sep 10, 19 15:20 **sudoku_matrix_loader.h** Page 1/1

```c
#ifndef _SUDOKU_MATRIX_LOADER_
#define _SUDOKU_MATRIX_LOADER_

int load_sudoku_matrix(int matrix[9][9]);

#endif
```

sep 10, 19 15:20 **sudoku_matrix_loader.c** Page 1/1

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

#define ERROR 1
#define SUCCESS 0
#define MATRIX_FILE_PATH "board.txt"
#define BUFFER_SIZE 19

#define MATRIX_LOAD_ERROR_MSG "Error while reading matrix data\n"

int load_file_in_matrix(FILE* file, int matrix[9][9]);
void load_line_in_matrix(const char* buffer, int* matrix_row);
FILE* open_file();

int load_sudoku_matrix(int matrix[9][9]) {
  FILE* matrix_file = open_file();
  if (¬matrix_file) {
    return ERROR;
  }
  if (load_file_in_matrix(matrix_file, matrix) ≡ ERROR) {
    return 1;
  }
  return 0;
}

int load_file_in_matrix(FILE* file, int matrix[9][9]){
  char buffer[BUFFER_SIZE];
  int i = 0;
  while (fgets(buffer, BUFFER_SIZE, file)) {
    load_line_in_matrix(buffer, matrix[i]);
    i++;
  }
  fclose(file);
  if (i < 8) {
    fprintf(stderr, MATRIX_LOAD_ERROR_MSG);
    return ERROR;
  }
  return SUCCESS;
}

FILE* open_file() {
  FILE* matrix_file = fopen(MATRIX_FILE_PATH, "r");
  if (¬matrix_file) {
    fprintf(stderr, MATRIX_LOAD_ERROR_MSG);
    return NULL;
  }
  return matrix_file;
}

void load_line_in_matrix(const char* buffer, int* matrix_row) {
  int i = 0;
  for (int j = 0; j ≤ 16; j+=2) {
    matrix_row[i] = buffer[j] − '0'; //ascii code fix
    i++;
  }
}
```

sep 10, 19 15:20 **sudoku_interface.h** Page 1/1

```c
#ifndef _SUDOKU_INTERFACE_H_
#define _SUDOKU_INTERFACE_H_

#include "client_interface.h"
#include "server_interface.h"

typedef struct {
  bool server_mode;
  client_interface_t client_interface;
  server_interface_t server_interface;
} sudoku_interface_t;

int sudoku_interface_init(sudoku_interface_t* sudoku_interface,
  int argc, char* argv[]);

int sudoku_interface_execute(sudoku_interface_t* sudoku_interface);

#endif
```

sep 10, 19 15:20 **sudoku_interface.c** Page 1/2

```c
#include "sudoku_interface.h"
#include <stdbool.h>
#include "string.h"

#include <stdio.h>

#define SUCCESS 0
#define ERROR 1
#define EXIT 2

int check_arguments(int argc, char* argv[]);
int init_mode_executed(sudoku_interface_t* sudoku_interface, char* argv[]);
bool first_argument_unsopported(char* arg);

#define UNSOPPORTED_MODE_MSG "Modo no soportado, el primer parÃ¡metro "\
                "debe ser server o client\n"

#define SV_MODE_WRONG_USE_MSG "Uso: ./tp server <puerto>\n"
#define CL_MODE_WRONG_USE_MSG "Uso: ./tp client <host> <puerto>\n"

int sudoku_interface_init(sudoku_interface_t* sudoku_interface,
  int argc, char* argv[]){
  if (check_arguments(argc, argv) == ERROR) {
    return ERROR;
  }
  if (init_mode_executed(sudoku_interface, argv) == ERROR) {
    return ERROR;
  }
  return SUCCESS;
}


int init_mode_executed(sudoku_interface_t* sudoku_interface, char* argv[]) {
  if (strcmp(argv[1], "server") == 0) {
    if (server_interface_init(&sudoku_interface->server_interface,
      argv[2]) == ERROR) {
      return ERROR;
    }
    sudoku_interface->server_mode = true;
  } else {
    if (client_interface_init(&sudoku_interface->client_interface,
      argv[2], argv[3]) == ERROR) {
      return ERROR;
    }
    sudoku_interface->server_mode = false;
  }
  return SUCCESS;
}


int sudoku_interface_execute(sudoku_interface_t* sudoku_interface) {
  if (sudoku_interface->server_mode == true) {
    if (server_interface_process(&sudoku_interface->server_interface) == ERROR) {
      return ERROR;
    }
  } else {
    int state = client_interface_process(&sudoku_interface->client_interface);
    if (state == ERROR) {
      return ERROR;
    } else if (state == EXIT) {
      return EXIT;
    }
  }
  return SUCCESS;
}

```

sep 10, 19 15:20 **sudoku_interface.c** Page 2/2

