma: Temperatura = (datos - 2000)/100
*/
char* server_sensor_read(struct server_sensor* self);

//comunica si quedan o no temperaturas por leer.
bool does_the_sensor_still_have_temperatures(struct server_sensor* self);

void server_sensor_destroy(struct server_sensor* self);

#endif

```

abr 01, 19 17:07 **server_sensor.c** Page 1/1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include <arpa/inet.h>

#include "server_sensor.h"

#define SIZE_OF_TEMPERATURE 2
#define MAX_LEN_TEMPERATURE_MESSAGE 200
#define SIZE_TO_REPLACE 9


bool server_sensor_create(struct server_sensor* self, char* filename) {
    FILE* file = fopen(filename, "r+b");
    if (¬file) {
      return false;
    }
    self→file = file;
    return true;
}


char* server_sensor_read(struct server_sensor* self) {
    unsigned short int read;
    size_t len = fread((void*)&read, SIZE_OF_TEMPERATURE, 1, self→file);
    if (¬len) return NULL;
    float temperature = (float) ntohs(read);
    temperature = (temperature - 2000) / 100;
    char* message = malloc(MAX_LEN_TEMPERATURE_MESSAGE);
    if ( ¬message ) return NULL;
    snprintf(message, SIZE_TO_REPLACE,"%.2f", temperature);
    return message;
}

bool does_the_sensor_still_have_temperatures(struct server_sensor* self) {
    return ¬feof(self→file);
}


void server_sensor_destroy(struct server_sensor* self) {
    fclose(self→file);
}
```

abr 01, 19 17:07 **server_request_processor.h** Page 1/1

```c
#ifndef SERVER_REQUEST_PROCESSOR_H
#define SERVER_REQUEST_PROCESSOR_H
#include <stdbool.h>


struct server_req_proc {
    char* request;
    bool is_method_resource_valid;
};


bool req_proc_create(struct server_req_proc* self, char* request);
void req_proc_destroy(struct server_req_proc* self);

/*
Verifica que el mÃ©todo utilizado sea del tipo "GET"
y el recurso sea "/sensor".

Si el mÃ©todo no es "GET", la respuesta serÃ¡ un
error de tipo "400 Bad request"
Si el recurso no es "/sensor", la respuesta serÃ¡ un error
de tipo "404 Not found".
Si el mÃ©todo y recurso son vÃ¡lidos, la respuesta es
de tipo "200 OK".
*/
char* req_porc_method_resource(struct server_req_proc* self);
bool req_porc_is_method_resource_valid(struct server_req_proc* self);

/*
Busca y devuelve el valor del user-agent en el
request con formato clave:valor.
*/
char* req_porc_user_agent(struct server_req_proc* self);

#endif
```

abr 01, 19 17:07 **server_request_processor.c** Page 1/2

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <errno.h>
#include <stdbool.h>
#include "server_request_processor.h"

#define METHOD_OFFSET 0
#define METHOD_ERROR_MESSAGE "HTTP/1.1 400 Bad request\n"
#define LEN_METHOD 3
#define CORRECT_METHOD "GET"

#define RESOURCE_OFFSET 4
#define RESOURCE_ERROR_MESSAGE "HTTP/1.1 404 Not found\n"

#define LEN_RESOURCE 7
#define CORRECT_RESOURCE "/sensor"

#define MAX_LEN_REQUEST 2000

#define METHOD_ERROR_MESSAGE "HTTP/1.1 400 Bad request\n"
#define RESOURCE_ERROR_MESSAGE "HTTP/1.1 404 Not found\n"
#define METHRES_SUCCESS_MESSAGE "HTTP/1.1 200 OK\n\n"
#define MAX_LEN_MESSAGE 26

