МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

Кафедра вычислительных систем

Отчёт

по курсовому проекту

по дисциплине «Сетевое программирование»

на тему «Разработка сетевого приложения «Сетевая игра». Реализация на базе протоколов UDP.»

Выполнил: студент группы ИС-142

Куликов Д.А.

Проверил: Ассистент

Третьяков Г. Н.

Новосибирск – 2024

Оглавление

[1. Постановка задачи 3](#_Toc167354778)

[2. Описание протокола UDP 4](#_Toc167354779)

[3. Описание реализации 5](#_Toc167354780)

[4. Скриншоты программы 6](#_Toc167354781)

[Список литературы 7](#_Toc167354782)

[Приложение 8](#_Toc167354783)

# Постановка задачи

Протокол UDP (User Datagram Protocol) — это один из основных протоколов транспортного уровня, используемых в интернете. Он обеспечивает передачу данных без установления соединения, что делает его быстрым и легковесным. Благодаря высокой скорости этот протокол часто используются в видеоиграх.

Цель: Разработать сетевое приложение – сетевую игру наподобие diep.io, использующую протокол UDP для обмена данными между клиентами и сервером. Основная работа заключается в создании сообщений для входа/выхода из игры, перемещения, стрельбы и получения урона.

Задачи:

* Разработка клиентской и серверной частей приложения.
* Реализация механизма обмена сообщениями с использованием протокола UDP.
* Обработка сообщений о входе и выходе из игры.
* Обработка сообщений о перемещении, стрельбе и получении урона.
* Обеспечение отображения игровых событий в реальном времени.

# 2. Описание протокола UDP

UDP (User Datagram Protocol) — это один из основных протоколов в интернете, используемый для передачи сообщений (датаграмм) без установления соединения. Он обеспечивает высокую скорость передачи данных, но не гарантирует доставку, порядок или защиту от дублирования пакетов.

Основные характеристики UDP:

* Отсутствие установления соединения перед передачей данных.
* Высокая скорость передачи данных.
* Нет гарантии доставки данных.
* Нет контроля последовательности доставки.
* Низкая задержка передачи.

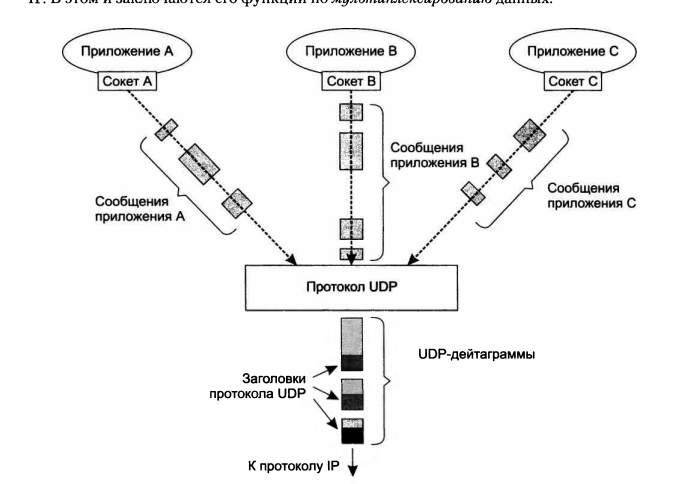
Преимущества:

* Быстрая передача данных.
* Низкие накладные расходы.

Недостатки:

* Возможная потеря пакетов.
* Неупорядоченная доставка пакетов.

UDP используется в приложениях, где важна скорость передачи данных, например, в онлайн-играх, потоковом видео и аудио, а также в DNS-запросах.



# 3. Описание реализации

Программа реализована на языке C++. Для клиента и сервера были реализованы специальные классы, что упростило разработку и повысило модульность программы. Основной функционал классов:

Клиент:

* Создает UDP-сокет.
* Отправляет запросы на сервер для входа в игру.
* Отправляет данные о перемещении, стрельбе и получении урона.
* Получает и обрабатывает сообщения от сервера о состоянии игры и других игроков.

Сервер:

* Создает UDP-сокет.
* Принимает запросы от клиентов.
* Обрабатывает данные о перемещении, стрельбе и получении урона.
* Рассылает клиентам актуальные данные о состоянии игры.

