```
#ifndef THCONNECTION_H_
#define THCONNECTION_H_

#include "commonThread.h"
#include "serverConnection.h"


class ThConnection : public Thread {
    private:
        Connection connection;
        bool dead;

    public:
        bool is_dead();
        ThConnection(Socket peer, Index & index);
        virtual void run();
        void stop();
};
#endif
```

```
#include "serverThConnection.h"

void Thread::start() {
    thread = std::thread(&Thread::run, this);
}

void Thread::join() {
    thread.join();
}

ThConnection::ThConnection(Socket peer, Index & index)://
 connection(std::move(peer), index){}

bool ThConnection::is_dead() {
  return this→dead;
}
void ThConnection::run() {
    dead = false;
    this→connection.run();
    dead = true;
}

void ThConnection::stop() {
  this→connection.stop();
}
```

```
#include <mutex>

class Lock {
private:
    std::mutex &m;

public:
    explicit Lock(std::mutex &m);

    ~Lock();

private:
    Lock(const Lock&) = delete;
  Lock& operator=(const Lock&) = delete;
  Lock(Lock∧) = delete;
  Lock& operator=(Lock∧) = delete;
};
```

```
#include "serverLock.h"


Lock::Lock(std::mutex &m) : m(m) {
    m.lock();
}

Lock::~Lock() {
    m.unlock();
}
```

```cpp
#include "serverConnection.h"
#include "serverThConnection.h"
#include <thread>
#include <vector>

class Server_listener{
private:
  Socket socket;
  Index index;
  std::thread principal_thread;
  std::vector<ThConnection*> client_threads;
  bool continue_listening;
  void listen();
public:
  //constructor que crea un socket de servidor
  Server_listener(char port[], char file_name[]);

  //comienza a escuchar conecciones
  void start_listening();

  //finaliza las conecciones
  void stop_listening();
};
```

```cpp
#include "serverListener.h"
#include <vector>
Server_listener::Server_listener(char port[], char file_name[])://
 socket(NULL,port), index(file_name){
  this->continue_listening = true;
  this->socket.bind_and_listen();
}

void Server_listener::start_listening(){
  this->principal_thread = std::thread(&Server_listener::listen, this);
}

void Server_listener::listen(){
  while(this->continue_listening){
    try{
      Socket peer = this->socket.accept_socket();
      client_threads.push_back(new ThConnection(std::move(peer), this->index));
      client_threads.back()->start();
    } catch(Error e){
      //cerraron el socket
    }
    }
}

void Server_listener::stop_listening(){
  this->continue_listening = false;

  this->socket.stop();

  for(unsigned int i = 0; i< client_threads.size(); i++){
    client_threads[i]->stop();
    client_threads[i]->join();
    delete client_threads[i];
  }

    this->principal_thread.join();
    this->index.close();
}
```

```
#ifndef __INDEX_H__
#define __INDEX_H__

#define FILE_TYPE "f"
#define TAG_TYPE "t"

#include <fstream>
#include <string>
#include <sstream>
#include <algorithm>
#include <map>
#include <set>
#include <mutex>
#include "commonError.h"
#include "serverLock.h"
using std::cout;
using std::ios;


struct cmp{
    bool operator()(const std::string& s1, const std::string& s2) const{
        for (std::string::const_iterator it1 = s1.begin(), it2 =  s2.begin();//
            it1≠s1.end() ∧ it2≠s2.end(); ++it1 , ++it2 ){
            if ((*it1) ≡ (*it2)){
                continue;
            }
            if (std::isdigit(*it1) ∧ ¬std::isdigit(*it2)){
                return true;
            }
            if (¬std::isdigit(*it1) ∧ std::isdigit(*it2)){
               return false;
            }
            if (std::isdigit(*it1) ∧ std::isdigit(*it2)) {
                int n1 = (*it1) −48;
                int n2 = (*it2) −48;
                return n1 < n2;
            }
        }
        return s1 < s2;
    }
};

class Index{
