Informe de Trabajo Practico Nº 1

27 de marzo de 2016

75.42 Taller de Programación I

Cátedra Veiga

Facultad de Ingeniera - UBA

Autor: Martín Stancanelli

Padrón: 95188

Objetivo

El objetivo del trabajo practico fue utilizar los conocimientos obtenidos en las clases de sockets y TDAs. Para ello se usaron las librerías de socket provistas por el entorno y se desarrollaron un conjunto de TDAs que se explicaran en secciones posteriores.

Configuración del entorno

El entorno de trabajo elegido para la realización del trabajo practico fue una PC con SO Linux Mint 17.3 y compilador gcc 4.8 configurado para utilizar el estándar de C99.

Explicación de enunciado

El trabajo consiste en realizar un sistema que haga una sincronización de archivos entre un cliente y un servidor. El cliente tendrá una copia local de un archivo y el servidor una nueva versión del mismo. El objetivo es que el cliente notifique al servidor el archivo que tiene a través de checksums de bloques de bytes utilizando una variante del *Adler32.*

El servidor recibe los checksums del cliente y compara contra los checksums de su archivo remoto. Para los que coincidan se le notificará al cliente el bloque para el cual el checksum coincide, evitando así reenviar todo el checksum y reutilizando el archivo local del cliente. En caso de que haya un bloque nuevo se le enviara al cliente los nuevos bytes que debe agregar a su archivo. Finalmente el cliente escribe en un nuevo archivo los datos que ya tenia y los que recibe del lado del servidor.

Desarrollo

A continuación se detallara el algoritmo mas complejo del trabajo, el cual es como el servidor compara los checksums recibidos con los de su archivo remoto y notifica al cliente.

En primer lugar el servidor abre su archivo local (*target*). Carga el primer bloque del archivo en memoria y le calcula su checksum usando la variante de *Adler32.* Seguido de esto comienza a iterar. En caso de que el checksum no este en la lista de checksums que recibió del cliente, desplaza la ventana de lectura del target en 1 byte y guarda el byte que quedo fuera de la ventana en un buffer. A continuación calcula el *rolling checksum*[1] del nuevo bloque y reinicia la iteración.

En el caso en que el checksum este en la lista de checksums que el cliente le envió, entonces se fija si tiene bytes que hayan sido desplazados de la ventana y guardados en el buffer. Si hay entonces le envía esos bytes al cliente. Si no hay bytes desplazados entonces envía al cliente el numero de bloque que este tiene en su archivo local para el cual el checksum del archivo target coincide. Para terminar mueve la ventana de lectura en un bloque, le calcula su checksum y

[1] *Se calcula el rolling checksum ya que es mas eficiente que volver a calcular el checksum del bloque entero*

reinicia la iteración.