```
67  int check_arguments(int argc, char* argv[]) {
68    if (argc < 2 ∨ first_argument_unsopported(argv[1])) {
69      fprintf(stderr, UNSOPPORTED_MODE_MSG);
70      return ERROR;
71    }
72    if ((strcmp(argv[1], "server") ≡ 0) ∧ (argc ≠ 3)) {
73      fprintf(stderr, SV_MODE_WRONG_USE_MSG);
74      return ERROR;
75    }
76    if ((strcmp(argv[1], "client") ≡ 0) ∧ (argc ≠ 4)) {
77      fprintf(stderr, CL_MODE_WRONG_USE_MSG);
78      return ERROR;
79    }
80    return SUCCESS;
81  }
82
83  bool first_argument_unsopported(char* arg) {
84    int sv_cmp = strcmp(arg, "server");
85    int cl_cmp = strcmp(arg, "client");
86    if (sv_cmp ≠ 0 ∧ cl_cmp ≠ 0) {
87      return true;
88    }
89    return false;
90  }
```

sep 10, 19 15:20 **sudoku_board.h** Page 1/1

```
1  #ifndef _SUDOKU_BOARD_H_
2  #define _SUDOKU_BOARD_H_
3
4  #include "cell.h"
5
6  typedef struct{
7    cell_t cells[9][9];
8  } sudoku_board_t;
9
10 void sudoku_board_init(sudoku_board_t* board,
11   int matrix[9][9]);
12
13 int sudoku_board_put(sudoku_board_t* board,
14   int number, int row, int column);
15
16 int sudoku_board_verify(sudoku_board_t* board);
17
18 void sudoku_board_restart(sudoku_board_t* board);
19
20 void sudoku_board_get(sudoku_board_t* board,
21   int matrix[9][9]);
22
23 #endif
```

sep 10, 19 15:20 **sudoku_board.c** Page 1/2

```c
#include "sudoku_board.h"
#include <stdbool.h>
#include <stdio.h>

bool row_is_valid(sudoku_board_t* board, int row_index);
bool column_is_valid(sudoku_board_t* board, int column_index);
bool sector_is_valid(sudoku_board_t* board, int sector_number);
void put_sector_cells_in_array(sudoku_board_t* board,
  int sector_number, cell_t* array);

void sudoku_board_init(sudoku_board_t* board, int matrix[9][9]){
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      int numb = matrix[i][j];
      if (numb ≡ 0) {
        cell_init(&board→cells[i][j], numb, true);
      } else {
        cell_init(&board→cells[i][j], numb, false);
      }
    }
  }
}


int sudoku_board_put(sudoku_board_t* board, int number,
  int row, int column) {
  cell_t* cell = &board→cells[row − 1][column − 1];
  if (cell_set_number(cell, number) ≡ 1) {
    return 1;
  }
  return 0;
}

int sudoku_board_verify(sudoku_board_t* board) {
  int i;
  for (i = 0; i < 9; i++) {
    if (¬row_is_valid(board, i) ∨ ¬column_is_valid(board, i)) {
      return 1;
    }
  }
  for (i = 0; i < 3; i++) {
    if (¬sector_is_valid(board, i)) {
      return 1;
    }
  }
  return 0;
}

bool row_is_valid(sudoku_board_t* board, int row_index) {
  int j;
  for (int i = 0; i < 9; i++) {
        for (j = i + 1; j < 9; j++) {
          if (¬cell_is_valid(&board→cells[row_index][i],
            &board→cells[row_index][j])) {
            return false;
          }
        }
    }
    return true;
}

bool column_is_valid(sudoku_board_t* board, int column_index) {
  int j;
  for (int i = 0; i < 9; i++) {
        for (j = i + 1; j < 9; j++) {
          if (¬cell_is_valid(&board→cells[i][column_index],
            &board→cells[j][column_index])) {
            return false;
          }
        }
    }
    return true;
}

bool sector_is_valid(sudoku_board_t* board, int sector_number) {
  int i, j;
  cell_t aux[9];
  put_sector_cells_in_array(board, sector_number, aux);
  for (i = 0; i < 9; i++) {
        for (j = i + 1; j < 9; j++) {
          if (¬cell_is_valid(&aux[i], &aux[j])) {
            return false;
          }
        }
    }
    return true;
}

void put_sector_cells_in_array(sudoku_board_t* board,
  int sector_number, cell_t* array) {
  int idx = sector_number * 3;
  int final_idx = idx + 3;
  int actual_pos = 0;
  for (int i = idx; i < final_idx; i++) {
    for (int j = idx; j < final_idx; j++) {
      array[actual_pos] = board→cells[i][j];
      actual_pos++;
    }
  }
}

void sudoku_board_restart(sudoku_board_t* board) {
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      cell_restart(&board→cells[i][j]);
    }
  }
}

void sudoku_board_get(sudoku_board_t* board, int matrix[9][9]) {
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      matrix[i][j] = cell_get_number(&board→cells[i][j]);
    }
  }
}
```

sep 10, 19 15:20 **sudoku_board.c** Page 2/2

sep 10, 19 15:20 **socket.h** Page 1/1

