

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

**Лабораторная работа №6-8 по курсу
«Операционные системы»**

Студент: Волков Евгений Андреевич
Группа: М8О-207Б-21
Вариант: 29
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Содержание

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

Репозиторий

Постановка задачи

Целью является приобретение практических навыков в:

1. Управлении серверами сообщений (No6)
2. Применение отложенных вычислений (No7)
3. Интеграция программных систем друг с другом (No8)

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.

Общие сведения о программе

Для реализации очередей сообщений используем ZeroMQ. interface.h, config.h — интерфейсы для interface.cpp и config.cpp. Файлы tree.h и tree.cpp помогают работать с бинарным деревом (в них представлена его реализация). Client.cpp — клиент, через который мы взаимодействуем с сервером server.cpp.

Топология 3: Все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.

Набора команд 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

Команда 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

Общий метод и алгоритм решения

Программа реализована с помощью сокетов типа ZMQ_REQ и ZMQ_REP. ZMQ_REQ сперва отправляет сообщение, потом принимает, ZMQ_REP — наоборот. На контрольном узле находится топология двоичное дерево, в котором лежат id (int). Оно нужно для того, чтобы проверять, существует ли уже узел или его нет, туда же добавляется id в тот момент, когда мы создаем процесс. С помощью флага ZMQ_SNDMORE мы можем передавать сообщения частями, поэтому например для вставки мы сперва передаем команду, потом по очереди все параметры команды. ./server (процесс) аккумулирует все необходимые параметры команды перед ее выполнением.

В дереве есть корень, который является управляющим узлом (значение id 15)

Также на каждом узле есть сокеты для связи с левым и правым потомком. Если при создании (create) левый или правый потомок узла в топологии NULL, то мы просто создаем процесс, запускаем его на ./server и соединяемся с ним вновь созданным сокетом. Иначе отправляем сообщение на сокет правого узла, на сервере аналогично проверяем.

При удалении (remove) узла вместе с ним удаляются все его поддеревья, и завершаются соответствующие процессы. После удаления мы закрываем сокет на узле.

Ping — проверяет доступность узла. Его реализация напоминает поиск в двоичном дереве, но вместо указателей мы передаем сообщение между сокетами, так как все узлы представлены процессами. Если до узла невозможно достучаться, выводим сообщение об ошибке.

Команда exes бывает двух видов: на проверку значения в словаре по ключу и на добавление.

Ищем необходимый узел также, как и другие команды, однако ZMQ_SNDMORE вызывается больше раз (в случае проверки нужно передать ключ по сокетам, в случае добавление — еще и значение).

Исходный код

tree.h

```
#pragma once

#include <iostream>

using namespace std;

struct Tree {

    int id;

    Tree* left;

    Tree* right;

};

Tree* createNode(Tree* root, int id);

bool existNode(Tree* root, int id);

Tree* deleteNode(Tree* root, int id);

Tree* createTree(int id);

void printTree(Tree* root, int n);
```

tree.cpp

```
#include "tree.h"

#include <iostream>

#include <stdio.h>

using namespace std;

Tree* createTree(int value)

{

    Tree* tree = (Tree*)malloc(sizeof(Tree));
```

```

    tree->id = value;

    tree->right = NULL;

    tree->left = NULL;


    return tree;
}

Tree* createNode(Tree* root, int value)
{
    Tree* res = createTree(value);

    if (value == root->id) {

        return root;
    }

    if (value > (root->id) && (root->right) == NULL) {

        root->right = createTree(value);

        return root;
    }

    if (value < (root->id) && (root->left) == NULL) {

        root->left = createTree(value);

        return root;
    }

    if (value > (root->id) && (root->right) != NULL) {

        root->right = createNode(root->right, value);
    }

    if (value < (root->id) && (root->left) != NULL) {

        root->left = createNode(root->left, value);
    }

    return root;
}

```

```

void printTree(Tree* root, int n)
{
    if (root != NULL)
    {
        printTree(root->right, n + 1);

        for (int i = 0; i < n; i++)

            printf("\t");

        printf("%d\n", root->id);

        printTree(root->left, n + 1);
    }
}

```

```

bool existNode(Tree* root, int id)
{
    if (root == NULL) {
        return false;
    }

    if (root->id == id) {
        return true;
    }

    return existNode(root->left, id) || existNode(root->right, id);
}

```

```

Tree* deleteNode(Tree* root, int id) {
    if (root == NULL)
        return root;

    if (id < root->id) {
        root->left = deleteNode(root->left, id);
        return root;
    }
}

```

```

    if (id > root->id) {

        root->right = deleteNode(root->right, id);

        return root;

    }

    free(root);

    root = NULL;

    return root;

}