Se incluye un diagrama de estados (1) explicando el flujo de este algoritmo

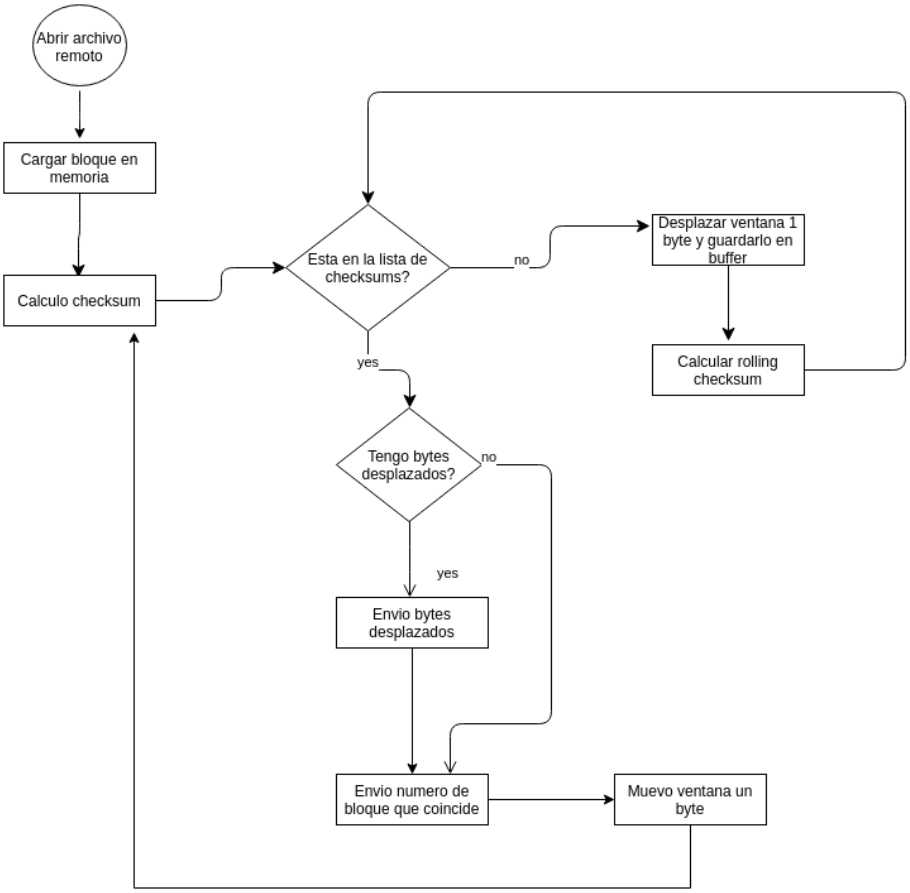

Figura 1: Diagrama de estados

Conclusión

En este trabajo se utilizaron conocimientos nuevos, aprendidos en las clases de la materia, sobre sockets y se reutilizaron algunos como los de TDAs que habíamos incorporado de materias anteriores. Si bien al principio el problema de los sockets puede ser un poco complejo, una correcta utilización de TDAs y por ende un correcto encapsulamiento permite que nos podamos enfocar de lleno en lo mas importante del enunciado del trabajo practico, como lo es la sincronización de archivos.

mar 27, 16 17:03 **server.h** Page 1/1

```c
/*
 * server.h
 *
 *  Created on: Mar 20, 2016
 *      Author: mastanca
 */

#ifndef SRC_SERVER_H_
#define SRC_SERVER_H_

#include <stdio.h>
#include <string.h>
#include "constants.h"
#include "lib_socket.h"
#include "list.h"
#include "lib_checksum.h"
#include "file_handler.h"

typedef struct server{
  socket_t* skt;
  unsigned int block_size;
  FILE* remote_file;
  list_t checksum_list;
}server_t;

int server_execution(int argc, char* argv[]);
int receive_remote_filename(socket_t* skt, server_t* server);
int receive_checksum_list(socket_t* skt, unsigned int block_size,
    server_t* server);
int start_comparison_sequence(server_t* server, socket_t* skt);
int checksum_not_found(char* block, list_t* window_out_bytes, server_t* server,
    checksum_t checksum);
int send_windowed_bytes(list_t* window_out_bytes, server_t* server,
    socket_t* client_skt);
int send_found_block_number(socket_t* client_skt, unsigned int index);
int send_eof(socket_t* client_skt);

#endif /* SRC_SERVER_H_ */
```

