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

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

# Управление серверами сообщений

Студент: Воробьева К.Н.
Группа: М8О –201Б-21
Вариант: 33
Преподаватель: Миронов Е.С.
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2020

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Тестирование
6. Демонстрация работы программы
7. Вывод

## Постановка задачи

Реализовать распределенную систему по обработке запросов. В данной 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 сервера сообщений zmq. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом.

**Вариант 33**

1. Топология — бинарное дерево.
2. Тип вычислительной команды — локальный таймер.
3. Тип проверки узлов на доступность — heartbeat time.

## Общие сведения о программе

Программа состоит из двух основных файлов и библиотеки, реализующей взаимодействия с узлами. Помимо этого используется библиотека zmq, которая реализует очередь сообщений.
1) main.cpp - программа, которая является управляющим узлом
2) child_main.cpp - программа, которая подгружает необходимые данные из библиотеки во время исполнения программы.
3) server.cpp, server.h - реализация библиотеки для взаимодействия между узлами.
4) zmq.hpp - статическая библиотека для межпроцессорного общения.
5) struct_server.cpp – файл топологии, заданной структуры.

Очередь сообщений - компонент, используемый для межпроцессного или межпотокового взаимодействия внутри одного процесса. Для обмена сообщениями используется очередь. Очереди сообщений предоставляют асинхронный протокол передачи данных, означая, что отправитель и получатель сообщения не обязаны взаимодействовать с очередью сообщений одновременно. Размещённые в очереди сообщения хранятся до тех пор, пока получатель не получит их.

ZMQ - библиотека асинхронных сообщений, предназначенная для использования в распределенных или параллельных приложениях. Он обеспечивает очередь сообщений, но в отличие от промежуточного программного обеспечения, ориентированного на сообщения, система ZMQ может работать без выделенного посредника сообщений.

Сокеты - название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Основные функции и вызовы:

1) context_t::context_t(int io_threads) - Функция инициализирует контекст ZMQ. Аргумент io_threads указывает размер пула потоков ZMQ для обработки операций ввода-вывода.

2) socket_t::socket_t(context_t &context, int type) - Функция должна создать сокет ZMQ в указанном контексте и вернуть непрозрачный дескриптор вновь созданному сокету. Аргумент type указывает тип сокета, который определяет семантику связи через сокет.
Вновь созданный сокет изначально не связан и не связан ни с какими конечными точками. Чтобы установить поток сообщений, сокет должен быть сначала подключен по крайней мере к одной конечной точке с помощью zmq_connect (3), или по крайней мере одна конечная точка должна быть создана для приема входящих соединений с помощью zmq_bind (3).
Сокет типа ZMQ_REQ используется клиентом для отправки запросов и получения ответов от службы.
ZMQ_REP используется службой для получения запросов и отправки ответов клиенту.

3) int execv(const char *path, char *const argv[]) - функция execv() заменяет текущий образ процесса новым. execv() предоставляет новой программе список аргументов в виде массива указателей на строки, заканчивающиеся null. Первый аргумент, по соглашению, должен указать на имя, ассоции-рованное с файлом, который необходимо запустить. Массив указателей должен заканчиваться указателем null.

4) void socket_t::bind(const char *endpoint) - Функция должна создать конечную точку для приема соединений и связать ее с сокетом, на который ссылается аргумент сокета.
Аргумент endpoint - это строка, состоящая из двух частей: transport: // address. Транспортная часть определяет базовый транспортный протокол для использования. Значение адресной части зависит от выбранного ос-новного транспортного протокола. Один из них TCP - это вездесущий, надежный, одноадресный транспорт.

5) bool socket_t::recv(message_t *msg, int flags = 0) - Функция должна по-лучить сообщение от сокета, на который ссылается аргумент сокета, и со-хранить его

в сообщении, на которое ссылается аргумент msg. Любой кон-тент, ранее сохраненный в msg, должен быть надлежащим образом осво-божден. Если в указанном сокете нет доступных сообщений, функция zmq_recv () блокируется до тех пор, пока запрос не будет удовлетворен. Аргумент flags можно опустить, либо указать ZMQ_NOBLOCK - Указывает, что операция должна выполняться в неблокирующем режиме. Если в указанном сокете нет доступных сообщений, функция zmq_recv () должна завершиться с ошибкой, когда для errno установлено значение EAGAIN.