    private:
    std::string  index_file_name;
    std::mutex file_mutex;
    std::mutex tag_mutex;
    std::map<std::string,std::set<std::string,cmp>> files;
    std::map<std::string,std::set<std::string,cmp>> tags;

    bool numeric_string_compare(const std::string& s1, const std::string& s2);
    //recibe un mapa un caracter de typo y escribe en el archivo
    //de indice  los datos del mapa con el formato correspondiente
    void write_index_file(std::map<std::string,//
    std::set<std::string,cmp>> & map, //
    const std::string & type);

public:
    //abre el archivo index_file_name y crea un mapa de files
    // y otro de tags, en los cuales la key es el nombre y
    // el value es un set de hashes
    explicit Index(const std::string & index_file_name);


    //devuelve true si el hash existe dentro de algun set de hashes
    //de alguno de los archivos
```

```
    bool does_hash_exist(const std::string & hash);

    //devuelve el nombre del archivo que tiene ese hash
    //dentro de su set de hashes
    std::string get_file_name(const std::string & hash);

    //agrega el nuevo hash para ese archivo
    //en el mapa de archivos
    //(si el file name no existia en el mapa lo agtrga)
    void add_file_hash(const std::string & file_name, const std::string & hash);

    //devuelve un set de archivos asociados a ese tag
    //si no existe devolvera un ser vacio
    std::set<std::string,cmp> get_tag_files(const std::string & tag_name);


    //agrega el nuevo hash para ese tag
    //en el mapa de tags
    //(si el tag no existia en el mapa lo agtrga)
    void add_tag_hash(const std::string & tag_name, const std::string & hash);


    //Sobre escribe el archivo llamado index_file_name
    //con el mapa de files y el mapa de tags.
    //la version inicial se pierde.
    void close();
};

#endif

```

```
#include "serverIndex.h"
#include <map>
#include <string>
#include <set>


Index::Index(const std::string & index_file_name){
    this→index_file_name = index_file_name;
    std::ifstream index_file(this→index_file_name);
    if (¬index_file.is_open()){
        throw Error("Unable to open index file %s\n",//
            this→index_file_name.c_str());
    }
    std::string line;
    std::string type;
    std::string name;
    std::string hash;
    while (std::getline(index_file,line,’;’)){
        line.erase(std::remove(line.begin(), line.end(), ’\n’), line.end());
      std::istringstream linereader(line, std::ios::binary);
      std::map<std::string,std::set<std::string,cmp>> * map;
      if (¬std::getline(linereader, type, ’ ’)){
        break;
      }
        if (type.compare(FILE_TYPE) ≡ 0){
            map = &files;
        } else if (type.compare(TAG_TYPE) ≡ 0){
            map = &tags;
        } else{
            throw Error("reading input file: "//
              "%s is not valid type\n"  //
              "shuld be %s for files and %s for tags\n",//
              type.c_str(), FILE_TYPE,TAG_TYPE);
        }

        std::getline(linereader, name, ’ ’);
        std::set<std::string,cmp> hashes;

        while (std::getline(linereader, hash, ’ ’)){
            hashes.insert(hash);
      }

      (*map)[name] = hashes;
    }
}

void Index::add_file_hash(const std::string & file_name, //
                                const std::string & hash){
    Lock l(file_mutex);
    std::set<std::string,cmp> hashes;

    std::map<std::string, std::set<std::string,cmp> >::iterator //
     it = files.find(file_name);
    if (it ≠ files.end()){
         hashes = it→second;
    }
    hashes.insert(hash);

    files[file_name] = hashes;
}

bool Index::does_hash_exist(const std::string &  hash){
    Lock l(file_mutex);
    for (std::map<std::string,std::set<std::string,cmp>>::iterator//
