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

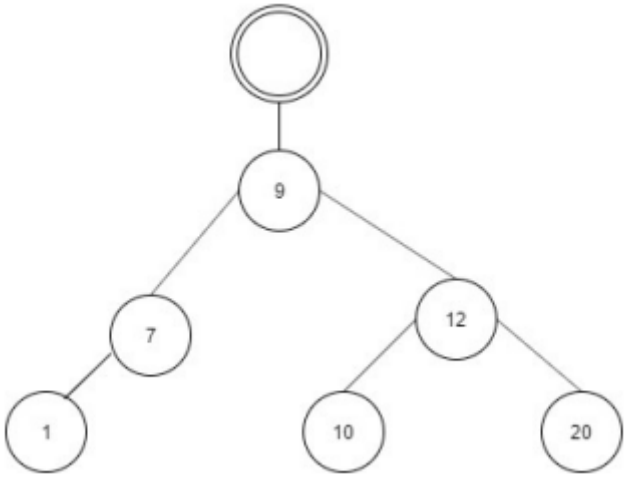
«Error:id: [Custom error]» - любая другая обрабатываемая ошибка

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

```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

```
⊗ lw5-7/build [main•] » ./server
Commands:
1. create (id)
2. exec (id) (name, value)
3. kill (id)
4. ping (id)
5. heartbeat (ping_time)
6. exit

create 10
Ok:152672
create 11
Ok:152721
create 43
Ok:152738
create 23
Ok:152803
exec 10 q 12
Ok:10
exec 10 q
Ok:10:12
exec 10 r 23
```

```
Ok:10
exec 10 r
Ok:10:23
exec 43 q
Ok:43: 'q' not found
exec 43 q 8
Ok:43
exec 43 q
Ok:43:8
heartbeat 1000
Node 10 is available.
Node 11 is available.
Node 43 is available.
Node 23 is available.
^Z
[1]  + 152614 suspended  ./server
● lw5-7/build [main•] » kill -9 152738
● lw5-7/build [main•] » fg %./server
[1]  + 152614 continued  ./server
heartbeat 1000
Node 23 is unavailable.
Node 43 is unavailable.
Node 11 is available.
Node 10 is available.
exec 10 q
Ok:10:12
exec 43 q
Error: Node is unavailable
exit
```

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.