abr 15, 18 21:48 **socket.h** Page 1/1

```c
#ifndef SOCKET_H
#define SOCKET_H

#include <stddef.h>
#include <stdbool.h>
#include <netdb.h>

// definicion estructura socket
typedef struct socket {
  int socket;
}socket_t;


// declaracion primitivas socket
int socket_create(socket_t* socket);
int socket_connect(socket_t* socket,
const char* hostname, const char* service_name);

int socket_send(socket_t* socket_,
unsigned char* chunk, size_t sizeof_chunk);

int socket_receive(socket_t* socket_, unsigned char* chunk,
size_t sizeof_chunk);

int socket_bind_and_listen(socket_t* socket, const char* port);
int socket_accept(socket_t* socket_, socket_t* new_socket);
void socket_shutdown_rw(socket_t* socket);
void socket_destroy(socket_t* socket_);


#endif // SOCKET_H
```

abr 15, 18 21:48 **socket.c** Page 1/3

```c
#define _POSIX_C_SOURCE 200112L
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdbool.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include "socket.h"
#define OK 0
#define ERROR -1
#define MAX_CLIENTS_WAITING 5


int socket_create(socket_t* socket_) {
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0) {
    fprintf(stderr, "Error: %s\n", strerror(errno));
    return ERROR;
  }
  socket_→socket = sock;
    return OK;
}


int socket_connect(socket_t* socket_, const char* hostname, const char* port) {
  struct sockaddr_in client;

  client.sin_family = AF_INET;
  client.sin_port = htons((uint16_t)atoi(port));
  client.sin_addr.s_addr = inet_addr(hostname);

  if (connect(socket_→socket, (struct sockaddr *)&client,
  sizeof(client)) < 0) {
    fprintf(stderr, "Error: %s\n", strerror(errno));
    return ERROR;
  }
    return OK;
}


int socket_send(socket_t* socket_, unsigned char* chunk, size_t sizeof_chunk) {
  int bytes_enviados = 0;
  int s;
  bool is_valid_socket = true;
  bool is_open_socket = true;

  while(bytes_enviados < sizeof_chunk ∧ is_valid_socket ∧ is_open_socket) {
    s = send(socket_→socket, &chunk[bytes_enviados],
    sizeof_chunk − bytes_enviados, MSG_NOSIGNAL);
    if (s < 0) {
      is_valid_socket = false;
    } else if (s ≡ 0) {
      is_open_socket = false;
    } else {
      bytes_enviados += s;
    }
  }
  if (is_valid_socket) {
    return OK;
  }
  return ERROR;
}


```

```
67  int socket_bind_and_listen(socket_t* socket_, const char* port) {
68    int val = 1;
69    setsockopt(socket_→socket, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
70
71    struct addrinfo hints;
72    struct addrinfo *result;
73    int s = 0;
74
75    memset(&hints, 0, sizeof(struct addrinfo));
76    hints.ai_family = AF_INET;
77    hints.ai_socktype = SOCK_STREAM;
78    hints.ai_flags = AI_PASSIVE;
79    s = getaddrinfo(NULL, port, &hints, &result);
80
81    s = bind(socket_→socket, result→ai_addr, result→ai_addrlen);
82    if (s ≡ -1) {
83      fprintf(stderr,"Error: %s\n", strerror(errno));
84      return ERROR;
85    }
86    freeaddrinfo(result);
87
88    s = listen(socket_→socket, MAX_CLIENTS_WAITING);
89    if (s ≡ -1) {
90      fprintf(stderr,"Error: %s\n", strerror(errno));
91      return ERROR;
92    }
93    return OK;
94  }
95
96  int socket_accept(socket_t* socket_, socket_t* new_socket){
97    int s = accept(socket_→socket, NULL, NULL);
98    if (s < 0) {
99      return ERROR;
100   }
101   new_socket→socket = s;
102   return OK;
103 }
104
105
106 int socket_receive(socket_t* socket_, unsigned char* chunk,
107 size_t sizeof_chunk) {
108   int bytes_recibidos = 0;
109   int s;
110   bool is_open_socket = true;
111   bool is_valid_socket = true;
112
113   while(bytes_recibidos < sizeof_chunk ∧ is_valid_socket ∧ is_open_socket) {
114     s = recv(socket_→socket, &chunk[bytes_recibidos],
115     sizeof_chunk - bytes_recibidos, 0);
116     if (s < 0) {
117       is_valid_socket = false;
118     } else if (s ≡ 0) {
119       is_open_socket = false;
120     } else {
121       bytes_recibidos += s;
122     }
123   }
124
125   if (¬is_open_socket ∧ is_valid_socket) {
126     return bytes_recibidos;
127   }
128   return ERROR;
129 }
130
131
132 void socket_shutdown_rw(socket_t* socket_) {
```



```
133   shutdown(socket_→socket, SHUT_RDWR);
134 }
135
136
137 void socket_destroy(socket_t* socket_) {
138   close(socket_→socket);
139 }
```