В клиентской и серверной программах используется по 2 потока. Один необходим для получения/отправления сообщений, а второй для вычислений игрового процесса. В программе из-за много поточности могут произойти конфликтов данных, чтобы их избежать использовались мьютексы. Мьютексы гарантируют, что два потока не будут одновременно работать с одними и теми же данными.

Основной поток клиента отвечает за рендеринг игрового процесса, вычисление коллизий и перемещение объектов. Второй поток нужен для получения актуальной информации об игре.

Основной поток в серверной программе отвечает за коммуникацию с клиентами. Второй поток вычисляет перемещение снарядов игроков, коллизию и нанесение урона.

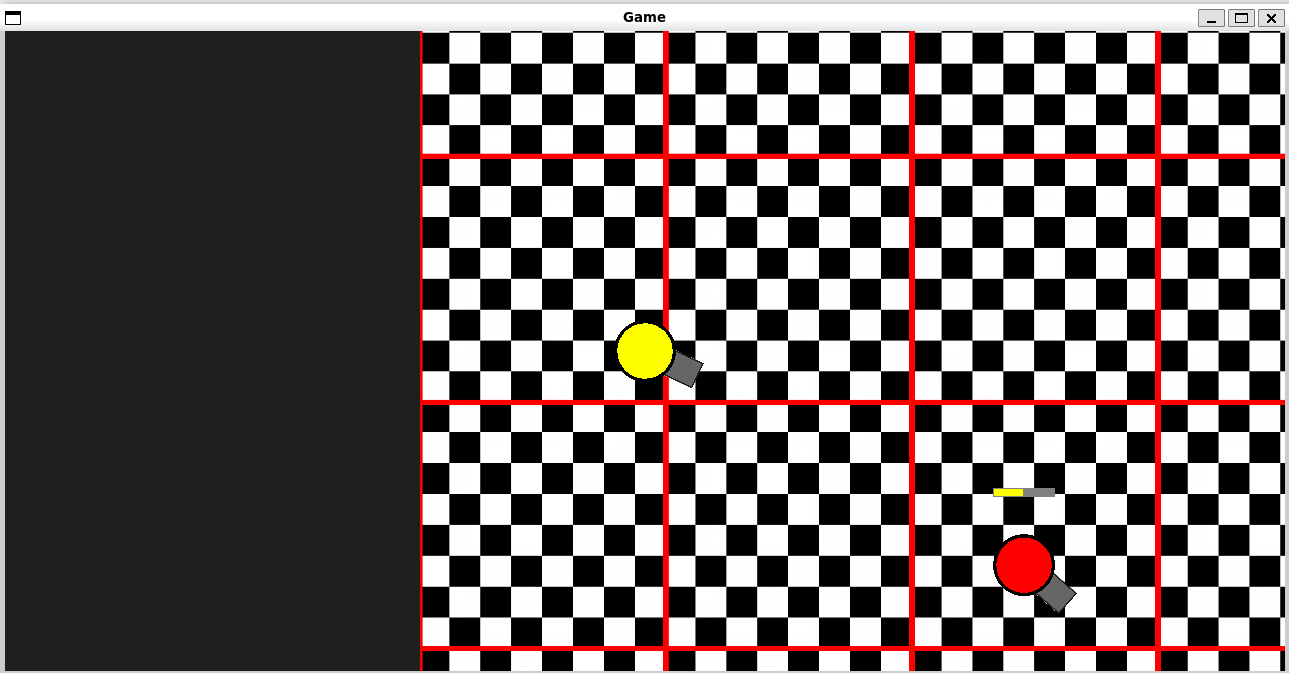
Для коммуникации сообщения имею определённый вид. Например, от клиента серверу:

* Код события (присоединение к игре, перемещения, выстрел)
* Дополнительная информация

Сообщения от сервера клиентам имеют дополнительное поле с ID игрока, которому обращено действие/событие.

Когда программа получает сообщение, она сначала проверяет код события и на его основании начинает выполнять какое-либо действие.

