```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_ */
```

```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;
  set_checksum(&checksum, block, strlen(block));

  while(¬feof(server→remote_file)){
    int i = 0;
    int found_index = 0;
    bool found = false;
    while(i < server→checksum_list.size ∧ ¬found){
      int i_element = list_get(&server→checksum_list, i);
      if(checksum.checksum ≡ i_element){
        found = true;
        found_index = i;
      }
      i++;
    }
    if (¬found){
      checksum_not_found(block, &window_out_bytes, server, &checksum);
    }else{
      if (window_out_bytes.size > 0){
        send_windowed_bytes(&window_out_bytes, server, skt);
      }
      send_found_block_number(skt, found_index);
      read_from_file(server→remote_file, block, strlen(block),
       &read_something);
      set_checksum(&checksum, block, strlen(block));
    }
  }
  // If there are remaining windowed bytes send them
  if (window_out_bytes.size > 0 ∨ ((strlen(block) > 0) ∧
   (read_something ≡ true))){
    for (int i = 0; i < strlen(block); ++i) {
      char remaining_char = block[i];
      list_append(&window_out_bytes, remaining_char);
    }
    send_windowed_bytes(&window_out_bytes, server, skt);
  }
  free(block);
  list_free(&window_out_bytes);
  send_eof(skt);

  return EXIT_SUCCESS;
}

int receive_remote_filename(socket_t* skt, server_t* server){
  int filename_length;
  int block_size;

  socket_receive(skt, (char*)&filename_length, sizeof(int));

  char *name = malloc(filename_length + 1);
  socket_receive(skt, name, filename_length);
  name[filename_length] = 0;

  socket_receive(skt, (char*)&block_size, sizeof(int));

  server→block_size = block_size;

  // Open remote file here and assign to server_t
  server→remote_file = fopen(name, "r");

  free(name);

  return EXIT_SUCCESS;
}

int receive_checksum_list(socket_t* skt, unsigned int block_size,
    server_t* server){
```

```
  char code = '\0';
  int checksum = 0;
  while (code ≠ END_OF_LIST){
    socket_receive(skt, (char*)&code, sizeof(code));

    if (CHECKSUM_INDICATOR ≡ code){
      socket_receive(skt, (char*)&checksum, sizeof(checksum));
      list_append(&(server→checksum_list), checksum);
    }
  }
  return EXIT_SUCCESS;
}

int checksum_not_found(char* block, list_t* window_out_bytes, server_t* server,
    checksum_t* checksum){
  char byte_to_window = block[0];
  list_append(window_out_bytes, byte_to_window);

  // Move cursor block size bytes to the left and return 1
  int index = WINDOW_BYTE_DISPLACEMENT * (server→block_size) +
   (-1 * WINDOW_BYTE_DISPLACEMENT);
  fseek(server→remote_file, index, SEEK_CUR);
  bool read_something = false;
  read_from_file(server→remote_file, block, server→block_size,
     &read_something);
  char* rolling_buffer = calloc(server→block_size + 1, sizeof(char));
  rolling_buffer[0] = byte_to_window;
  memcpy(rolling_buffer + strlen(rolling_buffer), block, strlen(block));
  checksum_t old_checksum;
  old_checksum = *checksum;

  rolling_checksum(checksum, &old_checksum, rolling_buffer +1,
      server→block_size);

  free(rolling_buffer);
  return EXIT_SUCCESS;
}

int send_windowed_bytes(list_t* window_out_bytes, server_t* server,
    socket_t* skt){
  char* buffer_to_send = calloc(window_out_bytes→size + 1, sizeof(char));
  for (int i = 0; i < window_out_bytes→size; ++i) {
    char i_element = list_get(window_out_bytes, i);
    strncat(buffer_to_send, &i_element, sizeof(char));
  }
  char new_bytes_indicator = NEW_BYTES_INDICATOR;

  socket_send(skt, (char*)&new_bytes_indicator, sizeof(new_bytes_indicator));

  // Send 4 bytes with the length of the new bytes
  int new_bytes_size = strlen(buffer_to_send);

  socket_send(skt, (char*)&new_bytes_size, sizeof(new_bytes_size));

  // Send the actual bytes
  socket_send(skt, buffer_to_send, strlen(buffer_to_send));
  free(buffer_to_send);
  list_free(window_out_bytes);
  list_init(window_out_bytes);
  return EXIT_SUCCESS;
}

int send_found_block_number(socket_t* skt, unsigned int index){