    map_iter=files.begin(); map_iter≠files.end(); ++map_iter){
        if (map_iter→second.find(hash) ≠ map_iter→second.end()){
```

```
            return true;
        }
    }
    return false;
}


std::set<std::string,cmp> Index::get_tag_files(const std::string & tag_name){
    Lock l(tag_mutex);
    std::map<std::string, std::set<std::string,cmp> >::iterator //
     it = tags.find(tag_name);
    if (it ≡ tags.end()){
        std::set<std::string,cmp> empty_set;
        return empty_set;
    }
    return it→second;
}

std::string Index::get_file_name(const std::string & hash){
    Lock l(file_mutex);
    for (std::map<std::string,std::set<std::string,cmp>>::iterator//
    map_iter=files.begin(); map_iter≠files.end(); ++map_iter){
        if (map_iter→second.find(hash) ≠ map_iter→second.end()){
            return map_iter→first;
        }
    }
    throw Error("invalid hash");
}

void Index::add_tag_hash(const std::string & tag_name,//
                             const std::string & hash){
    Lock l(tag_mutex);
    std::set<std::string,cmp> hashes;

    std::map<std::string, std::set<std::string,cmp>>::iterator //
     it = tags.find(tag_name);
    if (it ≠ tags.end()){
         hashes = it→second;
    }
    hashes.insert(hash);

    tags[tag_name] = hashes;
}



void Index::write_index_file(std::map<std::string,//
    std::set<std::string,cmp>> & map, //
    const std::string & type){
    std::ofstream index_file(this→index_file_name,ios::out | ios::app);

    if (¬index_file.is_open()){
        throw Error("Unable to open file %s\n",this→index_file_name);
    }
    for (std::map<std::string,std::set<std::string,cmp>>::iterator//
    map_iter=map.begin(); map_iter≠map.end(); ++map_iter){
        index_file  << type <<" "<<(*map_iter).first;

        for (std::set<std::string>::iterator//
            set_iter=(*map_iter).second.begin();//
            set_iter≠(*map_iter).second.end();//
            ++set_iter){
            index_file  <<  " " << (*set_iter);
        }
        index_file  << ";\n";
    }
    index_file.close();
}


void Index::close(){
    std::ofstream index(this→index_file_name,ios::out | ios::trunc);
    index.close();

    write_index_file(files, "f");
    write_index_file(tags, "t");
}
```

```
#ifndef __SERVER_H__
#define __SERVER_H__

#include "serverIndex.h"
#include "commonError.h"
#include "commonProtocol.h"
#include <string>


using std::cout;
using std::endl;
using std::string;

class Connection: Protocol{
  Index & index;
  void push();
  void tag();
  void pull();
public:
  Connection(Socket peer, Index & index);
  void run();
  void stop();
};

#endif
```

```cpp
#include "serverConnection.h"
#include <set>
#include <string>


Connection::Connection(Socket peer, Index & index)://
Protocol(std::move(peer)), index(index){}


void Connection::run(){
  int code = receive_code();
  switch(code){
    case 1:
      push();
      break;
    case 2:
      tag();
      break;
    case 3:
      pull();
      break;
    throw Error("codigo %i no valido", code);
  }
}

void Connection::push(){
  string file_name = receive_string();
  string hash = receive_string();
  if((this→index).does_hash_exist(hash)){
    send_code(NOT_OK);
  }else {
    send_code(OK);
    if(receive_file(hash)){
      this→index.add_file_hash(file_name, hash);
    }
  }
}

void Connection::tag(){
  ssize_t hashes_size = receive_size_first();
  string tag_name = receive_string();
  bool already_exist = (this→index).get_tag_files(tag_name).size() > 0;
  bool invalid_hash = false;
  for (int i = 0; i < hashes_size; i++){
    string hash = receive_string();
    if (¬(this→index).does_hash_exist(hash)){
      invalid_hash = true;
    } else {
      (this→index).add_tag_hash(tag_name, hash);
