

mar 27, 16 17:03

server.h

Page 1/1

```

1  /*
2   * server.h
3   *
4   * Created on: Mar 20, 2016
5   * Author: mastanca
6   */
7
8  #ifndef SRC_SERVER_H_
9  #define SRC_SERVER_H_
10
11 #include <stdio.h>
12 #include <string.h>
13 #include "constants.h"
14 #include "lib_socket.h"
15 #include "list.h"
16 #include "lib_checksum.h"
17 #include "file_handler.h"
18
19 typedef struct server{
20     socket_t* skt;
21     unsigned int block_size;
22     FILE* remote_file;
23     list_t checksum_list;
24 }server_t;
25
26 int server_execution(int argc, char* argv[]);
27 int receive_remote_filename(socket_t* skt, server_t* server);
28 int receive_checksum_list(socket_t* skt, unsigned int block_size,
29     server_t* server);
30 int start_comparison_sequence(server_t* server, socket_t* skt);
31 int checksum_not_found(char* block, list_t* window_out_bytes, server_t* server,
32     checksum_t* checksum);
33 int send_windowed_bytes(list_t* window_out_bytes, server_t* server,
34     socket_t* client_skt);
35 int send_found_block_number(socket_t* client_skt, unsigned int index);
36 int send_eof(socket_t* client_skt);
37
38 #endif /* SRC_SERVER_H_ */

```

mar 27, 16 17:03

server.c

Page 1/4

```

1  /*
2   * server.c
3   *
4   * Created on: Mar 20, 2016
5   * Author: mastanca
6   */
7
8  #include "server.h"
9
10 int server_execution(int argc, char* argv[]){
11     if (argc != 3){
12         return 1;
13     }
14     char* port = argv[2];
15     socket_t acep;
16     server_t server;
17     server.skt = &acep;
18
19     list_t list;
20     // Compiler warning if this values are not zero before list init
21     list.capacity = 0;
22     list.size = 0;
23     list.data = NULL;
24     server.checksum_list = list;
25     list_init(&server.checksum_list);
26
27     socket_init(server.skt, NULL, port);
28     // // Avoid time wait
29     // int option = 1;
30     // setsockopt(server.skt->fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
31     //     (char*)&option, sizeof(option));
32
33     socket_bind(server.skt);
34
35     socket_listen(server.skt, 5);
36     socket_t client_skt;
37     socket_accept(server.skt, &client_skt);
38
39     receive_remote_filename(&client_skt, &server);
40
41     receive_checksum_list(&client_skt, server.block_size, &server);
42
43     start_comparison_sequence(&server, &client_skt);
44
45     socket_destroy(&client_skt);
46
47     socket_destroy(server.skt);
48
49     fclose(server.remote_file);
50     // Free checksum list
51     list_free(&server.checksum_list);
52     return EXIT_SUCCESS;
53 }
54
55 int start_comparison_sequence(server_t* server, socket_t* skt){
56     bool read_something = false;
57     list_t window_out_bytes;
58     list_init(&window_out_bytes);
59
60     // Load new block from file
61     char* block = calloc(server->block_size + 1, sizeof(char));
62     read_from_file(server->remote_file, block, server->block_size,
63         &read_something);
64
65     // Get checksum of the new block
66     checksum_t checksum;

```

mar 27, 16 17:03

server.c

Page 2/4

```

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){

```

mar 27, 16 17:03

server.c

Page 3/4

```

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

```

1  /*
2  * list.h
3  *
4  * Created on: Mar 24, 2016
5  * Author: mastanca
6  */
7
8  #ifndef SRC_LIST_H_
9  #define SRC_LIST_H_
10
11 #define LIST_INITIAL_CAPACITY 100
12
13 typedef struct list {
14     int size; // slots used so far
15     int capacity; // total available slots
16     int *data; // array of integers we're storing
17 } list_t;
18
19 void list_init(list_t *list);
20
21 void list_append(list_t *list, int value);
22
23 int list_get(list_t *list, int index);
24
25 void list_set(list_t *list, int index, int value);
26
27 void list_double_capacity_if_full(list_t *list);
28
29 void list_free(list_t *list);
30
31 #endif /* SRC_LIST_H_ */

```

mar 27, 16 17:03

list.c

Page 1/1

