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»

ОЧЕРЕДИ СООБЩЕНИЙ

Студент: Белоносов Кирилл Алексеевич
Группа: М8О–208Б–21
Вариант: 38
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022.

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (№6)
- Применение отложенных вычислений (№7)
- Интеграция программных систем друг с другом (№8)

Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

Создание нового вычислительного узла

Формат команды: create id [parent]

id – целочисленный идентификатор нового вычислительного узла

parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода: «Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла

«Error: Already exists» - вычислительный узел с таким идентификатором уже существует

«Error: Parent not found» - нет такого родительского узла с таким идентификатором

«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться

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

Пример:

> create 10 5

Ok: 3128

Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.

Удаление существующего вычислительного узла
Формат команды: remove id
id – целочисленный идентификатор удаляемого вычислительного узла

Формат вывода:
«Ok» - успешное удаление
«Error: Not found» - вычислительный узел с таким идентификатором не найден
«Error: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом
«Error: [Custom error]» - любая другая обрабатываемая ошибка
Пример:
> remove 10
Ok
Примечание: при удалении узла из топологии его процесс должен быть завершен и работоспособность вычислительной сети не должна быть нарушена.
Исполнение команды на вычислительном узле
Формат команды: exec id [params]
id – целочисленный идентификатор вычислительного узла, на который отправляется команда
Формат вывода:
«Ok:id: [result]», где result – результат выполненной команды
«Error:id: Not found» - вычислительный узел с таким идентификатором не найден
«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом
«Error:id: [Custom error]» - любая другая обрабатываемая ошибка
Пример:
Можно найти в описании конкретной команды, определенной вариантом задания.
Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Общие сведения о программе

Основная программа компилируется из файла client.c. Также подключаются файлы zmq_struct.c — функции для работы с очередью. Server.c представляет собой вычислительную ноду к которой также подключается zmq_struct.c.
Системные вызовы:

1. **zmq_ctx_new() -** создать новый контекст
2. **zmq_socket() -** создать новый сокет
3. **zmq_setsockopt() -** задать настройки сокета
4. **zmq_bind() -** настраивает сокет на определенный IP-адрес и порт
5. **zmq_connect() -** подключает сокет к определенную адресу, сокет становится клиентским
6. **zmq_close() -** закрыть соединение по сокету
7. **zmq_ctx_destroy() -** уничтожить контекст
8. **zmq_msg_init() -** инициализировать новое сообщение zeromq
9. **zmq_msg_init_size() -** инициализировать размер сообщения
10. **zmq_msg_init_data() -** инициализировать данные сообщения
11. **zmq_msg_send() -** отправить сообщение
12. **zmq_msg_recv() -** получить сообщение
13. **zmq_msg_close() -** уничтожить сообщение

Общий метод и алгоритм решения.

ZeroMQ представляет собой брокер сообщений, каждый раз, когда мы связываем два сокета между ними возникает очередь сообщений. Реализуем главный узел, который будет отвечать за создание новых узлов и удаление старых. Реализуем создание новых узлов, с помощью системного вызова fork(), с помощью execlp() будем запускать экземпляры программ server. Также будем поддерживать древовидную структуру узлов, что помогает быстро найти нужную ноду. Также реализовал обработчик сигналов, который при завершении работы сервера, выключает все вычислительные ноды.

