Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №5-7 по курсу**

**«Операционные системы»**

Группа: М8О-203Б-23

Студент: Никитцев А.В.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 25.12.24

Москва, 2024

**Цель работы**

Целью является приобретение практических навыков в:

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

**Постановка задачи (Вариант 46)**

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.*

Топология 2 – дерево общего вида

Команда 2 - локальный целочисленный словарь

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name

Пример:

> exec 10 MyVar

Ok:10: 'MyVar' not found

> exec 10 MyVar 5

Ok:10

> exec 12 MyVar

Ok:12: 'MyVar' not found

> exec 10 MyVar

Ok:10: 5

> exec 10 MyVar 7

Ok:10

> exec 10 MyVar

Ok:10: 7

*Примечания: Можно использовать std:map.*

Тип проверки – 2

Формат команды: ping id

Команда проверяет доступность конкретного узла. Если узла нет, то необходимо выводить

ошибку: «Error: Not found»

Пример:

> ping 10

Ok: 1 // узел 10 доступен

> ping 17

Ok: 0 // узел 17 недоступен

**Код**

**Constants.h**

*//  
// Created by nikit on 12/24/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_CONSTANTS\_H\_  
#define OS\_LAB\_5\_SRC\_CONSTANTS\_H\_  
  
using namespace std::chrono\_literals;  
  
const inline std::string WORKER\_PATH = "./child";  
constexpr int basePort = 4000;  
const inline std::string baseAddress = "tcp://127.0.0.1:";  
constexpr std::chrono::duration TIMEOUT = 5s;  
  
#endif *//OS\_LAB\_5\_SRC\_CONSTANTS\_H\_*

**Net.h**

*//  
// Created by nikit on 12/24/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_NET\_H\_  
#define OS\_LAB\_5\_SRC\_NET\_H\_  
  
#include <iostream>  
#include "../../cmake-build-debug/\_deps/cppzmq-src/zmq.hpp"  
#include "constants.h"  
  
namespace net {  
 inline int bind(zmq::socket\_t \*socket, int id) {  
 int port = basePort + id;  
 while (true) {  
 std::string adress = baseAddress + std::to\_string(port);  
 try {  
 socket->bind(adress);  
 break;  
 } catch (std::exception &e) {  
 port++;  
 }  
 }  
 return port;  
 }  
  
 inline void connect(zmq::socket\_t \*socket, int port) {  
 std::string adress = baseAddress + std::to\_string(port);  
 socket->connect(adress);  
 }  
  
inline void unbind(zmq::socket\_t \*socket, int port) {  
 std::string adress = baseAddress + std::to\_string(port);  
 socket->unbind(adress);  
 }  
  
inline void disconnect(zmq::socket\_t \*socket, int port) {  
 std::string adress = baseAddress + std::to\_string(port);  
 socket->disconnect(adress);  
 }  
  
inline void send\_message(zmq::socket\_t \*socket, const std::string msg) {  
 zmq::message\_t message(msg.size());  
 memcpy(message.data(), msg.c\_str(), msg.size());  
 try {  
 socket->send(message);  
 } catch (...) {}  
 }  
  
inline std::string receive(zmq::socket\_t \*socket) {  
 zmq::message\_t message;  
 bool success = true;  
 try {  
 socket->recv(&message, 0);  
 } catch (...) {  
 success = false;  
 }  
 if (!success || message.empty()) {  
 throw std::runtime\_error("Error while receiving message");  
 }  
 std::string str(static\_cast<char \*>(message.data()), message.size());  
 return str;  
 }  
}  
  
#endif *//OS\_LAB\_5\_SRC\_NET\_H\_*

**Request.h**

*//  
// Created by nikit on 12/24/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_REQUEST\_H\_  
#define OS\_LAB\_5\_SRC\_REQUEST\_H\_  
  
