Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы №6-8 по курсу

«Операционные системы»

СЕРВЕРА СООБЩЕНИЙ

Студент: Суханов Е.А.
Группа: М8О–206Б–19
Вариант:23
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

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»;
- «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], где 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.

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, где 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.

  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], где 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;

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.

Мой вариант: 23.

Топология: Вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел (корень дерева). Данный узел имеет идентификатор -1.

Тип команды: Поиск подстроки в строке.
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;

Тип проверки доступности: 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

`

Общие сведения о программе

Программа состоит из нескольких модулей:

Я решил использовать ZMQ, так как он мне показался достаточно удобным. Для использования его функций нужно создать контекст: zmq::context_t.

Для отправки и приема сообщений используются сокеты: zmq::socket_t. Чтобы создать сокет, нужно указать контекст и типа сокета. Я использую только два типа сокетов ZMQ_PUB и ZMQ_SUB. Они реализуют паттерн “издатель - подписчик”. То есть некий издатель (сокет) может только отправлять сообщения. А подписчик (другой сокет) может только принимать сообщения. При этом, подписчик может подписаться на несколько издателей и принимать от них сообщения. В свою очередь, издатели могут иметь несколько подписчиков.

Для подключения одного сокета к другому, нужно указать способ общения и адрес подключения. Я выбрал межпроцессорное взаимодействие в качестве способа общения, так как он достаточно простой, удобный и быстрый. Он использует unix сокеты для общения между процессами.

Для серверов, то есть сокетов, к которым подключаются, используется функция bind, где указывается адрес, на котором будет работать данный сокет.

Для клиентов, то есть сокетов, которые подключаются, используется функция connect, где указывается адрес, к которому данный сокет будет подключаться.

Я использую функцию setsockopt. Она нужна для того, чтобы поменять какое-то свойство сокета. Например, для ZMQ_SUB сокетов, обязательным для установки является параметр ZMQ_SUBSCRIBE который определяет префикс, с которого должно начинаться сообщение, чтобы данный

подписчик посчитал нужным его принять. Кроме того, я меняю свойство ZMQ_RCVTIMEO, чтобы установить максимальное время ожидания приема сообщения.

Для отправки и приема сообщений я использую функции send и recv для соответствующих сокетов. Они отправляют и принимают объект типа zmq::message_t. Данный класс является оберткой над бинарными данными, он позволяет получить сообщение неопределенного размера, а также отправить. В последнем случае, при создании экземпляра класса, нужно задать размер сообщения в байтах. Далее можно получить указатель на буфер и заполнить его по своему усмотрению.

Для отключения используются соотвествтующие команды unbind (для bind) и disconnect (для connect). Которые отключают сокет от указанного адреса.

Для закрытия сокетов используется функция close.

- Server - он же пользовательский узел. Именно его и нужно запускать. Он выполняет введенные команды, служит окном для взаимодействия с вычислительными узлами. Он имеет два сокета для взаимодействия с детьми. Сокет для отправки сообщений и для их приема;
- Client - он же вычислительный узел. Напрямую пользователем он не запускается. Только через пользовательский узел. Он ожидает приема сообщений, обрабатывает и/или отсылает их дальше;
- Message - Вспомогательный модуль, который задает формат сообщений. С помощью функций fill_* - можно получить экземпляр класса zmq::message_t, готовое к отправке. А с помощью соответствующих функций get_* и get_header получить содержимое сообщения в удобном для работы формате;
- Zmq_tools - Вспомогательный модуль. Содержит функции для генерации адресов (имен сокетов).

Для компиляции файлов необходимо использовать библиотеку libzmq.

Программа запускается, командой ./server.

Общий метод и алгоритм решения

В программе server происходит стандартное считывание и выполнение пользовательских команд. Для каждой команды создан соответствующий обработчик.