# 4. Скриншоты программы



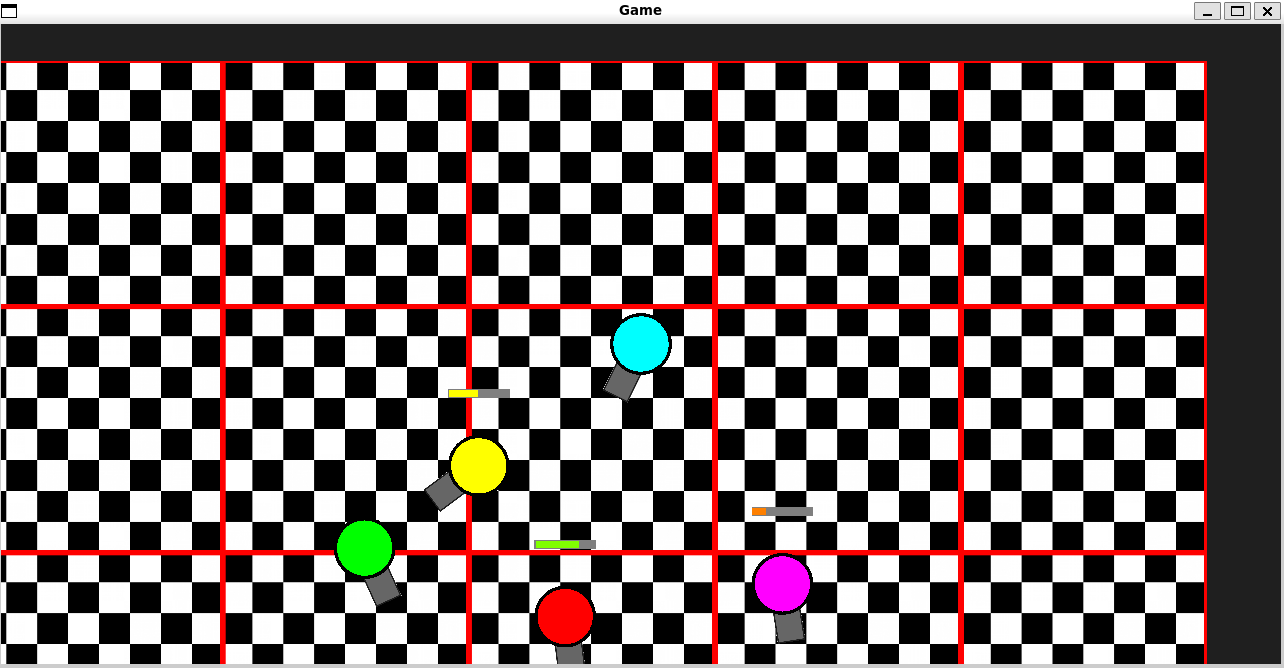


Рис. 1

# Список литературы

СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ Компьютерные Принципы, технологии, протоколы 5-е издание В. Олифер, Н. Олифер

Архитектура вычислительных систем - В.Г.Хорошевский

Приложение  
include  
#define MAX\_PLAYERS 6

#define MAX\_SIZE 64

#define PORT 8888

#define SERVER\_IP "127.0.0.1"

#define PLAYER\_SPEED 2.0

#define MAX\_HP 100

#define BULLET\_DAMAGE 100 / 4

#define BULLET\_SPEED PLAYER\_SPEED \* 1.5

enum {

    CLIENT\_CONNECT,

    CLIENT\_DISCONNECT,

    MOVE,

    FIRE,

    SET\_HP

};

struct MOVE\_INFO {

    int code\_message;

    Vector3<float> position;

    float direction;

};

struct FIRE\_INFO {

    int code\_message;

    Vector3<float> position;

    float direction;

};

struct ID\_MOVE\_INFO {

    int code\_message;

    Vector3<float> position;

    float direction;

    int id;

};

struct ID\_FIRE\_INFO {

    int code\_message;

    Vector3<float> position;

    float direction;

    int id;

};

struct ID\_SET\_HP\_INFO {

    int code\_message;

    int HP;

    int id;

};

struct ID\_DISCONNECT {

    int code\_message;

    int id;

};

using message\_type = int;

Client.hpp  
#pragma once

#include "../lib-project/lib.hpp"

#include "player.hpp"

#include "unit/unitInfo.hpp"

class Client

{

public:

    Client() {

        sockfd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

        if (sockfd < 0) {

            perror("Ошибка при создании сокета");

            exit(EXIT\_FAILURE);