```

interface.h

```

#ifndef __INTERFACE_H__

```

```

#define __INTERFACE_H__

```

```

#include <stdio.h>

```

```

#include <iostream>

```

```

#include <zmq.h>

```

```

#include <stdlib.h>

```

```

#include <string>

```

```

#include <unistd.h>

```

```

#include<cstring>

```

```

#include<stdexcept>

```

```

#include<assert.h>

```

```

#include<map>

```

```

using namespace std;

```

```

#define CLIENT_PREFIX "tcp://localhost:"

```

```

#define SERVER_PREFIX "tcp://*:"

```

```

#define BASE_PORT 4000

```

```

#define STR_LEN 64

```



```

#define EMPTY_STR ""

#define REQUEST_TIMEOUT 2000

typedef enum {

                                EXIT = 0,

    CREATE,

    REMOVE,

    EXEC,

    PRINT,

    PING,

    DEFAULT

} command_type;

```

```

// typedef enum {

//     ZMQ_SNDMORE,

//     0

// } send_more;

```

```

string convert_adr_client(unsigned short port);

string convert_adr_server(unsigned short port);

```

```

const char* int_to_str(unsigned a);

```

```

command_type get_command();

```

```

string unitread();

```

```

#endif

interface.cpp

#include "interface.h"

```

```

string unitread() //считывание по словам через пробел

```

```

{

    string result = "";

    char cur;

    while((cur = getchar()) != ' ') {

        if (cur == '\0' || cur == '\n') {

            break;

        }

        result += cur;

    }

    return result;

}

command_type get_command()

{

    string cmd = unitread();

    if (strcmp(cmd.c_str(), "print") == 0) {

        return PRINT;

    }

    if (strcmp(cmd.c_str(), "create") == 0) {

        return CREATE;

    }

    if (strcmp(cmd.c_str(), "exec") == 0) {

        return EXEC;

    }

    if (strcmp(cmd.c_str(), "exit") == 0) {

        return EXIT;

    }

    if (strcmp(cmd.c_str(), "remove") == 0) {

        return REMOVE;

    }

}

```

```

    }

    if (strcmp(cmd.c_str(),"ping") == 0) {

        return PING;

    }

    return DEFAULT;

}

```

```

string convert_adr_client(unsigned short port)

{

    string port_string = int_to_str(port);

    string name = CLIENT_PREFIX + port_string;

    return name;

}

```

```

string convert_adr_server(unsigned short port)

{

    string port_string = int_to_str(port);

    string name = SERVER_PREFIX + port_string;

    return name;

}

```

```

const char* int_to_str(unsigned a)

{

    int num = a, i = 0;

    if (a == 0)

        return "0";

    while (num > 0) {

        num /= 10;

        i++;

    }

}

```

```

char *result = (char *)calloc(sizeof(char), i + 1);

while (i >= 1) {

    result[--i] = a % 10 + '0';

    a /= 10;

}

return result;

}

```

config.h

```

#ifndef __CONFIG_H__

#define __CONFIG_H__

#include "interface.h"

#include "tree.h"

#define SERVER_PATH "./server"

void create_server_node(int id);

void send_create(void* socket, int id);

void send_exec(void* socket, int id, char* key, int value, int save);

void send_remove(void* socket, int id);

void send_exit(void *socket);

```

```
void send_heartbit(void *socket, unsigned int time);
```

```
bool available_receive(void* socket);
```

```
void send_ping(void* socket, int id);
```

```
char* receive(void* socket);
```

```
#endif
```

```
config.cpp
```

```
#include "config.h"
```

```
void create_server_node(int id)
```

```
{
```

```
    const char* arg = SERVER_PATH;
```

```
    const char* arg0 = int_to_str(id);
```

```
    execl(arg, arg0, NULL);
```

```
}
```

```
void send_create(void* socket, int id)
```

```
{
```

```
    command_type cmd = CREATE;
```

```
    zmq_msg_t command;
```

```
    zmq_msg_init_size(&command, sizeof(cmd)); //выделяем ресурсы для хранения сообщения
```

```
    memcpy(zmq_msg_data(&command), &cmd, sizeof(cmd)); //копируем содержимое одной области памяти в другую
```

```
    zmq_msg_send(&command, socket, ZMQ_SNDMORE); //ставит в очередь сообщение
```

```
    zmq_msg_close(&command);
```

```
    zmq_msg_t id_msg;
```

```
    zmq_msg_init_size(&id_msg, sizeof(id));
```

```
    memcpy(zmq_msg_data(&id_msg), &id, sizeof(id));
```

```

//указатель на содержимое сообщения

zmq_msg_send(&id_msg, socket, 0);

zmq_msg_close(&id_msg);

}

void send_remove(void* socket, int id)
{
    command_type cmd = REMOVE;

    zmq_msg_t command;

    zmq_msg_init_size(&command, sizeof(cmd));

    memcpy(zmq_msg_data(&command), &cmd, sizeof(cmd));

    zmq_msg_send(&command, socket, ZMQ_SNDMORE);

    zmq_msg_close(&command);

    zmq_msg_t id_msg;

    zmq_msg_init_size(&id_msg, sizeof(id));

    memcpy(zmq_msg_data(&id_msg), &id, sizeof(id));

    zmq_msg_send(&id_msg, socket, 0);

    zmq_msg_close(&id_msg);

}

void send_exec(void* socket, int id, char* key, int value, int save)
{
    command_type cmd = EXEC;

    zmq_msg_t command;

    zmq_msg_init_size(&command, sizeof(cmd));

    memcpy(zmq_msg_data(&command), &cmd, sizeof(cmd));

    zmq_msg_send(&command, socket, ZMQ_SNDMORE);

    zmq_msg_close(&command);

