Ciencia de la Computación

Redes y Comunicación

Docente Julio Santisteban

Informe: TicTacToe

Entregado el 23/04/2024

Ramirez Arredondo, Fernando

Semestre VII

2024-1

"El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo"

_____

Informe

Descripcion del juego:

TicTacToe es un juego de estrategia para dos jugadores, X y O, que se juega en una cuadrícula de 3x3. El objetivo del juego es ser el primer jugador en colocar tres de sus fichas (X o O) en línea horizontal, vertical o diagonal.
El juego se desarrolla de la siguiente manera:

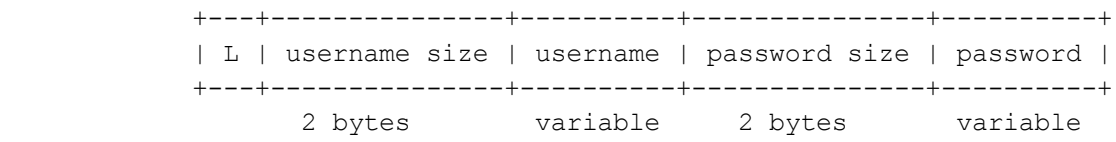
1. Los jugadores alternan turnos, el servidor determina que ficha inicia.
2. En su turno, cada jugador coloca su ficha en una casilla vacía de la cuadrícula.
3. El juego continúa hasta que uno de los jugadores consigue alinear tres de sus fichas en línea horizontal, vertical o diagonal, o hasta que la cuadrícula se llena sin que ningún jugador consiga ganar (empate).
4. El perdedor es quien empieza la siguiente partida, en caso de empate, no inicia el mismo.

El código implementa esta lógica, permitiendo a los jugadores realizar movimientos válidos, verificando si hay un ganador o si el juego ha terminado en empate, y reiniciando el juego para una nueva partida. Cualquier usuario puede participar del juego, las actualizaciones de la partida se informan a todos los usuarios mediante un mensaje broadcast.

Estructuras de datos:

