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: Пермяков Никита Александрович

Группа: М8О –208Б-19

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, ко-торая реализует очередь сообщений.

1) main.cpp - программа, которая является управляющим узлом

2) child\_main.сpp - программа, которая подгружает необходимые данные из библиотеки во время исполнения программы.

3) os06\_api.cpp и os06\_api.h - реализация библиотеки для взаимодействия между узлами.

4) zmq.hpp - статическая библиотека для межпроцессорного общения.

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 socket(context, ZMQ\_REP);

socket.connect(get\_port(port));

int left\_pid = 0;

int right\_pid = 0;

int left\_id = 0;

int right\_id = 0;

int linger = 0;

zmq::socket\_t left\_socket(context, ZMQ\_REQ);

zmq::socket\_t right\_socket(context, ZMQ\_REQ);

left\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

left\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

right\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

right\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

int left\_port = bind\_socket(left\_socket);

int right\_port = bind\_socket(right\_socket);

std::string request;

std::string msg;

std::string cmd;

int input\_id;

auto start = std::chrono::high\_resolution\_clock::now();

auto stop = std::chrono::high\_resolution\_clock::now();

auto time = 0;

bool clock\_time = false;

while (true) {

request = recieve\_msg(socket);

std::istringstream cmd\_stream(request);

cmd\_stream >> cmd;

if (cmd == "id") {

msg = "OK: " + std::to\_string(id);

send\_msg(socket, msg);

} else if (cmd == "pid") {

msg = "OK: " + std::to\_string(getpid());

send\_msg(socket, msg);

} else if (cmd == "create") {

cmd\_stream >> input\_id;

if (input\_id == id) {

msg = "Error: Already exists";

send\_msg(socket, msg);

} else if (input\_id < id) {

if (left\_pid == 0) {

left\_pid = fork();

if (left\_pid == -1) {

msg = "Error: Cannot fork";

send\_msg(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(socket, recieve\_msg(left\_socket));

}

} else {

send\_msg(left\_socket, request);

send\_msg(socket, recieve\_msg(left\_socket));

}

} else {

if (right\_pid == 0) {

right\_pid = fork();

if (right\_pid == -1) {

msg = "Error: Cannot fork";

send\_msg(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(socket, recieve\_msg(right\_socket));

}

} else {

send\_msg(right\_socket, request);

send\_msg(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(socket, msg);

} else if (left\_id == input\_id) {

msg = "kill\_children";

send\_msg(left\_socket, msg);

recieve\_msg(left\_socket);

kill(left\_pid, SIGTERM);

kill(left\_pid, SIGKILL);

left\_id = 0;

left\_pid = 0;

msg = "OK";

send\_msg(socket, msg);

} else {

send\_msg(left\_socket, request);

send\_msg(socket, recieve\_msg(left\_socket));

}

} else {

if (right\_id == 0) {

msg = "Error: Not found";

send\_msg(socket, msg);

} else if (right\_id == input\_id) {

msg = "kill\_children";

send\_msg(right\_socket, msg);

recieve\_msg(right\_socket);

kill(right\_pid,SIGTERM);

kill(right\_pid,SIGKILL);

right\_id = 0;

right\_pid = 0;

msg = "OK";

send\_msg(socket, msg);

} else {

send\_msg(right\_socket, request);

send\_msg(socket, recieve\_msg(right\_socket));

}

}

} else if (cmd == "exec") {

std::ostringstream res;

int size;

cmd\_stream >> cmd >> size;

std::vector<int> path(size);

for(int i = 0; i < size; ++i){

cmd\_stream >> path[i];

}

if(path.empty()) {

if(cmd == "start") {

start = std::chrono::high\_resolution\_clock::now();

clock\_time = true;

msg = "OK :" + std::to\_string(id);

send\_msg(socket, msg);

}

else if(cmd == "stop") {

if(clock\_time) {

stop = std::chrono::high\_resolution\_clock::now();

time += std::chrono::duration\_cast<std::chrono::milliseconds>(stop - start).count();

clock\_time = false;

msg = "OK :" + std::to\_string(id);

send\_msg(socket, msg);

}

} else if(cmd == "time") {

msg = "OK : " + std::to\_string(id) + ": " + std::to\_string(time);

send\_msg(socket, msg);

}

} else {

int next\_id = path.front();

path.erase(path.begin());

res << "exec " << cmd << " " << path.size();

for(int i: path){

res << " " << i;

}

if (next\_id == id) {

msg = "Node is available";

send\_msg(socket, msg);

} else if (next\_id < id) {

if (left\_pid == 0) {

msg = "Error:" + std::to\_string(next\_id) + ": Not found";

send\_msg(socket, msg);

} else {

send\_msg(left\_socket, request);

send\_msg(socket, recieve\_msg(left\_socket));

}

} else {

if (right\_pid == 0) {

msg = "Error:" + std::to\_string(next\_id) + ": Not found";

send\_msg(socket, msg);

} else {

send\_msg(right\_socket, request);

send\_msg(socket, recieve\_msg(right\_socket));

}

}

}

} else if (cmd == "pingall") {

std::ostringstream res;

std::string left\_res;

std::string right\_res;

if (left\_pid != 0) {

msg = "pingall";

send\_msg(left\_socket, msg);

left\_res = recieve\_msg(left\_socket);

}

if (right\_pid != 0) {

msg = "pingall";

send\_msg(right\_socket, msg);

right\_res = recieve\_msg(right\_socket);

}

if (!left\_res.empty() && left\_res.substr(std::min<int>(left\_res.size(), 5)) != "Error") {

res << left\_res;

}

if (!right\_res.empty() && right\_res.substr(std::min<int>(right\_res.size(), 5)) != "Error") {

res << right\_res;

}

send\_msg(socket, res.str());

} else if (cmd == "kill\_children") {

if (left\_pid == 0 && right\_pid == 0) {

msg = "OK";

send\_msg(socket, msg);

} else {

if (left\_pid != 0) {

msg = "kill\_children";

send\_msg(left\_socket, msg);

recieve\_msg(left\_socket);

kill(left\_pid, SIGTERM);

kill(left\_pid, SIGKILL);

}

if (right\_pid != 0) {

msg = "kill\_children";

send\_msg(right\_socket, msg);

recieve\_msg(right\_socket);

kill(right\_pid, SIGTERM);

kill(right\_pid, SIGKILL);

}

msg = "OK";

send\_msg(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**

**Выводы**

В результате данной лабораторной работы я научился 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.

Работа была выполнена на wsl, виртуальная машина которой поддерживает fork (!). Были опробованы и освоены cygwin и Windows Hyper-V (Manager). Продолжается освоение последней технологии.