```

```

zmq_msg_t id_msg;

zmq_msg_init_size(&id_msg, sizeof(id));

memcpy(zmq_msg_data(&id_msg), &id, sizeof(id));

zmq_msg_send(&id_msg, socket, ZMQ_SNDMORE);

zmq_msg_close(&id_msg);


if (save == 0) { //есть только ключ

    const char* send_key = (char*)malloc(sizeof(key));

    send_key = key;

    zmq_msg_t key_msg;

    zmq_msg_init_size(&key_msg, sizeof(send_key));

    memcpy(zmq_msg_data(&key_msg), send_key, sizeof(send_key));

    zmq_msg_send(&key_msg, socket, 0);

    zmq_msg_close(&key_msg);

    return;

}


const char* send_key = (char*)malloc(sizeof(key));

send_key = key;

zmq_msg_t key_msg;

zmq_msg_init_size(&key_msg, sizeof(send_key));

memcpy(zmq_msg_data(&key_msg), send_key, sizeof(send_key));

zmq_msg_send(&key_msg, socket, ZMQ_SNDMORE);

zmq_msg_close(&key_msg);


if (save == 1) {

    zmq_msg_t value_msg;

    zmq_msg_init_size(&value_msg, sizeof(value));

    memcpy(zmq_msg_data(&value_msg), &value, sizeof(value));

    zmq_msg_send(&value_msg, socket, 0);

    zmq_msg_close(&value_msg);

```

```
}
```

```
}
```

```
bool available_receive(void *socket) {////////?  
  
    zmq_pollitem_t items[1] = {{socket, 0, ZMQ_POLLIN, 0}};  
  
    int rc = zmq_poll(items, 1, REQUEST_TIMEOUT);  
  
    assert(rc != -1);  
  
    if (items[0].revents & ZMQ_POLLIN)  
        return true;  
  
    return false;  
}
```

```
char* receive(void* socket)  
  
{  
  
    zmq_msg_t reply;  
  
    zmq_msg_init(&reply); //инициализирует объект пустого сообщения  
  
    zmq_msg_rcv(&reply, socket, 0);  
  
    size_t result_size = zmq_msg_size(&reply);  
  
    char* result = (char*)calloc(sizeof(char), result_size + 1);  
  
    memcpy(result, zmq_msg_data(&reply), result_size);  
  
    zmq_msg_close(&reply);  
  
    return result;  
}
```

```
void send_exit(void *socket)  
  
{
```



```

command_type cmd = EXIT;

zmq_msg_t command_msg;

zmq_msg_init_size(&command_msg, sizeof(cmd));

memcpy(zmq_msg_data(&command_msg), &cmd, sizeof(cmd));

zmq_msg_send(&command_msg, socket, 0);

zmq_msg_close(&command_msg);
}

```

```

void send_ping(void* socket, int id)
{
    command_type cmd = PING;

    zmq_msg_t command;

    zmq_msg_init_size(&command, sizeof(cmd));

    memcpy(zmq_msg_data(&command), &cmd, sizeof(cmd));

    zmq_msg_send(&command, socket, ZMQ_SNDMORE);

    zmq_msg_close(&command);


    zmq_msg_t id_msg;

    zmq_msg_init_size(&id_msg, sizeof(id));

    memcpy(zmq_msg_data(&id_msg), &id, sizeof(id));

    zmq_msg_send(&id_msg, socket, 0);

    zmq_msg_close(&id_msg);
}

