Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №5-6-7 по курсу
«Операционные системы»

УПРАВЛЕНИЕ СЕРВЕРАМИ СООБЩЕНИЙ ПРИМЕНЕНИЕ ОТЛОЖЕННЫХ ВЫЧИСЛЕНИЙ ИНТЕГРАЦИЯ ПРОГРАММНЫХ СИСТЕМ ДРУГ С ДРУГОМ

Студент: Мирошников Дмитрий Евгеньевич
Группа: М8О–
210Б–22
Вариант: 11
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (№5)
- Применение отложенных вычислений (№6)
- Интеграция программных систем друг с другом (№7)

Задание

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 помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть 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 недоступными, но родительские узлы должны сохранить свою работоспособность.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

Создание нового вычислительного узла Формат команды:

create id [parent] id – целочисленный идентификатор нового вычислительного узла

parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода:

«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла

«Error: Already exists» - вычислительный узел с таким идентификатором уже существует

«Error: Parent not found» - нет такого родительского узла с таким идентификатором

«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться

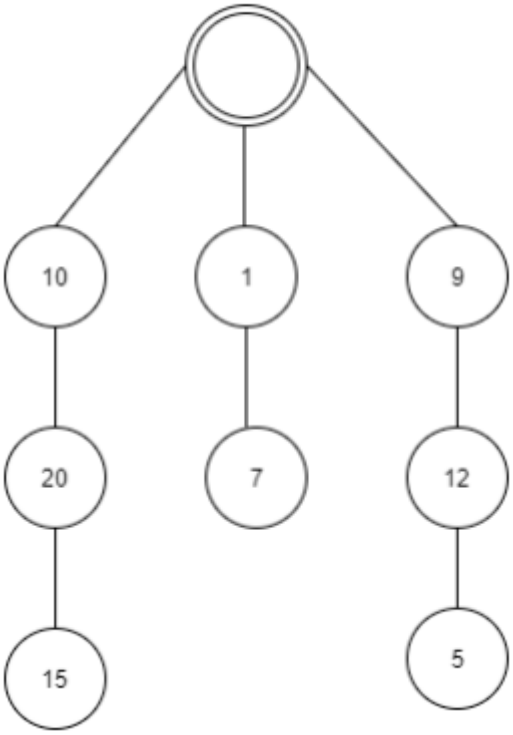
«Error: [Custom error]» - любая другая обрабатываемая ошибка

Пример:

> create 10 5

Ok: 3128

Топология: Все вычислительные узлы находятся в дереве общего вида



Набор команд: локальный таймер

Формат команды сохранения значения: exec id subcommand

subcommand – одна из трех команд: start, stop, time.

start – запустить таймер

stop – остановить таймер

time – показать время локального таймера в миллисекундах

Пример:

> exec 10 time

Ok:10: 0

>exec 10 start

Ok:10

>exec 10 start

Ok:10

прошло 10 секунд

> exec 10 time

Ok:10: 10000

прошло 2 секунды

>exec 10 stop

Ok:10

прошло 2 секунды

>exec 10 time

Ok:10: 12000

Команда проверки

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

Пример:

> pingall

Ok: -1 // Все узлы доступны

> pingall

Ok: 7;10;15 // узлы 7, 10, 15 — недоступны

Общие сведения о программе

Программа компилируется из файла controlNode.cpp. Также используется заголовочные файлы: iostream, map, zmq.hpp, unitstd.h, vector, cassert, и др. Для запросов и ответов написаны отедлые класс. Общение между процессами осуществляется с помощью передачи json-объектов.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Установить и изучить библиотеку zmq.hpp.
2. Написать 2 основных интерфейс файла

topology.h и myMQ.hpp для удобства реализации задачи.

3. Реализовать два файла для работы с сообщениями — это calulateNode.cpp и controlNode.cpp.
4. Написать правильный cmake файл, для правильно работы программы.
5. В controlNode реализовать простой интерфейс для упрощенный работы и тестировки программы.