#define USER_AGENT_KEY "User-Agent:"
#define USER_AGENT_VAL_OFFSET 12
#define END_USER_AGENT_VAL "\n"
#define MAX_LEN_USER_AGENT_VALUE 200




bool req_proc_create(struct server_req_proc* self, char* request) {
    self→request = malloc(MAX_LEN_REQUEST);
    if ( ¬self→request ) return false;
    snprintf(self→request, MAX_LEN_REQUEST, "%s", request);
    self→is_method_resource_valid = false;
    return true;
}

void req_proc_destroy(struct server_req_proc* self) {
    free(self→request);
}

//Verifica que el comando str se encuentre en request
//De ser asÃ- devuelve 0
//en caso contrariodevuelve 1.
int str_check(const char* request, size_t len,const char* str) {//, char* err) {
    for (int i = 0; i < len; ++i) {
      if (request[i] ≠ str[i]) {
          return 1;
      }
    }
    return 0;
}

char* req_porc_method_resource(struct server_req_proc* self) {
    char* answer = malloc(MAX_LEN_MESSAGE);
    if ( ¬answer ) return NULL;
    enum error {METHRES_SUCCESS, METHOD_ERROR, RESOURCE_ERROR};
    char* position = self→request + METHOD_OFFSET;
    if (str_check(position, LEN_METHOD, CORRECT_METHOD)) {
      snprintf(answer, MAX_LEN_MESSAGE, "%s", METHOD_ERROR_MESSAGE);
        self→is_method_resource_valid = false; //no es necesario,
```

abr 01, 19 17:07 **server_request_processor.c** Page 2/2

```
67          return answer;                          //pero si llegase a cambiar
68      }                                           //implementacion podria serlo
69      position = self→request + RESOURCE_OFFSET;
70      if (str_check(position, LEN_RESOURCE, CORRECT_RESOURCE)) {
71          snprintf(answer, MAX_LEN_MESSAGE, "%s", RESOURCE_ERROR_MESSAGE);
72          self→is_method_resource_valid = false;
73        return answer;
74      }
75      snprintf(answer, MAX_LEN_MESSAGE, "%s", METHRES_SUCCESS_MESSAGE);
76      self→is_method_resource_valid = true;
77      return answer;
78  }
79
80
81  bool req_porc_is_method_resource_valid(struct server_req_proc* self) {
82      return self→is_method_resource_valid;
83  }
84
85
86  char* req_porc_user_agent(struct server_req_proc* self) {
87      char* key_start = strstr(self→request, USER_AGENT_KEY);
88      if (¬key_start) {
89          return NULL;
90      }
91      char* value_start = key_start + USER_AGENT_VAL_OFFSET;
92      size_t len_value = 0;
93      while (value_start[len_value] ∧ \
94             strcmp(value_start + len_value, END_USER_AGENT_VAL)) {
95                 len_value++;
96      }
97      char* value = malloc(MAX_LEN_USER_AGENT_VALUE);
98      if (¬value) return NULL;
99      snprintf(value, MAX_LEN_USER_AGENT_VALUE, "%s", value_start);
100     char* value_end = strstr(value, "\n");
101     if (value_end) { //sino eof => strstr deja \n
102         *value_end = '\0';
103     }
104     return value;
105 }
```

abr 01, 19 17:07 **server_list.h** Page 1/1

```
1   #ifndef SERVER_LIST_H
2   #define SERVER_LIST_H
3   #include <stdbool.h>
4
5   struct nodo;
6
7   struct List {
8       struct nodo* first;
9       struct nodo* last;
10      size_t len;
11  };
12
13  struct List;
14  void list_create(struct List *self);
15  void list_destroy(struct List *self);
16
17  /*
18  Dada una lista y una clave se agrega la clave a la lista
19  y se inicializa su valor.
20  En caso de ya existir la clave solamente se aumenta en uno su valor.
21  */
22  bool list_insert(struct List *self, char* _key);
23
24  /*
25  Imprime todos los datos almacenados de la forma
26  * <clave1>: <valor1>
27  * <clave2>: <valor2>
28  */
29  void list_print(struct List *self);
30
31  #endif
32
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "server_list.h"

struct nodo {
    char* key;
    long int value;
    struct nodo* next;
} typedef nodo_t;



