Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу
«Операционные системы»

Управление серверами сообщений

Студент: Касимов Магомед Магомедсаламович
Группа: М80 – 206Б-18
Вариант: 27
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2019

Постановка задачи

Реализовать распределенную систему по обработке запросов. В данной 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 сервера сообщений zmq. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом.

Вариант задания: 27. Топология — дерево общего вида. Тип вычислительной команды — сумма n чисел. Тип проверки узлов на доступность — пинг всех узлов.

Общие сведения о программе

Программа состоит из двух файлов, которые компилируются в исполнительные файлы (которые представляют управляющий и вычислительные узлы), а также из статической библиотеки, которая подключается к вышеуказанным файлам. Общение между процессами происходит с помощью библиотеки zmq.

Общий метод и алгоритм решения

- Управляющий узел принимает команды, обрабатывает их и пересылает дочерним узлам(или выводит сообщение об ошибке).
- Дочерние узлы проверяют, может ли быть команда выполнена в данном узле, если нет, то команда пересылается в один из дочерних узлов, из которого возвращается некоторое сообщение(об успехе или об ошибке), которое потом пересылается обратно по дереву.
- Для корректной проверки на доступность узлов, используется дерево, эмулирующее поведение узлов в данной топологии(например, при удалении узла, удаляются все его потомки).
- Если узел недоступен, то по истечении таймаута будет сгенерировано сообщение о недоступности узла и оно будет передано вверх по дереву, к управляющему узлу.
- При удалении узла, все его потомки рекурсивно уничтожаются.

