```c
#ifndef SERVER_TEMPLATE_H
#define SERVER_TEMPLATE_H
#include "common_socket.h"
#define MAX_LEN_REPLY 2000

struct template {
    char text[MAX_LEN_REPLY];
    char* to_replace;
    struct socket* skt;
};


bool template_create(struct template *self, char* filename, struct socket* skt);
void template_send_cat(struct template *self, char* replacement);
void template_destroy(struct template *self);

#endif
```

```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 template_create(struct template *self, char* filename, \
                    struct socket* skt) {
    self→skt = skt;
    FILE* file = fopen(filename, "r");
    if (¬file) {
      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;
}

void template_send_cat(struct template *self, char* replacement) {
    char* p = strstr(self→text, TO_REPLACE);
    socket_send_all(self→skt, p − self→text, self→text);
    socket_send_all(self→skt, strlen(replacement), replacement);
    socket_send_all(self→skt, strlen(p + SIZE_TO_REPLACE) − 1 , \
                    p + SIZE_TO_REPLACE);
}

void template_destroy(struct template *self) {
    //do nothing
}
```

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

struct server_socket {
    char* host;
    char* port;
    int skt;
    int current_peerskt;
    struct addrinfo *result;
};


/*
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
*/
void server_socket_create(struct server_socket *self, char* _host, char* _port);

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

bool server_socket_connect(struct server_socket *self);

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

int server_socket_receive_some(struct server_socket *self, char* buf, \
                                size_t size);



int server_socket_send_all(struct server_socket *self, \
                                size_t request_len, char*request);
/*
Desactiva las operaciones de envÃo y recepciÃ³n para el cliente y para si mismo
*/
void server_socket_disable_client(struct server_socket *self);

#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 "server_socket.h"

#define MAX_WAITING_CLIENTS 20



int server_socket_receive_some(struct server_socket *self, char* buf, \
                                size_t size) {
    return recv(self→current_peerskt, buf , size, MSG_NOSIGNAL);
}




//+++++++ IGUAL ++++++++++

void server_socket_create(struct server_socket *self, char* _host,\
                            char* _port) {
    self→host = _host;
    self→port = _port;
    self→current_peerskt = 0;
}

void server_socket_destroy(struct server_socket *self) {
    freeaddrinfo(self→result);
    shutdown(self→skt, SHUT_RDWR);
    close(self→skt);
}

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

    struct addrinfo hints;

    self→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(self→host, self→port, &hints, &self→result);

    if (s ≠ 0) {
        return false;
    }
    return true;
}

int server_socket_send_all(struct server_socket *self, \
                                size_t size, char* buf) {
    int bytes_sent = 0;
    int s = 0;
    bool is_the_socket_valid = true;

    while (bytes_sent < size ∧ is_the_socket_valid) {
        s = send(self→current_peerskt, &buf[bytes_sent], \
                size−bytes_sent, MSG_NOSIGNAL);
        if (s ≤ 0) {
            return −1;
        } else {
            bytes_sent += s;
        }
    }
    return bytes_sent;
}

//+++++ UNICO ++++++++

bool server_socket_connect(struct server_socket *self) {
    struct addrinfo *ptr = self→result;
    int s = 0;

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

    if (self→skt ≡ −1) {
        return false;
    }

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

    s = bind(self→skt, ptr→ai_addr, ptr→ai_addrlen);
    if (s ≡ −1) {
        return false;
    }
    //
    s = listen(self→skt, MAX_WAITING_CLIENTS);
    if (s ≡ −1) {
        return false;
    }
    return true;
}

void server_socket_disable_client(struct server_socket *self) {
        shutdown(self→current_peerskt, SHUT_RDWR);
        close(self→current_peerskt);
}

