Московский авиационный институт

(национальный исследовательский университет)

# Факультет информационных технологий и прикладной математики

**Кафедра вычислительной математики и программирования**

## Лабораторная работа № 5-7 по курсу «Операционные системы»

**Реализация обмена информацией между процессами посредством очереди сообщений в операционной системе «UNIX». Реализация распределённой системы вычислений в операционной системе «UNIX». Организация отложенных вычислений в операционной системе «UNIX».**

Студент: Арусланов К. А. Преподаватель: Миронов Е. С.

Группа: М8О-203Б-23

Дата: Оценка: Подпись:

**Москва, 2024**

# Условие

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вы- числительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при по- мощи технологии очередей сообщений. Также в данной системе необходимо предусмот- реть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступны- ми, но родительские узлы должны сохранить свою работоспособность. Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислитель- ные узлы.

# Задание

Топология – список, один управляющий узел; команда проверки доступности – heartbeat; команда для вычислительных узлов – поиск подстроки в строке.

# Метод решения

В данной задаче один управляющий узел (Controller) принимал команды (create, exec, heartbeat, quit) от пользователя и распределял их среди вычислительных узлов (Worker), связываясь через ZeroMQ (паттерн ROUTER/DEALER). При create создавался новый процесс-воркер, при exec воркер выполнял поиск подстроки, при heartbeat воркер посылал сигналы «HB». Программа была спроектирована так, чтобы асинхронность достигалась с помощью сокетов ZeroMQ и потоков (pthread), а топология узлов (родитель—потомок) учитывалась при проверке доступности.

# Код программы

common.h

#ifndef COMMON\_H

#define COMMON\_H

#include <string>

#include <map>

#include <vector>

#include <chrono>

#include <pthread.h>

struct TNodeInfo {

int id;

int parent\_id;

std::string endpoint;

bool alive = true;

bool hb\_received = false;

std::chrono::steady\_clock::time\_point last\_heartbeat;

};

// Глобальный мьютекс для вывода

extern pthread\_mutex\_t g\_print\_mutex;

#endif

tcontroller.h

#ifndef TCONTROLLER\_H

#define TCONTROLLER\_H

#include "ttopology.h"

#include "tmessaging.h"

#include <string>

#include <chrono>

#include <atomic>

class TControllerNode {

private:

    void HandleCreate(int id, int parent);

    void HandleExec(int id);

    void HandleHeartbeat(int time\_ms);

    static void\* ReceiverThreadStatic(void\* arg);

    void ReceiverThreadFunc();

    void Quit();

    bool IsNodeAvailable(int id);

    void CheckHeartbeats();

    TTopology topology\_;

    TMessaging messaging\_;

    int heartbeat\_time\_ = 0;

    std::chrono::steady\_clock::time\_point last\_heartbeat\_check\_ = std::chrono::steady\_clock::now();

    std::atomic<bool> running\_{true};

    pthread\_t recv\_thread\_;

public:

    TControllerNode();

    bool Init(const std::string& endpoint);

    void Run();

};

#endif

tmessaging.h

#ifndef TMESSAGING\_H

#define TMESSAGING\_H

#include <string>

#include <zmq.hpp>

class TMessaging {

private:

    zmq::context\_t context\_{1};

    zmq::socket\_t socket\_{context\_, ZMQ\_ROUTER};

    bool is\_controller\_ = false;

    int self\_node\_id\_ = -1;

public:

    bool InitController(const std::string& endpoint);

    bool InitWorker(const std::string& controller\_endpoint, int node\_id);

    bool SendToWorker(int node\_id, const std::string& message);

    bool SendToController(const std::string& message);

    bool RecvFromAnyWorker(std::string& node\_id\_str, std::string& message);

    bool RecvFromController(std::string& message);

    bool IsController() const { return is\_controller\_; }

};

#endif

tsearch.h

#ifndef TSEARCH\_H

#define TSEARCH\_H

#include <string>

#include <vector>

class TSearch {

public:

    static std::vector<int> BoyerMooreSearch(const std::string& text, const std::string& pattern);

};