```c
#ifndef _SOCKET_H_
#define _SOCKET_H_

typedef struct socket {
  int fd;
} socket_t;

void socket_init(socket_t* socket);

void socket_release(socket_t* socket);

int socket_connect(socket_t* socket, const char* host, const char* service);

int socket_bind_and_listen(socket_t* socket, const char* service,
  int listen_amount);

int socket_accept_client(socket_t* sv_skt, socket_t* peer_skt);

int socket_recv_message(socket_t* socket, char *buf, int size);

int socket_send_message(socket_t* socket, char *buf, int size);

#endif
```

sep 10, 19 15:20 **socket.c** Page 1/3

```c
#include "socket.h"
#define _POSIX_C_SOURCE 200112L

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

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

//forward declarations
void socket_addr_iterate(socket_t* skt, struct addrinfo* result,
  bool* connection_established);

int socket_getaddrinfo(struct addrinfo **result, const char* host,
  const char* service, bool pasive);

int socket_bind(int sockfd, struct addrinfo* ptr);
int socket_listen(int sockfd, int listen_amount);


void socket_init(socket_t* socket) {
  socket→fd = -1; //initialize to invalid fd
}

void socket_release(socket_t* skt) {
  shutdown(skt→fd, SHUT_RDWR);
  close(skt→fd);
}

int socket_connect(socket_t* skt, const char* host, const char* service) {
  struct addrinfo *result = NULL;
  int s = socket_getaddrinfo(&result, host, service, false);
  if (s ≠ 0) {
      fprintf(stderr, "Error in getaddrinfo: %s\n", gai_strerror(s));
      freeaddrinfo(result);
      return 1;
    }
    bool connection_established = false;
    socket_addr_iterate(skt, result, &connection_established);
    if (¬connection_established) {
      fprintf(stderr, "Error: connection couldn't been established\n");
      freeaddrinfo(result);
      return 1;
    }
  freeaddrinfo(result);
  return 0;
}

int socket_bind_and_listen(socket_t* skt, const char* service,
  int listen_amount) {
  struct addrinfo *ptr = NULL;
  int s = socket_getaddrinfo(&ptr, NULL, service, true);
  if (s ≠ 0) {
      fprintf(stderr, "Error in getaddrinfo: %s\n", gai_strerror(s));
      return 1;
    }
    skt→fd = socket(ptr→ai_family, ptr→ai_socktype, ptr→ai_protocol);
  if (skt→fd ≡ -1) {
      fprintf(stderr, "Error: %s\n", strerror(errno));
      freeaddrinfo(ptr);
      return 1;
  }
```

```
67    if (socket_bind(skt→fd, ptr) ≡ 1) {
68      freeaddrinfo(ptr);
69      return 1;
70    }
71    freeaddrinfo(ptr);
72    if (socket_listen(skt→fd, listen_amount) ≡ 1) {
73      return 1;
74    }
75    return 0;
76  }
77
78  int socket_accept_client(socket_t* sv_skt, socket_t* peer_skt) {
79    peer_skt→fd = accept(sv_skt→fd, NULL, NULL);
80    if (peer_skt→fd ≡ -1){
81      fprintf(stderr, "Error: %s\n", strerror(errno));
82      return 1;
83    }
84    return 0;
85  }
86
87  int socket_recv_message(socket_t* skt, char *buf, int size){
88    int received = 0;
89    int s = 0;
90    while (received < size) {
91      s = recv(skt→fd, &buf[received], size-received, MSG_NOSIGNAL);
92      if (s ≡ 0 ∨ s ≡ -1) { //socket was closed or error occurred
93        return -1;
94      } else {
95          received += s;
96      }
97    }
98    return received;
99  }
100
101 int socket_send_message(socket_t* skt, char *buf, int size){
102   int sent = 0;
103   int s = 0;
104   while (sent < size) {
105     s = send(skt→fd, &buf[sent], size-sent, MSG_NOSIGNAL);
106     if (s ≡ 0 ∨ s ≡ -1) { //socket was closed or error occurred
107       return -1;
108     } else {
109       sent += s;
110     }
111   }
112   return sent;
113 }
114
115 void socket_addr_iterate(socket_t* skt, struct addrinfo* result,
116   bool* connection_established) {
117   struct addrinfo* ptr;
118   int s;
119   for (ptr = result; ptr ≠ NULL ∧ *connection_established ≡ false;
120     ptr = ptr→ai_next) {
121     skt→fd = socket(ptr→ai_family, ptr→ai_socktype, ptr→ai_protocol);
122     if (skt→fd ≡ -1) {
123       fprintf(stderr, "Error: %s\n", strerror(errno));
124     } else {
125       s = connect(skt→fd, ptr→ai_addr, ptr→ai_addrlen);
126           if (s ≡ -1) {
127               fprintf(stderr, "Error: %s\n", strerror(errno));
128               close(skt→fd);
129           }
130       *connection_established = (s ≠ -1); //are we connected now?
131     }
132   }
```



