mar 29, 16 0:58 **socket.h** Page 1/1

```c
#ifndef __SOCKET_H__
#define __SOCKET_H__

/**
 * AbtraccÃ³n del socket para utilizarlo con funciones de mÃ¡s alto nivel.
 */

typedef struct {
  int fd;
} socket_t;


/**
 * La creaciÃ³n del socket. Requiere de un posterior destroy.
 */
int socket_create(socket_t* s);

/**
 * Libera los recursos tomados por la creaciÃ³n.
 */
int socket_destroy(socket_t* s);

/**
 * Pone un socket a escuchar el puerto pasado por parÃ¡metro.
 */
int socket_bind_and_listen(socket_t* s, int port);

/**
 * Acepta una conecciÃ³n entrante para el socket pasado.
 */
socket_t socket_accept(socket_t* s);

/**
 * Conecta el socket al hostname y puerto pasados por parÃ¡metro
 */
int socket_connect(socket_t* s, const char* hostname, const char* port);

/**
 * Lee del socket un nÃºmero de bytes especificado y lo escribe en la cadena
 * buff.
 */
int socket_read(socket_t* s, char* buff, size_t bytes);

/**
 * Escribe en el socket un nÃºmero de bytes especificado de la cadena buff.
 */
int socket_write(socket_t* s, char* buff, size_t bytes);

#endif
```

mar 29, 16 0:58 **socket.c** Page 1/2

```c

#include "common.h"
#include "socket.h"


// Sin este define el SERCOM no compila.
// http://stackoverflow.com/questions/11405819/does-struct-hostent-have-a-field-h-addr
#define h_addr h_addr_list[0] /* for backward compatibility */

#define NUM_CLIENTS 5

#ifndef MSG_NOSIGNAL
// En Os X no existe el MSG_NOSIGNAL, entonces lo uso sin flags.
#define SOCKET_FLAGS 0
#else
#define SOCKET_FLAGS MSG_NOSIGNAL
#endif


int socket_create(socket_t* s) {
  s->fd = socket(AF_INET, SOCK_STREAM, 0);
  return s->fd;
}


int socket_bind_and_listen(socket_t* s, int port) {
  struct sockaddr_in serv_addr;
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(port);

  int c = bind(s->fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
  if (c < 0) return c;

  return listen(s->fd, NUM_CLIENTS);
}


socket_t socket_accept(socket_t* s) {
  struct sockaddr_in cli_addr;
  socket_t cli_socket;
  socklen_t cli_len = sizeof(cli_addr);
  cli_socket.fd = accept(s->fd, (struct sockaddr *) &cli_addr, &cli_len);

  return cli_socket;
}

// Por algun extraÃ±o motivo SERCOM no entiende el getaddrinfo (tira error
// de compilacion) - entonces no puedo usar esta versiÃ³n del connect.

// int socket_connect(socket_t* s, const char* hostname, const char* port) {
//   struct addrinfo hints, *servinfo, *p;

//   memset(&hints, 0, sizeof(hints));
//   hints.ai_family = AF_UNSPEC;
//   hints.ai_socktype = SOCK_STREAM;

//   getaddrinfo(hostname, port, &hints, &servinfo);

//   for (p = servinfo; p != NULL; p = p->ai_next) {
//     if (connect(s->fd, p->ai_addr, p->ai_addrlen) != -1) {
//       break;
//     }
//   }
```

