75:42 - Taller de Programación I

Ejercicio Nº ___ Padrón _____

Alumno _____ Firma _____

Nota:		**Corrige**:		**Entrega #1**
				Fecha de entrega
				Fecha de devolución

Nota:		**Corrige**:		**Entrega #2**
				Fecha de entrega
				Fecha de devolución

El presente trabajo, así como la entrega electrónica correspondiente al mismo, constituyen una obra de creación completamente personal, no habiendo sido inspirada ni siendo copia completa o parcial de ninguna fuente pública, privada, de otra persona o naturaleza.

Índice

1. Objetivos

Se desea implementar el milenario *Sudoku* con una conexión cliente-servidor que se pueda jugar mediante líneas de comandos. Para ello se hará uso de los conceptos vistos en clase de TDA's, Sockets y otros conceptos referidos a `C`.

2. Resolución del problema

La ejecución básica del algoritmo implementado se ilustra en la figura 1.

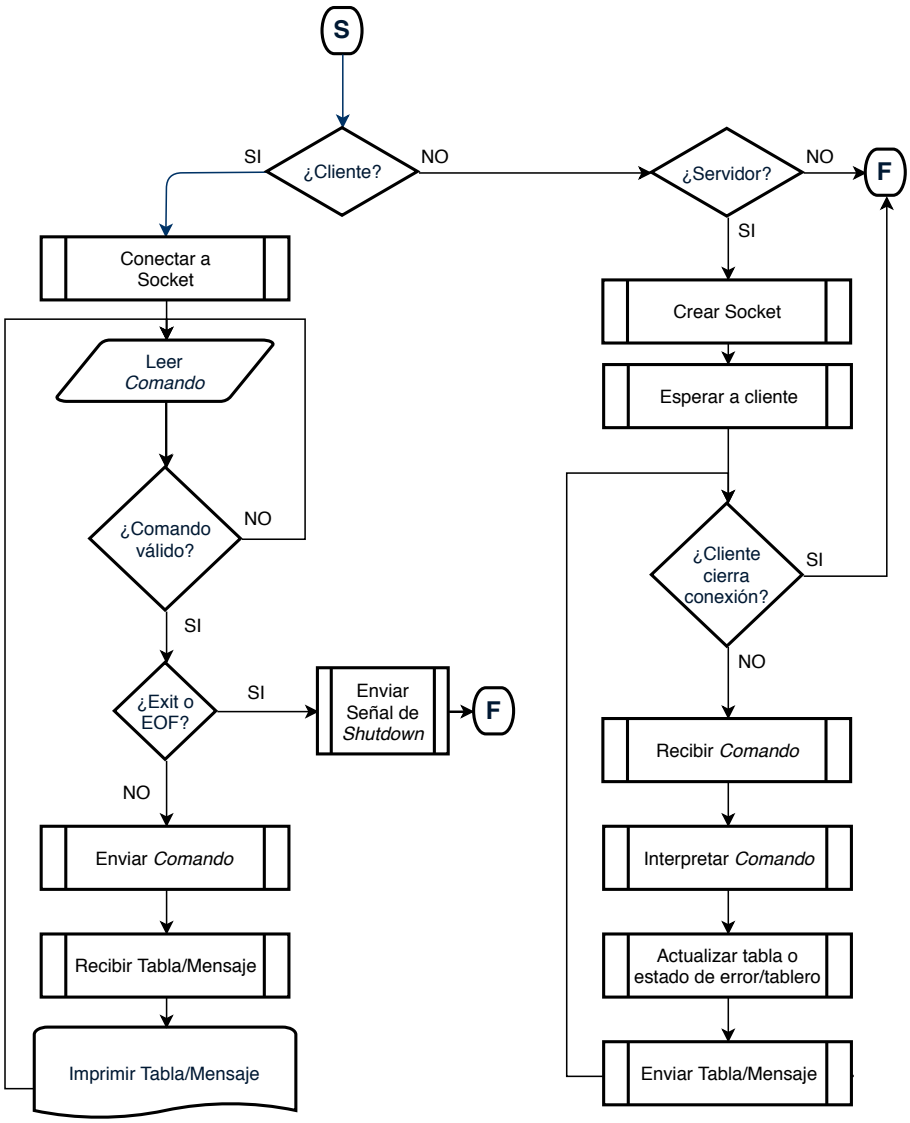

Figura 1: Diagrama del algoritmo.

3. TDA's implementados

- **Board:** Contiene el tablero en uso y una copia del tablero original. De esta forma se evita reacceder al archivo siendo el tamaño del tablero muy pequeño y conveniente para tener en memoria. El tablero se guarda sin darle el formato. Sólo se formatea para enviar.
- **Client:** Contiene las instrucciones necesarias para leer un comando e interpretarlo. Envía comandos. Recibe mensajes/tablas del servidor. Se conecta al servidor.
- **Server:** Contiene las instrucciones necesarias para recibir un comando y procesarlo. Leer un tablero de un archivo y cargarlo en *Board*. Envía el tablero. Crea un *Socket* y espera un usuario. Formatea la tabla.
- **Protocol:** Implementa el protocolo especificado en los requerimientos. Envía y Recibe comandos (1 o 4 bytes). Envía y recibe cadenas (primero envía el largo del string y luego el string).
- **Socket:** Implementa los algoritmos de conexión entre cliente y servidor. Implementa los algoritmos de en envío y recepción de n bytes. Crea un socket (servidor), se conecta a un socket(cliente->servidor), acepta un socket (servidor), etc.
- **Sudoku:** Contiene la lógica general del programa. Determina si el programa se ejecutará como servidor o cliente. Ejecuta las normas del cliente.