    }
  }
  if (already_exist ∨ hashes_size ≤ 0 ∨ invalid_hash){
    send_code(NOT_OK);
    return;
  }
  send_code(OK);
}

void Connection::pull(){
  string tag_name = receive_string();
  std::set<std::string,cmp> tag_files = (this→index).get_tag_files(tag_name);
  int size_of_files = tag_files.size();
  if(size_of_files ≡ 0){
    send_code(NOT_OK);
  }else {
    send_code(OK);
```

```cpp
    send_size_first(size_of_files);

    for (std::set<std::string>::iterator//
            set_iter=tag_files.begin();//
            set_iter≠tag_files.end();//
            ++set_iter){
            string hash = (*set_iter);
          string name = this→index.get_file_name(hash);
          send_string(name+"."+tag_name);
          send_file(hash);
        }
  }
}

void Connection::stop(){
  (this→socket).stop();
}
```

```

#define PORT 1
#define INDEX_FILE 2

#include "serverListener.h"
#include <thread>


int main(int argc, char * argv[]){
  if(argc < 2){
    throw Error("Parametros incorrectos");
  }

  Server_listener s(argv[PORT],argv[INDEX_FILE]);
  s.start_listening();

  char c = getchar();
  while(c ≠ 'q'){
    c = getchar();
  }

  s.stop_listening();

  return 0;
}
```

```
#ifndef THREAD_H_
#define THREAD_H_

#include <thread>

class Thread {
    private:
        std::thread thread;

    public:
        Thread() {}

        void start();

        void join();

        virtual void run() = 0;
        virtual ~Thread() {}

        Thread(const Thread&) = delete;
        Thread& operator=(const Thread&) = delete;

        Thread(Thread∧ other) {
            this→thread = std::move(other.thread);
        }

        Thread& operator=(Thread∧ other) {
            this→thread = std::move(other.thread);
            return *this;
        }
};
#endif
```

```
#ifndef __SOCKET_H__
#define __SOCKET_H__

#include <iostream>
#include <stdbool.h>
#include <stddef.h>
#include <sys/types.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>
#include <unistd.h>
#include <fcntl.h>
#include "commonError.h"


#define SUCCESS 0
#define ERROR 1

class Socket {
  private:
    int socket_num;
    const char * host_name;
    const char * port;
    bool is_connected;

  //data una estructura del tipo addrinfo, la recorre hasta
  //poder crear un socket y setea entonces el numero del socket
  struct addrinfo * define_socket_num(struct addrinfo * node);

  //crea un socket inicial
  void init();

  //devuelve el primer nodo de addrinfo
  struct addrinfo * addrinfo();

  //valida que el puerto sea valido
  bool is_valid_port(const char * port);



public:
  //Constructor por copia anulado
  Socket(const Socket& other) = delete;
  //Asignacion por copia anulada
  Socket& operator=(const Socket &other) = delete;
  //Constructor por movimiento anulado
  Socket(Socket∧ other);
  //Asignacion por movimiento anulada
  Socket& operator=(Socket∧ other);

  //constructor del socket
  Socket(const char * host_name, const char * port);

  //constructor del peer socket ya conectado
  explicit Socket(int socket_num);

  //trata de conectar al socket
  void connection();

  //sirve para servidores,
  //el socket se quedara esperando que se conecten con el
  void bind_and_listen();

  //sirve para servidores,
  //acepta a otro socket que quiera conectarse con el
  Socket accept_socket();
```

```

  //Debe estar conectado con el socket cuyo numero sea skt_num
  //porque a partir de Ã©l recibirÃ¡ datos
  //TratarÃ¡ de recibir tantos  datos como se especifique
  //en el parametro size.
  //guarda los datos recibidos en buffer que debe ser un array
  //de un tamaÃ±o mayor  o igual a size
  int receive_message(char* buffer, size_t size);



  //TratarÃ¡ de enviar tantos datos como se especifique
  //en el parÃ¡metro size.