mar 27, 16 17:03 **server.c** Page 1/4

```c
/*
 * server.c
 *
 *  Created on: Mar 20, 2016
 *      Author: mastanca
 */

#include "server.h"

int server_execution(int argc, char* argv[]){
  if (argc ≠ 3){
    return 1;
  }
  char* port = argv[2];
  socket_t acep;
  server_t server;
  server.skt = &acep;

  list_t list;
  // Compiler warning if this values are not zero before list init
  list.capacity = 0;
  list.size = 0;
  list.data = NULL;
  server.checksum_list = list;
  list_init(&server.checksum_list);

  socket_init(server.skt, NULL, port);
  // // Avoid time wait
  // int option = 1;
  // setsockopt(server.skt->fd,SOL_SOCKET,(SO_REUSEPORT | SO_REUSEADDR),
  //    (char*)&option,sizeof(option));

  socket_bind(server.skt);

  socket_listen(server.skt, 5);
  socket_t client_skt;
  socket_accept(server.skt, &client_skt);

  receive_remote_filename(&client_skt, &server);

  receive_checksum_list(&client_skt, server.block_size, &server);

  start_comparison_sequence(&server, &client_skt);

  socket_destroy(&client_skt);

  socket_destroy(server.skt);

  fclose(server.remote_file);
  // Free checksum list
  list_free(&server.checksum_list);
  return EXIT_SUCCESS;
}

int start_comparison_sequence(server_t* server, socket_t* skt){
  bool read_something = false;
  list_t window_out_bytes;
  list_init(&window_out_bytes);

  // Load new block from file
  char* block = calloc(server→block_size + 1, sizeof(char));
  read_from_file(server→remote_file, block, server→block_size,
      &read_something);

  // Get checksum of the new block
  checksum_t checksum;
```

```
67    set_checksum(&checksum, block, strlen(block));
68
69    while(¬feof(server→remote_file)){
70      int i = 0;
71      int found_index = 0;
72      bool found = false;
73      while(i < server→checksum_list.size ∧ ¬found){
74        int i_element = list_get(&server→checksum_list, i);
75        if(checksum.checksum ≡ i_element){
76          found = true;
77          found_index = i;
78        }
79        i++;
80      }
81      if (¬found){
82        checksum_not_found(block, &window_out_bytes, server, &checksum);
83      }else{
84        if (window_out_bytes.size > 0){
85          send_windowed_bytes(&window_out_bytes, server, skt);
86        }
87        send_found_block_number(skt, found_index);
88        read_from_file(server→remote_file, block, strlen(block),
89         &read_something);
90        set_checksum(&checksum, block, strlen(block));
91      }
92    }
93    // If there are remaining windowed bytes send them
94    if (window_out_bytes.size > 0 ∨ ((strlen(block) > 0) ∧
95     (read_something ≡ true))){
96      for (int i = 0; i < strlen(block); ++i) {
97        char remaining_char = block[i];
98        list_append(&window_out_bytes, remaining_char);
99      }
100     send_windowed_bytes(&window_out_bytes, server, skt);
101   }
102   free(block);
103   list_free(&window_out_bytes);
104   send_eof(skt);
105
106   return EXIT_SUCCESS;
107 }
108
109 int receive_remote_filename(socket_t* skt, server_t* server){
110   int filename_length;
111   int block_size;
112
113   socket_receive(skt, (char*)&filename_length, sizeof(int));
114
115   char *name = malloc(filename_length + 1);
116   socket_receive(skt, name, filename_length);
117   name[filename_length] = 0;
118
119   socket_receive(skt, (char*)&block_size, sizeof(int));
120
121   server→block_size = block_size;
122
123   // Open remote file here and assign to server_t
124   server→remote_file = fopen(name, "r");
125
126   free(name);
127
128   return EXIT_SUCCESS;
129 }
130
131 int receive_checksum_list(socket_t* skt, unsigned int block_size,
132     server_t* server){
```



```
133   char code = '\0';
134   int checksum = 0;
135   while (code ≠ END_OF_LIST){
136     socket_receive(skt, (char*)&code, sizeof(code));
137
138     if (CHECKSUM_INDICATOR ≡ code){
139       socket_receive(skt, (char*)&checksum, sizeof(checksum));
140       list_append(&(server→checksum_list), checksum);
141     }
142   }
143   return EXIT_SUCCESS;
144 }
145
146 int checksum_not_found(char* block, list_t* window_out_bytes, server_t* server,
147     checksum_t* checksum){
148   char byte_to_window = block[0];
149   list_append(window_out_bytes, byte_to_window);
150
151   // Move cursor block size bytes to the left and return 1
152   int index = WINDOW_BYTE_DISPLACEMENT * (server→block_size) +
153    (-1 * WINDOW_BYTE_DISPLACEMENT);
154   fseek(server→remote_file, index, SEEK_CUR);
155   bool read_something = false;
156   read_from_file(server→remote_file, block, server→block_size,
157     &read_something);
158   char* rolling_buffer = calloc(server→block_size + 1, sizeof(char));
159   rolling_buffer[0] = byte_to_window;
160   memcpy(rolling_buffer + strlen(rolling_buffer), block, strlen(block));
161   checksum_t old_checksum;
162   old_checksum = *checksum;
163
164   rolling_checksum(checksum, &old_checksum, rolling_buffer +1,
165       server→block_size);
166
167   free(rolling_buffer);
168   return EXIT_SUCCESS;
169 }
170
171 int send_windowed_bytes(list_t* window_out_bytes, server_t* server,
172     socket_t* skt){
173   char* buffer_to_send = calloc(window_out_bytes→size + 1, sizeof(char));
174   for (int i = 0; i < window_out_bytes→size; ++i) {
175     char i_element = list_get(window_out_bytes, i);
176     strncat(buffer_to_send, &i_element, sizeof(char));
177   }
178   char new_bytes_indicator = NEW_BYTES_INDICATOR;
179
180   socket_send(skt, (char*)&new_bytes_indicator, sizeof(new_bytes_indicator));
181
182   // Send 4 bytes with the length of the new bytes
183   int new_bytes_size = strlen(buffer_to_send);
184
185   socket_send(skt, (char*)&new_bytes_size, sizeof(new_bytes_size));
186
187   // Send the actual bytes
188   socket_send(skt, buffer_to_send, strlen(buffer_to_send));
189   free(buffer_to_send);
190   list_free(window_out_bytes);
191   list_init(window_out_bytes);
192   return EXIT_SUCCESS;
193 }
194
195 int send_found_block_number(socket_t* skt, unsigned int index){
196   char block_found_indicator = BLOCK_FOUND_INDICATOR;
197
198   socket_send(skt, (char*)&block_found_indicator,
```