Основные файлы программы

myMQ.hpp

```cpp
#pragma once

#include <zmq.hpp>
const int BASE_PORT = 5555;
std::string GetConPort(int id) {
    return "tcp://localhost:" + std::to_string(BASE_PORT + id);
}
std::string GetBindPort(int id) {
    return "tcp://*:" + std::to_string(BASE_PORT + id);
}
```

request.hpp

```cpp
#pragma once

#include <string>
#include <vector>
```

```cpp
enum Action {
    Time,
    Start,
    Stop,
    Ping,
    Create,
    Unknown
};
```

```cpp
struct Request {
    Action action;
    std::vector<int> path;
    int id;
    int depth;
    int timeToWait;

    Request() : action(Unknown) {}
    Request(const std::string& str) {
        if (str == "time") {
            action = Time;
        } else if (str == "start") {
            action = Start;
```

```
        } else if (str == "stop") {
            action = Stop;
        } else if (str == "ping") {
            action = Ping;
        } else {
            action = Unknown;
        }
    }
    Request(const Action& a, std::vector<int> _path, int _id): action(a), path(_path), id(_id)
{}
    Request(const std::string& str,const std::vector<int>& _path, int _id): Request(str) {
        path = _path;
        id = _id;
    }
```

```
    operator std::string() const {
        switch (action) {
            case Time:
                return "time";
            case Start:
                return "start";
            case Stop:
                return "stop";
            default:
                return "unknown";
        }
    }
```

```
    ~Request() = default;
};
```

response.hpp

```
#pragma once

#include <string>
#include <chrono>
```

```
enum Status {
    OK,
    ERROR
};
```

```
struct Response {
    Status status;
    int result;
    int pid;
    std::string error;
    std::vector<int> unavailable;
```

```
    Response(): status(OK) {}
    Response(Status _status) : status(_status) {}
    Response(const std::string& status) {
        if (status == "OK") {
            this->status = OK;
        } else {
            this->status = ERROR;
        }
    }
```

```
    operator std::string() const {
        if (status == OK) {
            return "OK";
        } else {
            return "ERROR";
        }
    }
```

```
    bool StatusOK() const {
        return status == OK;
    }
};
```

timer.hpp

```
#pragma once

#include <chrono>
#include <string>
```

```
std::chrono::system_clock::time_point start;
std::chrono::system_clock::time_point stop;
int countStopMilliseconds = -1;
bool stopped = false;
bool started = false;
```

```
typedef const char* Error;
```

```
std::string HandleStart();
```

```
Error StartTimer() {
    if (started && !stopped) {
        return "Timer is already running";
    }
    if (started && stopped) {
        auto now = std::chrono::system_clock::now();
        std::chrono::duration<double, std::milli> elapsed = now - stop;
        countStopMilliseconds += int(elapsed.count());
    }
    if (!started) {
        start = std::chrono::system_clock::now();
        countStopMilliseconds = -1;
    }
    stopped = false;
    started = true;
```

```
    return nullptr;
}
```

```
Error StopTimer() {
    if (!started) {
        return "Timer has not been starter yet";
    }
```

```
    if (stopped) {
        return "Timer is already stopped";
    }
```

```
    if (countStopMilliseconds == -1) {
        countStopMilliseconds = 0;
    }
    stop = std::chrono::system_clock::now();
    stopped = true;
```

```
    return nullptr;
}
```

```
int GetTime() {
    int result;
    if (countStopMilliseconds == -1) {
        if (started) {
            auto now = std::chrono::system_clock::now();
            std::chrono::duration<double, std::milli> elapsed = now - start;
            result = int(elapsed.count());
        } else {
            result = 0;
        }
    } else {
        if (stopped) {
            std::chrono::duration<double, std::milli> elapsed = stop - start;
            result = int(elapsed.count());
            result -= countStopMilliseconds;
        } else {
            auto now = std::chrono::system_clock::now();
            std::chrono::duration<double, std::milli> elapsed = now - start;
            result = int(elapsed.count());
            result -= countStopMilliseconds;
        }
    }
    return result;
}
```

