ГУО “БГУИР”

Инженерно-экономический факультет

Кафедра экономической информатики

Отчёт по

Лабораторной работе № 3

**Создание параллельного многопоточного сервера с установлением логического соединения TCP**

Подготовил:

Студент гр.110101

Пантус Р. В.

Проверил:

Бутов А. А.

Минск 2023

**Цель работы:** изучить методы создания серверных приложений на основе установления логического соединения *TCP*, используя алгоритм многопоточной обработки запросов.

**Вариант 3:** На сервере хранится список о студентах. Каждая запись списка содержит следующую информацию о студенте:

* Ф.И.О. студента;
* номер группы;
* размер стипендии;
* оценки по N предметам.

Таких записей должно быть не менее 5-ти.

По запросу клиента он получает от сервера список только тех студентов, которые не имеют оценки 3.

Код клиентской части:

**Файл tcp\_client.h**

#pragma once

#include <cstring>

#include <json/value.h>

#include <sstream>

// socket programming

#include <utility>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

#include <arpa/inet.h>

// json files

#include <json/json.h>

#include <json/writer.h>

class Client {

public:

Client(const int& kPort\_ = 9984,

std::string kIpAddress\_ = "127.0.0.1") : kPort(kPort\_), kIpAddress(std::move(kIpAddress\_))

{

client\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (client\_fd < 0) {

throw std::runtime\_error("Cant create socket descriptor.");

}

server\_address.sin\_family = AF\_INET; // IPv4

server\_address.sin\_port = htons(kPort);

if (inet\_pton(AF\_INET, kIpAddress.c\_str(), &server\_address.sin\_addr) <= 0) {

throw std::runtime\_error("Cant convert IP address from text to binary form");

}

if (connect(client\_fd, (sockaddr\*)&server\_address, sizeof(server\_address)) < 0) {

throw std::runtime\_error("Cant connect to the server with ip: " + kIpAddress);

}

}

auto convertJsonString(const Json::Value& json) -> std::string {

Json::StreamWriterBuilder builder;

return Json::writeString(builder, json);

}

template<class T>

auto convertStringJson(const T& str) const -> Json::Value{

Json::Value fmt\_str;

std::stringstream input(str);

input >> fmt\_str;

return fmt\_str;

}

auto getListOfStudent(int needed\_mark) -> Json::Value {

// just need to get reponse from server

Json::Value prepared\_msg;

prepared\_msg["mark"] = needed\_mark;

// send block

const auto msg\_to\_sent{convertJsonString(prepared\_msg)};

send(client\_fd, msg\_to\_sent.data(), msg\_to\_sent.size(), 0);

// read block

std::memset(buffer, '\0', sizeof(buffer));

read(client\_fd, buffer, sizeof(buffer)); // unformated json list

return convertStringJson(buffer)["students"];

}

private:

const int kPort = 0;

char buffer[2049];

const std::string kIpAddress;

sockaddr\_in server\_address{}; // server information

int client\_fd = 0; // socket information

};

**Файл main.h**

#include <iostream>

#include "tcp\_client.h"

void ShowStudent(const Json::Value& json\_fmt\_student) {

const char\* name = "name",

\* group\_number = "group\_number";

const std::vector<const char\*> subjects = {"Math", "Physics", "Biology"};

std::cout << name << ": " << json\_fmt\_student[name] << '\n';

std::cout << group\_number << ": " << json\_fmt\_student[group\_number] << '\n';

std::cout << "Subject Marks: " << '\n';

for (int i = 0; i < (int)subjects.size(); ++i) {

std::cout << subjects[i] << ": "

<< json\_fmt\_student["subjects"][i][subjects[i]] << '\n';

}

}

int main() {

const std::string& kMessage = "Input [1 - 10] -> get list of students without this mark\nelse -> quit";

const std::string& kInputMsg = "Input: ";

Client client;

std::cout << kMessage << std::endl;

while (true) {

std::cout << std::endl << kInputMsg;

std::string request;

std::getline(std::cin, request);

std::stringstream input(request);

int parsed\_request = 0;

input >> parsed\_request;

char c = 0;

if (parsed\_request > 10 || parsed\_request < 1 || input.fail() || input.get(c)) {

std::cout << "Bye!" << std::endl;

break;

} else {

try {

auto response = client.getListOfStudent(parsed\_request);

if (response.isNull()) {

std::cout << "no students in the database!" << std::endl;

} else {

for (int i = 0; i < (int)response.size(); ++i) {

ShowStudent(response[i]);

if (i + 1 != (int)response.size()) {

std::cout << '\n';

}

}

}

} catch (std::runtime\_error& e) {

std::cout << e.what() << std::endl;

} catch (Json::RuntimeError& e) {

std::cout << e.what() << std::endl;

}

}

}

return 0;

}

Код Серверной части:

#pragma once

#include <unordered\_map>

// multithreading

#include <mutex>

#include <thread>

// custom files

#include "students.h"

#include "log.h"

#include "thread\_info.h"

// socket programing

#include <sys/socket.h>

#include <netinet/in.h>

#include <thread>

#include <unistd.h>

#include <arpa/inet.h>

// json files

#include <json/json.h>

#include <json/writer.h>

class Server {

public:

Server() {

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

if (socket\_fd < 0) {

throw std::runtime\_error("Cant create socket descriptor.");

}

// setup server information

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

server\_address.sin\_addr.s\_addr = INADDR\_ANY;