6) zmq::message_t msg(size_t size) - создается экземпляр класса msg разме-ра size, который имеет доступ к функциям для создания, уничтожения и управления сообщениями ZMQ.
Функция должна распределять любые ресурсы, необходимые для хранения длинного байта сообщения, и инициализировать объект сообщения, на ко-торый ссылается msg, для представления вновь выделенного сообщения.
Реализация должна выбрать, хранить ли содержимое сообщения в стеке (маленькие сообщения) или в куче (большие сообщения). По соображениям производительности не должен очищать данные сообщения.

7) bool socket_t::send(message_t &msg, int flags = 0) - Функция помещает в очередь сообщение, на которое ссылается аргумент msg, для отправки в со-кет, на который ссылается аргумент socket. Аргумент flags представляет со-бой комбинацию флагов, определенных ниже:
ZMQ_NOBLOCK
Указывает, что операция должна выполняться в неблокирующем режиме. Если сообщение не может быть поставлено в очередь в сокете, функция socket_t::send () должна завершиться с ошибкой, когда errno установлено в EAGAIN.
ZMQ_SNDMORE
Указывает, что отправляемое сообщение является сообщением, состоящим из нескольких частей, и что последующие части сообщения должны следо-вать.

8) void *memcpy(void *dest, const void *source, size_t count) - Функция memcpy() копирует count символов из массива, на который указывает source, в массив, на который указывает dest. Если массивы перекрываются, поведение memcpy() не определено.

## Общий метод и алгоритм решения

1. Управляющий узел принимает команды, обрабатывает их и пересылает дочерним узлам или выводит сообщение об ошибке.
2. Дочерние узлы проверяют, может ли быть команда выполнена в данном узле, если нет, то команда пересылается в один из дочерних узлов, из которого возвращается некоторое сообщение (об успехе или об ошибке), которое потом пересылается обратно по дереву.
3. Для корректной проверки на доступность узлов, используется дерево, эмулирующее поведение узлов в данной топологии (например, при удалении узла, удаляются все его потомки).
4. Если узел недоступен, то по истечении таймаута будет сгенерировано сообщение о недоступности узла и оно будет передано вверх по дереву, к управляющему узлу. При удалении узла, все его потомки уничтожаются.