OK

LOGIN client -> server

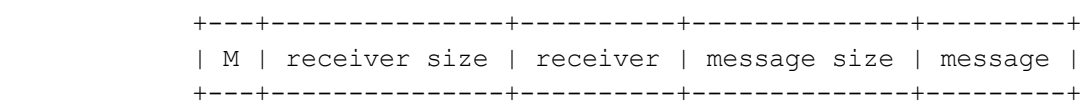


LOGOUT client -> server

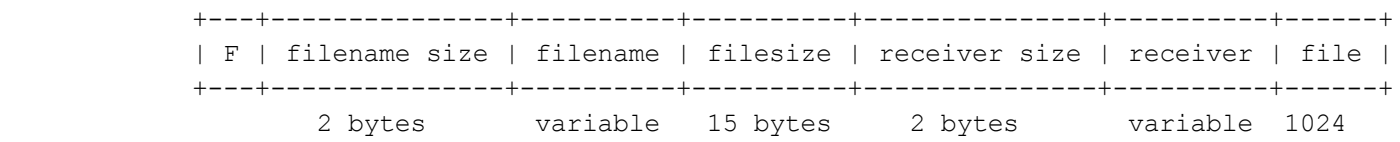
LIST client -> server

BROADCAST client -> server

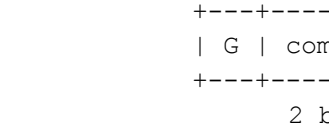
PRIVATE MESSAGE client -> server



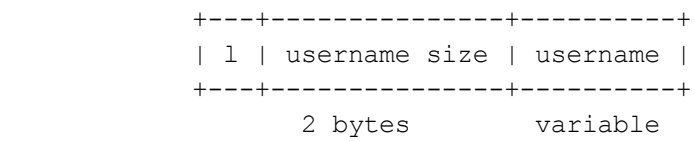
FILE client -> server



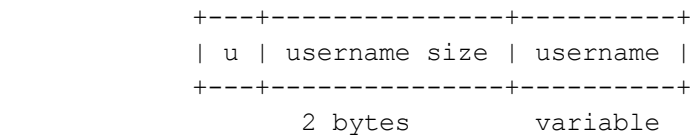
GAME client -> server



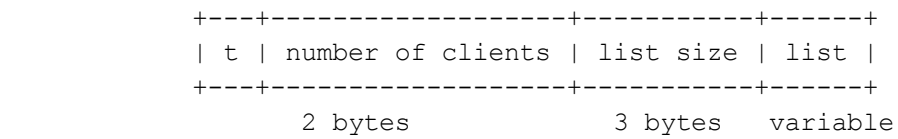
LOGIN server -> client



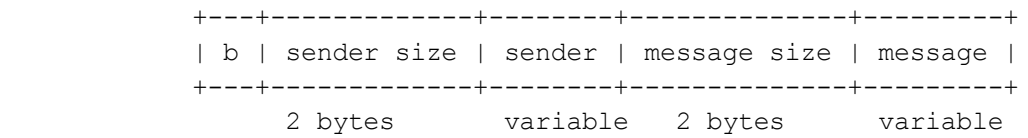
LOGOUT server -> client



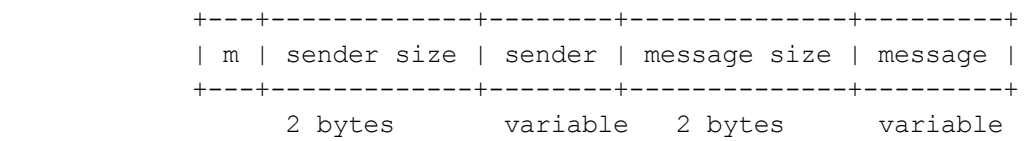
LIST server -> client



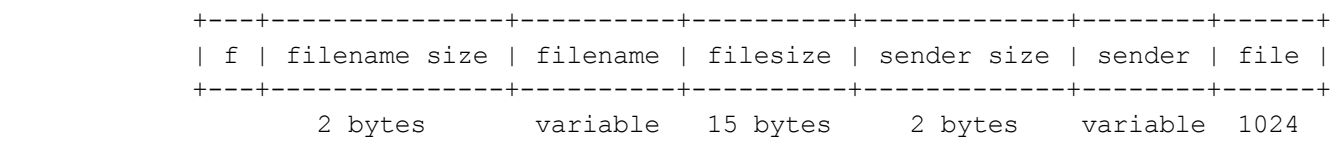
BROADCAST server -> client



PRIVATE MESSAGE server -> client



FILE server -> client



ERROR

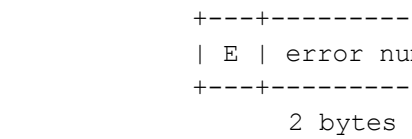


Diagrama de secuencia:

Diagrama de secuencia de TicTacToe.

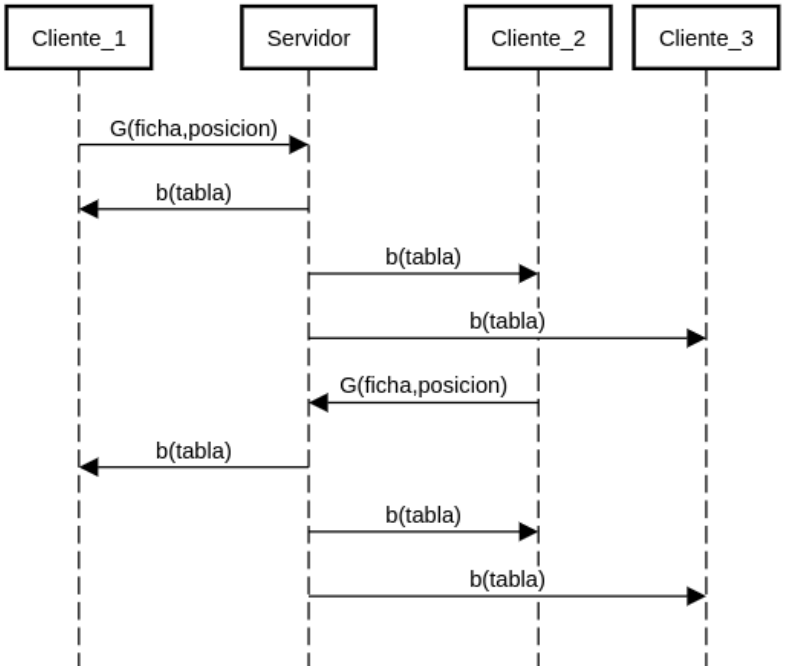
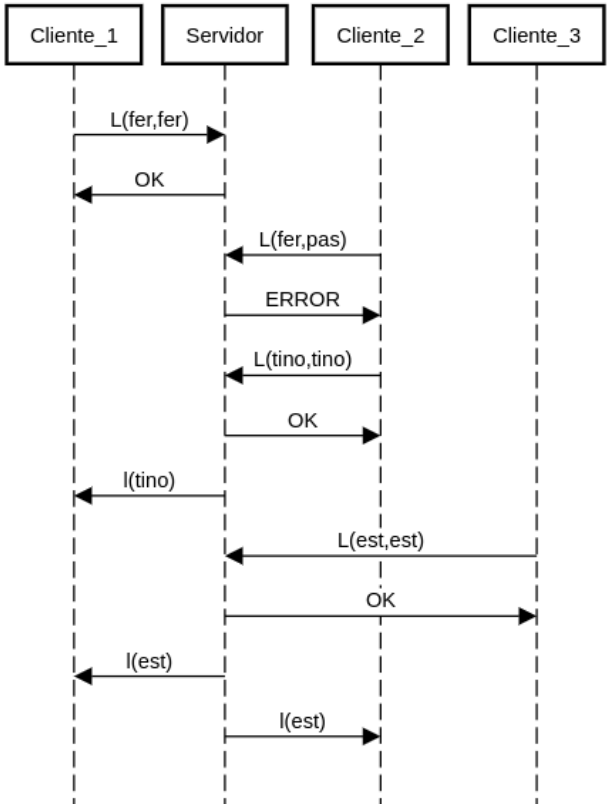


Diagrama de secuencia de login.



Codigo:

Server: net_utilities.hpp

```
#ifndef NET_UTILITIES_HPP
#define NET_UTILITIES_HPP

namespace net {
   std::string format_size(const int, int);
   class PROTOCOL {
       public:
       std::string ErrorMessage(int);
       std::string OkMessage();
       std::string WelcomeMessage(std::string);
       std::string GoodbyeMessage(std::string);
       std::string ListMessage(const std::map<std::string, int>&);
       std::string BroadcastMessage(std::string, std::string);
       std::string PrivateMessage(std::string, std::string);
       std::string FileMessage(std::string, ssize_t, std::string,
std::string);
   };
}

#endif // NET_UTILITIES_HPP
```