  char block_found_indicator = BLOCK_FOUND_INDICATOR;

  socket_send(skt, (char*)&block_found_indicator,
```

```c
     sizeof(block_found_indicator));
   int block_number = index;

   socket_send(skt, (char*)&block_number, sizeof(block_number));
   return EXIT_SUCCESS;
 }

 int send_eof(socket_t* skt){
   char eof_indicator = EOF_INDICATOR;

   socket_send(skt, (char*)&eof_indicator, sizeof(eof_indicator));
   return EXIT_SUCCESS;
 }
```

```c
/*
 * main.c

 *
 *  Created on: Mar 10, 2016
 *      Author: mastanca
 */

#include <string.h>
#include "client.h"
#include "server.h"

#define CLIENT "client"
#define SERVER "server"

int main(int argc, char *argv[]){
  char* execution_type = argv[1];
  if (strcmp(execution_type, CLIENT) ≡ 0){
    return client_execution(argc, argv);
  }else if (strcmp(execution_type, SERVER) ≡ 0){
    return server_execution(argc, argv);
  } else {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```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_ */
```

```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);
}
```

```
/*
 * 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_ */
```

```
/*
 * 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) {
```

```c
  if (listen(skt→fd, max_clients) ≡ −1){
    handle_error("listen");
    return 1;
  }
  return EXIT_SUCCESS;
}

int socket_accept(socket_t* skt, socket_t* client_skt) {
  client_skt→fd = accept(skt→fd, NULL, NULL);
  if (client_skt→fd ≡ −1){
    handle_error("accept");
    return 1;
  }
  return EXIT_SUCCESS;
}

int socket_connect(socket_t* skt) {
  int s = 0;
  struct addrinfo *ptr;
  bool are_we_connected = false;
  for (ptr = skt→result; ptr ≠ NULL ∧ are_we_connected ≡ false;
      ptr = ptr→ai_next) {
    s = connect(skt→fd, ptr→ai_addr, ptr→ai_addrlen);
    if (s ≡ −1){
      handle_error("connect");
      close(skt→fd);
      skt→fd = socket(ptr→ai_family, ptr→ai_socktype, ptr→ai_protocol);
    }
    are_we_connected = (s ≠ −1);
  }
  freeaddrinfo(skt→result);
  if (are_we_connected ≡ false){
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}

int socket_receive(socket_t* skt, char* buffer, int size) {
  int received = 0;
  int response = 0;
  bool is_a_valid_socket = true;

  while (received < size ∧ is_a_valid_socket) {
    response = recv(skt→fd, &buffer[received], size−received, MSG_NOSIGNAL);

    if (response ≡ 0){
      // Socket was closed
      is_a_valid_socket = false;
    }else if (response < 0) {
      // There was an error
      is_a_valid_socket = false;
    } else {
      received += response;
    }
  }

  if (is_a_valid_socket) {
    return received;
  } else {
    return −EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}

int socket_send(socket_t* skt, char* buffer, int size) {
```

```c
  int sent =0;
  int response = 0;
  bool is_a_valid_socket = true;

  while (sent < size ∧ is_a_valid_socket) {
    response = send(skt→fd, &buffer[sent], size−sent, MSG_NOSIGNAL);

    if (response ≡ 0){
      // Socket was closed
      is_a_valid_socket = false;
    }else if (response < 0) {
      // There was an error
      is_a_valid_socket = false;
    } else {
      sent += response;
    }
  }

  if (is_a_valid_socket) {
    return sent;
  } else {
    return −EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}

int handle_error(char* function_name){
//  fprintf(stderr, "Error on %s: ", function_name);
//  fprintf(stderr, "%s\n", strerror(errno));
  return EXIT_SUCCESS;
}
```

```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_ */
```

```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;
}
```

```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_ */
```

```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_ */
```

```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;
```

```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;
```

```c
    socket_send(client->skt, (char*)&code, sizeof(code));
    free(buffer);
    return EXIT_SUCCESS;
}


```

Table of Contents