#endif

ttopology.h

#ifndef TTOPOLOGY\_H

#define TTOPOLOGY\_H

#include "common.h"

class TTopology {

private:

    std::map<int, TNodeInfo> nodes\_;

public:

    bool AddNode(int id, int parent\_id, const std::string& endpoint);

    bool RemoveNode(int id);

    TNodeInfo\* GetNode(int id);

    bool NodeExists(int id);

    std::vector<int> GetChildren(int id);

    std::map<int, TNodeInfo>& GetAllNodes() { return nodes\_; }

};

#endif

tworker.h

#ifndef TWORKER\_H

#define TWORKER\_H

#include <string>

#include <atomic>

#include "tmessaging.h"

class TWorkerNode {

private:

    int id\_;

    int parent\_id\_;

    std::string endpoint\_;

    TMessaging messaging\_;

    std::atomic<bool> running\_{true};

    void HandleExec(const std::string& text, const std::string& pattern);

    void SendHeartbeat();

    static void\* HeartbeatThreadStatic(void\* arg);

public:

    TWorkerNode(int id, int parent\_id, const std::string& endpoint);

    bool Init();

    void Run();

};

#endif

tcontroller.cpp

#include "tcontroller.h"

#include <iostream>

#include <sstream>

#include <pthread.h>

#include <chrono>

#include <unistd.h> // execl, fork

TControllerNode::TControllerNode() {}

bool TControllerNode::Init(const std::string& endpoint) {

    if (!messaging\_.InitController(endpoint)) {

        std::cerr << "Controller: Failed to init messaging.\n";

        return false;

    }

    return true;

}

void TControllerNode::Run() {

    std::cout << "Available commands:\n";

    std::cout << "  create <id> [parent]\n";

    std::cout << "  exec <id>\n";

    std::cout << "  heartbeat <time\_ms>\n";

    std::cout << "  quit\n";

    // Поток приёма сообщений

    pthread\_create(&recv\_thread\_, NULL, &TControllerNode::ReceiverThreadStatic, this);

    pthread\_detach(recv\_thread\_);

    while (running\_) {

        std::cout << "> ";

        std::string line;

        if (!std::getline(std::cin, line)) break;

        if (line.empty()) continue;

        std::istringstream iss(line);

        std::string cmd;

        iss >> cmd;

        if (cmd == "create") {

            std::string id\_str, parent\_str;

            id\_str = "";

            parent\_str = "-1";

            iss >> id\_str; // id

            if (iss.good()) iss >> parent\_str;

            int id, parent;

            try {

                if (id\_str.empty()) {

                    std::cout << "Error: invalid arguments for create\n";

                    continue;

                }

                id = std::stoi(id\_str);

                parent = std::stoi(parent\_str);

            } catch (...) {

                std::cout << "Error: invalid arguments for create\n";

                continue;

            }

            HandleCreate(id, parent);

        } else if (cmd == "exec") {

            std::string id\_str;

            iss >> id\_str;

            if (id\_str.empty()) {

                std::cout << "Error: invalid id for exec\n";

                continue;

            }

            int id;

            try {

                id = std::stoi(id\_str);

            } catch (...) {

                std::cout << "Error: invalid id for exec\n";

                continue;

            }

            HandleExec(id);

        } else if (cmd == "heartbeat") {

            std::string t\_str;

            iss >> t\_str;

            if (t\_str.empty()) {

                std::cout << "Error: invalid time for heartbeat\n";

                continue;

            }

            int t;

            try {

                t = std::stoi(t\_str);

            } catch (...) {

                std::cout << "Error: invalid time for heartbeat\n";

                continue;

            }

            HandleHeartbeat(t);

        } else if (cmd == "quit") {

            Quit();

        } else {

            std::cout << "Unknown command\n";

        }

        CheckHeartbeats();

    }

    // Дадим время всем завершиться

    usleep(500\*1000); // 0.5 секунды

}

void\* TControllerNode::ReceiverThreadStatic(void\* arg) {

    TControllerNode\* self = static\_cast<TControllerNode\*>(arg);

    self->ReceiverThreadFunc();

    return NULL;