abr 15, 18 21:48 **servidor.h** Page 1/1

```h
#ifndef SERVIDOR_H
#define SERVIDOR_H

int servidor(const char* service_name, unsigned char* key);

#endif //SERVIDOR_H
```

abr 15, 18 21:48 **servidor.c** Page 1/2

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "arc4.h"
#include "socket.h"
#include "fprintf_hexa.h"
#define OK 0
#define ERROR -1
#define INPUT_ERROR "Parametros Incorrectos"
#define SIZEOF_CHUNK 50
#define NAME_OUTPUT_FILE "out"
#define BYTE_SIZE 1


int servidor(const char* service_name, unsigned char* key) {
  FILE* file = fopen(NAME_OUTPUT_FILE, "wb");
  if(¬file) {
    fprintf(stderr, INPUT_ERROR);
    return ERROR;
  }

  // creo el socket del servidor
  socket_t socket_;
  int s = socket_create(&socket_);
  if (s < 0) {
    fclose(file);
    return ERROR;
  }

  // defino interface a la que se conecta el socket y cuantas conexiones
  // puede tener en espera
  s = socket_bind_and_listen(&socket_, service_name);
  if (s < 0) {
    fclose(file);
    socket_destroy(&socket_);
    return ERROR;
  }

  // dejo el socket esperando que acepte la conexion con un socket cliente
  // creo un nuevo socket asociado al socket cliente
  socket_t socket_asociado;
  s = socket_accept(&socket_, &socket_asociado);
  if (s < 0) {
    fclose(file);
    socket_shutdown_rw(&socket_);
    socket_destroy(&socket_);
    return ERROR;
  }

  unsigned char crypted_chunk[SIZEOF_CHUNK];
  bool is_valid_socket = true;
  arc4_t arc4_;
  arc4_create(&arc4_, key, (unsigned int) strlen((const char*)key));

  // mientras siga recibiendo datos a traves del socket asociado, voy a
  // desencriptarlos e imprimirlos
  while(is_valid_socket) {
    size_t bytes_received = socket_receive(&socket_asociado,
    crypted_chunk, SIZEOF_CHUNK);
    is_valid_socket = (bytes_received > 0);
    if (¬is_valid_socket) {
      break;
    }

    int len_chunk;
```

```
67      if (bytes_received < SIZEOF_CHUNK) {
68        len_chunk = bytes_received;
69      } else {
70        len_chunk = SIZEOF_CHUNK;
71      }
72
73      unsigned char output[len_chunk];
74      unsigned char key_stream[len_chunk];
75
76      // desencripto los datos recibidos a traves del socket asociado
77      arc4_process(crypted_chunk, len_chunk, output, key_stream, &arc4_);
78
79      // fprintfs que demanda la catedra para el tp
80      fwrite(output, BYTE_SIZE, len_chunk, file);
81      fprintf_hexa(stderr, "%02X", key_stream, len_chunk);
82      fprintf_hexa(stdout, "%02x", output, len_chunk);
83    }
84
85    arc4_destroy(&arc4_);
86    socket_shutdown_rw(&socket_asociado);
87    socket_destroy(&socket_asociado);
88    socket_shutdown_rw(&socket_);
89    socket_destroy(&socket_);
90    fclose(file);
91    return OK;
92  }
```