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

// connect socket with port

if (bind(socket\_fd, reinterpret\_cast<sockaddr\*>(&server\_address), sizeof(server\_address)) < 0) {

throw std::runtime\_error("Cant connect socket with port.");

}

// put server in a passive mode

if (listen(socket\_fd, MAX\_QUERIES) < 0) {

throw std::runtime\_error("Cant listen to socket");

}

log.ShowMessage("\nListening! Port:", PORT, "IpAddress: localhost");

students.readFromFile();

}

~Server() {

close(socket\_fd);

}

template <class T>

auto convertStringJson(const T& str) const -> Json::Value {

Json::Value fmt\_str;

std::stringstream input(str);

input >> fmt\_str;

return fmt\_str;

}

auto convertJsonString(const Json::Value& json) -> std::string {

Json::StreamWriterBuilder builder;

return Json::writeString(builder, json);

}

Json::Value ReadJson(const int& new\_socket, const std::string& client\_readable\_ip) {

size\_t buf\_size = sizeof(buffer);

std::memset(buffer, '\0', buf\_size);

if (!read(new\_socket, buffer, buf\_size)) {

throw std::runtime\_error("connection lost with: " + client\_readable\_ip);

}

return convertStringJson(buffer);

}

auto getClientIp() -> std::string {

std::string client\_readable\_ip(INET\_ADDRSTRLEN, '\0');

inet\_ntop(server\_address.sin\_family,

(sockaddr\*)&server\_address.sin\_addr,

client\_readable\_ip.data(),

client\_readable\_ip.size());

while (!client\_readable\_ip.empty() && client\_readable\_ip.back() == '\0') {

client\_readable\_ip.pop\_back();

}

return client\_readable\_ip;

}

void ProcessNewClientConnection(int new\_socket, int port) {

auto client\_readable\_ip = getClientIp();

log.ShowMessage("Connection established with: " + client\_readable\_ip);

{

std::lock\_guard<std::mutex> lock(connects\_mut);

std::thread::id id = std::this\_thread::get\_id();

connections\_info[id] = ThreadInfo(id, port);

}

while (true) {

try {

auto received\_mark = ReadJson(new\_socket, client\_readable\_ip);

Json::Value response;

{

// TODO:

const std::lock\_guard<std::mutex> lock(students\_mut);

response = students.findWithoutMark(received\_mark["mark"].asInt());

}

log.ShowMessage(Log::kReqSep, Log::kRequestMsg, received\_mark.toStyledString(),

Log::kLineSep, Log::kResponseMsg, response.toStyledString(), Log::kReqSep);

auto rsp\_msg = convertJsonString(response);

send(new\_socket, rsp\_msg.data(), rsp\_msg.size(), 0);

} catch (std::runtime\_error& e) {

// connection lost

log.ShowMessage(e.what());

break;

}

}

{

std::lock\_guard<std::mutex> lock(connects\_mut);

connections\_info.erase(std::this\_thread::get\_id());

}

}

[[noreturn]] void ServerCommunication() {

while (true) {

int new\_socket = accept(socket\_fd, (sockaddr\*)(&server\_address), (socklen\_t\*)&address\_length);

if (new\_socket < 0) {

close(new\_socket);

throw std::runtime\_error("Error while Accepting on socket");

}

std::thread new\_client\_connection(&Server::ProcessNewClientConnection, this, new\_socket, htons(server\_address.sin\_port));

new\_client\_connection.detach();

}

}

auto ShowMenu(const auto& menu) -> std::string {

for (auto item : menu) {

std::cout << item << std::endl;

}

std::string choice;

std::cout << "Make a choice: ";

std::getline(std::cin, choice);

return choice;

}

void ShowAppearance() {

const std::vector<const char \*> menu = {

"1 - Show current students",

"2 - Add one more student",

"3 - Remove last student",

"4 - Show active clients", // TODO:

"else - exit"

};

while (true) {

system("clear");

auto choice = ShowMenu(menu);

if (choice.size() == 1 && 0 < choice[0] - '0' && choice[0] - '0' <= 3) {

const std::lock\_guard<std::mutex> lock(students\_mut);

switch(choice[0] - '0') {

case 1: {

int index = 0;

students.visitor([&]<class T>(T& item) {

std::cout << '\n' << ++index << ".\n" << item << '\n';

});

break;

}

case 2: {

students.addStudent(students.createStudent());

break;

}

case 3: {

students.removeLastStudent();

break;

}

}

} else if (choice == "4") {

const std::lock\_guard<std::mutex> lock(connects\_mut);

if (connections\_info.empty()) {

std::cout << "No connections are available" << std::endl;

}

for (auto& [id, info] : connections\_info) {

info.showInfo();

}

} else {

break;

}

std::getline(std::cin, choice); // ~ system("pause");

}

}

void run() {

std::thread appearance\_thread(&Server::ShowAppearance, this);

std::thread server\_communication\_thread(&Server::ServerCommunication, this);

appearance\_thread.join();

server\_communication\_thread.join();

}

private: // required for messages

Log log;

private: // variables

const int PORT = 9984;

const int MAX\_QUERIES = 3;

sockaddr\_in server\_address{}; // server information

const int address\_length = sizeof(server\_address);

int socket\_fd = 0; // file descriptor

std::mutex students\_mut; // mutex for students

std::mutex connects\_mut; // mutex for connections

std::unordered\_map<std::thread::id, ThreadInfo> connections\_info;

Students students;

char buffer[50]; // 50 -> size of a buffer

};