МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"

Кафедра 806 "Вычислительная математика и программирование"

Лабораторные работы №5-7

По курсу «Операционные системы»

Студент: Теребаев К. Д.

Группа: М8О-203Б-22

Преподаватель: Миронов Е. С.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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)

**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

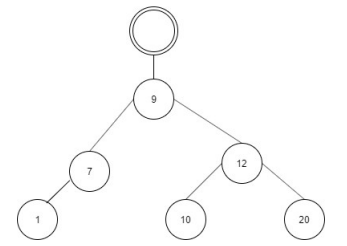
«Error:id: [Custom error]» - любая другая обрабатываемая ошибка

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.*

**Топология**



Все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.

**Тип команд для вычислительных узлов -** (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 foun

> 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 доступности узлов**Формат команды: heartbit time

Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от узла нет сигнала в течении 4\*time миллисекунд, то должна выводится пользователю строка: «Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

**Общие сведения о программе**

1. fork () –– создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.
2. exec () — используется для выполнения другой программы. Эта другая программа, называемая процессом-потомком (child process), загружается поверх программы, содержащей вызов exec. Имя файла, содержащего процесс-потомок, задано с помощью первого аргумента.
3. zmq\_ctx\_new () –– создает новый контекст ZMQ.
4. zmq\_connect () — создает входящее соединение на сокет.
5. zmq\_disconnect () — отсоединяет сокет от заданного endpoint.
6. zmq\_socket () — создает ZMQ сокет.
7. zmq\_close () — закрывает ZMQ сокет.
8. zmq\_ctx\_destroy () –– уничтожает контекст ZMQ.

**Общий алгоритм решения**

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZMQ.
2. Проработать принцип общения между клиентскими узлами и между первым клиентом и сервером и алгоритм выполнения команд клиентами.
3. Реализовать необходимые функции-обертки над вызовами функций библиотеки ZMQ.
4. Написать программу сервера и клиента.

**Основные файлы программы**

tree.hpp  
#pragma once

#include <vector>

struct Node {

    int id;

    Node\* left;

    Node\* right;

    bool found;

};

class Tree {

public:

    void push(int);

    void kill(int);

    std::vector<int> get\_nodes();

    ~Tree();

    Node\* root = NULL;

private:

    Node\* push(Node\* t, int);

    Node\* kill(Node\* t, int);

    void get\_nodes(Node\*, std::vector<int>&);

    void delete\_node(Node\*);

};

void print\_tree(Node\* root, int depth);

tree.cpp

#include "tree.hpp"

#include <algorithm>

#include <iostream>

#include <vector>

Tree::~Tree() {

    delete\_node(root);

}

void Tree::push(int id) {

    root = push(root, id);

}

void Tree::kill(int id) {

    root = kill(root, id);

}

void Tree::delete\_node(Node\* node) {

    if (node == NULL) {

        return;

    }

    delete\_node(node->right);

    delete\_node(node->left);

    delete node;

}

std::vector<int> Tree::get\_nodes() {

    std::vector<int> result;

    get\_nodes(root, result);

    return result;

}

void Tree::get\_nodes(Node\* node, std::vector<int>& v) {

    if (node == NULL) {

        return;

    }

    get\_nodes(node->left, v);

    v.push\_back(node->id);

    get\_nodes(node->right, v);

}

Node\* Tree::push(Node\* root, int val) {

    if (root == NULL) {

        root = new Node;

        root->id = val;

        root->left = NULL;

        root->right = NULL;

        root->found = false;

        return root;

    } else if (val < root->id) {

        root->left = push(root->left, val);

    } else if (val >= root->id) {

        root->right = push(root->right, val);

    }

    return root;

}

Node\* Tree::kill(Node\* root\_node, int val) {

    Node\* node;

    if (root\_node == NULL) {

        return NULL;

    } else if (val < root\_node->id) {

        root\_node->left = kill(root\_node->left, val);

    } else if (val > root\_node->id) {

        root\_node->right = kill(root\_node->right, val);

    } else {

        node = root\_node;

        if (root\_node->left == NULL) {

            root\_node = root\_node->right;

        } else if (root\_node->right == NULL) {

            root\_node = root\_node->left;

