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**

<https://github.com/VolkovEvgeny/OS-labs/tree/main/lab6>

**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 ото 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 exec 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 availible\_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 availible\_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\_recv(&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 occured");

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 щшибке

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 soket wieh 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 occured");

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 (availible\_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<<" unavailible"<<endl;

}

break;

}

}

if (child\_id < CLIENT\_ROOT\_ID) {

if (socket\_left != NULL) {

int replied = 0;

send\_create(socket\_left, child\_id);

if (availible\_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<<" unavailible"<<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 (availible\_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<<" unavailible"<<endl;

}

break;

}

else if (remove\_id < CLIENT\_ROOT\_ID) {

int replied = 0;

send\_remove(socket\_left, remove\_id);

if (availible\_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<<" unavailible"<<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 клюс 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 (availible\_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<<" unavailible"<<endl;

}

break;

}

else if (exec\_id < CLIENT\_ROOT\_ID) {

int replied = 0;

send\_exec(socket\_left, exec\_id, send\_key, value, save);

if (availible\_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<<" unavailible"<<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 (availible\_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 (availible\_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\_recv(&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 occured");

}

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: sosocket 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 occured");

}

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 (availible\_receive(socket\_right)) {

reply = receive(socket\_right);

if (strcmp(EMPTY\_STR, reply) != 0) {

replied = 1;

}

}

if (replied == 0) {

cout<<"Error: node "<<child\_id<<" unavailible"<<endl;

}

}

else if ((child\_id < self\_id) && socket\_left != NULL) {

send\_create(socket\_left, child\_id);

if (availible\_receive(socket\_left)) {

reply = receive(socket\_left);

if (strcmp(EMPTY\_STR, reply) != 0) {

replied = 1;

}

}

if (replied == 0) {

cout<<"Error: node "<<child\_id<<" unavailible"<<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 (availible\_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 (availible\_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 (availible\_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 (availible\_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 (availible\_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 (availible\_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;

}

}

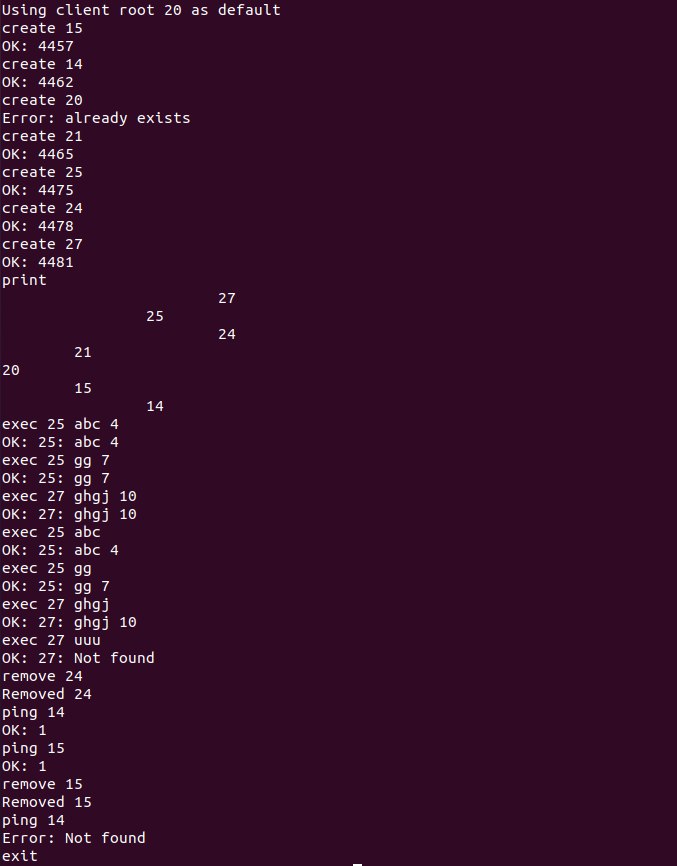
zmq\_close(self\_socket);

zmq\_ctx\_destroy(context);

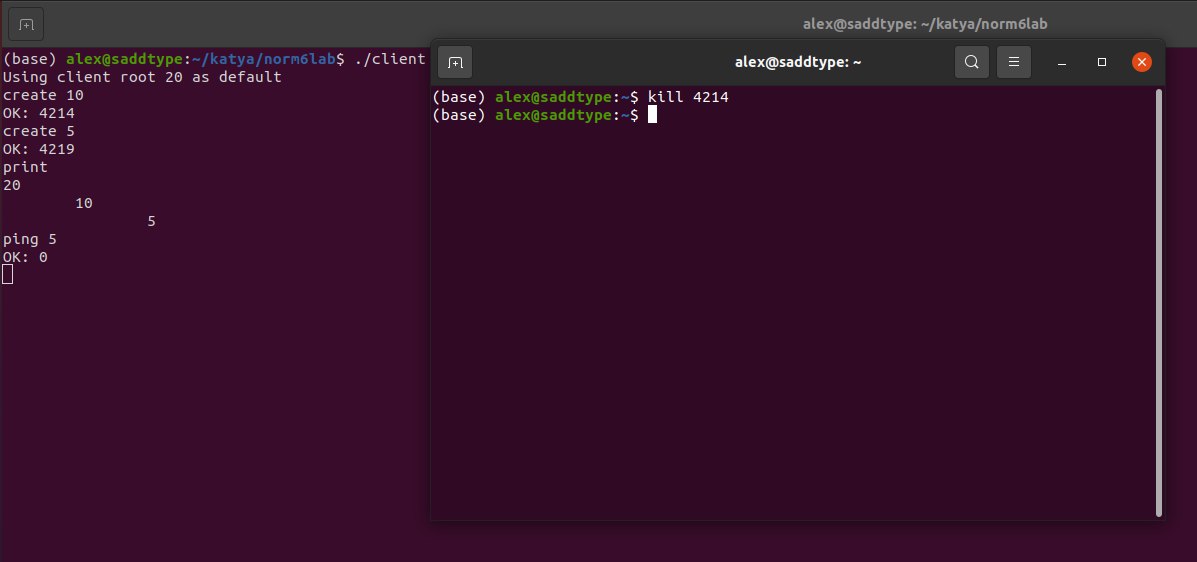
}

**Демонстрация работы программы**

Работаем с сервером в обычном режиме:



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



**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.