Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ

к лабораторной работе №8

на тему

Интерфейс сокетов и основы сетевого программирования (Windows). Программирование взаимодействия через сеть с использованием интерфейса сокетов. Реализация сетевых протоколов: собственных или стандартных

Студент: гр.153502

Леоненко А.О.

Проверил: Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc153543953)

[2 Теоретические сведения 4](#_Toc153543954)

[3 Описание функций программы 7](#_Toc153543955)

[Список использованных источников 10](#_Toc153543956)

[Приложение А 11](#_Toc153543957)

1 ЦЕЛЬ РАБОТЫ

Целью выполнения лабораторной работы является создание приложения на *Win32 API*, которое использует сокеты для обмена данными и реализует собственный или стандартный протокол.

В качестве задачи необходимо создать сервер для многопользовательской онлайн-игры с использованием TCP сокетов для передачи игровых данных

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Операционная система *Windows* предоставляет механизмы для упрощения обмена данными и обмена данными между приложениями. В совокупности действия, включенные этими механизмами, называются межпроцессными коммуникациями (*IPC*).

Как правило, приложения могут использовать *IPC*, классифицированные как клиенты или серверы. Клиент – это приложение или процесс, запрашивающий службу из какого-либо другого приложения или процесса. Сервер – это приложение или процесс, который отвечает на запрос клиента.

Программный компонент, использующий *COM*, может взаимодействовать с множеством других компонентов, даже тех, которые еще не были написаны. Компоненты взаимодействуют как объекты и клиенты. Распределенная модель *COM* расширяет модель программирования *COM*, чтобы она работала по сети.

Сопоставление файлов позволяет процессу обрабатывать содержимое файла как блок памяти в адресном пространстве процесса. Процесс может использовать простые операции указателя для проверки и изменения содержимого файла. Если два или более процессов обращаются к одному сопоставлению файлов, каждый процесс получает указатель на память в своем адресном пространстве, который он может использовать для чтения или изменения содержимого файла. Процессы должны использовать объект синхронизации, например семафор, чтобы предотвратить повреждение данных в многозадачности.

Почтовые слои обеспечивают односторонний обмен данными. Любой процесс, создающий *mailslot*, является сервером *mailslot*. Другие процессы, называемые клиентами *mailslot*, отправляют сообщения на сервер *mailslot*, записывая сообщения в его *mailslot*. Входящие сообщения всегда добавляются к почтовому слою. *Mailslot* сохраняет сообщения до тех пор, пока сервер *mailslot* не прочитает их. Процесс может быть как сервером *mailslot*, так и клиентом *mailslot*, поэтому двусторонняя связь возможна с использованием нескольких почтовых слотов.

Существует два типа каналов для двусторонней связи: анонимные и именованные каналы. Анонимные каналы позволяют связанным процессам передавать информацию друг другу. Как правило, анонимный канал используется для перенаправления стандартных входных или выходных данных дочернего процесса, чтобы он смог обмениваться данными с родительским процессом. Для обмена данными в обоих направлениях (дуплексная операция) необходимо создать два анонимных каналов. Родительский процесс записывает данные в один канал с помощью дескриптора записи, а дочерний процесс считывает данные из этого канала с помощью дескриптора чтения. Аналогичным образом дочерний процесс записывает данные в другой канал, а родительский процесс считывает из него. Анонимные каналы не могут использоваться по сети, а также между несвязанными процессами.

Именованные каналы используются для передачи данных между процессами, которые не являются связанными, и между процессами на разных компьютерах. Как правило, процесс сервера с именованным каналом создает именованный канал с известным именем или именем, которое должно быть передано клиентам. Клиентский процесс с именованным каналом, который знает имя канала, может открыть его другой конец с ограничениями доступа, заданными серверным процессом именованного канала. После подключения сервера и клиента к каналу они могут обмениваться данными, выполняя операции чтения и записи в канале [1].

Буфер обмена – это набор функций и сообщений, которые позволяют приложениям передавать данные. Так как все приложения имеют доступ к буферу обмена, данные можно легко передавать между приложениями или внутри приложения. Буфер обмена управляется пользователем. Окно должно передавать данные в буфер обмена или из нее только в ответ на команду пользователя. Окно не должно использовать буфер обмена для передачи данных без знаний пользователя. [2]

Копирование данных реализуется через команду *WM\_COPYDATA* и структуру данных *COPYDATASTRUCT*.