mar 27, 16 17:03 **server.c** Page 4/4

```
199     sizeof(block_found_indicator));
200   int block_number = index;
201
202   socket_send(skt, (char*)&block_number, sizeof(block_number));
203   return EXIT_SUCCESS;
204 }
205
206 int send_eof(socket_t* skt){
207   char eof_indicator = EOF_INDICATOR;
208
209   socket_send(skt, (char*)&eof_indicator, sizeof(eof_indicator));
210   return EXIT_SUCCESS;
211 }
```

mar 27, 16 17:03 **main.c** Page 1/1

```
1  /*
2   * main.c
3   *
4   *
5   *  Created on: Mar 10, 2016
6   *      Author: mastanca
7   */
8
9  #include <string.h>
10 #include "client.h"
11 #include "server.h"
12
13 #define CLIENT "client"
14 #define SERVER "server"
15
16 int main(int argc, char *argv[]){
17   char* execution_type = argv[1];
18   if (strcmp(execution_type, CLIENT) == 0){
19     return client_execution(argc, argv);
20   }else if (strcmp(execution_type, SERVER) == 0){
21     return server_execution(argc, argv);
22   } else {
23     return EXIT_FAILURE;
24   }
25   return EXIT_SUCCESS;
26 }
```

mar 27, 16 17:03 **list.h** Page 1/1

```c
/*
 * list.h
 *
 *  Created on: Mar 24, 2016
 *      Author: mastanca
 */

#ifndef SRC_LIST_H_
#define SRC_LIST_H_

#define LIST_INITIAL_CAPACITY 100

typedef struct list {
  int size;      // slots used so far
  int capacity;  // total available slots
  int *data;     // array of integers we're storing
} list_t;

void list_init(list_t *list);

void list_append(list_t *list, int value);

int list_get(list_t *list, int index);

void list_set(list_t *list, int index, int value);

void list_double_capacity_if_full(list_t *list);

void list_free(list_t *list);

#endif /* SRC_LIST_H_ */
```

mar 27, 16 17:03 **list.c** Page 1/1

```c
/*
 * list.c
 *
 *  Created on: Mar 24, 2016
 *      Author: mastanca
 */

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

#include "list.h"

void list_init(list_t *list) {
  list→size = 0;
  list→capacity = LIST_INITIAL_CAPACITY;
  list→data = malloc(sizeof(int) * list→capacity);
}

void list_append(list_t *list, int value) {
  list_double_capacity_if_full(list);
  list→data[list→size++] = value;
}

// Return the block at the given index
int list_get(list_t *list, int index) {
  if (index ≥ list→size ∨ index < 0) {
    printf("Index %d out of bounds for list of size %d\n", index,
        list→size);
    return -1;
  }
  return list→data[index];
}

void list_set(list_t *list, int index, int value) {
  while (index ≥ list→size) {
    list_append(list, 0);
  }

  list→data[index] = value;
}

void list_double_capacity_if_full(list_t *list) {
  if (list→size ≥ list→capacity) {
    list→capacity *= 2;
    list→data = realloc(list→data, sizeof(int) * list→capacity);
  }
}

void list_free(list_t *list) {
  free(list→data);
}
```