        }

        memset(&server\_addr, 0, sizeof(server\_addr));

        server\_addr.sin\_family = AF\_INET;

        server\_addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);

        server\_addr.sin\_port = htons(PORT);

        if (connect(sockfd, (const struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

            perror("Ошибка при установлении соединения");

            exit(EXIT\_FAILURE);

        }

    }

    void connect\_game() {

        buffer[0] = CLIENT\_CONNECT;

        send(sockfd, (const message\_type \*)&buffer[0], MAX\_SIZE, 0);

        struct ID\_MOVE\_INFO id\_move\_info;

        n = recv(sockfd, (ID\_MOVE\_INFO\*)&id\_move\_info, sizeof(struct ID\_MOVE\_INFO), 0);

        mutex\_lock();

        id = id\_move\_info.id;

        printf("Получено от сервера: %d\n", id);

        mutex\_unlock();

        spawn = id\_move\_info.position;

    }

    void disconnect\_game() {

        buffer[0] = CLIENT\_DISCONNECT;

        send(sockfd, (const message\_type \*)&buffer[0], MAX\_SIZE, 0);

    }

    void fire(const Vector3<float>& position, const float direction, const int id) {

        struct FIRE\_INFO fire\_info = {

            FIRE,

            position,

            direction

        };

        send(sockfd, (const struct FIRE\_INFO \*)&fire\_info, sizeof(struct FIRE\_INFO), 0);

    }

    void move(const Vector3<float>& position, const float direction, const int id) {

        struct MOVE\_INFO move\_info = {

            MOVE,

            position,

            direction

        };

        send(sockfd, (const struct FIRE\_INFO \*)&move\_info, sizeof(struct MOVE\_INFO), 0);

    }

    int GetID() { return id; }

    void SetScene(Scene\* scene\_) { scene = scene\_; }

    void mutex\_lock()   { pthread\_mutex\_lock(&player\_mutex); }

    void mutex\_unlock() { pthread\_mutex\_unlock(&player\_mutex); }

    void move(struct ID\_MOVE\_INFO\* id\_move\_info) {

        if (id\_move\_info->id == scene->player->id) {

            mutex\_lock();

            scene->player->Teleport(id\_move\_info->position);

            scene->player->args.deg = id\_move\_info->direction;

            mutex\_unlock();

        } else {

            if (id\_move\_info->id < 0 || id\_move\_info->id >= MAX\_PLAYERS) return;

            scene->players[id\_move\_info->id].Teleport(id\_move\_info->position);

            scene->players[id\_move\_info->id].args.deg = id\_move\_info->direction;

        }

    }

    void fire(struct ID\_FIRE\_INFO\* id\_fire\_info) {

        move(reinterpret\_cast<ID\_MOVE\_INFO\*>(id\_fire\_info));

        Bullet\* bullet = scene->players[id\_fire\_info->id].Fire();

        if (bullet != nullptr) {

            scene->bullets.push\_back(bullet);

        }

    }

    void set\_hp(struct ID\_SET\_HP\_INFO\* id\_set\_hp\_\_info) {

        scene->players[id\_set\_hp\_\_info->id].params.HP = id\_set\_hp\_\_info->HP;

        if (scene->players[id\_set\_hp\_\_info->id].params.HP <= 0) {

            scene->players[id\_set\_hp\_\_info->id].GetTransform()->Move(100000.0, 100000.0, 0.0);

        }

    }

    bool client\_disconnect(struct ID\_DISCONNECT\* id\_disconnect) {

        return scene->player->id == id\_disconnect->id;

    }

    void callback() {

        while (true) {

            n = recv(sockfd, (message\_type \*)buffer, MAX\_SIZE, 0);

            switch (buffer[0])

            {

            case MOVE:

                move(reinterpret\_cast<ID\_MOVE\_INFO\*>(buffer));

                break;

            case FIRE:

                fire(reinterpret\_cast<ID\_FIRE\_INFO\*>(buffer));

                break;

            case SET\_HP:

                set\_hp(reinterpret\_cast<ID\_SET\_HP\_INFO\*>(buffer));

                break;

            case CLIENT\_DISCONNECT:

                if (client\_disconnect(reinterpret\_cast<ID\_DISCONNECT\*>(buffer))) return;

                break;

            default:

                break;

