

abr 10, 19 14:32

server_template.h

Page 1/1

```

1  #ifndef SERVER_TEMPLATE_H
2  #define SERVER_TEMPLATE_H
3  #include "common_socket.h"
4  #define MAX_LEN_REPLY 2000
5
6  struct template {
7      char text[MAX_LEN_REPLY];
8      char* to_replace;
9      struct socket* skt;
10 };
11
12
13 bool template_create(struct template *self, char* filename, struct socket* skt);
14 void template_send_cat(struct template *self, char* replacement);
15 void template_destroy(struct template *self);
16
17 #endif

```

abr 10, 19 14:32

server_template.c

Page 1/1

```

1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <errno.h>
6  #include <stdbool.h>
7  #include "server_template.h"
8
9
10 #define TO_REPLACE "{{datos}}"
11 #define SIZE_TO_REPLACE 9
12 #define MAX_LEN_REPLY 2000
13
14
15
16
17 bool template_create(struct template *self, char* filename, \
18                     struct socket* skt) {
19     self->skt = skt;
20     FILE* file = fopen(filename, "r");
21     if (!file) {
22         return false;
23     }
24
25     size_t i = 0;
26     while (!feof(file) & i < MAX_LEN_REPLY) {
27         self->text[i] = (char) fgetc(file);
28         i++;
29     }
30     self->text[i-1] = '\0';
31     self->to_replace = strstr(self->text, TO_REPLACE);
32     fclose(file);
33     return true;
34 }
35
36 void template_send_cat(struct template *self, char* replacement) {
37     char* p = strstr(self->text, TO_REPLACE);
38     socket_send_all(self->skt, p - self->text, self->text);
39     socket_send_all(self->skt, strlen(replacement), replacement);
40     socket_send_all(self->skt, strlen(p + SIZE_TO_REPLACE) - 1, \
41                    p + SIZE_TO_REPLACE);
42 }
43
44 void template_destroy(struct template *self) {
45     //do nothing
46 }

```

abr 10, 19 14:32

server_socket.h

Page 1/1

```

1  #ifndef SERVER_SOCKET_H
2  #define SERVER_SOCKET_H
3  #include <stdlib.h>
4  #include <stdbool.h>
5
6  struct server_socket {
7      char* host;
8      char* port;
9      int skt;
10     int current_peerskt;
11     struct addrinfo *result;
12 };
13
14
15 /*
16  Crea e incializa el socket definiendo la familia, el tipo de socket y el
17  protocolo para poder conectarse al cliente por medio del port y host indicados
18  */
19 void server_socket_create(struct server_socket *self, char* _host, char* _port);
20
21 /*
22  Almacena los parametros necesarios para la incializaciÃ³n del socket.
23  */
24 bool server_socket_start(struct server_socket *self);
25
26 bool server_socket_connect(struct server_socket *self);
27
28 void server_socket_destroy(struct server_socket *self);
29 int server_socket_accept_client(struct server_socket *self);
30
31 int server_socket_receive_some(struct server_socket *self, char* buf, \
32                               size_t size);
33
34
35
36 int server_socket_send_all(struct server_socket *self, \
37                            size_t request_len, char*request);
38
39 /*
40  Desactiva las operaciones de envÃ­o y recepciÃ³n para el cliente y para si mismo
41  */
42 void server_socket_disable_client(struct server_socket *self);
43
44 #endif

```

abr 10, 19 14:32

server_socket.c

Page 1/2

```

1  #define _POSIX_C_SOURCE 200112L
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #include <errno.h>
7  #include <stdbool.h>
8
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include <unistd.h>
13 #include "server_socket.h"
14
15 #define MAX_WAITING_CLIENTS 20
16
17
18
19 int server_socket_receive_some(struct server_socket *self, char* buf, \
20                               size_t size) {
21     return recv(self->current_peerskt, buf, size, MSG_NOSIGNAL);
22 }
23
24
25
26
27 //+++++ IGUAL ++++++
28
29 void server_socket_create(struct server_socket *self, char* _host, \
30                           char* _port) {
31     self->host = _host;
32     self->port = _port;
33     self->current_peerskt = 0;
34 }
35
36 void server_socket_destroy(struct server_socket *self) {
37     freeaddrinfo(self->result);
38     shutdown(self->skt, SHUT_RDWR);
39     close(self->skt);
40 }
41
42 bool server_socket_start(struct server_socket *self) {
43     int s = 0;
44
45     struct addrinfo hints;
46
47     self->skt = 0;
48
49     memset(&hints, 0, sizeof(struct addrinfo));
50     hints.ai_family = AF_INET;
51     hints.ai_socktype = SOCK_STREAM;
52     hints.ai_flags = AI_PASSIVE;
53
54     s = getaddrinfo(self->host, self->port, &hints, &self->result);
55
56     if (s != 0) {
57         return false;
58     }
59     return true;
60 }
61
62 int server_socket_send_all(struct server_socket *self, \
63                            size_t size, char* buf) {
64     int bytes_sent = 0;
65     int s = 0;
66     bool is_the_socket_valid = true;