tree.hpp

```
#pragma once

#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
```

```
typedef const char* Error;
```

```
struct Node {
    int id;
    std::vector<std::shared_ptr<Node>> children;
```

```
    Node(int value) : id(value) {}
    void insert_child(std::shared_ptr<Node> child) {
        children.push_back(child);
    }

};
```

```
class Tree {
    private:
```

```
        std::shared_ptr<Node> root;
```

```
        std::shared_ptr<Node> search_helper(int id, std::shared_ptr<Node>& root) {
            if (root == nullptr) {
                return nullptr;
            }
            if (root->id == id) {
                return root;
            }
            for (std::shared_ptr<Node>& child : root->children) {
                std::shared_ptr<Node> temp = search_helper(id, child);
                if (temp != nullptr) {
                    return temp;
                }
            }
            return nullptr;
        }
```

```
        void remove_subtree(std::shared_ptr<Node>& root) {
            if (root == nullptr) {
                return;
            }
            for (std::shared_ptr<Node>& child : root->children) {
                remove_subtree_helper(child);
            }
            root->children.clear();
            return;
        }
```

```
        void remove_subtree_helper(std::shared_ptr<Node>& root) {
            if (root == nullptr) {
                return;
            }
            for (std::shared_ptr<Node>& child : root->children) {
                remove_subtree_helper(child);
            }
            root = nullptr;
            return;
        }
```

```
        void remove_helper(int id, std::shared_ptr<Node>& root) {
            if (root == nullptr) {
                return;
            }
            if (root->id == id) {
                if (root->children.size() != 0) {
                    remove_subtree(root);
                }
                root = nullptr;
                return;
            }
            for (int i = 0; i < root->children.size(); ++i) {
                if (root->children[i]->id == id) {
                    if (root->children[i]->children.size() != 0) {
                        remove_subtree(root->children[i]);
                    }
                    root->children[i] = nullptr;
                    root->children.erase(root->children.begin() + i);
                    return;
                }
```

```
            }
            for (std::shared_ptr<Node>& child : root->children) {
                remove_helper(id, child);
            }
        }

        bool findPath_helper(std::shared_ptr<Node>& root, int id, std::vector<int>& path) {
            if (root == nullptr) {
                return false;
            }

            path.push_back(root->id);

            if (root->id == id) {
                return true;
            }

            for (std::shared_ptr<Node>& child : root->children) {
                if (findPath_helper(child, id, path)) {
                    return true;
                }
            }

            path.pop_back();
            return false;
        }

        int depth_helper(std::shared_ptr<Node>& root) {
            if (root == nullptr) {
                return 0;
            }

            int maxChildDepth = 0;
            for (std::shared_ptr<Node> child : root->children) {
                maxChildDepth = std::max(maxChildDepth, depth_helper(child));
            }

            return maxChildDepth + 1;
        }

        void print_helper(std::shared_ptr<Node>& root) {
            if (root == nullptr) {
                return;
            }
            std::cout << root->id << std::endl;
            for (std::shared_ptr<Node>& child : root->children) {
                print_helper(child);
            }
            return;
        }

    public:
        Tree() {
            root = nullptr;
        }

        Tree(int id) {
            root = std::make_shared<Node>(id);
        }
```

```
        std::shared_ptr<Node>& getrootptr() {
            return root;
        }

        std::shared_ptr<Node> search(int id) {
            return this->search_helper(id, root);
        }
```

```
        bool Empty() {
            if (root) {
                return false;
            }
            return true;
        }
```

```
        Error insert_root(int id) {
            if (root) {
                return "Error : root is already exists";
            }
```

```
            root = std::make_shared<Node>(id);
            return nullptr;
        }
```

```
        Error insert(int parent_id, int id) {
            std::shared_ptr<Node> have = search(id);
            if (have) {
                return "Error : Already exist";
            }
```

```
            std::shared_ptr<Node> parent = search(parent_id);
            if (!parent) {
                return "Error : Parent not found";
            }
```

```
            std::shared_ptr<Node> child = std::make_shared<Node>(id);
            parent->insert_child(child);
            return nullptr;
        }
```

```
        Error remove(int id) {
            std::shared_ptr<Node> have = search(id);
            if (!have) {
                return "Error : id not exist";
            }
```

```
            remove_helper(id, root);
            return nullptr;
        }
```

```
        std::vector<int> findPath(int id) {
            std::vector<int> path;
            findPath_helper(root, id, path);
            return path;
        }
```

```
        int depth() {
            return this->depth_helper(root);
        }
```

```
        std::vector<int> get_nodes(int id) {
            std::shared_ptr<Node> node_ptr = search(id);
            if (!node_ptr) {
                return {};
            }
            std::vector<int> answers;
            for (std::shared_ptr<Node> child : node_ptr->children) {
                answers.push_back(child->id);
                std::vector<int> child_answers = get_nodes(child->id);
                answers.insert(answers.end(), child_answers.begin(), child_answers.end());
            }
            return answers;
        }
```

```
        void print() {
            print_helper(root);
        }
```

```
        ~Tree() {
            remove_subtree(root);
            root = nullptr;
        }
};
```

