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: Кунавин К. В.
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 «вычислительный». 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.