## Листинг программы

**main.cpp**

```
#include<zmq.hpp>
#include <signal.h>

#include<iostream>
#include<set>
#include<string>
#include<vector>

#include"server.h"
#include"struct_server.h"



int main() {
    BinTree tree;

    std::string cmd;

    int child_pid = 0;
    int child_id = 0;
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    int linger = 0;
    main_socket.setsockopt(ZMQ_SNDTIMEO, 2000);
    main_socket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
```

```
int port = bind_socket(main_socket);

int input_id;
std::string result;
std::string msg;

while (true) {
    std::cin >> cmd;
    if (cmd == "create") {
        std::cin >> input_id;
        if (child_pid == 0) {
            child_pid = fork();
            if (child_pid == -1) {
                std::cout << "Unable to create first worker node" << std::endl;
                child_pid = 0;
                exit(1);
            } else if (child_pid == 0) {
                create_node(input_id, port);
            } else {
                child_id = input_id;
                msg = "pid";
                send_msg(main_socket, msg);
                result = recieve_msg(main_socket);
            }
        } else {
            std::ostringstream msg_stream;
            msg_stream << "create " << input_id;
            send_msg(main_socket, msg_stream.str());
            result = recieve_msg(main_socket);
        }

        if (result.substr(0,2) == "OK") {
            tree.insert(input_id);
        }
        std::cout << result << std::endl;

    } else if (cmd == "remove") {
        if (child_pid == 0) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }

        std::cin >> input_id;
        if (input_id == child_id) {
            kill(child_pid, SIGTERM);
```

```
            kill(child_pid, SIGKILL);
            child_id = 0;
            child_pid = 0;
            std::cout << "OK" << std::endl;
            tree.erase(input_id);
            continue;
        }
        msg = "remove " + std::to_string(input_id);
        send_msg(main_socket, msg);
        result = recieve_msg(main_socket);
        if (result.substr(0, std::min<int>(result.size(), 2)) == "OK") {
            tree.erase(input_id);
        }
        std::cout << result << std::endl;

    } else if (cmd == "exec") {
        std::cin >> input_id >> cmd;
        std::vector<int> path = tree.get_path_to(input_id);
        if(path.empty()) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        path.erase(path.begin());
        msg = "exec " + cmd + " " + std::to_string(path.size());
        for (int i = 0; i < input_id; ++i) {
            msg += " " + std::to_string(path[i]);
        }
        send_msg(main_socket, msg);
        result = recieve_msg(main_socket);
        std::cout << result << std::endl;

    } else if (cmd == "pingall") {
        msg = "pingall";
        send_msg(main_socket, msg);
        result = recieve_msg(main_socket);
        std::istringstream is;
        if (result.substr(0,std::min<int>(result.size(), 5)) == "Error") {
            is = std::istringstream("");
        } else {
            is = std::istringstream(result);
        }

        std::set<int> recieved_tree;
        while (is >> input_id) {
            recieved_tree.insert(input_id);
```

```
            }
            std::vector<int> from_tree = tree.get_all_nodes();
            auto part_it = std::partition(from_tree.begin(), from_tree.end(),
[&recieved_tree] (int a) {
                return recieved_tree.count(a) == 0;
            });
            if (part_it == from_tree.begin()) {
                std::cout << "OK: -1" << std::endl;
            } else {
                std::cout << "OK:";
                for (auto it = from_tree.begin(); it != part_it; ++it) {
                    std::cout << " " << *it;
                }
                std::cout << std::endl;
            }

        } else if (cmd == "exit") {
            break;
        }
    }
    return 0;
}
```

