МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №6-8

по курсу операционные системы I семестр, 2021/22 уч. год

Студент *Сикорский Александр Александрович, группа М8О-208Б-20*
Преподаватель *Миронов Евгений Сергеевич*
Вариант *2*
Оценка
Дата
Подпись

Содержание

Репозиторий

`https://github.com/sikorskii/os/tree/master/lab6`

Постановка задачи

Задание: Вариант 4:

Дерево общего вида, поиск подстроки в тексте, Heartbit.

Цель работы

Целью является приобретение практических навыков в:

1. Управлении серверами сообщений (№6)
2. Применение отложенных вычислений (№7)
3. Интеграция программных систем друг с другом (№8)

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

Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы

Исполнение команды на вычислительном узле

Формат команды: exec id [params]

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Формат вывода: «Ok:id: [result]», где result – результат выполненной команды

«Error:id: Not found» - вычислительный узел с таким идентификатором не найден

«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

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

Пример: Можно найти в описании конкретной команды, определенной вариантом задания. Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Набор команд 4 (поиск подстроки в строке)

Формат команды:

> exec id

> text$_s tring$

> pattern$_s tring$

[result] – номера позиций, где найден образец, разделенный точкой с запятой text$_s tring^-$, . : $[A - Za - z0 - 9]$.108

pattern$_s tring^-$

Пример:

> exec 10

> abracadabra

> abra

Ok:10:0;7

> exec 10

> abracadabra

> mmm

Ok:10: -1

Примечания: Выбор алгоритма поиска не важен

Общие сведения о программе

Исходный код лежит в 11 файлах, еще один - CMakeList:

1. main.cpp - файл, в котором происходит запуск приложения, с помощью вызова метода run у класса SpringBootApplication

2. server.cpp- код серверов(вычислительных узлов).
3. MessageData.h - класс, определяющий содержание сообщения.
4. ZmqUtils.h - класс со статическими утилитными методами.
5. MessageTypes.h - enum, содержащий типы сообщений.
6. MessageBuilder.h - класс, собирающий сообщения по требованию. В нем также содержатся средства сериализации-десериализации.
7. Message.h - класс сообщения.
8. AbstractNode.h - абстрактный узел.
9. SpringBootApplication.h - клиент(управляющий узел). Наследуется от абстрактного.
10. ServerNode.h - сервер(вычислительный узел). Наследуется от абстрактного.
11. ChildNodeInfo.h - класс, содержащий представление дочернего узла при хранении в мапе потомков данного.

Программа собирается с помощью Cmake. Нужно получитьдва исполняемых файла для клиента и серверов.

Общий метод и алгоритм решения

Имеем два исполняемых файла, но непосредственно работаем только с одним. Другой используется внутри вызова execl для создания дочернего вычислительного узла. Нам нужно организовать асинхронность. Для этого запускаем в клиенте несколько потоков конвеерообразно. Первый работает с пользователем, принимает от него команды и через push-pull сокет передает в поток, отвечающий за отправку сообщения и получение резельтата. И если данный поток может надолго заснуть в ожидании ответа от вычислительного узла, первый потенциально никогда не заблокируется надолго и сможет продолжать работать с пользователем, заполняя очередь для следующего потока. Кроме этого есть поток, отвечающий за вывод результатов запросов и поток, отвечающий за регистрацию свежесозданных узлов. В дереве общего вида мы не знаем, где расположена нода, на которую идет запрос и как её найти, в отличие от бинарного. Так что каждый узел, начиная с управляющего, при необходимости отправить сообщение проходит по всем дочерним узлам, подключается и отправляет сообщение. Далее узлы выполняют запрос, если он адресован им, или пробрасывают его дальше по дереву. Так запрос волной пройдет по всем узлам и точно найдет тот, которому он адресован, если он, конечно, доступен. Heartbit запускается по команде в отдельном потоке. В идеале узлы должны именно сообщать о том, что они живы. В моем же случае всё межпроцессное взаимодействие основано на req-rep сокетах, так что узел нужно спросить, жив ли он. Если узел не отвечает указанное время - считаем его мертвым.