Server: net_utilities.cpp

```
#include "../include/net_utilities.hpp"

std::string net::format_size(const int size, int n) {
   std::ostringstream oss;
   oss << std::setw(n) << std::setfill('0') << size;
   return oss.str();
}
// E##
std::string net::PROTOCOL::ErrorMessage(int err_n) {
   if (err_n < 0 || err_n > 99) {
       return "E00";
   }
   std::ostringstream oss;
   oss << "E" << std::setw(2) << std::setfill('0') << err_n;
```

```
    return oss.str();
}
// O
std::string net::PROTOCOL::OkMessage(){
    std::ostringstream oss;
    oss << "O";
    return oss.str();
}
// l##USERNAME
std::string net::PROTOCOL::WelcomeMessage(std::string username){
    std::ostringstream oss;
    oss << "l" << net::format_size(username.size(), 2) << username;
    return oss.str();
}
// u##USERNAME
std::string net::PROTOCOL::GoodbyeMessage(std::string username){
    std::ostringstream oss;
    oss << "u" << net::format_size(username.size(), 2) << username;
    return oss.str();
}
// t## ###LIST
std::string net::PROTOCOL::ListMessage(const std::map<std::string,
int>& myMap){
    std::ostringstream oss;
    auto it = myMap.begin();
    if (it != myMap.end()) {
        oss << it->first;
        ++it;
    }
    for (; it != myMap.end(); ++it) {
        oss << "," << it->first;
    }
    std::string map_size = net::format_size(myMap.size(), 2);
    std::string list_size = net::format_size(oss.str().size(), 3);
    std::string list = oss.str();
    oss.str("");
    oss << "t" << map_size << list_size << list;
    return oss.str();
}
// b##SENDER##MSG
std::string net::PROTOCOL::BroadcastMessage(std::string msg,
std::string sender){
    std::ostringstream oss;
    oss << "b" << net::format_size(sender.size(), 2) << sender <<
net::format_size(msg.size(), 2) << msg;
```

```
    return oss.str();
}
// m##SENDER##MSG
std::string net::PROTOCOL::PrivateMessage(std::string msg,
std::string sender){
    std::ostringstream oss;
    oss << "m" << net::format_size(sender.size(), 2) << sender <<
net::format_size(msg.size(), 2) << msg;
    return oss.str();
}
// f##FILENAME## ##SENDER FILE
std::string net::PROTOCOL::FileMessage(std::string file_name, ssize_t
file_size, std::string sender, std::string data){
    std::ostringstream oss;
    oss << "f" << net::format_size(file_name.size(), 2) << file_name
<< net::format_size(file_size, 15) << net::format_size(sender.size(),
2) << sender << data;
    return oss.str();
}
```

Server: tictactoe.hpp

```
#ifndef TICTACTOE_HPP
#define TICTACTOE_HPP

class TicTacToe {
private:
    char board[3][3];
    char currentPlayer;

public:
    TicTacToe(char startingPlayer);

    std::string makeMove(const std::string& move);
    std::string getBoardAsString() const;
    bool checkWin() const;
    bool isBoardFull() const;
    void reset(char startingPlayer);
    char getCurrentPlayer() const;
};

#endif
```

Server: tictactoe.cpp

```
#include "../include/tictactoe.hpp"

TicTacToe::TicTacToe(char startingPlayer) :
currentPlayer(startingPlayer){ reset(startingPlayer); }

std::string TicTacToe::makeMove(const std::string& move) {
   std::string output = "\n";
   if (move.size() != 2) {
       output += move;
       output += "Invalid move format. Please use format 'PM', where
P is player (X or O) and M is move (1-9).";
       return output;
   }

   char player = move[0];
   char cell = move[1];

   if (player != currentPlayer) {
       output += "It's not ";
       output += player;
       output += "'s turn.";
       output += currentPlayer;
       output += ".";
       return output;
   }

   if (cell < '1' || cell > '9') {
       output += "Invalid move. Move must be a number between 1 and
9.";
       return output;
   }

   int row = (cell - '1') / 3;
   int col = (cell - '1') % 3;

   if (board[row][col] != ' ') {
       output += "Invalid move. Cell already occupied.";
       return output;
   }

   board[row][col] = player;
   if (checkWin()) {
```

```
        output += "Player ";
        output += player;
        output += " wins!\n";
        output += getBoardAsString();
        reset(currentPlayer);
    } else if (isBoardFull()) {
        output += "It's a draw!\n";
        output += getBoardAsString();
        reset(currentPlayer);
    } else {
        output += getBoardAsString();
    }
    currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
    return output;
}

std::string TicTacToe::getBoardAsString() const {
    std::string boardStr;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            boardStr += board[i][j];
            if (j < 2) boardStr += "|";
        }
        if (i < 2) boardStr += "\n-+-+-\n";
    }
    return boardStr;
}

bool TicTacToe::checkWin() const {
    for (int i = 0; i < 3; ++i) {
        if (board[i][0] != ' ' && board[i][0] == board[i][1] &&
board[i][0] == board[i][2]) return true; // Row
        if (board[0][i] != ' ' && board[0][i] == board[1][i] &&
board[0][i] == board[2][i]) return true; // Column
    }
    if (board[0][0] != ' ' && board[0][0] == board[1][1] &&
board[0][0] == board[2][2]) return true; // Diagonal 1
    if (board[0][2] != ' ' && board[0][2] == board[1][1] &&
board[0][2] == board[2][0]) return true; // Diagonal 2
    return false;
}

bool TicTacToe::isBoardFull() const {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
```

```
            if (board[i][j] == ' ') return false;
        }
    }
    return true;
}

void TicTacToe::reset(char startingPlayer) {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            board[i][j] = ' ';
        }
    }
    currentPlayer = startingPlayer;
}

char TicTacToe::getCurrentPlayer() const {
    return currentPlayer;
}
```