//−1 si falla
int server_socket_accept_client(struct server_socket *self){
    int peerskt = accept(self→skt, NULL, NULL);
    self→current_peerskt = peerskt;
    return peerskt;
}
```

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

struct sensor {
    FILE* file;
};

bool sensor_create(struct 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* sensor_read(struct sensor* self);

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

void sensor_destroy(struct sensor* self);

#endif

```

```c
#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 sensor_create(struct sensor* self, char* filename) {
    FILE* file = fopen(filename, "rb");
    if (¬file) {
      return false;
    }
    self→file = file;
    return true;
}


char* sensor_read(struct 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 sensor* self) {
    return ¬feof(self→file);
}


void sensor_destroy(struct sensor* self) {
    fclose(self→file);
}
```

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


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


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

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

#endif
```

```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 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 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 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,
```

```c
        return answer;                            //pero si llegase a cambiar
    }                                             //implementacion podria serlo
    position = self→request + RESOURCE_OFFSET;
    if (str_check(position, LEN_RESOURCE, CORRECT_RESOURCE)) {
        snprintf(answer, MAX_LEN_MESSAGE, "%s", RESOURCE_ERROR_MESSAGE);
        self→is_method_resource_valid = false;
      return answer;
    }
    snprintf(answer, MAX_LEN_MESSAGE, "%s", METHRES_SUCCESS_MESSAGE);
    self→is_method_resource_valid = true;
    return answer;
}


bool req_porc_is_method_resource_valid(struct req_proc* self) {
    return self→is_method_resource_valid;
}


char* req_porc_user_agent(struct req_proc* self) {
    char* key_start = strstr(self→request, USER_AGENT_KEY);
    if (¬key_start) {
        return NULL;
    }
    char* value_start = key_start + USER_AGENT_VAL_OFFSET;
    size_t len_value = 0;
    while (value_start[len_value] ∧ \
           strcmp(value_start + len_value, END_USER_AGENT_VAL)) {
               len_value++;
    }
    char* value = malloc(MAX_LEN_USER_AGENT_VALUE);
    if (¬value) return NULL;
    snprintf(value, MAX_LEN_USER_AGENT_VALUE, "%s", value_start);
    char* value_end = strstr(value, "\n");
    if (value_end) { //sino eof => strstr deja \n
        *value_end = '\0';
    }
    return value;
}
```

server_list.h

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

struct nodo;

struct List {
    struct nodo* first;
    struct nodo* last;
    size_t len;
};

struct List;
void list_create(struct List *self);
void list_destroy(struct List *self);

/*
Dada una lista y una clave se agrega la clave a la lista
y se inicializa su valor.
En caso de ya existir la clave solamente se aumenta en uno su valor.
*/
bool list_insert(struct List *self, char* _key);

/*
Imprime todos los datos almacenados de la forma
* <clave1>: <valor1>
* <clave2>: <valor2>
*/
void list_print(struct List *self);

#endif

```

server_list.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;
```

```c
    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 "common_socket.h"
#include "server_sensor.h"
#include "server_request_processor.h"
#include "server_tamplate.h"

#define MAX_LEN_BUF 2000

void recive(struct socket* socket, char* buffer, size_t len) {
    memset(buffer, 0, len);
    int received = 0;
    int status = 0;
    bool is_there_an_error = false;
    while (received < len ∧ ¬is_there_an_error) {
        status = socket_receive_some(socket, buffer + received, \
                    len − received);
        if (status ≡ 0) {
            is_there_an_error = true;
        } else if (status < 0) {
            is_there_an_error = true;
        } else {
            received += status;
        }
    }
}


bool process_client(struct List* list, char* temperature, \
                    struct template *template, \
                    struct socket* socket) {
    bool status = true;
    char buffer[MAX_LEN_BUF];
    recive(socket, buffer, MAX_LEN_BUF);


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

    if (¬ req_porc_is_method_resource_valid(&processor)) {
        req_proc_destroy(&processor);

        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(answer);
        free(us_ag);
        return false;
    }
    template_send_cat(template, temperature);

    req_proc_destroy(&processor);