Исходный код

main.cpp

```
#include <iostream>

#include "ServerNode.h"
#include <sys/wait.h>
#include "SpringBootApplication.h"

void child(int sig) {
    pid_t pid;
    pid = wait(nullptr);
}

int main() {
    signal(SIGCHLD, child);
    SpringBootApplication app;
    app.run();
}
```

server.cpp

```
//
// Created by aldes on 19.12.2021.
//

#include <zmq.hpp>
#include <unistd.h>
#include <iostream>
#include <signal.h>
#include "wait.h"
#include "SpringBootApplication.h"
#include "ServerNode.h"
void child(int sig) {
    pid_t pid;
    pid = wait(nullptr);
}

int main(int argc, char** argv) {
    signal(SIGCHLD, child);
    if (argc != 4) {
        return 0;
    }
```

```
        ServerNode serverNode(atoi(argv[0]), atoi(argv[1]), atoi(argv[2]), atoi(argv[3]))
        std::cout << "server created" << std::endl;
        serverNode.run();
        return 0;
}
```

MessageTypes.h

```
//
// Created by aldes on 15.12.2021.
//

#ifndef LAB6_MESSAGETYPES_H
#define LAB6_MESSAGETYPES_H


enum class MessageTypes : int {
    CREATE_REQUEST,
    CREATE_RESULT,
    CREATE_FAIL,
    EXEC_REQUEST,
    EXEC_RESULT,
    EXEC_FAIL,
    HEARTBIT_REQUEST,
    HEARTBIT_RESULT,
    HEARTBIT_FAIL,
    RELATE_RESULT,
    RELATE_REQUEST,
    QUIT,
    EMPTY,
    TEST,
};

#endif //LAB6_MESSAGETYPES_H
```

MessageData.h

```
//
// Created by aldes on 16.12.2021.
//

#ifndef LAB6_MESSAGEDATA_H
#define LAB6_MESSAGEDATA_H
```

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

class MessageData {
public:
    int id;
    int len1;
    int len2;
    std::vector<std::string> data;

};


#endif //LAB6_MESSAGEDATA_H
```

MessageBuider.h

```
//
// Created by aldes on 15.12.2021.
//

#ifndef LAB6_MESSAGEBUILDER_H
#define LAB6_MESSAGEBUILDER_H

#include <iostream>
#include "Message.h"
#include "MessageData.h"

class MessageBuilder {
public:
    static Message buildCreateMessage(int Id) {
        int data[2];
        std::cin >> data[0] >> data[1];
        std::cout << "add new node " << data[0] << " to parent " << data[1] << std::endl
        return {MessageTypes::CREATE_REQUEST, Id, Id, sizeof(data), (void*)&data};
    }

    static Message buildPingRequest(int time, int id) {
        return {MessageTypes::HEARTBIT_REQUEST, id, -1, sizeof(int), (void*)&time};
    }
```

```
static Message buildPingMessage(unsigned long long time, int Id) {
    return {MessageTypes::HEARTBIT_RESULT, Id, -1, sizeof(unsigned long long), (void
}


static Message buildTestMessage() {
    const char *testMessage = "first";
    return {MessageTypes::TEST, -1, -1, testMessage};
}

static Message buildExitMessage(int Id) {
    return {MessageTypes::QUIT, Id, Id};
}

static Message buildExecMessage() {
    int id;
    std::cin >> id;
    std::vector<std::string> data(2);
    std::cin >> data[0] >> data[1];
    std::cout << id << " " << data[0] << " " << data[1] << "\n";
    void* body = serialize(id, data);
    return {MessageTypes::EXEC_REQUEST, -1, id, getSize(data), body};
}

static int getSize(std::vector<std::string>& vector) {
    int size = 3 * sizeof(int);
    for (auto x : vector) {
        size += x.size() * sizeof(char);
    }
    return size;
}

static void* serialize(int& id, std::vector<std::string> &data) {
    void* body = malloc(getSize(data));
    memcpy(body, &id, sizeof(int));
    int size1 = data[0].size();
    int size2 = data[1].size();
    int offset = sizeof(int);
    memcpy((char*)body + offset, &size1, sizeof(int));
    offset += sizeof(int);
    memcpy((char*)body + offset, data[0].c_str(), size1 * sizeof(char));
    offset += size1 * sizeof(char);
```

```
        memcpy((char*) body + offset, &size2, sizeof(int));
        offset += sizeof(int);
        memcpy((char*) body + offset, data[1].c_str(), size2 * sizeof(char));
        return body;
    }

    static MessageData deserialize(void* messageBody) {
        MessageData data;
        data.id = *(int*)((char*)(messageBody));
        int offset = sizeof(int);
        int size1 = *(int*)((char*)messageBody + offset);
        void* str1 = malloc(size1);
        offset += sizeof(int);
        memcpy(str1, ((char*)messageBody + offset), size1 * sizeof(char));
        offset += size1 * sizeof(char);
        int size2 = *(int*)((char*)messageBody + offset);
        offset += sizeof(int);
        void* str2 = malloc(size2);
        memcpy(str2, ((char*)messageBody + offset), size2 * sizeof(char));
        //std::cout << data.id << " " << size1 << " "  << (char*)str1 << " " << size2 <<
        data.len1 = size1;
        data.len2 = size2;
        data.data.emplace_back((char*)str1);
        data.data.emplace_back((char*)str2);
        return data;
    }
};