4. Sistemas y software utilizados

- WSL 18.04 (Windows Subsystem for Linux).
- Ubuntu 18.04 nativo en otra computadora.
- SERCOM.
- `gdb`
- `Valgrind`
- VSCode

5. Problemas encontrados y observaciones

Los siguientes items refieren a la ejecución del programa utilizando los archivos de entrada y scripts de SERCOM tanto localmente como en SERCOM.

1. **Diferencia entre SERCOM y pruebas locales.** Se encontró un problema en el que el servidor enviaba 4 caracteres más de lo debido en SERCOM mientras que en las pruebas locales no. Esto se debía a una comparación `strcmp(aux,"P")` la cual fué reemplazada por `aux[0] == 'P'` para verificar el comando recibido.

2. **Corridas no finalizadas en SERCOM.** Se debió a validaciones no correctas de `EOF`, lo cual provocaba un loop al utilizar un archivo como entrada estándar.

3. **Problemas de memoria.** Valgrind encontró múltiples errores debido a valores sin inicializar:
   `Conditional jump or move depends on uninitialised value(s)`
   Esto se solventó inicializando los elementos correspondientes con `memset`.

4. **Layout de la tabla.** Para formatear la tabla se consideró utilizar un `#define` de líneas múltiples pero SERCOM no permite esta implementación por lo cual se optó por imprimir múltiples líneas mediante `snprintf()` . Otra posible implementación sería mediante un algoritmo particular que introduzca los valores y el formato al mismo tiempo. También podría guardarse la tabla ya con formato consumiendo un poco más de memoria a cambio de menor uso de CPU ya que no necesita formatear la tabla en cada envío.

5. **Checkeo de reglas del juego.** Para checkear las reglas del juego se utilizaron operadores de bits. Se utilizaron 2 bytes donde las últimas 9 posiciones representaban los dígitos del 0 al 9. En cada fila se reinicia el byte (a 0) y se coloca un 1 en caso de encontrar un número. Si se da que en esa posición ya había un 1, las reglas no se cumplen. De la misma forma se verifican columnas y sectores.

6. **Consideraciones varias.** *TDA Sudoku* podría contener un atributo `Board_t` para guardar el tablero. Se utilizaron tipos enumerativos por lo cual sería prudente implementar funciones de impresión de mensajes de error.

6. Conclusiones

Si bien la implementación realizada posee margen de mejora, se realizó un primer acercamiento al uso de *Sockets* para comprender los conceptos de envío y recepción básicos y se puso en práctica el uso de TDA's en C. En implementaciones posteriores se desea profundizar la división de responsabilidades en ciertos TDA's y mejorar la legibilidad del código en varios aspectos.

sep 10, 19 3:19 **sudoku.h** Page 1/1

```
#ifndef SUDOKUH
#define SUDOKUH

#include "protocol.h"
#include "client.h"
#include "server.h"
#include "socket.h"
#include "board.h"

#define MAX_COMMAND_LENGTH 20
#define RECEIVED_BOARD 800

#define SUDOKU_ERR_SERVER_CMD "Uso: ./tp server <puerto>\n"
#define SUDOKU_ERR_CLIENT_CMD "Uso: ./tp client <host> <puerto>\n"
#define SUDOKU_ERR_CMD "Modo no soportado, el primer parÃ¡metro debe ser "
#define SUDOKU_ERR_CMD_OPTIONS "server o client\n"
#define SUDOKU_MSG_SERVER "server"
#define SUDOKU_MSG_CLIENT "client"

typedef enum{
    SUDOKU_OK,
    SUDOKU_ARGS_OK,
    SUDOKU_ARGS_INVALID,
    SUDOKU_ARGS_NULL,
    SUDOKU_ENDGAME
} sudoku_status_t;

typedef struct{
    char mode;
    char * arg1;
    char * arg2;
    sudoku_status_t status;
} sudoku_t;

void sudoku_set_mode(sudoku_t *self, int argc, char *argv[]);
void sudoku_run(sudoku_t *self);
void sudoku_run_server(sudoku_t *self, server_t *, socket_t *, board_t *);
void sudoku_play(sudoku_t *self, client_t *, socket_t *, board_t *);

#endif
```