1. Create. Для создания новго узла указывается индекс родителя и новой вершины. Затем проверяется доступность родителя и недоступность новой вершины. Если родителем является нода под номером -1 (т.е. пользовательский узел), то ребенок создается на сервере, если же нет, то отправляется команда на родительский узел;
2. Remove. Указывается номер узла, который мы хотим удалить. Если он недоступен, то информируем пользователя об этом и выходим. Если же доступен, то отправляем ему команду удаления;
3. Exec. Команда поиска подстроки в строке на вычислительном узле. Проверяем доступность узла. Формируем команду. Отправляем. Ожидаем ответ, выводим его.
4. Ping. Команда проверки доступности узла. Отправляем команду ping на указаный узел. Ожидаем ответ. Если ответ пришел, то узел доступен, если ответа нет - нет.

Для сериализации команд, используются функции fill_* и get_*. Любая команда состоит из заголовка, который одинаков для всех команд и хранит в себе информацию о том куда направлено данное сообщение и какого типа данная команда. Для некоторых команд нужно хранить дополнительную информацию, в таком случае считывается тело сообщения.

Основные файлы программы

client.cpp:

```
// Вычислительный узел
#include <iostream>
#include <string>
#include <zmq.hpp>
#include <unistd.h>
#include <csignal>

#include "message.hpp"
#include "zmq_tools.hpp"

static zmq::context_t context;
static zmq::socket_t pub_sock_to_children(context, ZMQ_PUB);
static zmq::socket_t pub_sock_to_parent(context, ZMQ_PUB);
static zmq::socket_t sub_sock(context, ZMQ_SUB);

static std::string pub_socket_to_children_name;
static std::string pub_socket_to_server_name;

static std::string parent_socket_name;
static std::vector<std::string> children_sockets_name;

static int node_id;
static bool run;

// node_id, parent_socket_name
void client_init(int argc, char* argv[]){
    std::stringstream sstream;
    sstream << argv[1];
    sstream >> node_id;

    parent_socket_name = argv[2];

    pub_socket_to_children_name =
create_name_of_socket_to_children(node_id);
    pub_socket_to_server_name =
create_name_of_socket_to_parent(node_id);

    pub_sock_to_children.bind(pub_socket_to_children_name);
    pub_sock_to_parent.bind(pub_socket_to_server_name);

    sub_sock.connect(parent_socket_name);
    sub_sock.setsockopt(ZMQ_SUBSCRIBE, 0, 0);

    run = true;
}

void handle_create(zmq::message_t& data) {
    create_body body = get_message_create(data);
    int child_id = body.child_id;
    int fork_pid = fork();
    // Ошибка
    if(fork_pid == -1) {
```

```
            zmq::message_t ans = fill_message_create_answer(-1,
strerror(errno));
            pub_sock_to_parent.send(ans);
            return;
        }
        // Процесс ребенка
        if(fork_pid == 0) {
            std::stringstream sstream;
            sstream << child_id;
            execl("client", "client", sstream.str().c_str(),
pub_socket_to_children_name.c_str(), NULL);
        }

        std::string parent_pub_socket_name =
create_name_of_socket_to_parent(child_id);
        children_sockets_name.push_back(parent_pub_socket_name);
        sub_sock.connect(parent_pub_socket_name);

        // Отсылаем ответ, что все вроде ОК
        zmq::message_t ans = fill_message_create_answer(fork_pid, "");
        pub_sock_to_parent.send(ans);
    }

    void handle_remove(zmq::message_t& data) {
        header_t* header = get_message_header(data);
        // Отсылаем детям
        header->to_id_node = BROADCAST_ID;
        pub_sock_to_children.send(data);
        // Удаляем текущий узел
        run = false;
    }

    void handle_exec(zmq::message_t& data) {
        exec_body body = get_message_exec(data);
        std::vector<int> entries;

        std::string::size_type pos = 0;
        while(std::string::npos != (pos = body.text.find(body.pattern,
pos))){
            entries.push_back(pos);
            ++pos;
        }

        zmq::message_t ans = fill_message_exec_answer(entries);
        pub_sock_to_parent.send(ans);
    }

    void handle_ping(zmq::message_t& data) {
        zmq::message_t ans = fill_message_ping_answer(node_id);
        pub_sock_to_parent.send(ans);
    }

    void handle_task(zmq::message_t& data){
        header_t* header = get_message_header(data);
        if(header->type == MSG_CREATE)
            handle_create(data);
```

```
    else if(header->type == MSG_REMOVE)
        handle_remove(data);
    else if(header->type == MSG_EXEC)
        handle_exec(data);
    else if(header->type == MSG_PING)
        handle_ping(data);
}

void client_run(){
    while(run){
        try {
            zmq::message_t data;
            sub_sock.recv(data);
            header_t* header = get_message_header(data);
            // Если сообщение адресовано нам
            if(header->to_id_node == node_id || header->to_id_node
== BROADCAST_ID)
                handle_task(data);
            // Иначе пропускаем его дальше
            if(header->to_id_node != node_id){
                if(header->dir == DIR_TO_SERVER)
                    pub_sock_to_parent.send(data);
                else
                    pub_sock_to_children.send(data);
            }
        }
        catch(zmq::error_t) {
            std::cout << zmq_strerror(errno) << std::endl;
        }

    }
}

void client_deinit(){
    // Нужно удалить детей
    zmq::message_t msg = fill_message_remove(BROADCAST_ID);
    pub_sock_to_children.send(msg);

    // Закрываемся сами
    pub_sock_to_children.unbind(pub_socket_to_children_name);
    pub_sock_to_children.close();

    pub_sock_to_parent.unbind(pub_socket_to_server_name);
    pub_sock_to_parent.close();

    for(auto& name : children_sockets_name)
        sub_sock.disconnect(name);
    sub_sock.close();
}

void client_term(int){
    client_deinit();
    exit(0);
}

int main(int argc, char* argv[]) {
```

```
    std::signal(SIGINT, client_term);
    std::signal(SIGTERM, client_term);

    client_init(argc, argv);
    client_run();
    client_deinit();
    return 0;
}
```