```
133 }
134
135 int socket_getaddrinfo(struct addrinfo **addrinfo_ptr, const char* host,
136   const char* service, bool passive) {
137   struct addrinfo hints;
138   memset(&hints, 0, sizeof(struct addrinfo));
139   hints.ai_family = AF_INET;
140   hints.ai_socktype = SOCK_STREAM;
141
142   if (passive){
143     hints.ai_flags = AI_PASSIVE;
144     return getaddrinfo(NULL, service, &hints, addrinfo_ptr);
145   } else {
146     hints.ai_flags = 0;
147     return getaddrinfo(host, service, &hints, addrinfo_ptr);
148   }
149 }
150
151 int socket_bind(int sockfd, struct addrinfo* ptr) {
152   int s = bind(sockfd, ptr→ai_addr, ptr→ai_addrlen);
153     if (s ≡ -1) {
154       fprintf(stderr, "Error: %s\n", strerror(errno));
155       close(sockfd);
156       return 1;
157   }
158   return 0;
159 }
160
161 int socket_listen(int sockfd, int listen_amount) {
162   int s = listen(sockfd, listen_amount);
163   if (s ≡ -1) {
164     fprintf(stderr, "Error: %s\n", strerror(errno));
165       close(sockfd);
166       return 1;
167   }
168   return 0;
169 }
```

sep 10, 19 15:20 **server_socket.h** Page 1/1

```c
#ifndef _SERVER_H_
#define _SERVER_H_

#include "socket.h"

typedef struct server {
  socket_t sv_skt;
  socket_t peer_skt;
} server_socket_t;

int server_socket_init(server_socket_t* self, const char* port);

int server_socket_accept_client(server_socket_t* self);

int server_socket_recv_message(server_socket_t* self, char* buf, int size);

int server_socket_send_message(server_socket_t* self, char* buf, int size);

void server_socket_release(server_socket_t* self);

#endif
```

sep 10, 19 15:20 **server_socket.c** Page 1/1

```c
#include "server_socket.h"
#define BACKLOG 1

int server_socket_init(server_socket_t* self, const char* port) {
  socket_init(&self→sv_skt);
  socket_init(&self→peer_skt);
  return socket_bind_and_listen(&self→sv_skt, port, BACKLOG);
}

int server_socket_accept_client(server_socket_t* self) {
  return socket_accept_client(&self→sv_skt, &self→peer_skt);
}

int server_socket_recv_message(server_socket_t* self, char* buf, int size) {
  return (socket_recv_message(&self→peer_skt, buf, size) ≡ −1);
}

int server_socket_send_message(server_socket_t* self, char* buf, int size) {
  return (socket_send_message(&self→peer_skt, buf, size) ≡ −1);
}

void server_socket_release(server_socket_t* self) {
  socket_release(&self→sv_skt);
  socket_release(&self→peer_skt);
}
```

sep 10, 19 15:20 **server_protocol.h** Page 1/1

```c
#ifndef _SERVER_PROTOCOL_H
#define _SERVER_PROTOCOL_H

#include "sudoku_board.h"
#include "server_socket.h"

typedef struct {
  server_socket_t* skt;
  sudoku_board_t* board;
} server_protocol_t;

void server_protocol_init(server_protocol_t* protocol,
  server_socket_t* skt, sudoku_board_t* board);

int server_protocol_process(server_protocol_t* protocol);

#endif
```

sep 10, 19 15:20 **server_protocol.c** Page 1/3

```c
#include "server_protocol.h"
#include <string.h>
#include <stdint.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <stdio.h>
#include "board_representation_maker.h"

#define PUT_ROW_BUFFER_IDX 0
#define PUT_COL_BUFFER_IDX 1
#define PUT_NUMB_BUFFER_IDX 2

#define PUT_COMMAND_MES 'P'
#define VERIFY_COMMAND_MES 'V'
#define RESET_COMMAND_MES 'R'
#define GET_COMMAND_MES 'G'

#define VALID_BOARD_MSG "OK\n"
#define INVALID_BOARD_MSG "ERROR\n"
#define UNMODIFIABLE_CELL_MSG "La celda indicada no es modificable\nâM−^@M−^K"

//forward declarations

uint32_t calculate_str_len(char* str);

int process_message(server_protocol_t* protocol, char message);

int process_p_message(server_protocol_t* protocol);
int process_g_message(server_protocol_t* protocol);
int process_r_message(server_protocol_t* protocol);
int process_v_message(server_protocol_t* protocol);

int show_board_to_client(server_protocol_t* protocol);

int send_message_to_client(server_protocol_t* protocol, char* mes);
int send_unmodifiable_cell_message(server_protocol_t* protocol);
int send_modifiable_cell_message(server_protocol_t* protocol);
int send_invalid_board_message(server_protocol_t* protocol);
int send_valid_board_message(server_protocol_t* protocol);

void server_protocol_init(server_protocol_t* protocol,
  server_socket_t* skt, sudoku_board_t* board) {
  protocol→skt = skt;
  protocol→board = board;
}

int server_protocol_process(server_protocol_t* protocol) {
  char buffer;
  if (server_socket_recv_message(protocol→skt, &buffer, 1)) {
    return 1;
  }
  if (process_message(protocol, buffer)) {
    return 1;
  }
  return 0;
}

int process_message(server_protocol_t* protocol, char message) {
  if (message ≡ PUT_COMMAND_MES) {
    return process_p_message(protocol);