            }

        }

    }

// private:

    struct sockaddr\_in server\_addr;

    int sockfd, n;

    int id = 777;

    message\_type buffer[MAX\_SIZE];

    Vector3<float> spawn;

    Scene\* scene = nullptr;

    pthread\_mutex\_t player\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

};

Main\_client.cpp  
#include <render/glfw.hpp>

#include <game/gameManager.hpp>

void GameCallback()

{

    while (!GameManager::IsEnd) {

        double xpos, ypos;

        glfwGetCursorPos(GameManager::window, &xpos, &ypos);

        xpos -= GameManager::width / 2;

        ypos -= GameManager::height / 2;

        ypos = -ypos ;

        GameManager::deg = atan2(ypos, xpos);

        GameManager::deg = -ToDegree(GameManager::deg);

        glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

        GameManager::IsEnd = RenderSceneCB(GameManager::render, GameManager::scene);

        glfwSwapBuffers(GameManager::window);

        glfwPollEvents();

    }

}

int main(int argc, char\*\* argv)

{

    std::thread client\_callback([&]() {

        GameManager::client.connect\_game();

        bool wait = true;

        while (wait) {

            GameManager::client.mutex\_lock();

            wait = GameManager::scene == nullptr;

            GameManager::client.mutex\_unlock();

        }

        GameManager::client.scene = GameManager::scene;

        GameManager::client.callback();

        GameManager::IsEnd = true;

    });

    client\_callback.detach();

    bool wait = true;

    while(wait) {

        GameManager::client.mutex\_lock();

        wait = GameManager::client.GetID() == 777;

        GameManager::client.mutex\_unlock();

    }

    if (GameManager::client.GetID() >= MAX\_PLAYERS) return -1;

    const int width = 1280, height = 640;

    Render \*render = GameManager::InitializeGLFW(width, height);

    GameManager::InitializeObjects();

    Scene \*scene = createScene(GameManager::client.GetID());

    GameManager::client.mutex\_lock();

    GameManager::PushScene(scene);

    GameManager::client.mutex\_unlock();

    GameCallback();

    glfwTerminate();

    GameManager::client.disconnect\_game();

    return 0;

}

Glfw.cpp  
#include <game/gameManager.hpp>

#include <entities/templates/playable/unit.hpp>

#include <entities/templates/mobs/bullet.hpp>

float RandF() {

    float min = -7.5;

    float max = 7.5;

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_real\_distribution<float> dis(min, max);

    return dis(gen);

}

std::chrono::milliseconds totalTime(0);

bool RenderSceneCB(Render \*render, Scene \*scene)

{

    static time\_t prev = time(0);

    static int frame = 0;

    static const int targetFPS = 120;

    static const int targetMessage = 40;

    static const std::chrono::milliseconds frameDuration(1000 / targetFPS);

    static const std::chrono::milliseconds frameDurationMessage(1000 / targetMessage);

    static std::chrono::steady\_clock::time\_point framePrevious = std::chrono::steady\_clock::now();

    static std::chrono::steady\_clock::time\_point framePreviousMessage = std::chrono::steady\_clock::now();

    static std::chrono::steady\_clock::time\_point previous = std::chrono::steady\_clock::now();

    std::chrono::steady\_clock::time\_point now = std::chrono::steady\_clock::now();

    // std::chrono::steady\_clock::time\_point cut = now - previous;

    // std::chrono::steady\_clock::time\_point frameEnd = now + frameDuration;

    // std::this\_thread::sleep\_until(frameEnd);

    scene->Time.Update();

    static float cut = scene->Time.GetCurrentTime();

    static float prev\_time = scene->Time.GetCurrentTime();

    scene->Time.Update();

    cut = (scene->Time.GetCurrentTime() - prev\_time) \* 1e-8;

    prev\_time = scene->Time.GetCurrentTime();

    // std::cout << cut << std::endl;

    Actor::PushTime(scene->Time.GetCurrentTime());

    for (std::vector<Component>::iterator it = scene->getIterator(); it != scene->component.end(); it++)

        GameManager::render->drawObject(&it->transform, it->sprite);