#include <string>  
#include <utility>  
#include <vector>  
#include "Utils.h"  
namespace Request {  
  
class Request {  
 size\_t id;  
 std::string command;  
 std::vector<std::string> args;  
 int receiver;  
  
 public:  
 Request() : command("UNKNOWN"), args(std::vector<std::string>()), receiver(-1), id(0) {}  
 Request(std::string command, std::vector<std::string> args, int receiver, size\_t id) : command(std::move(command)), args(std::move(args)), receiver(receiver), id(id) {}  
 std::string getCommand() { return command; }  
 std::vector<std::string> getArgs() { return args; }  
 int getReceiver() const { return receiver; }  
 size\_t getId() const { return id; }  
 static Request fromStringRequest(const std::string& data) {  
 std::vector<std::string> parts = split(data, ' ');  
 if (parts.size() < 3) throw std::invalid\_argument("Invalid request format");  
 return {parts[1], parts.size() == 3 ? std::vector<std::string>() : std::vector<std::string>(parts.begin() + 3, parts.end()), fromString<int>(parts[2]), fromString<size\_t>(parts[0])};  
 }  
 static Request fromStringRequest(const std::string& data, const size\_t id) {  
 std::vector<std::string> parts = split(data, ' ');  
 if (parts.size() < 2) throw std::invalid\_argument("Invalid request format");  
 return {parts[0], parts.size() == 2 ? std::vector<std::string>() : std::vector<std::string>(parts.begin() + 2, parts.end()), fromString<int>(parts[1]), id};  
 }  
  
 std::string toString() {  
 std::string result = std::to\_string(id) + " " + command + " " + std::to\_string(receiver);  
 for (const auto& arg : args) result += " " + arg;  
 return result;  
 }  
};  
  
} *// Request*#endif *//OS\_LAB\_5\_SRC\_REQUEST\_H\_*

**Response.cpp**

#include <utility>  
#include <string>  
#include <vector>  
#include "Utils.h"  
#include "Request.h"  
  
*//  
// Created by nikit on 12/24/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_RESPONSE\_H\_  
#define OS\_LAB\_5\_SRC\_RESPONSE\_H\_  
  
inline std::string concat(const std::vector<std::string>& data) {  
 std::string result;  
 for (const auto &item : data) result += item + " ";  
 if (!result.empty()) {  
 result.pop\_back();  
 }  
 return result;  
}  
  
namespace Response {  
  
enum ResponseType {  
 SUCCESS,  
 FAILURE  
};  
  
class Response {  
 ResponseType type;  
 std::string message;  
 int handlerId{};  
 Request::Request request;  
  
 public:  
 Response() : type(FAILURE), message("Unknown error"), handlerId(-1), request() {};  
 Response(ResponseType type, std::string message, int handlerId) : type(type), message(std::move(message)), handlerId(handlerId), request() {}  
 std::string constructResponseString() {  
 std::string stringType = type == SUCCESS ? "SUCCESS" : "FAILURE";  
 return request.toString() + "\n" + stringType + " " + std::to\_string(handlerId) + " " + message;  
  
 }  
 static Response fromStringResponse(const std::string& data) {  
 std::vector<std::string> rows = split(data, '\n');  
 Request::Request req = Request::Request::fromStringRequest(rows[0]);  
 std::vector<std::string> parts = split(rows[1], ' ');  
 if (parts.size() < 3) throw std::invalid\_argument("Invalid response format");  
 auto response = Response(  
 parts[0] == "SUCCESS" ? SUCCESS : FAILURE,  
 concat(std::vector<std::string>(parts.begin() + 2, parts.end())),  
 fromString<int>(parts[1])  
 );  
 response.setRequest(req);  
 return response;  
 }  
 static Response generateMalformedResponse(int handlerId) {  
 return {FAILURE, "Message malformed", handlerId};  
 }  
 static Response generateUnknownCommandResponse(int handlerId) {  
 return {FAILURE, "Unknown command", handlerId};  
 }  
  
 Request::Request getRequest() const { return request; }  
 ResponseType getType() const { return type; }  
 std::string getMessage() const { return message; }  
 int getHandlerId() const { return handlerId; }  
  
 void setRequest(const Request::Request& req) { Response::request = req; }  
};  
  
} *// Response*#endif *//OS\_LAB\_5\_SRC\_RESPONSE\_H\_*

**Tree.h**

*//  
// Created by nikit on 12/25/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_TREE\_H\_  
#define OS\_LAB\_5\_SRC\_TREE\_H\_  
  
#include "iostream"  
#include "vector"  
  