Код программы

main_node.cpp:

```
#include <iostream>
#include "zmq.hpp"
#include <string>
#include <zconf.h>
#include <vector>
#include <csignal>
#include <sstream>
#include <set>
#include <algorithm>
#include <memory>
#include <unordered_map>
#include "sf.h"

template <typename T>
std::ostream& operator << (std::ostream& os, std::vector<T> v) {
    for (const T& i : v) {
        os << i << " ";
    }
    return os;
}

struct TreeNode {
    TreeNode(int id, std::weak_ptr<TreeNode> parent)
            : id_(id), parent_(parent) {}

    int id_;
    std::weak_ptr<TreeNode> parent_;
    std::unordered_map<int, std::shared_ptr<TreeNode>> nodes_;
};

class IdIndexingTree {
public:
    IdIndexingTree() = default;
    ~IdIndexingTree() = default;

    bool Insert(int elem, int parent_id) {
        if (root_ == nullptr) {
            root_ = std::make_shared<TreeNode>(elem,
std::weak_ptr<TreeNode>());
            return true;
        }
        std::vector<int> path = GetPathTo(parent_id);
        if (path.empty()) {
            return false;
        }
        path.erase(path.begin());
        std::shared_ptr<TreeNode> node = root_;
        for (int i : path) {
            if (node->nodes_.count(i) == 0) {
                throw std::logic_error("Shit happened");
            }
            node = node->nodes_[i];
        }
        node->nodes_[elem] = std::make_shared<TreeNode>(elem,node);
        return true;
    }

    bool Erase(int elem) {
        std::vector<int> path = GetPathTo(elem);
        if (path.empty()) {
```

```
            return false;
        }
        path.erase(path.begin());
        std::shared_ptr<TreeNode> node = root_;
        for (int i : path) {
            if (node->nodes_.count(i) == 0) {
                throw std::logic_error("Shit happened");
            }
            node = node->nodes_[i];
        }
        if (node->parent_.lock() == nullptr) {
            root_ = nullptr;
            return true;
        }
        node = node->parent_.lock();
        node->nodes_.erase(elem);
        return true;
    }

    [[nodiscard]] std::vector<int> GetPathTo(int id) const {
        std::vector<int> v;
        if (!SearchFunc(root_, id, v)) {
            return {};
        }
        return v;
    }

    std::vector<int> GetNodes() const {
        std::vector<int> v;
        GetNodes(root_, v);
        return v;
    }

private:
    bool SearchFunc(std::shared_ptr<TreeNode> node, int id, std::vector<int>&
v) const {
        if (node == nullptr) {
            return false;
        }
        if (node->id_ == id) {
            v.push_back(node->id_);
            return true;
        }
        v.push_back(node->id_);
        for (auto [child_id, child_node] : node->nodes_) {
            if (SearchFunc(child_node, id, v)) {
                return true;
            }
        }
        v.pop_back();
        return false;
    }

    void GetNodes(std::shared_ptr<TreeNode> node, std::vector<int>& v) const
{
        if (node == nullptr) {
            return;
        }
        v.push_back(node->id_);
        for (auto [child_id, child_ptr] : node->nodes_) {
            GetNodes(child_ptr, v);
        }
    }
```

```
    std::shared_ptr<TreeNode> root_ = nullptr;
};


int main() {
    std::string command;
    IdIndexingTree ids;
    size_t child_pid = 0;
    int child_id = 0;
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    int linger = 0;
    main_socket.setsockopt(ZMQ_SNDTIMEO, 2000);
    main_socket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
    int port = bind_socket(main_socket);
    while (true) {
        std::cin >> command;
        if (command == "create") {
            size_t node_id, parent_id;
            std::string result;
            std::cin >> node_id >> parent_id;
            if (child_pid == 0) {
                child_pid = fork();
                if (child_pid == -1) {
                    std::cout << "Unable to create first worker node\n";
                    child_pid = 0;
                    exit(1);
                } else if (child_pid == 0) {
                    create_node(node_id, parent_id, port);
                } else {
                    parent_id = 0;
                    //ids.Insert(node_id,0);
                    child_id = node_id;
                    send_message(main_socket,"pid");
                    result = recieve_message(main_socket);