```
1   #include <stdio.h>
2   #include <string.h>
3   #include "cliente.h"
4   #include "servidor.h"
5   #define TODO_OK 0
6   #define CLIENTE "client"
7   #define SERVIDOR "server"
8   #define ERROR 1
9   #define ERROR_ENTRADA "Parametros Incorrectos"
10  #define CANTIDAD_MINIMA_PARAMETROS 1
11
12
13  int main(int argc, char* argv[]) {
14    if (argc < CANTIDAD_MINIMA_PARAMETROS) {
15      fprintf(stderr, ERROR_ENTRADA);
16      return ERROR;
17    }
18
19    if (strcmp(argv[1], CLIENTE) ≡ 0) {
20      return cliente((const char*)argv[2],
21      (const char*)argv[3], (unsigned char*)argv[4], argv[5]);
22    } else if (strcmp(argv[1], SERVIDOR) ≡ 0) {
23      return servidor((const char*)argv[2], (unsigned char*)argv[3]);
24    } else {
25      fprintf(stderr, ERROR_ENTRADA);
26      return ERROR;
27    }
28  }
```

abr 15, 18 21:48 **fprintf_hexa.h** Page 1/1

```h
#ifndef FPRINTF_HEXA_H
#define FPRINTF_HEXA_H

void fprintf_hexa(FILE* file, char* flag, unsigned char* input, int len_input);

#endif // FPRINTF_HEXA_H
```

abr 15, 18 21:48 **fprintf_hexa.c** Page 1/1

```c
#include <stdio.h>

void fprintf_hexa(FILE* file, char* flag, unsigned char* input, int len_input) {
  for (int i = 0; i < len_input; i++){
    fprintf(file, flag, input[i]);
  }
}
```

abr 15, 18 21:48 **cliente.h** Page 1/1

```c
#ifndef CLIENTE_H
#define CLIENTE_H

int cliente(const char* hostname, const char* port, unsigned char* key,
char* filename);

#endif //CLIENTE_H
```

abr 15, 18 21:48 **cliente.c** Page 1/2

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "arc4.h"
#include "socket.h"
#include "fprintf_hexa.h"
#define OK 0
#define ERROR -1
#define INPUT_ERROR "Parametros Incorrectos"
#define SIZEOF_CHUNK 64
#define BYTE_SIZE 1


int cliente(const char* hostname, const char* port, unsigned char* key,
char* filename) {
  // abro el archivo
  int s;
  FILE* file;
  if (filename ≠ NULL) {
    file = fopen(filename, "r");
  } else {
    file = stdin;
  }
  if (¬file) {
    fprintf(stderr, INPUT_ERROR);
    return ERROR;
  }

  // creo el socket del cliente
  socket_t socket;
  s = socket_create(&socket);
  if (s < 0) {
    fclose(file);
    return ERROR;
  }

  // me conecto con otro socket de tipo cliente
  s = socket_connect(&socket, hostname, port);
  if (s < 0) {
    fclose(file);
    socket_destroy(&socket);
    return ERROR;
  }

  unsigned char chunk[SIZEOF_CHUNK];
  bool is_valid_socket = true;
  arc4_t arc4_;
  arc4_create(&arc4_, key, (unsigned int) strlen((const char*)key));

  // mientras no haya errores y pueda seguir leyendo el archivo, voy a enviar
  // datos a un socket de tipo servidor asociado a este socket de tipo cliente
  while (is_valid_socket) {
    size_t bytes_read = fread(chunk, BYTE_SIZE, SIZEOF_CHUNK, file);
    int len_chunk;
    if (bytes_read < SIZEOF_CHUNK) {
      len_chunk = bytes_read;
    } else {
      len_chunk = SIZEOF_CHUNK;
    }

    unsigned char output[len_chunk];
    unsigned char key_stream[len_chunk];

    // encripto lo leido en el archivo
    arc4_process(chunk, len_chunk, output, key_stream, &arc4_);
```