ControlNode.cpp

```
#include <iostream>
#include <zmq.hpp>
#include <nlohmann/json.hpp>
#include <csignal>
#include <thread>
#include <cstdlib>

#include "tree.hpp"
#include "request.hpp"
#include "response.hpp"
#include "myMQ.hpp"
```

```
zmq::context_t context(1);
zmq::socket_t socket(context, zmq::socket_type::pair);
bool connected = false;
```

```
int rootID;
```

```
std::vector<int> pids;
```

```
int readID() {
    std::string id_str;
    std::cin >> id_str;
    char* end;
    int id = (int)strtol(id_str.c_str(), &end, 10);
    if ((*end != '\0' && *end != '\n')) {
        return -1;
    }
    return id;
}
```

```
void KillProcess() {
```

```
    for (auto& pid : pids) {
        kill(pid, SIGKILL);
    }
}
```

```
int main() {
    std::cout << "\t\tGuide" << std::endl;
    std::cout << "Creating a new node: create <id> <parent>" << std::endl;
    std::cout << "Execution of a command on a computing node: exec <id> <subcommand>" <<
std::endl;
    std::cout << " Subcommands:" << std::endl;
    std::cout << "  - time: get current time" << std::endl;
    std::cout << "  - start: start timer" << std::endl;
    std::cout << " - stop: stop timer" << std::endl;
    std::cout << "Output of all unavailable nodes: pingall" << std::endl;
    std::cout << "Remove node: remove <id>" << std::endl;
    std::cout << std::endl;
```

```
    Tree tree;

    std::string action;
    while(std::cout << "> " && std::cin >> action) {
        if (action == "create") {
            int id = readID();
            if (id == -1) {
                std::cout << "Error: invalid id" << std::endl;
                continue;
            }
            int parent_id = readID();
            if (parent_id == -1 && !tree.Empty()) {
                std::cout << "Error : invalid parent id" << std::endl;
                continue;
            }
```

```
            Error err;
            if (tree.Empty()) {
                err = tree.insert_root(id);
            } else {
                err = tree.insert(parent_id, id);
            }
```

```
            if (err) {
                std::cout << err << std::endl;
                continue;
            }
```

```
            if (!connected) {
                pid_t pid = fork();
```

```
                if (pid == -1) {
                    std::cout << "Error : fork failed" << std::endl;
                    continue;
                }
```

```
                if (pid == 0) {
                    execl("calculation", "calculation", std::to_string(id).c_str(),
std::to_string(1).c_str(), nullptr);
                }
```

```
                if (pid > 0) {
```

```
                    socket.connect(GetConPort(id).c_str());
                    connected = true;
                    std::cout << "OK: " << pid << std::endl;
                    pids.push_back(pid);
                }
                rootID = id;
            } else {
                std::vector<int> path = tree.findPath(id);
                path.pop_back();
```

```
                Request req(Create, path, id);
                nlohmann::json jsonReq = {
                    {"action", req.action},
                    {"path", req.path},
                    {"id", req.id},
                    {"depth", tree.depth()}
                };
```

```
                std::string jsonReqString = jsonReq.dump();
                zmq::message_t msg(jsonReqString.begin(), jsonReqString.end());
                socket.send(msg, zmq::send_flags::none);
```

```
                std::this_thread::sleep_for(tree.depth() * std::chrono::milliseconds(50));
```

```
                zmq::message_t reply;
                bool replyed;
                try {
                    replyed = bool(socket.recv(reply, zmq::recv_flags::dontwait));
                } catch(zmq::error_t& e) {
                    replyed = false;
                }
```

```
                if (replyed) {
                    std::string replyStr = std::string(static_cast<char*>(reply.data()),
reply.size());
```

```
                    nlohmann::json jsonReply = nlohmann::json::parse(replyStr);
```

```
                    Response resp;
```

```
                    resp.status = jsonReply.at("status");
```

```
                    if (resp.status == ERROR) {
                        std::string error = jsonReply["error"];
                        std::cout << error << std::endl;
                    } else if (resp.status == OK) {
                        int pid = jsonReply["pid"];
                        std::cout << "OK: " << pid << std::endl;
                        pids.push_back(pid);
                    }
                } else {
                    std::cout << "Error: parent node with id = " << id << " not found" <<
