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»**

**Тема работы**

**«Управление серверами сообщений. Применение отложенных вычислений. Интеграция программных систем друг с другом»**

Студент: Лютоев Илья Александрович

Группа: М8О-207Б-21

Вариант: 18

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

#### Содержание

1. Репозиторий
2. Постановка задачи
3. Общий метод и алгоритм решения
4. Исходный код
5. Демонстрация работы программы
6. Выводы

#### Репозиторий

Постановка задачи

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.

**Топология**

Все вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду:

create id -1.

**Набор команды «Локальный целочисленный словарь»**

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name

Пример:

> exec 10 MyVar

Ok:10: 'MyVar' not found

> exec 10 MyVar 5

Ok:10

> exec 12 MyVar

Ok:12: 'MyVar' not found

> exec 10 MyVar

Ok:10: 5

> exec 10 MyVar 7

Ok:10

> exec 10 MyVar

Ok:10: 7

Примечания: Можно использовать std:map.

**Тип проверки доступности узлов**

Формат команды: heartbit time

Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от узла нет сигнала в течении 4\*time миллисекунд, то должна выводится пользователю строка:

«Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

Пример:

> heartbit 2000

Ok

Пример:

> ping 10

Ok: 1 // узел 10 доступен

> ping 17

Ok: 0 // узел 17 недоступен

Общий метод и алгоритм решения

Для решения данной задачи была использована очередь сообщений ZeroMQ. Она достаточно проста в применении и легко была внедрена в проект. Далее для асинхронной обработки запросов был написан клиент, который принимает пользовательские команды и отправляет их api программы, которая в последствии уже и общается с деревом вычислительных узлов. Также была написана программа отвечающая за работу api, то есть обработки входящий команд пользователя и вывод результатов их выполнения. После чего была создана программа отвечающая за логику работы вычислительного узла, для нахождения необходимого узла в дереве, при вводе некоторых команд, она реплицирует сообщение от родителя в каждого своего ребенка, с помощью этого достигается возможность более надежной работы программы.

Исходный код

tree.h

#include <set>

#include <map>

*class* Tree

{

*public:*

Tree();

*bool* create\_node(*int* *id*, *int* *parent*);

*bool* delete\_node(*int* *id*);

std::set <*int*> get\_node\_list();

*void* print\_tree();

*private:*

std::map <*int*, std::set <*int*>> tree;

};

Zmq\_manage.h

#ifndef LAB6\_ZMQ\_MANAGE\_H

#define LAB6\_ZMQ\_MANAGE\_H

#include <string>

#include <zmq.hpp>

#include <iostream>

#include <unistd.h>

#include <set>

#include <map>

const *int* API\_PORT = 5040;

const *int* BASE\_PORT = 5050;

const *int* WAIT\_TIME = 1000;

const *int* manage\_worker\_id = -1;

std::string get\_port\_name(const *int* *port*);

*bool* send\_message(zmq::socket\_t &*socket*, const std::string &*message\_string*);

std::string recieve\_message(zmq::socket\_t &*socket*);

*void* create\_node(*int* *id*, *int* *port*);

#endif

api\_daemon.cpp

#include "../include/zmq\_manage.h"

#include "../include/tree.h"

#include <chrono>

#include <thread>

Tree T = Tree();

zmq::context\_t contex(3);

zmq::socket\_t main\_socket(contex, zmq::socket\_type::req);

*bool* flag = true;

*void* heartbeat(*int* *wait*) {

while (flag) {

std::set <*int*> s = T.get\_node\_list();

for (*auto* iter : s) {

send\_message(main\_socket, "ping " + std::to\_string(iter));

std::string result = recieve\_message(main\_socket);

if (result != "Ok: 1") {

std::cout << std::endl;

std::cout << "Heartbit: node " << iter << " is unavailable now" << std::endl;

T.delete\_node(iter);

}

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(*wait*));

}

return;

}

*int* main(*int* *argc*, *char* const \**argv*[])

{

std::thread t;

main\_socket.bind(get\_port\_name(BASE\_PORT + manage\_worker\_id + 1));

pid\_t manage\_pid = fork();

if (manage\_pid == -1) {

std::cerr << "Error to create manage worker\n";

return EXIT\_FAILURE;

} else if (manage\_pid == 0) {

create\_node(manage\_worker\_id, BASE\_PORT + manage\_worker\_id + 1);