```

client.cpp

```
#include "config.h"
```

```
using namespace std;
```

```
#define CLIENT_ROOT_ID 15
```

```

int main()

{

    cout<<"Using client root "<<CLIENT_ROOT_ID<<" as default"<<endl;

    Tree* system;

    system = createTree(CLIENT_ROOT_ID);

    void *context = zmq_ctx_new();// create new context

    if (context == NULL) {

        throw runtime_error("Error: Can't initialize context");

    }


    void* socket_left = NULL;

    void* socket_right = NULL;

    int ex = 0;

    while (true) {

        command_type cur_command = get_command(); // считываем команду из ввода

        string child_id_str;

        int child_id;

        string remove_id_str;

        int remove_id;

        int exec_id;

        int ping_id;

        int save;

        string ping_id_str;

        string exec_id_str;

        int value;

        string key;

        char *reply = (char *)calloc(sizeof(char), 64);

        switch(cur_command) {

```

```

case PRINT:

    printTree(system,0);

    break;

case CREATE:

    child_id_str = unitread(); // считывание след слова через пробел

    child_id = atoi(child_id_str.c_str());

    if (child_id <= 0) {

        cout<<"Error: invalid id"<<endl;

        break;

    }

    if (existNode(system, child_id)) {

        cout<<"Error: already exists"<<endl;

        break;

    }

    system = createNode(system, child_id);

    if (child_id > CLIENT_ROOT_ID) {

        if (socket_right == NULL) {

            int fork_pid = fork();

            if (fork_pid == -1) {

                throw runtime_error("Error: fork problem occurred");

                break;

            }

            if (fork_pid == 0) {

                create_server_node(child_id);

            }

            socket_right = zmq_socket(context, ZMQ_REQ); //create new request socket

            cout<<"OK: "<<fork_pid<<endl;

            int opt = 0;

```

```

int rc = zmq_setsockopt(socket_right, ZMQ_LINGER, &opt, sizeof(opt)); //установка периода ожидания для
сокета

assert(rc == 0); //пишет сообщение об ошибке

if (socket_right == NULL) {

    throw runtime_error("Error: socket not created");

}

rc = zmq_connect(socket_right, convert_adr_client(BASE_PORT + child_id).c_str()); //connect socket with
endpoint - адресом ребенка

assert(rc == 0);

break;

}

}

if (child_id < CLIENT_ROOT_ID) {

    if (socket_left == NULL) {

        int fork_pid = fork();

        if (fork_pid == -1) {

            throw runtime_error("Error: fork problem occurred");

            break;

        }

        if (fork_pid == 0) {

            create_server_node(child_id);

        }

        socket_left = zmq_socket(context, ZMQ_REQ);

        cout<<"OK: "<<fork_pid<<endl;

        int opt = 0;

        int rc = zmq_setsockopt(socket_left, ZMQ_LINGER, &opt, sizeof(opt));

        assert(rc == 0);

        if (socket_left == NULL) {

            throw runtime_error("Error: socket not created");

```

```

    }

    rc = zmq_connect(socket_left, convert_adr_client(BASE_PORT + child_id).c_str());

    assert(rc == 0);

    break;
}

}

if (child_id > CLIENT_ROOT_ID) {

    if (socket_right != NULL) {

        int replied = 0;

        send_create(socket_right, child_id);

        if (available_receive(socket_right)) {

            reply = receive(socket_right);

            if (strcmp(EMPTY_STR, reply) != 0) {

                replied = 1;

                cout<<reply<<endl;

            }

        }

        if (replied == 0) {

            cout<<"Error: node "<<child_id<<" unavailable"<<endl;

        }

        break;

    }

}

if (child_id < CLIENT_ROOT_ID) {

    if (socket_left != NULL) {

        int replied = 0;

        send_create(socket_left, child_id);

        if (available_receive(socket_left)) {

            reply = receive(socket_left);

            if (strcmp(EMPTY_STR, reply) != 0) {

                replied = 1;

```

```

        cout<<reply<<endl;

    }

}

if (replied == 0) {

    cout<<"Error: node "<<child_id<<" unavailable"<<endl;

}

break;

}

}

break;

case REMOVE:

    remove_id_str = unitread(); //считывание id которое мы хотим удалить

    remove_id = atoi(remove_id_str.c_str());

    if (remove_id <= 0) {

        cout<<"Error: invalid id"<<endl;

        break;

    }

    if (!existNode(system, remove_id)) {

        cout<<"Error: Not found"<<endl;

        break;

    }

    if (CLIENT_ROOT_ID == remove_id) {

        cout<<"Error: can't delete manager root"<<endl;

        break;

    }

    system = deleteNode(system, remove_id);

    if (remove_id > CLIENT_ROOT_ID) {

        int replied = 0;

        send_remove(socket_right, remove_id);

        if (available_receive(socket_right)) {