#include <memory>  
  
namespace Tree {  
  
template<class T>  
class Node {  
 T data;  
 std::vector<std::shared\_ptr<Node<T>>> children;  
  
  
 public:  
 explicit Node(const T& data) : data(data), children() {}  
 Node(const T& data, const std::shared\_ptr<Node<T>>& parent) : data(data), children() {}  
 void attachNode(const T& nodeToAttachTo, const T& value){  
 if (data != nodeToAttachTo) {  
 for (std::shared\_ptr<Node<T>> child: children){  
 child->attachNode(nodeToAttachTo, value);  
 }  
 return;  
 }  
 children.push\_back(std::make\_shared<Node<T>>(value));  
 }  
  
 bool exists(const T& d){  
 if (data == d){  
 return true;  
 }  
 bool result = false;  
 for (std::shared\_ptr<Node<T>> child: children){  
 result |= child->exists(d);  
 }  
 return result;  
 }  
  
 void detachNode(const T& d) {  
 std::shared\_ptr<Node<T>> value = nullptr;  
 for (auto i = 0; i < children.size(); ++i){  
 if (children[i]->getData() == d){  
 value = children[i];  
 break;  
 }  
 }  
  
 if (value != nullptr) children.erase(std::remove(children.begin(), children.end(),value), children.end());  
 for (std::shared\_ptr<Node<T>> child: children){  
 child->detachNode(d);  
 }  
 };  
  
 T getData() { return data; }  
 std::vector<std::shared\_ptr<Node<T>>> getChildren() { return children; }  
 ~Node() {  
 std::cout << "Deleting node with value " << data << std::endl;  
 };  
};  
  
template<class T>  
class Tree {  
 std::shared\_ptr<Node<T>> root;  
 public:  
 explicit Tree(const T& data): root(std::make\_shared<Node<T>>(data)){}  
 std::shared\_ptr<Node<T>> getRoot() { return root; }  
 void remove(const T& data){  
 root->detachNode(data);  
 }  
 void attach(const T& nodeToAttachTo, const T& value){  
 root->attachNode(nodeToAttachTo, value);  
 }  
 bool exists(const T& data){  
 return root->exists(data);  
 }  
 ~Tree() {  
 std::cout << "Deleting tree..." << std::endl;  
 };  
};  
  
} *// Tree*#endif *//OS\_LAB\_5\_SRC\_TREE\_H\_*

**Utils.h**

*//  
// Created by nikit on 12/24/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_UTILS\_H\_  
#define OS\_LAB\_5\_SRC\_UTILS\_H\_  
  
#include <vector>  
#include <string>  
#include <sstream>  
#include <csignal>  
  
inline std::vector<std::string> split(const std::string &data, char splitter) {  
 std::istringstream iss(data);  
 std::string buffer;  
 std::vector<std::string> result;  
 while (getline(iss, buffer, splitter)){  
 result.push\_back(buffer);  
 }  
 return result;  
}  
  
template<class T>  
inline T fromString(const std::string& data){  
 std::istringstream iss(data);  
 T result;  
 iss >> result;  
 return result;  
}  
  
inline int startProcess(const std::string &childPath, const size\_t id, const int parentPortInput, const int parentPortOutput) {  
 pid\_t pid = fork();  
  
 if (pid == -1){  
 return -1;  
 }  
  
 if (pid == 0){  
 if (execl(childPath.c\_str(), std::to\_string(id).c\_str(), std::to\_string(parentPortInput).c\_str(), std::to\_string(parentPortOutput).c\_str(), nullptr) == -1){  
 return -1;  
 }  
 }  
  
 return pid;  
}  
  
#endif *//OS\_LAB\_5\_SRC\_UTILS\_H\_*

**Worker.cpp**

*//  
// Created by nikit on 12/24/2024.  
//*#include <string>  
#include <iostream>  
#include <vector>  
#include <variant>  
#include <thread>  
#include "Worker.h"  
#include "Utils.h"  
#include "constants.h"  
#include "Request.h"  
  
