

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: Друхольский А.К.  
Группа: М8О-207Б-21  
Вариант: 27  
Преподаватель: Черемисинов Максим  
Оценка: \_\_\_\_\_  
Дата: \_\_\_\_\_  
Подпись: \_\_\_\_\_

Москва, 2022

**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/ssForz/OS-labs>

### **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

interface.h, config.h — интерфейсы для interface.cpp и config.cpp. Tree.h и tree.cpp содержат алгоритмы работы с бинарным деревом и само его представление. Client.cpp — клиент, через который мы взаимодействуем с сервером server.cpp. Для очередей сообщений используем ZeroMQ.

### Общий метод и алгоритм решения

Топология: Все вычислительные узлы хранятся в бинарном дереве поиска.

Команда: `exec id n k1 k2 ... kn` — узел `id` считает сумму `n` целых чисел

пример: `exec 15 2 1 2` — ОК: 15: 3

Проверка доступности узла: `ping id`

пример: `ping 15` — ОК: 1 //работает

`ping 16` — ОК: 0 //не доступен

### 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>
```

```
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;
```

```
string convert_adr_client(unsigned short port);
string convert_adr_server(unsigned short port);
```

```
const char* int_to_str(unsigned a);
void prt_info();
```

```
command_type get_command();  
string unitread();
```

```
#endif
```

### **interface.cpp**

```
#include "interface.h"
```

```
string unitread()
```

```
{  
  
    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;
}

void prt_info()
{
    cout<<"create [id]"<<endl;
}

```



### **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, int size_arr, int* argv);

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;
9
```

```

    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_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, int size_arr, int* argv)
{
    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);
}

```

```

zmq_msg_t size_arr_msg;
zmq_msg_init_size(&size_arr_msg, sizeof(size_arr));
memcpy(zmq_msg_data(&size_arr_msg), &size_arr, sizeof(size_arr));
zmq_msg_send(&size_arr_msg, socket, ZMQ_SNDMORE);
zmq_msg_close(&size_arr_msg);
for (int i = 0; i < size_arr; i++) {
    zmq_msg_t arg;
    zmq_msg_init_size(&arg, sizeof(argv[i]));
    memcpy(zmq_msg_data(&arg), &argv[i], sizeof(argv[i]));
    if (i == size_arr - 1) {
        zmq_msg_send(&arg, socket, 0);
    }
    if (i != size_arr - 1) {
        zmq_msg_send(&arg, socket, ZMQ_SNDMORE);
    }
    zmq_msg_close(&size_arr_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);
    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 20
```

```

int main()
{
    cout<<"Using client root "<<CLIENT_ROOT_ID<<" as default"<<endl;
    Tree* system;
    system = createTree(CLIENT_ROOT_ID);

```

```

void *context = zmq_ctx_new();
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();
    string child_id_str;
    int child_id;
    string remove_id_str;
    int remove_id;
    int exec_id;
    int ping_id;
    string ping_id_str;
    string exec_id_str;
    int* argv;
    int size_arr;
    char *reply = (char *)calloc(sizeof(char), 64);
    switch(cur_command) {
        case PRINT:
            printTree(system,0);
            break;
        case CREATE:
            child_id_str = unitread();
            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);

    cout<<"OK: "<<fork_pid<<endl;
    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);
    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();
    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());
    cin>>size_arr;
    argv = (int*)calloc(sizeof(int), size_arr);
    for (int i = 0; i < size_arr; i++) {

```



```

        cin>>argv[i];
    }
    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, size_arr, argv);
        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, size_arr, argv);
        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)
{
    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);
assert(rc == 0);
void* socket_left = NULL;
void* socket_right = NULL;
int rm = 0;
int ex = 0;
int id_left, id_right;
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;
    while (true) {
        rm = 0;
        ex = 0;
        zmq_msg_t piece;
        int get = zmq_msg_init(&piece);
        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));
                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;
        }
    }
}