#endif //LAB6_MESSAGEBUILDER_H
```

Message.h

```
//
// Created by aldes on 15.12.2021.
//

#ifndef LAB6_MESSAGE_H
#define LAB6_MESSAGE_H

#include "zmq.hpp"
#include <chrono>
#include <thread>
```

```
#include "MessageTypes.h"


class Message {
public:
    Message(MessageTypes messageType, int senderId, int receiverId, int _sizeOfBody, voi
            messageType), senderId(senderId), recieverId(receiverId), sizeOfBody(_sizeOf
        if (sizeOfBody > 0) {
            body = malloc(sizeOfBody);
            memcpy(body, _body, _sizeOfBody);
        }
    }

    Message(): Message(MessageTypes::EMPTY, -1, -1, 0, nullptr) {
    }

    Message(MessageTypes messageType, int senderId, int receiverId):
            Message(messageType, senderId, receiverId, 0, nullptr)
    {
    }
    ~Message() {
        if (body != nullptr) {
            free(body);
            body = nullptr;
        }
    }

    Message(Message&& message) noexcept {
        messageType = message.messageType;
        senderId = message.senderId;
        recieverId = message.recieverId;
        sizeOfBody = message.sizeOfBody;
        body = message.body;
        message.body = nullptr;
    }

    Message& operator= (Message&& message) noexcept {
        if (body != nullptr) {
            free(body);
            body = nullptr;
        }
```

```
    messageType = message.messageType;
    senderId = message.senderId;
    recieverId = message.recieverId;
    sizeOfBody = message.sizeOfBody;
    body = message.body;
    message.body = nullptr;
    return *this;
}
Message(MessageTypes messageType, int senderId, int receiverId, const char* str):
        Message(messageType, senderId, receiverId)
{
    sizeOfBody = strlen(str) + 1;
    if (sizeOfBody > 0) {
        body = operator new(sizeOfBody);
        memcpy(body, (void*)str, sizeOfBody);
    }
}

void sendMessage(zmq::socket_t& socket) {
    int header[] = {senderId, recieverId, sizeOfBody};
    zmq::message_t typeOfMessage(&messageType, sizeof(messageType));
    zmq::message_t messageHeader(&header, sizeof(header));
    zmq::message_t messageBody(body, sizeOfBody);
    //send compound message using SNDMORE flag in every message part except the last
    socket.send(typeOfMessage, zmq::send_flags::sndmore);
    socket.send(messageHeader, zmq::send_flags::sndmore);
    socket.send(messageBody, zmq::send_flags::none);
}

void receiveMessage(zmq::socket_t& socket) {
    zmq::message_t receivedType;
    zmq::message_t receivedHeader;
    zmq::message_t receivedBody;
    for (bool t = false; !t; t = getMessage(socket, receivedType, receivedHeader, re
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    messageType = *(MessageTypes*)receivedType.data();
    senderId = ((int*)receivedHeader.data())[0];
    recieverId = ((int*)receivedHeader.data())[1];
    sizeOfBody = ((int*)receivedHeader.data())[2];
    operator delete(body);
    body = nullptr;
```

```
    if (sizeOfBody > 0) {
        body = operator new(sizeOfBody);
        memcpy(body, receivedBody.data(), sizeOfBody);
    }
}

bool receiveMessage(zmq::socket_t& socket, std::chrono::milliseconds time) {
    zmq::message_t receivedType;
    zmq::message_t receivedHeader;
    zmq::message_t receivedBody;
    unsigned long long passed = 0;
    bool t = false;
    std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_c
    for ( ; !t && passed < time.count(); t = getMessage(socket, receivedType, receiv
        passed = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    if (!t)
        return false;
    messageType = *(MessageTypes*)receivedType.data();
    senderId = ((int*)receivedHeader.data())[0];
    recieverId = ((int*)receivedHeader.data())[1];
    sizeOfBody = ((int*)receivedHeader.data())[2];
    operator delete(body);
    body = nullptr;
    if (sizeOfBody > 0) {
        body = operator new(sizeOfBody);
        memcpy(body, receivedBody.data(), sizeOfBody);
    }
    return true;
}

void update(int senderID, int receiverId) {
    senderId = senderID;
    recieverId = receiverId;
}

MessageTypes messageType;
int senderId;
int recieverId;
int sizeOfBody;
void* body;
```

```
private:
    bool getMessage(zmq::socket_t& socket, zmq::message_t& typeOfMessage,
                    zmq::message_t& header, zmq::message_t& bodyOfMessage) {
        return (socket.recv(typeOfMessage, zmq::recv_flags::dontwait) &&
        socket.recv(header, zmq::recv_flags::dontwait) &&
        socket.recv(bodyOfMessage, zmq::recv_flags::dontwait));
    }
};