Основные файлы программы

zmq_struct.c:

```
#include "zmq_struct.h"
#include <zmq.h>
#include <assert.h>
#include <string.h>
```

```
#include <errno.h>

int create_conn(int id, void * socket, void * context) {
    int rc;
    char path[40] = "tcp://*:";
    int new_port = id + 5555;
    char port[40];
    sprintf(port, "%d", new_port);
    strcat(path, port);
    rc = zmq_bind(socket, path);
    assert(rc == 0);
    return 1;
}
int ping_send(void * socket, message_zmq * msg) {
    int rc;
    zmq_msg_t message;
    zmq_msg_init(&message);
    rc = zmq_msg_init_size(&message, sizeof(message_zmq));
    assert(rc == 0);
    rc = zmq_msg_init_data(&message, msg, sizeof(message_zmq), NULL, NULL);
    assert(rc == 0);
    rc = zmq_msg_send(&message, socket, 0);
    if (rc == -1) {
        zmq_msg_close(&message);
        printf("Error: sending timeout\n");
        return 0;
    }
    return 1;
}

message_zmq * ping_receive(void * socket) {
    int rc;
    zmq_msg_t ans;
```

```
    zmq_msg_init(&ans);
    rc = zmq_msg_recv(&ans, socket, 0);
    if (rc == -1) {
        zmq_msg_close(&ans);
        return NULL;
    }
    assert(rc == sizeof(message_zmq));
    message_zmq * data = (message_zmq *)zmq_msg_data(&ans);
    rc = zmq_msg_close(&ans);
    assert(rc == 0);
    return data;
}
void send_string(void *socket, const char *string) {
            zmq_send(socket, strdup(string), strlen(string), 0);
}
char * recv_string (void *socket, long long size) {
    char buffer[size];
    zmq_recv (socket, buffer, size-1, 0);
    buffer [size] = '\0';
    return strdup (buffer);
}
```

zmq_struct.h

```
#ifndef ZMQ_STRUCT
#define ZMQ_STRUCT

typedef struct
{
    int key;
    int id;
    long long size;
} message_zmq;

enum Status{
```

```
    CHECK,
    PING,
    CREATE,
    REMOVE,
    EXEC,
    SUCCESS,
    FAIL,
    FAIL_PING,
    FAIL_CREATE
};

int ping_send(void * socket, message_zmq * msg);
message_zmq * ping_receive(void * socket);
int create_conn(int id, void * socket, void * context);
void send_string(void *socket, const char *string);
char * recv_string (void *socket, long long size);
#endif
```