Server: server.hpp

```
#ifndef SERVER_HPP
#define SERVER_HPP

#include "tictactoe.hpp"
#include "net_utilities.hpp"

class SERVER {
public:
    SERVER(const char *);
    ~SERVER();

private:
    std::map<std::string, int> CLIENTS;
    int sockFD;
    net::PROTOCOL protocol;
    TicTacToe game;

    void start_();
    int accept_();
    void session_(const int);
    void read_write_(const void *);
```

```
    enum MessageType {
        LOGIN = 'L',
        OK = 'O',
        ERROR = 'E',
        LOGOUT = 'U',
        LIST = 'T',
        BROADCAST = 'B',
        PRIVATE_MESSAGE = 'M',
        FILE_TRANSFER = 'F',
        GAME = 'G'
    };

    std::string recv_data(const int, const int);
    void send_data(const int, const std::string);

    using handler_function = std::function<void(int, std::string&)>;
    std::unordered_map<char,handler_function> handle_map;
    void init_handle_map();
    void add_handler(const char, const handler_function);

    void handle_login(const int, std::string&);
    void handle_ok();
    void handle_error(const int, const std::string&);
    void handle_logout(const std::string&);
    void handle_list(const int);
    void handle_broadcast(const int, const std::string&);
    void handle_private_message(const int, const std::string&);
    void handle_file_transfer(const int, const std::string&);
    void handle_tictactoe(const int, const std::string&);
};

#endif // SERVER_HPP
```