```

1  /*
2  * list.c
3  *
4  * Created on: Mar 24, 2016
5  * Author: mastanca
6  */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 #include "list.h"
12
13 void list_init(list_t *list) {
14     list->size = 0;
15     list->capacity = LIST_INITIAL_CAPACITY;
16     list->data = malloc(sizeof(int) * list->capacity);
17 }
18
19 void list_append(list_t *list, int value) {
20     list_double_capacity_if_full(list);
21     list->data[list->size++] = value;
22 }
23
24 // Return the block at the given index
25 int list_get(list_t *list, int index) {
26     if (index ≥ list->size ∨ index < 0) {
27         printf("Index %d out of bounds for list of size %d\n", index,
28             list->size);
29         return -1;
30     }
31     return list->data[index];
32 }
33
34 void list_set(list_t *list, int index, int value) {
35     while (index ≥ list->size) {
36         list_append(list, 0);
37     }
38     list->data[index] = value;
39 }
40
41 void list_double_capacity_if_full(list_t *list) {
42     if (list->size ≥ list->capacity) {
43         list->capacity *= 2;
44         list->data = realloc(list->data, sizeof(int) * list->capacity);
45     }
46 }
47
48 void list_free(list_t *list) {
49     free(list->data);
50 }
51

```

mar 27, 16 17:03

lib_socket.h

Page 1/1

```

1  /*
2   * lib_socket.h
3   *
4   * Created on: Mar 18, 2016
5   * Author: mastanca
6   */
7
8  #ifndef SRC_LIB_SOCKET_H_
9  #define SRC_LIB_SOCKET_H_
10
11 #ifndef _POSIX_C_SOURCE
12 #define _POSIX_C_SOURCE 1
13 #endif
14
15 #include <sys/types.h>
16 #include <sys/socket.h>
17 #include <netdb.h>
18 #include <stdlib.h>
19 #include <stdio.h>
20 #include <string.h>
21 #include <unistd.h>
22 #include <errno.h>
23 #include <stdbool.h>
24
25 typedef struct socket{
26     int fd;
27     struct addrinfo* result;
28 }socket_t;
29
30 int socket_init(socket_t* skt, char* hostname, char* port);
31 int socket_destroy(socket_t* skt);
32 int socket_bind(socket_t* skt);
33 int socket_listen(socket_t* skt, int max_clients);
34 int socket_accept(socket_t* skt, socket_t* client_skt);
35 int socket_connect(socket_t* skt);
36 int socket_receive(socket_t* skt, char* buffer, int size);
37 int socket_send(socket_t* skt, char* buffer, int size);
38
39 int handle_error(char* function_name);
40
41
42
43
44 #endif /* SRC_LIB_SOCKET_H_ */

```

mar 27, 16 17:03

lib_socket.c

Page 1/3

```

1  /*
2   * lib_socket.c
3   *
4   * Created on: Mar 18, 2016
5   * Author: mastanca
6   */
7
8  #include "lib_socket.h"
9
10 int socket_init(socket_t* skt, char* hostname, char* port){
11     int s = 0;
12     struct addrinfo hints;
13     int flag = 0;
14
15     if (hostname == NULL || strcmp(hostname, "127.0.0.1")){
16         hostname = NULL;
17         flag = AI_PASSIVE;
18     }
19
20     const char *serviceName = port;
21
22     memset(&hints, 0, sizeof(struct addrinfo));
23     hints.ai_family = AF_INET; /* IPv4 (or AF_INET6 for IPv6) */
24     hints.ai_socktype = SOCK_STREAM; /* TCP (or SOCK_DGRAM for UDP) */
25     hints.ai_flags = flag; /* 0 (or AI_PASSIVE for server) */
26
27     s = getaddrinfo(hostname, serviceName, &hints, &skt->result);
28
29     if (s != 0) {
30         fprintf(stderr, "Error in getaddrinfo: %s\n", gai_strerror(s));
31         return 1;
32     }
33
34     skt->fd = socket(skt->result->ai_family, skt->result->ai_socktype,
35                     skt->result->ai_protocol);
36     if (skt->fd == -1){
37         handle_error("init");
38         return 1;
39     }
40     return EXIT_SUCCESS;
41 }
42
43 int socket_destroy(socket_t* skt){
44     if (shutdown(skt->fd, SHUT_RDWR) == -1){
45         handle_error("destroy (shutdown)");
46         return 1;
47     }
48     if (close(skt->fd) == -1){
49         handle_error("destroy (close)");
50         return 1;
51     }
52     return EXIT_SUCCESS;
53 }
54
55 int socket_bind(socket_t* skt){
56     if (bind(skt->fd, skt->result->ai_addr, skt->result->ai_addrlen) == -1){
57         handle_error("bind");
58         close(skt->fd);
59         freeaddrinfo(skt->result);
60         return 1;
61     }
62     freeaddrinfo(skt->result);
63     return EXIT_SUCCESS;
64 }
65
66 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

```

1  /*
2   * checksum.h
3   *
4   * Created on: Mar 20, 2016
5   * Author: mastanca
6   */
7
8  #ifndef SRC_LIB_CHECKSUM_H_
9  #define SRC_LIB_CHECKSUM_H_
10
11 #include <stddef.h>
12 #define M 0x00010000
13
14 typedef unsigned long ulong;
15
16 typedef struct checksum{
17     ulong checksum;
18     ulong lower;
19     ulong higher;
20 } checksum_t;
21
22 int set_checksum(checksum_t* checksum, char* input, size_t size);
23 int rolling_checksum(checksum_t* new_checksum, checksum_t* old_checksum,
24     char* buffer, size_t size);
25
26 #endif /* SRC_LIB_CHECKSUM_H_ */