}

void TControllerNode::ReceiverThreadFunc() {

    while (running\_) {

        std::string node\_id\_str;

        std::string msg;

        if (!messaging\_.RecvFromAnyWorker(node\_id\_str, msg)) {

            // Нет сообщений, подождём

            usleep(100 \* 1000); // задержка на 100 ms

            continue;

        }

        int nid;

        try {

            nid = std::stoi(node\_id\_str);

        } catch (...) {

            std::cerr << "Controller: Invalid node\_id\_str received: " << node\_id\_str << "\n";

            std::cerr << msg << "\n"; // Выведем полученное сообщение

            continue;

        }

        if (msg == "HB") {

            // Heartbeat signal

            TNodeInfo\* node = topology\_.GetNode(nid);

            if (node) {

                node->last\_heartbeat = std::chrono::steady\_clock::now();

                node->hb\_received = true;

                if (!node->alive) {

                    // Узел снова послал HB, можно считать его живым

                    node->alive = true;

                }

            } else {

                // Неизвестный узел

                std::cerr << "Controller: Received HB from unknown node " << nid << "\n";

            }

        } else if (msg == "Ok" || msg.rfind("Ok:",0)==0 || msg.rfind("Error:",0)==0) {

            // Ответ на команду exec или другую команду

            std::cout << msg << "\n";

        } else {

            // Неизвестное сообщение

            std::cout << "Unknown message from node " << nid << ": " << msg << "\n";

        }

    }

}

void TControllerNode::HandleCreate(int id, int parent) {

    if (topology\_.NodeExists(id)) {

        std::cout << "Error: Already exists\n";

        return;

    }

    if (parent != -1 && !topology\_.NodeExists(parent)) {

        std::cout << "Error: Parent not found\n";

        return;

    }

    if (parent != -1 && !IsNodeAvailable(parent)) {

        std::cout << "Error: Parent is unavailable\n";

        return;

    }

    const char\* workerPath = std::getenv("WORKER\_PATH");

    if (!workerPath) {

        std::cout << "Error: WORKER\_PATH environment variable is not set\n";

        return;

    }

    std::string endpoint = "tcp://127.0.0.1:" + std::to\_string(6000 + id);

    pid\_t pid = fork();

    if (pid == 0) {

        execl(workerPath, "worker", std::to\_string(id).c\_str(), std::to\_string(parent).c\_str(), endpoint.c\_str(), (char\*)NULL);

        std::cerr << "Error: Failed to exec worker from WORKER\_PATH\n";

        \_exit(1);

    } else if (pid < 0) {

        std::cout << "Error: Failed to create process\n";

        return;

    }

    if (!topology\_.AddNode(id, parent, endpoint)) {

        std::cout << "Error: Internal AddNode failed\n";

        return;

    }

    std::cout << "Ok: " << pid << "\n";

}

void TControllerNode::HandleExec(int id) {

    if (!topology\_.NodeExists(id)) {

        std::cout << "Error:" << id << ": Not found\n";

        return;

    }

    if (!IsNodeAvailable(id)) {

        std::cout << "Error:" << id << ": Node is unavailable\n";

        return;

    }

    std::cout << "(text\_string): ";

    std::string text;

    if (!std::getline(std::cin, text) || text.empty()) std::getline(std::cin, text);

    std::cout << "(pattern\_string): ";

    std::string pattern;

    if (!std::getline(std::cin, pattern) || pattern.empty()) std::getline(std::cin, pattern);

    std::string msg = "EXEC\n" + text + "\n" + pattern;

    if(!messaging\_.SendToWorker(id, msg)) {

        std::cout << "Error:" << id << ": Failed to send message\n";

    }

}

void TControllerNode::HandleHeartbeat(int time\_ms) {

    heartbeat\_time\_ = time\_ms;

    std::cout << "Ok\n";

    // Посылаем всем узлам "HB\_START"

    for (auto& [nid, info] : topology\_.GetAllNodes()) {

        messaging\_.SendToWorker(nid, "HB\_START");

    }

}

void TControllerNode::Quit() {

    running\_ = false;

    std::cout << "Exiting...\n";