                }

            } else {
                if (!ids.GetPathTo(node_id).empty()) {
                    std::cout << "Error: Node already exists" << "\n";
                    continue;
                }
                std::vector<int> path = ids.GetPathTo(parent_id);
                if (path.empty()) {
                    std::cout << "Error: No parent node" << "\n";
                    continue;
                }
                path.erase(path.begin());
                std::ostringstream msg_stream;
                msg_stream << "create " << path.size(); //CЃPSP°C‡P°P»P°
PїCѓC,CЊ PїPsC,PsPj PёPґ
                for (int i : path) {
                    msg_stream << " " << i;
                }
                msg_stream << " " << node_id;
                send_message(main_socket, msg_stream.str());
                result = recieve_message(main_socket);
            }

            if (result.substr(0,2) == "Ok") {
                ids.Insert(node_id, parent_id);
            }
            std::cout << result << "\n";
```

```
        } else if (command == "remove") {
            if (child_pid == 0) {
                std::cout << "Error: No such node\n";
                continue;
            }
            size_t node_id;
            std::cin >> node_id;
            if (node_id == child_id) {
                send_message(main_socket, "kill");
                recieve_message(main_socket);
                kill(child_pid, SIGTERM);
                kill(child_pid, SIGKILL);
                child_id = 0;
                child_pid = 0;
                std::cout << "Ok\n";
                ids.Erase(node_id);
                continue;
            }
            std::vector<int> path = ids.GetPathTo(node_id);
            if (path.empty()) {
                std::cout << "Error: No such node" << "\n";
                continue;
            }
            path.erase(path.begin());
            std::ostringstream msg_stream;
            msg_stream << "remove " << path.size() - 1;
            for (int i : path) {
                msg_stream << " " << i;
            }
            send_message(main_socket, msg_stream.str());
            std::string recieved_message = recieve_message(main_socket);
            if (recieved_message.substr(0, std::min<int>(recieved_mes-
sage.size(), 2)) == "Ok") {
                ids.Erase(node_id);
            }
            std::cout << recieved_message << "\n";

        } else if (command == "exec") {
            int id, n;
            std::cin >> id >> n;
            std::vector<int> path = ids.GetPathTo(id);
            if (path.empty()) {
                std::cout << "Error: No such node" << "\n";
                continue;
            }
            path.erase(path.begin());
            std::vector<int> numbers(n);
            for (int i = 0; i < n; ++i) {
                std::cin >> numbers[i];
            }
            std::ostringstream msg_stream;
            msg_stream << "exec " << path.size();
            for (int i : path) {
                msg_stream << " " << i;
            }
            msg_stream << " " << n;
            for (int i : numbers) {
                msg_stream << " " << i;
            }
            std::string test = msg_stream.str();
            send_message(main_socket, msg_stream.str());
            std::string recieved_message = recieve_message(main_socket);
            std::cout << recieved_message << "\n";
```

```
        } else if (command == "pingall") {
            if (child_pid == 0) {
                std::cout << "No nodes\n";
                continue;
            }
            send_message(main_socket,"pingall");
            std::string recieved = recieve_message(main_socket);
            std::istringstream is(recieved);
            std::vector<int> recieved_nodes;
            int elem;
            while (is >> elem) {
                recieved_nodes.push_back(elem);
            }
            std::sort(recieved_nodes.begin(), recieved_nodes.end());
            std::vector<int> all_nodes = ids.GetNodes();
            std::sort(all_nodes.begin(), all_nodes.end());
            std::cout << "Recieved nodes " << recieved_nodes << "\n";
            std::cout << "All nodes " << all_nodes << "\n";
        } else if (command == "exit") {
            break;
        }

    }

    return 0;
}
```