mar 29, 16 0:58 **socket.c** Page 2/2

```
66
67  //   freeaddrinfo(servinfo);
68  //   return 0;
69  // }
70
71
72
73  // Ambas versiones de socket_connect pierden memoria en OS X segÃºn el valgrind.
74  // Parece ser un feature y no un bug:
75  // http://stackoverflow.com/questions/13229913/getaddrinfo-memory-leak
76
77  int socket_connect(socket_t* s, const char* hostname, const char* port) {
78    struct hostent *server;
79    struct sockaddr_in serv_addr;
80
81    server = gethostbyname(hostname);
82    if (server == NULL) return 1;
83
84    bzero((char *) &serv_addr, sizeof(serv_addr));
85    serv_addr.sin_family = AF_INET;
86    bcopy((char *)server->h_addr,
87          (char *)&serv_addr.sin_addr.s_addr,
88          server->h_length);
89    serv_addr.sin_port = htons(atoi(port));
90
91    return connect(s->fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
92  }
93
94
95  int socket_read(socket_t* s, char* buff, size_t bytes) {
96    int length = bytes;
97    while (length > 0)
98    {
99      int i = recv(s->fd, buff, length, SOCKET_FLAGS);
100     if (i < 1) return i;
101     buff += i;
102     length -= i;
103   }
104   return 0;
105 }
106
107
108 int socket_write(socket_t* s, char* buff, size_t bytes) {
109   int length = bytes;
110   while (length > 0)
111   {
112     int i = send(s->fd, buff, length, SOCKET_FLAGS);
113     if (i < 1) return i;
114     buff += i;
115     length -= i;
116   }
117   return 0;
118 }
119
120
121 int socket_destroy(socket_t* s) {
122   return close(s->fd);
123 }
```

mar 29, 16 0:58 **server.h** Page 1/1

```
1   #ifndef __SERVER_H__
2   #define __SERVER_H__
3
4   /**
5    * AbstracciÃ³n del servidor
6    */
7
8   #include "bag.h"
9   #include "socket.h"
10
11  typedef struct {
12    socket_t skt;  // Accept socket
13    socket_t c_skt;  // Client socket
14    int port;
15    char* new_file;
16    int block_size;
17    bag_t checksums;
18  } server_t;
19
20  /**
21   * CreaciÃ³n del servidor. Necesita de un posterior destroy.
22   *
23   * @param  s    Puntero al servidor
24   * @param  port Puerto donde correrÃ¡ el servidor
25   * @return      Distinto de 0 si hubo algÃºn error
26   */
27  int server_create(server_t* s, int port);
28
29  /**
30   * Realiza la comunicaciÃ³n con el cliente. Espera la conecciÃ³n del cliente y
31   * envia la informaciÃ³n necesaria.
32   */
33  int server_run(server_t* s);
34
35  /**
36   * Libera los recursos tomados por la creaciÃ³n del servidor.
37   */
38  int server_destroy(server_t* s);
39
40  #endif
```

```c

#include "common.h"
#include "server.h"
#include "protocol.h"


int server_create(server_t* s, int port) {
  socket_t skt;
  socket_create(&skt);
  s→skt = skt;
  s→port = port;
  bag_create(&(s→checksums));
  return 0;
}


int server_receive_filename(server_t* s) {
  int ret;
  char *data = (char*) &ret;
  socket_read(&(s→c_skt), data, 4);

  ret = ntohl(ret);

  s→new_file = malloc(ret + 1);
  socket_read(&(s→c_skt), s→new_file, ret);
  s→new_file[ret] = '\0';

  return 0;
}

int server_receive_block_size(server_t* s) {
  int block_size;
  char *data = (char*) &block_size;
  socket_read(&(s→c_skt), data, 4);

  s→block_size = ntohl(block_size);
  return 0;
}

int server_receive_checksums(server_t* s) {
  char flag = 0;
  int checksum;
  do
  {
    socket_read(&(s→c_skt), &flag, 1);
    if (flag ≡ P_CHECKSUM_START) {
      socket_read(&(s→c_skt), (char*) &checksum, 4);
      bag_add(&(s→checksums), ntohl(checksum));
    }
  } while (flag ≠ P_NO_MORE_CHECKSUMS);
  return 0;
}

int server_send_chunk(server_t* s, char* chunk, int bytes) {
  char flag = P_NEW_FILE_CHUNK;
  socket_write(&(s→c_skt), &flag, 1);
  bytes = htonl(bytes);
  socket_write(&(s→c_skt), (char*) &bytes, 4);
  return socket_write(&(s→c_skt), chunk, ntohl(bytes));
}

int server_sync_file(server_t* s) {
  FILE *new_fd;
  size_t bytes_read;
  int block_index = 0;
  int chunk_size = 0;
```



```c
  char flag = 0;
  char chunk[256];


  new_fd = fopen(s→new_file, "rb");
  if (¬new_fd) return -1;

  char* block = malloc(s→block_size + 1);
  bzero(chunk, 256);

  while (¬feof(new_fd)) {
    bzero(block, s→block_size + 1);
    bytes_read = fread(block, 1, s→block_size, new_fd);

    // Si lei menos bytes que el block_size, los agrego al chunk y corto el
    // ciclo
    if (bytes_read < s→block_size) {
      strncat(chunk, block, bytes_read);
      chunk_size += bytes_read;
      break;
    }

    int cs = checksum(block, s→block_size);

    if ((block_index = bag_search(&(s→checksums), cs)) ≥ 0) {
      // Si encuentro un checksum, primero tengo que vaciar el chunk que tengo
      if (chunk_size > 0) {
        server_send_chunk(s, chunk, chunk_size);
        bzero(chunk, 256);
        chunk_size = 0;
      }
      // Luego envio el numero de bloque encontrado
      // Estoy confiando en que el cliente me mando los checksums en orden
      flag = P_BLOCK_FOUND;
      socket_write(&(s→c_skt), &flag, 1);

      block_index = htonl(block_index);
      socket_write(&(s→c_skt), (char*) &block_index, 4);
    } else {
      // Si no encuentro checksum, agrego el primer byte del bloque al chunk
      // y muevo el puntero del archivo (1 - el tamaÃ±o del bloque).
      strncat(chunk, block, 1);
      fseek(new_fd, 1 - s→block_size, SEEK_CUR);
      chunk_size++;
    }
  }

  if (chunk_size > 0) {
    server_send_chunk(s, chunk, chunk_size);
  }
  flag = P_END_OF_FILE;
  socket_write(&(s→c_skt), &flag, 1);

  fclose(new_fd);
  free(block);
  return 0;
}


int server_run(server_t* s) {
  socket_bind_and_listen(&(s→skt), s→port);
  s→c_skt = socket_accept(&(s→skt));

  server_receive_filename(s);
  server_receive_block_size(s);
  server_receive_checksums(s);
```