sep 10, 19 3:19 **sudoku.c** Page 1/2

```
#include "sudoku.h"

//Esta funciÃ³n valida los argumentos de cmdline y setea el modo de juego.
void sudoku_set_mode(sudoku_t *self, int argc, char *argv[]){
    self→status = SUDOKU_ARGS_INVALID;
    if (argv ≡ NULL)
        self→status = SUDOKU_ARGS_NULL;

    if (argc ≡ 3 ∧ ¬strcmp(argv[1],SUDOKU_MSG_SERVER)){
        self→mode = 'S';
        self→arg1 = argv[2];
        self→status = SUDOKU_ARGS_OK;
    } else if (argc ≡ 4 ∧ ¬strcmp(argv[1],SUDOKU_MSG_CLIENT)){
        self→mode = 'C';
        self→arg1 = argv[2];
        self→arg2 = argv[3];
        self→status = SUDOKU_ARGS_OK;
    } else if (argc ≡ 2 ∧ ¬strcmp(argv[1],SUDOKU_MSG_SERVER)){
        printf("%s", SUDOKU_ERR_SERVER_CMD);
    } else if (argc ≡ 2 ∧ ¬strcmp(argv[1],SUDOKU_MSG_CLIENT)){
        printf("%s", SUDOKU_ERR_CLIENT_CMD);
    } else {
        printf("%s%s", SUDOKU_ERR_CMD, SUDOKU_ERR_CMD_OPTIONS);
    }
}

void sudoku_run(sudoku_t *self){
    socket_t socket;
    board_t board;
    client_t client;
    server_t server;
    client.row = 0;
    client.col = 0;
    client.value = 0;
    memset(board.board, 0, sizeof(board.board));
    memset(board.board_original, 0, sizeof(board.board_original));

    switch ( self→mode ){
        case 'S':
            sudoku_run_server(self, &server, &socket, &board);
            break;
        case 'C':
            sudoku_play(self, &client, &socket, &board);
            break;
    }
}

//Esta funciÃ³n ejecuta el servidor para jugar al sudoku.
void sudoku_run_server(sudoku_t *self,
                       server_t *server,
                       socket_t *socket,
                       board_t *board){
    server_wait_user(server, socket, self→arg1);
    if ( socket→status ≡ SKT_CONNECTED ){
        server_board_read(server, board);
        server_receive_command(server, socket);
        while ( socket→status ≡ SKT_RECEPTION_SUCCESS ){
            server_process_command(server, board, socket);
            server_receive_command(server, socket);
        }
    }
    if (socket→status ≡ SKT_RECEPTION_END)
        self→status = SUDOKU_ENDGAME;
    server_destroy(server, socket);
}

```

sep 10, 19 3:19 **sudoku.c** Page 2/2

```
67  //Esta funciÃ³n ejecuta el cliente para jugar al sudoku.
68  void sudoku_play(sudoku_t *self,
69                       client_t *client,
70                       socket_t *socket,
71                       board_t *board){
72      char received[RECEIVED_BOARD];
73      char client_command[MAX_COMMAND_LENGTH];
74      client→command = 'X';
75
76      memset(received, 0, RECEIVED_BOARD);
77      memset(client_command, 1, MAX_COMMAND_LENGTH);
78
79      client_connect(client, socket, self→arg1, self→arg2);
80      if ( client→status ≡ CLIENT_CONNECTED ){
81          while ( client→status ≠ CLIENT_ENDGAME ){
82              client→status = CLIENT_COMMAND_INVALID;
83              while (client→status ≡ CLIENT_COMMAND_INVALID){
84                  client→status = CLIENT_OK;
85                  client_read_command(client, client_command);
86                  client_command_process(client, client_command);
87              }
88              if (client→command ≡ 'E' ∨ client→status ≡ CLIENT_EOF_REACHED){
89                  client→status = CLIENT_ENDGAME;
90                  self→status = SUDOKU_ENDGAME;
91                  skt_destroy(socket);
92              }
93              if (client→status ≡ CLIENT_COMMAND_VALID){
94                  client_send_command(client, socket);
95                  client_receive_string(client, socket, received);
96              }
97          }
98      }
99  }
```