namespace Worker {  
 Response::Response Worker::handleExec(const std::vector<std::string>& args) {  
 if (args.size() == 2) {  
 if (data.contains(args[0])) data[args[0]] = fromString<int>(args[1]);  
 else data.insert({args[0], fromString<int>(args[1])});  
 return {Response::ResponseType::*SUCCESS*, args[1], id};  
 }  
 if (args.size() != 1) return Response::Response::generateMalformedResponse(id);  
 if (data.contains(args[0])) return {Response::ResponseType::*SUCCESS*, std::to\_string(data[args[0]]), id};  
 return {Response::ResponseType::*FAILURE*, args[0] + " key does not exist", id};  
 }  
  
 std::variant<Response::Response, Option> Worker::handleCommands(  
 const std::string& command,  
 const int receiver,  
 const std::vector<std::string>& args,  
 const size\_t requestId  
 ) {  
 if (command == "kill") {  
 if (receiver == -1 && id == -1) return Option::*BREAK*;  
 if (receiver != -2) {  
 if (!children.contains(receiver)) return Option::*PASS\_MESSAGE*;  
 return handleKill(receiver, requestId);  
 }  
 }  
  
 if (receiver != id) {  
 return Option::*PASS\_MESSAGE*;  
 }  
  
 if (command == "create") {  
 return handleCreate(args);  
 } else if (command == "exec") {  
 return handleExec(args);  
 } else if (command == "ping"){  
 return handlePing();  
 } else if (command == "kill") {  
 return handleKill(receiver, requestId);  
 }  
  
 std::cout << "Unknown command" << std::endl;  
 return Response::Response::generateUnknownCommandResponse(id);  
 }  
  
 void Worker::run(const std::function<Request::Request(zmq::socket\_t\*)>& receiveData, const std::function<void(zmq::socket\_t\*, std::string)>& sendResult) {  
 bool finished = false;  
 std::thread workerThread([&sendResult, this, &finished](){  
 std::string result;  
 while(!finished) {  
 try {  
 result = net::receive(&current);  
 sendResult(&parentOutput, result);  
 } catch (...) {  
 continue;  
 }  
 }  
 });  
  
 while (true) {  
 try {  
 auto request = receiveData(&parentInput);  
 auto response = handleCommands(request.getCommand(), request.getReceiver(), request.getArgs(), request.getId());  
 if (std::holds\_alternative<Response::Response>(response)) {  
 auto res = std::get<Response::Response>(response);  
 res.setRequest(request);  
 sendResult(&parentOutput, res.constructResponseString());  
 } else if (std::get<Option>(response) == Option::*BREAK*) {  
 break;  
 } else if (std::get<Option>(response) == Option::*PASS\_MESSAGE*) {  
 for (auto &child : children) {  
 net::send\_message(child.second, request.toString());  
 }  
 }  
 } catch (std::exception& e) {  
 std::cout << e.what() << std::endl;  
 }  
  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  
 }  
  
 workerThread.join();  
  
 std::cout << "Terminating worker with id " << id << std::endl;  
 }  
  
 Response::Response Worker::handlePing() const {  
 return {Response::*SUCCESS*, "Ping successful", id};  
 }  
  
 Response::Response Worker::handleCreate(const std::vector<std::string> &args) {  
 if (args.size() != 1) return Response::Response::generateMalformedResponse(id);  
  
 auto childId = fromString<int>(args[0]);  
 children.insert({childId, new zmq::socket\_t(context, **ZMQ\_DEALER**)});  
 auto port = net::bind(children[childId], childId);  
 childrenPorts.insert({childId, port});  
 auto pid = startProcess(WORKER\_PATH, childId, childrenPorts[childId], currentPort);  
 if (pid == -1) return {Response::*FAILURE*, "Failed to create process for worker with id " + args[0], id};  
 children[childId]->setsockopt(**ZMQ\_RCVTIMEO**, 3000);  
  
 return {Response::*SUCCESS*, "Worker with id " + args[0] + " created (pid - " + std::to\_string(pid) + ")", id};  
 }  
  