std::endl;
                    continue;
                }
            }
        } else if (action == "exec") {
            int id = readID();
            if (id == -1) {
                std::cout << "Error: invalid id" << std::endl;
```

```
                continue;
            }

            std::string command;
            std::cin >> command;

            if (command != "time" && command != "start" && command != "stop") {
                std::cout << "Error: invalid subcommand" << std::endl;
                continue;
            }

            std::shared_ptr<Node> finded = tree.search(id);
            if (!finded) {
                std::cout << "Error : node with id = " << id << " not found" << std::endl;
                continue;
            }

            std::vector<int> path = tree.findPath(id);

            Request req(command, path, id);
            nlohmann::json jsonReq = {
                {"action", req.action},
                {"path", req.path},
                {"id", req.id},
                {"depth", tree.depth()}
            };

            std::string jsonReqString = jsonReq.dump();
            zmq::message_t msg(jsonReqString.begin(), jsonReqString.end());
            socket.send(msg, zmq::send_flags::none);

            std::this_thread::sleep_for(tree.depth() *
std::chrono::milliseconds(tree.depth()*50));

            zmq::message_t reply;
            bool replyed;
            try {
                replyed = bool(socket.recv(reply, zmq::recv_flags::dontwait));
            } catch(zmq::error_t& e) {
                replyed = false;
            }

            Response resp;
            nlohmann::json jsonReply;

            if (replyed) {
                std::string replyStr = std::string(static_cast<char*>(reply.data()),
reply.size());
                jsonReply = nlohmann::json::parse(replyStr);
            } else {
                std::cout << "Error : node with id = " << id << " not available" << std::endl;
                continue;
            }

            resp.status = jsonReply.at("status");
            if (resp.status == ERROR) {
                std::string error = jsonReply["error"];
                std::cout << error << std::endl;
            } else if (resp.status == OK) {
                std::cout << "OK: " << id;
```

```
                if (req.action == Time) {
                    std::cout << " " << jsonReply["result"] << "ms";
                }
                std::cout << std::endl;
            }
        } else if (action == "pingall") {
            Request req("ping");
            req.depth = tree.depth();
            req.timeToWait = 1024;
```

```
            nlohmann::json jsonReq = {
                {"action", req.action},
                {"depth", req.depth},
                {"timeToWait", req.timeToWait}
            };
            std::string jsonReqStr = jsonReq.dump();
            zmq::message_t msg(jsonReqStr.begin(), jsonReqStr.end());
            socket.send(msg, zmq::send_flags::none);
```

```
            std::this_thread::sleep_for(std::chrono::milliseconds(req.timeToWait));
```

```
            zmq::message_t reply;
            bool replyed;
            try {
                replyed = bool(socket.recv(reply, zmq::recv_flags::dontwait));
            } catch(zmq::error_t& error) {
                replyed = false;
            }
```

```
            if (replyed) {
                std::string replyStr = std::string(static_cast<char*>(reply.data()),
reply.size());
                nlohmann::json jsonReply = nlohmann::json::parse(replyStr);
```

```
                Response resp;
                resp.unavailable = std::vector<int>(jsonReply.at("unavailable"));
```

```
                std::cout << "OK: ";
                if (resp.unavailable.empty()) {
                    std::cout << -1 << std::endl;
                    continue;
                }
```

```
                for (auto& elem : resp.unavailable) {
                    auto unavailableSons = tree.get_nodes(elem);
                    std::cout << elem << "; ";
                    for (auto& son : unavailableSons) {
                        std::cout << son << "; ";
                    }
                }
            } else {
                auto unavailable = tree.get_nodes(rootID);
                std::cout << "OK: " << rootID << "; ";
                for (auto& elem : unavailable) {
                    std::cout << elem << "; ";
                }
                std::cout << std::endl;
            }
            std::cout << std::endl;
        } else {
```

```
            std::cout << "Unknown command" << std::endl;
        }
    }
```

```
    KillProcess();
    return 0;
}
```