    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 sensor sensor;
    if ( ¬sensor_create(&sensor, sensor_filename) ) return 1;

    struct List list;
    list_create(&list);

    struct socket socket;
    socket_create(&socket,NULL, port);

    if ( ¬socket_start(&socket) ) {
        socket_destroy(&socket);
        sensor_destroy(&sensor);
        list_destroy(&list);
        return 1;
    }

    if ( ¬ socket_connect_with_clients(&socket) ) {
        socket_destroy(&socket);
        sensor_destroy(&sensor);
        list_destroy(&list);
    }

    struct template template;
    if ( ¬template_create(&template, template_filename, &socket) ) {
        socket_destroy(&socket);
        sensor_destroy(&sensor);
        list_destroy(&list);
        return 1;
    }

    bool is_there_an_error = false;

  bool was_last_client_valid = true;
    char* temperature;

    while (true) {
        if (was_last_client_valid) {
            temperature = sensor_read(&sensor);
        }
```

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

    printf("# Estadisticas de visitantes\n");
    list_print(&list);

    template_destroy(&template);
    sensor_destroy(&sensor);
    socket_destroy(&socket);
    list_destroy(&list);

    if (is_there_an_error) {
        return 1;
    }

    return 0;
}
```

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

struct socket {
    char* host;
    char* port;
    int skt;
    int current_peerskt;
    struct addrinfo *result;
};


/*
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
*/
void socket_create(struct socket *self, char* _host, char* _port);

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

bool socket_connect_with_clients(struct socket *self);

void socket_destroy(struct socket *self);
int socket_accept_client(struct socket *self);

int socket_receive_some(struct socket *self, char* buf, \
                                size_t size);



int socket_send_all(struct socket *self, \
                                size_t request_len, char*request);
/*
Desactiva las operaciones de envÃo y recepciÃ³n para el cliente y para si mismo
*/
void socket_disable_client(struct socket *self);

//desabilita el canal de escritura
void socket_disables_send_operations(struct socket *self);
bool socket_connect_with_server(struct socket *self);

#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 "common_socket.h"

#define MAX_WAITING_CLIENTS 20

//+++++++ IGUAL ++++++++++

int socket_receive_some(struct socket *self, char* buf, \
                                size_t size) {
    return recv(self→current_peerskt, buf , size, MSG_NOSIGNAL);
}

int socket_send_all(struct socket *self, \
                                size_t size, char* buf) {
    int bytes_sent = 0;
    int s = 0;
    bool is_the_socket_valid = true;

    while (bytes_sent < size ∧ is_the_socket_valid) {
        s = send(self→current_peerskt, &buf[bytes_sent], \
                size−bytes_sent, MSG_NOSIGNAL);
        if (s ≤ 0) {
            return −1;
        } else {
            bytes_sent += s;
        }
    }
    return bytes_sent;
}


void socket_create(struct socket *self, char* _host,\
                            char* _port) {
    self→host = _host;
    self→port = _port;
    self→skt = 0;
    self→current_peerskt = 0;
}

void socket_destroy(struct socket *self) {
    freeaddrinfo(self→result);
    shutdown(self→skt, SHUT_RDWR);
    close(self→skt);
}

bool socket_start(struct socket *self) {
    int s = 0;

    struct addrinfo hints;

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

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

    if (s ≠ 0) {
        return false;
    }
    return true;
}



//+++++ UNICO ++++++++

bool socket_connect_with_clients(struct socket *self) {
    struct addrinfo *ptr = self→result;
    int s = 0;

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

    if (self→skt ≡ −1) {
        return false;
    }

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

    s = bind(self→skt, ptr→ai_addr, ptr→ai_addrlen);
    if (s ≡ −1) {
        return false;
    }
    //
    s = listen(self→skt, MAX_WAITING_CLIENTS);
    if (s ≡ −1) {
        return false;
    }
    return true;
}

void socket_disable_client(struct socket *self) {
        shutdown(self→current_peerskt, SHUT_RDWR);
        close(self→current_peerskt);
}