    GameManager::client.mutex\_lock();

    scene->player->MoveForward(static\_cast<float>(cut) \* scene->player->GetSpeed());

    GameManager::client.mutex\_unlock();

    scene->player->UpdateCameraPos();

    scene->player->args.deg = GameManager::deg;

    // Коллизия пуль

    for (auto blt = scene->bullets.begin(); blt != scene->bullets.end();) {

        (\*blt)->MoveForward(cut \* (\*blt)->GetSpeed());

        if ((scene->Time.GetCurrentTime() - (\*blt)->GetBirthTime()) > Bullet::lifetime) {

            blt = scene->bullets.erase(blt);

            if (blt == scene->bullets.end()) break;

        }

        for (auto& plr : GameManager::scene->players) {

            if ((plr.first != (\*blt)->id) && (plr.second.params.HP > 0)) {

                if ((\*blt)->GetTransform()->WorldPos.Distance(plr.second.GetTransform()->WorldPos) < (1.0 + 0.5)) {

                    blt = scene->bullets.erase(blt);

                    break;

                }

            }

        }

        if (blt == scene->bullets.end()) break;

        ++blt;

    }

    // отрисовка игроков

    for (auto& it : scene->players) {

        it.second.Update();

        std::vector<Component\*> ActorComponents = it.second.getActorComponents();

        for (auto ac : ActorComponents)  GameManager::render->drawObject(&ac->transform, ac->sprite);

    }

    // выстрелы

    if (scene->player->isFire) {

        Bullet\* bullet = scene->player->Fire();

        GameManager::client.fire(scene->player->GetTransform()->GetWorldPos(), scene->player->args.deg, scene->player->id);

        scene->player->isFire = false;

        if (bullet != nullptr) {

            scene->bullets.push\_back(bullet);

        }

    }

    // отрисовка пуль

    for (auto blt = scene->bullets.begin(); blt != scene->bullets.end();) {

        std::vector<Component\*> ActorComponents;

        (\*blt)->Update();

        ActorComponents = (\*blt)->getActorComponents();

        for (auto it : ActorComponents) GameManager::render->drawObject(&it->transform, it->sprite);

        blt++;

    }

    if (now - framePreviousMessage >= frameDurationMessage) {

        GameManager::client.move(scene->player->GetTransform()->GetWorldPos(), scene->player->args.deg, scene->player->id);

        framePreviousMessage = now;

    }

    std::chrono::steady\_clock::time\_point frameEnd = now + frameDuration;

    std::this\_thread::sleep\_until(frameEnd);

    return GameManager::IsEnd;

}

Server.hpp  
#pragma once

#include "../lib-project/lib.hpp"

#include "player.hpp"

// #include "unit/unitInfo.hpp"

#include "../entities/templates/playable/unit.hpp"

#include "../entities/templates/mobs/bullet.hpp"

#include <map>

#include <utility>

#include <set>

#define TPS 30

int generateRandomInt(int min, int max) {

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_int\_distribution<int> dis(min, max);

    return dis(gen);

}

Unit gun;

class Server

{

public:

    Server() {

        sockfd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

        if (sockfd < 0) {

            perror("Ошибка при создании сокета");

            exit(EXIT\_FAILURE);

        }

        memset(&server\_addr, 0, sizeof(server\_addr));

        server\_addr.sin\_family = AF\_INET;

        server\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

        server\_addr.sin\_port = htons(PORT);

        if (bind(sockfd, (const struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

            perror("Ошибка при привязке сокета");

            exit(EXIT\_FAILURE);

        }

        for (int i = 0; i < MAX\_PLAYERS; i++)  id[i] = true;

    }

    int CreatePlayer(const struct sockaddr\_in& client\_addr\_) {

        std::pair<int, int> player\_info(client\_addr\_.sin\_addr.s\_addr, client\_addr\_.sin\_port);

        int free\_id;

        for (free\_id = 0; free\_id < MAX\_PLAYERS; free\_id++) {

            if (id[free\_id]) break;

        }

        if (free\_id == MAX\_PLAYERS) return free\_id;

        Player player(client\_addr\_, free\_id);

        players.insert(std::make\_pair(player\_info, player));

        id[free\_id] = false;

        for (int i = 0; i < MAX\_PLAYERS; i++) {

            std::cout << id[i] << " ";