  //enciarÃ¡ los datos en buffer que debe ser un array
  //de un tamaÃ±o mayor o igual a size
  int send_message(const char* buffer, size_t size);

  void stop();

  //destructor del socket
  ~Socket();
};

#endif
```

```
#include "commonSocket.h"

struct addrinfo * Socket::define_socket_num(struct addrinfo * node){
    while (node ≠ NULL) {
      socket_num = //
        socket(node→ai_family, //
        node→ai_socktype, //
        node→ai_protocol);
      if (socket_num ≠ −1) {
          this→socket_num = socket_num;
         return node;
      }
      node = node→ai_next;
    }
    return NULL;
}



void Socket::init(){
    struct addrinfo *addrinfoNode = addrinfo();
    if(define_socket_num(addrinfoNode)≡NULL){
      throw Error("Error en init\n");
    }
    free(addrinfoNode);
}

struct addrinfo * Socket::addrinfo(){
    struct addrinfo hints;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       //IPv4
    hints.ai_socktype = SOCK_STREAM; //TCP
    hints.ai_flags = 0;

    struct addrinfo *addrinfoNode;


    int addrinfo_returnvalue = //
    getaddrinfo(this→host_name, this→port, &hints, &addrinfoNode);

    if (addrinfo_returnvalue ≠ SUCCESS){
      gai_strerror(addrinfo_returnvalue);
    }
    return addrinfoNode;
}

bool Socket::is_valid_port(const char * port){
  for (unsigned int i = 0; i < strlen(port); i ++){
    if (¬isdigit(port[i])){
      return false;
    }
  }
  return true;
}




Socket::Socket(const char * host_name, const char * port){
    if(¬is_valid_port(port)){
      throw Error("%s no es un purto valido\n"//
       "deben ser todos caracteres numÃ©ricos", port);
    }

    this→host_name = host_name;
```

```
    this→port = port;
    this→is_connected = false;

    init();
    int val = 1;
    if (setsockopt(this→socket_num, SOL_SOCKET, SO_REUSEADDR, &val, //
        sizeof(val)) ≡ −1) {
        close(this→socket_num);
        throw Error("Error in reuse socket: %s\n", strerror(errno));
    }
}

Socket::Socket(int socket_num){
  this→is_connected = true;
  this→socket_num = socket_num;
}



Socket::Socket(Socket∧ other){
  this→socket_num = other.socket_num;
  this→host_name = other.host_name;
  this→port = other.port;
  this→is_connected = other.is_connected;

  other.socket_num =−1;
  other.host_name ="null";
  other.port ="null";
  other.is_connected = false;
}

Socket& Socket::operator=(Socket∧ other){
  if (this ≡ &other) {
    return *this; // other is myself!
  }
  this→socket_num = other.socket_num;
  this→host_name = other.host_name;
  this→port = other.port;
  this→is_connected = other.is_connected;

  other.socket_num =−1;
  other.host_name ="null";
  other.port ="null";
  other.is_connected = false;
  return *this;
}


void Socket::connection(){
  struct addrinfo *addrinfoNode = addrinfo();

  struct addrinfo *node  = define_socket_num(addrinfoNode);

  while (node ≠ NULL ∧ this→is_connected ≡ false) {
      int connection_returnvalue = connect(this→socket_num, //
        node→ai_addr, node→ai_addrlen);
      this→is_connected = (connection_returnvalue ≠ −1);
      if (this→is_connected){
          break;
      }
      node = define_socket_num(addrinfoNode);
  }
  freeaddrinfo(addrinfoNode);
  if (this→is_connected ≡ false) {
      close(this→socket_num);
      throw Error("Error in socket connection: %s\n", strerror(errno));
  }
}

void Socket::bind_and_listen(){
    struct addrinfo *addrinfoNode = addrinfo();

    int bindReturnValue = −1;
    while (addrinfoNode ≠ NULL) {
      bindReturnValue = bind(this→socket_num, //
      addrinfoNode→ai_addr, //
      addrinfoNode→ai_addrlen);
      if (bindReturnValue ≠ −1) {
          break;
      }
      addrinfoNode = addrinfoNode→ai_next;
    }
    free(addrinfoNode);