server.cpp:

```
// Пользовательский узел
#include <iostream>
#include <string>
#include <zmq.hpp>
#include <unistd.h>
#include <csignal>

#include "message.hpp"
#include "zmq_tools.hpp"

static zmq::context_t context;

static zmq::socket_t pub_sock(context, ZMQ_PUB);
static zmq::socket_t sub_sock(context, ZMQ_SUB);

static std::string socket_name;
static std::vector<std::string> sockets_of_children;

void server_init(){
    try {
        socket_name = create_name_of_socket(0);
        pub_sock.bind(socket_name);

        sub_sock.setsockopt(ZMQ_SUBSCRIBE, 0, 0);
        int timeout = 1000;
        sub_sock.setsockopt(ZMQ_RCVTIMEO, timeout);
    }
    catch (zmq::error_t) {
        std::cout << "Error:ZMQ: " << zmq_strerror(errno) <<
std::endl;
        exit(-1);
    }

}

void server_deinit(){
    try {
        zmq::message_t msg = fill_message_remove(BROADCAST_ID);
        pub_sock.send(msg);

        pub_sock.unbind(socket_name);
        pub_sock.close();

        for(auto& name : sockets_of_children)
```

```
            sub_sock.disconnect(name);
        sub_sock.close();

        context.~context_t();
    }
    catch (zmq::error_t){
        std::cout << "Error:ZMQ: " << zmq_strerror(errno) <<
std::endl;
        exit(-2);
    }
}

bool receive_msg(zmq::message_t& msg){
    zmq::recv_result_t res;
    res = sub_sock.recv(msg);

    return res.has_value();
}

bool is_exists(int id) {
    zmq::message_t data = fill_message_ping(id);
    pub_sock.send(data);

    zmq::message_t receive_data;
    header_t* header;

    if(!receive_msg(receive_data))
        return false;

    ping_body_answer ans = get_message_ping_answer(receive_data);
    if(ans.src_id ≠ id)
        return false;

    return true;
}

void handle_create(){
    int new_node_pid = -1;
    int parent_id, child_id;
    std::cin >> child_id >> parent_id;

    if(parent_id ≠ -1 && !is_exists(parent_id)){
        std::cout << "Error: Parent not found" << std::endl;
        return;
    }
    if(is_exists(child_id)){
        std::cout << "Error: Already exists" << std::endl;
        return;
    }

    if(parent_id == -1){
        // Родитель - пользовательский узел
        int fork_pid = fork();
        // Ошибка fork
        if(fork_pid == -1) {
```

```
            std::cout << "Error:fork: " << strerror(errno) <<
std::endl;
            return;
        }
        // Процесс ребенка
        if(fork_pid == 0) {
            std::stringstream sstream;
            sstream << child_id;
            execl("client", "client", sstream.str().c_str(),
socket_name.c_str(), NULL);
        }
        // Подписываемся на дочерний узел
        std::string parent_pub_socket_name =
create_name_of_socket_to_parent(child_id);
        sockets_of_children.push_back(parent_pub_socket_name);
        sub_sock.connect(parent_pub_socket_name);

        new_node_pid = fork_pid;
    }
    else {
        // Отослать комманду
        zmq::message_t data = fill_message_create(parent_id,
child_id);
        pub_sock.send(data);

        zmq::message_t receive_data;
        receive_msg(receive_data);
        create_body_answer ans =
get_message_create_answer(receive_data);

        if(ans.pid == -1){
            std::cout << "Error:remote_create: " << ans.error <<
std::endl;
            return;
        }

        new_node_pid = ans.pid;
    }

    std::cout << "OK: " << new_node_pid << std::endl;
}

void handle_remove() {
    int remove_id;
    std::cin >> remove_id;

    if(!is_exists(remove_id)) {
        std::cout << "Error: Not found" << std::endl;
        return;
    }

    zmq::message_t data = fill_message_remove(remove_id);
    pub_sock.send(data);

    std::cout << "OK" << std::endl;
}
```

```
void handle_exec() {
    std::string text, pattern;
    int dest_id;

    std::cin.get();
    std::getline(std::cin, text);
    std::getline(std::cin, pattern);
    std::cin >> dest_id;

    if(!is_exists(dest_id)) {
        std::cout << "Error:" << dest_id << ": Not found" <<
std::endl;
        return;
    }

    zmq::message_t data = fill_message_exec(dest_id, text,
pattern);
    pub_sock.send(data);

    zmq::message_t receive_data;
    header_t* header;
    receive_msg(receive_data);
    exec_body_answer ans = get_message_exec_answer(receive_data);

    std::cout << "OK:" << dest_id << ": [";
    if(ans.entries.size() > 0)
        std::cout << ans.entries[0];
    for(int i = 1; i < ans.entries.size(); ++i) {
        std::cout << ", " << ans.entries[i];
    }
    std::cout << "]" << std::endl;
}

void handle_ping() {
    int dest_id;
    std::cin >> dest_id;
    if(is_exists(dest_id))
        std::cout << "OK: 1" << std::endl;
    else
        std::cout << "Error: Not found" << std::endl;
}

void server_run(){
    std::string input_cmd;
    while(std::cin >> input_cmd) {
        try{
            if(input_cmd == "create")
            handle_create();
            else if(input_cmd == "remove")
                handle_remove();
            else if(input_cmd == "exec")
                handle_exec();
            else if(input_cmd == "ping")
                handle_ping();
            else {
```

```
                std::cout << "Error: данной команды нет!" <<
std::endl;
            }
        }
        catch(error_t) {
            std::cout << "Error:ZMQ: " << zmq_strerror(errno) <<
std::endl;
        }
    }
}

void server_term(int) {
    server_deinit();
    exit(0);
}

int main() {
    std::signal(SIGINT, server_term);
    std::signal(SIGTERM, server_term);

    server_init();
    server_run();
    server_deinit();
}
```