bool nodo_create(nodo_t* self, char* _key) {
    self→value = 1;
    self→next = NULL;
    size_t len_key = strlen(_key) + 1;
    self→key = malloc(len_key);
    if ( ¬self→key ) return false;
    snprintf(self→key, len_key, "%s", _key);
    return true;
}

void list_create(struct List *self) {
    self→first = NULL;
    self→last = NULL;
    //self->len = 0;
}

void list_destroy(struct List *self) {
    nodo_t* current = self→first;
    nodo_t* aux;
    while (current) {
  aux = current→next;
       free(current→key);
  free(current);
  current = aux;
    }
}

bool list_insert(struct List *self, char* _key) {
    bool status = true;
    nodo_t* current = self→first;
    while (current) {
      if ( ¬strcmp(current→key, _key) ){
          current→value++;
          return true;
      }
      current = current→next;
    }

    nodo_t* nodo = malloc(sizeof(nodo_t));
    if (¬nodo) return false;
    status = nodo_create(nodo, _key);
    if ( ¬status ) {
        free(nodo);
        return false;
    }
    if (¬self→first) {
      self→first = nodo;
      self→last = nodo;
      return true;
    }
    self→last→next = nodo;
    self→last = nodo;
    return true;
}

void list_print(struct List *self){
    //printf("# Estadisticas de visitantes\n");
    nodo_t* current = self→first;
    while (current) {
        printf("\n* %s: %ld", current→key, current→value);
        current = current→next;
    }
    printf("\n");
}
```

```c
#define _POSIX_C_SOURCE 200112L

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

#include <arpa/inet.h>


#include "server_list.h"
#include "server_socket.h"
#include "server_sensor.h"
#include "server_request_processor.h"
#include "server_tamplate.h"

bool process_client(struct List* list, char* temperature, \
                    struct server_template *template, \
                    struct server_socket* socket) {
    bool status = true;
    char* buffer = server_socket_receive_message(socket);
    if ( ¬buffer ) return false;

    struct server_req_proc processor;
    status = req_proc_create(&processor, buffer);
    if ( ¬status ) {
        free(buffer);
        return false;
    }
    char* answer = req_porc_method_resource(&processor);
    if ( ¬answer ) return false;
    server_socket_send_message(socket, answer, strlen(answer));

    if (¬ req_porc_is_method_resource_valid(&processor)) {
        req_proc_destroy(&processor);
        free(buffer);
        free(answer);
        return false;
    }
    char* us_ag = req_porc_user_agent(&processor);
    if (¬us_ag) return false;
    status = list_insert(list, us_ag);
    if ( ¬status ) {
        free(buffer);
        free(answer);
        free(us_ag);
        return false;
    }

    char* reply = server_template_cat(template, temperature);
    if ( ¬reply ) {
        free(buffer);
        free(answer);
        free(us_ag);
        return false;
    }

    server_socket_send_message(socket, reply, strlen(reply));

    req_proc_destroy(&processor);
```



```c
    free(reply);
    free(buffer);
    free(answer);
    free(us_ag);
    return true;
}

int main(int argc, char* argv[]) {
    if (argc ≠4) {
      fprintf(stderr, "Uso:\n./server <puerto> <input> [<template>]\n");
      return 1;
    }

    char* port = argv[1];
    char* sensor_filename = argv[2];
    char* template_filename = argv[3];

    struct server_sensor sensor;
    if ( ¬server_sensor_create(&sensor, sensor_filename) ) return 1;

    struct List list;
    list_create(&list);

    struct server_socket socket;
    if ( ¬server_socket_create(&socket, port) ) {
        server_sensor_destroy(&sensor);
        list_destroy(&list);
        return 1;
    }
    if ( ¬server_socket_start(&socket) ) {
        server_socket_destroy(&socket);
        server_sensor_destroy(&sensor);
        list_destroy(&list);
        return 1;
    }

    struct server_template template;
    if ( ¬server_template_create(&template, template_filename) ) {
        server_socket_destroy(&socket);
        server_sensor_destroy(&sensor);
        list_destroy(&list);
        return 1;
    }
    bool was_last_client_valid = true;
    char* temperature;
    bool is_there_an_error = false;