    // Посылаем QUIT всем узлам, чтобы они завершились

    for (auto& [nid, info] : topology\_.GetAllNodes()) {

        messaging\_.SendToWorker(nid, "QUIT");

    }

}

bool TControllerNode::IsNodeAvailable(int id) {

    TNodeInfo\* node = topology\_.GetNode(id);

    if (!node) return false;

    if (!node->alive) return false;

    if (heartbeat\_time\_ > 0) {

        auto now = std::chrono::steady\_clock::now();

        auto diff = std::chrono::duration\_cast<std::chrono::milliseconds>(now - node->last\_heartbeat).count();

        if (diff > 4 \* heartbeat\_time\_) {

            return false;

        }

    }

    return true;

}

void TControllerNode::CheckHeartbeats() {

    if (heartbeat\_time\_ <= 0) return;

    auto now = std::chrono::steady\_clock::now();

    for (auto& [nid, info] : topology\_.GetAllNodes()) {

        if (!info.hb\_received) continue; // Для недавно активированных

        auto diff = std::chrono::duration\_cast<std::chrono::milliseconds>(now - info.last\_heartbeat).count();

        if (diff > 4 \* heartbeat\_time\_ && info.alive) {

            info.alive = false;

            std::cout << "Heartbeat: node " << nid << " is unavailable now\n";

        }

    }

}

tmessaging.cpp

#include "tmessaging.h"

#include <iostream>

bool TMessaging::InitController(const std::string& endpoint) {

    is\_controller\_ = true;

    socket\_ = zmq::socket\_t(context\_, ZMQ\_ROUTER);

    try {

        socket\_.bind(endpoint.c\_str());

    } catch (const zmq::error\_t& e) {

        std::cerr << "Messaging: Failed to bind: " << e.what() << "\n";

        return false;

    }

    return true;

}

bool TMessaging::InitWorker(const std::string& controller\_endpoint, int node\_id) {

    is\_controller\_ = false;

    self\_node\_id\_ = node\_id;

    socket\_ = zmq::socket\_t(context\_, ZMQ\_DEALER);

    std::string identity = std::to\_string(node\_id);

    socket\_.setsockopt(ZMQ\_IDENTITY, identity.c\_str(), identity.size());

    try {

        socket\_.connect(controller\_endpoint.c\_str());

    } catch (const zmq::error\_t& e) {

        std::cerr << "Messaging: Failed to connect worker: " << e.what() << "\n";

        return false;

    }

    return true;

}

bool TMessaging::SendToWorker(int node\_id, const std::string& message) {

    if (!is\_controller\_) return false;

    zmq::message\_t id\_msg(std::to\_string(node\_id).data(), std::to\_string(node\_id).size());

    // zmq::message\_t empty\_msg;

    zmq::message\_t msg(message.data(), message.size());

    try {

        socket\_.send(id\_msg, ZMQ\_SNDMORE);

        // socket\_.send(empty\_msg, ZMQ\_SNDMORE);

        socket\_.send(msg);

    } catch (const zmq::error\_t& e) {

        std::cerr << "Messaging: SendToWorker error: " << e.what() << "\n";

        return false;

    }

    return true;

}

bool TMessaging::SendToController(const std::string& message) {

    if (is\_controller\_) return false;

    zmq::message\_t msg(message.data(), message.size());

    try {

        socket\_.send(msg);

    } catch (const zmq::error\_t& e) {

        std::cerr << "Messaging: SendToController error: " << e.what() << "\n";

        return false;

    }

    return true;

}

bool TMessaging::RecvFromAnyWorker(std::string& node\_id\_str, std::string& message) {

    if (!is\_controller\_) return false;

    zmq::message\_t id\_msg;

    // zmq::message\_t empty\_msg;

    zmq::message\_t msg;

    try {

        if (!socket\_.recv(&id\_msg)) return false;

        // if (!socket\_.recv(&empty\_msg)) return false;

        if (!socket\_.recv(&msg)) return false;

    } catch (const zmq::error\_t& e) {

        std::cerr << "Messaging: RecvFromAnyWorker error: " << e.what() << "\n";

        return false;