```

```

        reply = receive(socket_right);

        if (strcmp(EMPTY_STR, reply) != 0) {

            replied = 1;

            cout<<reply<<endl;

        }

    }

    if (replied == 0) {

        cout<<"Error: node "<<child_id<<" unavailable"<<endl;

    }

    break;

}

else if (remove_id < CLIENT_ROOT_ID) {

    int replied = 0;

    send_remove(socket_left, remove_id);

    if (available_receive(socket_left)) {

        reply = receive(socket_left);

        if (strcmp(EMPTY_STR, reply) != 0) {

            replied = 1;

            cout<<reply<<endl;

        }

    }

    if (replied == 0) {

        cout<<"Error: node "<<child_id<<" unavailable"<<endl;

    }

    break;

}

break;

case EXEC: {

    exec_id_str = unitread();

    exec_id = atoi(exec_id_str.c_str());

    char* send_key = (char*)calloc(1, sizeof(char));

```

```

char str_value[64];

char c;

int check = 1;

int cnt_args = 1;

int it = 0;

int cur_size = 1;

while((c = getchar())) { //считываем ключ значение

    if (c == '\n') {

        if (cnt_args == 1) {

            check = 0;

        }

        break;

    }

    if (c == ' '){

        cnt_args++;

        it = 0;

        continue;

    }

    if (cnt_args == 1) {

        send_key[it] = c;

        cur_size ++;

        send_key = (char*)realloc(send_key, cur_size * sizeof(char));

    } else {

        str_value[it] = c;

    }

    it++;

}

if (check == 1) {

    value = stoi(str_value);

    save = 1;

```



```

    } else {

        value = 0;

        save = 0;

    }

    if (exec_id <= 0) {

        cout<<"Error: invalid id"<<endl;

        break;

    }

    if (!existNode(system, exec_id)) {

        cout<<"Error: Not found"<<endl;

        break;

    }

    if (CLIENT_ROOT_ID == exec_id) {

        cout<<"Error: it is a manager root"<<endl;

        break;

    }

    if (exec_id > CLIENT_ROOT_ID) {

        int replied = 0;

        send_exec(socket_right, exec_id, send_key, value, save);

        if (available_receive(socket_right)) {

            reply = receive(socket_right);

            if (strcmp(EMPTY_STR, reply) != 0) {

                replied = 1;

                cout<<reply<<endl;

            }

        }

        if (replied == 0) {

            cout<<"Error: node "<<exec_id<<" unavailable"<<endl;

        }
    }

```

```

        break;
    }

    else if (exec_id < CLIENT_ROOT_ID) {

        int replied = 0;

        send_exec(socket_left, exec_id, send_key, value, save);

        if (available_receive(socket_left)) {

            reply = receive(socket_left);

            if (strcmp(EMPTY_STR, reply) != 0) {

                replied = 1;

                cout<<reply<<endl;

            }

        }

        if (replied == 0) {

            cout<<"Error: node "<<exec_id<<" unavailable"<<endl;

        }

        break;

    }

    break;

}

case PING: {

    ping_id_str = unitread();

    ping_id = atoi(ping_id_str.c_str());

    if (ping_id <= 0) {

        cout<<"Error: invalid id"<<endl;

        break;

    }

    if (!existNode(system, ping_id)) {

        cout<<"Error: Not found"<<endl;

        break;

    }

    if (CLIENT_ROOT_ID == ping_id) {

```

```

        cout<<"Error: it is a manager root"<<endl;

        break;
    }

    if (ping_id > CLIENT_ROOT_ID) {

        int replied = 0;

        send_ping(socket_right, ping_id);

        if (available_receive(socket_right)) {

            reply = receive(socket_right);

            if (strcmp(EMPTY_STR, reply) != 0) {

                replied = 1;

                cout<<reply<<endl;

            }

        }

        if (replied == 0) {

            cout<<"OK: 0"<<endl;

        }

        break;
    }

    else if (ping_id < CLIENT_ROOT_ID) {

        int replied = 0;

        send_ping(socket_left, ping_id);

        if (available_receive(socket_left)) {

            reply = receive(socket_left);

            if (strcmp(EMPTY_STR, reply) != 0) {

                replied = 1;

                cout<<reply<<endl;

            }

        }

        if (replied == 0) {

            cout<<"OK: 0"<<endl;

        }
    }

```

```

        break;

    }

    break;

}

case EXIT:

    if (socket_right != NULL) {

        send_exit(socket_right);

    }

    if (socket_left != NULL) {

        send_exit(socket_left);

    }

    ex = 1;

    break;

}

if (ex == 1) {

    break;

}

free(reply);

}

zmq_close(socket_right);

zmq_close(socket_left);

zmq_ctx_destroy(context);

}server.cpp
#include "config.h"

int main(int argc, const char** argv) //массив строк
{

    void* context = zmq_ctx_new();