 Response::Response Worker::handleKill(int receiver, size\_t requestId) {  
 if (receiver != -2) {  
 Response::Response msg;  
 try {  
 net::send\_message(children[receiver], Request::Request("kill", {}, -2, requestId).toString());  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(500));  
 } catch (...) {}  
 net::unbind(children[receiver], childrenPorts[receiver]);  
 children[receiver]->close();  
 children.erase(receiver);  
 childrenPorts.erase(receiver);  
 return {Response::ResponseType::*SUCCESS*, std::to\_string(receiver) + " killed", id};  
 }  
  
 for (const auto& [childId, child] : children) {  
 net::send\_message(&parentOutput, Response::Response(Response::ResponseType::*SUCCESS*, "Killed", id).constructResponseString());  
 net::send\_message(child, Request::Request("kill", {}, -2, requestId).toString());  
 net::unbind(child, childrenPorts[childId]);  
 child->close();  
 }  
 children.clear();  
 childrenPorts.clear();  
 return {Response::ResponseType::*SUCCESS*, std::to\_string(receiver) + " killed", id};  
 }  
} *// Worker*

**Worker.h**

*//  
// Created by nikit on 12/24/2024.  
//*#ifndef OS\_LAB\_5\_SRC\_WORKER\_H\_  
#define OS\_LAB\_5\_SRC\_WORKER\_H\_  
  
#include "Response.h"  
#include "../../cmake-build-debug/\_deps/cppzmq-src/zmq.hpp"  
#include "../../cmake-build-debug/\_deps/zeromq-src/include/zmq.h"  
#include "net.h"  
#include <map>  
#include <variant>  
#include <fstream>  
  
namespace Worker {  
  
 enum Option {  
 BREAK,  
 PASS\_MESSAGE  
 };  
  
 class Worker {  
 int id = -1;  
 zmq::context\_t context;  
 std::unordered\_map<int, zmq::socket\_t \*> children;  
 std::unordered\_map<int, int> childrenPorts;  
 zmq::socket\_t parentInput;  
 zmq::socket\_t parentOutput;  
 int parentPortInput;  
 int parentPortOutput;  
  
 zmq::socket\_t current;  
 int currentPort;  
  
 std::map<std::string, int> data;  
  
 private:  
 std::variant<Response::Response, Option> handleCommands(  
 const std::string& command,  
 int receiver,  
 const std::vector<std::string>& args,  
 size\_t requestId  
 );  
 Response::Response handleExec(  
 const std::vector<std::string>& args  
 );  
 Response::Response handleCreate(  
 const std::vector<std::string>& args  
 );  
 Response::Response handlePing() const;  
 Response::Response handleKill(int receiver, size\_t requestId);  
 public:  
 Worker() : id(-1), context(), children(), childrenPorts(), parentPortInput(-1), parentPortOutput(-1), parentInput(), parentOutput(), data() {  
 current = zmq::socket\_t(context, ZMQ\_DEALER);  
 currentPort = net::bind(&current, 0);  
 }  
 Worker(int id, int parentPortInput, int parentPortOutput) :  
 id(id),  
 context(),  
 parentInput(context, ZMQ\_DEALER),  
 parentOutput(context, ZMQ\_DEALER),  
 parentPortInput(parentPortInput),  
 parentPortOutput(parentPortOutput),  
 children(),  
 childrenPorts(),  
 data()  
 {  
 if (id != -1){  
 net::connect(&parentOutput, parentPortOutput);  
 net::connect(&parentInput, parentPortInput);  
 }  
  
 current = zmq::socket\_t(context, ZMQ\_DEALER);  
 currentPort = net::bind(&current, id);  
 }  
 void run(const std::function<Request::Request(zmq::socket\_t\*)>& receiveData, const std::function<void(zmq::socket\_t\*, std::string)>& sendResult);  
 ~Worker() = default;  
 };  
  
} *// Worker*#endif *//OS\_LAB\_5\_SRC\_WORKER\_H\_*

**Tests.h**

#include <gtest/gtest.h>  
#include "../src/Tree.h"  
#include "../src/Request.h"  
#include "../src/Response.h"  
  
TEST(TreeTests, ShouldInsertCorrectly){  
 std::array<size\_t, 5> data = {  
 4,  
 2,  
 3,  
 5,  
 6,  
 };  
 auto tree = Tree::Tree(data[0]);  
 tree.attach(data[0], data[1]);  
 tree.attach(data[0], data[2]);  
 tree.attach(data[1], data[3]);  
 tree.attach(data[2], data[4]);  
  
 EXPECT\_EQ(tree.getRoot()->getData(),data[0]);  
  