    }

    node\_id\_str = std::string((char\*)id\_msg.data(), id\_msg.size());

    message = std::string((char\*)msg.data(), msg.size());

    return true;

}

bool TMessaging::RecvFromController(std::string& message) {

    if (is\_controller\_) return false;

    zmq::message\_t msg;

    try {

        if (!socket\_.recv(&msg)) return false;

    } catch (const zmq::error\_t& e) {

        std::cerr << "Messaging: RecvFromController error: " << e.what() << "\n";

        return false;

    }

    message = std::string((char\*)msg.data(), msg.size());

    return true;

}

tsearch.cpp

#include "tsearch.h"

#include <vector>

#include <string>

#include <unordered\_map>

std::vector<int> TSearch::BoyerMooreSearch(const std::string& text, const std::string& pattern) {

    std::vector<int> result;

    int n = (int)text.size();

    int m = (int)pattern.size();

    if (m == 0) {

        for (int i = 0; i < n; i++) result.push\_back(i);

        return result;

    }

    if (m > n) {

        result.push\_back(-1);

        return result;

    }

    // Построение таблицы смещений badChar

    std::vector<int> badChar(256, -1);

    for (int i = 0; i < m; i++) {

        badChar[(unsigned char)pattern[i]] = i;

    }

    bool found = false;

    int s = 0;

    while (s <= (n - m)) {

        int j = m - 1;

        while (j >= 0 && pattern[j] == text[s+j]) {

            j--;

        }

        if (j < 0) {

            result.push\_back(s);

            found = true;

            s += (s + m < n) ? m - badChar[(unsigned char)text[s+m]] : 1;

        } else {

            int shift = j - badChar[(unsigned char)text[s+j]];

            if (shift < 1) shift = 1;

            s += shift;

        }

    }

    if (!found) result.push\_back(-1);

    return result;

}

ttopology.cpp

#include "ttopology.h"

#include <algorithm>

bool TTopology::AddNode(int id, int parent\_id, const std::string& endpoint) {

    if (nodes\_.find(id) != nodes\_.end()) return false;

    TNodeInfo info;

    info.id = id;

    info.parent\_id = parent\_id;

    info.endpoint = endpoint;

    info.alive = true;

    info.last\_heartbeat = std::chrono::steady\_clock::now();

    nodes\_[id] = info;

    return true;

}

bool TTopology::RemoveNode(int id) {

    return nodes\_.erase(id) > 0;

}

TNodeInfo\* TTopology::GetNode(int id) {

    auto it = nodes\_.find(id);

    if (it == nodes\_.end()) return nullptr;

    return &it->second;

}

bool TTopology::NodeExists(int id) {

    return nodes\_.find(id) != nodes\_.end();

}

std::vector<int> TTopology::GetChildren(int id) {

    std::vector<int> children;

    for (auto& [nid, info] : nodes\_) {

        if (info.parent\_id == id) children.push\_back(nid);

    }

    return children;

}

tworker.cpp

#include "tworker.h"

#include "tsearch.h"

#include "common.h"

#include <iostream>

#include <chrono>

#include <pthread.h>

pthread\_mutex\_t g\_print\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

TWorkerNode::TWorkerNode(int id, int parent\_id, const std::string& endpoint)

: id\_(id), parent\_id\_(parent\_id), endpoint\_(endpoint) {}

bool TWorkerNode::Init() {

    if (!messaging\_.InitWorker("tcp://127.0.0.1:5555", id\_)) {

        std::cerr << "Worker " << id\_ << ": Failed to init messaging\n";

        return false;

    }

    return true;

}

void\* TWorkerNode::HeartbeatThreadStatic(void\* arg) {

    TWorkerNode\* self = static\_cast<TWorkerNode\*>(arg);

    while (self->running\_) {

        self->SendHeartbeat();

        usleep(2000 \* 1000); // 2000 ms = 2 s

    }

    return NULL;

}

void TWorkerNode::Run() {

    bool hb\_started = false;

    pthread\_t hb\_thread;

    while (running\_) {

        std::string msg;

        if (!messaging\_.RecvFromController(msg)) {

            usleep(100 \* 1000); // 100 ms задержка

            continue;