```

mar 27, 16 17:03

lib_checksum.c

Page 1/1

```

1  /*
2   * checksum.c
3   *
4   * Created on: Mar 20, 2016
5   * Author: mastanca
6   */
7
8  #include "lib_checksum.h"
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 static int checksum_init(checksum_t* checksum){
13     checksum->lower = 0;
14     checksum->higher = 0;
15     checksum->checksum = 0;
16     return EXIT_SUCCESS;
17 }
18
19 static int set_checksum_result(checksum_t* checksum){
20     checksum->lower %= M;
21     checksum->higher %= M;
22     checksum->checksum = checksum->lower + checksum->higher*M;
23     return EXIT_SUCCESS;
24 }
25
26 // Stores the resulting checksum in checksum arg
27 int set_checksum(checksum_t* checksum, char* input, size_t size){
28     checksum_init(checksum);
29
30     for (int i = 0; i < size; ++i) {
31         checksum->lower += input[i];
32         checksum->higher += ((size-i)*input[i]);
33     }
34
35     set_checksum_result(checksum);
36
37     return EXIT_SUCCESS;
38 }
39
40 // Rolling checksum assumes buffer is contiguous in memory
41 int rolling_checksum(checksum_t* new_checksum, checksum_t* old_checksum,
42     char* buffer, size_t size){
43     checksum_init(new_checksum);
44
45     new_checksum->lower = ((old_checksum->lower - (ulong)buffer[-1] +
46         (ulong)buffer[size-1])) % M;
47     new_checksum->higher = old_checksum->higher - (size * (ulong)buffer[-1]) +
48         new_checksum->lower;
49
50     set_checksum_result(new_checksum);
51
52     return EXIT_SUCCESS;
53 }
54

```

mar 27, 16 17:03

file_handler.h

Page 1/1

```

1  /*
2   * file_handler.h
3   *
4   * Created on: Mar 24, 2016
5   * Author: mastanca
6   */
7
8  #ifndef SRC_FILE_HANDLER_H_
9  #define SRC_FILE_HANDLER_H_
10
11 #include <stdio.h>
12 #include <errno.h>
13 #include <string.h>
14 #include <unistd.h>
15 #include <stdlib.h>
16 #include <stdbool.h>
17
18 int read_from_file(FILE* file, char* buffer, size_t block_size,
19     bool* read_something);
20
21
22 #endif /* SRC_FILE_HANDLER_H_ */