```

abr 10, 19 14:32

server_socket.c

Page 2/2

```

67     while (bytes_sent < size ^ is_the_socket_valid) {
68         s = send(self->current_peerskt, &buf[bytes_sent], \
69             size-bytes_sent, MSG_NOSIGNAL);
70         if (s ≤ 0) {
71             return -1;
72         } else {
73             bytes_sent += s;
74         }
75     }
76     return bytes_sent;
77 }
78
79 //+++++ UNICO +++++
80
81 bool server_socket_connect(struct server_socket *self) {
82     struct addrinfo *ptr = self->result;
83     int s = 0;
84
85     self->skt = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
86
87     if (self->skt ≡ -1) {
88         return false;
89     }
90
91     int val = 1;
92     s = setsockopt(self->skt, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
93     if (s ≡ -1) {
94         return false;
95     }
96
97     s = bind(self->skt, ptr->ai_addr, ptr->ai_addrlen);
98     if (s ≡ -1) {
99         return false;
100     }
101     //
102     s = listen(self->skt, MAX_WAITING_CLIENTS);
103     if (s ≡ -1) {
104         return false;
105     }
106     return true;
107 }
108
109 void server_socket_disable_client(struct server_socket *self) {
110     shutdown(self->current_peerskt, SHUT_RDWR);
111     close(self->current_peerskt);
112 }
113
114 // -1 si falla
115 int server_socket_accept_client(struct server_socket *self){
116     int peerskt = accept(self->skt, NULL, NULL);
117     self->current_peerskt = peerskt;
118     return peerskt;
119 }
120

```

abr 10, 19 14:32

server_sensor.h

Page 1/1

```

1  #ifndef SERVER_SENSOR_H
2  #define SERVER_SENSOR_H
3  #include <stdbool.h>
4
5  struct sensor {
6      FILE* file;
7  };
8
9  bool sensor_create(struct sensor* self, char* filename);
10 /*
11 Se ocupa de leer de del archivo binario file una temperatura
12 almacenada en 16 bits y formato big-endian.
13 La misma la interpreta de la siguiente forma: Temperatura = (datos - 2000)/100
14 */
15 char* sensor_read(struct sensor* self);
16
17 //comunica si quedan o no temperaturas por leer.
18 bool does_the_sensor_still_have_temperatures(struct sensor* self);
19
20 void sensor_destroy(struct sensor* self);
21
22 #endif
23

```

abr 10, 19 14:32

server_sensor.c

Page 1/1

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <errno.h>
5  #include <stdbool.h>
6  #include <arpa/inet.h>
7
8  #include "server_sensor.h"
9
10 #define SIZE_OF_TEMPERATURE 2
11 #define MAX_LEN_TEMPERATURE_MESSAGE 200
12 #define SIZE_TO_REPLACE 9
13
14
15 bool sensor_create(struct sensor* self, char* filename) {
16     FILE* file = fopen(filename, "rb");
17     if (!file) {
18         return false;
19     }
20     self->file = file;
21     return true;
22 }
23
24
25 char* sensor_read(struct sensor* self) {
26     unsigned short int read;
27     size_t len = fread((void*)&read, SIZE_OF_TEMPERATURE, 1, self->file);
28     if (!len) return NULL;
29     float temperature = (float) ntohs(read);
30     temperature = (temperature - 2000) / 100;
31     char* message = malloc(MAX_LEN_TEMPERATURE_MESSAGE);
32     if (!message) return NULL;
33     snprintf(message, SIZE_TO_REPLACE, "%.2f", temperature);
34     return message;
35 }
36
37 bool does_the_sensor_still_have_temperatures(struct sensor* self) {
38     return !feof(self->file);
39 }
40
41
42 void sensor_destroy(struct sensor* self) {
43     fclose(self->file);
44 }

```

abr 10, 19 14:32

server_request_processor.h

Page 1/1