    while (true) {
        if (was_last_client_valid) {
            temperature = server_sensor_read(&sensor);
        }
        if ( ¬does_the_sensor_still_have_temperatures(&sensor) ){
            break;
        }
        if ( ¬temperature ) {
            is_there_an_error = true;
            break;
        }
        int s = server_socket_accept_client(&socket);
        if (s ≡ −1) {
            is_there_an_error = true;
            break;
        }
        was_last_client_valid = process_client(&list, temperature,\
                                               &template, &socket);
        server_socket_disable_client(&socket);
```

abr 01, 19 17:07 **server.c** Page 3/3

```
133             if (was_last_client_valid) free(temperature);
134         }
135
136         printf("# Estadisticas de visitantes\n");
137         list_print(&list);
138
139         list_destroy(&list);
140         server_socket_destroy(&socket);
141         server_sensor_destroy(&sensor);
142         server_template_destoy(&template);
143         if (is_there_an_error) {
144             return 1;
145         }
146
147         return 0;
148     }
```

abr 01, 19 17:07 **client_socket.h** Page 1/1

```
1   #ifndef CLIENT_SOCKET_H
2   #define CLIENT_SOCKET_H
3   #include <stdlib.h>
4
5   struct client_socket {
6       size_t request_len;
7       char* request;
8       char* host;
9       char* port;
10      int skt;
11  };
12
13  /*
14  Guarda los parametros que se necesitaran al inciar el socket
15  */
16  void client_socket_create(struct client_socket *self, size_t _request_len,\
17                            char* _request, char* _host, char* _port);
18  void client_socket_destroy(struct client_socket *self);
19  /*
20  Crea el socket definiendo la familia, el tipo de socket y el protocolo
21  para poder conectarse al servidor por medio del port y host indicados.
22  */
23  bool client_socket_start(struct client_socket *self);
24  bool client_socket_receive_reponse(struct client_socket *self);
25  bool client_socket_send_request(struct client_socket *self);
26
27  #endif
```

```c
#define _POSIX_C_SOURCE 200112L
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <errno.h>
#include <stdbool.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include "client_socket.h"

#define REQUEST_MAX_LEN 2000
#define RESPONSE_MAX_LEN 2000
#define LEN_HOST 13
#define LEN_PORT 6


bool client_socket_send_request(struct client_socket *self) {
    int s = 0;
    int bytes_sent = 0;

    while (bytes_sent < self→request_len) {
        s = send(self→skt, &self→request[bytes_sent], \
                self→request_len − bytes_sent, MSG_NOSIGNAL);

        if (s ≤ 0) {
            shutdown(self→skt, SHUT_RDWR);
            close(self→skt);
            return false;
        } else {
            bytes_sent += s;
        }
    }
    shutdown(self→skt, SHUT_WR);
    return true;
}

bool client_socket_receive_reponse(struct client_socket *self) {
    int s = 0;
    int bytes_receive = 0;
    char response[RESPONSE_MAX_LEN];

    while ( true ) {
        s = recv(self→skt, &response[bytes_receive], \
                RESPONSE_MAX_LEN − bytes_receive − 1, MSG_NOSIGNAL);
        if (s < 0) { //socker error
            return false;
        } else if (s ≡ 0) {
            break;
        } else {
            bytes_receive = s;
            response[bytes_receive] = 0;
            printf("%s", response);
            bytes_receive = 0;
        }
    }
    printf("\n");
    return true;
}

bool client_socket_start(struct client_socket *self) {
    int s = 0;
    bool are_we_connected = false;
```