**node.cpp**

```
#include<zmq.hpp>
#include<csignal>

#include<iostream>
#include<string>
#include<vector>
#include<unistd.h>

#include"server.h"
#include"struct_server.h"



int main(int argc, char** argv) {
    if(argc != 3) {
        std::cerr << "Not enough parameters" << std::endl;
        exit(-1);
    }
    int id = std::stoi(argv[1]);
```

```
int port = std::stoi(argv[2]);
zmq::context_t context(3);
zmq::socket_t parent_socket(context, ZMQ_REP);

parent_socket.connect(get_host_port(port));

int left_pid = 0;
int right_pid = 0;
int left_id = 0;
int right_id = 0;

zmq::socket_t left_socket(context, ZMQ_REQ);
zmq::socket_t right_socket(context, ZMQ_REQ);

left_socket.set(zmq::sockopt::sndtimeo, 2000);
right_socket.set(zmq::sockopt::sndtimeo, 2000);

int left_port = bind_socket(left_socket);
int right_port = bind_socket(right_socket);

std::string request;
std::string msg;
std::string cmd;
std::string subcmd;

int value;

std::ostringstream res;
std::string left_res;
std::string right_res;

int input_id;

auto start_clock = std::chrono::high_resolution_clock::now();
auto stop_clock = std::chrono::high_resolution_clock::now();
auto time_clock = 0;
bool flag_clock = false;


while (true) {
    request = recieve_msg(parent_socket);

    std::istringstream cmd_stream(request);
    cmd_stream >> cmd;
```

```
if (cmd == "id") {
    msg = "OK: " + std::to_string(id);
    send_msg(parent_socket, msg);
} else if (cmd == "pid") {
    msg = "OK: " + std::to_string(getpid());
    send_msg(parent_socket, msg);
} else if (cmd == "create") {
    cmd_stream >> input_id;
    if (input_id == id) {
        msg = "Error: Already exists";
        send_msg(parent_socket, msg);
    } else if (input_id < id) {
        if (left_pid == 0) {
            left_pid = fork();
            if (left_pid == -1) {
                msg = "Error: Cannot fork";
                send_msg(parent_socket, msg);
                left_pid = 0;
            } else if (left_pid == 0) {
                create_node(input_id, left_port);
            } else {
                left_id = input_id;
                msg = "pid";
                send_msg(left_socket, msg);
                send_msg(parent_socket, recieve_msg(left_socket));
            }
        } else {
            send_msg(left_socket, request);
            send_msg(parent_socket, recieve_msg(left_socket));
        }
    } else {
        if (right_pid == 0) {
            right_pid = fork();
            if (right_pid == -1) {
                msg = "Error: Cannot fork";
                send_msg(parent_socket, msg);
                right_pid = 0;
            } else if (right_pid == 0) {
                create_node(input_id, right_port);
            } else {
                right_id = input_id;
                msg = "pid";
                send_msg(right_socket, msg);
                send_msg(parent_socket, recieve_msg(right_socket));
            }
```

```
        } else {
            send_msg(right_socket, request);
            send_msg(parent_socket, recieve_msg(right_socket));
        }
    }
} else if (cmd == "remove") {
    cmd_stream >> input_id;
    if (input_id < id) {
        if (left_id == 0) {
            msg = "Error: Not found";
            send_msg(parent_socket, msg);
        } else if (left_id == input_id) {
            msg = "kill_child";
            send_msg(left_socket, msg);
            msg = recieve_msg(left_socket);
            kill(left_pid, SIGTERM);
            kill(left_pid, SIGKILL);
            left_id = 0;
            left_pid = 0;
            send_msg(parent_socket, msg);
        } else {
            send_msg(left_socket, request);
            send_msg(parent_socket, recieve_msg(left_socket));
        }
    } else {
        if (right_id == 0) {
            msg = "Error: Not found";
            send_msg(parent_socket, msg);
        } else if (right_id == input_id) {
            msg = "kill_child";
            send_msg(right_socket, msg);
            msg = recieve_msg(right_socket);
            kill(right_pid, SIGTERM);
            kill(right_pid, SIGKILL);
            right_id = 0;
            right_pid = 0;
            send_msg(parent_socket, msg);
        } else {
            send_msg(right_socket, request);
            send_msg(parent_socket, recieve_msg(right_socket));
        }
    }

} else if (cmd == "exec") {
    cmd_stream >> subcmd >> value;
```

```
            std::vector<int> path(value);
            for(int i = 0; i < value; ++i){
                cmd_stream >> path[i];
            }
            if(path.empty()) {
                msg = "OK: " + std::to_string(id) + " " + subcmd;
                if(subcmd == "start") {
                    start_clock = std::chrono::high_resolution_clock::now();
                    flag_clock = true;
                }
                else if(subcmd == "stop") {
                    if(flag_clock) {
                        stop_clock = std::chrono::high_resolution_clock::now();
                        time_clock                                                    +=
std::chrono::duration_cast<std::chrono::milliseconds>(stop_clock                      -
start_clock).count();
                        flag_clock = false;
                    }
                } else if(subcmd == "time") {
                    msg += ": " + std::to_string(time_clock);
                }
                send_msg(parent_socket, msg);
            } else {
                input_id = path.front();
                path.erase(path.begin());
                res << "exec " << subcmd << " " << path.size();
                for(int i: path){
                    res << " " << i;
                }
                if (input_id == id) {
                    msg = "Node is available";
                    send_msg(parent_socket, msg);
                } else if (input_id < id) {
                    send_msg(left_socket, res.str());
                    send_msg(parent_socket, recieve_msg(left_socket));
                } else {
                    send_msg(right_socket, res.str());
                    send_msg(parent_socket, recieve_msg(right_socket));
                }
            }

        } else if (cmd == "pingall") {
            msg = "pingall";
            if (left_pid != 0) {
                send_msg(left_socket, msg);
```

```
            left_res = recieve_msg(left_socket);
        }
        if (right_pid != 0) {
            send_msg(right_socket, msg);
            right_res = recieve_msg(right_socket);
        }
        if (!left_res.empty() && left_res.substr(0, 5) != "Error") {
            res << left_res;
        }
        if (!right_res.empty() && right_res.substr(0, 5) != "Error") {
            res << right_res;
        }
        send_msg(parent_socket, res.str());

    } else if (cmd == "heartbeat") {
        msg = "OK: " + std::to_string(id);
        send_msg(parent_socket, msg);

    } else if (cmd == "kill_child") {
        if (left_pid == 0 && right_pid == 0) {
            msg = "OK";
            send_msg(parent_socket, msg);
        } else {
            if (left_pid != 0) {
                msg = "kill_child";
                send_msg(left_socket, msg);
                recieve_msg(left_socket);
                kill(left_pid, SIGTERM);
                kill(left_pid, SIGKILL);
            }
            if (right_pid != 0) {
                msg = "kill_child";
                send_msg(right_socket, msg);
                recieve_msg(right_socket);
                kill(right_pid, SIGTERM);
                kill(right_pid, SIGKILL);
            }
            msg = "OK";
            send_msg(parent_socket, msg);
        }
    }
    if (port == 0) {
        break;
    }
}
```

```
    return 0;
}
```