        }

        delete node;

    }

    if (root\_node == NULL) {

        return root\_node;

    }

    return root\_node;

}

void print\_tree(Node\* root, int depth) {

    if (root == nullptr) {

        return;

    }

    print\_tree(root->right, depth + 1);

    for (int i = 0; i < depth; ++i) {

        std::cout << "    ";

    }

    std::cout << root->id << std::endl;

    print\_tree(root->left, depth + 1);

}

main.cpp

#include <signal.h>

#include <unistd.h>

#include <cassert>

#include <chrono>

#include <iostream>

#include <sstream>

#include <string>

#include <thread>

#include <vector>

#include <zmq.hpp>

#include "tree.hpp"

using namespace std;

const int TIMER = 500;

const int DEFAULT\_PORT = 5050;

int n = 2, flag\_exit = 1;

pthread\_mutex\_t mutex1;

zmq::context\_t context(1);

zmq::socket\_t main\_socket(context, ZMQ\_REQ);

bool send\_message(zmq::socket\_t& socket, const string& message\_string) {

    zmq::message\_t message(message\_string.size());

    memcpy(message.data(), message\_string.c\_str(), message\_string.size());

    return socket.send(message);

}

string receive\_message(zmq::socket\_t& socket) {

    zmq::message\_t message;

    bool ok = false;

    try {

        ok = socket.recv(&message);

    } catch (...) {

        ok = false;

    }

    string received\_message(static\_cast<char\*>(message.data()), message.size());

    if (received\_message.empty() || !ok) {

        return "Root is dead";

    }

    return received\_message;

}

void create\_node(int id, int port) {

    char\* arg0 = strdup("./client");

    char\* arg1 = strdup((to\_string(id)).c\_str());

    char\* arg2 = strdup((to\_string(port)).c\_str());

    char\* args[] = {arg0, arg1, arg2, NULL};

    execv("./client", args);

}

string get\_port\_name(const int port) {

    return "tcp://127.0.0.1:" + to\_string(port);

}

bool is\_number(string val) {

    try {

        int tmp = stoi(val);

        return true;

    } catch (exception& e) {

        cout << "Error: " << e.what() << "\n";

        return false;

    }

}

typedef struct {

    int ping\_time;

    int ping\_id;

} heartbeat\_params;

void\* heartbeat\_iter(void\* param) {

    heartbeat\_params\* heartbeat\_param = (heartbeat\_params\*)param;

    chrono::milliseconds timespan(heartbeat\_param->ping\_time);

    string message\_string = "heartbeat " + to\_string(heartbeat\_param->ping\_id) + " " + to\_string(heartbeat\_param->ping\_time);

    int count = 0;

    for (int j = 0; j < 4; j++) {

        pthread\_mutex\_lock(&mutex1);

        send\_message(main\_socket, message\_string);

        string received\_message = receive\_message(main\_socket);

        pthread\_mutex\_unlock(&mutex1);

        this\_thread::sleep\_for(timespan);

        if (received\_message.substr(0, min<int>(received\_message.size(), 9)) != "Available") {

            break;

        }

        count += 1;

    }

    if (count == 0) {

        cout << "Node " + to\_string(heartbeat\_param->ping\_id) + " is unavailable.\n";

    } else {

        cout << "Node " + to\_string(heartbeat\_param->ping\_id) + " is available.\n";

    }

    pthread\_exit(0);

}

int main() {

    Tree T;

    std::vector<int> nodes;

    string command;

    int child\_pid = 0;

    int child\_id = 0;

    pthread\_mutex\_init(&mutex1, NULL);

    cout << "Commands:\n";

    cout << "1. create (id)\n";

    cout << "2. exec (id) (name, value)\n";

    cout << "3. kill (id)\n";

    cout << "4. ping (id)\n";

    cout << "5. heartbeat (ping\_time)\n";

    cout << "6. exit\n" << endl;

    while (true) {

        cin >> command;

        if (command == "create") {

            n++;

            size\_t node\_id = 0;

            string str = "";

            string result = "";

            cin >> str;

            if (!is\_number(str)) {

                continue;