```c

    struct addrinfo hints;
    struct addrinfo *result, *ptr;

    self→skt = 0;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = 0;

    s = getaddrinfo(self→host, self→port, &hints, &result);

    if (s ≠ 0) {
        return false;
    }


    for (ptr = result; ptr ≠ NULL ∧ are_we_connected ≡ false;\
        ptr = ptr→ai_next) {
        self→skt = socket(ptr→ai_family, ptr→ai_socktype, ptr→ai_protocol);
        if (self→skt ≡ −1) {
        } else {
            s = connect(self→skt, ptr→ai_addr, ptr→ai_addrlen);
            if (s ≡ −1) {
                close(self→skt);
            }
            are_we_connected = (s ≠ −1);
        }
    }

    freeaddrinfo(result);

    return are_we_connected;
}

void client_socket_create(struct client_socket *self, size_t _request_len,\
                          char* _request, char* _host, char* _port) {
    self→host = malloc(LEN_HOST);
    self→port = malloc(LEN_PORT);
    self→request = malloc(_request_len + 1);

    self→request_len = _request_len;
    snprintf(self→request, _request_len  + 1, "%s", _request);
    snprintf(self→host, LEN_HOST , "%s", _host);
    snprintf(self→port, LEN_PORT , "%s", _port);
}

void client_socket_destroy(struct client_socket *self) {
    shutdown(self→skt, SHUT_RDWR);
    close(self→skt);

    free(self→request);
    free(self→host);
    free(self→port);
}
```

abr 01, 19 17:07 **client_file_copier.h** Page 1/1

```
#ifndef CLIENT_FILE_COPIER_H
#define CLIENT_FILE_COPIER_H
#include <stdint.h>

struct file_copier {
    char* filename;
    char* path;
    size_t* path_len;
};

void file_copier_create(struct file_copier* self,\
                    char* filename, size_t* path_len, char* path);

/*
abre el archivo cuyo nombre tiene almacenado como atributo
y copia su contenido en un buffer tambiÃ©n almacenado como atributo
*/
bool file_copier_start(struct file_copier* self);

#endif
```

abr 01, 19 17:07 **client_file_copier.c** Page 1/1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include "client_file_copier.h"

#define MAX_LEN_FILE 2000

void file_copier_create(struct file_copier* self,\
                        char* filename, size_t* path_len, char* path){
    self→filename = filename;
    self→path = path;
    self→path_len = path_len;
}

bool file_copier_start(struct file_copier* self) {
    FILE* file = fopen(self→filename, "r");
    if (¬file) {
      return false;
    }

    int i = 0;
    while (¬feof(file) ∧ i < MAX_LEN_FILE) {
      int c = fgetc(file);
        self→path[i] = c;
      i++;
    }
    self→path[i-2] = (int)'\0'; //me estaba leyendo un \n y un eof o algo asi
    fclose(file);
    *self→path_len = i - 2;
    return true;
}

```

abr 01, 19 17:07 **client.c** Page 1/1

```c
#define _POSIX_C_SOURCE 200112L
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

#include "client_socket.h"

#include "client_file_copier.h"

#define MAX_LEN_FILENAME 100
#define MAX_LEN_FILE 2000

int main(int argc, char* argv[]) {
    if (argc ≠ 3 ∧ argc ≠4) {
      fprintf(stderr, "Uso:\n./client <direccion> <puerto> [<input>]\n");
      return 1;
    }
    char filename[MAX_LEN_FILENAME];
    if (argc ≡ 3) {
      char* status = fgets(filename, MAX_LEN_FILENAME, stdin);
      filename[strlen(filename) −1] = '\0';
      if (¬status) {
           return 1;
        }
    } else {
      snprintf(filename, MAX_LEN_FILENAME, "%s", argv[3]);
    }

    struct file_copier copier;
    char path[MAX_LEN_FILE];
    size_t path_len;
    file_copier_create(&copier, filename, &path_len, path);

    if (¬file_copier_start(&copier)){
      return 1;
    }

    char* host = argv[1];
    char* port = argv[2];
    struct client_socket socket;
    client_socket_create(&socket, path_len, path, host, port);
    if (¬client_socket_start(&socket)) {
       return 1;
    }
    client_socket_send_request(&socket);
    client_socket_receive_reponse(&socket);
    client_socket_destroy(&socket);
    return 0;
}
```

abr 01, 19 17:07 **Table of Content** Page 1/1

**Table of Contents**