mar 27, 16 17:03 **lib_socket.h** Page 1/1

```c
/*
 * lib_socket.h
 *
 *  Created on: Mar 18, 2016
 *      Author: mastanca
 */

#ifndef SRC_LIB_SOCKET_H_
#define SRC_LIB_SOCKET_H_

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 1
#endif

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <stdbool.h>

typedef struct socket{
  int fd;
  struct addrinfo* result;
}socket_t;

int socket_init(socket_t* skt, char* hostname, char* port);
int socket_destroy(socket_t* skt);
int socket_bind(socket_t* skt);
int socket_listen(socket_t* skt, int max_clients);
int socket_accept(socket_t* skt, socket_t* client_skt);
int socket_connect(socket_t* skt);
int socket_receive(socket_t* skt, char* buffer, int size);
int socket_send(socket_t* skt, char* buffer, int size);

int handle_error(char* function_name);




#endif /* SRC_LIB_SOCKET_H_ */
```

mar 27, 16 17:03 **lib_socket.c** Page 1/3

```c
/*
 * lib_socket.c
 *
 *  Created on: Mar 18, 2016
 *      Author: mastanca
 */

#include "lib_socket.h"

int socket_init(socket_t* skt, char* hostname, char* port){
  int s = 0;
  struct addrinfo hints;
  int flag = 0;

  if (hostname ≡ NULL ∨ ¬strcmp(hostname, "127.0.0.1")){
    hostname = NULL;
    flag = AI_PASSIVE;
  }

  const char *serviceName = port;

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;       /* IPv4 (or AF_INET6 for IPv6)     */
  hints.ai_socktype = SOCK_STREAM; /* TCP  (or SOCK_DGRAM for UDP)    */
  hints.ai_flags = flag;      /* 0 (or AI_PASSIVE for server)           */

  s = getaddrinfo(hostname, serviceName, &hints, &skt→result);

  if (s ≠ 0) {
    fprintf(stderr, "Error in getaddrinfo: %s\n", gai_strerror(s));
    return 1;
  }

  skt→fd = socket(skt→result→ai_family, skt→result→ai_socktype,
      skt→result→ai_protocol);
  if (skt→fd ≡ −1){
    handle_error("init");
    return 1;
  }
  return EXIT_SUCCESS;
}

int socket_destroy(socket_t* skt){
  if (shutdown(skt→fd, SHUT_RDWR) ≡ −1){
    handle_error("destroy (shutdown)");
    return 1;
  }
  if (close(skt→fd) ≡ −1){
    handle_error("destroy (close)");
    return 1;
  }
  return EXIT_SUCCESS;
}

int socket_bind(socket_t* skt){
  if (bind(skt→fd, skt→result→ai_addr, skt→result→ai_addrlen) ≡ −1){
    handle_error("bind");
    close(skt→fd);
    freeaddrinfo(skt→result);
    return 1;
  }
  freeaddrinfo(skt→result);
  return EXIT_SUCCESS;
}

int socket_listen(socket_t* skt, int max_clients) {
```