#endif //LAB6_MESSAGE_H
```

AbstractNode.h

```
//
// Created by aldes on 15.12.2021.
//

#ifndef LAB6_ABSTRACTNODE_H
#define LAB6_ABSTRACTNODE_H


#include <map>
#include <unistd.h>
#include "zmq.hpp"
#include "ChildNodeInfo.h"
#include <mutex>
#include <utility>
#include "ZmqUtils.h"

using std::to_string;

class AbstractNode {
public:
    explicit AbstractNode(int id) : Id(id), Port(ZmqUtils::PORT_TO_BIND_FROM),
                                    outerNodes(), context(1), Receiver(context, zmq::soc
        occupyPort();
    }

    AbstractNode() {}

    pid_t addChild(int id, int registerPort) {
        pid_t child = fork();
```

```
        if (child == 0) {
            execl("server", to_string(id).c_str(), to_string(Id).c_str(),
                  to_string(Port).c_str(), to_string(registerPort).c_str(), NULL);
        }
        return child;
    }

protected:
    int Id;
    int Port;
    std::map<int, ChildNodeInfo> outerNodes;
    zmq::context_t context;
    zmq::socket_t Receiver;

    void occupyPort() {
        Port = ZmqUtils::occupyPort(Receiver);
    }
};


#endif //LAB6_ABSTRACTNODE_H
```

SpringBootApplication.h

```
//
// Created by aldes on 15.12.2021.
//

#ifndef LAB6_SPRINGBOOTAPPLICATION_H
#define LAB6_SPRINGBOOTAPPLICATION_H


#include <set>
#include <mutex>
#include <condition_variable>

#include "ServerNode.h"
#include "AbstractNode.h"
#include "Message.h"
#include "ZmqUtils.h"
#include "MessageBuilder.h"
```

```
class SpringBootApplication : public AbstractNode{
public:

    SpringBootApplication() : AbstractNode(-1), created(true) {
        serversId.insert(-1);
    }

    void run() {
        std::thread([this]() {inputProcessing();}).detach();
        std::thread([this]() {messageProcessing();}).detach();
        std::thread([this]() {registrator();}).detach();
        messageOutput();
    }

private:
    std::set<int> serversId;
    std::mutex creationLock;
    std::condition_variable creationalCondition;
    bool created;

    void inputProcessing() {
        std::string command;
        zmq::socket_t toMessageProcessor(context, zmq::socket_type::push);
        toMessageProcessor.connect(ZmqUtils::MESSAGE_PROCESSOR_URL);

        while (true) {
            std::cin >> command;
            if (command == "create") {
                Message message = MessageBuilder::buildCreateMessage(Id);
                message.sendMessage(toMessageProcessor);
            }
            if (command == "exit") {
                Message message = MessageBuilder::buildExitMessage(Id);
                message.sendMessage(toMessageProcessor);
                break;
            }
            if (command == "exec") {
                Message message = MessageBuilder::buildExecMessage();
                message.sendMessage(toMessageProcessor);
            }
            if (command == "hb") {
                int time;
```

```
            std::cin >> time;
            Message message = MessageBuilder::buildPingRequest(time, -1);
            message.sendMessage(toMessageProcessor);
        }
        else {
            std::cin.clear();
        }

    }

}

void messageProcessing() {
    zmq::socket_t fromInputProcessor(context, zmq::socket_type::pull);
    zmq::socket_t toOutput(context, zmq::socket_type::push);
    fromInputProcessor.bind(ZmqUtils::MESSAGE_PROCESSOR_URL);
    toOutput.connect(ZmqUtils::MESSAGE_SENDER_URL);
    MessageData data;
    std::thread pl;
    Message toSend;
    bool quit = false;
    while(!quit) {
        Message receivedMessage;
        receivedMessage.receiveMessage(fromInputProcessor);
        int time;
        std::thread th;
        switch (receivedMessage.messageType) {
            case MessageTypes::CREATE_REQUEST:
                toSend = processCreateMessage(std::move(receivedMessage));
                toSend.sendMessage(toOutput);
                break;
            case MessageTypes::EXEC_REQUEST:
                toSend = processExecMessage(std::move(receivedMessage));
                toSend.sendMessage(toOutput);
                break;
            case MessageTypes::HEARTBIT_REQUEST:
                time = *(int*)receivedMessage.body;
                pl = std::thread([this, &time, &quit]() { pingListener(*this, time,
                pl.detach();
                processPingRequest(time);
                break;
            case MessageTypes::QUIT:
```