            }

            node\_id = stoi(str);

            if (child\_pid == 0) {

                main\_socket.bind(get\_port\_name(DEFAULT\_PORT + node\_id));

                main\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                main\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                child\_pid = fork();

                if (child\_pid == -1) {

                    cerr << "Unable to create first worker node\n";

                    child\_pid = 0;

                    exit(1);

                } else if (child\_pid == 0) {

                    create\_node(node\_id, DEFAULT\_PORT + node\_id);

                } else {

                    child\_id = node\_id;

                    main\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                    main\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                    send\_message(main\_socket, "pid");

                    result = receive\_message(main\_socket);

                }

            } else {

                main\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                main\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                string msg\_s = "create " + to\_string(node\_id);

                send\_message(main\_socket, msg\_s);

                result = receive\_message(main\_socket);

            }

            if (result.substr(0, 2) == "Ok") {

                T.push(node\_id);

                nodes.push\_back(node\_id);

            }

            cout << result << "\n";

        } else if (command == "kill") {

            int node\_id = 0;

            string str = "";

            cin >> str;

            if (!is\_number(str)) {

                continue;

            }

            node\_id = stoi(str);

            if (child\_pid == 0) {

                cout << "Error: Not found\n";

                continue;

            }

            if (node\_id == child\_id) {

                kill(child\_pid, SIGTERM);

                kill(child\_pid, SIGKILL);

                child\_id = 0;

                child\_pid = 0;

                T.kill(node\_id);

                cout << "Ok\n";

                continue;

            }

            string message\_string = "kill " + to\_string(node\_id);

            send\_message(main\_socket, message\_string);

            string received\_message;

            received\_message = receive\_message(main\_socket);

            if (received\_message.substr(0, min<int>(received\_message.size(), 2)) == "Ok") {

                T.kill(node\_id);

            }

            cout << received\_message << "\n";

        } else if (command == "exec") {

            string input\_string;

            string id\_str = "";

            string name = "";

            string value = "0";

            int id = 0;

            getline(cin, input\_string);

            istringstream iss(input\_string);

            vector<std::string> words;

            std::string word;

            while (iss >> word) {

                words.push\_back(word);

            }

            id\_str = words[0];

            if (!is\_number(id\_str)) {

                continue;

            }

            id = stoi(id\_str);

            name = words[1];

            if (words.size() == 2) {

                string message\_string = "exec " + to\_string(id) + " " + name + " " + "NOVALUE";

                send\_message(main\_socket, message\_string);

                string received\_message = receive\_message(main\_socket);

                cout << received\_message << "\n";

            }

            if (words.size() == 3) {

                value = words[2];

                string message\_string = "exec " + to\_string(id) + " " + name + " " + value;

                send\_message(main\_socket, message\_string);

                string received\_message = receive\_message(main\_socket);

                cout << received\_message << "\n";

            }

        } else if (command == "ping") {

            string id\_str = "";

            int id = 0;

            cin >> id\_str;

            if (!is\_number(id\_str)) {

                continue;

            }

            id = stoi(id\_str);

            string message\_string = "ping " + to\_string(id);

            send\_message(main\_socket, message\_string);

            string received\_message = receive\_message(main\_socket);

            cout << received\_message << "\n";

        } else if (command == "heartbeat") {

            string time\_str = "";

            int ping\_time = 0;

            cin >> time\_str;

            if (!is\_number(time\_str)) {

                continue;

            }

            ping\_time = stoi(time\_str);

            string message\_string;

            std::vector<int> check\_nodes = T.get\_nodes();

            pthread\_t tid[check\_nodes.size()];

            heartbeat\_params hb[check\_nodes.size()];

            for (int i = 0; i < check\_nodes.size(); i++) {

                hb[i].ping\_time = ping\_time;

                hb[i].ping\_id = check\_nodes[i];

                pthread\_create(&tid[i], NULL, heartbeat\_iter, &hb[i]);

            }

            for (int i = 0; i < check\_nodes.size(); i++) {

                pthread\_join(tid[i], NULL);

            }

            if (check\_nodes.size() == 0) {

                cout << "There isn't calculation nodes" << endl;