mar 27, 16 17:03 **lib_socket.c** Page 2/3

```
67    if (listen(skt→fd, max_clients) ≡ −1){
68      handle_error("listen");
69      return 1;
70    }
71    return EXIT_SUCCESS;
72  }
73
74  int socket_accept(socket_t* skt, socket_t* client_skt) {
75    client_skt→fd = accept(skt→fd, NULL, NULL);
76    if (client_skt→fd ≡ −1){
77      handle_error("accept");
78      return 1;
79    }
80    return EXIT_SUCCESS;
81  }
82
83  int socket_connect(socket_t* skt) {
84    int s = 0;
85    struct addrinfo *ptr;
86    bool are_we_connected = false;
87    for (ptr = skt→result; ptr ≠ NULL ∧ are_we_connected ≡ false;
88        ptr = ptr→ai_next) {
89      s = connect(skt→fd, ptr→ai_addr, ptr→ai_addrlen);
90      if (s ≡ −1){
91        handle_error("connect");
92        close(skt→fd);
93        skt→fd = socket(ptr→ai_family, ptr→ai_socktype, ptr→ai_protocol);
94      }
95      are_we_connected = (s ≠ −1);
96    }
97    freeaddrinfo(skt→result);
98    if (are_we_connected ≡ false){
99      return EXIT_FAILURE;
100   }
101   return EXIT_SUCCESS;
102 }
103
104 int socket_receive(socket_t* skt, char* buffer, int size) {
105   int received = 0;
106   int response = 0;
107   bool is_a_valid_socket = true;
108
109   while (received < size ∧ is_a_valid_socket) {
110     response = recv(skt→fd, &buffer[received], size−received, MSG_NOSIGNAL);
111
112     if (response ≡ 0){
113       // Socket was closed
114       is_a_valid_socket = false;
115     }else if (response < 0) {
116       // There was an error
117       is_a_valid_socket = false;
118     } else {
119       received += response;
120     }
121   }
122
123   if (is_a_valid_socket) {
124     return received;
125   } else {
126     return −EXIT_FAILURE;
127   }
128
129   return EXIT_SUCCESS;
130 }
131
132 int socket_send(socket_t* skt, char* buffer, int size) {
```

mar 27, 16 17:03 **lib_socket.c** Page 3/3

```
133   int sent =0;
134   int response = 0;
135   bool is_a_valid_socket = true;
136
137   while (sent < size ∧ is_a_valid_socket) {
138     response = send(skt→fd, &buffer[sent], size−sent, MSG_NOSIGNAL);
139
140     if (response ≡ 0){
141       // Socket was closed
142       is_a_valid_socket = false;
143     }else if (response < 0) {
144       // There was an error
145       is_a_valid_socket = false;
146     } else {
147       sent += response;
148     }
149   }
150
151   if (is_a_valid_socket) {
152     return sent;
153   } else {
154     return −EXIT_FAILURE;
155   }
156
157   return EXIT_SUCCESS;
158 }
159
160 int handle_error(char* function_name){
161 //  fprintf(stderr, "Error on %s: ", function_name);
162 //  fprintf(stderr, "%s\n", strerror(errno));
163   return EXIT_SUCCESS;
164 }
165
```

mar 27, 16 17:03 **lib_checksum.h** Page 1/1

```c
/*
 * checksum.h
 *
 *  Created on: Mar 20, 2016
 *      Author: mastanca
 */

#ifndef SRC_LIB_CHECKSUM_H_
#define SRC_LIB_CHECKSUM_H_

#include <stddef.h>
#define M 0x00010000

typedef unsigned long ulong;

typedef struct checksum{
  ulong checksum;
  ulong lower;
  ulong higher;
} checksum_t;

int set_checksum(checksum_t* checksum, char* input, size_t size);
int rolling_checksum(checksum_t* new_checksum, checksum_t* old_checksum,
    char* buffer, size_t size);

#endif /* SRC_LIB_CHECKSUM_H_ */
```

mar 27, 16 17:03 **lib_checksum.c** Page 1/1

```c
/*
 * checksum.c
 *
 *  Created on: Mar 20, 2016
 *      Author: mastanca
 */

#include "lib_checksum.h"
#include <stdio.h>
#include <stdlib.h>

static int checksum_init(checksum_t* checksum){
  checksum->lower = 0;
  checksum->higher = 0;
  checksum->checksum = 0;
  return EXIT_SUCCESS;
}

static int set_checksum_result(checksum_t* checksum){
  checksum->lower %= M;
  checksum->higher %= M;
  checksum->checksum = checksum->lower + checksum->higher*M;
  return EXIT_SUCCESS;
}

// Stores the resulting checksum in checksum arg
int set_checksum(checksum_t* checksum, char* input, size_t size){
  checksum_init(checksum);

  for (int i = 0; i < size; ++i) {
    checksum->lower += input[i];
    checksum->higher += ((size-i)*input[i]);
  }

  set_checksum_result(checksum);

  return EXIT_SUCCESS;
}

// Rolling checksum assumes buffer is contiguous in memory
int rolling_checksum(checksum_t* new_checksum, checksum_t* old_checksum,
    char* buffer, size_t size){
  checksum_init(new_checksum);

  new_checksum->lower = ((old_checksum->lower - (ulong)buffer[-1] +
      (ulong)buffer[size-1])) % M;
  new_checksum->higher = old_checksum->higher - (size * (ulong)buffer[-1]) +
      new_checksum->lower;

  set_checksum_result(new_checksum);

  return EXIT_SUCCESS;
}