sep 10, 19 3:19 **socket.h** Page 1/1

```
1   #ifndef SOCKETH
2   #define SOCKETH
3
4   #include <stdio.h>
5   #include <string.h>
6   #include <stdlib.h>
7   #include <stdbool.h>
8
9   #include <sys/socket.h>
10  #include <unistd.h>
11  #include <sys/types.h>
12  #include <netdb.h>
13
14  #define WAITING_CLIENTS 1
15  #define MSG_NEW_CLIENT "Nuevo cliente"
16
17  typedef enum {
18      SKT_CONNECTION_FAILURE,
19      SKT_CREATE_FAILURE,
20      SKT_RECEPTION_FAILURE,
21      SKT_RECEPTION_SUCCESS,
22      SKT_RECEPTION_END,
23      SKT_SEND_FAILURE,
24      SKT_SEND_SUCCESS,
25      SKT_CONNECTED,
26      SKT_SHUTDOWN
27  } socket_status_t;
28
29  typedef struct {
30      int skt;
31      socket_status_t status;
32  } socket_t;
33
34  typedef struct addrinfo addrinfo_t;
35
36  void skt_connect(socket_t *self, char *, char *);
37  void skt_create(socket_t *self, char *);
38  void skt_destroy(socket_t *self);
39  void skt_accept(socket_t *self);
40  void skt_send(socket_t *self, int, char *);
41  void skt_listen(socket_t *self);
42  void skt_bind(socket_t *self, addrinfo_t *);
43  void skt_setsockoptions(socket_t *self, addrinfo_t *);
44  int skt_receive(socket_t *self, int, char *);
45
46  #endif
```

```c
#define _POSIX_C_SOURCE 200112L

#include "socket.h"

//Esta funciÃ³n conecta el cliente al servidor
void skt_connect(socket_t *self, char *hostname, char *port){
   int status; //para resoluciÃ³n de direcciÃ³n
   struct addrinfo hints; //configuraciÃ³n conexiÃ³n
   struct addrinfo *res; //lista de estructuras de direcciones

   self→status = SKT_CONNECTED;

   //configuro cliente
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_family = AF_INET; //IPv4
   hints.ai_socktype = SOCK_STREAM; //TCP
   hints.ai_flags = 0; //server

   status = getaddrinfo(hostname, port, &hints, &res);
   if (status ≠ 0)
      self→status = SKT_CONNECTION_FAILURE;

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

   if (self→skt ≡ -1)
      self→status = SKT_CONNECTION_FAILURE;

   status = connect(self→skt, res→ai_addr, res→ai_addrlen);
   if (status ≡ -1){
      close(self→skt);
      self→status = SKT_CONNECTION_FAILURE;
   }

   freeaddrinfo(res);
}

//Esta funciÃ³n crea un servidor
void skt_create(socket_t *self, char *port){
   int c = 1;
   int status; //para resoluciÃ³n de direcciÃ³n
   struct addrinfo hints; //configuraciÃ³n conexiÃ³n
   struct addrinfo *res; //lista de estructuras de direcciones

   //configuro servidor
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_family = AF_INET; //IPv4
   hints.ai_socktype = SOCK_STREAM; //TCP
   hints.ai_flags = AI_PASSIVE; //server
   status = getaddrinfo(NULL, port, &hints, &res);

   if (status ≠ 0)
      self→status = SKT_CREATE_FAILURE;
   self→skt = socket(res→ai_family, res→ai_socktype, res→ai_protocol);
   if (self→skt ≡ -1){
      freeaddrinfo(res); //libera info de lista de direcciones
      self→status = SKT_CREATE_FAILURE;
   }
   status = setsockopt(self→skt, SOL_SOCKET, SO_REUSEADDR, &c, sizeof(&c));
   if (status ≡ -1){
      self→status = SKT_CREATE_FAILURE;
   }
   skt_bind(self, res);
   skt_listen(self);
   freeaddrinfo(res);
}

```



```c
//Configura cantidad de clientes en espera
void skt_listen(socket_t *self){
   int status;

   status = listen(self→skt, WAITING_CLIENTS);
   if (status ≡ -1){
      close(self→skt);
      self→status = SKT_CREATE_FAILURE;
   }
}

//Configuro puerto para que no sea aleatorio
void skt_bind(socket_t *self, addrinfo_t *res){
   int status;
   status = bind(self→skt, res→ai_addr, res→ai_addrlen);
   if (status ≡ -1){
      close(self→skt);
      freeaddrinfo(res); //libera info de lista de direcciones
      self→status = SKT_CREATE_FAILURE;
   }
}

//Esta funciÃ³n espera un cliente y lo acepta si es correcto
void skt_accept(socket_t *self){
   self→skt = accept(self→skt, NULL, NULL);   // aceptamos un cliente
   self→status = SKT_CONNECTED;
   printf("%s\n", MSG_NEW_CLIENT);
}

void skt_destroy(socket_t *self){
   self→status = SKT_SHUTDOWN;
   shutdown(self→skt, SHUT_RDWR);
   close(self→skt);
}

//Esta funciÃ³n recibe un string de largo len en el buffer buf.
int skt_receive(socket_t *self, int len, char *buf){
   bool socket_running = true;
   int bytes_received = 0;
   int status = 0;
   while (socket_running ∧ bytes_received < len) {
      status = recv(self→skt, &buf[bytes_received], len-bytes_received, 0);
      if (status ≡ 0) {
         socket_running = false;
      } else if (status ≡ -1) {
         socket_running = false;
      } else {
         bytes_received = bytes_received + status;
      }
   }
   if ( bytes_received > 0 ) {
      self→status = SKT_RECEPTION_SUCCESS;
      return bytes_received;
   } else if (bytes_received ≡ 0){
      self→status = SKT_RECEPTION_END;
      return -1;
   } else {
      self→status = SKT_RECEPTION_FAILURE;
      return -1;
   }
}

//Esta funciÃ³n envÃ-a un string de largo len en el buffer buf.
void skt_send(socket_t *self, int len, char *buf){
   bool socket_running = true;
   int bytes_sent = 0;
```