```

```

    }
    break;
case 2:
    switch (cur_command) {
        case EXEC:
            memcpy(&size_arr, zmq_msg_data(&piece), zmq_msg_size(&piece));
            zmq_msg_close((&piece));
            if (!zmq_msg_more(&piece)) {
                break;
            }

            argv = (int*)calloc(sizeof(int), size_arr);
            for (int i = 0; i < size_arr; i++) {
                get = zmq_msg_init(&piece);
                assert(get == 0);
                get = zmq_msg_recv(&piece, self_socket, 0);
                assert(get != -1);
                memcpy(&argv[i], zmq_msg_data(&piece), zmq_msg_size(&piece));
                if (!zmq_msg_more(&piece)) {
                    break;
                }
                zmq_msg_close((&piece));
            }
            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) {
        for (int i = 0; i < size_arr; i++) {
            sum += argv[i];
        }
        sprintf(reply, "OK: %d: sum is %d", id_target, sum);
        replied = 1;
    }

    if (id_target > self_id) {
        if (socket_right != NULL) {

            send_exec(socket_right, id_target, size_arr, argv);
            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, size_arr, argv);
            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;
}

```

```

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**

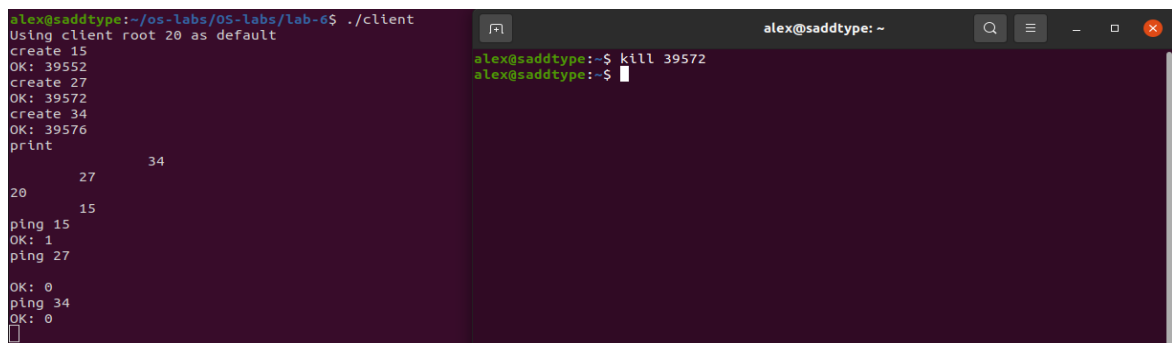
Работаем с сервером в обычном режиме:

```

alex@saddtype:~/os-labs/OS-labs/lab-6$ ./client
Using client root 20 as default
create 15
OK: 39387
create 16
OK: 39392
create 24
OK: 39400
create 23
OK: 39403
create 21
OK: 39406
print
      24
    23
  21
20
      16
    15
remove 23
Removed 23
print
      24
20
      16
    15
exec 24 4 1 2 3 4
OK: 24: sum is 10
exec 15 2 1 2
OK: 15: sum is 3

```

Предположим сбой 27-го узла, 15-й всё ещё работает. 34-й недоступен так как он дочерний к 27-му.



```

alex@saddtype:~/os-labs/OS-labs/lab-6$ ./client
Using client root 20 as default
create 15
OK: 39552
create 27
OK: 39572
create 34
OK: 39576
print
      34
    27
20
      15
ping 15
OK: 1
ping 27
OK: 0
ping 34
OK: 0

```

```

alex@saddtype:~$ kill 39572
alex@saddtype:~$

```

## Выводы

В данной лабораторной работе я узнал про очереди сообщений, познакомился с zero mq api. Данная лабораторная работа была весьма сложной и постоянно приходилось заниматься отладкой, что так же в какой-то степени развило мои навыки программирования. Так же я более детально изучил процессы на Linux.