            }

        } else if (command == "exit") {

            try {

                system("killall client");

                flag\_exit = 0;

            } catch (exception& e) {

                cout << "Error: " << e.what() << "\n";

            }

            break;

        } else if (command == "print") {

            print\_tree(T.root, 0);

        }

    }

    return 0;

}

client.cpp

#include <signal.h>

#include <stdlib.h>

#include <unistd.h>

#include <exception>

#include <iostream>

#include <map>

#include <sstream>

#include <string>

#include <zmq.hpp>

using namespace std;

const int TIMER = 500;

const int DEFAULT\_PORT = 5050;

int n = 2;

std::map<std::string, int> m;

bool send\_message(zmq::socket\_t& socket, const string& message\_string) {

    zmq::message\_t message(message\_string.size());

    memcpy(message.data(), message\_string.c\_str(), message\_string.size());

    return socket.send(message);

}

string receive\_message(zmq::socket\_t& socket) {

    zmq::message\_t message;

    bool ok = false;

    try {

        ok = socket.recv(&message);

    } catch (...) {

        ok = false;

    }

    string received\_message(static\_cast<char\*>(message.data()), message.size());

    if (received\_message.empty() || !ok) {

        return "";

    }

    return received\_message;

}

void create\_node(int id, int port) {

    char\* arg0 = strdup("./client");

    char\* arg1 = strdup((to\_string(id)).c\_str());

    char\* arg2 = strdup((to\_string(port)).c\_str());

    char\* args[] = {arg0, arg1, arg2, NULL};

    execv("./client", args);

}

string get\_port\_name(const int port) {

    return "tcp://127.0.0.1:" + to\_string(port);

}

void real\_create(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& create\_id, int& id, int& pid) {

    cout << to\_string(id);

    if (pid == -1) {

        send\_message(parent\_socket, "Error: Cannot fork");

        pid = 0;

    } else if (pid == 0) {

        create\_node(create\_id, DEFAULT\_PORT + create\_id);

    } else {

        id = create\_id;

        send\_message(socket, "pid");

        send\_message(parent\_socket, receive\_message(socket));

    }

}

void real\_kill(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& delete\_id, int& id, int& pid, string& request\_string) {

    if (id == 0) {

        send\_message(parent\_socket, "Error: Not found");

    } else if (id == delete\_id) {

        send\_message(socket, "kill\_children");

        receive\_message(socket);

        kill(pid, SIGTERM);

        kill(pid, SIGKILL);

        id = 0;

        pid = 0;

        send\_message(parent\_socket, "Ok");

    } else {

        send\_message(socket, request\_string);

        send\_message(parent\_socket, receive\_message(socket));

    }

}

void real\_exec(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& id, int& pid, string& request\_string) {

    if (pid == 0) {

        string receive\_message = "Error:" + to\_string(id);

        receive\_message += ": Not found";

        send\_message(parent\_socket, receive\_message);

    } else {

        send\_message(socket, request\_string);

        string str = receive\_message(socket);

        if (str == "") str = "Error: Node is unavailable";

        send\_message(parent\_socket, str);

    }

}

void real\_ping(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& id, int& pid, string& request\_string) {

    if (pid == 0) {

        string receive\_message = "Error:" + to\_string(id);

        receive\_message += ": Not found";

        send\_message(parent\_socket, receive\_message);

    } else {

        send\_message(socket, request\_string);

        string str = receive\_message(socket);

        if (str == "") str = "Ok: 0";

        send\_message(parent\_socket, str);

    }

}

void real\_heartbeat(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& id, int& pid, string& request\_string) {

    if (pid == 0) {

        string receive\_message = "Error:" + to\_string(id);

        receive\_message += ": Not found";

        send\_message(parent\_socket, receive\_message);

    } else {

        send\_message(socket, request\_string);

        string str = receive\_message(socket);

        if (str == "") str = "Ok: 0";

        send\_message(parent\_socket, str);

    }

}

void exec(istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid,

          int& id, string& request\_string) {

    string name, value;

    int exec\_id;

    command\_stream >> exec\_id;

    if (exec\_id == id) {

        command\_stream >> name;

        command\_stream >> value;

        string receive\_message = "";

        string answer = "";

        if (value == "NOVALUE") {

            receive\_message = "Ok:" + to\_string(id) + ":";

            if (m.contains(name)) {

                receive\_message += to\_string(m[name]);