```

1  #ifndef SERVER_REQUEST_PROCESSOR_H
2  #define SERVER_REQUEST_PROCESSOR_H
3  #include <stdbool.h>
4
5
6  struct req_proc {
7      char* request;
8      bool is_method_resource_valid;
9  };
10
11
12 bool req_proc_create(struct req_proc* self, char* request);
13 void req_proc_destroy(struct req_proc* self);
14
15 /*
16  Verifica que el método utilizado sea del tipo "GET"
17  y el recurso sea "/sensor".
18
19  Si el método no es "GET", la respuesta será; un
20  error de tipo "400 Bad request"
21  Si el recurso no es "/sensor", la respuesta será; un error
22  de tipo "404 Not found".
23  Si el método y recurso son válidos, la respuesta es
24  de tipo "200 OK".
25  */
26 char* req_porc_method_resource(struct req_proc* self);
27 bool req_porc_is_method_resource_valid(struct req_proc* self);
28
29 /*
30  Busca y devuelve el valor del user-agent en el
31  request con formato clave:valor.
32  */
33 char* req_porc_user_agent(struct req_proc* self);
34
35 #endif

```

abr 10, 19 14:32

server_request_processor.c

Page 1/2

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include <errno.h>
6  #include <stdbool.h>
7  #include "server_request_processor.h"
8
9  #define METHOD_OFFSET 0
10 #define METHOD_ERROR_MESSAGE "HTTP/1.1 400 Bad request\n"
11 #define LEN_METHOD 3
12 #define CORRECT_METHOD "GET"
13
14 #define RESOURCE_OFFSET 4
15 #define RESOURCE_ERROR_MESSAGE "HTTP/1.1 404 Not found\n"
16
17 #define LEN_RESOURCE 7
18 #define CORRECT_RESOURCE "/sensor"
19
20 #define MAX_LEN_REQUEST 2000
21
22 #define METHOD_ERROR_MESSAGE "HTTP/1.1 400 Bad request\n"
23 #define RESOURCE_ERROR_MESSAGE "HTTP/1.1 404 Not found\n"
24 #define METHRES_SUCCESS_MESSAGE "HTTP/1.1 200 OK\n\n"
25 #define MAX_LEN_MESSAGE 26
26
27 #define USER_AGENT_KEY "User-Agent:"
28 #define USER_AGENT_VAL_OFFSET 12
29 #define END_USER_AGENT_VAL "\n"
30 #define MAX_LEN_USER_AGENT_VALUE 200
31
32
33
34 bool req_proc_create(struct req_proc* self, char* request) {
35     self->request = malloc(MAX_LEN_REQUEST);
36     if ( !self->request ) return false;
37     snprintf(self->request, MAX_LEN_REQUEST, "%s", request);
38     self->is_method_resource_valid = false;
39     return true;
40 }
41
42
43 void req_proc_destroy(struct req_proc* self) {
44     free(self->request);
45 }
46
47 //Verifica que el comando str se encuentre en request
48 //De ser asÃ- devuelve 0
49 //en caso contrariodevuelve 1.
50 int str_check(const char* request, size_t len, const char* str) { //, char* err) {
51     for (int i = 0; i < len; ++i) {
52         if (request[i] != str[i]) {
53             return 1;
54         }
55     }
56     return 0;
57 }
58
59 char* req_porc_method_resource(struct req_proc* self) {
60     char* answer = malloc(MAX_LEN_MESSAGE);
61     if ( !answer ) return NULL;
62     enum error {METHRES_SUCCESS, METHOD_ERROR, RESOURCE_ERROR};
63     char* position = self->request + METHOD_OFFSET;
64     if (str_check(position, LEN_METHOD, CORRECT_METHOD)) {
65         snprintf(answer, MAX_LEN_MESSAGE, "%s", METHOD_ERROR_MESSAGE);
66         self->is_method_resource_valid = false; //no es necesario,

```

abr 10, 19 14:32

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 req_proc* self) {
82     return self->is_method_resource_valid;
83 }
84
85
86 char* req_porc_user_agent(struct 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] != '\n' & \
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 10, 19 14:32