**server.cpp**

```
#include"server.h"



bool send_msg(zmq::socket_t& socket, const std::string& msg) {
    int msg_size = msg.size();
    zmq::message_t message(msg_size);
    memcpy(message.data(), msg.c_str(), msg_size);
    try {
        socket.send(message, zmq::send_flags::none);
        return true;
    } catch(...) {
        return false;
    }
}

std::string recieve_msg(zmq::socket_t& socket) {
    zmq::message_t request;
    socket.recv(request, zmq::recv_flags::none);
    std::string recieve_msg(static_cast<char*>(request.data()), request.size());
    if (recieve_msg.empty())
        throw std::logic_error("Error: Node is not available");
    return recieve_msg;
}

std::string get_port(int& port) {
    return "tcp://127.0.0.1:" + std::to_string(port);
}

int bind_socket(zmq::socket_t& socket) {
    int port = 3000;
    while (true) {
        try {
            socket.bind(get_port(port));
            break;
        } catch(zmq::error_t &e) {
            ++port;
            std::cout << "[ERROR]: bind_socket " << e.what() << std::endl;
        }
    }
```

```
    return port;
}

void create_node(int& id, int& port) {
    char* arg_id = strdup((std::to_string(id)).c_str());
    char* arg_port = strdup((std::to_string(port)).c_str());
    char* args[] = {strdup("./node"), arg_id, arg_port, NULL};
    execv("./node", args);
}
```

**server.h**

```
#pragma once
#include<zmq.hpp>
#include<unistd.h>
#include<iostream>
#include<string>


bool send_msg(zmq::socket_t& socket, const std::string& msg);
std::string recieve_msg(zmq::socket_t& socket);

std::string get_port(int& port);

int bind_socket(zmq::socket_t& socket);
void create_node(int& id, int& port);
```