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

```
133     int status = 0;
134
135     while (socket_running ∧ bytes_sent < len) {
136        status = send(self→skt, &buf[bytes_sent], len-bytes_sent, 0);
137        if (status ≡ 0) {
138           socket_running = false;
139        } else if (status ≡ -1) {
140           socket_running = false;
141        } else {
142           bytes_sent = bytes_sent + status;
143        }
144     }
145     if (socket_running) {
146        self→status = SKT_SEND_SUCCESS;
147     } else {
148        self→status = SKT_SEND_FAILURE;
149     }
150 }
```

sep 10, 19 3:19 **server.h** Page 1/1

```
1  #ifndef SERVERH
2  #define SERVERH
3
4  #include <stdbool.h>
5  #include "socket.h"
6  #include "board.h"
7  #include "protocol.h"
8
9  #define BOARD_SENT_SIZE 723
10 #define BOARD_ROW_SEP_CELL "U===========U===========U===========U\n"
11 #define BOARD_ROW_SEP "U---+---+---U---+---+---U---+---+---U\n"
12 #define BOARD_ROW "U | | U | | U | | U\n"
13 #define BOARD_COL_SEP_CELL "U"
14 #define BOARD_COL_SEP "|"
15 #define CH_PER_ROW 38
16 #define ROWS 19
17
18 #define MSG_NON_MODIFIABLE_CELL "La celda indicada no es modificable\n"
19 #define MSG_ERROR_VERIFY "ERROR\n"
20 #define MSG_OK_VERIFY "OK\n"
21 #define MSG_INVALID_CMD "Se ingresÃ³ un comando invalido "
22 #define MSG_FAILED_CONNECTION "Conexion fallida\n"
23 #define MSG_CLIENT_CONNECTED "Cliente conectado en puerto "
24
25 typedef enum{
26     SERVER_OK,
27     SERVER_COMMAND_INVALID,
28     SERVER_DISCONNECTED_CLIENT
29 } server_status_t;
30
31 typedef struct{
32     char command;
33     uint8_t row;
34     uint8_t col;
35     uint8_t value;
36     server_status_t status;
37 } server_t;
38
39 void server_send_board(server_t *self, socket_t *, board_t *);
40 void server_receive_command(server_t *self, socket_t *);
41 void server_process_command(server_t *self, board_t *, socket_t *);
42 void server_board_read(server_t *self, board_t *board);
43 void server_wait_user(server_t *self, socket_t *skt, char *port);
44 void server_destroy(server_t *server, socket_t *skt);
45 void server_format_board(char *board);
46
47 #endif
```

sep 10, 19 3:19 **server.c** Page 1/2

```c
#include "protocol.h"
#include "server.h"

//Esta funciÃ³n envÃa la tabla formateada al cliente.
void server_send_board(server_t *self, socket_t *socket, board_t *board){
    char * sb = (char*)malloc(BOARD_SENT_SIZE * sizeof(char));//tabla a enviar
    int j = 0;
    int k = 1;
    char aux;

    printf("\n");
    server_format_board(sb);

    for (int i=40; i≤680; i=i+4){
        j++;
        if ( (aux = board_value_get(board, k, j)) ≠ '0' )
            sb[i] = aux;

        //Para el salto de fila.
        if (j≡9){
            j=0;
            k++;
            i=i+2+CH_PER_ROW;
        }
    }
    fprintf(stdout, "%s",sb);

    protocol_send_string(socket, sb);
    free(sb);
}

void server_format_board(char *board){
    snprintf(board, BOARD_SENT_SIZE,
             "%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s",
             BOARD_ROW_SEP_CELL,
             BOARD_ROW,
             BOARD_ROW_SEP,
             BOARD_ROW,
             BOARD_ROW_SEP,
             BOARD_ROW,
             BOARD_ROW_SEP_CELL,
             BOARD_ROW,
             BOARD_ROW_SEP,
             BOARD_ROW,
             BOARD_ROW_SEP,
             BOARD_ROW,
             BOARD_ROW_SEP_CELL,
             BOARD_ROW,
             BOARD_ROW_SEP,
             BOARD_ROW,
             BOARD_ROW_SEP,
             BOARD_ROW,
             BOARD_ROW_SEP_CELL);
}

void server_process_command(server_t *self, board_t *board, socket_t *skt){
    switch (self→command)
    {
        case 'P':
            board_put(board, self→row, self→col, self→value);
            if (board→status ≡ BOARD_NON_MODIFIABLE_VALUE){
                protocol_send_string(skt, MSG_NON_MODIFIABLE_CELL);
                board→status = BOARD_OK;
            } else {
                server_send_board(self, skt, board);
            }
            break;
        case 'V':
            board_verify(board);
            if (board→status ≡ BOARD_INVALID){
                protocol_send_string(skt, MSG_ERROR_VERIFY);
            } else {
                protocol_send_string(skt, MSG_OK_VERIFY);
            }
            break;
        case 'R':
            board_reset(board);
            server_send_board(self, skt, board);
            break;
        case 'G':
            server_send_board(self, skt, board);
            break;
        case 'E':
            self→status = SERVER_DISCONNECTED_CLIENT;
        default:
            protocol_send_string(skt, MSG_INVALID_CMD);
            break;
    }
}

void server_receive_command(server_t *self, socket_t *skt){
    protocol_receive_command(skt, self);
}

void server_board_read(server_t *self, board_t *board){
    board_read(board);
    board→status = BOARD_OK;
}

void server_wait_user(server_t *self, socket_t *skt, char *port){
    skt_create(skt, port);
    skt_accept(skt);
    if (skt→status ≠ SKT_CONNECTED){
        printf("%s", MSG_FAILED_CONNECTION);
    } else {
        printf("%s%s\n", MSG_CLIENT_CONNECTED, port);
    }
}

void server_destroy(server_t *server, socket_t *skt){
   skt→status = SKT_SHUTDOWN;
   close(skt→skt);
}
```