```

mar 27, 16 17:03

file_handler.c

Page 1/1

```

1  /*
2   * file_handler.c
3   *
4   * Created on: Mar 24, 2016
5   * Author: mastanca
6   */
7
8  #include "file_handler.h"
9
10 // Reads block_size chars from file and return result in buffer
11 int read_from_file(FILE* file, char* buffer, size_t block_size,
12     bool* read_something){
13     *read_something = false;
14     char* tmp_buffer = calloc(block_size + 1, sizeof(char));
15     if (!feof(file)){
16         int read_bytes = fread(tmp_buffer, 1, block_size, file);
17         if (read_bytes != 0){
18             if (strlen(tmp_buffer) <= block_size){
19                 memset(buffer, 0, strlen(buffer));
20                 strncpy(buffer, tmp_buffer, strlen(tmp_buffer));
21                 *read_something = true;
22             }
23         }
24     }
25     free(tmp_buffer);
26     return EXIT_SUCCESS;
27 }

```


mar 27, 16 17:03

constants.h

Page 1/1

```

1  /*
2  *  constants.h
3  *
4  *   Created on: Mar 25, 2016
5  *   Author: mastanca
6  */
7
8  #ifndef SRC_CONSTANTS_H_
9  #define SRC_CONSTANTS_H_
10
11 // Server constants
12 #define CHECKSUM_INDICATOR '1'
13 #define END_OF_LIST '2'
14 #define NEW_BYTES_INDICATOR '3'
15 #define BLOCK_FOUND_INDICATOR '4'
16 #define EOF_INDICATOR '5'
17 #define WINDOW_BYTE_DISPLACEMENT -1
18
19 // Client constants
20 #define CHECKSUM_INDICATOR '1'
21 #define END_OF_LIST '2'
22
23 #endif /* SRC_CONSTANTS_H_ */

```

mar 27, 16 17:03

client.h

Page 1/1

```

1  /*
2  *  client.h
3  *
4  *   Created on: Mar 20, 2016
5  *   Author: mastanca
6  */
7
8  #ifndef SRC_CLIENT_H_
9  #define SRC_CLIENT_H_
10
11 #include <stdio.h>
12 #include <string.h>
13 #include "constants.h"
14 #include "lib_socket.h"
15 #include "file_handler.h"
16 #include "lib_checksum.h"
17
18 typedef struct client{
19     socket_t* skt;
20     FILE* old_file;
21     FILE* new_file;
22     unsigned int block_size;
23 }client_t;
24
25 int client_execution(int argc, char* argv[]);
26 int send_remote_filename(socket_t* skt, char* filename,
27     unsigned int block_size);
28 int send_file_chunks(client_t* client, FILE* file, unsigned int block_size);
29 int receive_new_bytes(client_t* client);
30 int receive_existing_block(client_t* client);
31 int receive_server_response(client_t* client);
32
33 #endif /* SRC_CLIENT_H_ */

```

mar 27, 16 17:03

client.c

Page 1/3

```

1  /*
2  * client.c
3  *
4  * Created on: Mar 20, 2016
5  * Author: mastanca
6  */
7
8  #include "client.h"
9
10 int client_execution(int argc, char* argv[]){
11     client_t client;
12     char* hostname = argv[2];
13     char* port = argv[3];
14
15     char* old_file_name = argv[4];
16     char* new_file_name = argv[5];
17     char* remote_file_name = argv[6];
18     client.block_size = atoi(argv[7]);
19
20     socket_t skt;
21     client.skt = &skt;
22     socket_init(client.skt, hostname, port);
23
24     if (socket_connect(client.skt) == 0){
25         // Open new file
26         client.new_file = NULL;
27         client.new_file = fopen(new_file_name, "w");
28
29         send_remote_filename(client.skt, remote_file_name, client.block_size);
30
31         // Open old file
32         client.old_file = NULL;
33         client.old_file = fopen(old_file_name, "r");
34         if (client.old_file != NULL){
35             send_file_chunks(&client, client.old_file, client.block_size);
36         }
37         receive_server_response(&client);
38
39         fclose(client.old_file);
40         fclose(client.new_file);
41     }
42     socket_destroy(client.skt);
43     return EXIT_SUCCESS;
44 }
45
46 int receive_server_response(client_t* client){
47     // Receive server code
48     char server_code = -1;
49     while (server_code != EOF_INDICATOR){
50         socket_receive(client->skt, (char*)&server_code, sizeof(char));
51
52         if (server_code == NEW_BYTES_INDICATOR){
53             receive_new_bytes(client);
54         } else if (server_code == BLOCK_FOUND_INDICATOR){
55             receive_existing_block(client);
56         }
57     }
58
59     printf("RECV End of file\n");
60
61     return EXIT_SUCCESS;
62 }
63
64 int receive_new_bytes(client_t* client){
65     int new_bytes_longitude = 0;