        }

        if (msg == "HB\_START") {

            if (!hb\_started) {

                hb\_started = true;

                pthread\_create(&hb\_thread, NULL, &HeartbeatThreadStatic, this);

                pthread\_detach(hb\_thread);

            }

            continue;

        }

        if (msg == "QUIT") {

            // Завершаем работу узла

            running\_ = false;

            pthread\_mutex\_lock(&g\_print\_mutex);

            std::cerr << id\_ << " quitting...\n";

            pthread\_mutex\_unlock(&g\_print\_mutex);

            break;

        }

        if (msg.rfind("EXEC", 0) == 0) {

            size\_t pos1 = msg.find('\n');

            size\_t pos2 = msg.find('\n', pos1+1);

            if (pos1 == std::string::npos || pos2 == std::string::npos) {

                messaging\_.SendToController("Error:" + std::to\_string(id\_) + ": Invalid EXEC format");

                continue;

            }

            std::string text = msg.substr(pos1+1, pos2 - (pos1+1));

            std::string pattern = msg.substr(pos2+1);

            HandleExec(text, pattern);

        } else {

            messaging\_.SendToController("Error:" + std::to\_string(id\_) + ": Unknown command");

        }

    }

}

void TWorkerNode::HandleExec(const std::string& text, const std::string& pattern) {

    auto positions = TSearch::BoyerMooreSearch(text, pattern);

    std::string result;

    if (positions.size() == 1 && positions[0] == -1) {

        result = "Ok:" + std::to\_string(id\_) + ": -1";

    } else {

        result = "Ok:" + std::to\_string(id\_) + ":";

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

            if (i > 0) result += ";";

            result += std::to\_string(positions[i]);

        }

    }

    messaging\_.SendToController(result);

}

void TWorkerNode::SendHeartbeat() {

    messaging\_.SendToController("HB");

}

main\_controller.cpp

#include "tcontroller.h"

#include <iostream>

int main() {

    TControllerNode controller;

    if (!controller.Init("tcp://127.0.0.1:5555")) {

        std::cerr << "Error: Failed to init controller\n";

        return 1;

    }

    controller.Run();

    return 0;

}

main\_worker.cpp

#include "tworker.h"

#include <iostream>

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

    if (argc < 4) {

        std::cerr << "Usage: worker id parent endpoint\n";

        return 1;

    }

    int id = std::stoi(argv[1]);

    int parent = std::stoi(argv[2]);

    std::string endpoint = argv[3];

    TWorkerNode worker(id, parent, endpoint);

    if (!worker.Init()) {

        std::cerr << "Error: Failed to init worker\n";

        return 1;

    }

    worker.Run();

    return 0;

}

CMakeLists.txt

find\_package(PkgConfig REQUIRED)

pkg\_check\_modules(ZMQ REQUIRED libzmq)

add\_executable(lab5-7\_controller

    main\_controller.cpp

    src/tcontroller.cpp

    src/tmessaging.cpp

    src/tsearch.cpp

    src/ttopology.cpp

)

target\_link\_libraries(lab5-7\_controller PRIVATE zmq)

target\_include\_directories(lab5-7\_controller PRIVATE include)

add\_executable(lab5-7\_worker

    main\_worker.cpp

    src/tworker.cpp

    src/tmessaging.cpp

    src/tsearch.cpp

    src/ttopology.cpp

)

target\_link\_libraries(lab5-7\_worker PRIVATE zmq)

target\_include\_directories(lab5-7\_worker PRIVATE include)

lab5-7\_test.cpp

#include <gtest/gtest.h>

#include "tsearch.h"

#include "ttopology.h"

#include "tmessaging.h"

#include <atomic>

#include <string>

#include <unistd.h>

#include <pthread.h>

TEST(TSearchTest, EmptyPattern) {

    std::string text = "abracadabra";

    std::string pattern = "";

    auto positions = TSearch::BoyerMooreSearch(text, pattern);

    // Пустой паттерн – вхождение в каждую позицию

    ASSERT\_EQ((int)positions.size(), (int)text.size());