**srtuct_server.h**

```
#pragma once
#include<iostream>
#include<vector>


class BinTree {
    private:
        struct Node {
            Node(int id): id(id) {}
            int id;
            Node* left = nullptr;
            Node* right = nullptr;
        };

        Node* head = nullptr;
```

```
public:
    BinTree() = default;
    ~BinTree(){
        this->delete_recursive(this->head);
    }

    std::vector<int> get_all_nodes(Node* node=nullptr) {
        std::vector<int> result;
        if (node == nullptr)
            node = this->head;
        this->all_nodes(node, result);
        return result;
    }

    std::vector<int> get_path_to(int& id, Node* node=nullptr) {
        std::vector<int> path;
        if (node == nullptr)
            node = this->head;
        this->find_path(node, id, path);
        return path;
    }

    bool contains(int& id) const{
        Node* tmp = this->head;
        while(tmp != nullptr){
            if(tmp->id == id)
                break;
            else if(id > tmp->id)
                tmp = tmp->right;
            else if(id < tmp->id)
                tmp = tmp->right;
        }
        return tmp != nullptr;
    }

    void insert(int& id){
        if(this->head == nullptr){
            this->head = new Node(id);
            return;
        }
        Node* tmp = this->head;
        while(tmp != nullptr){
            if(tmp->id == id)
                return;
```

```
            else if(id < tmp->id){
                if(tmp->left == nullptr){
                    tmp->left = new Node(id);
                    return;
                }
                tmp = tmp->left;
            }
            else if(id > tmp->id){
                if(tmp->right == nullptr){
                    tmp->right = new Node(id);
                    return;
                }
                tmp = tmp->right;
            }
        }
    }

    void erase(int& id){
        Node* prev = nullptr;
        Node* tmp = this->head;
        while(tmp != nullptr){
            if (id == tmp->id) {
                if (prev == nullptr) {
                    this->head = nullptr;
                } else {
                    if (prev->left == tmp) {
                        prev->left = nullptr;
                    } else {
                        prev->right = nullptr;
                    }
                }
                delete_recursive(tmp);
            } else if(id < tmp->id) {
                prev = tmp;
                tmp = tmp->left;
            } else if(id > tmp->id) {
                prev = tmp;
                tmp = tmp->right;
            }
        }
    }

private:

    void all_nodes(Node* node, std::vector<int>& vec) const{
```

```
        if(node == nullptr)
            return;
        this->all_nodes(node->left, vec);
        vec.push_back(node->id);
        this->all_nodes(node->right, vec);
    }

    void find_path(Node* node, int& id, std::vector<int>& path) {
        while(node != nullptr){
            path.push_back(node->id);
            if(node->id == id)
                break;
            else if(id > node->id)
                node = node->right;
            else if(id < node->id)
                node = node->right;
        }
    }

    void delete_recursive(Node* node){
        if(node == nullptr)
            return;
        delete_recursive(node->right);
        delete_recursive(node->left);
        delete node;
    }

    friend class TestBitTree;
};
```

**CmakeList.txt**

```
cmake_minimum_required(VERSION 3.10)
project(6_network_nods)


add_executable(main main.cpp struct_server.h)
add_executable(node node.cpp struct_server.h)
add_library(server server.cpp server.h)

target_link_libraries(server zmq)
target_link_libraries(main zmq server)
target_link_libraries(node zmq server)
```

**Тестирование**

---

create [id]

remove [id]

exec [id] [cmd - start/stop/time]

pingall

heartbeat [time (ms)]

menu

exit

---

create 6

OK: 3141

create 8

OK: 3146

create 43

OK: 3151

pingall

OK: 6 8 43

menu

---

create [id]

remove [id]

exec [id] [cmd - start/stop/time]

pingall

heartbeat [time (ms)]

menu

exit

---

heartbeat

2000

OK: 6

OK: 6

OK: 6

create 6

OK: 3167

create 8

OK: 3172

create 43

OK: 3178

remove 8

OK

pingall

OK: 6

create 7

OK: 3186

create 9

OK: 3191

create 7

OK: 3226

exec 7 start

OK: 7 start

exec 7 stop

OK: 7 stop

exec 7 time

OK: 7 time: 4836

## Выводы

В результате данной лабораторной работы я научилась работать с технологией очереди сообщений, создающие и связывающие процессы в определенные топологии, понимать клиент-серверную архитектуру, читать документацию и осваивать новые библиотеки (zmq) в кратчайшие сроки.

Получен вывод с помощью практического опыта, что разделение исполняемого кода на клиентов и сервер – удобная практика для поддержания независимых изменений в реализации обеих сторон, без ограничений текущей функциональности.