//−1 si falla
int socket_accept_client(struct socket *self){
    int peerskt = accept(self→skt, NULL, NULL);
    self→current_peerskt = peerskt;
    return peerskt;
}


//+++++ UNICO ++++++++

bool socket_connect_with_server(struct socket *self) {
    int s = 0;
    bool are_we_connected = false;
    struct addrinfo *ptr;

    for (ptr = self→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) continue;
        s = connect(self→skt, ptr→ai_addr, ptr→ai_addrlen);
        are_we_connected = (s ≠ −1);
```

```c
    }
    self→current_peerskt = self→skt;

    return are_we_connected;
}

void socket_disables_send_operations(struct socket *self) {
    shutdown(self→skt, SHUT_WR);
}
```

```c
#ifndef CLIENT_SOCKET_H
#define CLIENT_SOCKET_H
#include <stdlib.h>


struct client_socket {
    char* host;
    char* port;
    int skt;
    int current_peerskt;
    struct addrinfo *result;
};

/*
Guarda los parametros que se necesitaran al inciar el socket
*/
void client_socket_create(struct client_socket *self, char* _host, char*_port);
void client_socket_destroy(struct client_socket *self);
/*
Crea el socket definiendo la familia, el tipo de socket y el protocolo
para poder conectarse al servidor por medio del port y host indicados.
*/
bool client_socket_start(struct client_socket *self);
int client_socket_send_all(struct client_socket *self, \
                                size_t size, char* buf);
//desabilita el canal de escritura
void client_socket_disables_send_operations(struct client_socket *self);
int client_socket_receive_some(struct client_socket *self, char* buf, \
                                size_t size);


bool client_socket_connect(struct client_socket *self);
#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



int client_socket_receive_some(struct client_socket *self, char* buf, \
                                size_t size) {
    int i = recv(self→skt, buf , size, MSG_NOSIGNAL);
    return i;
}



//+++++++ IGUAL ++++++++++

void client_socket_create(struct client_socket *self, char* _host,\
                            char*_port) {
    self→host = _host;
    self→port = _port;
    self→current_peerskt = 0;
}

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

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

    struct addrinfo hints;

    self→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(self→host, self→port, &hints, &self→result);

    if (s ≠ 0) {
        return false;
    }
    return true;
}

int client_socket_send_all(struct client_socket *self, \
                                size_t size, char* buf) {
```

```
    int bytes_sent = 0;
    int s = 0;
    bool is_the_socket_valid = true;


    while (bytes_sent < size ∧ is_the_socket_valid) {
        s = send(self→skt, &buf[bytes_sent], \
                size − bytes_sent, MSG_NOSIGNAL);
        if (s ≤ 0) {
            return −1;
        } else {
            bytes_sent += s;
        }
    }
    return bytes_sent;
}

//+++++ UNICO ++++++++

bool client_socket_connect(struct client_socket *self) {
    int s = 0;
    bool are_we_connected = false;
    struct addrinfo *ptr;

    for (ptr = self→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) continue;
        s = connect(self→skt, ptr→ai_addr, ptr→ai_addrlen);
        are_we_connected = (s ≠ −1);
    }

    return are_we_connected;
}

void client_socket_disables_send_operations(struct client_socket *self) {
    shutdown(self→skt, SHUT_WR);
}
```

```
#ifndef CLIENT_FILE_SENDER_H
#define CLIENT_FILE_SENDER_H
#include <stdint.h>
#include "common_socket.h"

struct file_sender {
    char* filename;
    struct socket* socket;
};

void file_sender_create(struct file_sender* self,\
                     char* _filename, struct socket* _socket);

/*
abre el archivo cuyo nombre tiene almacenado como atributo
y envia su contenido a travÃ©s de socket que tiene almacenado como atributo
*/
bool file_sender_start(struct file_sender* self);

void file_sender_destroy(struct file_sender* self);