```
                toSend = exitProcessor(std::move(receivedMessage), quit);
                quit = true;
                toSend.sendMessage(toOutput);
                return;
            default:
                break;
        }
    }
}

static void pingListener(SpringBootApplication& that, int& time, bool& exit) {
    while(!exit) {
        for (auto server: that.outerNodes) {
            zmq::socket_t sender(that.context, zmq::socket_type::req);
            sender.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
            Message pingRequest = MessageBuilder::buildPingRequest(time, server.firs
            try {
                pingRequest.sendMessage(sender);
                if (!pingRequest.receiveMessage(sender, std::chrono::milliseconds(ti
                    std::cout << "node " << server.first << " and all her children d
                    //that.outerNodes.erase(server.first);
                }
            }
            catch (const zmq::error_t& error) {
                continue;
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(time));
        }
    }
}

void processPingRequest(int time) {
    for (auto server: outerNodes) {
        Message pingRequest = MessageBuilder::buildPingRequest(time, Id);
        zmq::socket_t socket(context, zmq::socket_type::req);
        socket.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
        pingRequest.sendMessage(socket);
        pingRequest.receiveMessage(socket, std::chrono::milliseconds(100));
        //std::cout << "message sent to node " << server.first << std::endl;
        socket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
    }
}
```

```
void messageOutput() {
    zmq::socket_t fromMessageProcessor(context, zmq::socket_type::pull);
    fromMessageProcessor.bind(ZmqUtils::MESSAGE_SENDER_URL);
    Message receivedMessage;
    bool quit{false};
    while (!quit) {
        receivedMessage.receiveMessage(fromMessageProcessor);
        switch (receivedMessage.messageType) {
            case MessageTypes::CREATE_RESULT:
                createOutput(receivedMessage);
                break;
            case MessageTypes::EXEC_RESULT:
                execOutput(receivedMessage);
                break;
            case MessageTypes::QUIT:
                std::cout << "EXIT" << std::endl;
                quit = true;
                return;

            default:
                std::cout << "another message\n";
                break;
        }
    }
}



void ready() {
    std::unique_lock<std::mutex> lock(creationLock);
    created = true;
    creationalCondition.notify_one();
}

void registrator() {
    Message messageIn;
    Message messageOut;
    while (true) {
        messageIn.receiveMessage(Receiver);
        if (messageIn.messageType == MessageTypes::QUIT) {
            messageIn.sendMessage(Receiver);
```

```
            break;
        }
        messageOut = processRegisterMessage(messageIn);
        ready();
        messageOut.sendMessage(Receiver);

    }
}


Message processRegisterMessage(Message& message) {
    auto body = (char*)message.body;
    pid_t pid = *(pid_t*)(body);
    int recPort = *(int*)(body + sizeof(pid_t));
    int regPort = *(int*)(body + sizeof(pid_t) + sizeof(int));
    outerNodes.emplace(message.senderId, ChildNodeInfo(pid, recPort, regPort));
    return {MessageTypes::RELATE_RESULT, Id, message.senderId};
}

Message processCreateMessage(Message&& message) {
    int id = ((int*)message.body)[0];
    int parentId = ((int*)message.body)[1];
    return create(id, parentId);
}

Message create(int id, int pId) {
    if (serversId.find(id) != serversId.end()) {
        std::cout << "already exist" << std::endl;
        return {MessageTypes::CREATE_FAIL, Id, Id, "Already exist"};
    }
    if (serversId.find(pId) == serversId.end()) {
        std::cout << "parent not found" << std::endl;
        return {MessageTypes::CREATE_FAIL, Id, Id, "Parent not found"};
    }
    if (pId == Id) {
        pid_t pid = addChild(id, Port);
        std::cout << "received Pid: " << pid << std::endl;
        serversId.insert(id);
        std::unique_lock<std::mutex> lock(creationLock);
        created = false;
        while(!created) {
            creationalCondition.wait(lock);
```

```
            }
            std::cout << "awaited" << std::endl;
            return {MessageTypes::CREATE_RESULT, Id, Id, sizeof(pid), (void*)&pid};
        }
        for (auto server: outerNodes) {
            zmq::socket_t requestSocket(context, zmq::socket_type::req);
            requestSocket.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort)
            int data[] = {id, pId};
            Message request(MessageTypes::CREATE_REQUEST, Id, server.first, sizeof(data)
            Message result;
            request.sendMessage(requestSocket);
            //result.receiveMessage(requestSocket);
            if (!result.receiveMessage(requestSocket, std::chrono::milliseconds(2000)))
                requestSocket.disconnect(ZmqUtils::getOutputAddress(server.second.Receiv
                continue;
            }
            if (result.messageType == MessageTypes::CREATE_RESULT) {
                requestSocket.disconnect(ZmqUtils::getOutputAddress(server.second.Receiv
                serversId.insert(id);
                return {MessageTypes::CREATE_RESULT, Id, Id, result.sizeOfBody, result.b
            }
            requestSocket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPo
        }
        return {MessageTypes::CREATE_FAIL, Id, Id, "Parent is uavailable"};
    }