```

```

if (context == NULL) {

    throw runtime_error("Error: Can't initialize context");

}

int self_id = atoi(argv[0]);

void* self_socket = zmq_socket(context, ZMQ_REP);

if (self_socket == NULL) {

    throw runtime_error("Error: Can't initialize socket");

}

const char* self_adr = convert_adr_server(BASE_PORT + self_id).c_str();


int rc = zmq_bind(self_socket, self_adr); //привязывает сокет к локальной конечной точке, а затем принимает входящие
соединения на этой конечной точке.

assert(rc == 0);

void* socket_left = NULL;

void* socket_right = NULL;

int rm = 0;

int ex = 0;

int id_left, id_right;

map <string, int> LocalDict;

map <string,int> :: iterator it;

while (true) {

    int flag = 0;

    command_type cur_command = DEFAULT;

    int sum = 0;

    int count_args = 0;

    int id_target = 0;

    int* argv;

    int size_arr = 0;

    char* key;

    int value = 0;

```

```

int save = 0;

while (true) {

    rm = 0;

    ex = 0;

    zmq_msg_t piece;

    int get = zmq_msg_init(&piece); //инициализирует объект пустого сообщения

    assert(get == 0);

    get = zmq_msg_rcv(&piece, self_socket, 0);

    assert(get != -1);


    switch (count_args) {

        case 0:

            memcpy(&cur_command, zmq_msg_data(&piece), zmq_msg_size(&piece)); //копируем содержимое одной
области памяти в другую

            break;

        case 1:

            switch (cur_command) {

                case CREATE:

                    memcpy(&id_target, zmq_msg_data(&piece), zmq_msg_size(&piece));

                    break;

                case REMOVE:

                    memcpy(&id_target, zmq_msg_data(&piece), zmq_msg_size(&piece));

                    break;

                case EXEC:

                    memcpy(&id_target, zmq_msg_data(&piece), zmq_msg_size(&piece));

                    break;

                case PING:

                    memcpy(&id_target, zmq_msg_data(&piece), zmq_msg_size(&piece));

                    break;

                default:

                    break;
            }
        }
    }
}

```

```

    }

    break;

case 2:

    switch (cur_command) {

        case EXEC:

            memcpy(key, zmq_msg_data(&piece), zmq_msg_size(&piece));

            zmq_msg_close((&piece));

            if (!zmq_msg_more(&piece)) {

                break;

            }


            save = 1;

            zmq_msg_t piece2;

            get = zmq_msg_init(&piece2);

            get = zmq_msg_recv(&piece2, self_socket, 0);

            memcpy(&value, zmq_msg_data(&piece2), zmq_msg_size(&piece2));

            zmq_msg_close((&piece2));

            flag = 1;


            break;


        default:

            break;

    }

    break;

default:

    throw runtime_error("Error: wrong command received");

    break;

}

zmq_msg_close((&piece));

```

```

count_args++;

if (flag == 1) {

    break;

}

if (!zmq_msg_more(&piece)) {

    break;

}

}

char *reply = (char *)calloc(sizeof(char), 64);

int replied = 0;

if (cur_command == EXIT) {

    if (socket_right != NULL) {

        send_exit(socket_right);

    }

    if (socket_left != NULL) {

        send_exit(socket_left);

    }

    break;

}

if (cur_command == CREATE) {

    int child_id = id_target;

    if ((child_id > self_id) && socket_right == NULL) {

        int fork_pid = fork();

        if (fork_pid == -1) {

            throw runtime_error("Error: fork problem occurred");

        }

    }

}

```



```

if (fork_pid == 0) {

    create_server_node(child_id);

    break;

}

socket_right = zmq_socket(context, ZMQ_REQ);

int opt = 0;

int rc = zmq_setsockopt(socket_right, ZMQ_LINGER, &opt, sizeof(opt));

assert(rc == 0);

if (socket_right == NULL) {

    throw runtime_error("Error: socket not created");

}

rc = zmq_connect(socket_right, convert_adr_client(BASE_PORT + child_id).c_str());

assert(rc == 0);

const char* fork_pid_str = int_to_str(fork_pid);

sprintf(reply, "OK: %s", fork_pid_str);

replied = 1;

} else if ((child_id < self_id) && socket_left == NULL) {

    if (socket_left == NULL) {

        int fork_pid = fork();

        if (fork_pid == -1) {

            throw runtime_error("Error: fork problem occurred");

        }

        if (fork_pid == 0) {