    if (bindReturnValue ≡ −1) {
      throw Error("Error in bind: %s\n", strerror(errno));
    }
    int listenReturnValue = listen(this→socket_num, 20);
    if (listenReturnValue ≡ −1) {
      throw Error("Error in listen: %s\n", strerror(errno));
    }
    this→ is_connected = true;
}

Socket Socket::accept_socket(){
    int new_sockfd = accept(this→socket_num, NULL, NULL);
    if (new_sockfd ≡ −1){
      throw Error("Error in accept");
    }
    return std::move(Socket(new_sockfd));
}

int Socket::receive_message(char* buffer, const size_t size){
    int total_received = 0;
    int bytes_recived = 0;


    while ((bytes_recived = recv(this→socket_num, //
    &buffer[total_received],//
    size − total_received, MSG_NOSIGNAL)) >0) {
      total_received += bytes_recived;
      if (size −total_received≡ 0){
        break;
      }
    }
    if (bytes_recived < 0) {
      throw Error(" Error recibing info: %s\n", strerror(errno));
    }
    return total_received;
}


int Socket::send_message(const char* buffer,const size_t size){
    int total_sent = 0;
    int bytes_sent = 0;

    while ((bytes_sent = send(this→socket_num, //
      &buffer[total_sent], size−total_sent, MSG_NOSIGNAL)) >0){
        total_sent += bytes_sent;
    }
    if (bytes_sent < 0) {
      throw Error(" Error sending info: %s\n", strerror(errno));
```

```
    }
    return total_sent;
}

void Socket::stop(){
  shutdown(this→socket_num, SHUT_RDWR);
  close(this→socket_num);
}

Socket::~Socket(){
  if (this→socket_num > 0){
    shutdown(this→socket_num, SHUT_RDWR);
    close(this→socket_num);
  }
}
```

```
#ifndef __PROTOCOL_H__
#define __PROTOCOL_H__


#include "commonSocket.h"


#include <string>
#include <stdio.h>
#include <iostream>
#include <fstream>


using std::cout;
using std::endl;
using std::string;


#define CHUNK_LEN 64


#define PROTOCOL_MSG_SIZE 4
#define PROTOCOL_CODE 2
#define NO_PEER_SKT -1

#define PUSH_CODE 1
#define TAG_CODE 2
#define PULL_CODE 3

#define OK 1
#define NOT_OK 0



class Protocol{
private:
int get_digits(unsigned int num){
    int digits = 1;
    while ( num > 0 ) {
        num /= 10;
        digits++;
    }
    return digits;
}



protected:
  Socket socket;
  explicit Protocol(Socket socket);


  //envia en un solo byte un numero
  void send_code(int code_to_send);

  //envia un string
  //siempre antes enviando el tamaÃ±o
  void send_string(const string & string_to_send);

  //envia un archivo de a chunks
  //siempre antes enviando el tamaÃ±o total
  void send_file(const std::string & file_name);

  //envia 4 bytes con el tamaÃ±o
  void send_size_first(unsigned int size);

```

```

  //recibe un solo byte con un numero
  int receive_code();

  //recibe primero la longitud y luego un string
  std::string receive_string();

  //recibe primero la longitud total y
  //luego el archivo de a chunks
  bool receive_file(const std::string & hash);


  //recibe el tamaÃ±o en 4 bytes
  ssize_t receive_size_first();
};

#endif

```

```cpp
#include "commonProtocol.h"
#include <string>

Protocol::Protocol(Socket socket):socket(std::move(socket)) {}

void Protocol::send_code(int code_to_send){
  char code[PROTOCOL_CODE];
  snprintf(code,PROTOCOL_CODE, "%d", code_to_send);
  (this→socket).send_message(code,PROTOCOL_CODE);
}


void Protocol::send_string(const string & string_to_send){
  int size = string_to_send.size();
  send_size_first(size);
  (this→socket).send_message(string_to_send.c_str(),size);
}


void Protocol::send_file(const std::string & file_name){
    std::ifstream file(file_name,std::ifstream::binary);
    if (file.fail()){
      send_code(NOT_OK);
      throw Error("No se encontro el archivo %s\n", file_name.c_str());
    }
    send_code(OK);
    file.seekg(0, file.end);