mar 29, 16 0:58 **server.c** Page 3/3

```
133
134   server_sync_file(s);
135
136   return 0;
137 }
138
139 int server_destroy(server_t* s) {
140   socket_destroy(&(s→skt));
141   socket_destroy(&(s→c_skt));
142   free(s→new_file);
143   bag_destroy(&(s→checksums));
144   return 0;
145 }
```

mar 29, 16 0:58 **protocol.h** Page 1/1

```
1  #ifndef __PROTOCOL_H__
2  #define __PROTOCOL_H__
3
4  /**
5   * AbstracciÃ³n del protocolo − las partes necesarias para que el cliente y
6   * el servidor se puedan entender.
7   *
8   * Uso el prefijo P_ para las definiciones globales de flags para la
9   * comunicaciÃ³n entre ellos.
10  */
11
12 #define P_CHECKSUM_START 1
13 #define P_NO_MORE_CHECKSUMS 2
14
15 #define P_NEW_FILE_CHUNK 3
16 #define P_BLOCK_FOUND 4
17 #define P_END_OF_FILE 5
18
19
20 /**
21  * Variante del checksum Adler32. El parÃ¡metro bytes define cuantos bytes
22  * de la cadena de caracteres x se van a utilizar.
23  */
24 int checksum(char* x, int bytes);
25
26 #endif
```

mar 29, 16 0:58 **protocol.c** Page 1/1

```c

#include "protocol.h"

#define M 0x00010000


int checksum(char* x, int bytes) {
  int lower = 0, higher = 1;

  for (int i = 0; i < bytes; ++i)
  {
      lower = (lower + x[i]) % M;
      higher = ((bytes - i) * higher) % M;
  }

  return lower + (higher * M);
}
```

mar 29, 16 0:58 **main.c** Page 1/1

```c

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

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


int main(int argc, char const *argv[])
{
  if (argc < 3) {
    return 1;
  } else if (¬strncmp(argv[1], "server", 6)) {
    server_t s;
    server_create(&s, atoi(argv[2]));
    server_run(&s);
    server_destroy(&s);

  } else if (¬strncmp(argv[1], "client", 6)) {
    client_t c;
    // hostname port old_local new_local new_remote block_size
    client_create(&c, argv[2], argv[3], argv[4],
                      argv[5], argv[6], atoi(argv[7]));
    client_run(&c);
    client_destroy(&c);
  }

  return 0;
}
```

mar 29, 16 0:58 **common.h** Page 1/1

```c
#ifndef __COMMON_H__
#define __COMMON_H__

#include <string.h>
#include <strings.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>

#endif