Протокол *DDE* (динамического обмена данными) – это набор сообщений и рекомендаций. Он отправляет сообщения между приложениями, которые совместно используют данные, и использует общую память для обмена данными между приложениями. Приложения могут использовать протокол *DDE* для однократной передачи данных и для непрерывного обмена, в ходе которого приложения отправляют обновления друг другу по мере того, как новые данные становятся доступными

*DDEML* – это библиотека динамической компоновки (*DLL*), которую приложения могут использовать для совместного использования данных. *DDEML* предоставляет функции и сообщения, упрощающие задачу добавления возможности *DDE* в приложение [3].

Сокет – это виртуальная конструкция из *IP*-адреса и номера порта, служащая для передачи данных [4].

Обмен данными, ориентированный на соединения, может использовать надежную связь, для обеспечения которой протокол уровня 4 посылает подтверждения о получении данных и запрашивает повторную передачу, если данные не получены или искажены. Протокол *TCP* использует именно такую надежную связь. *TCP* используется в таких прикладных протоколах, как *HTTP*, *FTP*.

Протокол *TCP* требует, чтобы перед отправкой сообщения было открыто соединение. Серверное приложение должно выполнить так называемое пассивное открытие (*passive open*), чтобы создать соединение с известным номером порта, и, вместо того чтобы отправлять вызов в сеть, сервер переходит в ожидание поступления входящих запросов. Клиентское приложение должно выполнить активное открытие (*active open*), отправив серверному приложению синхронизирующий порядковый номер (*SYN*), идентифицирующий соединение. Клиентское приложение может использовать динамический номер порта в качестве локального порта. Сервер должен отправить клиенту подтверждение (*ACK*) вместе с порядковым номером (*SYN*) сервера. В свою очередь клиент отвечает *АСК*, и соединение устанавливается. После этого может начаться процесс отправки и получения сообщений. При получении сообщения в ответ всегда отправляется сообщение *АСК*. Если до получения *АСК* отправителем истекает тайм-аут, сообщение помещается в очередь на повторную передачу.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Для реализации задачи было выбрано построение многопользовательского онлайн-игрового сервера с возможностью синхронизации состояния игры между игроками. Согласно формулировке задачи, были спроектированы следующие функции программы:

– соединение игроков;

– отправление и отображение ходов в игре;

– сообщение о состоянии подключения.

Приложение состоит из двух частей. Первая отвечает за работу сервера, к которому подключаются игроки. Вторая часть отвечает за доступ игрока к игре. В серверной части обрабатываются два процесса: ожидание нового подключения и ожидание сообщения от игрока.

Клиентская часть помимо функций обращения к серверу создаёт пользовательский интерфейс (рисунок 3.1).

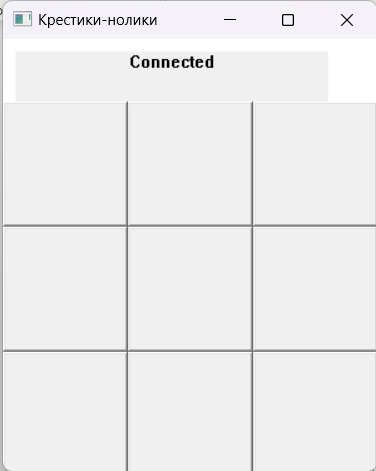


Рисунок 3.1 – Окно приложения

Окно приложения содержит поле состояния игры и кнопки поля. Подключение к серверу осуществляется автоматически при запуске приложения. После соединения с другим пользователем поле состояния игры сообщает о том, что игра началась и отображает очередность ходов (рисунок 3.2).

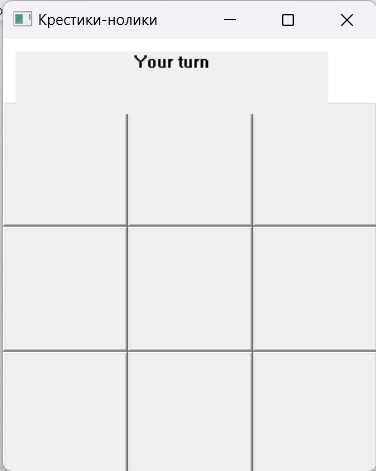


Рисунок 3.2 – Окно приложения после подключения к игре другого игрока

Далее, соблюдая очередность хода, игроки могут располагать на поле символы игры (рисунок 3.3).