    int file_len = file.tellg();
    file.seekg(0, file.beg);
    send_size_first(file_len);

    int bytes_sent = 0;
    int total_sent = 0;
    char request[CHUNK_LEN+1];

    while (total_sent < file_len ∧  ¬file.eof()){
      memset(request, 0, CHUNK_LEN+1);
      file.read(request, CHUNK_LEN);

      int request_len = CHUNK_LEN;
      if (file_len − total_sent < CHUNK_LEN){
          request_len = file_len − total_sent;
      }
      bytes_sent = (this→socket).send_message(request, request_len);

      if (bytes_sent < 0){
        break;
      }
      total_sent += bytes_sent;
    }
}

void Protocol::send_size_first(unsigned int size){
  int digitos = get_digits(size);
  if(digitos > PROTOCOL_MSG_SIZE){
    throw Error("Mensaje demasiado largo!");
  }
  char msg_size[PROTOCOL_MSG_SIZE];
  memset(msg_size, 0, PROTOCOL_MSG_SIZE);
  snprintf(msg_size,PROTOCOL_MSG_SIZE, "%d", size);
   (this→socket).send_message(msg_size,PROTOCOL_MSG_SIZE);
}


int Protocol::receive_code(){
  char code[PROTOCOL_CODE];
```

```cpp
  (this→socket).receive_message(code, PROTOCOL_CODE);
  return atoi(code);
}



std::string Protocol::receive_string(){
  ssize_t msg_size = receive_size_first();
  char chunk[CHUNK_LEN];
  (this→socket).receive_message(chunk, msg_size);
  chunk[msg_size] = 0;
  std::string string_received(chunk);
  return string_received;
}


bool Protocol::receive_file(const std::string & name){
  if (receive_code() ≡ NOT_OK){
    return false;
  }
  std::ofstream file(name,std::ofstream::out| std::ofstream::binary);
  ssize_t file_len = receive_size_first();
  char chunk[CHUNK_LEN+1];
  int total_received = 0;
  int bytes_received = 0;
  while (total_received < file_len){
        memset(chunk, 0, CHUNK_LEN+1);

        int request_len = CHUNK_LEN;
        if (file_len − total_received < CHUNK_LEN){
            request_len = file_len − total_received;
        }
        bytes_received = (this→socket).receive_message(chunk, request_len);

        total_received +=bytes_received;
        if (bytes_received ≤ 0){
          break;
        }
        file.write(chunk, bytes_received);
    }
  return true;
}

ssize_t Protocol::receive_size_first(){
  char msg_size[PROTOCOL_MSG_SIZE];
  (this→socket).receive_message(msg_size, PROTOCOL_MSG_SIZE);
  return atoi(msg_size);
}
```

may 15, 18 16:13 **commonError.h** Page 1/1

```cpp
#ifndef __ERROR_H__
#define __ERROR_H__


#include <iostream>
#include <cstdio>
#include <cstdarg>

#define BUF_LEN 256

class Error : public std::exception {
private:
  char msg[BUF_LEN];
public:
  explicit Error(const char * format,...) noexcept;
  virtual const char * what() const noexcept;
  virtual ~Error() noexcept;
};

#endif
```

may 15, 18 16:13 **commonError.cpp** Page 1/1

```cpp
#include "commonError.h"

Error::Error(const char* format, ...) noexcept {
  va_list args;
  va_start(args, format);
  int s = vsnprintf(msg, BUF_LEN, format, args);
  msg[s+1] = 0;
  va_end(args);
}

//devuelve el error
const char * Error::what() const noexcept{
  return msg;
}
Error::~Error() noexcept{}

```

```
#ifndef __CLIENT_H__
#define __CLIENT_H__
#include "commonProtocol.h"
#include "error.h"
#include <stack>
#include <string>
#include <stdio.h>
#include <iostream>


using std::cout;
using std::endl;
using std::string;

#define SERVER_NAME 1
#define PORT 2
#define COMAND 3
#define FILE_NAME 4
#define HASH 5
#define TAG_NAME 4
class Client: Protocol{
  void push(const string & file_name, const string & hash);

  void tag(const string & tag_name, std::stack<string> & hashes);

  void pull(const string & tag_name);

public:
  Client(char * host_name, char * port);

  //ejecuta push tag o pull segun corresponda
  void execute_command(int argc, char * argv[]);