Server: server.cpp

```
#include "../include/server.hpp"
#include "../include/net_utilities.hpp"
#include "../include/tictactoe.hpp"

#define BACKLOG 10
const int ERR_LOGIN_DUPLICATES = 1;
const int ERR_LOGIN = 2;
```

```
const int ERR_PRIV = 11;
const int ERR_PRIV_RECV = 12;
const int ERR_FILE = 21;
const int ERR_FILE_RECV = 22;
const int STANDARD_MESSAGE_SIZE = 2;
const int STANDARD_FILE_SIZE = 15;

void sigchld_handler(int s) {
   int saved_errno = errno;
   while (waitpid(-1, NULL, WNOHANG) > 0);
   errno = saved_errno;
}

SERVER::SERVER(const char *port) : game('O') {
   addrinfo hints, *servinfo, *p;
   struct sigaction sa;
   int yes = 1, rv;

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

   if ((rv = getaddrinfo(NULL, port, &hints, &servinfo)) != 0){
       std::cerr << "getaddrinfo: " << gai_strerror(rv) << std::endl;
       exit(1);
   }
   for (p = servinfo; p != NULL; p = p->ai_next){
       if ((sockFD = socket(p->ai_family, p->ai_socktype,
p->ai_protocol)) ==  -1){
           perror("server: socket");
           continue;
       }
       if (setsockopt(sockFD, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)) == -1){
           perror("setsockopt");
           exit(1);
       }
       if (bind(sockFD, p->ai_addr, p->ai_addrlen) == -1){
           close(sockFD);
           perror("server: bind");
           continue;
       }
       break;
   }
```

```
    freeaddrinfo(servinfo);
    if (p == NULL){
        std::cerr << "server: failed to bind" << std::endl;
        exit(1);
    }
    if (listen(sockFD, BACKLOG) == -1){
        perror("listen");
        exit(1);
    }
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1){
        perror("sigaction");
        exit(1);
    }
    init_handle_map();
    std::cout << "server: waiting for connections...        PORT = "
<< port << std::endl;
    start_();
}

SERVER::~SERVER() {
    shutdown(sockFD, SHUT_RDWR);
}

void SERVER::start_() {
    while(1){
        int accepted_connection = accept_();
        session_(accepted_connection);
    }
}

int SERVER::accept_() {
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof client_addr;
    int client_sockFD = accept(sockFD, (sockaddr *)&client_addr,
&client_addr_size);
    if (client_sockFD == -1){
        perror("accept");
    }
    return client_sockFD;
}

void SERVER::session_(const int _session_socket){
```

```
    std::thread worker_thread([this,
_session_socket](){read_write_(reinterpret_cast<void
*>(_session_socket));});
    worker_thread.detach();
}

void SERVER::read_write_(const void *_void_socket) {
    int session_socket = (intptr_t)_void_socket;
    std::string session_username;
    char type_buffer[1];

    while (true) {
        type_buffer[0] = '\n';
        if (recv(session_socket, type_buffer, 1, 0) == -1) {
            perror("recv");
        }
        char message_type = type_buffer[0];
        auto it = handle_map.find(message_type);
        if (it != handle_map.end()) {
            std::cout << "-> " << message_type;
            it->second(session_socket, session_username);
        } else { } // unknown message type
    }
}

std::string SERVER::recv_data(const int _socket, const int
_size_size) {
    std::unique_ptr<char[]> size_buffer(new char[_size_size]);
    if (recv(_socket, size_buffer.get(), _size_size, 0) == -1) {
        perror("recv");
    }
    std::cout << std::string(size_buffer.get(), _size_size);
    int data_size = atoi(size_buffer.get());

    std::unique_ptr<char[]> buffer(new char[data_size]);
    ssize_t numbytes = recv(_socket, buffer.get(), data_size, 0);
    if (numbytes == -1) {
        perror("recv");
    }
    std::cout << std::string(buffer.get(), data_size);
    return std::string(buffer.get(), numbytes);
}

void SERVER::send_data(const int _socket, const std::string _message)
{
```

```
        if (send(_socket, _message.c_str(), _message.size(), 0) == -1)
            perror("send");
        else
            std::cout << "<- " << _message << std::endl;
    }

void SERVER::init_handle_map() {
    add_handler(LOGIN, [this](int _sock, std::string& _username) {
handle_login(_sock, _username); });
    add_handler(OK, [this](int _sock, std::string& _username) {
handle_ok(); });
    add_handler(ERROR, [this](int _sock, std::string& _username) {
handle_error(_sock, _username); });
    add_handler(LOGOUT, [this](int _sock, std::string& _username) {
handle_logout(_username); });
    add_handler(LIST, [this](int _sock, std::string& _username) {
handle_list(_sock); });
    add_handler(BROADCAST, [this](int _sock, std::string& _username) {
handle_broadcast(_sock, _username); });
    add_handler(PRIVATE_MESSAGE, [this](int _sock, std::string&
_username) { handle_private_message(_sock, _username); });
    add_handler(FILE_TRANSFER, [this](int _sock, std::string&
_username) { handle_file_transfer(_sock, _username); });
    add_handler(GAME, [this] (int _sock, std::string & _username) {
handle_tictactoe(_sock, _username); });