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

```

abr 10, 19 14:32

server_list.c

Page 1/2

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "server_list.h"
6
7  struct nodo {
8      char* key;
9      long int value;
10     struct nodo* next;
11 } typedef nodo_t;
12
13
14
15 bool nodo_create(nodo_t* self, char* _key) {
16     self->value = 1;
17     self->next = NULL;
18     size_t len_key = strlen(_key) + 1;
19     self->key = malloc(len_key);
20     if ( !self->key ) return false;
21     snprintf(self->key, len_key, "%s", _key);
22     return true;
23 }
24
25 void list_create(struct List *self) {
26     self->first = NULL;
27     self->last = NULL;
28     //self->len = 0;
29 }
30
31 void list_destroy(struct List *self) {
32     nodo_t* current = self->first;
33     nodo_t* aux;
34     while (current) {
35         aux = current->next;
36         free(current->key);
37         free(current);
38         current = aux;
39     }
40 }
41
42 bool list_insert(struct List *self, char* _key) {
43     bool status = true;
44     nodo_t* current = self->first;
45     while (current) {
46         if ( !strcmp(current->key, _key) ){
47             current->value++;
48             return true;
49         }
50         current = current->next;
51     }
52
53     nodo_t* nodo = malloc(sizeof(nodo_t));
54     if (!nodo) return false;
55     status = nodo_create(nodo, _key);
56     if ( !status ) {
57         free(nodo);
58         return false;
59     }
60     if ( !self->first ) {
61         self->first = nodo;
62         self->last = nodo;
63         return true;
64     }
65     self->last->next = nodo;
66     self->last = nodo;

```

abr 10, 19 14:32

server_list.c

Page 2/2

```

67     return true;
68 }
69
70 void list_print(struct List *self){
71     //printf("# Estadísticas de visitantes\n");
72     nodo_t* current = self->first;
73     while (current) {
74         printf("\n* %s: %ld", current->key, current->value);
75         current = current->next;
76     }
77     printf("\n");
78 }

```

abr 10, 19 14:32

server.c

Page 1/3

```

1  #define _POSIX_C_SOURCE 200112L
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <errno.h>
7  #include <stdbool.h>
8
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include <unistd.h>
13
14 #include <arpa/inet.h>
15
16
17 #include "server_list.h"
18 #include "common_socket.h"
19 #include "server_sensor.h"
20 #include "server_request_processor.h"
21 #include "server_template.h"
22
23 #define MAX_LEN_BUF 2000
24
25 void recive(struct socket* socket, char* buffer, size_t len) {
26     memset(buffer, 0, len);
27     int received = 0;
28     int status = 0;
29     bool is_there_an_error = false;
30     while (received < len ^ !is_there_an_error) {
31         status = socket_receive_some(socket, buffer + received, \
32             len - received);
33         if (status == 0) {
34             is_there_an_error = true;
35         } else if (status < 0) {
36             is_there_an_error = true;
37         } else {
38             received += status;
39         }
40     }
41 }
42
43
44 bool process_client(struct List* list, char* temperature, \
45     struct template *template, \
46     struct socket* socket) {
47     bool status = true;
48     char buffer[MAX_LEN_BUF];
49     recive(socket, buffer, MAX_LEN_BUF);
50
51
52     struct req_proc processor;
53     status = req_proc_create(&processor, buffer);
54     if ( !status ) {
55         return false;
56     }
57     char* answer = req_porc_method_resource(&processor);
58     if ( !answer ) return false;
59     socket_send_all(socket, strlen(answer), answer);
60
61     if ( !req_porc_is_method_resource_valid(&processor) ) {
62         req_proc_destroy(&processor);
63
64         free(answer);
65         return false;
66     }