```

```

        create_server_node(child_id);

        break;
    }

    socket_left = zmq_socket(context, ZMQ_REQ);

    int opt = 0;

    int rc = zmq_setsockopt(socket_left, ZMQ_LINGER, &opt, sizeof(opt));

    assert(rc == 0);

    if (socket_left == NULL) {

        throw runtime_error("Error: socket not created");

    }

    rc = zmq_connect(socket_left, convert_adr_client(BASE_PORT + child_id).c_str());

    assert(rc == 0);


    const char* fork_pid_str2 = int_to_str(fork_pid);

    sprintf(reply, "OK: %s", fork_pid_str2);

    replied = 1;

}

}

else if ((child_id > self_id) && socket_right != NULL) {

    send_create(socket_right, child_id);

    if (available_receive(socket_right)) {

        reply = receive(socket_right);

        if (strcmp(EMPTY_STR, reply) != 0) {

            replied = 1;

        }

    }

}

```

```

    }

    if (replied == 0) {

        cout<<"Error: node "<<child_id<<" unavailable"<<endl;

    }

}

else if ((child_id < self_id) && socket_left != NULL) {

    send_create(socket_left, child_id);

    if (available_receive(socket_left)) {

        reply = receive(socket_left);

        if (strcmp(EMPTY_STR, reply) != 0) {

            replied = 1;

        }

    }

    if (replied == 0) {

        cout<<"Error: node "<<child_id<<" unavailable"<<endl;

    }

}

}

if (cur_command == REMOVE) {

    int remove_id = id_target;

    if (remove_id == self_id) {

        if (socket_right != NULL) {

            send_exit(socket_right);

        }

    }

}

```

```

    if (socket_left != NULL) {

        send_exit(socket_left);

    }


    sprintf(reply, "Removed %d", remove_id);

    replied = 1;

    rm = 1;

}

else if (id_target > self_id) {

    if (socket_right != NULL) {

        send_remove(socket_right, remove_id);

        if (available_receive(socket_right)) {

            reply = receive(socket_right);

            if ((strcmp(EMPTY_STR, reply)) != 0) {

                replied = 1;

            }

        }

    }

}

}

else if (id_target < self_id) {

    if (socket_left != NULL) {

        send_remove(socket_left, remove_id);

        if (available_receive(socket_left)) {

            reply = receive(socket_left);

            if ((strcmp(EMPTY_STR, reply)) != 0) {

```

```

        replied = 1;

    }

}

}

}

}

if (cur_command == EXEC) {

    if(id_target == self_id) {

        if (save == 1) {

            LocalDict[key] = value;

            sprintf(reply, "OK: %d: %s %d", id_target, key, value);

            replied = 1;

        }

        if (save == 0) {

            it = LocalDict.find(key);

            if (it == LocalDict.end()) {

                sprintf(reply, "OK: %d: Not found", id_target);

                replied = 1;

            } else {

                sprintf(reply, "OK: %d: %s %d", id_target, key, LocalDict[key]);

                replied = 1;

            }

        }

    }

}

if (id_target > self_id) {

    if (socket_right != NULL) {

        send_exec(socket_right, id_target, key, value, save);

```

```

        if (available_receive(socket_right)) {

            reply = receive(socket_right);

            if ((strcmp(EMPTY_STR, reply)) != 0) {

                replied = 1;

            }

        }

    }

}

if (id_target < self_id) {

    if (socket_left != NULL) {

        send_exec(socket_left, id_target, key, value, save);

        if (available_receive(socket_left)) {

            reply = receive(socket_left);

            if ((strcmp(EMPTY_STR, reply)) != 0) {

                replied = 1;

            }

        }

    }

}

}

if (cur_command == PING) {

    if (id_target == self_id) {

        sprintf(reply, "OK: 1");

        replied = 1;

    }

    if (id_target > self_id) {

        if (socket_right != NULL) {

```

```

        send_ping(socket_right, id_target);

        if (available_receive(socket_right)) {

            reply = receive(socket_right);

            if ((strcmp(EMPTY_STR, reply)) != 0) {

                replied = 1;

            }

        }

    }

}

if (id_target < self_id) {

    if (socket_left != NULL) {

        send_ping(socket_left, id_target);

        if (available_receive(socket_left)) {

            reply = receive(socket_left);

            if ((strcmp(EMPTY_STR, reply)) != 0) {

                replied = 1;

            }

        }

    }

}

if (replied == 0) {

    reply = EMPTY_STR;

}

//отправка ответа

size_t rep_len = strlen(reply) + 1;

zmq_msg_t create_response;

```

```

int rec = zmq_msg_init(&create_response);

assert(rec != -1);

zmq_msg_init_size(&create_response, rep_len);

memcpy(zmq_msg_data(&create_response), reply, rep_len);

zmq_msg_send(&create_response, self_socket, 0);

zmq_msg_close(&create_response);

if (rm == 1) {

    break;

}

if (ex == 1) {

    break;

}

}

zmq_close(self_socket);

zmq_ctx_destroy(context);

}