}

void SERVER::add_handler(const char _message_type, const
handler_function _handler) {
    auto result = handle_map.emplace(_message_type, _handler);
    if (!result.second) { std::cerr << "Error: Unable to add handler
to the map" << std::endl; }
}

void SERVER::handle_login(const int _socket, std::string&
_this_username) {
    std::string username = recv_data(_socket, STANDARD_MESSAGE_SIZE);
    std::string password = recv_data(_socket, STANDARD_MESSAGE_SIZE);
    std::cout << std::endl;
    bool is_duplicate = CLIENTS.find(username) != CLIENTS.end();

    if (is_duplicate) {
        std::string error_message =
protocol.ErrorMessage(ERR_LOGIN_DUPLICATES);
        send_data(_socket, error_message);
```

```
    } else {
        _this_username = username;
        CLIENTS.emplace(username, _socket);
        std::string ok_message = protocol.OkMessage();
        send_data(_socket, ok_message);

        std::string welcome_message =
protocol.WelcomeMessage(username);
        for (const auto& client : CLIENTS) {
            if (client.first != username) { send_data(client.second,
welcome_message); }
        }
    }
}

void SERVER::handle_ok() {
    std::cout << std::endl;
}

void SERVER::handle_error(const int _socket, const std::string&
_session_username) {
    char error_type_buffer[2];
    if (recv(_socket, error_type_buffer, 2, 0) == -1) {
perror("recv"); }
    std::cout << "RECEIVED ERROR " << std::string(error_type_buffer,
2) << " FROM " << _session_username <<std::endl;
}

void SERVER::handle_logout(const std::string& _session_username) {
    std::cout << std::endl;
    auto it = CLIENTS.find(_session_username);
    if (it != CLIENTS.end()) {
        std::string goodbye_message =
protocol.GoodbyeMessage(_session_username);
        for (auto it_client = CLIENTS.begin(); it_client !=
CLIENTS.end();) {
            if (it_client->first != _session_username) {
                send_data(it_client->second, goodbye_message);
                ++it_client;
            } else { it_client = CLIENTS.erase(it_client); }
        }
    }
}

void SERVER::handle_list(const int _socket) {
```

```
    std::cout << std::endl;
    std::string list_message = protocol.ListMessage(CLIENTS);
    send_data(_socket, list_message);
}

void SERVER::handle_broadcast(const int _socket, const std::string&
_session_username) {
    std::string broadcast = recv_data(_socket, STANDARD_MESSAGE_SIZE);
    std::cout << std::endl;
    std::string broadcast_message =
protocol.BroadcastMessage(broadcast, _session_username);
    for (const auto& client : CLIENTS) {
        if (client.first != _session_username) {
send_data(client.second, broadcast_message); }
    }
}

void SERVER::handle_private_message(int _socket, const std::string&
_session_username) {
    std::string receiver = recv_data(_socket, STANDARD_MESSAGE_SIZE);
    std::string message = recv_data(_socket, STANDARD_MESSAGE_SIZE);
    std::cout << std::endl;
    std::string private_message = protocol.PrivateMessage(message,
_session_username);

    auto receiver_iter = CLIENTS.find(receiver);
    if (receiver_iter != CLIENTS.end()) {
send_data(receiver_iter->second, private_message); }
    else { send_data(_socket, protocol.ErrorMessage(ERR_PRIV)); }
}

void SERVER::handle_file_transfer(const int _socket, const
std::string& _session_username) {
    std::string file_name = recv_data(_socket, STANDARD_MESSAGE_SIZE);
    char file_size_buffer[STANDARD_FILE_SIZE];
    if (recv(_socket, file_size_buffer, STANDARD_FILE_SIZE, 0) == -1)
{ perror("recv"); }
    std::cout << std::string(file_size_buffer, STANDARD_FILE_SIZE);
    ssize_t file_size = std::atoi(file_size_buffer);
    std::string receiver = recv_data(_socket, STANDARD_MESSAGE_SIZE);

    std::unique_ptr<unsigned char[]> file_buffer(new unsigned
char[file_size]);
    ssize_t numbytes = recv(_socket, file_buffer.get(), file_size, 0);
    if (numbytes == -1) { perror("recv"); }
```

```
   std::cout.write(reinterpret_cast<const char*>(file_buffer.get()),
numbytes);
   std::cout << std::endl;

   auto receiver_iter = CLIENTS.find(receiver);
   if (receiver_iter != CLIENTS.end()) {
       std::string file_message = protocol.FileMessage(file_name,
file_size, _session_username,
std::string(reinterpret_cast<char*>(file_buffer.get()), file_size));
       send_data(receiver_iter->second, file_message);
   } else { send_data(_socket, protocol.ErrorMessage(ERR_PRIV)); }
}

void SERVER::handle_tictactoe(int _socket, const std::string&
_session_username){
   char TTT_command_buffer[STANDARD_MESSAGE_SIZE];
   if (recv(_socket, TTT_command_buffer, STANDARD_MESSAGE_SIZE, 0) ==
-1) { perror("recv"); }
   std::string TTT_command(TTT_command_buffer,STANDARD_MESSAGE_SIZE);
   std::cout << TTT_command << std::endl;
   std::string TTT_broadcast_message =
protocol.BroadcastMessage(game.makeMove(TTT_command),
_session_username);
   for (const auto& client : CLIENTS) { send_data(client.second,
TTT_broadcast_message); }
}
```