 EXPECT\_EQ(tree.getRoot()->getChildren().size(), 2);  
 EXPECT\_EQ(tree.getRoot()->getChildren()[0]->getData(), data[1]);  
 EXPECT\_EQ(tree.getRoot()->getChildren()[1]->getData(), data[2]);  
  
 EXPECT\_EQ(tree.getRoot()->getChildren()[0]->getChildren().size(), 1);  
 EXPECT\_EQ(tree.getRoot()->getChildren()[0]->getChildren()[0]->getData(), data[3]);  
  
 EXPECT\_EQ(tree.getRoot()->getChildren()[1]->getChildren().size(), 1);  
 EXPECT\_EQ(tree.getRoot()->getChildren()[1]->getChildren()[0]->getData(), data[4]);  
}  
  
TEST(TreeTests, ShouldRemoveCorrectly){  
 std::array<size\_t, 5> data = {  
 4,  
 2,  
 3,  
 5,  
 6,  
 };  
 auto tree = Tree::Tree(data[0]);  
 tree.attach(data[0], data[1]);  
 tree.attach(data[0], data[2]);  
 tree.attach(data[1], data[3]);  
 tree.attach(data[2], data[4]);  
  
 tree.remove(data[1]);  
  
 EXPECT\_EQ(tree.getRoot()->getData(),data[0]);  
 EXPECT\_EQ(tree.getRoot()->getChildren().size(), 1);  
  
 EXPECT\_EQ(tree.getRoot()->getChildren()[0]->getData(), data[2]);  
  
 EXPECT\_EQ(tree.getRoot()->getChildren()[0]->getChildren().size(), 1);  
 EXPECT\_EQ(tree.getRoot()->getChildren()[0]->getChildren()[0]->getData(), data[4]);  
}  
  
TEST(RequestParserTest, ShouldParseRequestCorrectlyWithoutId) {  
 const std::string request = "kill 2";  
 auto r = Request::Request::fromStringRequest(request, 1);  
 EXPECT\_EQ(r.getCommand(), "kill");  
 EXPECT\_EQ(r.getId(), 1);  
 EXPECT\_EQ(r.getArgs().size(), 0);  
 EXPECT\_EQ(r.getReceiver(), 2);  
}  
  
TEST(RequestParserTest, ShouldParseRequestCorrectlyWithId) {  
 const std::string request = "1 create 2 3";  
 auto r = Request::Request::fromStringRequest(request);  
 EXPECT\_EQ(r.getCommand(), "create");  
 EXPECT\_EQ(r.getId(), 1);  
 EXPECT\_EQ(r.getArgs().size(), 1);  
 EXPECT\_EQ(fromString<size\_t>(r.getArgs()[0]), 3);  
 EXPECT\_EQ(r.getReceiver(), 2);  
}  
  
TEST(ResponseParserTest, ShouldParseCorrectly) {  
 const std::string response = "1 create 2 3\nSUCCESS 2 CREATED SUCCESSFULLY";  
 auto res = Response::Response::fromStringResponse(response);  
  
 auto req = res.getRequest();  
  
 EXPECT\_EQ(req.getCommand(), "create");  
 EXPECT\_EQ(req.getId(), 1);  
 EXPECT\_EQ(req.getArgs().size(), 1);  
 EXPECT\_EQ(fromString<size\_t>(req.getArgs()[0]), 3);  
 EXPECT\_EQ(req.getReceiver(), 2);  
  
 EXPECT\_EQ(res.getType(), Response::ResponseType::SUCCESS);  
 EXPECT\_EQ(res.getMessage(), "CREATED SUCCESSFULLY");  
 EXPECT\_EQ(res.getHandlerId(), 2);  
}  
  
  
int main(int argc, char \*\*argv) {  
 testing::InitGoogleTest(&argc, argv);  
 return RUN\_ALL\_TESTS();  
}