  } else if (message ≡ GET_COMMAND_MES) {
    return process_g_message(protocol);

  } else if (message ≡ RESET_COMMAND_MES) {
    return process_r_message(protocol);
```

```
67
68    } else {
69      return process_v_message(protocol);
70    }
71  }
72
73
74  int process_p_message(server_protocol_t* protocol) {
75    char buffer[3];
76    if (server_socket_recv_message(protocol→skt, buffer, 3)) {
77      return 1;
78    }
79    int numb = buffer[PUT_NUMB_BUFFER_IDX] - '0';
80    int row = buffer[PUT_ROW_BUFFER_IDX] - '0';
81    int col = buffer[PUT_COL_BUFFER_IDX] - '0';
82
83    int unmodifiable_cell = sudoku_board_put(protocol→board, numb, row, col);
84    if (unmodifiable_cell) {
85      return send_unmodifiable_cell_message(protocol);
86    } else {
87      return send_modifiable_cell_message(protocol);
88    }
89  }
90
91  int send_unmodifiable_cell_message(server_protocol_t* protocol) {
92    if (send_message_to_client(protocol, UNMODIFIABLE_CELL_MSG) ≡ 1) {
93      return 1;
94    }
95    return 0;
96  }
97
98  int send_modifiable_cell_message(server_protocol_t* protocol) {
99    return show_board_to_client(protocol);
100 }
101
102 int process_g_message(server_protocol_t* protocol) {
103   return show_board_to_client(protocol);
104 }
105
106 int process_r_message(server_protocol_t* protocol) {
107   sudoku_board_restart(protocol→board);
108   return show_board_to_client(protocol);
109 }
110
111 int process_v_message(server_protocol_t* protocol) {
112   if (sudoku_board_verify(protocol→board)) {
113     return send_invalid_board_message(protocol);
114   }
115   return send_valid_board_message(protocol);
116 }
117
118 int send_invalid_board_message(server_protocol_t* protocol) {
119   if (send_message_to_client(protocol, INVALID_BOARD_MSG) ≡ 1) {
120     return 1;
121   }
122   return 0;
123 }
124
125 int send_valid_board_message(server_protocol_t* protocol) {
126   if (send_message_to_client(protocol, VALID_BOARD_MSG) ≡ 1) {
127     return 1;
128   }
129   return 0;
130 }
131
132 int show_board_to_client(server_protocol_t* protocol) {
```



```
133   int matrix[9][9];
134   sudoku_board_get(protocol→board, matrix);
135   char* board_representation = assemble_board_representation(matrix);
136   int error = send_message_to_client(protocol, board_representation);
137   free(board_representation);
138   return error;
139 }
140
141 int send_message_to_client(server_protocol_t* protocol, char* msg) {
142   uint32_t len = calculate_str_len(msg);
143   uint32_t len_ton = htonl(len);
144   if (server_socket_send_message(protocol→skt, (char*)&len_ton, 4)) {
145     return 1;
146   }
147   for (int i = 0; i < len; i++) {
148     if (server_socket_send_message(protocol→skt, &msg[i], 1)) {
149       return 1;
150     }
151   }
152   return 0;
153 }
154
155 uint32_t calculate_str_len(char* str) {
156   uint32_t i = 0;
157   while (str[i] ≠ '\0') {
158     i++;
159   }
160   return i;
161 }
```

sep 10, 19 15:20 **server_interface.h** Page 1/1

```c
#ifndef _SERVER_INTERFACE_H_
#define _SERVER_INTERFACE_H_

#include "server_protocol.h"
#include "server_socket.h"
#include "sudoku_board.h"

typedef struct {
  sudoku_board_t board;
  server_socket_t skt;
  server_protocol_t protocol;
} server_interface_t;

int server_interface_init(server_interface_t* self, const char* port);

int server_interface_process(server_interface_t* self);

#endif
```

sep 10, 19 15:20 **server_interface.c** Page 1/1

```c
#include "server_interface.h"
#include "sudoku_matrix_loader.h"

#define ERROR 1
#define SUCCESS 0

int server_interface_init(server_interface_t* self, const char* port) {
  int matrix[9][9];
  if (load_sudoku_matrix(matrix) == ERROR) {
    return ERROR;
  }
  sudoku_board_init(&self->board, matrix);
  if (server_socket_init(&self->skt, port) == ERROR) {
    return ERROR;
  }
  if (server_socket_accept_client(&self->skt) == ERROR) {
    server_socket_release(&self->skt);
    return ERROR;
  }
  server_protocol_init(&self->protocol, &self->skt, &self->board);
  return SUCCESS;
}

int server_interface_process(server_interface_t* self) {
  if (server_protocol_process(&self->protocol)) {
    server_socket_release(&self->skt);
    return ERROR;
  }
  return SUCCESS;
}
```

sep 10, 19 15:20 **protocol_message_maker.h** Page 1/1