CalculationNode.cpp

```
#include <fstream>
#include <zmq.hpp>
#include <nlohmann/json.hpp>
#include <chrono>
#include <thread>
#include <iostream>
#include <unordered_map>

#include "myMQ.hpp"
#include "request.hpp"
#include "response.hpp"
#include "timer.hpp"
```

```
int id, current_depth;
```

```
zmq::context_t ParentContext(1);
zmq::socket_t ParentSocket(ParentContext, zmq::socket_type::pair);
```

```
std::unordered_map<int, std::pair<zmq::context_t, zmq::socket_t>> children_data;
```

```
std::string HandleCreate(nlohmann::json& jsonData, Request& req) {
    req.path = std::vector<int>(jsonData.at("path"));
    req.id = jsonData.at("id");
    req.depth = jsonData.at("depth");
```

```
    Response resp;
    nlohmann::json jsonResp;
```

```
    if (req.path.size() == 1) {
        pid_t pid = fork();
```

```
        if (pid == -1) {
            resp.status = ERROR;
```

```
            jsonResp = {
                {"status", resp.status},
                {"error", "forked error"}
            };
```

```
            return jsonResp.dump();
        }
```

```
        if (pid == 0) {
            execl("calculation", "calculation", std::to_string(req.id).c_str(),
std::to_string(current_depth + 1).c_str(), nullptr);
        }
```

```
        if (pid > 0) {
            children_data[req.id].first = zmq::context_t(1);
            children_data[req.id].second = zmq::socket_t(children_data[req.id].first,
zmq::socket_type::pair);
            children_data[req.id].second.connect(GetConPort(req.id).c_str());

            resp.status = OK;
            resp.pid = pid;

            jsonResp = {
                {"status", resp.status},
                {"pid", resp.pid}
            };

            return jsonResp.dump();
        }
    } else {
        Request reqToSon;
        reqToSon.action = Create;
        req.path.erase(req.path.begin());
        reqToSon.path = req.path;
        reqToSon.id = req.id;
        reqToSon.depth = req.depth;

        nlohmann::json jsonReq = {
            {"action", reqToSon.action},
            {"path", reqToSon.path},
            {"id", reqToSon.id},
            {"depth", reqToSon.depth}
        };

        std::string jsonReqString = jsonReq.dump();

        zmq::message_t message(jsonReqString.begin(), jsonReqString.end());
        zmq::message_t reply;

        children_data[reqToSon.path[0]].second.send(message, zmq::send_flags::none);

        std::this_thread::sleep_for(std::chrono::milliseconds(req.depth - current_depth + 1) *
20);

        bool replyed;

        try {
            replyed = bool(children_data[reqToSon.path[0]].second.recv(reply,
zmq::recv_flags::dontwait));
        } catch(const zmq::error_t& error) {
            resp.status = ERROR;
            resp.error = std::string(error.what());

            replyed = false;
        }

        if (replyed) {
            std::string jsonRespString = std::string(static_cast<char*>(reply.data()),
reply.size());
            jsonResp = nlohmann::json::parse(jsonRespString);
        } else {
```

```
            resp.status = ERROR;
            resp.error = "Process with id " + std::to_string(reqToSon.path[0]) + " not
available";
```

```
            jsonResp = {
                {"status", resp.status},
                {"error", resp.error}
            };
        }
```

```
        return jsonResp.dump();
    }
    return "";
}
```

```
std::string HandleExec(nlohmann::json& jsonData, Request& req) {
    req.path = std::vector<int>(jsonData.at("path"));
    req.id = jsonData.at("id");
    req.depth = jsonData.at("depth");
```

```
    Response resp;
    nlohmann::json jsonResp;
```

```
    if (req.path.size() == 1) {
        if (req.action == Start) {
            Error err = StartTimer();
            if (err != nullptr) {