abr 15, 18 21:48 **cliente.c** Page 2/2

```
67
68      // fprintfs que demanda la catedra para el tp
69      fprintf_hexa(stderr, "%02X", key_stream, len_chunk);
70      fprintf_hexa(stdout, "%02x", output, len_chunk);
71
72      // envio los datos encriptados al socket de tipo servidor
73      int s = socket_send(&socket, output, len_chunk);
74
75      is_valid_socket = (s ≡ 0);
76      if (¬is_valid_socket) {
77        break;
78      }
79      if (feof(file)) {
80        break;
81      }
82    }
83
84    arc4_destroy(&arc4_);
85    socket_shutdown_rw(&socket);
86    socket_destroy(&socket);
87    if (file ≠ stdin) {
88      fclose(file);
89    }
90    return OK;
91  }
```

abr 15, 18 21:48 **arc4.h** Page 1/1

```
1   #ifndef ARC4_H
2   #define ARC4_H
3
4   #include <stddef.h>
5
6   typedef struct {
7     unsigned char* state_array;
8     unsigned int i;
9     unsigned int j;
10  }arc4_t;
11
12  // algoritmo de cifrado
13  void arc4_create(arc4_t* arc4_, unsigned char* key, unsigned int len_key);
14  void arc4_destroy(arc4_t* arc4_);
15  void arc4_process(unsigned char* input, size_t len_input,
16  unsigned char* output, unsigned char* key_stream, arc4_t* arc4_);
17
18  #endif //ARC4_H
```

abr 15, 18 21:48 **arc4.c** Page 1/1

```c
#include "arc4.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#define BYTES 256


void swap_(unsigned int k, unsigned int l, arc4_t* arc4_) {
  unsigned char c = arc4_→state_array[k];
  arc4_→state_array[k] = arc4_→state_array[l];
  arc4_→state_array[l] = c;
}


unsigned char gen_pseudorandom(arc4_t* arc4_) {
  arc4_→i = (arc4_→i + 1) % BYTES;
  arc4_→j = (arc4_→j + arc4_→state_array[arc4_→i]) % BYTES;
  swap_(arc4_→i, arc4_→j, arc4_);
  return arc4_→state_array[(arc4_→state_array[arc4_→i] +
  arc4_→state_array[arc4_→j]) % BYTES];
}


void arc4_create(arc4_t* arc4_, unsigned char* key, unsigned int len_key) {
  arc4_→state_array = (unsigned char*) malloc(sizeof(unsigned char) * BYTES);
  unsigned int i = 0, j = 0;
  for(; i < BYTES; i++) {
    arc4_→state_array[i] = (unsigned char)i;
  }
  /* Key-Scheduling (KSA) */
  for (i = 0, j = 0; i < BYTES; i++) {
    j = (j + key[i % len_key] + arc4_→state_array[i]) % BYTES;
    swap_(i, j, arc4_);
  }
  arc4_→i = 0;
  arc4_→j = 0;
}


void arc4_destroy(arc4_t* arc4_) {
  free(arc4_→state_array);
}


void arc4_process(unsigned char* input, size_t len_input,
unsigned char* output, unsigned char* key_stream, arc4_t* arc4_) {
  /* lleno output y key_stream*/
  for (size_t k = 0; k < len_input; k++) {
    unsigned char char_pseudorandom = gen_pseudorandom(arc4_);
    output[k] = input[k] ^ char_pseudorandom;
    key_stream[k] = char_pseudorandom;
  }
}
```

abr 15, 18 21:48 **Table of Content** Page 1/1

Table of Contents