child_node.cpp:

```
#include <iostream>
#include "zmq.hpp"
#include <string>
#include <sstream>
#include <zconf.h>
#include <exception>
#include <csignal>
#include <unordered_map>
#include "sf.h"


int main(int argc, char** argv) { //P°CЂPiCґPjPµPSC,C‹ - P°PЉPrPë Pë
PSPsPjPµCЂ PїPsCЂC,P°, Pє PePsC,PsCЂPsPjCґ PSCґP¶PSPs
PїPsPrPeP»CЂC‡PëC,CЊCЃCЏ

    int id = std::stoi(argv[1]);
    int parent_id = std::stoi(argv[2]);
    int parent_port = std::stoi(argv[3]);

    zmq::context_t context(3);
    zmq::socket_t parent_socket(context, ZMQ_REP);

    parent_socket.connect(get_port_name(parent_port));

    std::unordered_map<int, zmq::socket_t> sockets;
    std::unordered_map<int, int> pids;
    std::unordered_map<int, int> ports;

    while (true) {
        std::string request_string;

        request_string = recieve_message(parent_socket);

        std::istringstream command_stream(request_string);
```

```
        std::string command;
        command_stream >> command;
        if (command == "id") {
            std::string parent_string = "Ok:" + std::to_string(id);
            send_message(parent_socket, parent_string);
        } else if (command == "pid") {
            std::string parent_string = "Ok:" + std::to_string(getpid());
            send_message(parent_socket, parent_string);
        } else  if (command == "create") {
            int size, node_id;
            command_stream >> size;
            std::vector<int> path(size);
            for (int i = 0; i < size; ++i) {
                command_stream >> path[i];
            }
            command_stream >> node_id;
            if (size == 0) {
                sockets.emplace(std::piecewise_construct,
                                std::forward_as_tuple(node_id),
                                std::forward_as_tuple(context, ZMQ_REQ));
                int port = bind_socket(sockets.at(node_id));
                int pid = fork();
                if (pid == -1) {
                    send_message(parent_socket, "Cannot fork");
                    continue;
                } else if (pid == 0) {
                    create_node(node_id, id, port);
                } else {
                    ports[node_id] = port;
                    pids[node_id] = pid;
                    send_message(sockets.at(node_id), "pid");
                    send_message(parent_socket, recieve_message(sock-
ets.at(node_id)));
                }
            } else {
                int next_id = path.front();
                path.erase(path.begin());
                std::ostringstream msg_stream;
                msg_stream << "create " << path.size();
                for (int i : path) {
                    msg_stream << " " << i;
                }
                msg_stream << " " << node_id;
                send_message(sockets.at(next_id), msg_stream.str());
                send_message(parent_socket, recieve_message(sock-
ets.at(next_id)));
            }
        } else if (command == "remove") {
            int size, node_id;
            command_stream >> size;
            std::vector<int> path(size);
            for (int i = 0; i < size; ++i) {
                command_stream >> path[i];
            }
            command_stream >> node_id;
            if (path.empty()) {
                send_message(sockets.at(node_id), "kill");
                recieve_message(sockets.at(node_id));
                kill(pids[node_id], SIGTERM);
                kill(pids[node_id], SIGKILL);
                pids.erase(node_id);
                sockets.at(node_id).discon-
nect(get_port_name(ports[node_id]));
                ports.erase(node_id);
```

```
                sockets.erase(node_id);
                send_message(parent_socket, "Ok");
            } else {
                int next_id = path.front();
                path.erase(path.begin());
                std::ostringstream msg_stream;
                msg_stream << "remove " << path.size();
                for (int i : path) {
                    msg_stream << " " << i;
                }
                msg_stream << " " << node_id;
                send_message(sockets.at(next_id), msg_stream.str());
                send_message(parent_socket, recieve_message(sock-
ets.at(next_id)));
            }
        } else if (command == "exec") {
            int path_size;
            command_stream >> path_size;
            std::vector<int> path(path_size);
            for (int i = 0; i < path_size; ++i) {
                command_stream >> path[i];
            }
            int number_size;
            command_stream >> number_size;
            std::vector<int> numbers(number_size);
            for (int i = 0; i < number_size; ++i) {
                command_stream >> numbers[i];
            }
            if (path.empty()) {
                int sum = 0;
                for (int i : numbers) {
                    sum += i;
                }
                send_message(parent_socket, "Ok " + std::to_string(id) + ":"
+ std::to_string(sum));
            } else {
                int next_id = path.front();
                path.erase(path.begin());
                std::ostringstream msg_stream;
                msg_stream << "exec " << path.size();
                for (int i : path) {
                    msg_stream << " " << i;
                }
                msg_stream << " " << number_size;
                for (int i : numbers) {
                    msg_stream << " " << i;
                }
                send_message(sockets.at(next_id), msg_stream.str());
                send_message(parent_socket, recieve_message(sock-
ets.at(next_id)));
            }


        } else if (command == "pingall") {
            std::ostringstream res;
            for (auto& [child_id, child_socket] : sockets) {
                send_message(child_socket, "pingall");
                std::string local_result = recieve_message(child_socket);
                if (!local_result.empty() && local_result.sub-
str(std::min<int>(local_result.size(),5)) != "Error") {
                    res << local_result << " ";
                }
            }
            res << id << " ";
```

```
            send_message(parent_socket, res.str());

        } else if (command == "kill") {
            for (auto& [child_id, child_socket] : sockets) {
                send_message(child_socket, "kill");
                recieve_message(child_socket);
                kill(pids[child_id], SIGTERM);
                kill(pids[child_id], SIGKILL);

            }
            send_message(parent_socket, "Ok");
        }
        if (parent_port == 0) {
            break;
        }
    }

}
```