server.c

```
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include "library/zmq_struct.h"
#include <signal.h>


int do_something = 1;

void sig_handler(int signum){
    do_something = 0;
```

```
}

const int WAIT_TIME = 1000;

int main (int argc, char * argv[])
{
    int rc;
    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_PAIR);
    assert(argc == 2);
    int id = atoi(argv[1]);
    int new_port = id + 5555;
    char port[40];
    sprintf(port, "%d", new_port);
    char path[40] = "tcp://localhost:";
    strcat(path, port);
    rc = zmq_connect(socket, path);
    assert(rc == 0);
    void * right_context;
    void * right_socket;
    int right_tree = 0;
    void * left_context;
    void * left_socket;
    int left_tree = 0;
    while (do_something) {
        message_zmq * data = ping_receive(socket);
        if(data != NULL) {
            int new_id = data->id;
            int new_size = data->size;
            if(data->key == CHECK) {
                message_zmq answer = {SUCCESS, getpid()};
                ping_send(socket, &answer);
            }
```

```
        if(data->key == CREATE) {
            if(new_id < id) {
                if(left_tree == 0) {
                    int rc;
                    left_context = zmq_ctx_new();
                    left_socket = zmq_socket(left_context, ZMQ_PAIR);
                    rc = zmq_setsockopt(left_socket, ZMQ_RCVTIMEO, &WAIT_TIME,
sizeof(int));
                    assert(rc == 0);
                    rc = zmq_setsockopt(left_socket, ZMQ_SNDTIMEO, &WAIT_TIME,
sizeof(int));
                    assert(rc == 0);
                    create_conn(new_id, left_socket, left_context);
                    char id_str[12];
                    sprintf(id_str, "%d", new_id);
                    if(fork() == 0) {
                        execlp("./server", "server", id_str, NULL);
                        perror("execlp");
                        exit(EXIT_FAILURE);
                    } else {
                        message_zmq reply = {CHECK, getpid()};
                        ping_send(left_socket, &reply);
                        message_zmq * new_data = ping_receive(left_socket);
                        if(new_data != NULL) {
                            int pid = new_data->id;
                            if(new_data->key == SUCCESS) {
                                left_tree = new_id;
                                message_zmq new_reply = {SUCCESS, pid};
                                ping_send(socket, &new_reply);
                            } else {
                                rc = zmq_close(left_socket);
                                assert(rc == 0);
                                rc = zmq_ctx_destroy(left_context);
                                assert(rc == 0);
```

```
                        message_zmq new_reply = {FAIL_CREATE, pid};
                        ping_send(socket, &new_reply);
                    }
                } else {
                    message_zmq new_reply = {FAIL_PING, new_id};
                    ping_send(socket, &new_reply);
                }
            }
        }
        else {
            message_zmq reply = {CREATE, new_id};
            ping_send(left_socket, &reply);
            message_zmq * new_data = ping_receive(left_socket);
            if(new_data != NULL) {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            } else {
                message_zmq new_reply = {FAIL_PING, new_id};
                ping_send(socket, &new_reply);
            }
        }
    } else if(new_id == id) {
        message_zmq reply = {FAIL, getpid()};
        ping_send(socket, &reply);
    } else {
        if(right_tree == 0) {
            int rc;
            right_context = zmq_ctx_new();
            right_socket = zmq_socket(right_context, ZMQ_PAIR);
            rc = zmq_setsockopt(right_socket, ZMQ_RCVTIMEO, &WAIT_TIME,
sizeof(int));
            assert(rc == 0);
            rc = zmq_setsockopt(right_socket, ZMQ_SNDTIMEO, &WAIT_TIME,
sizeof(int));
```

```
    assert(rc == 0);
    create_conn(new_id, right_socket, right_context);
    char id_str[10];
    sprintf(id_str, "%d", new_id);
    if(fork() == 0) {
        execlp("./server", "server", id_str, NULL);
        perror("execlp");
        exit(EXIT_FAILURE);
    } else {
        message_zmq reply = {CHECK, getpid()};
        ping_send(right_socket, &reply);
        message_zmq * new_data = ping_receive(right_socket);
        if(new_data != NULL) {
            int pid = new_data->id;