                resp.status = ERROR;
                resp.error = err;
                jsonResp = {
                    {"status", resp.status},
                    {"error", resp.status}
                };
            } else {
                resp.status = OK;
                jsonResp = {
                    {"status", resp.status}
                };
            }
        } else if (req.action == Stop) {
            Error err = StopTimer();
            if (err != nullptr) {
                resp.status = ERROR;
                resp.error = err;
                jsonResp = {
                    {"status", resp.status},
                    {"error", resp.error}
                };
            } else {
                resp.status = OK;
                jsonResp = {
                    {"status", resp.status}
                };
            }
        } else if (req.action == Time) {
            int res = GetTime();
```

```
            resp.status = OK;
            resp.result = res;
            jsonResp = {
```

```
                {"status", resp.status},
                {"result", resp.result}
            };
        } else {
            resp.status = ERROR;
            resp.error = "Wrong action";
            jsonResp = {
                {"status", resp.status},
                {"error", resp.error}
            };
        }
```

```
        return jsonResp.dump();
    } else {
        Request reqToSon;
        reqToSon.action = req.action;
        req.path.erase(req.path.begin());
        reqToSon.path = req.path;
        reqToSon.id = req.id;
        reqToSon.depth = req.depth;
```

```
        nlohmann::json jsonReq = {
            {"action", reqToSon.action},
            {"path", reqToSon.path},
            {"id", reqToSon.id},
            {"depth", reqToSon.depth}
        };
```

```
        std::string jsonReqString = jsonReq.dump();
        zmq::message_t message(jsonReqString.begin(), jsonReqString.end());
```

```
        zmq::message_t reply;
```

```
        children_data[reqToSon.path[0]].second.send(message, zmq::send_flags::none);
```

```
        std::this_thread::sleep_for(std::chrono::milliseconds(req.depth - current_depth + 1) *
20);
```

```
        bool replyed;
```

```
        try {
            replyed = bool(children_data[reqToSon.path[0]].second.recv(reply,
zmq::recv_flags::none));
        } catch (const zmq::error_t& err) {
            replyed = false;
        }
```

```
        if (replyed) {
            std::string jsonRespString = std::string(static_cast<char*>(reply.data()),
reply.size());
            jsonResp = nlohmann::json::parse(jsonRespString);
        } else {
            resp.status = ERROR;
            resp.error = "Process with id" + std::to_string(reqToSon.path[0]) + " not
available";
```

```
            jsonResp = {
                {"status", resp.status},
                {"error", resp.error}
            };
```

```
        }

        return jsonResp.dump();
    }

    return "";
}

std::string HandlePing(nlohmann::json& jsonData, Request& req) {
    req.depth = jsonData.at("depth");
    req.timeToWait = jsonData.at("timeToWait");
    req.timeToWait /= 2;

    Response resp;
    nlohmann::json jsonResp;
    std::vector<int> unavailable;

    std::vector<std::pair<Request, int>> children_requests(children_data.size());

    int count = 0;
    for (const auto& [id, Pair] : children_data) {
        children_requests[count].second = id;
        ++count;
    }

    for (int i = 0; i < children_requests.size(); ++i) {
        children_requests[i].first.action = req.action;
        children_requests[i].first.depth = req.depth;
        children_requests[i].first.timeToWait = req.timeToWait;
        nlohmann::json jsonReqToSon = {
            {"action", children_requests[i].first.action},