message.cpp:

```
#include "message.hpp"
#include <cstring>

zmq::message_t fill_message_exec(int dest_id, std::string& text,
std::string& pattern){
    header_t header {
        DIR_TO_CLIENT,
        MSG_EXEC,
        dest_id,
    };

    int size_of_message = sizeof(header_t) + text.size() +
pattern.size() + 2 * sizeof(int);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();

    // Копируем header
    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    // Копируем размер текста
    int text_size = text.size();
    std::memcpy(dest_ptr, &text_size, sizeof(text_size));
    dest_ptr += sizeof(text_size);

    // копируем сам текст
    std::copy(text.begin(), text.end(), dest_ptr);
    dest_ptr += text.size();
```

```
    // Копируем размер паттерна
    int pattern_size = pattern.size();
    std::memcpy(dest_ptr, &pattern_size, sizeof(pattern_size));
    dest_ptr += sizeof(pattern_size);

    // копируем сам паттерн
    std::copy(pattern.begin(), pattern.end(), dest_ptr);
    dest_ptr += pattern.size();

    return data;
}

zmq::message_t fill_message_exec_answer(std::vector<int>& enrties)
{
    header_t header {
        DIR_TO_SERVER,
        MSG_EXEC_ANSWER,
        -1,
    };

    int size_of_message = sizeof(header_t) + sizeof(int) *
enrties.size() + sizeof(int);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();

    // Копируем header
    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    // Копируем размер массива обнаруженных вхождений
    int entries_size = enrties.size();
    std::memcpy(dest_ptr, &entries_size, sizeof(entries_size));
    dest_ptr += sizeof(entries_size);

    // копируем сам массив
    std::copy(enrties.begin(), enrties.end(), (int*)dest_ptr);
    dest_ptr += sizeof(int) * enrties.size();

    return data;
}

zmq::message_t fill_message_create(int parent_id, int child_id){
    header_t header {
        DIR_TO_CLIENT,
        MSG_CREATE,
        parent_id
    };

    create_body body {
        child_id
    };

    int size_of_message = sizeof(header_t) + sizeof(body);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();
```

```
    // Копируем header
    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    // Копируем body
    std::memcpy(dest_ptr, &body, sizeof(body));
    dest_ptr += sizeof(body);

    return data;
}

zmq::message_t fill_message_create_answer(int pid, std::string
error){
    header_t header {
        DIR_TO_SERVER,
        MSG_CREATE_ANSWER,
        -1
    };

    int size_of_message = sizeof(header_t) + sizeof(pid) +
sizeof(int) + error.size() * sizeof(error);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();

    // Копируем header
    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    // Копируем pid
    std::memcpy(dest_ptr, &pid, sizeof(pid));
    dest_ptr += sizeof(pid);

    // Копируем error.size
    int error_size = error.size();
    std::memcpy(dest_ptr, &error_size, sizeof(error_size));
    dest_ptr += sizeof(error_size);

    // Копируем error
    std::copy(error.begin(), error.end(), dest_ptr);