server_functions.cpp:

```
#include "server_functions.h"

std::string get_port_name(int port) {
    return "tcp://127.0.0.1:" + std::to_string(port);
}


Message::Message(std::string type)
        : type_(std::move(type)) {}


const std::string &Message::GetType() const {
    return type_;
}

zmq::message_t Message::GetZmqMessage() const {
    std::string message_string = GetString();
    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(), message_string.size());
    return message;
}

template<typename T>
std::unique_ptr<T> Message::MessageCast(std::unique_ptr<Message> &to_cast) {
    if (dynamic_cast<T *>(to_cast.get())) {
        std::unique_ptr<T> result;
        result.reset(dynamic_cast<T *>(to_cast.release()));
        return result;
    } else {
        return nullptr;
    }
}


CreationMessage::CreationMessage(int id, int parent, std::vector<int> path)
        : Message("create"), id_(id), parent_(parent), path_(std::move(path))
{}

std::string CreationMessage::GetString() const {
    std::ostringstream os;
    os << type_ << " " << id_ << " " << parent_ << path_.size() << " ";
    for (int i : path_) {
```

```
        os << i << " ";
    }
    return os.str();
}

int CreationMessage::GetId() const {
    return id_;
}

int CreationMessage::GetParent() const {
    return parent_;
}

void CreationMessage::PathPopFront() {
    if (path_.empty()) {
        throw std::logic_error("Empty path");
    }
    path_.erase(path_.begin());
}

int CreationMessage::GetFrontId() const {
    return *path_.begin();
}


DeletionMessage::DeletionMessage(int id, std::vector<int> path)
        : Message("remove"), id_(id), path_(std::move(path)) {}

std::string DeletionMessage::GetString() const {
    std::ostringstream os;
    os << type_ << " " << id_ << " " << path_.size();
    for (int i : path_) {
        os << i << " ";
    }
    return os.str();
}

int DeletionMessage::GetId() const {
    return id_;
}

int DeletionMessage::GetFrontId() const {
    return *path_.begin();
}

void DeletionMessage::PathPopFront() {
    if (path_.empty()) {
        throw std::logic_error("Empty path");
    }
    path_.erase(path_.begin());
}


ExecutionMessage::ExecutionMessage(int id, std::vector<int> v, std::vec-
tor<int> path)
        : Message("exec"), id_(id), numbers_(std::move(v)),
path_(std::move(path)) {}

std::string ExecutionMessage::GetString() const {
    std::ostringstream os;
    os << type_ << " " << id_ << " " << numbers_.size();
    for (int i : numbers_) {
        os << " " << i;
    }
```

```
    os << " " << path_.size();
    for (int i : path_) {
        os << " " << i;
    }
    return os.str();
}

int ExecutionMessage::GetFrontId() const {
    return *path_.begin();
}

void ExecutionMessage::PathPopFront() {
    if (path_.empty()) {
        throw std::logic_error("Empty path");
    }
    path_.erase(path_.begin());
}

int ExecutionMessage::GetId() const {
    return id_;
}

const std::vector<int> &ExecutionMessage::GetNumbers() const {
    return numbers_;
}


PingMessage::PingMessage()
        : Message("ping") {}

std::string PingMessage::GetString() const {
    std::ostringstream os;
    os << type_;
    return os.str();
}


ChildrenMurderMessage::ChildrenMurderMessage()
        : Message("kill") {}

std::string ChildrenMurderMessage::GetString() const {
    return type_;
}

std::string GetStrFromType(ErrorType type) {
    switch (type) {
        case ErrorType::NoSuchNode :
            return "No_Such_Node";
        case ErrorType::NodeUnavailable:
            return "Node_Unavailable";
    }
}

ErrorType GetTypeFromStr(const std::string &str) {
    if (str == "No_Such_Node") {
        return ErrorType::NoSuchNode;
    }
    if (str == "Node_Unavailable") {
        return ErrorType::NodeUnavailable;
    }
    throw std::logic_error("Wrong str");
}
```

```
ErrorMessage::ErrorMessage(ErrorType type)
        : Message("error"), error_type_(type) {}

std::string ErrorMessage::GetString() const {
    return type_ + " " + GetStrFromType(error_type_);
}

ErrorType ErrorMessage::GetErrorType() const {
    return error_type_;
}


OkMessage::OkMessage(std::string containment)
        : Message("ok"), message_(std::move(containment)) {}