    Message processExecMessage(Message&& message) {
        MessageData data = MessageBuilder::deserialize(message.body);
        if (serversId.find(data.id) == serversId.end()) {
            std::cout << "id " << data.id << " not found" << std::endl;
            return {MessageTypes::EXEC_FAIL, Id, Id, "Node not found"};
        }
        for (auto server : outerNodes) {
            zmq::socket_t request(context, zmq::socket_type::req);
            request.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
            message.senderId = Id;
            message.recieverId = server.first;
            message.sendMessage(request);
            Message result;
            //result.receiveMessage(request);
            if (!result.receiveMessage(request, std::chrono::milliseconds(2000))) {
                request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort
```

```
            continue;
        }
        if (result.messageType == MessageTypes::EXEC_RESULT) {
            result.update(Id, Id);
            return result;
        }
        request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
    }
    std::cout << "node unaviable" << std::endl;
    return Message(MessageTypes::EXEC_FAIL, Id, Id, "node unaviable");
}

Message exitProcessor(Message&& message, bool& exit) {
    zmq::socket_t exitRegister(context, zmq::socket_type::req);
    exitRegister.connect(ZmqUtils::getOutputAddress(Port));
    Message messageExit(MessageTypes::QUIT, Id, Id);
    Message temporary;
    messageExit.sendMessage(exitRegister);
    temporary.receiveMessage(exitRegister);
    exitRegister.disconnect(ZmqUtils::getOutputAddress(Port));
    std::cout << "closing begins" << std::endl;
    exit = true;
    for (auto server: outerNodes) {
        zmq::socket_t request(context, zmq::socket_type::req);
        request.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
        messageExit.sendMessage(request);
        messageExit.receiveMessage(request, std::chrono::milliseconds(2000));
        //messageExit.sendMessage(Receiver);
        Message exitResult;
        //messageExit.receiveMessage(Receiver, std::chrono::milliseconds(1000));
        //exitResult.receiveMessage(request);
        request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
    }
    std::cout << "servers dead" << std::endl;
    return messageExit;
}

void createOutput(Message& message) {
    std::cout << "Ok:" << *(int*)message.body << std::endl;
}

void execOutput(Message& message) {
```

```
        MessageData data = MessageBuilder::deserialize(message.body);
        std::cout << "Exec Ok: " << data.id << " " << data.data[0] << " " << data.data[1
    }
};


#endif //LAB6_SPRINGBOOTAPPLICATION_H
```

ServerNode.h

```
//
// Created by aldes on 15.12.2021.
//

#ifndef LAB6_SERVERNODE_H
#define LAB6_SERVERNODE_H


#include "SpringBootApplication.h"
#include "AbstractNode.h"
#include "Message.h"
#include <chrono>
#include "MessageData.h"
#include "MessageBuilder.h"

class ServerNode : public AbstractNode{
public:
    ServerNode(int id, int _parentId, int _parentPort, int _registerPort) :
            AbstractNode(id), parentId(_parentId), parentPort(_parentPort), parentRegPor
            registerSocket(context, zmq::socket_type::rep) {
        occupyPort();
        registerPort = ZmqUtils::occupyPort(registerSocket);
        selfRelation(parentRegPort);
    }

    void run() {
        std::cout << "SERVER CREATED" << std::endl;
        //std::cout << "port " << Port << " parent port " << parentPort << std::endl;
        std::thread([this]() {registrator();}).detach();
        bool quit = false;
        bool ping = false;
        while (true) {
            Message messageOut;
```

```
            Message messageIn;
            //std::cout << "before recieving" << std::endl;
            if (!messageIn.receiveMessage(Receiver, std::chrono::milliseconds(100))) {
                continue;
            }
            //std::cout << "after recevinge" << std::endl;
            int time;
            std::thread th;
            switch (messageIn.messageType) {
                case MessageTypes::CREATE_REQUEST:
                    messageOut = createProcessor(messageIn);
                    messageOut.sendMessage(Receiver);
                    break;
                case MessageTypes::EXEC_REQUEST:
                    messageOut = execProcessor(messageIn);
                    messageOut.sendMessage(Receiver);
                    break;
                case MessageTypes::HEARTBIT_REQUEST:
                    //std::cout << "Heartbit request arrived at server " << Id << std::e
                    time = *(int*)messageIn.body;
                    processPingRequest(time);
                    if (!ping) {
                        ping = true;
                        th = std::thread([this, &time, &quit]() { bitGenerator(*this, ti
                        th.detach();
                    }
                    messageIn.sendMessage(Receiver);
                    break;
                case MessageTypes::QUIT:
                    messageOut = quitProcessor(messageIn);
                    messageOut.sendMessage(Receiver);
                    quit = true;
                    exit(1);
                default:
                    break;
            }
        }
    }
private:
    int parentId;
    int parentPort;
    int parentRegPort;
```