};

#endif
```

```
#include "client.h"
#include <string>
#include <stack>

void Client::push(const string & file_name, const string & hash){
  send_code(PUSH_CODE);
  send_string(file_name);
  send_string(hash);
  if(receive_code() ≡ OK){
    try{
      send_file(file_name);
    } catch(Error e){
      cout << "Error: archivo inexistente.\n";
    }
  }
}

void Client::tag(const string & tag_name, std::stack<string> & hashes){
  send_code(TAG_CODE);
  send_size_first(hashes.size());
  send_string(tag_name);
  while (¬hashes.empty()){
      string hash = hashes.top();
      hashes.pop();
      send_string(hash);
    }
    if(receive_code()≠ OK){
      cout << "Error: tag/hash incorrecto.\n";
    }
}

void Client::pull(const string & tag_name){
  send_code(PULL_CODE);
  send_string(tag_name);
  if(receive_code() ≡ NOT_OK){
    cout << "Error: tag/hash incorrecto.\n";
    return;
  }
  int size_of_files = receive_size_first();
  for(int i = 0; i  < size_of_files; i++){
    string file_name = receive_string();
    receive_file(file_name);
  }
}

Client::Client(char * host_name, char * port)://
Protocol(std::move(Socket(host_name, port))){
  this→socket.connection();
}

void Client::execute_command(int argc, char * argv[]){
  const string command = argv[COMAND];

  if (command.compare("pull") ≡ 0){
    const string tag_name = argv[TAG_NAME];
    pull(tag_name);
    return;
  }
  if(argc <6){
    cout << "Error: argumentos invalidos.\n";
    return;
  }

  if (command.compare("push") ≡ 0){
    const string file = argv[FILE_NAME];
    const string hash = argv[HASH];
    push(file, hash);
    return;
  }
  if (command.compare("tag") ≡ 0){
    const string tag_name = argv[TAG_NAME];
    std::stack<string>  hashes;
    for(int i = TAG_NAME+1; i< argc; i++){
      hashes.push(argv[i]);
    }
    tag(tag_name,hashes);
    return;
  }
  cout << "Error: argumentos invalidos.\n";
}

```

```
#include "error.h"
#include "client.h"

#define SERVER_NAME 1
#define PORT 2


int main(int argc, char * argv[]){
  if (argc < 4){
    throw Error("Parametros incorrectos");
  }

  Client cliente(argv[SERVER_NAME],argv[PORT]);
  cliente.execute_command(argc, argv);
  return 0;
}

```

```
Table of Contents
 1 serverThConnection.h sheets   1 to   1 ( 1) pages   1-  1   20 lines
 2 serverThConnection.cpp sheets   1 to   1 ( 1) pages   2-  2   33 lines
 3 serverLock.h........ sheets   2 to   2 ( 1) pages   3-  3   18 lines
 4 serverLock.cpp...... sheets   2 to   2 ( 1) pages   4-  4   12 lines
 5 serverListener.h.... sheets   3 to   3 ( 1) pages   5-  5   24 lines
 6 serverListener.cpp.. sheets   3 to   3 ( 1) pages   6-  6   39 lines
 7 serverIndex.h....... sheets   4 to   4 ( 1) pages   7-  8   97 lines
 8 serverIndex.cpp..... sheets   5 to   6 ( 2) pages   9- 11  144 lines
 9 serverConnection.h.. sheets   6 to   6 ( 1) pages  12- 12   26 lines
10 serverConnection.cpp sheets   7 to   7 ( 1) pages  13- 14   87 lines
11 serverApp.cpp....... sheets   8 to   8 ( 1) pages  15- 15   26 lines
12 commonThread.h...... sheets   8 to   8 ( 1) pages  16- 16   33 lines
13 commonSocket.h...... sheets   9 to   9 ( 1) pages  17- 18   91 lines
14 commonSocket.cpp.... sheets  10 to  11 ( 2) pages  19- 22  215 lines
15 commonProtocol.h.... sheets  12 to  12 ( 1) pages  23- 24   85 lines
16 commonProtocol.cpp.. sheets  13 to  13 ( 1) pages  25- 26  118 lines
17 commonError.h....... sheets  14 to  14 ( 1) pages  27- 27   21 lines
18 commonError.cpp..... sheets  14 to  14 ( 1) pages  28- 28   17 lines
19 client.h............ sheets  15 to  15 ( 1) pages  29- 29   36 lines
20 client.cpp.......... sheets  15 to  16 ( 2) pages  30- 31   83 lines
21 clientApp.cpp....... sheets  16 to  16 ( 1) pages  32- 32   18 lines
```