```

mar 27, 16 17:03 **file_handler.h** Page 1/1

```c
/*
 * file_handler.h
 *
 *  Created on: Mar 24, 2016
 *      Author: mastanca
 */

#ifndef SRC_FILE_HANDLER_H_
#define SRC_FILE_HANDLER_H_

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

int read_from_file(FILE* file, char* buffer, size_t block_size,
   bool* read_something);


#endif /* SRC_FILE_HANDLER_H_ */
```

mar 27, 16 17:03 **file_handler.c** Page 1/1

```c
/*
 * file_handler.c
 *
 *  Created on: Mar 24, 2016
 *      Author: mastanca
 */

#include "file_handler.h"

// Reads block_size chars from file and return result in buffer
int read_from_file(FILE* file, char* buffer, size_t block_size,
   bool* read_something){
  *read_something = false;
 char* tmp_buffer = calloc(block_size + 1, sizeof(char));
 if (¬feof(file)){
   int read_bytes = fread(tmp_buffer, 1, block_size, file);
   if (read_bytes ≠ 0){
     if (strlen(tmp_buffer) ≤ block_size){
       memset(buffer, 0, strlen(buffer));
       strncpy(buffer, tmp_buffer, strlen(tmp_buffer));
       *read_something = true;
     }
   }
 }
 free(tmp_buffer);
 return EXIT_SUCCESS;
}
```

mar 27, 16 17:03 **constants.h** Page 1/1

```c
/*
 * constants.h
 *
 *  Created on: Mar 25, 2016
 *      Author: mastanca
 */

#ifndef SRC_CONSTANTS_H_
#define SRC_CONSTANTS_H_

// Server constants
#define CHECKSUM_INDICATOR '1'
#define END_OF_LIST '2'
#define NEW_BYTES_INDICATOR '3'
#define BLOCK_FOUND_INDICATOR '4'
#define EOF_INDICATOR '5'
#define WINDOW_BYTE_DISPLACEMENT -1

// Client constants
#define CHECKSUM_INDICATOR '1'
#define END_OF_LIST '2'

#endif /* SRC_CONSTANTS_H_ */
```

mar 27, 16 17:03 **client.h** Page 1/1

```c
/*
 * client.h
 *
 *  Created on: Mar 20, 2016
 *      Author: mastanca
 */

#ifndef SRC_CLIENT_H_
#define SRC_CLIENT_H_

#include <stdio.h>
#include <string.h>
#include "constants.h"
#include "lib_socket.h"
#include "file_handler.h"
#include "lib_checksum.h"

typedef struct client{
  socket_t* skt;
  FILE* old_file;
  FILE* new_file;
  unsigned int block_size;
}client_t;

int client_execution(int argc, char* argv[]);
int send_remote_filename(socket_t* skt, char* filename,
    unsigned int block_size);
int send_file_chunks(client_t* client, FILE* file, unsigned int block_size);
int receive_new_bytes(client_t* client);
int receive_existing_block(client_t* client);
int receive_server_response(client_t* client);

#endif /* SRC_CLIENT_H_ */
```