**Child.cpp**

*//  
// Created by nikit on 12/24/2024.  
//*#include <fstream>  
#include "src/Utils.h"  
#include "src/Worker.h"  
  
int main(int argc, char\*\* argv) {  
 if (argc != 3) {  
 return -1;  
 }  
  
 auto id = fromString<int>(argv[0]);  
 auto parentPortInput = fromString<int>(argv[1]);  
 auto parentPortOutput = fromString<int>(argv[2]);  
  
 auto worker = Worker::Worker(id, parentPortInput, parentPortOutput);  
  
 auto receiver = [](zmq::socket\_t\* parent){  
 auto message = net::receive(parent);  
 return Request::Request::fromStringRequest(message);  
 };  
  
 auto sender = [](zmq::socket\_t\* parent, const std::string& message){  
 net::send\_message(parent, message);  
 };  
  
 worker.run(receiver, sender);  
}

**Main.cpp**

*//  
// Created by nikit on 12/24/2024.  
//*#include <iostream>  
#include <set>  
#include <thread>  
#include "src/Utils.h"  
#include "src/Worker.h"  
#include "src/Tree.h"  
  
int main(int argc, char\*\* argv) {  
 auto worker = Worker::Worker();  
  
 size\_t id = 0;  
  
 std::set<size\_t> unfinishedRequests;  
 Tree::Tree<int> tree(-1);  
  
 std::vector<std::thread> threads;  
  
 auto receiver = [&threads,&unfinishedRequests, &id, &tree](zmq::socket\_t\* parent){  
 std::string buffer;  
 std::cout << "Enter message: ";  
 std::getline(std::cin, buffer, '\n');  
 auto request = Request::Request::fromStringRequest(buffer, id);  
  
 if (request.getCommand() == "create" && tree.exists(fromString<int>(request.getArgs()[0]))) {  
 std::cout << "Node with id " << fromString<int>(request.getArgs()[0]) << " already exists" << std::endl;  
 throw std::invalid\_argument("Node with id " + std::to\_string(fromString<int>(request.getArgs()[0])) + " already exists");  
 } else if (request.getCommand() == "create" && !tree.exists(request.getReceiver())) {  
 std::cout << "Node with id " << request.getReceiver() << " does not exist (Receiver)" << std::endl;  
 throw std::invalid\_argument("Node with id " + std::to\_string(request.getReceiver()) + " does not exist (Receiver)");  
 }else if (request.getCommand() == "kill" && !tree.exists(request.getReceiver())) {  
 std::cout << "Node with id " << request.getReceiver() << " does not exist" << std::endl;  
 throw std::invalid\_argument("Node with id " + std::to\_string(request.getReceiver()) + " does not exist");  
 }  
  
 unfinishedRequests.insert(request.getId());  
  
 std::thread thread([&unfinishedRequests, &request, &tree](){  
 auto start = std::chrono::steady\_clock::now();  
 while (std::chrono::steady\_clock::now() - start < TIMEOUT) {  
 if (unfinishedRequests.contains(request.getId())) {  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  
 } else {  
 return;  
 }  
 }  
 if (tree.exists(request.getReceiver())){  
 tree.remove(request.getReceiver());  
 }  
 unfinishedRequests.erase(request.getId());  
 std::cout << "Request \"" << request.toString() << "\" timed out" << std::endl;  
 });  
  
 threads.push\_back(std::move(thread));  
  
 id++;  
  
 return request;  
 };  
  
 auto sender = [&tree, &unfinishedRequests](zmq::socket\_t\* parent, const std::string& message){  
 Response::Response response = Response::Response::fromStringResponse(message);  
 auto req = response.getRequest();  
 if (!unfinishedRequests.contains(req.getId())) { return; }  
 std::cout<< "Got message: " << split(response.constructResponseString(), '\n')[1] << std::endl;  
 if (req.getCommand() == "create"){  
 tree.attach(req.getReceiver(), fromString<int>(req.getArgs()[0]));  
 } else if (req.getCommand() == "kill") {  
 tree.remove(req.getReceiver());  
 }  
  
 unfinishedRequests.erase(req.getId());  
 };  
  
 worker.run(receiver, sender);  
  
 for (auto& thread: threads){  
 thread.join();  
 }  
}

**Заключение**

В ходе работы были успешно реализованы распределенная система для асинхронной обработки запросов с управляющим и вычислительными узлами. Обеспечена эффективная интеграция программных компонентов через очереди сообщений, а также реализована устойчивость системы к сбоям вычислительных узлов. Полученные результаты подтверждают освоение принципов управления серверами сообщений, применения отложенных вычислений и интеграции программных систем, что достигло поставленных целей работы.