std::string OkMessage::GetString() const {
    return type_ + " " + message_;
}

std::string OkMessage::GetContentOnly() const {
    return message_;
}


std::unique_ptr<Message> MessageFactory::CreateMessage(const zmq::message_t
&msg) {
    std::istringstream is(std::string(static_cast<const char *>(msg.data()),
msg.size()));
    std::string msg_type;
    is >> msg_type;
    if (msg_type == "create") {
        int id, parent_id, size;
        is >> id >> parent_id >> size;
        std::vector<int> v;
        for (int i = 0; i < size; ++i) {
            int elem;
            is >> elem;
            v.push_back(elem);
        }
        return std::make_unique<CreationMessage>(id, parent_id,
std::move(v));
    } else if (msg_type == "remove") {
        int id, size;
        is >> id >> size;
        std::vector<int> v;
        for (int i = 0; i < size; ++i) {
            int elem;
            is >> elem;
            v.push_back(elem);
        }
        return std::make_unique<DeletionMessage>(id, std::move(v));
    } else if (msg_type == "exec") {
        int id, numbers_count, size;
        is >> id >> numbers_count;
        std::vector<int> numbers(numbers_count);
        for (int i = 0; i < numbers_count; ++i) {
            is >> numbers[i];
        }
        is >> size;
        std::vector<int> path(size);
        for (int i = 0; i < size; ++i) {
            is >> path[i];
        }
```

```
        return std::make_unique<ExecutionMessage>(id, std::move(numbers),
std::move(path));
    } else if (msg_type == "ping") {
        int id;
        is >> id;
        return std::make_unique<PingMessage>();
    } else if (msg_type == "kill") {
        return std::make_unique<ChildrenMurderMessage>();
    } else if (msg_type == "error") {
        std::string str;
        getline(is, str);
        return std::make_unique<ErrorMessage>(GetTypeFromStr(str));
    } else if (msg_type == "ok") {
        std::string str;
        getline(is, str);
        return std::make_unique<OkMessage>(std::move(str));
    } else {
        throw std::logic_error("Unknown type of message");
    }
}


std::unique_ptr<Message> NodeInfo::RecieveMessage() {
    zmq::message_t msg;
    try {
        socket_.recv(&msg);
    } catch (...) {
        return std::make_unique<ErrorMessage>(ErrorType::NodeUnavailable);
    }
    return MessageFactory::CreateMessage(msg);
}

void NodeInfo::SendMessage(const Message &msg) {
    socket_.send(msg.GetZmqMessage());
};

int NodeInfo::GetId() const {
    return id_;
}

int NodeInfo::GetPid() const {
    return pid_;
}

int NodeInfo::GetPort() const {
    return port_;
}

bool NodeInfo::HasError() const {
    return false;
}


NodeInfo::NodeInfo(int id, int pid, int port, zmq::context_t &context, int
ZMQ_SOCKET_TYPE)
        : id_(id), pid_(pid), port_(port), socket_(context, ZMQ_SOCKET_TYPE)
{}


ParentNodeInfo::ParentNodeInfo(int parent_id, int parent_port, zmq::context_t
&context)
        : NodeInfo(parent_id, getppid(), parent_port, context, ZMQ_REP) {
    socket_.connect(get_port_name(port_));
}
```

```
ChildNodeInfo::ChildNodeInfo(int child_id, int child_pid, zmq::context_t
&context)
        : NodeInfo(child_id, child_pid, 30000, context, ZMQ_REQ) {
    socket_.setsockopt(ZMQ_SNDTIMEO, 2000);
    while (true) {
        try {
            socket_.bind(get_port_name(port_));
            break;
        } catch (zmq::error_t &error) {
            port_++;
        }
    }
}


Node::Node(int id, int parent_id, int parent_port)
        : id_(id), context_(3), parent_node_(parent_id, parent_port, con-
text_) {}

bool Node::CreateNode(int id) {
    int cpid = fork();
    child_infos_.emplace(std::piecewise_construct,
                         std::forward_as_tuple(id),
                         std::forward_as_tuple(id, cpid, context_));
    if (cpid == -1) {
        child_infos_.erase(id);
        return false;
    }
    if (cpid == 0) {
        char *arg1 = strdup((std::to_string(id)).c_str());
        char *arg2 = strdup((std::to_string(id_)).c_str()); //parent_id
        char *arg3 = strdup((std::to_string(child_infos_[id].Get-
Port())).c_str());
        char *args[] = {"./child_node", arg1, arg2, arg3, NULL};
        execv("./child_node", args);