```c
#ifndef _PROTOCOL_MESSAGE_MAKER_H_
#define _PROTOCOL_MESSAGE_MAKER_H_

char* build_protocol_message(char* command);

#endif
```

sep 10, 19 15:20 **protocol_message_maker.c** Page 1/2

```c
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "protocol_message_maker.h"

#define RESET_COMMAND_MES 'R'
#define PUT_COMMAND_MES 'P'
#define VERIFY_COMMAND_MES 'V'
#define GET_COMMAND_MES 'G'

#define GET_COMMAND "get"
#define RESET_COMMAND "reset"
#define VERIFY_COMMAND "verify"
#define PUT_COMMAND "put"

#define PUT_COMMAND_NUMB_POS 4
#define PUT_COMMAND_IDX_A_POS 9
#define PUT_COMMAND_IDX_B_POS 11

char* build_get_message();
char* build_reset_message();
char* build_verify_message();
char* build_put_message(char* command);
char* init_message(int len);

char* build_protocol_message(char* command) {
  if (strcmp(GET_COMMAND, command) == 0) {
    return build_get_message(command);

  } else if (strcmp(RESET_COMMAND, command) == 0) {
    return build_reset_message(command);

  } else if (strcmp(VERIFY_COMMAND, command) == 0) {
    return build_verify_message(command);

  } else {
    return build_put_message(command);
  }
}

char* build_get_message() {
  char* message = init_message(1);
  message[0] = GET_COMMAND_MES;
  return message;
}

char* build_reset_message() {
  char* message = init_message(1);
  message[0] = RESET_COMMAND_MES;
  return message;
}

char* build_verify_message() {
  char* message = init_message(1);
  message[0] = VERIFY_COMMAND_MES;
  return message;
}

char* build_put_message(char* command) { //command format: "put x in y,z"
  char* message = init_message(4);
  message[0] = PUT_COMMAND_MES;
  message[1] = command[PUT_COMMAND_IDX_A_POS];
  message[2] = command[PUT_COMMAND_IDX_B_POS];
  message[3] = command[PUT_COMMAND_NUMB_POS];
  return message;
}
```

sep 10, 19 15:20 **protocol_message_maker.c** Page 2/2

```
67
68  char* init_message(int len) {
69    char* mes = malloc(sizeof(char)*len + 1);
70    mes[len] = '\0';
71    return mes;
72  }
```

sep 10, 19 15:20 **main.c** Page 1/1

```
1   #include "sudoku_interface.h"
2
3   #define SUCCESS 0
4   #define ERROR 1
5   #define EXIT 2
6
7   int main(int argc, char* argv[]){
8     sudoku_interface_t sudoku_interface;
9
10    int game_init = sudoku_interface_init(&sudoku_interface, argc, argv);
11    if (game_init == ERROR) {
12      return 1;
13    }
14
15    int execution;
16    while(1) {
17      execution = sudoku_interface_execute(&sudoku_interface);
18      if (execution == EXIT) {
19        return SUCCESS;
20      } else if (execution == ERROR) {
21        return ERROR;
22      }
23    }
24    return 0;
25  }
```

sep 10, 19 15:20 **client_socket.h** Page 1/1

```c
#ifndef _CLIENT_SOCKET_H_
#define _CLIENT_SOCKET_H_

#include "socket.h"

typedef struct client {
  socket_t skt;
} client_socket_t;

int client_socket_init(client_socket_t* self, const char* host,
  const char* service);

int client_socket_recv_message(client_socket_t* self, char* buf, int size);

int client_socket_send_message(client_socket_t* self, char* buf, int size);

void client_socket_release(client_socket_t* self);

#endif
```

sep 10, 19 15:20 **client_socket.c** Page 1/1

```c
#include "client_socket.h"

int client_socket_init(client_socket_t* self, const char* host,
  const char* service) {
  socket_init(&self→skt);
  return socket_connect(&self→skt, host, service);
}

int client_socket_recv_message(client_socket_t* self, char* buf, int size) {
  return (socket_recv_message(&self→skt, buf, size) ≡ −1);
}

int client_socket_send_message(client_socket_t* self, char* buf, int size) {
  return (socket_send_message(&self→skt, buf, size) ≡ −1);
}

void client_socket_release(client_socket_t* self) {
  socket_release(&self→skt);
}
```

sep 10, 19 15:20 **client_protocol.h** Page 1/1

```c
#ifndef _CLIENT_PROTOCOL_H_
#define _CLIENT_PROTOCOL_H_

#include "client_socket.h"

typedef struct {
  client_socket_t* skt;
} client_protocol_t;

void client_protocol_init(client_protocol_t* protocol, client_socket_t* skt);

int client_protocol_send_message(client_protocol_t* protocol, char* command);

char* client_protocol_recv_answer(client_protocol_t* protocol);