            if(new_data->key == SUCCESS) {
                right_tree = new_id;
                message_zmq new_reply = {SUCCESS, pid};
                ping_send(socket, &new_reply);
            } else {
                rc = zmq_close(right_socket);
                assert(rc == 0);
                rc = zmq_ctx_destroy(right_context);
                assert(rc == 0);
                message_zmq new_reply = {FAIL_CREATE, pid};
                ping_send(socket, &new_reply);
            }
        }
        else {
            message_zmq new_reply = {FAIL_PING, new_id};
            ping_send(socket, &new_reply);
        }
    }
}
```

```
        else {
            message_zmq reply = {CREATE, new_id};
            ping_send(right_socket, &reply);
            message_zmq * new_data = ping_receive(right_socket);
            if(new_data != NULL) {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            } else {
                message_zmq new_reply = {FAIL_PING, new_id};
                ping_send(socket, &new_reply);
            }
        }
    }
}
if(data->key == REMOVE) {
    if(new_id < id) {
        if(left_tree == 0) {
            message_zmq reply = {FAIL, getpid()};
            ping_send(socket, &reply);
        }
        else if(left_tree == new_id) {
            message_zmq reply = {REMOVE, left_tree};
            ping_send(left_socket, &reply);
            message_zmq * new_data = ping_receive(left_socket);
            if(new_data != NULL) {
                if(new_data->key == SUCCESS) {
                    message_zmq msg = *new_data;
                    ping_send(socket, &msg);
                    left_tree = 0;
                    rc = zmq_close(left_socket);
                    assert(rc == 0);
                    rc = zmq_ctx_destroy(left_context);
                    assert(rc == 0);
```

```
            } else {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            }
        }
        else {
            message_zmq new_reply = {FAIL_PING, new_id};
            ping_send(socket, &new_reply);
        }
    }
    else {
        message_zmq reply = {REMOVE, new_id};
        ping_send(left_socket, &reply);
        message_zmq * new_data = ping_receive(left_socket);
        if(new_data != NULL) {
            message_zmq msg = *new_data;
            ping_send(socket, &msg);
        } else {
            message_zmq new_reply = {FAIL_PING, new_id};
            ping_send(socket, &new_reply);
        }
    }
}
else if(new_id == id) {
    if(left_tree != 0) {
        message_zmq reply = {REMOVE, left_tree};
        ping_send(left_socket, &reply);
        message_zmq * new_data = ping_receive(left_socket);
        if(new_data != NULL) {
            message_zmq msg = *new_data;
            ping_send(socket, &msg);
        } else {
            message_zmq new_reply = {FAIL_PING, new_id};
```

```
            ping_send(socket, &new_reply);
        }
    }
    if(right_tree != 0) {
        message_zmq reply = {REMOVE, right_tree};
        ping_send(right_socket, &reply);
        message_zmq * new_data = ping_receive(right_socket);
        if(new_data != NULL) {
            message_zmq msg = *new_data;
            ping_send(socket, &msg);
        } else {
            message_zmq new_reply = {FAIL_PING, new_id};
            ping_send(socket, &new_reply);
        }
    }
    do_something = 0;
    message_zmq reply = {SUCCESS, getpid()};
    ping_send(socket, &reply);
}
else {
    if(right_tree == 0) {
        message_zmq reply = {FAIL, getpid()};
        ping_send(socket, &reply);
    }
    else if(right_tree == new_id) {
        message_zmq reply = {REMOVE, right_tree};
        ping_send(right_socket, &reply);
        message_zmq * new_data = ping_receive(right_socket);
        if(new_data != NULL) {
            if(new_data->key == SUCCESS) {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
                right_tree = 0;
```

```
                rc = zmq_close(right_socket);
                assert(rc == 0);
                rc = zmq_ctx_destroy(right_context);
                assert(rc == 0);
            } else {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            }
        }
        else {
            message_zmq new_reply = {FAIL_PING, new_id};
            ping_send(socket, &new_reply);
        }
    }
    else {
        message_zmq reply = {REMOVE, new_id};
        ping_send(right_socket, &reply);
        message_zmq * new_data = ping_receive(right_socket);
        if(new_data != NULL) {
            message_zmq msg = *new_data;