```

mar 27, 16 17:03

client.c

Page 2/3

```

67     socket_receive(client->skt, (char*)&new_bytes_longitude,
68         sizeof(new_bytes_longitude));
69     // Weird bug when using stack, so malloc!
70     char* new_bytes_buffer = malloc(new_bytes_longitude);
71     memset(new_bytes_buffer, 0, new_bytes_longitude);
72     socket_receive(client->skt, new_bytes_buffer, new_bytes_longitude);
73
74     printf("RECV File chunk %i bytes\n", new_bytes_longitude);
75     fwrite(new_bytes_buffer, sizeof(char), new_bytes_longitude, client->new_file);
76     free(new_bytes_buffer);
77     return EXIT_SUCCESS;
78 }
79
80 int receive_existing_block(client_t* client){
81     int existing_block_index = -1;
82     socket_receive(client->skt, (char*)&existing_block_index,
83         sizeof(existing_block_index));
84
85     printf("RECV Block index %i\n", existing_block_index);
86     fseek(client->old_file, client->block_size * existing_block_index,
87         SEEK_SET);
88
89     char* old_bytes_buffer = calloc(client->block_size + 1, sizeof(char));
90     bool read_something = false;
91     read_from_file(client->old_file, old_bytes_buffer, client->block_size,
92         &read_something);
93     fwrite(old_bytes_buffer, sizeof(char), strlen(old_bytes_buffer),
94         client->new_file);
95     free(old_bytes_buffer);
96     return EXIT_SUCCESS;
97 }
98
99 int send_remote_filename(socket_t* skt, char* filename,
100     unsigned int block_size){
101     int filename_length = strlen(filename);
102     char *buffer = malloc(filename_length + 2 * sizeof(int));
103
104     memcpy(buffer, &filename_length, sizeof(int));
105     memcpy(buffer + sizeof(int), filename, filename_length);
106     memcpy(buffer + sizeof(int) + filename_length, &block_size, sizeof(int));
107
108     socket_send(skt, buffer, filename_length + 2 * sizeof(int));
109
110     free(buffer);
111
112     return EXIT_SUCCESS;
113 }
114
115 int send_file_chunks(client_t* client, FILE* file, unsigned int block_size){
116     bool read_something = false;
117     checksum_t checksum;
118     char* buffer = calloc(block_size + 1, sizeof(char));
119     while (!feof(file)){
120         read_from_file(file, buffer, block_size, &read_something);
121         if (strcmp(buffer, "") != 0) {
122             char code = CHECKSUM_INDICATOR;
123
124             socket_send(client->skt, (char*)&code, sizeof(code));
125             set_checksum(&checksum, buffer, block_size);
126             int number_to_send = checksum.checksum;
127
128             socket_send(client->skt, (char*)&number_to_send, sizeof(number_to_send));
129             memset(buffer, 0, strlen(buffer));
130         }
131     }
132     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

1

Table of Contents

2

1 server.h..... sheets

1 to

1 (1) pages

1-

1

39 lines

3

2 server.c..... sheets

1 to

3 (3) pages

2-

5

212 lines

4

3 main.c..... sheets

3 to

3 (1) pages

6-

6

27 lines

5

4 list.h..... sheets

4 to

4 (1) pages

7-

7

32 lines

6

5 list.c..... sheets

4 to

4 (1) pages

8-

8

52 lines

7

6 lib_socket.h..... sheets

5 to

5 (1) pages

9-

9

45 lines

8

7 lib_socket.c..... sheets

5 to

6 (2) pages

10-

12

166 lines

9

8 lib_checksum.h..... sheets

7 to

7 (1) pages

13-

13

27 lines

10

9 lib_checksum.c..... sheets

7 to

7 (1) pages

14-

14

55 lines

11

10 file_handler.h..... sheets

8 to

8 (1) pages

15-

15

23 lines

12

11 file_handler.c..... sheets

8 to

8 (1) pages

16-

16

28 lines

13

12 constants.h..... sheets

9 to

9 (1) pages

17-

17

24 lines

14

13 client.h..... sheets

9 to

9 (1) pages

18-

18

34 lines

15

14 client.c..... sheets

10 to

11 (2) pages

19-

21

139 lines