```
zmq::socket_t registerSocket;
int registerPort;
std::mutex creationalBlock;
std::condition_variable creationalCondition;
bool created;

static void bitGenerator(ServerNode& that, int& time, bool& exit) {
    while (!exit) {
        for (auto server: that.outerNodes) {
            zmq::socket_t sender(that.context, zmq::socket_type::req);
            sender.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
            Message pingRequest = MessageBuilder::buildPingRequest(time, server.firs
            try {
                pingRequest.sendMessage(sender);
                if (!pingRequest.receiveMessage(sender, std::chrono::milliseconds(ti
                    std::cout << "node " << server.first << " and all her children d
                    //that.outerNodes.erase(server.first);
                }
            }
            catch (const zmq::error_t& error) {
                continue;
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(time));
        }
    }
}

void processPingRequest(int time) {
    for (auto server: outerNodes) {
        Message pingRequest = MessageBuilder::buildPingRequest(time, Id);
        zmq::socket_t socket(context, zmq::socket_type::req);
        socket.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
        pingRequest.sendMessage(socket);
        pingRequest.receiveMessage(socket, std::chrono::milliseconds(1000));
        //std::cout << "message sent to node " << server.first << std::endl;
        socket.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
    }
}
void registrator() {
    Message messageIn;
    std::cout << "from registrator " << getpid() << std::endl;
    Message messageOut;
```

```
    while (true) {
        messageIn.receiveMessage(registerSocket);
        std::cout << "register message got" << std::endl;
        if (messageIn.messageType == MessageTypes::QUIT) {
            messageIn.sendMessage(registerSocket);
            break;
        }
        messageOut = registerProcessor(messageIn);
        ready();
        messageOut.sendMessage(registerSocket);
    }
}


void ready() {
    std::unique_lock<std::mutex> lock(creationalBlock);
    created = true;
    creationalCondition.notify_one();
}

void* createRelation(pid_t pid, int receiverPort, int registerPort) {
    auto mem = new char[sizeof(pid) + 2 * sizeof(int)];
    auto memPid = (pid_t*)(mem);
    auto memReceiverPort = (int*)(mem + sizeof(pid_t));
    auto memRegisterPort = (int*)(mem + sizeof(pid_t) + sizeof(int));
    *memPid = pid;
    *memReceiverPort = receiverPort;
    *memRegisterPort = registerPort;
    return (void*)mem;
}

void selfRelation(int _registerPort) {
    zmq::socket_t sender(context, zmq::socket_type::req);
    sender.connect(ZmqUtils::getOutputAddress(_registerPort));
    void* relationBody = createRelation(getpid(), Port, registerPort);
    Message message(MessageTypes::RELATE_REQUEST, Id, parentId, 2 * sizeof(int) + si
    message.sendMessage(sender);
    message.receiveMessage(sender);
    sender.disconnect(ZmqUtils::getOutputAddress(_registerPort));
    delete [](char*)relationBody;
    relationBody = nullptr;
}
```

```
Message registerProcessor(Message& message) {
    auto messageBody = (char*)message.body;
    pid_t pid = *(pid_t*)(messageBody);
    int receiverPort = *(int*)(messageBody + sizeof(pid_t));
    int registerPort = *(int*)(messageBody + sizeof(pid_t) + sizeof(int));
    outerNodes.emplace(message.senderId, ChildNodeInfo(pid, receiverPort, registerPo
    return Message(MessageTypes::RELATE_RESULT, Id, message.senderId);
}

Message createProcessor(Message& message) {
    int id = ((int*)message.body)[0];
    int parId = ((int*)message.body)[1];
    int sndId = message.senderId;
    if (parId == Id) {
        pid_t pid = addChild(id, registerPort);
        std::unique_lock<std::mutex> lock(creationalBlock);
        created = false;
        while (!created) {
            creationalCondition.wait(lock);
        }
        return {MessageTypes::CREATE_RESULT, Id, parentId, sizeof(pid), &pid};
    }
    else if (outerNodes.empty()) {
        return {MessageTypes::CREATE_FAIL, Id, parentId};
    }
    int pid = -1;
    for (auto server: outerNodes) {
        zmq::socket_t sender(context, zmq::socket_type::req);
        sender.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
        int data[] = {id, parId};
        Message request(MessageTypes::CREATE_REQUEST, Id, server.first, sizeof(data)
        Message result;
        request.sendMessage(sender);
        //result.receiveMessage(sender);
        if (!result.receiveMessage(sender, std::chrono::milliseconds(2000))) {
            sender.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort)
            continue;
        }
        if (result.messageType == MessageTypes::CREATE_RESULT) {
            pid = *(int*)result.body;
            sender.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort)
```