            {"depth", children_requests[i].first.depth},
            {"timeToWait", children_requests[i].first.timeToWait}
        };

        std::string jsonReqToSonString = jsonReqToSon.dump();
        zmq::message_t messageToSon(jsonReqToSon.begin(), jsonReqToSon.end());

        children_data[children_requests[i].second].second.send(messageToSon,
zmq::send_flags::none);

        std::this_thread::sleep_for(std::chrono::milliseconds(req.timeToWait));

        zmq::message_t replyFromSon;
        bool replyed = true;

        try {
            replyed = bool(children_data[children_requests[i].second].second.recv(replyFromSon,
zmq::recv_flags::dontwait));
        } catch (const zmq::error_t& error) {
            replyed = false;
        }

        if (!replyed) {
            unavailable.push_back(children_requests[i].second);
        } else {
            std::string jsonReplyString = std::string(static_cast<char*>(replyFromSon.data()),
replyFromSon.size());
```

```
            nlohmann::json jsonReply = nlohmann::json::parse(jsonReplyString);
            std::vector<int> new_unavailable = std::vector<int>(jsonReply.at("unavailable"));
            unavailable.insert(unavailable.begin(), new_unavailable.begin(),
new_unavailable.end());
        }
    }
```

```
    resp.status = OK;
    resp.unavailable = unavailable;
```

```
    jsonResp = {
        {"status", resp.status},
        {"unavailable", resp.unavailable}
    };
```

```
    return jsonResp.dump();
}
```

```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        return 1;
    }
```

```
    id = std::stoi(argv[1]);
    current_depth = std::stoi(argv[2]);
```

```
    ParentSocket.bind(GetBindPort(id));
    zmq::message_t request;
    while(ParentSocket.recv(request, zmq::recv_flags::none)) {
        std::string jsonReqString = std::string(static_cast<char*>(request.data()),
request.size());
```

```
        nlohmann::json jsonData = nlohmann::json::parse(jsonReqString);
        Request req;
        req.action = jsonData["action"];
```

```
        std::string jsonRespString;
```

```
        if (req.action == Time || req.action == Start || req.action == Stop) {
            jsonRespString = HandleExec(jsonData, req);
        } else if (req.action == Ping) {
            jsonRespString = HandlePing(jsonData, req);
        } else if (req.action == Create) {
            //std::cout << id << std::endl;
            jsonRespString = HandleCreate(jsonData, req);
        }
```

```
        zmq::message_t reply(jsonRespString.begin(), jsonRespString.end());
        ParentSocket.send(reply, zmq::send_flags::none);
    }
```

```
    ParentSocket.close();
    ParentContext.close();
    return 0;
}
```

Пример работы программы

mitrijmrsh@LAPTOP-7SMT8REA:~/Labs-git/os_lab_5-7/src/build$ ./control

Guide

Creating a new node: create <id> <parent>

Execution of a command on a computing node: exec <id> <subcommand>

Subcommands:

- time: get current time
- start: start timer
- stop: stop timer

Output of all unavailable nodes: pingall

Remove node: remove <id>

> create 1 1

OK: 15670

> create 2 1

OK: 15697

> create 3 1

OK: 15705

> exec 3 start

OK: 3

> exec 3 time

OK: 3 3323ms

> exec 3 time

OK: 3 9200ms

> exec 3 stop

OK: 3

> exec 3 time

OK: 3 12105ms

> pingall
OK: 3;

Вывод

В завершении своей работы, хочу заметить, что изучение сокетов и очереди сообщений очень важные и интересные темы для программиста, который занимается бэкенд-разработкой. Знания работы сокетов, позволяет понимать базовые принципы работы многих технологий общения сервера и клиента. Изучив ZMQ, как технологию для очереди сообщений, я понял насколько она удобная и простая в понимании. Работа очередей сообщений позволяет двусвязно передавать сообщения в обе стороны и производить над ними операции.