```

mar 29, 16 0:58 **client.h** Page 1/1

```c
#ifndef __CLIENT_H__
#define __CLIENT_H__

/**
 * AbstracciÃ³n del cliente.
 */

#include "socket.h"

typedef struct {
  socket_t skt;  // Server socket
  const char* hostname;
  const char* port;
  const char* old_local_file;
  const char* new_local_file;
  const char* remote_file;
  int block_size;
} client_t;


/**
 * CreaciÃ³n del cliente. Necesita de un destroy posteriormente.
 *
 * @param  c              Puntero al cliente
 * @param  hostname       Hostname del servidor, ej: 127.0.0.1
 * @param  port           Puerto del servidor
 * @param  old_local_file La ruta al archivo que se desea actualizar
 * @param  new_local_file La ruta al archivo donde guardar el resultado
 * @param  remote_file    La ruta al archivo remoto en el servidor
 * @param  block_size     TamaÃ±o del bloque a sacar checksums
 * @return                Distinto de 0 si hay un error.
 */
int client_create(client_t* c, const char* hostname, const char* port,
                  const char* old_local_file, const char* new_local_file,
                  const char* remote_file, int block_size);

/**
 * Realiza la comunicaciÃ³n con el servidor y la actualziaciÃ³n del archivo.
 */
int client_run(client_t* c);

/**
 * Libera los recursos que tomÃ³ la creaciÃ³n del cliente.
 */
int client_destroy(client_t* c);

#endif
```

```c

#include "common.h"
#include "client.h"
#include "protocol.h"


int client_create(client_t* c, const char* hostname, const char* port,
                  const char* old_local_file, const char* new_local_file,
                  const char* remote_file, int block_size) {
  socket_t skt;
  socket_create(&skt);
  c→skt = skt;
  c→hostname = hostname;
  c→port = port;
  c→old_local_file = old_local_file;
  c→new_local_file = new_local_file;
  c→remote_file = remote_file;
  c→block_size = block_size;
  return 0;
}


int client_send_remote_filename(client_t* c) {
  int32_t size = htonl(strlen(c→remote_file));

  socket_write(&(c→skt), (char*) &size, 4);

  socket_write(&(c→skt), (char*) c→remote_file, strlen(c→remote_file));

  return 0;
}

int client_send_block_size(client_t* c) {
  int32_t size = htonl(c→block_size);

  socket_write(&(c→skt), (char*) &size, 4);

  return 0;
}


int client_send_checksums(client_t* c) {
  FILE *old_file;
  size_t bytes_read;
  char flag;

  old_file = fopen(c→old_local_file, "rb");
  if (¬old_file) return -1;  // TODO: manage error
  char* buffer = malloc(c→block_size);
  while ((bytes_read = fread(buffer, 1, c→block_size, old_file))
                                              ≥ c→block_size) {
    int cs = checksum(buffer, c→block_size);
    cs = htonl(cs);
    flag = P_CHECKSUM_START;
    socket_write(&(c→skt), &flag, 1);

    socket_write(&(c→skt), (char*) &cs, 4);
  }
  flag = P_NO_MORE_CHECKSUMS;
  socket_write(&(c→skt), &flag, 1);

  fclose(old_file);
  free(buffer);
  return 0;
}