sep 10, 19 3:19 **protocol.h** Page 1/1

```c
#ifndef PROTOCOLH
#define PROTOCOLH

#define COMMAND_SIZE 2
#define COMMAND_PAMS_SIZE 5
#define HEADER_LEN 4

#include "socket.h"
#include "client.h"
#include "server.h"

typedef enum {
    PROTOCOL_OK,
    PROTOCOL_ERROR
} protocol_status_t;

typedef struct {
    char command[COMMAND_SIZE];
    char command_pams[COMMAND_PAMS_SIZE];
    protocol_status_t status;
} protocol_t;

void protocol_send_command(socket_t*, client_t*);
void protocol_receive_command(socket_t*, server_t*);
void protocol_send_string(socket_t *, char*);
void protocol_receive_string(socket_t *, char*);

#endif
```

sep 10, 19 3:19 **protocol.c** Page 1/2

```c
#include "protocol.h"

void protocol_send_command(socket_t *socket, client_t *command){
    char aux = (char)(command→command);

    skt_send(socket, 1, &aux);
    if (aux ≡ 'P'){
        aux = (char)(command→row);
        skt_send(socket, 1, &aux);
        aux = (char)(command→col);
        skt_send(socket, 1, &aux);
        aux = (char)(command→value);
        skt_send(socket, 1, &aux);
    }
}

void protocol_receive_command(socket_t *socket, server_t *command){
    char aux[10] = "00";
    skt_receive(socket, 1, aux);
    if (socket→status≡SKT_RECEPTION_SUCCESS){
        command→command = aux[0];
        if ( aux[0] ≡ 'P' ) {
            command→command = aux[0];
            skt_receive(socket, 1, aux);
            command→row = (uint8_t)aux[0];
            skt_receive(socket, 1, aux);
            command→col = (uint8_t)aux[0];
            skt_receive(socket, 1, aux);
            command→value = (uint8_t)aux[0];
        }
    }
}

void protocol_send_string(socket_t *self, char *command){
    char message_len[HEADER_LEN];
    int len = 0;

    len = strlen(command);
    if (len < 10){
        snprintf(message_len, sizeof(message_len), "00%d", len);
    } else if (len <100 ∧ len≥10) {
        snprintf(message_len, sizeof(message_len), "%0d", len);
    } else {
        snprintf(message_len, sizeof(message_len), "%d", len);
    }
    skt_send(self, HEADER_LEN−1, message_len);
    skt_send(self, len, command);
}

void protocol_receive_string(socket_t *self, char *received){
    int len = 0;
    char header[HEADER_LEN];

    //primero recibe un header con cuantos bytes voy a recibir
    memset(header, 0, HEADER_LEN);
    skt_receive(self, HEADER_LEN−1, header);

    //esta es la cantidad de bytes de mi string a recibir
    len = atoi(header);
    if (len ≡ 0) {
        self→status = SKT_RECEPTION_FAILURE;
    } else {
        //ahora recibo lo necesario
        skt_receive(self, len, received);
        received[len−1] = '\0';
        printf("%s\n", received);
```

```
67        }
68  }
```



```
1   #define SUCCESS 0
2   #define FAILURE 1
3
4   #include <stdio.h>
5   #include <string.h>
6   #include <stdlib.h>
7
8   #include "protocol.h"
9   #include "client.h"
10  #include "server.h"
11  #include "socket.h"
12  #include "board.h"
13  #include "sudoku.h"
14
15  int main(int argc, char *argv[]){
16    sudoku_t game;
17
18    sudoku_set_mode(&game, argc, argv);
19    if (game.status ≡ SUDOKU_ARGS_OK)
20      sudoku_run(&game);
21    if (game.status ≠ SUDOKU_ENDGAME)
22      return FAILURE;
23    return SUCCESS;
24  }
```

sep 10, 19 3:19 **client.h** Page 1/1

```c
#ifndef CLIENTH
#define CLIENTH

#include "socket.h"
#include "protocol.h"

#define ERROR_INDEX  "Error en los Ã-ndices. Rango soportado: [1,9]\n"
#define ERROR_INVALID_PUT_COMMAND  "Sintaxis: put <numero> in <fila>,<columna>"
#define ERROR_INPUT_VAL  "Error en el valor ingresado. Rango soportado: [1,9]\n"
#define ERROR_INVALID_COMMAND  "Ingrese:\n-put\n-verify\n-get\n-reset\n-exit\n"
#define DELIMITER " "

#define MAX_INPUT 20
#define READ_BUFF_MAX 50

typedef enum{
    CLIENT_OK,
    CLIENT_COMMAND_VALID,
    CLIENT_COMMAND_INVALID,
    CLIENT_ENDGAME,
    CLIENT_CONNECTED,
    CLIENT_EOF_REACHED,
    CLIENT_INVALID_INDEX
} client_status_t;

typedef struct{
    char command;
    uint8_t row;
    uint8_t col;
    uint8_t value;
    client_status_t status;
} client_t;

void client_read_command(client_t *self, char *command);
void client_command_process(client_t *self, char*);
void client_receive_string(client_t *self, socket_t*, char*);
void client_send_command(client_t*, socket_t*);
void client_connect(client_t *self, socket_t *skt, char*, char*);

#endif
```