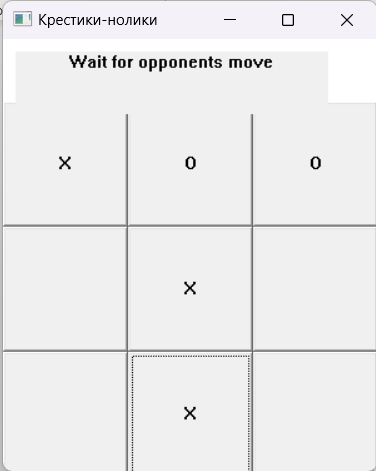


Рисунок 3.3 – Процесс игры

При завершении игры будет отображена соответствующая информация для всех игроков (рисунок 3.4).

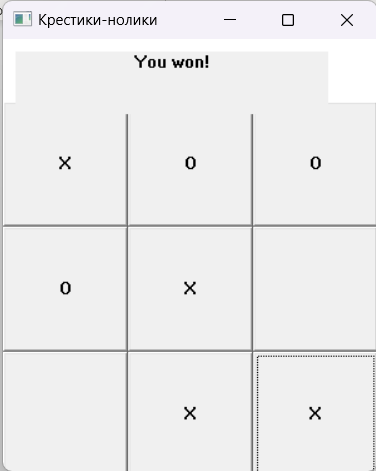


Рисунок 3.4 – Сообщение о выигрыше игрока

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Взаимодействие между процессами [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/ipc/interprocess-communications.

[2] Сведения о буфере обмена [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/dataxchg/about-the-clipboard.

[3] Сведения о динамическом обмене данными [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/dataxchg/about-dynamic-data-exchange.

[4] Что такое сокет и зачем он нужен [Электронный ресурс]. – Режим доступа: https://thecode.media/socket/.

[5] Протоколы *TCP* и *UDP* [Электронный ресурс]. – Режим доступа: https://professorweb.ru/my/csharp/web/level1/1\_4.php.

ПРИЛОЖЕНИЕ А

**Исходный код программы**

**Файл Lab7\_8\_Server.cpp**

#include <iostream>

#include <string>

#include <vector>

#include <winsock2.h>

#pragma comment(lib, "ws2\_32.lib")

#include <ws2tcpip.h>

int connected\_num = 0;

enum gameCondition {

START\_GAME\_YOU = 0,

START\_GAME\_WAIT = 1,

MOVE = 2,

DISCONNECT = 3,

READY = 4,

};

char\* SendData(gameCondition cond, int x = 0, int y = 0, bool senderWon = false);

class GameRoom {

public:

int join(SOCKET gamerSocket)

{

gamers.push\_back(gamerSocket);

std::cout << "Player connected\n";

if (connected\_num == 2) {

char\* msg1 = SendData(START\_GAME\_YOU);

char\* msg2 = SendData(START\_GAME\_WAIT);

send(gamers[0], msg1, sizeof(msg1), 0);

send(gamers[1], msg2, sizeof(msg2), 0);

delete(msg1);

delete(msg2);

std::cout << "Game started\n";

}

return 1;

}

void leave(SOCKET clientSocket)

{

auto it = std::find\_if(gamers.begin(), gamers.end(), [clientSocket](const SOCKET& info) {

return info == clientSocket;

});

if (it != gamers.end()) {

gamers.erase(it);

}

}

void sendMessage(const char\* message, SOCKET srcSocket, SOCKET excludeGamer = INVALID\_SOCKET)

{

for (const auto& gamer : gamers) {

if (srcSocket != excludeGamer && srcSocket != gamer) {

send(gamer, message, sizeof(message), 0);

}

}

}

private:

std::vector<SOCKET> gamers;

};

GameRoom globalRoom;

DWORD WINAPI ClientThread(LPVOID lpParam)

{

SOCKET clientSocket = reinterpret\_cast<SOCKET>(lpParam);

char buffer[1024];

int recvResult;