```

abr 10, 19 14:32

server.c

Page 2/3

```

67 char* us_ag = req_porc_user_agent(&processor);
68 if (!us_ag) return false;
69 status = list_insert(list, us_ag);
70 if (!status) {
71     free(answer);
72     free(us_ag);
73     return false;
74 }
75 template_send_cat(template, temperature);
76
77 req_proc_destroy(&processor);
78
79 free(answer);
80 free(us_ag);
81 return true;
82 }
83
84 int main(int argc, char* argv[]) {
85     if (argc != 4) {
86         fprintf(stderr, "Uso:\n/server <puerto> <input> [<template>]\n");
87         return 1;
88     }
89
90     char* port = argv[1];
91     char* sensor_filename = argv[2];
92     char* template_filename = argv[3];
93
94     struct sensor sensor;
95     if (!sensor_create(&sensor, sensor_filename)) return 1;
96
97     struct List list;
98     list_create(&list);
99
100     struct socket socket;
101     socket_create(&socket, NULL, port);
102
103     if (!socket_start(&socket)) {
104         socket_destroy(&socket);
105         sensor_destroy(&sensor);
106         list_destroy(&list);
107         return 1;
108     }
109
110     if (!socket_connect_with_clients(&socket)) {
111         socket_destroy(&socket);
112         sensor_destroy(&sensor);
113         list_destroy(&list);
114     }
115
116     struct template template;
117     if (!template_create(&template, template_filename, &socket)) {
118         socket_destroy(&socket);
119         sensor_destroy(&sensor);
120         list_destroy(&list);
121         return 1;
122     }
123
124     bool is_there_an_error = false;
125
126     bool was_last_client_valid = true;
127     char* temperature;
128
129     while (true) {
130         if (was_last_client_valid) {
131             temperature = sensor_read(&sensor);
132         }

```

abr 10, 19 14:32

server.c

Page 3/3

```

133     if (!does_the_sensor_still_have_temperatures(&sensor)) {
134         break;
135     }
136     if (!temperature) {
137         is_there_an_error = true;
138         break;
139     }
140     int s = socket_accept_client(&socket);
141     if (s == -1) {
142         is_there_an_error = true;
143         break;
144     }
145     was_last_client_valid = process_client(&list, temperature, \
146                                           &template, &socket);
147     socket_disable_client(&socket);
148     if (was_last_client_valid) free(temperature);
149 }
150
151 printf("# Estadísticas de visitantes\n");
152 list_print(&list);
153
154 template_destroy(&template);
155 sensor_destroy(&sensor);
156 socket_destroy(&socket);
157 list_destroy(&list);
158
159 if (is_there_an_error) {
160     return 1;
161 }
162
163 return 0;
164 }

```


abr 10, 19 14:32

common_socket.h

Page 1/1

```

1  #ifndef COMMON_SOCKET_H
2  #define COMMON_SOCKET_H
3  #include <stdlib.h>
4  #include <stdbool.h>
5
6  struct socket {
7      char* host;
8      char* port;
9      int skt;
10     int current_peerskt;
11     struct addrinfo *result;
12 };
13
14
15 /*
16  Crea e incializa el socket definiendo la familia, el tipo de socket y el
17  protocolo para poder conectarse al cliente por medio del port y host indicados
18  */
19 void socket_create(struct socket *self, char* _host, char* _port);
20
21 /*
22  Almacena los parametros necesarios para la incializaciÃ³n del socket.
23  */
24 bool socket_start(struct socket *self);
25
26 bool socket_connect_with_clients(struct socket *self);
27
28 void socket_destroy(struct socket *self);
29 int socket_accept_client(struct socket *self);
30
31 int socket_receive_some(struct socket *self, char* buf, \
32                        size_t size);
33
34
35
36 int socket_send_all(struct socket *self, \
37                    size_t request_len, char*request);
38
39 /*
40  Desactiva las operaciones de envÃ­o y recepciÃ³n para el cliente y para si mismo
41  */
42 void socket_disable_client(struct socket *self);
43
44 //desabilita el canal de escritura
45 void socket_disables_send_operations(struct socket *self);
46 bool socket_connect_with_server(struct socket *self);
47
48 #endif