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

        EXPECT\_EQ(positions[i], i);

    }

}

TEST(TSearchTest, NotFound) {

    std::string text = "abracadabra";

    std::string pattern = "zzz";

    auto positions = TSearch::BoyerMooreSearch(text, pattern);

    ASSERT\_EQ((int)positions.size(), 1);

    EXPECT\_EQ(positions[0], -1);

}

TEST(TSearchTest, SimpleFound) {

    std::string text = "abracadabra";

    std::string pattern = "abra";

    auto positions = TSearch::BoyerMooreSearch(text, pattern);

    // Ожидаем вхождения: в позициях 0 и 7

    ASSERT\_EQ((int)positions.size(), 2);

    EXPECT\_EQ(positions[0], 0);

    EXPECT\_EQ(positions[1], 7);

}

TEST(TSearchTest, MultipleOverlapFound) {

    std::string text = "aaaaa";

    std::string pattern = "aa";

    auto positions = TSearch::BoyerMooreSearch(text, pattern);

    // Вхождения: 0,1,2,3

    ASSERT\_EQ((int)positions.size(), 4);

    EXPECT\_EQ(positions[0], 0);

    EXPECT\_EQ(positions[1], 1);

    EXPECT\_EQ(positions[2], 2);

    EXPECT\_EQ(positions[3], 3);

}

TEST(TTopologyTest, AddNode) {

    TTopology topo;

    EXPECT\_FALSE(topo.NodeExists(10));

    EXPECT\_TRUE(topo.AddNode(10, -1, "endpoint10"));

    EXPECT\_TRUE(topo.NodeExists(10));

    TNodeInfo\* info = topo.GetNode(10);

    ASSERT\_NE(info, nullptr);

    EXPECT\_EQ(info->id, 10);

    EXPECT\_EQ(info->parent\_id, -1);

    EXPECT\_EQ(info->endpoint, "endpoint10");

    EXPECT\_TRUE(info->alive);

}

TEST(TTopologyTest, AddDuplicateNode) {

    TTopology topo;

    EXPECT\_TRUE(topo.AddNode(10, -1, "endpoint10"));

    EXPECT\_FALSE(topo.AddNode(10, -1, "endpoint10\_bis")); // Уже существует

}

TEST(TTopologyTest, RemoveNode) {

    TTopology topo;

    topo.AddNode(10, -1, "end10");

    topo.AddNode(20, 10, "end20");

    EXPECT\_TRUE(topo.NodeExists(20));

    EXPECT\_TRUE(topo.RemoveNode(20));

    EXPECT\_FALSE(topo.NodeExists(20));

}

TEST(TTopologyTest, GetChildren) {

    TTopology topo;

    topo.AddNode(10, -1, "end10");

    topo.AddNode(20, 10, "end20");

    topo.AddNode(15, 10, "end15");

    topo.AddNode(12, -1, "end12");

    auto children10 = topo.GetChildren(10);

    ASSERT\_EQ((int)children10.size(), 2);

    EXPECT\_TRUE(std::find(children10.begin(), children10.end(), 20) != children10.end());

    EXPECT\_TRUE(std::find(children10.begin(), children10.end(), 15) != children10.end());

    auto childrenRoot = topo.GetChildren(-1);

    ASSERT\_EQ((int)childrenRoot.size(), 2); // 10 и 12

    EXPECT\_TRUE(std::find(childrenRoot.begin(), childrenRoot.end(), 10) != childrenRoot.end());

    EXPECT\_TRUE(std::find(childrenRoot.begin(), childrenRoot.end(), 12) != childrenRoot.end());

}

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

    testing::InitGoogleTest(&argc, argv);

    return RUN\_ALL\_TESTS();

}

# Выводы

Асинхронная архитектура с использованием ZeroMQ позволила масштабировать передачу сообщений, не блокируя управляющий узел при ожидании ответов от множества воркеров. Лабораторная работа показала, как распределённая система может быть организована через очереди сообщений и асинхронные сокеты, делая её гибкой и отказоустойчивой при сбоях отдельных узлов.