            } else {

                receive\_message += " '" + name + "' not found";

            }

        } else {

            m[name] = stoi(value);

            receive\_message = "Ok:" + to\_string(id);

        }

        send\_message(parent\_socket, receive\_message);

    } else if (exec\_id < id) {

        real\_exec(parent\_socket, left\_socket, exec\_id, left\_pid, request\_string);

    } else {

        real\_exec(parent\_socket, right\_socket, exec\_id, right\_pid, request\_string);

    }

}

void ping(istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid,

          int& id, string& request\_string) {

    int ping\_id;

    string receive\_message;

    command\_stream >> ping\_id;

    if (ping\_id == id) {

        receive\_message = "Ok: 1";

        send\_message(parent\_socket, receive\_message);

    } else if (ping\_id < id) {

        real\_ping(parent\_socket, left\_socket, ping\_id, left\_pid, request\_string);

    } else {

        real\_ping(parent\_socket, right\_socket, ping\_id, right\_pid, request\_string);

    }

}

void heartbeat(istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket, int& left\_pid,

               int& right\_pid, int& id, string& request\_string) {

    int ping\_id;

    int ping\_time;

    string receive\_message;

    command\_stream >> ping\_id;

    command\_stream >> ping\_time;

    if (ping\_id == id) {

        receive\_message = "Available:" + to\_string(id);

        send\_message(parent\_socket, receive\_message);

    } else if (ping\_id < id) {

        real\_heartbeat(parent\_socket, left\_socket, ping\_id, left\_pid, request\_string);

    } else {

        real\_heartbeat(parent\_socket, right\_socket, ping\_id, right\_pid, request\_string);

    }

}

void kill\_children(zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid) {

    if (left\_pid == 0 && right\_pid == 0) {

        send\_message(parent\_socket, "Ok");

    } else {

        if (left\_pid != 0) {

            send\_message(left\_socket, "kill\_children");

            receive\_message(left\_socket);

            kill(left\_pid, SIGTERM);

            kill(left\_pid, SIGKILL);

        }

        if (right\_pid != 0) {

            send\_message(right\_socket, "kill\_children");

            receive\_message(right\_socket);

            kill(right\_pid, SIGTERM);

            kill(right\_pid, SIGKILL);

        }

        send\_message(parent\_socket, "Ok");

    }

}

int main(int argc, char\*\* argv) {

    int id = stoi(argv[1]);

    int parent\_port = stoi(argv[2]);

    zmq::context\_t context(3);

    zmq::socket\_t parent\_socket(context, ZMQ\_REP);

    parent\_socket.connect(get\_port\_name(parent\_port));

    parent\_socket.set(zmq::sockopt::rcvtimeo, TIMER);

    parent\_socket.set(zmq::sockopt::sndtimeo, TIMER);

    int left\_pid = 0;

    int right\_pid = 0;

    int left\_id = 0;

    int right\_id = 0;

    zmq::socket\_t left\_socket(context, ZMQ\_REQ);

    zmq::socket\_t right\_socket(context, ZMQ\_REQ);

    while (true) {

        string request\_string = receive\_message(parent\_socket);

        istringstream command\_stream(request\_string);

        string command;

        command\_stream >> command;

        if (command == "id") {

            string parent\_string = "Ok:" + to\_string(id);

            send\_message(parent\_socket, parent\_string);

        } else if (command == "pid") {

            string parent\_string = "Ok:" + to\_string(getpid());

            send\_message(parent\_socket, parent\_string);

        } else if (command == "create") {

            int create\_id;

            command\_stream >> create\_id;

            if (create\_id == id) {

                string message\_string = "Error: Already exists";

                send\_message(parent\_socket, message\_string);

            } else if (create\_id < id) {

                if (left\_pid == 0) {

                    left\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

                    left\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                    left\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                    left\_pid = fork();

                    real\_create(parent\_socket, left\_socket, create\_id, left\_id, left\_pid);