        }

        std::cout << std::endl;

        return free\_id;

    }

    void callback() {

        struct sockaddr\_in client\_addr;

        unsigned int len = sizeof(client\_addr);

        while (true) {

            n = recvfrom(sockfd, (message\_type \*)buffer, MAX\_SIZE, 0, (struct sockaddr \*)&client\_addr, &len);

            std::pair<int, int> player\_info(client\_addr.sin\_addr.s\_addr, client\_addr.sin\_port);

            Player \*player = nullptr;

            auto player\_it = players.find(player\_info);

            if (buffer[0] == CLIENT\_CONNECT) {

                printf("Получено от клиента %s:%d: %s\n", inet\_ntoa(client\_addr.sin\_addr), ntohs(client\_addr.sin\_port), buffer);

                int player\_id;

                std::cout << "Size: " << players.size() << std::endl;

                if (players.size() >= MAX\_PLAYERS) {

                    player\_id = MAX\_PLAYERS;

                    std::cout << "Сервер заполнен" << std::endl;

                    sendto(sockfd, (int\*)&player\_id, 1, 0, (const struct sockaddr \*)&client\_addr, len);

                } else {

                    if (player\_it == players.end()) {

                        player\_id = CreatePlayer(client\_addr);

                        std::cout << "Send id " << player\_id << std::endl;

                        player\_it = players.begin();

                        while (player\_it->second.params.id != player\_id) ++player\_it;

                        struct ID\_MOVE\_INFO id\_move\_info = {

                            CLIENT\_CONNECT,

                            player\_it->second.params.position,

                            player\_it->second.params.direction,

                            player\_it->second.params.id

                        };

                        sendto(sockfd, (const struct ID\_MOVE\_INFO \*)&id\_move\_info, sizeof(struct ID\_MOVE\_INFO), 0, (const struct sockaddr \*)&(client\_addr), sizeof(client\_addr));

                        for (const auto& it : players) {

                            if (it.second.params.id == player\_id) continue;

                            sendto(sockfd, (const struct ID\_MOVE\_INFO \*)&id\_move\_info, sizeof(struct ID\_MOVE\_INFO), 0, (const struct sockaddr \*)&(it.second.client\_addr), sizeof(it.second.client\_addr));

                        }

                    } else {

                        player\_id = (\*player\_it).second.params.id;

                        std::cout << "Игрок уже существует" << std::endl;

                        sendto(sockfd, (int\*)&player\_id, 1, 0, (const struct sockaddr \*)&client\_addr, len);

                    }

                }

            } else {

                if (player\_it == players.end()) break;

                player = &player\_it->second;

                switch (buffer[0])

                {

                case CLIENT\_DISCONNECT:

                {

                    printf("Игрок #%d покинул игру.\n", player->params.id);

                    id[player->params.id] = true;

                    players.erase(player\_it);

                    break;

                }

                case FIRE:

                {

                    struct FIRE\_INFO\* fire\_info = reinterpret\_cast<FIRE\_INFO\*>(buffer);

                    gun.SetID(player->params.id);

                    gun.Teleport(fire\_info->position);

                    gun.args.deg = fire\_info->direction;

                    gun.SetID(player->params.id);

                    Bullet\* bullet = gun.Fire();

                    if (bullet != nullptr) {

                        bullets.push\_back(bullet);

                    }

                    // UnitInfo::UnitFire(fire\_info->position, fire\_info->direction, player->params.id);

                    struct ID\_FIRE\_INFO id\_fire\_info = {

                        FIRE,

                        player->params.position,

                        player->params.direction,

                        player->params.id

                    };

                    for (const auto& it : players) {

                        if (it.second.params.id == player->params.id) continue;

                        sendto(sockfd, (const struct ID\_FIRE\_INFO \*)&id\_fire\_info, sizeof(struct ID\_FIRE\_INFO), 0, (const struct sockaddr \*)&(it.second.client\_addr), sizeof(it.second.client\_addr));

                    }

                    break;

                }

                case MOVE:

                {

                    struct MOVE\_INFO\* move\_info = reinterpret\_cast<MOVE\_INFO\*>(buffer);