#endif
```

sep 10, 19 15:20 **client_protocol.c** Page 1/1

```c
#include "client_protocol.h"
#include "protocol_message_maker.h"
#include <stdio.h>
#include <arpa/inet.h>
#include "string.h"
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>

void client_protocol_init(client_protocol_t* protocol, client_socket_t* skt) {
  protocol→skt = skt;
}

char* client_protocol_recv_answer(client_protocol_t* protocol) {
  char len_buffer[4];
  if (client_socket_recv_message(protocol→skt, len_buffer, 4)) {
    return NULL;
  }
  uint32_t text_len;
  memcpy(&text_len, len_buffer, 4);
  text_len = ntohl(text_len);
  char* text_buffer = malloc(sizeof(char)*text_len + 1);
  text_buffer[text_len] = '\0';

  if (client_socket_recv_message(protocol→skt, text_buffer, text_len)) {
    free(text_buffer);
    return NULL;
  }
  return text_buffer;
}

int client_protocol_send_message(client_protocol_t* protocol, char* command) {
  char* message = build_protocol_message(command);
  if (client_socket_send_message(protocol→skt, message, strlen(message))) {
    free(message);
    return 1;
  }
  free(message);
  return 0;
}
```

sep 10, 19 15:20 **client_interface.h** Page 1/1

```c
#ifndef _CLIENT_INTERFACE_H
#define _CLIENT_INTERFACE_H

#include "client_socket.h"
#include "client_protocol.h"

typedef struct {
  client_socket_t skt;
  client_protocol_t protocol;
} client_interface_t;

int client_interface_init(client_interface_t* client_interface,
  const char* host, const char* service);

int client_interface_process(client_interface_t* client_interface);

#endif
```

sep 10, 19 15:20 **client_interface.c** Page 1/3

```c
#include "client_interface.h"
#include "client_protocol.h"
#include <string.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define EXIT_COMMAND "exit"
#define PUT_COMMAND "put"

#define FGETS_SIZE 25
#define COMMAND_MAX_SIZE 15

#define SUCCESS 0
#define ERROR 1
#define EXIT 2
#define INVALID_COMMAND 3

#define INDEX_ERROR_MES "Error en los Ã-ndices. Rango soportado: [1,9]"
#define VALUE_ERROR_MES "Error en el valor ingresado. Rango soportado: [1,9]"

int process_user_input(char* input);
int execute_command(client_protocol_t* protocol, char* command);
bool command_has_valid_indexes(char* input);
bool command_has_valid_values(char* input);
void get_command_first_arg(char* buffer, char* input);
bool index_is_allowed(char* index);
bool value_is_allowed(char* value);

int client_interface_init(client_interface_t* client_interface,
  const char* host, const char* service) {
  if (client_socket_init(&client_interface→skt, host, service)) {
    return ERROR;
  }
  client_protocol_init(&client_interface→protocol, &client_interface→skt);
  return SUCCESS;
}

int client_interface_process(client_interface_t* client_interface) {
  char buffer[FGETS_SIZE];
  int input_state = process_user_input(buffer);
  if (input_state ≡ EOF) {
    client_socket_release(&client_interface→skt);
    return ERROR;
  }
  if (input_state ≡ INVALID_COMMAND) {
    return SUCCESS;
  }
  if (strcmp(buffer, EXIT_COMMAND) ≡ 0) {
    client_socket_release(&client_interface→skt);
    return EXIT;
  }
  if (execute_command(&client_interface→protocol, buffer)) {
    client_socket_release(&client_interface→skt);
    return ERROR;
  }
  return SUCCESS;
}

int execute_command(client_protocol_t* protocol, char* command) {
  int command_exec = client_protocol_send_message(protocol, command);
  if (command_exec ≡ ERROR) {
    return ERROR;
  }
  char* sv_ans = client_protocol_recv_answer(protocol);
  if (¬sv_ans) {
```

```
67      return ERROR;
68    }
69    printf("%s", sv_ans);
70    free(sv_ans);
71    return SUCCESS;
72  }
73
74  int process_user_input(char* input) {
75    if (¬fgets(input, FGETS_SIZE, stdin)) {
76      return EOF;
77    }
78    input[strlen(input) − 1] = '\0'; //replace '\n' with '\0'
79    if (¬command_has_valid_indexes(input)) {
80      fprintf(stderr, "%s\n" , INDEX_ERROR_MES);
81      return INVALID_COMMAND;
82    } else if (¬command_has_valid_values(input)) {
83      fprintf(stderr, "%s\n", VALUE_ERROR_MES);
84      return INVALID_COMMAND;
85    }
86    return SUCCESS;
87  }
88
89  bool command_has_valid_indexes(char* input) {
90    char command_first_arg[COMMAND_MAX_SIZE];
91    get_command_first_arg(command_first_arg, input);
92    if (strcmp(command_first_arg, PUT_COMMAND) ≡ 0) {
93      char index_a[4], index_b[4];
94      index_a[3] = '\0';
95      index_b[3] = '\0';
96      sscanf(input, "%*s %*s %*s %3[^,]%*[, ]%3s", index_a, index_b);
97      if (¬index_is_allowed(index_a) ∨ ¬index_is_allowed(index_b)) {
98        return false;
99      }
100   }
101   return true;
102 }
103
104 bool command_has_valid_values(char* input) {
105   char command_first_arg[COMMAND_MAX_SIZE];
106   get_command_first_arg(command_first_arg, input);
107   if (strcmp(command_first_arg, PUT_COMMAND) ≡ 0) {
108     char value[4];
109     value[3] = '\0';
110     sscanf(input, "%*s %3s", value);
111     if (¬value_is_allowed(value)) {
112       return false;
113     }
114   }
115   return true;
116 }
117
118 void get_command_first_arg(char* buffer, char* input) {
119   sscanf(input, "%14s", buffer);
120   buffer[COMMAND_MAX_SIZE − 1] = '\0';
121 }
122
123 bool index_is_allowed(char* index) {
124   int idx = atoi(index);
125   if (idx > 9 ∨ idx < 1) {
126     return false;
127   }
128   return true;
129 }
130
131 bool value_is_allowed(char* value) {
132   int v = atoi(value);
```