                } else {

                    send\_message(left\_socket, request\_string);

                    string str = receive\_message(left\_socket);

                    if (str == "") {

                        left\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

                        left\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                        left\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                        left\_pid = fork();

                        real\_create(parent\_socket, left\_socket, create\_id, left\_id, left\_pid);

                    } else {

                        send\_message(parent\_socket, str);

                        n++;

                        left\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                        left\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                    }

                }

            } else {

                if (right\_pid == 0) {

                    right\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

                    right\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                    right\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                    right\_pid = fork();

                    real\_create(parent\_socket, right\_socket, create\_id, right\_id, right\_pid);

                } else {

                    send\_message(right\_socket, request\_string);

                    string str = receive\_message(right\_socket);

                    if (str == "") {

                        right\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

                        right\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                        right\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                        right\_pid = fork();

                        real\_create(parent\_socket, right\_socket, create\_id, right\_id, right\_pid);

                    } else {

                        send\_message(parent\_socket, str);

                        n++;

                        right\_socket.set(zmq::sockopt::rcvtimeo, n \* TIMER);

                        right\_socket.set(zmq::sockopt::sndtimeo, n \* TIMER);

                    }

                }

            }

        } else if (command == "kill") {

            int delete\_id;

            command\_stream >> delete\_id;

            if (delete\_id < id) {

                real\_kill(parent\_socket, left\_socket, delete\_id, left\_id, left\_pid, request\_string);

            } else {

                real\_kill(parent\_socket, right\_socket, delete\_id, right\_id, right\_pid, request\_string);

            }

        } else if (command == "exec") {

            exec(command\_stream, parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid, id, request\_string);

        } else if (command == "ping") {

            ping(command\_stream, parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid, id, request\_string);

        } else if (command == "heartbeat") {

            heartbeat(command\_stream, parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid, id, request\_string);

        } else if (command == "kill\_children") {

            kill\_children(parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid);

        }

        if (parent\_port == 0) {

            break;

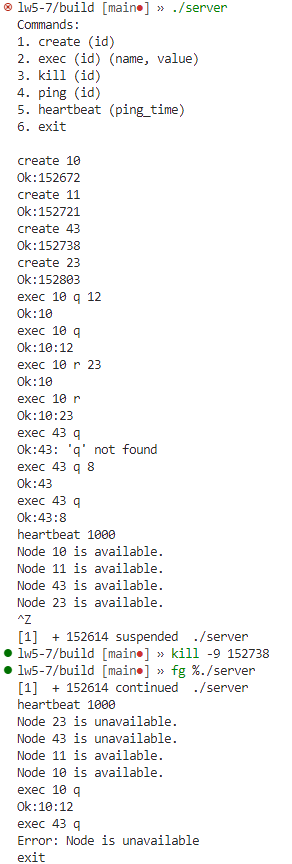
        }

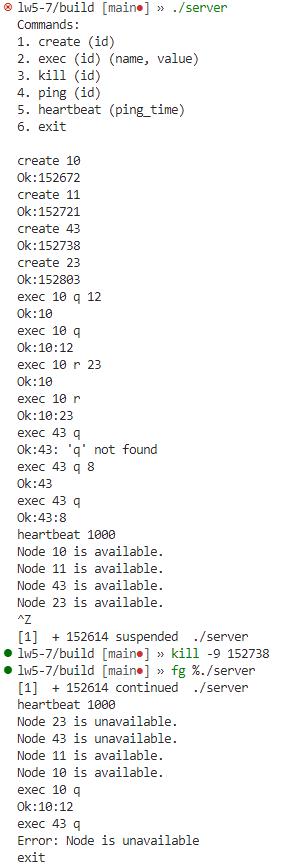
    }

    return 0;

}

**Пример работы**





**Вывод**

В ходе выполнения лабораторной работы я изучил технологию передачи сообщений ZeroMQ, с помощью которой создал систему по асинхронной обработке запросов. Пригодились знания из первой лабораторной работы, такие как: умение использовать системные вызовы fork и execl, - которые нужны были при создании нового рабочего узла. ZeroMQ определенно является технологией, которая упрощает передачу сообщений при создании большой системы.