//recvResult = recv(clientSocket, buffer, sizeof(buffer), 0); // returns the number of bytes received

//socket descriptor, pointer to buffer, len of buffer, flags

int res = globalRoom.join(clientSocket);

while (true) {

recvResult = recv(clientSocket, buffer, sizeof(buffer), 0);

std::cout << "recieved some messages\n";

if (recvResult > 0) {

buffer[recvResult] = '\0';

std::cout << "some message was sent\n";

globalRoom.sendMessage(buffer, clientSocket);

}

else if (recvResult == 0) {

globalRoom.leave(clientSocket);

closesocket(clientSocket);

connected\_num--;

std::cout << "Client disconnected\n";

break;

}

else {

std::cerr << "Recv failed\n";

globalRoom.leave(clientSocket);

closesocket(clientSocket);

connected\_num--;

globalRoom.sendMessage(SendData(DISCONNECT), clientSocket);

break;

}

}

return 0;

}

int main()

{

WSADATA wsaData; // information about the Windows Sockets implementation

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) { // initiates use of the Winsock DLL by a process like a load library

//version and wsadata

std::cerr << "WSAStartup failed\n";

return -1;

}

SOCKET serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);

// af\_inet - ipv4 (adress family specification), sock\_stream - for stream socket

if (serverSocket == INVALID\_SOCKET) {

std::cerr << "Error creating socket\n";

WSACleanup();

return -1;

}

sockaddr\_in serverAddr; //specifies a transport address and port for the AF\_INET address family

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(8888); // converts port to big-endian

serverAddr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind(serverSocket, reinterpret\_cast<sockaddr\*>(&serverAddr), sizeof(serverAddr)) == SOCKET\_ERROR) { // bind local address with a socket

// socket descriptor, pointer to sockaddr, length of sockaddr

std::cerr << "Bind failed\n";

closesocket(serverSocket);

WSACleanup();

return -1;

}

if (listen(serverSocket, 2) == SOCKET\_ERROR) { //somaxconn - max

std::cerr << "Listen failed\n";

closesocket(serverSocket);

WSACleanup();

return -1;

}

std::cout << "Server is listening on port 8888...\n";

while (true) {

SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);

if (clientSocket == INVALID\_SOCKET) {

std::cerr << "Accept failed\n";

break;

}

if (connected\_num >= 2) {

std::cerr << "Max amount of players\n";

char\* msg = SendData(DISCONNECT);

send(clientSocket, msg, sizeof(msg), 0);

delete(msg);

closesocket(clientSocket);

continue;

}

connected\_num++;

std::cout << "Client connected\n";

HANDLE threadHandle = CreateThread(nullptr, 0, ClientThread, reinterpret\_cast<LPVOID>(clientSocket), 0, nullptr);

//pointer to a SECURITY\_ATTRIBUTES (null - cannot be inherited), size (0 - default value is used), function, pointer to params to pass, flags (0 - runs immidiately)

//pointer to a variable that recieves thread identifier

if (threadHandle == nullptr) {

std::cerr << "Failed to create thread\n";

closesocket(clientSocket);

break;

}

}

closesocket(serverSocket);

WSACleanup();

return 0;

}

char\* SendData(gameCondition cond, int x, int y, bool senderWon) {

char\* buff = new char[4];

buff[0] = static\_cast<WCHAR>(cond);

buff[1] = static\_cast<WCHAR>(x);

buff[2] = static\_cast<WCHAR>(y);

buff[3] = senderWon ? L'1' : L'0';

return buff;

}

**Файл Lab7\_8\_Game\_Client.cpp**

#include <WS2tcpip.h>

#include <windows.h>

#include <string>

#include <commctrl.h>

#pragma comment(lib, "ws2\_32.lib")

#pragma comment(lib, "comctl32.lib")

//HWND g\_hListView;

SOCKET g\_clientSocket;

HANDLE g\_hThread;

//HWND hwndEdit;

//HWND hwndBtn;

HWND hwndText;

HWND gameButtons[9];

bool your\_turn = false;

bool game\_end = false;

enum gameCondition {

START\_GAME\_YOU = 0,

START\_GAME\_WAIT = 1,

MOVE = 2,

DISCONNECT = 3,

READY = 4,

};

void SendData(gameCondition cond, int x, int y, bool senderWon);

void ReadData(char\* data);

void ClearField();

struct GameState {

char field[9];

char winner;