```



```c

int client_sync_file(client_t* c) {
  char flag;
  char* chunk;
  int bytes_to_write, block_index;
  FILE *old_file, *new_file;

  old_file = fopen(c→old_local_file, "rb");
  if (¬old_file) return -1;
  new_file = fopen(c→new_local_file, "wb");
  if (¬new_file) return -1;

  socket_read(&(c→skt), &flag, 1);

  while (flag ≠ P_END_OF_FILE) {
    if (flag ≡ P_NEW_FILE_CHUNK) {
      socket_read(&(c→skt), (char*) &bytes_to_write, 4);
      bytes_to_write = ntohl(bytes_to_write);
      printf("RECV File chunk %d bytes\n", bytes_to_write);

      chunk = malloc(bytes_to_write);
      socket_read(&(c→skt), chunk, bytes_to_write);

      fwrite(chunk, 1, bytes_to_write, new_file);

      free(chunk);

    } else if (flag ≡ P_BLOCK_FOUND) {
      socket_read(&(c→skt), (char*) &block_index, 4);
      block_index = ntohl(block_index);
      printf("RECV Block index %d\n", block_index);

      chunk = malloc(c→block_size + 1);
      fseek(old_file, c→block_size * block_index, SEEK_SET);
      if (fread(chunk, 1, c→block_size, old_file)) {
        fwrite(chunk, 1, c→block_size, new_file);
      } else {
        return -1;
      }

      free(chunk);

    } else {
      return -1;  // Incorrect flag
    }

    socket_read(&(c→skt), &flag, 1);
  }

  printf("RECV End of file\n");

  fclose(old_file);
  fclose(new_file);

  return 0;
}


int client_run(client_t* c) {
  socket_connect(&(c→skt), c→hostname, c→port);

  client_send_remote_filename(c);
  client_send_block_size(c);
  client_send_checksums(c);

  client_sync_file(c);
```

mar 29, 16 0:58 **client.c** Page 3/3

```
133
134   return 0;
135 }
136
137
138
139 int client_destroy(client_t* c) {
140   socket_destroy(&(c→skt));
141   return 0;
142 }
```

mar 29, 16 0:58 **bag.h** Page 1/1

```
1  #ifndef __BAG_H__
2  #define __BAG_H__
3
4  /**
5   * La bolsa (bag) es una estructura sencilla que permite guardar items de tipo
6   * int, y luego buscarlos por valor. Lo justo para guardar checksums en el
7   * servidor.
8   *
9   * EstÃ¡ implementada como una lista enlazada.
10  */
11
12 typedef struct bag_node_t {
13   int item;
14   struct bag_node_t* next;
15   struct bag_node_t* previous;
16 } bag_node_t;
17
18 typedef struct {
19   bag_node_t* initial;
20   bag_node_t* current;
21 } bag_t;
22
23
24 /**
25  * Crea de la bolsa.
26  */
27 int bag_create(bag_t* b);
28
29 /**
30  * Agrega un elemento en la bolsa en la posiciÃ³n actual.
31  */
32 int bag_add(bag_t* b, int item);
33
34 /**
35  * Busca el elemento y devuelve su Ã-ndice en la bolsa o −1 en el caso de
36  * que no se encuentre.
37  */
38 int bag_search(bag_t* b, int item);
39
40 /**
41  * Libera los recursos tomados por la creaciÃ³n.
42  */
43 int bag_destroy(bag_t* b);
44
45 #endif
```

mar 29, 16 0:58 **bag.c** Page 1/1

```c

#include <stddef.h>
#include <stdlib.h>

#include "bag.h"


int bag_create(bag_t* b) {
  b→initial = malloc(sizeof(bag_node_t));
  b→current = b→initial;
  b→initial→next = NULL;
  return 0;
}

int bag_add(bag_t* b, int item) {
  b→current→item = item;
  b→current→next = malloc(sizeof(bag_node_t));
  b→current→next→previous = b→current;
  b→current = b→current→next;
  b→current→next = NULL;
  return 0;
}

int bag_search(bag_t* b, int item) {
  b→current = b→initial;
  if (b→current ≡ NULL) return −1;
  int i = 0;
  while (b→current→next ≠ NULL) {
    if (b→current→item ≡ item) return i;
    b→current = b→current→next;
    i++;
  }
  return −1;
}

int bag_destroy(bag_t* b) {
  b→current = b→initial;
  while (b→current→next) {
    bag_node_t* temp = b→current→next;
    free(b→current);
    b→current = temp;
  }
  free(b→current);
  return 0;
}
```

mar 29, 16 0:58 **Table of Content** Page 1/1