```
133   if (v > 9 ∨ v < 1) {
134     return false;
135   }
136   return true;
137 }
```

sep 10, 19 15:20 **cell.h** Page 1/1

```c
#ifndef _CELL_H_
#define _CELL_H_

#include <stdbool.h>

typedef struct cell {
  bool modifiable;
  int number;
} cell_t;

void cell_init(cell_t* cell, int number, bool modifiable);

int cell_set_number(cell_t* cell, int numb);

int cell_get_number(cell_t* cell);

bool cell_is_valid(cell_t* cell_a, cell_t* cell_b);

void cell_restart(cell_t* cell);

#endif
```

sep 10, 19 15:20 **cell.c** Page 1/1

```c
#include "cell.h"

void cell_init(cell_t* cell, int number, bool modifiable) {
  cell→modifiable = modifiable;
  cell→number = number;
}

int cell_set_number(cell_t* cell, int numb) {
  if (¬cell→modifiable) {
    return 1;
  }
  cell→number = numb;
  return 0;
}

int cell_get_number(cell_t* cell) {
  return cell→number;
}

bool cell_is_valid(cell_t* cell_a, cell_t* cell_b) {
  return (cell_a→number ≠ cell_b→number ∨ cell_a→number ≡ 0);
}


void cell_restart(cell_t* cell) {
  if (cell→modifiable) {
    cell→number = 0;
  }
}
```

sep 10, 19 15:20 **board_representation_maker.h** Page 1/1

```h
#ifndef _BOARD_REPRESENTATION_MAKER_
#define _BOARD_REPRESENTATION_MAKER_

char* assemble_board_representation(int matrix[9][9]);

#endif
```

sep 10, 19 15:20 **board_representation_maker.c** Page 1/2

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define BOARD_REPR_SIZE 723
#define BOARD_REPR_ROWS 19

void add_sector_separator(char* buffer);
void add_normal_separator(char* buffer);
void add_matrix_row(char* buffer, int matrix[9][9], int row_index);
void iterate_row_representation(char* row_representation, int* row, int len);

char* assemble_board_representation(int matrix[9][9]) {
  char* board_representation = malloc(sizeof(char)*BOARD_REPR_SIZE);
  int matrix_row_index = 0;
  for (int i = 0; i < BOARD_REPR_ROWS; i++) {
    int actual_index = i*38;
    if (i % 6 ≡ 0) {
      add_sector_separator(&board_representation[actual_index]);
    } else if (i % 2 ≡ 0) {
      add_normal_separator(&board_representation[actual_index]);
    } else {
      add_matrix_row(&board_representation[actual_index],
        matrix, matrix_row_index);
      matrix_row_index += 1;
    }
  }
  board_representation[BOARD_REPR_SIZE-1] = '\0';
  return board_representation;
}

void add_sector_separator(char* buffer) {
  char sector_separator[] = "U==========U==========U==========U\n";
  size_t len = strlen(sector_separator);
  for (int i = 0; i < len; i++) {
    buffer[i] = sector_separator[i];
  }
}

void add_normal_separator(char* buffer) {
  char normal_separator[] = "U---+---+---U---+---+---U---+---+---U\n";
  size_t len = strlen(normal_separator);
  for (int i = 0; i < len; i++) {
    buffer[i] = normal_separator[i];
  }
}

void add_matrix_row(char* buffer, int matrix[9][9], int row_index) {
  char row_representation[] = "U | | U | | U | | U\n";
  size_t len = strlen(row_representation);
  iterate_row_representation(row_representation, matrix[row_index], len);
  for (int i = 0; i < len; i++) {
    buffer[i] = row_representation[i];
  }
}


void iterate_row_representation(char* row_representation, int* row, int len) {
  int i = 2;
  int j = 0;
  char aux[2];
  while (i < len) {
    if (row[j] ≠ 0) {
      snprintf(aux, sizeof(aux)/sizeof(char), "%d", row[j]);
      row_representation[i] = aux[0];
    }
    i += 4;
```

```
67      j++;
68    }
69  }
```


Table of Contents