    }

    return true;
}

void Node::Run() {
    while (true) {
        std::unique_ptr<Message> parent_message = parent_node_.RecieveMes-
sage();
        std::string type = parent_message->GetType();
        std::unique_ptr<Message> result;
        if (type == "create") {
            result = ProcessMessage(Message::MessageCast<Creation-
Message>(parent_message));
        } else if (type == "remove") {
            result = ProcessMessage(Message::MessageCast<Deletion-
Message>(parent_message));
        } else if (type == "ping") {
            result = ProcessMessage(Message::MessageCast<PingMessage>(par-
ent_message));
        } else if (type == "kill") {
            result = ProcessMessage(Message::MessageCast<ChildrenMurderMes-
sage>(parent_message));
        } else if (type == "exec") {
            result = ProcessMessage(Message::MessageCast<Execution-
Message>(parent_message));
        }
```

```
        parent_node_.SendMessage(*result);
        if (id_ < 0) {
            break;
        }
    }
}

std::unique_ptr<Message> Node::ProcessMessage(std::unique_ptr<Creation-
Message> msg) {
    if (!msg) {
        throw std::runtime_error("bad_ptr");
    }
    if (msg->GetParent() == id_) {
        CreateNode(msg->GetId());
        return std::make_unique<OkMessage>("pid " + std::to_string(child_in-
fos_[msg->GetId()].GetPid()));
    } else {
        int next_id = msg->GetFrontId();
        msg->PathPopFront();
        child_infos_[next_id].SendMessage(*msg);
        return child_infos_[next_id].RecieveMessage();
    }
}

std::unique_ptr<Message> Node::ProcessMessage(std::unique_ptr<Deletion-
Message> msg) {
    if (!msg) {
        throw std::runtime_error("bad_ptr");
    }
    if (child_infos_.count(msg->GetId()) == 1) {
        child_infos_[msg->GetId()].SendMessage(ChildrenMurderMessage());
        child_infos_[msg->GetId()].RecieveMessage();
        kill(child_infos_[msg->GetId()].GetPid(), SIGTERM);
        kill(child_infos_[msg->GetId()].GetPid(), SIGKILL);
        int erased_pid = child_infos_[msg->GetId()].GetPid();
        return std::make_unique<OkMessage>("pid " +
std::to_string(erased_pid));
    } else {
        int next_id = msg->GetFrontId();
        msg->PathPopFront();
        child_infos_[next_id].SendMessage(*msg);
        return child_infos_[next_id].RecieveMessage();
    }
}

std::unique_ptr<Message> Node::ProcessMessage(std::unique_ptr<PingMessage>
msg) {
    if (!msg) {
        throw std::runtime_error("bad_ptr");
    }
    std::vector<std::unique_ptr<Message>> recieved_messages;
    for (auto&[id, node] : child_infos_) {
        recieved_messages.push_back(node.RecieveMessage());
    }
    std::string ok_messages = std::to_string(id_) + " ";
    for (std::unique_ptr<Message> &ptr : recieved_messages) {
        std::unique_ptr<OkMessage> ok_msg = Message::MessageCast<OkMes-
sage>(ptr);
        if (ok_msg) { // cast will corrupt original unique_ptr: carefully
            ok_messages += " " + ok_msg->GetContentOnly();
        }
    }
    return std::make_unique<OkMessage>(ok_messages);
}
```

```
std::unique_ptr<Message> Node::ProcessMessage(std::unique_ptr<ChildrenMurder-
Message> msg) {
    if (!msg) {
        throw std::runtime_error("bad_ptr");
    }
    for (auto&[id, node] : child_infos_) {
        node.SendMessage(ChildrenMurderMessage());
        node.RecieveMessage();
    }
    for (auto&[id, node] : child_infos_) {
        int pid = node.GetPid();
        kill(pid, SIGTERM);
        kill(pid, SIGKILL);
    }
    child_infos_.clear();
    return std::make_unique<OkMessage>("");
}

std::unique_ptr<Message> Node::ProcessMessage(std::unique_ptr<Execution-
Message> msg) {
    if (!msg) {
        throw std::runtime_error("bad_ptr");
    }
    if (msg->GetId() == id_) {
        int sum = 0;
        std::for_each(msg->GetNumbers().begin(), msg->GetNumbers().end(),
[&sum](int elem) {
            sum += elem;
        });
        return std::make_unique<OkMessage>(std::to_string(id_) + " " +
std::to_string(sum));
    } else {
        int next_id = msg->GetFrontId();
        msg->PathPopFront();
        child_infos_[next_id].SendMessage(*msg);
        return child_infos_[next_id].RecieveMessage();
    }
}
```