```

abr 10, 19 14:32

common_socket.c

Page 1/3

```

1  #define _POSIX_C_SOURCE 200112L
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #include <errno.h>
7  #include <stdbool.h>
8
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include <unistd.h>
13 #include "common_socket.h"
14
15 #define MAX_WAITING_CLIENTS 20
16
17 //***** IGUAL *****
18
19 int socket_receive_some(struct socket *self, char* buf, \
20                        size_t size) {
21     return recv(self->current_peerskt, buf, size, MSG_NOSIGNAL);
22 }
23
24 int socket_send_all(struct socket *self, \
25                    size_t size, char* buf) {
26     int bytes_sent = 0;
27     int s = 0;
28     bool is_the_socket_valid = true;
29
30     while (bytes_sent < size ^ is_the_socket_valid) {
31         s = send(self->current_peerskt, &buf[bytes_sent], \
32                size-bytes_sent, MSG_NOSIGNAL);
33         if (s <= 0) {
34             return -1;
35         } else {
36             bytes_sent += s;
37         }
38     }
39     return bytes_sent;
40 }
41
42
43 void socket_create(struct socket *self, char* _host, \
44                  char* _port) {
45     self->host = _host;
46     self->port = _port;
47     self->skt = 0;
48     self->current_peerskt = 0;
49 }
50
51 void socket_destroy(struct socket *self) {
52     freeaddrinfo(self->result);
53     shutdown(self->skt, SHUT_RDWR);
54     close(self->skt);
55 }
56
57 bool socket_start(struct socket *self) {
58     int s = 0;
59
60     struct addrinfo hints;
61
62     memset(&hints, 0, sizeof(struct addrinfo));
63     hints.ai_family = AF_INET;
64     hints.ai_socktype = SOCK_STREAM;
65     hints.ai_flags = AI_PASSIVE;
66

```

abr 10, 19 14:32

common_socket.c

Page 2/3

```

67     s = getaddrinfo(self->host, self->port, &hints, &self->result);
68
69     if (s != 0) {
70         return false;
71     }
72     return true;
73 }
74
75
76
77 //+++++ UNICO ++++++
78
79 bool socket_connect_with_clients(struct socket *self) {
80     struct addrinfo *ptr = self->result;
81     int s = 0;
82
83     self->skt = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
84
85     if (self->skt == -1) {
86         return false;
87     }
88
89     int val = 1;
90     s = setsockopt(self->skt, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
91     if (s == -1) {
92         return false;
93     }
94
95     s = bind(self->skt, ptr->ai_addr, ptr->ai_addrlen);
96     if (s == -1) {
97         return false;
98     }
99     //
100    s = listen(self->skt, MAX_WAITING_CLIENTS);
101    if (s == -1) {
102        return false;
103    }
104    return true;
105 }
106
107 void socket_disable_client(struct socket *self) {
108     shutdown(self->current_peerskt, SHUT_RDWR);
109     close(self->current_peerskt);
110 }
111
112 // -1 si falla
113 int socket_accept_client(struct socket *self) {
114     int peerskt = accept(self->skt, NULL, NULL);
115     self->current_peerskt = peerskt;
116     return peerskt;
117 }
118
119
120 //+++++ UNICO ++++++
121
122 bool socket_connect_with_server(struct socket *self) {
123     int s = 0;
124     bool are_we_connected = false;
125     struct addrinfo *ptr;
126
127     for (ptr = self->result; ptr != NULL ^ are_we_connected == false; \
128          ptr = ptr->ai_next) {
129         self->skt = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
130         if (self->skt == -1) continue;
131         s = connect(self->skt, ptr->ai_addr, ptr->ai_addrlen);
132         are_we_connected = (s != -1);

```

abr 10, 19 14:32

common_socket.c

Page 3/3

```

133     }
134     self->current_peerskt = self->skt;
135
136     return are_we_connected;
137 }
138
139 void socket_disables_send_operations(struct socket *self) {
140     shutdown(self->skt, SHUT_WR);
141 }

```

abr 10, 19 14:32

client_socket.h

Page 1/1

```

1  #ifndef CLIENT_SOCKET_H
2  #define CLIENT_SOCKET_H
3  #include <stdlib.h>
4
5
6  struct client_socket {
7      char* host;
8      char* port;
9      int skt;
10     int current_peerskt;
11     struct addrinfo *result;
12 };
13
14 /*
15  Guarda los parametros que se necesitaran al inciar el socket
16  */
17 void client_socket_create(struct client_socket *self, 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 int client_socket_send_all(struct client_socket *self, \
25                          size_t size, char* buf);
26 //desabilita el canal de escritura
27 void client_socket_disables_send_operations(struct client_socket *self);
28 int client_socket_receive_some(struct client_socket *self, char* buf, \
29                              size_t size);
30
31
32 bool client_socket_connect(struct client_socket *self);
33 #endif