            ping_send(socket, &msg);
        } else {
            message_zmq new_reply = {FAIL_PING, new_id};
            ping_send(socket, &new_reply);
        }
    }
  }
}
if(data->key == PING) {
  if(new_id < id) {
    if(left_tree == 0) {
        message_zmq reply = {FAIL, 0};
        ping_send(socket, &reply);
```

```
    }
    else if(left_tree == new_id) {
        message_zmq reply = {CHECK, new_id};
        if(ping_send(left_socket, &reply)) {
            message_zmq * new_data = ping_receive(left_socket);
            if(new_data != NULL) {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            } else {
                message_zmq new_data = {FAIL_PING, new_id};
                ping_send(socket, &new_data);
            }
        }
        else {
            message_zmq new_data = {FAIL_PING, new_id};
            ping_send(socket, &new_data);
        }
    }
    else {
        message_zmq reply = {PING, new_id};
        ping_send(left_socket, &reply);
        message_zmq * new_data = ping_receive(left_socket);
        if(new_data != NULL) {
            message_zmq msg = *new_data;
            ping_send(socket, &msg);
        } else {
            message_zmq new_data = {FAIL_PING, new_id};
            ping_send(socket, &new_data);
        }
    }
}
else if(new_id == id) {
    message_zmq reply = {SUCCESS, 1};
```

```
    ping_send(socket, &reply);
}
else {
    if(right_tree == 0) {
        message_zmq reply = {FAIL, 2};
        ping_send(socket, &reply);
    }
    else if(right_tree == new_id) {
        message_zmq reply = {CHECK, new_id};
        if(ping_send(right_socket, &reply)) {
            message_zmq * new_data = ping_receive(right_socket);
            if(new_data != NULL) {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            } else {
                message_zmq new_data = {FAIL_PING, new_id};
                ping_send(socket, &new_data);
            }
        }
        else {
            message_zmq new_data = {FAIL_PING, new_id};
            ping_send(socket, &new_data);
        }
    }
    else {
        message_zmq reply = {PING, new_id};
        ping_send(right_socket, &reply);
        message_zmq * new_data = ping_receive(right_socket);
        if(new_data != NULL) {
            message_zmq msg = *new_data;
            ping_send(socket, &msg);
        } else {
            message_zmq new_data = {FAIL_PING, new_id};
```

```
                ping_send(socket, &new_data);
            }
        }
    }
}
if(data->key == EXEC) {
    char * query = recv_string(socket, new_size);
    if(new_id < id) {
        if(left_tree == 0) {
            message_zmq reply = {FAIL, new_id};
            ping_send(socket, &reply);
        }
        else {
            message_zmq reply = {EXEC, new_id, new_size};
            ping_send(left_socket, &reply);
            send_string(left_socket, query);
            message_zmq * new_data = ping_receive(left_socket);
            if(new_data != NULL) {
                message_zmq msg = *new_data;
                ping_send(socket, &msg);
            } else {
                message_zmq new_data = {FAIL_PING, new_id};
                ping_send(socket, &new_data);
            }
        }
    }
    else if(new_id == id) {
        strtok(query, " ");
        int n = atoi(query);
        long long sum = 0;
        for(int i = 0; i < n; ++i) {
            query = strtok(NULL, " ");
            int a = atoi(query);
```

```
                sum += a;
            }
            message_zmq reply = {SUCCESS, new_id, sum};
            ping_send(socket, &reply);
        }
        else {
            if(right_tree == 0) {
                message_zmq reply = {FAIL, new_id};
                ping_send(socket, &reply);
            }
            else {
                message_zmq reply = {EXEC, new_id, new_size};
                ping_send(right_socket, &reply);
                send_string(right_socket, query);
                message_zmq * new_data = ping_receive(right_socket);
                if(new_data != NULL) {
                    message_zmq msg = *new_data;
                    ping_send(socket, &msg);
                } else {
                    message_zmq new_data = {FAIL_PING, new_id};
                    ping_send(socket, &new_data);
                }
            }
        }
    }
    else {
        message_zmq new_reply = {FAIL_PING, id};
        ping_send(socket, &new_reply);
    }
}
rc = zmq_close(socket);
assert(rc == 0);
```

```
    rc = zmq_ctx_destroy(context);
    assert(rc == 0);
    return 0;
}
```