```

Демонстрация работы программы

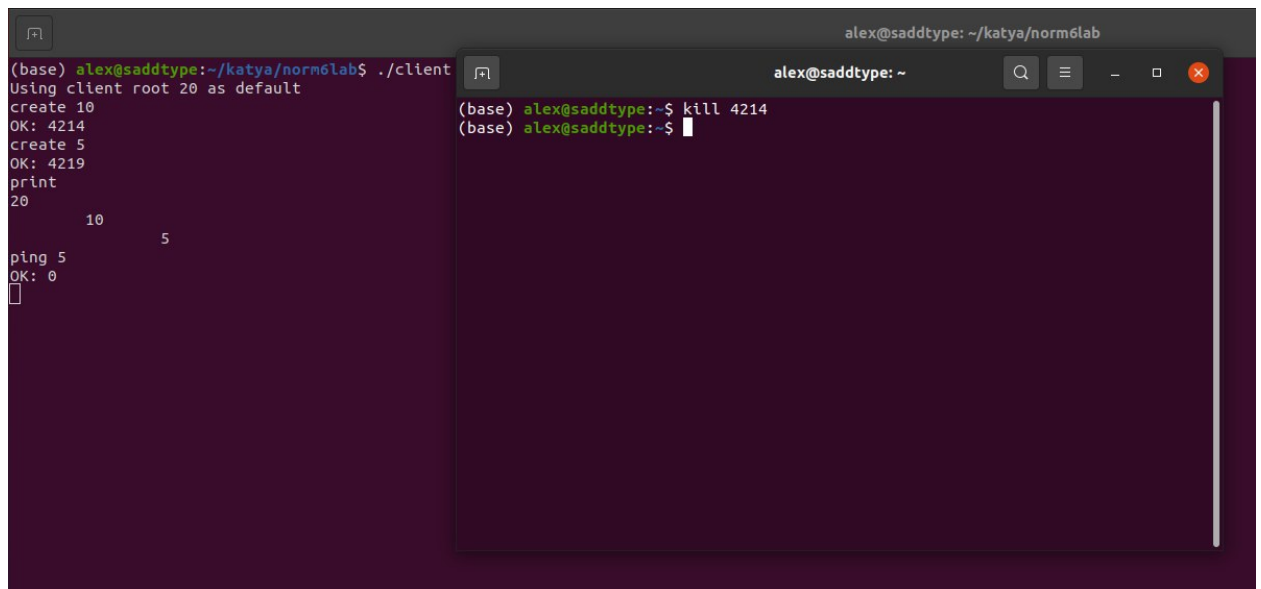
Работаем с сервером в обычном режиме:


```

Using client root 20 as default
create 15
OK: 4457
create 14
OK: 4462
create 20
Error: already exists
create 21
OK: 4465
create 25
OK: 4475
create 24
OK: 4478
create 27
OK: 4481
print
      27
    25
      24
  21
20
  15
    14
exec 25 abc 4
OK: 25: abc 4
exec 25 gg 7
OK: 25: gg 7
exec 27 ghgj 10
OK: 27: ghgj 10
exec 25 abc
OK: 25: abc 4
exec 25 gg
OK: 25: gg 7
exec 27 ghgj
OK: 27: ghgj 10
exec 27 uuu
OK: 27: Not found
remove 24
Removed 24
ping 14
OK: 1
ping 15
OK: 1
remove 15
Removed 15
ping 14
Error: Not found
exit

```

Предположим, что произошел сбой 10-го узла. 5-й узел становится недоступным так как он дочерний к 10-му.



The image shows two terminal windows. The left window, titled 'alex@saddtype: ~/katya/norm6lab', shows the execution of a client program. It starts with the command './client', followed by 'Using client root 20 as default'. Then, it sends 'create 10', 'create 5', and 'ping 5' commands, receiving responses 'OK: 4214', 'OK: 4219', and 'OK: 0' respectively. The right window, titled 'alex@saddtype: ~', shows the command 'kill 4214' being entered, which kills the client process.

```
(base) alex@saddtype:~/katya/norm6lab$ ./client
Using client root 20 as default
create 10
OK: 4214
create 5
OK: 4219
ping 5
OK: 0
```

```
(base) alex@saddtype:~$ kill 4214
(base) alex@saddtype:~$
```

Выводы

Благодаря данной лабораторной работе я приобрел навыки в работе с ZMQ для реализации очередей сообщений между процессами. Я смог реализовать распределенную систему по асинхронной обработке запросов с «управляющим» и «вычислительными» узлами.