```

abr 10, 19 14:32

client_socket.c

Page 1/2

```

1  #define _POSIX_C_SOURCE 200112L
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #include <errno.h>
7  #include <stdbool.h>
8
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include <unistd.h>
13 #include "client_socket.h"
14
15 #define REQUEST_MAX_LEN 2000
16 #define RESPONSE_MAX_LEN 2000
17 #define LEN_HOST 13
18 #define LEN_PORT 6
19
20
21
22 int client_socket_receive_some(struct client_socket *self, char* buf, \
23                              size_t size) {
24     int i = recv(self->skt, buf, size, MSG_NOSIGNAL);
25     return i;
26 }
27
28
29 //***** IGUAL *****
30
31 void client_socket_create(struct client_socket *self, char* _host, \
32                          char* _port) {
33     self->host = _host;
34     self->port = _port;
35     self->current_peerskt = 0;
36 }
37
38 void client_socket_destroy(struct client_socket *self) {
39     freeaddrinfo(self->result);
40     shutdown(self->skt, SHUT_RDWR);
41     close(self->skt);
42 }
43
44 bool client_socket_start(struct client_socket *self) {
45     int s = 0;
46
47     struct addrinfo hints;
48
49     self->skt = 0;
50
51     memset(&hints, 0, sizeof(struct addrinfo));
52     hints.ai_family = AF_INET;
53     hints.ai_socktype = SOCK_STREAM;
54     hints.ai_flags = AI_PASSIVE;
55
56     s = getaddrinfo(self->host, self->port, &hints, &self->result);
57
58     if (s != 0) {
59         return false;
60     }
61     return true;
62 }
63
64 int client_socket_send_all(struct client_socket *self, \
65                          size_t size, char* buf) {

```

abr 10, 19 14:32

client_socket.c

Page 2/2

```

67     int bytes_sent = 0;
68     int s = 0;
69     bool is_the_socket_valid = true;
70
71
72     while (bytes_sent < size ^ is_the_socket_valid) {
73         s = send(self->skt, &buf[bytes_sent], \
74             size - bytes_sent, MSG_NOSIGNAL);
75         if (s ≤ 0) {
76             return -1;
77         } else {
78             bytes_sent += s;
79         }
80     }
81     return bytes_sent;
82 }
83
84 //+++++ UNICO ++++++
85
86 bool client_socket_connect(struct client_socket *self) {
87     int s = 0;
88     bool are_we_connected = false;
89     struct addrinfo *ptr;
90
91     for (ptr = self->result; ptr ≠ NULL ^ are_we_connected ≡ false; \
92         ptr = ptr->ai_next) {
93         self->skt = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
94         if (self->skt ≡ -1) continue;
95         s = connect(self->skt, ptr->ai_addr, ptr->ai_addrlen);
96         are_we_connected = (s ≠ -1);
97     }
98
99     return are_we_connected;
100 }
101
102 void client_socket_disables_send_operations(struct client_socket *self) {
103     shutdown(self->skt, SHUT_WR);
104 }

```

abr 10, 19 14:32

client_file_sender.h

Page 1/1

```

1  #ifndef CLIENT_FILE_SENDER_H
2  #define CLIENT_FILE_SENDER_H
3  #include <stdint.h>
4  #include "common_socket.h"
5
6  struct file_sender {
7      char* filename;
8      struct socket* socket;
9  };
10
11 void file_sender_create(struct file_sender* self, \
12     char* _filename, struct socket* _socket);
13
14 /*
15 abre el archivo cuyo nombre tiene almacenado como atributo
16 y envia su contenido a través de socket que tiene almacenado como atributo
17 */
18 bool file_sender_start(struct file_sender* self);
19
20 void file_sender_destroy(struct file_sender* self);
21
22 #endif

```

abr 10, 19 14:32

client_file_sender.c

Page 1/1

```

1 #define _POSIX_C_SOURCE 200809L //getline
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <errno.h>
6 #include <stdbool.h>
7 #include "client_file_sender.h"
8
9 void file_sender_create(struct file_sender* self,\
10     char* _filename, struct socket* _socket){
11     self->filename = _filename;
12     self->socket = _socket;
13 }
14
15 bool file_sender_start(struct file_sender* self) {
16     FILE* file = fopen(self->filename, "r");
17     if (!file) {
18         return false;
19     }
20
21     char* lineptr = NULL; size_t n = 0; size_t len;
22     while (true) {
23         len = getline(&lineptr, &n, file);
24         if (len == -1) break;
25         if ( socket_send_all(self->socket, len, lineptr) == -1 ) {
26             return false;
27         }
28     }
29
30     fclose(file);
31     free(lineptr);
32     return true;
33 }
34
35 void file_sender_destroy(struct file_sender* self) {
36     //do nothing
37 }
38