mar 27, 16 17:03 **client.c** Page 1/3

```c

/*
 * client.c
 *
 *  Created on: Mar 20, 2016
 *      Author: mastanca
 */

#include "client.h"

int client_execution(int argc, char* argv[]){
  client_t client;
  char* hostname = argv[2];
  char* port = argv[3];

  char* old_file_name = argv[4];
  char* new_file_name = argv[5];
  char* remote_file_name = argv[6];
  client.block_size = atoi(argv[7]);

  socket_t skt;
  client.skt = &skt;
  socket_init(client.skt, hostname, port);

  if (socket_connect(client.skt) == 0){
    // Open new file
    client.new_file = NULL;
    client.new_file = fopen(new_file_name, "w");

    send_remote_filename(client.skt, remote_file_name, client.block_size);

    // Open old file
    client.old_file = NULL;
    client.old_file = fopen(old_file_name, "r");
    if (client.old_file != NULL){
      send_file_chunks(&client, client.old_file, client.block_size);
    }
    receive_server_response(&client);

    fclose(client.old_file);
    fclose(client.new_file);
  }
  socket_destroy(client.skt);
  return EXIT_SUCCESS;
}

int receive_server_response(client_t* client){
  // Receive server code
  char server_code = -1;
  while (server_code != EOF_INDICATOR){
    socket_receive(client->skt, (char*)&server_code, sizeof(char));

    if (server_code == NEW_BYTES_INDICATOR){
      receive_new_bytes(client);
    } else if (server_code == BLOCK_FOUND_INDICATOR){
      receive_existing_block(client);
    }
  }

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

  return EXIT_SUCCESS;
}

int receive_new_bytes(client_t* client){
  int new_bytes_longitude = 0;
```

mar 27, 16 17:03 **client.c** Page 2/3

```c
  socket_receive(client->skt, (char*)&new_bytes_longitude,
      sizeof(new_bytes_longitude));
  // Weird bug when using stack, so malloc!
  char* new_bytes_buffer = malloc(new_bytes_longitude);
  memset(new_bytes_buffer, 0, new_bytes_longitude);
  socket_receive(client->skt, new_bytes_buffer, new_bytes_longitude);

  printf("RECV File chunk %i bytes\n", new_bytes_longitude);
  fwrite(new_bytes_buffer, sizeof(char), new_bytes_longitude, client->new_file);
  free(new_bytes_buffer);
  return EXIT_SUCCESS;
}

int receive_existing_block(client_t* client){
  int existing_block_index = -1;
  socket_receive(client->skt, (char*)&existing_block_index,
      sizeof(existing_block_index));

  printf("RECV Block index %i\n", existing_block_index);
  fseek(client->old_file, client->block_size * existing_block_index,
      SEEK_SET);

  char* old_bytes_buffer = calloc(client->block_size + 1, sizeof(char));
  bool read_something = false;
  read_from_file(client->old_file, old_bytes_buffer, client->block_size,
     &read_something);
  fwrite(old_bytes_buffer, sizeof(char), strlen(old_bytes_buffer),
   client->new_file);
   free(old_bytes_buffer);
  return EXIT_SUCCESS;
}

int send_remote_filename(socket_t* skt, char* filename,
    unsigned int block_size){
  int filename_length = strlen(filename);
  char *buffer = malloc(filename_length + 2 * sizeof(int));

  memcpy(buffer, &filename_length, sizeof(int));
  memcpy(buffer + sizeof(int), filename, filename_length);
  memcpy(buffer + sizeof(int) + filename_length, &block_size, sizeof(int));

  socket_send(skt, buffer, filename_length + 2 * sizeof(int));

  free(buffer);

  return EXIT_SUCCESS;
}

int send_file_chunks(client_t* client, FILE* file, unsigned int block_size){
  bool read_something = false;
  checksum_t checksum;
  char* buffer = calloc(block_size + 1, sizeof(char));
  while(!feof(file)){
    read_from_file(file, buffer, block_size, &read_something);
    if (strcmp(buffer, "") != 0) {
      char code = CHECKSUM_INDICATOR;

      socket_send(client->skt, (char*)&code, sizeof(code));
      set_checksum(&checksum, buffer, block_size);
      int number_to_send = checksum.checksum;

      socket_send(client->skt, (char*)&number_to_send, sizeof(number_to_send));
      memset(buffer, 0, strlen(buffer));
    }
  }
  int code = END_OF_LIST;
```

mar 27, 16 17:03 **client.c** Page 3/3

```
133     socket_send(client→skt, (char*)&code, sizeof(code));
134     free(buffer);
135     return EXIT_SUCCESS;
136 }
137
138
```

mar 27, 16 17:03 **Table of Content** Page 1/1