sep 10, 19 3:19 **client.c** Page 1/2

```c
#define _POSIX_C_SOURCE 200112L

#include <ctype.h>
#include <string.h>
#include "protocol.h"
#include "client.h"

void client_read_command(client_t *self, char *command){
    int c;

    c = fgetc(stdin);
    if (c ≡ EOF){
        self→status = CLIENT_EOF_REACHED;
    } else {
        ungetc(c, stdin);
        if (fgets(command, READ_BUFF_MAX, stdin) ≡ NULL)
            self→status = CLIENT_ENDGAME;
    }
}

void client_receive_string(client_t *self, socket_t * skt, char *board){
    protocol_receive_string(skt, board);
}

void client_send_command(client_t *self, socket_t * skt){
    protocol_send_command(skt, self);
}

/*Esta funciÃ³n se asegura de validar correctamente la entrada por
teclado para que los comandos se encuentren correctamente
restringidos y debido a eso posee una considerable extensiÃ³n.*/
void client_command_process(client_t *self, char *command){
    char * partial_command;
    char * str_ptr;
    char delim[] = DELIMITER;
    char aux_command[MAX_INPUT];
    memset(aux_command,1,MAX_INPUT);

    if (self→status ≠ CLIENT_EOF_REACHED){
        snprintf(aux_command, sizeof(aux_command), "%s", command);
        partial_command = strtok_r(aux_command, delim, &str_ptr);
        if (¬strcmp(partial_command,"put")){
            self→command = 'P';
            partial_command = strtok_r(str_ptr, delim, &str_ptr);
            if (strlen(partial_command)≡1 ∧ isdigit(partial_command[0])){
                self→value = (uint8_t)partial_command[0] - 48;
                partial_command = strtok_r(str_ptr, delim, &str_ptr);
                if (¬strcmp(partial_command,"in")){
                    partial_command = strtok_r(str_ptr, delim, &str_ptr);
                    if (strlen(partial_command)≡4  ∧
                        isdigit(partial_command[0]) ∧
                        partial_command[0]≠'0'      ∧
                        partial_command[1]≡','      ∧
                        isdigit(partial_command[2]) ∧
                        partial_command[2]≠'0'      ∧
                        partial_command[3]≡'\n'){
                            self→row = (uint8_t)partial_command[0] - 48;
                            self→col = (uint8_t)partial_command[2] - 48;
                            self→status = CLIENT_COMMAND_VALID;
                        } else {
                            fprintf(stderr, "%s", ERROR_INDEX);
                        }
                } else {
                    fprintf(stderr, "%s", ERROR_INVALID_PUT_COMMAND);
                    self→status = CLIENT_COMMAND_INVALID;
                }
```

sep 10, 19 3:19 **client.c** Page 2/2

```
67              } else {
68                  self→status = CLIENT_COMMAND_INVALID;
69                  fprintf(stderr, "%s", ERROR_INPUT_VAL);
70              }
71          } else if (¬strcmp(command,"verify\n")) {
72              self→command = 'V';
73              self→status = CLIENT_COMMAND_VALID;
74          } else if (¬strcmp(command,"get\n")) {
75              self→command = 'G';
76              self→status = CLIENT_COMMAND_VALID;
77          } else if (¬strcmp(command,"reset\n")) {
78              self→command = 'R';
79              self→status = CLIENT_COMMAND_VALID;
80          } else if (¬strcmp(command,"exit\n")) {
81              self→command = 'E';
82              self→status = CLIENT_COMMAND_VALID;
83          } else if (command[0] ≡ EOF) {
84              self→command = 'E';
85              self→status = CLIENT_COMMAND_VALID;
86          } else {
87              fprintf(stderr, "%s", ERROR_INVALID_COMMAND);
88              self→status = CLIENT_COMMAND_INVALID;
89          }
90      }
91  }
92
93  void client_connect(client_t *self, socket_t *skt, char* host, char* port){
94      skt_connect(skt, host, port);
95      if (skt→status ≡ SKT_CONNECTED)
96          self→status = CLIENT_CONNECTED;
97  }
```