                    Vector3<float> offset = move\_info->position;

                    offset -= player->params.position;

                    player->params.position = move\_info->position;

                    player->params.direction = move\_info->direction;

                    struct ID\_MOVE\_INFO id\_move\_info = {

                        MOVE,

                        player->params.position,

                        player->params.direction,

                        player->params.id

                    };

                    for (const auto& it : players) {

                        if (it.second.params.id == player->params.id) continue;

                        sendto(sockfd, (const struct ID\_MOVE\_INFO \*)&id\_move\_info, sizeof(struct ID\_MOVE\_INFO), 0, (const struct sockaddr \*)&(it.second.client\_addr), sizeof(it.second.client\_addr));

                    }

                    break;

                }

                default:

                    break;

                }

            }

        }

    }

    void DealingDamage(Player& player\_, int damage) {

        player\_.params.HP -= damage;

        struct ID\_SET\_HP\_INFO id\_set\_hp\_info = {

            SET\_HP,

            player\_.params.HP,

            player\_.params.id

        };

        for (const auto& it : players) {

            sendto(sockfd, (const struct ID\_SET\_HP\_INFO \*)&id\_set\_hp\_info, sizeof(struct ID\_SET\_HP\_INFO), 0, (const struct sockaddr \*)&(it.second.client\_addr), sizeof(it.second.client\_addr));

        }

        if (player\_.params.HP <= 0) {

            auto it = players.begin();

            while (it->second.params.id != player\_.params.id) ++it;

            struct ID\_DISCONNECT id\_disconnect = {

                CLIENT\_DISCONNECT,

                player\_.params.id

            };

            sendto(sockfd, (const struct ID\_DISCONNECT \*)&id\_disconnect, sizeof(struct ID\_DISCONNECT), 0, (const struct sockaddr \*)&(it->second.client\_addr), sizeof(it->second.client\_addr));

            players.erase(it);

        }

    }

    void tick() {

        static const int targetFPS = 33;

        static const std::chrono::milliseconds frameDuration(1000 / targetFPS);

        static std::chrono::steady\_clock::time\_point framePrevious = std::chrono::steady\_clock::now();

        std::chrono::steady\_clock::time\_point now = std::chrono::steady\_clock::now();

        std::chrono::steady\_clock::time\_point frameEnd = now + frameDuration;

        std::this\_thread::sleep\_until(frameEnd);

        Time.Update();

        static float cut = Time.GetCurrentTime();

        static float prev\_time = Time.GetCurrentTime();

        Time.Update();

        cut = (Time.GetCurrentTime() - prev\_time) \* 1e-8;

        prev\_time = Time.GetCurrentTime();

        Actor::PushTime(Time.GetCurrentTime());

        // Коллизия пуль

        for (auto blt = bullets.begin(); blt != bullets.end();) {

            (\*blt)->MoveForward(cut \* (\*blt)->GetSpeed());

            if ((Time.GetCurrentTime() - (\*blt)->GetBirthTime()) > Bullet::lifetime) {

                blt = bullets.erase(blt);

                if (blt == bullets.end()) break;

            }

            for (auto& plr : players) {

                if ((plr.second.params.id != (\*blt)->id) && (plr.second.params.HP > 0)) {

                    if ((\*blt)->GetTransform()->WorldPos.Distance(plr.second.params.position) < (1.0 + 0.5)) {

                        std::cout << "DealingDamage" << std::endl;

                        DealingDamage(plr.second, BULLET\_DAMAGE);

                        blt = bullets.erase(blt);

                        break;

                    }

                }

            }

            if (blt == bullets.end()) break;

            ++blt;

        }

    }

private:

    std::map<std::pair<int, int>, Player> players;

    struct sockaddr\_in server\_addr;

    int sockfd, n;

    bool id[MAX\_PLAYERS];

    message\_type buffer[MAX\_SIZE];

    std::list<Bullet\*> bullets;

    GameTime Time;

};

Server.cpp  
#include <game/gameTime.hpp>

#include <server/server.hpp>

#include <thread>

int main(int argc, char\*\* argv)

{

    Server server;

    std::thread client\_callback([&]() {

        while(true) server.tick();

    });

    client\_callback.detach();

    server.callback();

    return 0;

}