client.c

```
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <assert.h>
#include "library/zmq_struct.h"
#include <signal.h>

const int WAIT_TIME = 1000;

int process = 1;
int sig = 0;
int children = 0;
void * context;
void * socket;

void sig_handler(int signum){
    process = 0;
    sig = 1;
}

int main (int argc, char * argv[])
{
    signal(SIGINT, sig_handler);
    char * s = NULL;
    size_t len = 0;
    while(process) {
```

```
getline(&s, &len, stdin);
strtok(s, " ");
if(!strcmp(s,"create")) {
    s = strtok(NULL, " ");
    int id = atoi(s);
    if(children == 0) {
        int rc;
        context = zmq_ctx_new();
        socket = zmq_socket(context, ZMQ_PAIR);
        rc = zmq_setsockopt(socket, ZMQ_RCVTIMEO, &WAIT_TIME, sizeof(int));
        assert(rc == 0);
        rc = zmq_setsockopt(socket, ZMQ_SNDTIMEO, &WAIT_TIME, sizeof(int));
        assert(rc == 0);
        create_conn(id, socket, context);
        char id_str[10];
        sprintf(id_str, "%d", id);
        if(fork() == 0) {
            execlp("./server", "server", id_str, NULL);
            perror("execlp");
            exit(EXIT_FAILURE);
        } else {
            message_zmq reply = {CHECK, 0};
            ping_send(socket, &reply);
            message_zmq * data = ping_receive(socket);
            if(data->key == SUCCESS) {
                printf("OK:%d\n", data->id);
                children = id;
            }
            if(data->key == FAIL_CREATE){
                printf("Error: Port is busy\n");
                rc = zmq_close(socket);
                assert(rc == 0);
                rc = zmq_ctx_destroy(context);
```

```
                assert(rc == 0);
            }
        }
    } else {
        message_zmq reply = {CREATE, id};
        ping_send(socket, &reply);
        message_zmq * data = ping_receive(socket);
        if(data->key == SUCCESS) {
            printf("OK:%d\n", data->id);
        }
        if(data->key == FAIL) {
            printf("Error: Already exists\n");
        }
        if(data->key == FAIL_CREATE || data->key == FAIL_PING) {
            printf("Error: Port is busy\n");
        }
    }
}
if(!strcmp(s,"remove")) {
    s = strtok(NULL, " ");
    int id = atoi(s);
    if(children == 0) {
        printf("Error: Not found\n");
    }
    else if(children == id) {
        message_zmq reply = {REMOVE, id};
        ping_send(socket, &reply);
        message_zmq * data = ping_receive(socket);
        if(data->key == FAIL) {
            printf("Error: Not found\n");
        }
        if(data->key == SUCCESS) {
            int rc;
```

```
            printf("Ok\n");
            rc = zmq_close(socket);
            assert(rc == 0);
            rc = zmq_ctx_destroy(context);
            assert(rc == 0);
            children = 0;
        }
    }
    else {
        message_zmq reply = {REMOVE, id};
        ping_send(socket, &reply);
        message_zmq * data = ping_receive(socket);
        if(data->key == FAIL) {
            printf("Error: Not found\n");
        }
        if(data->key == FAIL_PING) {
            printf("Error: Node is unavailable\n");
        }
        if(data->key == SUCCESS) {
            printf("Ok\n");
        }
    }
}
if(!strcmp(s,"ping")) {
    s = strtok(NULL, " ");
    int id = atoi(s);
    if(children == 0) {
        printf("Error: Not found\n");
    }
    else {
        message_zmq reply = {PING, id};
        ping_send(socket, &reply);
        message_zmq * data = ping_receive(socket);
```

```
        if(data->key == FAIL) {
            printf("Error: Not found\n");
        }
        if(data->key == SUCCESS) {
            printf("Ok: 1\n");
        }
        if(data->key == FAIL_PING) {
            printf("Ok: 0\n");
        }
    }
}
if(!strcmp(s,"exec")) {
    s = strtok(NULL, " ");
    int id = atoi(s);
    s = strtok(NULL, " ");
    int n = atoi(s);
    char * query = s;
    for(int i = 0; i < n; ++i) {
        s = strtok(NULL, " ");
        sprintf(query, "%s %s",query, s);
    }
    int size = strlen(query);
    if(children == 0) {
        printf("Error: Not found\n");
    }
    else {
        message_zmq reply = {EXEC, id, size};
        ping_send(socket, &reply);
        send_string(socket, query);
        message_zmq * data = ping_receive(socket);
        if(data->key == FAIL) {
            printf("Error: Not found\n");
        }
```

```
            if(data->key == FAIL_PING) {
                printf("Error: Node is unavailable\n");
            }
            if(data->key == SUCCESS) {
                printf("OK:%d:%lld\n", data->id, data->size);
            }
        }
    }
    if(!strcmp(s,"down\n")) {
        process = 0;
    }
    while((s = strtok(NULL, " ")) != NULL);
}
if (children != 0 && sig) {
    int rc;
    rc = zmq_close(socket);
    assert(rc == 0);
    rc = zmq_ctx_destroy(context);
    assert(rc == 0);
    children = 0;
}
if(children != 0 && !sig) {
    message_zmq reply = {REMOVE, children};
    ping_send(socket, &reply);
    message_zmq * data = ping_receive(socket);
    if(data->key == SUCCESS) {
        int rc;
        rc = zmq_close(socket);
        assert(rc == 0);
        rc = zmq_ctx_destroy(context);
        assert(rc == 0);
        children = 0;
    }
```

```
    }
    printf("Server is down\n");
    return 0;
}
```

Пример работы

```
kirill@kirill:~/localProjects/OSlabs/lab6/test$ ./run.sh
[ 33%] Built target zmq_s
[ 66%] Built target client
[100%] Built target server
------------------------test 1------------------------
OK:4425
OK:4430
OK:4435
OK:4440
OK:4445
Ok
Ok: 1
Ok: 1
Error: Not found
Error: Not found
Server is down
------------------------test 2------------------------
OK:4451
OK:132:1
OK:132:0
Error: Not found
OK:4456
OK:200:9
Server is down
```

kirill@kirill:~/localProjects/OSlabs/lab6/test$

Вывод

В результате данной лабораторной работы я познакомился с библиотекой ZeroMQ, изучил концепцию очередей сообщений, а также паттерны взаимодействия между узлами очереди сообщений. Реализовал структуру дерева узлов, которые могут выполнять отложенные операции. Использование очередей сообщений позволяет повысить эффективность работы, разбить вычисления на несколько узлов. Но в тоже время это повышает сложность системы и возможности её отладки. Также изученные методы могут пригодиться в разработке высоконагруженных систем.