Server: main.cpp

```
#include "include/server.hpp"

int main(int argc, char *argv[]) {
   if (argc != 2) {
       std::cerr << "usage: server port" << std::endl;
       exit(1);
   }

   SERVER server(argv[1]);

   return 0;
}
```

Client: net_utilities.hpp

```
#ifndef NET_UTILITIES_HPP
#define NET_UTILITIES_HPP

namespace net {
   std::string format_size(const int, int);
   std::vector<std::vector<unsigned char>> readAndDivideFile(const
std::string&, size_t);
   class PROTOCOL{
       public:
       std::string ErrorMessage(int);
       std::string OkMessage();
       std::string LoginMessage(std::string, std::string);
       std::string LogoutMessage();
       std::string ListMessage();
       std::string BroadcastMessage(std::string);
       std::string PrivateMessage(std::string, std::string);
       std::vector<std::string> FileMessages(std::string,
std::string);
       std::string TicTacToeMessage(std::string);
   };
   static std::string helpMessage;
}

#endif // NET_UTILITIES_HPP
```

Client: net_utilities.cpp

```
#include "../include/net_utilities.hpp"

std::string net::format_size(const int size, int n) {
   std::ostringstream oss;
   oss << std::setw(n) << std::setfill('0') << size;
   return oss.str();
}

std::vector<std::vector<unsigned char>> net::readAndDivideFile(const
std::string& filename, size_t chunkSize) {
   std::ifstream file(filename, std::ios::binary);
   if (!file.is_open()) {
       std::cerr << "Error opening file!" << std::endl;
```

```
        return {};
    }

    file.seekg(0, std::ios::end);
    size_t fileSize = file.tellg();
    file.seekg(0, std::ios::beg);

    std::vector<unsigned char> fileData(fileSize);
    file.read(reinterpret_cast<char*>(fileData.data()), fileSize);
    std::vector<std::vector<unsigned char>> chunks;

    size_t offset = 0;
    while (offset < fileSize) {
        size_t remainingSize = fileSize - offset;
        size_t chunkSizeBytes = std::min(remainingSize, chunkSize);

        std::vector<unsigned char> chunk(fileData.begin() + offset,
fileData.begin() + offset + chunkSizeBytes);
        chunks.push_back(std::move(chunk));

        offset += chunkSizeBytes;
    }
    return chunks;
}
// E##
std::string net::PROTOCOL::ErrorMessage(int err_n) {
    if (err_n < 0 || err_n > 99) {
        return "E00";
    }
    std::ostringstream oss;
    oss << "E" << std::setw(2) << std::setfill('0') << err_n;
    return oss.str();
}
// O
std::string net::PROTOCOL::OkMessage(){
    std::ostringstream oss;
    oss << "O";
    return oss.str();
}
// L##USERNAME##PASSWORD
std::string net::PROTOCOL::LoginMessage(std::string username,
std::string password){
    std::ostringstream oss;
    oss << "L" << net::format_size(username.size(), 2) << username <<
net::format_size(password.size(), 2) << password;
```

```
   return oss.str();
}
// U
std::string net::PROTOCOL::LogoutMessage(){
   std::ostringstream oss;
   oss << "U";
   return oss.str();
}
// T
std::string net::PROTOCOL::ListMessage(){
   std::ostringstream oss;
   oss << "T";
   return oss.str();
}
// B##MSG
std::string net::PROTOCOL::BroadcastMessage(std::string msg){
   std::ostringstream oss;
   oss << "B" << net::format_size(msg.size(), 2) << msg;
   return oss.str();
}
// M##RECEIVER##MSG
std::string net::PROTOCOL::PrivateMessage(std::string msg,
std::string receiver){
   std::ostringstream oss;
   oss << "M" << net::format_size(receiver.size(), 2) << receiver <<
net::format_size(msg.size(), 2) << msg;
   return oss.str();
}
// F##FILENAME####RECEIVERFILE
std::vector<std::string> net::PROTOCOL::FileMessages(std::string
file_name, std::string receiver) {
   std::vector<std::string> messages;
   const int chunkSize = 1024;
   std::vector<std::vector<unsigned char>> chunks =
net::readAndDivideFile(file_name, chunkSize);
   std::ostringstream oss;

   for (size_t i = 0; i < chunks.size(); ++i) {
       oss.str("");
       oss << "F" << net::format_size(file_name.size(), 2) <<
file_name
           << net::format_size(chunks[i].size(), 15)
           << net::format_size(receiver.size(), 2) << receiver;

       for (unsigned char val : chunks[i]) {
```

```
            oss << val;
        }
        messages.push_back(oss.str());
    }
    return messages;
}
//G##
std::string net::PROTOCOL::TicTacToeMessage(std::string msg){
    std::ostringstream oss;
    oss << "G" << msg;
    return oss.str();
}
```