    dest_ptr += error.size();

    return data;
}

zmq::message_t fill_message_remove(int dest_id){
    header_t header {
        DIR_TO_CLIENT,
        MSG_REMOVE,
        dest_id
    };

    int size_of_message = sizeof(header_t);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();
```

```
    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    return data;
}

zmq::message_t fill_message_ping(int dest_id){
    header_t header {
        DIR_TO_CLIENT,
        MSG_PING,
        dest_id
    };

    int size_of_message = sizeof(header_t);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();

    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    return data;
}

zmq::message_t fill_message_ping_answer(int src_id){
    header_t header {
        DIR_TO_SERVER,
        MSG_PING_ANSWER,
        -1
    };

    ping_body_answer body {
        src_id
    };

    int size_of_message = sizeof(header_t) + sizeof(body);
    zmq::message_t data(size_of_message);
    char* dest_ptr = (char*)data.data();

    std::memcpy(dest_ptr, &header, sizeof(header));
    dest_ptr += sizeof(header);

    std::memcpy(dest_ptr, &body, sizeof(body));
    dest_ptr += sizeof(body);

    return data;
}

header_t* get_message_header(zmq::message_t& data){
    return (header_t*)data.data();
}

create_body get_message_create(zmq::message_t& data){
    create_body body;
    std::memcpy(&body, data.data() + sizeof(header_t),
sizeof(body));
    return body;
```

```
}

create_body_answer get_message_create_answer(zmq::message_t& data)
{
    char* src_ptr = (char*)data.data() + sizeof(header_t);

    // Восстанавливаем pid
    int pid;
    std::memcpy(&pid, src_ptr, sizeof(int));
    src_ptr += sizeof(pid);

    // Восстанавливаем error size
    int error_size;
    std::memcpy(&error_size, src_ptr, sizeof(int));
    src_ptr += sizeof(error_size);

    // Восстанавливаем error
    std::string error(src_ptr, error_size);
    src_ptr += error_size;

    return {pid, error};
}

exec_body get_message_exec(zmq::message_t& data){
    char* src_ptr = (char*)data.data() + sizeof(header_t);

    int text_size;
    std::memcpy(&text_size, src_ptr, sizeof(int));
    src_ptr += sizeof(int);

    std::string text(src_ptr, text_size);
    src_ptr += text_size;

    int pattern_size;
    std::memcpy(&pattern_size, src_ptr, sizeof(int));
    src_ptr += sizeof(int);

    std::string pattern(src_ptr, pattern_size);
    src_ptr += pattern_size;

    return {text, pattern};
}

exec_body_answer get_message_exec_answer(zmq::message_t& data){
    char* src_ptr = (char*)data.data() + sizeof(header_t);

    int entries_size;
    std::memcpy(&entries_size, src_ptr, sizeof(int));
    src_ptr += sizeof(int);

    std::vector<int> entries;
    int* src_int_ptr = (int*)src_ptr;
    std::copy(src_int_ptr, src_int_ptr + entries_size,
std::back_inserter(entries));

    return {entries};
```

```
}

ping_body_answer get_message_ping_answer(zmq::message_t& data){
    ping_body_answer body;
    std::memcpy(&body, data.data() + sizeof(header_t),
sizeof(body));
    return body;
}
```