```
            break;
        }
        sender.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
    }
    if (pid == -1) {
        return {MessageTypes::CREATE_FAIL, Id, sndId, sizeof(pid), &pid};
    }
    return {MessageTypes::CREATE_RESULT, Id, sndId, sizeof(pid), &pid};
}

Message execProcessor(Message& message) {
    MessageData data = MessageBuilder::deserialize(message.body);
    std::string text = data.data[0];
    std::string pattern = data.data[1];
    //std::cout << "EXEC INFO: Id " << Id << "target id " << data.id << std::endl;
    if (data.id == Id) {
        text = text.substr(0, data.len1);
        pattern = pattern.substr(0, data.len2);
        size_t found = text.find(pattern, 0);
        std::string answer;
        if (found != std::string::npos) {
            while (found != std::string::npos) {
                answer += std::to_string(found);
                answer += " ";
                found = text.find(pattern, found + 1);
            }
            std::cout << answer << std::endl;
        }
        else {
            std::cout << "NOT FOUND!" << std::endl;
        }
        std::vector<std::string> newData(2);
        newData[0] = answer;
        newData[1] = "";
        void* newBody = MessageBuilder::serialize(data.id, newData);
        int newSize = MessageBuilder::getSize(newData);
        std::cout << "RECEVIEd!! " << text << " " << pattern << std::endl;
        return {MessageTypes::EXEC_RESULT, Id, parentId, newSize, newBody};
    }
    for (auto server : this->outerNodes) {
        zmq::socket_t requester(context, zmq::socket_type::req);
        requester.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
```

```
            message.update(Id, server.first);
            message.sendMessage(requester);
            Message result;
            //result.receiveMessage(requester);
            if (!result.receiveMessage(requester, std::chrono::milliseconds(2000))) {
                requester.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPo
                continue;
            }
            if (result.messageType == MessageTypes::EXEC_RESULT) {
                result.update(Id, parentId);
                requester.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPo
                return result;
            }
            requester.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort))
        }
        return {MessageTypes::EXEC_FAIL, Id, parentId, "node is unaviable"};
    }

    Message quitProcessor(Message& message) {
        Message messageQuit(MessageTypes::QUIT, Id, Id);
        for (auto server: outerNodes) {
            zmq::socket_t request(context, zmq::socket_type::req);
            request.connect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
            messageQuit.sendMessage(request);
            Message messageResult;
            //messageResult.receiveMessage(request);
            if (!messageResult.receiveMessage(request, std::chrono::milliseconds(2000)))
                request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort
                continue;
            }
            request.disconnect(ZmqUtils::getOutputAddress(server.second.ReceiverPort));
        }
        zmq::socket_t request(context, zmq::socket_type::req);
        request.connect(ZmqUtils::getOutputAddress(registerPort));
        messageQuit.sendMessage(request);
        //messageQuit.receiveMessage(request);
        messageQuit.receiveMessage(request, std::chrono::milliseconds(2000));
        request.disconnect(ZmqUtils::getOutputAddress(registerPort));
        return messageQuit;
    }

};
```

```
#endif //LAB6_SERVERNODE_H
```

Демонстрация работы программы

Ниже приведен пример работы программы.

test.txt

```
aldes@aldes:~/dev/MAI-labs/OS/lab6/cmake-build-debug$ ./lab6
>create 1 -1
add new node 1 to parent -1
received Pid: 10491
awaited
server created
SERVER CREATED
from registrator 10491
Ok:10491
>exec 1 labalaba
>lab
1 labalaba lab
0 4
RECEVIEd!! labalaba lab
Exec Ok: 1 0 4
>create 2 1
add new node 2 to parent 1
register message got
server created
SERVER CREATED
from registrator 10502
Ok:10502
>create 3 1
add new node 3 to parent 1
register message got
server created
SERVER CREATED
from registrator 10506
Ok:10506
>create 4 1
add new node 4 to parent 1
register message got
server created
```

```
SERVER CREATED
from registrator 10512
Ok:10512
>exec 4 asdfgdshgdjghfkhjgdfsgdfafsghdj fgh
4 asdfgdshgdjghfkhjgdfsgdfafsghdj fgh
NOT FOUND!
RECEVIEd!! asdfgdshgdjghfkhjgdfsgdfafsghdj fgh
Exec Ok: 4
>exit
closing begins
register message got
register message got
register message got
register message got
servers dead
EXIT
```

Выводы

В ходе работы я познакомился с тем, как можно построить распределенную асинхронную систему с помощью очередей сообщений. Можно добавить, что программа сделана таким образом, что при убийстве какого-нибудь вычислительного узла его потомки само собой станут недоступны, но мы сможем создавать новые узлы и работать уже с созданными и дальше.