};

void InitializeGameState(GameState\* gameState) {

for (int i = 0; i < 9; i++) {

gameState->field[i] = ' ';

}

gameState->winner = ' ';

}

bool CheckWinner(GameState\* gameState) {

// Check rows

for (int i = 0; i < 3; i++) {

if (gameState->field[i \* 3] == gameState->field[i \* 3 + 1] &&

gameState->field[i \* 3] == gameState->field[i \* 3 + 2] &&

gameState->field[i \* 3] != ' ') {

gameState->winner = gameState->field[i \* 3];

return true;

}

}

// Check columns

for (int i = 0; i < 3; i++) {

if (gameState->field[i] == gameState->field[i + 3] &&

gameState->field[i] == gameState->field[i + 6] &&

gameState->field[i] != ' ') {

gameState->winner = gameState->field[i];

return true;

}

}

// Check diagonals

if (gameState->field[0] == gameState->field[4] &&

gameState->field[0] == gameState->field[8] &&

gameState->field[0] != ' ') {

gameState->winner = gameState->field[0];

return true;

}

if (gameState->field[2] == gameState->field[4] &&

gameState->field[2] == gameState->field[6] &&

gameState->field[2] != ' ') {

gameState->winner = gameState->field[2];

return true;

}

bool freeSpace = false;

for (int i = 0; i < 9; i++) {

if (gameState->field[i] == ' ') {

freeSpace = true;

}

}

if (!freeSpace) {

return true;

}

return false;

}

GameState\* currentGameState = new GameState;

void SendMessageToServer(const char\* message)

{

send(g\_clientSocket, message, sizeof(message), 0);

}

DWORD WINAPI MessageListener(LPVOID lpParam)

{

char buffer[1024];

while (true) {

int recvResult = recv(g\_clientSocket, buffer, sizeof(buffer), 0);

if (recvResult > 0) {

buffer[recvResult] = '\0';

char\* message = new char[1024];

for (int i = 0; i < 1024; i++) {

message[i] = buffer[i];

}

ReadData(message);

delete(message);

}

else if (recvResult == 0) {

closesocket(g\_clientSocket);

MessageBox(nullptr, L"Server disconnected", L"Info", MB\_OK | MB\_ICONINFORMATION);

break;

}

else {

MessageBox(nullptr, L"Error receiving message", L"Error", MB\_OK | MB\_ICONERROR);

closesocket(g\_clientSocket);

break;

}

}

}

bool isConnected;

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

int button\_i = 0;

int x, y;

switch (uMsg) {

case WM\_CREATE: {

INITCOMMONCONTROLSEX icex;

icex.dwSize = sizeof(INITCOMMONCONTROLSEX);

icex.dwICC = ICC\_LISTVIEW\_CLASSES;

InitCommonControlsEx(&icex);

hwndText = CreateWindow(

L"STATIC", L"window text", WS\_CHILD | WS\_VISIBLE | SS\_CENTER,

10, 10, 250, 50, hwnd, nullptr, nullptr, nullptr);

for (button\_i = 0; button\_i < 9; button\_i++) {

gameButtons[button\_i] = CreateWindow(L"BUTTON", L"", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

(button\_i % 3) \* 100, (button\_i / 3) \* 100 + 50, 100, 100, hwnd, (HMENU)(1000 + button\_i), NULL, NULL);

}

break;

}

case WM\_COMMAND: {

if (game\_end) {

SendData(START\_GAME\_WAIT, 0, 0, false);

SetWindowText(hwndText, L"Your turn");

ClearField();

game\_end = false;

your\_turn = true;

}

else if (your\_turn) {

your\_turn = false;

button\_i = LOWORD(wParam) % 1000;

x = button\_i / 3;

y = button\_i % 3;

// check for rules

SetWindowText(gameButtons[button\_i], L"X");

currentGameState->field[button\_i] = 'X';

if (CheckWinner(currentGameState)) {

if (currentGameState->winner == 'X') {

SetWindowText(hwndText, L"You won!");

SendData(MOVE, x, y, true);

game\_end = true;