#endif
```

```c
#define _POSIX_C_SOURCE 200809L //getline
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include "client_file_sender.h"

void file_sender_create(struct file_sender* self,\
                        char* _filename, struct socket* _socket){
    self→filename = _filename;
    self→socket = _socket;
}

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

    char* lineptr = NULL; size_t n = 0; size_t len;
    while (true) {
        len = getline(&lineptr, &n, file);
        if (len ≡ −1) break;
        if ( socket_send_all(self→socket, len, lineptr) ≡ −1 ) {
            return false;
        }
    }

    fclose(file);
    free(lineptr);
    return true;
}

void file_sender_destroy(struct file_sender* self) {
    //do nothing
}
```

```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 "common_socket.h"

#include "client_file_sender.h"

#define MAX_LEN_FILENAME 100
#define MAX_LEN_BUF 2000

bool resive(struct socket* socket) {
    int status = 0;
    int received = 0;
    char buf[MAX_LEN_BUF];

    while ( true ) {
        status = socket_receive_some(socket, &buf[received], \
                 MAX_LEN_BUF − received − 1);
        if (status < 0) { //socker error
            return false;
        } else if (status ≡ 0) {
            break;
        } else {
            received = status;
            buf[received] = 0;
            printf("%s", buf);
            received = 0;
        }
    }
    printf("\n");
    return true;
}


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_sender fs;
    char* host = argv[1];
    char* port = argv[2];
    struct socket socket;
    socket_create(&socket, host, port);

    if (¬socket_start(&socket)) {
        socket_destroy(&socket);
        return 1;
    }
    if (¬socket_connect_with_server(&socket)) {
        socket_destroy(&socket);
        return 1;
    }

    int status = 0;
    file_sender_create(&fs, filename, &socket);

    if (¬file_sender_start(&fs)){
        status = 1;
    }


    socket_disables_send_operations(&socket);
    if ( ¬resive(&socket) ) {
        status = 1;
    }

    socket_destroy(&socket);
    file_sender_destroy(&fs);
    return status;
}
```

Table of Contents

```
 1 server_tamplate.h... sheets   1 to   1 ( 1) pages   1-  1   18 lines
 2 server_tamplate.c... sheets   1 to   1 ( 1) pages   2-  2   47 lines
 3 server_socket.h..... sheets   2 to   2 ( 1) pages   3-  3   44 lines
 4 server_socket.c..... sheets   2 to   3 ( 2) pages   4-  5  121 lines
 5 server_sensor.h..... sheets   3 to   3 ( 1) pages   6-  6   24 lines
 6 server_sensor.c..... sheets   4 to   4 ( 1) pages   7-  7   45 lines
 7 server_request_processor.h sheets   4 to   4 ( 1) pages   8-  8   36 lines
 8 server_request_processor.c sheets   5 to   5 ( 1) pages   9- 10  106 lines
 9 server_list.h....... sheets   6 to   6 ( 1) pages  11- 11   33 lines
10 server_list.c....... sheets   6 to   7 ( 2) pages  12- 13   79 lines
11 server.c............ sheets   7 to   8 ( 2) pages  14- 16  165 lines
12 common_socket.h..... sheets   9 to   9 ( 1) pages  17- 17   48 lines
13 common_socket.c..... sheets   9 to  10 ( 2) pages  18- 20  142 lines
14 client_socket.h..... sheets  11 to  11 ( 1) pages  21- 21   34 lines
15 client_socket.c..... sheets  11 to  12 ( 2) pages  22- 23  105 lines
16 client_file_sender.h sheets  12 to  12 ( 1) pages  24- 24   23 lines
17 client_file_sender.c sheets  13 to  13 ( 1) pages  25- 25   39 lines
18 client.c............ sheets  13 to  14 ( 2) pages  26- 27   92 lines
```