zmq_tools.cpp:

```cpp
#include "zmq_tools.hpp"
const int BROADCAST_ID = -2;

std::string create_name_of_socket(int node_id){
    std::stringstream sstream;
    sstream << "ipc://sock_" << node_id;
    return sstream.str();
}

std::string create_name_of_socket_to_children(int node_id){
    std::stringstream sstream;
    sstream << create_name_of_socket(node_id) << "_to_children";
    return sstream.str();
}

std::string create_name_of_socket_to_parent(int node_id){
    std::stringstream sstream;
    sstream << create_name_of_socket(node_id) << "_to_parent";
    return sstream.str();
}
```

Пример работы

```
reterer@serv:~/OS/os_exercise_06/src$ ./server
create 1 -1
OK: 51690
create 2 1
OK: 51741
ping 2
OK: 1
exec
hello world hello world helloasdfasdf
hello
2
OK:2: [0, 12, 24]
remove 1
OK
ping 2
Error: Not found
```

Вывод

Данная ЛР была одной из самых сложных. Так как нужно было самостоятельно изучить стороннюю библиотеку и использовать ее.

Я научился использовать данную библиотеку для передачи сообщений. Кроме того, поработал с сериализацией и десериализацией сообщений.

ZMQ позволяет организовать систему очереди сообщений. Предлагая нам довольно простой способ отправки и получения сообщений (или задач). С ZMQ не нужно заниматься низкоуровневой реализацией. Эти задачи ZMQ уже решил. Нам не нужно думать, как отправлять сообщение неопределенного размера или, например, кому их отправлять. Мы можем использовать разные протоколы общения почти без изменения кода. Кроме того, данная библиотека ориентирована на скорость. И в отличии от других *MQ приложений, позволяет конструировать свои, идеально подходящие под нужды, решения.