```

abr 10, 19 14:32

client.c

Page 1/2

```

1 #define _POSIX_C_SOURCE 200112L
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <errno.h>
6 #include <stdbool.h>
7
8 #include <sys/types.h>
9 #include <sys/socket.h>
10 #include <netdb.h>
11 #include <unistd.h>
12
13 #include "common_socket.h"
14
15 #include "client_file_sender.h"
16
17 #define MAX_LEN_FILENAME 100
18 #define MAX_LEN_BUF 2000
19
20 bool resive(struct socket* socket) {
21     int status = 0;
22     int received = 0;
23     char buf[MAX_LEN_BUF];
24
25     while ( true ) {
26         status = socket_receive_some(socket, &buf[received], \
27             MAX_LEN_BUF - received - 1);
28         if (status < 0) { //socket error
29             return false;
30         } else if (status == 0) {
31             break;
32         } else {
33             received = status;
34             buf[received] = 0;
35             printf("%s", buf);
36             received = 0;
37         }
38     }
39     printf("\n");
40     return true;
41 }
42
43
44 int main(int argc, char* argv[]) {
45     if (argc != 3 ^ argc != 4) {
46         fprintf(stderr, "Uso:\n./client <direccion> <puerto> [<input>]\n");
47         return 1;
48     }
49     char filename[MAX_LEN_FILENAME];
50     if (argc == 3) {
51         char* status = fgets(filename, MAX_LEN_FILENAME, stdin);
52         filename[strlen(filename) - 1] = '\0';
53         if (!status) {
54             return 1;
55         }
56     } else {
57         snprintf(filename, MAX_LEN_FILENAME, "%s", argv[3]);
58     }
59
60     struct file_sender fs;
61     char* host = argv[1];
62     char* port = argv[2];
63     struct socket socket;
64     socket_create(&socket, host, port);
65
66     if (!socket_start(&socket)) {

```

abr 10, 19 14:32client.cPage 2/2

```
67     socket_destroy(&socket);
68     return 1;
69 }
70 if (!socket_connect_with_server(&socket)) {
71     socket_destroy(&socket);
72     return 1;
73 }
74
75 int status = 0;
76 file_sender_create(&fs, filename, &socket);
77
78 if (!file_sender_start(&fs)){
79     status = 1;
80 }
81
82
83 socket_disables_send_operations(&socket);
84 if (!resive(&socket) ) {
85     status = 1;
86 }
87
88 socket_destroy(&socket);
89 file_sender_destroy(&fs);
90 return status;
91 }
```

abr 10, 19 14:32Table of ContentPage 1/1

1	Table of Contents					
2	1	server_tamplate.h...	sheets	1 to	1 (1) pages	1- 1 18 lines
3	2	server_tamplate.c...	sheets	1 to	1 (1) pages	2- 2 47 lines
4	3	server_socket.h....	sheets	2 to	2 (1) pages	3- 3 44 lines
5	4	server_socket.c....	sheets	2 to	3 (2) pages	4- 5 121 lines
6	5	server_sensor.h....	sheets	3 to	3 (1) pages	6- 6 24 lines
7	6	server_sensor.c....	sheets	4 to	4 (1) pages	7- 7 45 lines
8	7	server_request_processor.h	sheets	4 to	4 (1) pages	8- 8 36 lines
9	8	server_request_processor.c	sheets	5 to	5 (1) pages	9- 10 106 lines
10	9	server_list.h.....	sheets	6 to	6 (1) pages	11- 11 33 lines
11	10	server_list.c.....	sheets	6 to	7 (2) pages	12- 13 79 lines
12	11	server.c.....	sheets	7 to	8 (2) pages	14- 16 165 lines
13	12	common_socket.h....	sheets	9 to	9 (1) pages	17- 17 48 lines
14	13	common_socket.c....	sheets	9 to	10 (2) pages	18- 20 142 lines
15	14	client_socket.h....	sheets	11 to	11 (1) pages	21- 21 34 lines
16	15	client_socket.c....	sheets	11 to	12 (2) pages	22- 23 105 lines
17	16	client_file_sender.h	sheets	12 to	12 (1) pages	24- 24 23 lines
18	17	client_file_sender.c	sheets	13 to	13 (1) pages	25- 25 39 lines
19	18	client.c.....	sheets	13 to	14 (2) pages	26- 27 92 lines