}

else if (currentGameState->winner == ' ') {

SendData(MOVE, x, y, false);

SetWindowText(hwndText, L"Nobody won");

game\_end = true;

}

}

else {

SendData(MOVE, x, y, false);

SetWindowText(hwndText, L"Wait for opponents move");

}

}

break;

}

case WM\_DESTROY: {

PostQuitMessage(0);

break;

}

default: {

return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

}

return 0;

}

int WINAPI wWinMain(\_In\_ HINSTANCE hInstance, \_In\_opt\_ HINSTANCE hPrevInstance, \_In\_ LPWSTR lpCmdLine, \_In\_ int nCmdShow)

{

UNREFERENCED\_PARAMETER(hPrevInstance);

UNREFERENCED\_PARAMETER(lpCmdLine);

WSADATA wsaData;

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

return -1;

}

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;

wc.hInstance = hInstance;

wc.lpszClassName = L"GameClientClass";

if (RegisterClass(&wc)) {

HWND hwnd = CreateWindowEx(

0, L"GameClientClass", L"Крестики-нолики",

WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, CW\_USEDEFAULT, 315, 385,

nullptr, nullptr, hInstance, nullptr);

if (hwnd) {

ShowWindow(hwnd, nCmdShow);

InitializeGameState(currentGameState);

if (!isConnected) {

g\_clientSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (g\_clientSocket == INVALID\_SOCKET) {

MessageBox(hwnd, L"Error creating socket", L"Error", MB\_OK | MB\_ICONERROR);

return 0;

}

sockaddr\_in serverAddr;

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(8888);

inet\_pton(AF\_INET, "127.0.0.1", &serverAddr.sin\_addr); // ip to binary

//family, address, buffer

if (connect(g\_clientSocket, reinterpret\_cast<sockaddr\*>(&serverAddr), sizeof(serverAddr)) == SOCKET\_ERROR) {

MessageBox(hwnd, L"Error connecting to server", L"Error", MB\_OK | MB\_ICONERROR);

closesocket(g\_clientSocket);

return 0;

}

// first data send

//WCHAR buff[1024];

//GetWindowText(hwndEdit, buff, 1024);

//SendMessageToServer(buff);

isConnected = true;

SetWindowText(hwndText, L"Connected");

g\_hThread = CreateThread(nullptr, 0, MessageListener, nullptr, 0, nullptr);

//SetWindowText(hwndBtn, L"Send");

}

MSG msg;

while (GetMessage(&msg, nullptr, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

}

return 0;

}

void SendData(gameCondition cond, int x = 0, int y = 0, bool senderWon = false) {

char\* buff = new char[4];

buff[0] = static\_cast<WCHAR>(cond);

buff[1] = static\_cast<WCHAR>(x);

buff[2] = static\_cast<WCHAR>(y);

buff[3] = senderWon ? L'1' : L'0';

SendMessageToServer(buff);

}

void ReadData(char\* data) {

gameCondition cond = static\_cast<gameCondition>(data[0]);

int x = static\_cast<int>(data[1]);

int y = static\_cast<int>(data[2]);

bool senderWon = data[3] == '1' ? true : false;

switch (cond) {

case START\_GAME\_WAIT:

SetWindowText(hwndText, L"Wait for opponents move");

your\_turn = false;

game\_end = false;

ClearField();

break;

case START\_GAME\_YOU:

SetWindowText(hwndText, L"Your turn");

your\_turn = true;

break;

case MOVE:

SetWindowText(gameButtons[x \* 3 + y], L"O");

currentGameState->field[x \* 3 + y] = 'O';

if (CheckWinner(currentGameState)) {

if (currentGameState->winner == 'O') {

SetWindowText(hwndText, L"You losed!");

game\_end = true;

}

else if (currentGameState->winner == ' ') {

SetWindowText(hwndText, L"Nobody won");

game\_end = true;

}

}

else {

SetWindowText(hwndText, L"Your turn");

your\_turn = true;

}

break;

case DISCONNECT:

SetWindowText(hwndText, L"Unable to connect");

your\_turn = false;

break;

}

}

void ClearField() {

for (int i = 0; i < 9; i++) {

SetWindowText(gameButtons[i], L"");

currentGameState->field[i] = ' ';

currentGameState->winner = ' ';

}

}