sep 10, 19 3:19 **board.h** Page 1/1

```
1   #ifndef BOARDH
2   #define BOARDH
3
4   #define BOARD_SIZE 200
5   #define BOARD_SENT_SIZE 723
6   #define CHARS_PER_ROW 18
7
8   #define BOARD_FILE "board.txt"
9   #define MSG_BOARD_RESTART "Tablero reiniciado."
10
11  typedef enum {
12      BOARD_OK,
13      BOARD_NON_MODIFIABLE_VALUE,
14      BOARD_FILE_ERROR,
15      BOARD_INVALID
16  } board_status_t;
17
18  typedef struct {
19      char board[BOARD_SIZE];
20      char board_original[BOARD_SIZE];
21      board_status_t status;
22  } board_t;
23
24  void board_read(board_t *self);
25  void board_verify(board_t *self);
26  void board_row_verify(board_t *self);
27  void board_col_verify(board_t *self);
28  void board_cell_verify(board_t *self);
29  void board_put(board_t *self, uint8_t i, uint8_t j, uint8_t value);
30  void board_reset(board_t *self);
31  void board_send(board_t *self);
32  char board_value_get(board_t *self, uint8_t i, uint8_t j);
33  char board_value_get_original(board_t *self, uint8_t i, uint8_t j);
34  void board_value_set(board_t *self, uint8_t i, uint8_t j, uint8_t value);
35
36  #endif
```

```c
#include "protocol.h"
#include "client.h"
#include "server.h"
#include "socket.h"
#include "board.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <stdbool.h>

void board_read(board_t *self){
    FILE * pboard;
    char c;
    int i = 0;

    if ( (pboard = fopen(BOARD_FILE,"r")) ≡ NULL ){
        self→status = BOARD_FILE_ERROR;
    } else {
        while ( (c = fgetc(pboard)) ≠ EOF ){
            self→board[i] = c;
            self→board_original[i] = c;
            i++;
        }
        self→status = BOARD_OK;
        fclose(pboard);
    }
}

char board_value_get(board_t *self, uint8_t i, uint8_t j){
    uint8_t tmp = 0;

    while (j > 1){
        tmp = tmp + 2;
        j--;
    }

    return self→board[CHARS_PER_ROW*(i-1) + tmp];
}

char board_value_get_original(board_t *self, uint8_t i, uint8_t j){
    uint8_t tmp = 0;

    while (j > 1){
        tmp = tmp + 2;
        j--;
    }

    return self→board_original[CHARS_PER_ROW*(i-1) + tmp ];
}

void board_value_set(board_t *self, uint8_t i, uint8_t j, uint8_t value){
    uint8_t tmp = 0;

    while (j > 1){
        tmp = tmp + 2;
        j--;
    }

    self→board[CHARS_PER_ROW*(i-1) + tmp ] = (char)value + 48;
}

void board_put(board_t *self, uint8_t i, uint8_t j, uint8_t value){
    if (((int)board_value_get_original(self, i, j)-48)≠0){
        self→status = BOARD_NON_MODIFIABLE_VALUE;
    } else {
```



```c
        board_value_set(self, i, j, value);
    }
}

void board_verify(board_t *self){
    board_row_verify(self);
    board_col_verify(self);
    board_cell_verify(self);
}

void board_row_verify(board_t *self){
    short row = 0; //Bits para marcar digitos xxxx-xxx1-2345-6789
    int value = 0;

    for (int i=1; i≤9; i++){
        for (int j=1; j≤9; j++){
            value = (int)board_value_get(self, i, j) - 48; //paso a int
            if ( ( (row >> (9 - value)) & 1) ) {
                self→status = BOARD_INVALID;
            } else {
                if (value ≠ 0)
                    row = row | (1 << (9-value));
            }
        }
        row = 0;
    }
}

void board_col_verify(board_t *self){
    short col = 0; //Bits para marcar digitos xxxx-xxx1-2345-6789
    int value = 0;

    for (int i=1; i≤9; i++){
        for (int j=1; j≤9; j++){
            value = (int)board_value_get(self, j, i) - 48; //paso a int
            if ( ( (col >> (9 - value)) & 1) ) {
                self→status = BOARD_INVALID;
            } else {
                if (value ≠ 0)
                    col = col | (1 << (9-value));
            }
        }
        col = 0;
    }
}

//La extensiÃ³n de esta funciÃ³n se debe a que debe recorrer cada celda
//lo cual resulta en un algoritmo particular.
void board_cell_verify(board_t *self){
    short cell = 0; //Bits para marcar digitos xxxx-xxx1-2345-6789
    int value = 0;

    //9 celdas
    for (int i = 1; i ≤ 3; i++){
        for (int j = 1; j ≤ 3; j++){
            //verifico una celda
            for (int k = 3*i-2; k ≤ 3*i; k++){
                for (int l = 3*j-2; l ≤ 3*j; l++){
                    value = (int)board_value_get(self, k, l) - 48; //paso a int
                    if ( ( (cell >> (9 - value)) & 1) ) {
                        self→status = BOARD_INVALID;
                    } else {
                        if (value ≠ 0)
                            cell = cell | (1 << (9-value));
                    }
                }
```

```
            }
            cell = 0;
        }
    }
}

void board_reset(board_t *self){
        snprintf(self→board, sizeof(self→board), "%s", self→board_original);
        printf("%s", MSG_BOARD_RESTART);
        self→status = BOARD_OK;
}
```


Table of Contents

Referencias

[1] **http://valgrind.org**

[2] **https://www.gnu.org/software/gdb/**

[3] **http://man7.org/linux/man-pages/man3/getaddrinfo.3.html**

[4] **http://man7.org/linux/man-pages/man2/socket.2.html**

[5] **http://man7.org/linux/man-pages/man2/accept.2.html**

[6] **http://man7.org/linux/man-pages/man2/bind.2.html**

[7] **http://man7.org/linux/man-pages/man2/close.2.html**

[8] **http://man7.org/linux/man-pages/man2/connect.2.html**

[9] **http://man7.org/linux/man-pages/man2/listen.2.html**

[10] **http://man7.org/linux/man-pages/man2/shutdown.2.html**

[11] **http://man7.org/linux/man-pages/man2/send.2.html**

[12] **http://man7.org/linux/man-pages/man2/recv.2.html**