server_functions.h:

```
#pragma once
#include <string>
#include <zconf.h>
#include "zmq.hpp"
#include <csignal>
#include <algorithm>
#include <sstream>
#include <memory>
#include <unordered_map>



class Message {
public:
    Message(std::string type);

    virtual ~Message() = default;

    [[nodiscard]] virtual std::string GetString() const = 0;
```

```
    const std::string& GetType() const;

    zmq::message_t GetZmqMessage() const;

    template <typename T>
    static std::unique_ptr<T> MessageCast(std::unique_ptr<Message>& to_cast);

protected:
    std::string type_;
};

class CreationMessage : public Message {
public:
    CreationMessage(int id, int parent, std::vector<int> path);

    [[nodiscard]] std::string GetString() const override;

    [[nodiscard]] int GetId() const;

    [[nodiscard]] int GetParent() const;

    void PathPopFront();

    int GetFrontId() const;

private:
    int id_;
    int parent_;
    std::vector<int> path_;
};

class DeletionMessage : public Message {
public:
    DeletionMessage(int id, std::vector<int> path);

    [[nodiscard]] std::string GetString() const override;

    [[nodiscard]] int GetId() const;

    int GetFrontId() const;

    void PathPopFront();

private:
    int id_;
    std::vector<int> path_;
};

class ExecutionMessage : public Message {
public:
    ExecutionMessage(int id, std::vector<int> v, std::vector<int> path);

    std::string GetString() const override;

    int GetFrontId() const;

    void PathPopFront();

    [[nodiscard]] int GetId() const;

    [[nodiscard]] const std::vector<int>&  GetNumbers() const;

private:
    int id_;
```

```
    std::vector<int> numbers_;
    std::vector<int> path_;
};

class PingMessage : public Message {
public:
    PingMessage();

    [[nodiscard]] std::string GetString() const override;
};

class ChildrenMurderMessage : public Message {
public:
    ChildrenMurderMessage();

    [[nodiscard]] std::string GetString() const override;
};

enum class ErrorType {
    NoSuchNode,
    NodeUnavailable
};

std::string GetStrFromType(ErrorType type);
ErrorType GetTypeFromStr(const std::string& str);

class ErrorMessage : public Message {
public:
    ErrorMessage(ErrorType type);

    [[nodiscard]] std::string GetString() const override;

    ErrorType GetErrorType() const;

private:
    ErrorType error_type_;
};

class OkMessage : public Message {
public:
    OkMessage(std::string containment);

    [[nodiscard]] std::string GetString() const override;

    [[nodiscard]] std::string GetContentOnly() const;

private:
    std::string message_;
};

class MessageFactory {
public:
    MessageFactory() = default;
    static std::unique_ptr<Message> CreateMessage(const zmq::message_t& msg);
};

class NodeInfo {
public:
    virtual ~NodeInfo() = default;
    virtual std::unique_ptr<Message> RecieveMessage();

    virtual void SendMessage(const Message& msg);

    [[nodiscard]] int GetId() const;
```

```
    [[nodiscard]] int GetPid() const;

    [[nodiscard]] int GetPort() const;

    bool HasError() const;

protected:
    NodeInfo(int id, int pid, int port, zmq::context_t& context, int
ZMQ_SOCKET_TYPE);

    int id_;
    int pid_;
    int port_;
    zmq::socket_t socket_;
};

class ParentNodeInfo : public NodeInfo {
public:
    ParentNodeInfo(const ParentNodeInfo&) = delete;
    ParentNodeInfo(int parent_id, int parent_port, zmq::context_t& context);

};

class ChildNodeInfo : public NodeInfo {
public:
    ChildNodeInfo(const ChildNodeInfo&) = delete;
    ChildNodeInfo(int child_id, int child_pid, zmq::context_t& context);

};

class Node {
public:
    Node(int id, int parent_id, int parent_port);

    bool CreateNode(int id);

    void Run();

private:

    std::unique_ptr<Message> ProcessMessage(std::unique_ptr<CreationMessage>
msg);
    std::unique_ptr<Message> ProcessMessage(std::unique_ptr<DeletionMessage>
msg);
    std::unique_ptr<Message> ProcessMessage(std::unique_ptr<PingMessage>
msg);
    std::unique_ptr<Message> ProcessMessage(std::unique_ptr<ChildrenMurder-
Message> msg);
    std::unique_ptr<Message> ProcessMessage(std::unique_ptr<ExecutionMessage>
msg);

    int id_;
    zmq::context_t context_;
    ParentNodeInfo parent_node_;
    std::unordered_map<int, ChildNodeInfo> child_infos_;
};
```

Демонстрация работы программы

create 1 -1

Ok:10200

create 2 1

Ok:10205

create 3 1

Ok:10210

create 5 2

Ok:10215

exec 5 3 1 2 3

Ok:5:6

pingall

Ok: 1 2 3 5

exit

Вывод

В результате данной лабораторной работы я научился работать с технологией очереди сообщений на языке C++. Освоил базовые навыки работы с сокетом ZeroMQ.