Client: client.hpp

```
#ifndef CLIENT_HPP
#define CLIENT_HPP

#include "net_utilities.hpp"

class CLIENT {
public:
    CLIENT(char *, char *);
    ~CLIENT();

private:
    std::string username;
    int sockFD;
    net::PROTOCOL protocol;

    void write_();
    void read_();
    void start_session_();

    void try_login();

    enum MessageType {
        LOGIN = 'l',
        OK = 'O',
        ERROR = 'E',
        LOGOUT = 'u',
        LIST = 't',
```

```
        BROADCAST = 'b',
        PRIVATE_MESSAGE = 'm',
        FILE_TRANSFER = 'f'
    };

    using command_action = std::function<std::string(const
std::string&)>;
    std::map<std::string, command_action> command_actions;
    int send_message(const std::string&);

    using handler_function = std::function<void(char)>;
    std::unordered_map<char, handler_function> handle_map;
    void init_handle_map();
    void add_handler(const char, const handler_function);

    void handle_login_logout(char);
    void handle_list_users();
    void handle_message(char);
    void handle_error_message();
    void handle_file_message();

    std::string recv_string(int, int);
    void *get_in_addr(struct sockaddr *);
};

#endif // CLIENT_HPP
```

Client: client.cpp

```
#include "../include/client.hpp"
#include "../include/net_utilities.hpp"

CLIENT::CLIENT(char * hostname, char * port){
    addrinfo hints, *servinfo, *p;
    int rv;
    char server[INET6_ADDRSTRLEN];

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

    if ((rv = getaddrinfo(hostname, port, &hints, &servinfo)) != 0){
```

```
        std::cerr << "getaddrinfo: " << gai_strerror(rv) << std::endl;
        exit(1);
    }

    for(p = servinfo; p != NULL; p = p->ai_next){
        if ((sockFD = socket(p->ai_family, p->ai_socktype,
p->ai_protocol)) == -1){
            perror("client: socket");
            continue;
        }
        if (connect(sockFD, p->ai_addr, p->ai_addrlen) == -1){
            close(sockFD);
            perror("client: connect");
            continue;
        }
        break;
    }
    if (p == NULL){
        std::cerr << "client: failed to connect" << std::endl;
        exit(2);
    }
    inet_ntop(p->ai_family, get_in_addr((sockaddr *)p->ai_addr),
server, sizeof server);
    std::cout << "client: connecting to " << server << std::endl;
    freeaddrinfo(servinfo);

    init_handle_map();
    try_login();
    std::cout << "type .help to be saved" << std::endl;
    start_session_();
}

CLIENT::~CLIENT(){
    close(sockFD);
}

void CLIENT::write_() {
    std::cout << std::endl;
    std::string buffer;
    bool running = true;

    while (running) {
        std::getline(std::cin, buffer);
        if (buffer.empty()) continue;
```

```
        std::string formatted_message;
        if (buffer[0] == '@') {                          // private
message
            size_t delimiter = buffer.find(' ');
            if (delimiter != std::string::npos) {
                formatted_message =
protocol.PrivateMessage(buffer.substr(delimiter + 1),
buffer.substr(1, delimiter - 1));
            }
        } else if (buffer == ".help") {                  // list users
            std::cout << net::helpMessage << std::endl;
            continue;
        } else if (buffer == ".list") {                  // list users
            formatted_message = protocol.ListMessage();
        } else if (buffer == ".logout") {                // logout
            formatted_message = protocol.LogoutMessage();
            running = false;
        } else if (buffer.substr(0, 5) == ".file") {     // file
transfer
            size_t last_space = buffer.find_last_of(' ');
            size_t prev_space = buffer.find_last_of(' ', last_space -
1);
            std::string receiver = buffer.substr(last_space + 1);
            std::string file_name = buffer.substr(prev_space + 1,
last_space - prev_space - 1);
            std::vector<std::string> formatted_messages =
protocol.FileMessages(file_name, receiver);
            for (const auto& message : formatted_messages) {
                if (!message.empty()) {
                    if (send_message(message) == -1) { perror("send");
}
                }
            }
            buffer.clear();
            continue;
        } else if (buffer.substr(0, 4) == ".ttt") {      // tictactoe
            size_t last_space = buffer.find_last_of(' ');
            std::string command = buffer.substr(last_space + 1);
            std::cout << command << std::endl;
            formatted_message = protocol.TicTacToeMessage(command);

        } else if (!buffer.empty()){                     // public
message
            formatted_message = protocol.BroadcastMessage(buffer);
        }
```

```
        if (!formatted_message.empty()) {
            if (send_message(formatted_message) == -1) {
perror("send"); }
            buffer.clear(); formatted_message.clear();
        }
    }
}

int CLIENT::send_message(const std::string& message) {
    return send(sockFD, message.c_str(), message.size(), 0);
}

void CLIENT::read_() {
    char type_buffer[1], message_type;
    while (true) {
        if (recv(sockFD, type_buffer, 1, 0) == -1) { perror("recv"); }
        message_type = type_buffer[0];
        auto it = handle_map.find(message_type);
        if (it != handle_map.end()) {
            it->second(message_type);
        } else { }  // unknown message type
    }
    close(sockFD);
}

void CLIENT::start_session_(){
    std::thread worker_thread([this](){read_();});
    worker_thread.detach();
    write_();
}

void CLIENT::init_handle_map() {
    add_handler(LOGIN, [this](char message_type) {
handle_login_logout(message_type); });
    add_handler(LOGOUT, [this](char message_type) {
handle_login_logout(message_type); });
    add_handler(LIST, [this](char message_type)  {
handle_list_users(); });
    add_handler(BROADCAST, [this](char message_type)  {
handle_message(message_type); });
    add_handler(PRIVATE_MESSAGE, [this](char message_type)  {
handle_message(message_type); });
    add_handler(ERROR, [this](char message_type)  {
handle_error_message(); });
```

```
   add_handler(OK, [](char message_type)  {}); // no action needed
for 'OK'
   add_handler(FILE_TRANSFER, [this](char message_type)  {
handle_file_message(); });
}

void CLIENT::add_handler(const char _message_type, const
handler_function _handler) {
   handle_map[_message_type] = _handler;
}

void CLIENT::try_login() {
   std::string buffer, user, pass, login_message;
   size_t at_position;
   while (true) {
       std::cout << "Type your credentials in the format USER@PASS:
";
       std::getline(std::cin, buffer);
       at_position = buffer.find('@');

       if (at_position != std::string::npos) {
           user = buffer.substr(0, at_position);
           pass = buffer.substr(at_position + 1);
           login_message = protocol.LoginMessage(user, pass);
           if (send_message(login_message) == -1) { perror("send"); }

           char response_buffer[1];
           if (recv(sockFD, response_buffer, 1, 0) == -1) {
perror("recv"); }

           if (response_buffer[0] == 'O') {
               std::cout << "Login successful" << std::endl;
               break;
           } else if (response_buffer[0] == 'E') {
               handle_error_message();
               std::cerr << "Username already taken." << std::endl;
           }
       } else {
           std::cerr << "Invalid input format." << std::endl;
       }
   }
}

std::string CLIENT::recv_string(int _sockFD, int size_size) {
   std::unique_ptr<char[]> size_buffer(new char[size_size]);
```

```
   if (recv(_sockFD, size_buffer.get(), size_size, 0) == -1) {
perror("recv"); }
   int data_size = atoi(size_buffer.get());

   std::unique_ptr<char[]> buffer(new char[data_size]);
   ssize_t numbytes = recv(_sockFD, buffer.get(), data_size, 0);
   if (numbytes == -1) { perror("recv"); }
   return std::string(buffer.get(), numbytes);
}

void *CLIENT::get_in_addr(struct sockaddr *sa) {
   if (sa->sa_family == AF_INET) { return &(((struct
sockaddr_in*)sa)->sin_addr); }
   return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

void CLIENT::handle_login_logout(char message_type) {
   std::string action = (message_type == 'l') ? " is here!" : " left
the room.";
   std::string username = recv_string(sockFD, 2);
   std::cout << username << action << std::endl;
}
void CLIENT::handle_list_users() {
   char n_users_buffer[2];
   if (recv(sockFD, n_users_buffer, 2, 0) == -1) { perror("recv"); }
   int n_users = std::atoi(n_users_buffer);
   std::string user_list = recv_string(sockFD, 3);
   std::cout << user_list << std::endl;
}
void CLIENT::handle_message(char message_type) {
   std::string sender = recv_string(sockFD, 2);
   std::string msg = recv_string(sockFD, 2);
   std::cout << (message_type == 'b' ? sender : "priv from " +
sender) << ": " << msg << std::endl;
}
void CLIENT::handle_error_message() {
   char buffer[3];
   if (recv(sockFD, buffer, 2, 0) == -1) { perror("recv"); }
   buffer[2] = '\0';
   std::cout << "ERROR " << buffer << std::endl;
}
void CLIENT::handle_file_message() {
   std::string file_name = recv_string(sockFD, 2);
   char file_size_buffer[15];
```

```
   if (recv(sockFD, file_size_buffer, 15, 0) == -1) { perror("recv");
}
   ssize_t file_size = std::atol(file_size_buffer);
   std::string sender = recv_string(sockFD, 2);

   std::cout << "Receiving " << file_name << " from " << sender <<
std::endl;

   std::ofstream file(file_name, std::ios::binary | std::ios::app);
   if (!file.is_open()) {
       std::cerr << "Error opening file for writing: " << file_name
<< std::endl;
       return;
   }

   std::vector<unsigned char> buffer(1024);
   ssize_t remaining_bytes = file_size;
   while (remaining_bytes > 0) {
       ssize_t bytes_to_receive =
std::min(static_cast<ssize_t>(buffer.size()), remaining_bytes);
       ssize_t bytes_received = recv(sockFD,
reinterpret_cast<char*>(buffer.data()), bytes_to_receive, 0);
       if (bytes_received == -1) {
           perror("recv");
           file.close();
           return;
       }
       file.write(reinterpret_cast<const char*>(buffer.data()),
bytes_received);
       remaining_bytes -= bytes_received;
   }
   file.close();
}
```

Client: main.cpp

```
#include "include/client.hpp"

int main(int argc, char *argv[]) {
   if (argc != 3){
       std::cerr << "usage: client hostname port" << std::endl;
       exit(1);
   }

   CLIENT client(argv[1],argv[2]); return 0; }
```