}

std::string command;

std::string result = "";

std::string msg\_s = "";

zmq::socket\_t api\_socket(contex, zmq::socket\_type::pair);

api\_socket.connect(get\_port\_name(API\_PORT));

while (flag) {

std::cout << result << '\n';

result = "";

std::vector <std::string> args;

std::string str;

str = recieve\_message(api\_socket);

std::istringstream ist(str);

for (std::string word; ist >> word; args.push\_back(word));

if (args[0] == "print" && args.size() == 1) {

T.print\_tree();

}

if (args[0] == "ping" && args.size() == 2) {

*int* id = std::stoi(args[1]);

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

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

continue;

}

if (args[0] == "create" && args.size() == 3) {

*int* id = std::stoi(args[1]), parent = std::stoi(args[2]);

if (id < 0) {

std::cout << "Error: id will be greather or equal to 0\n";

continue;

}

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

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result != "Ok: 0") {

std::cout << "Error: Already exists\n";

continue;

}

msg\_s = "ping " + std::to\_string(parent);

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result != "Ok: 1") {

std::cout << "Error: Not found " << parent << "\n";

continue;

}

msg\_s = "create " + std::to\_string(id) + " " + std::to\_string(parent);

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result.substr(0, 2) == "Ok") {

T.create\_node(id, parent);

}

continue;

}

if (args[0] == "remove" && args.size() == 2) {

*int* id = std::stoi(args[1]);

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

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result != "Ok: 1") {

std::cout << "Error: Not found " << id << "\n";

continue;

}

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

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result.substr(0, 2) == "Ok") {

T.delete\_node(id);

}

continue;

}

if (args[0] == "exec" && args.size() == 3) {

*int* id = std::stoi(args[1]);

std::string var\_name = args[2];

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

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result != "Ok: 1") {

std::cout << "Error: Not found " << id << "\n";

continue;

}

msg\_s = "get\_value " + std::to\_string(id) + " " + var\_name;

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

continue;

}

if (args[0] == "exec" && args.size() == 4) {

*int* id = std::stoi(args[1]);

std::string var\_name = args[2];

*int* value = std::stoi(args[3]);

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

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

if (result != "Ok: 1") {

std::cout << "Error: Not found " << id << "\n";

continue;

}

msg\_s = "set\_value " + std::to\_string(id) + " " + var\_name + " " + std::to\_string(value);

send\_message(main\_socket, msg\_s);

result = recieve\_message(main\_socket);

continue;

}

if (args[0] == "heartbeat" && args.size() == 2) {

*int* time = std::stoi(args[1]);

t = std::thread(heartbeat, time);

result = "Ok";

continue;

}

if (args[0] == "exit") {

msg\_s = "kill";

send\_message(main\_socket, msg\_s);

std::cout << "Ok\n";

flag = false;

continue;

}

}

t.detach();

return EXIT\_SUCCESS;

}

client.cpp

#include "../include/zmq\_manage.h"

*int* main(*int* *argc*, *char* const \**argv*[])

{

pid\_t api\_daemon\_pid = fork();

if (api\_daemon\_pid == -1) {

std::cerr << "Error to fork api\_daemon\n";

return EXIT\_FAILURE;

} else if (api\_daemon\_pid == 0) {

execl("./api\_daemon", NULL);

}

std::cout

<< "commands:\n"

<< "\tcreate id parent\n"

<< "\tremove id\n"

<< "\texec id var\_name value\n"

<< "\texec id var\_name\n"

<< "\tping id\n"

<< "\theartbeat time\n";

zmq::context\_t contex(1);

zmq::socket\_t api\_socket(contex, zmq::socket\_type::pair);

api\_socket.bind(get\_port\_name(API\_PORT));

std::string str;

*bool* flag = true;

while (flag) {

std::getline(std::cin, str);

send\_message(api\_socket, str);

if (str == "exit") {

flag = false;

}

}

return EXIT\_SUCCESS;

}

server.cpp

#include "../include/zmq\_manage.h"

*int* main(*int* *argc*, *char* const \**argv*[])

{

*int* id = atoi(*argv*[1]);

*int* parent\_port = atoi(*argv*[2]);

//std::cout << "client start with " << id << ' ' << parent\_port << '\n';

std::map <std::string, *int*> node\_map;

zmq::context\_t context(3);

zmq::socket\_t parent\_socket(context, zmq::socket\_type::rep);

parent\_socket.connect(get\_port\_name(parent\_port));

std::vector <zmq::socket\_t> childs;

std::string result, msg\_str;

*bool* flag = true;

while (flag) {

std::string request\_string = recieve\_message(parent\_socket);

std::istringstream command\_stream(request\_string);

std::string command;

result = "", msg\_str = "";

//std::cout << "log " << id << ": " << request\_string << '\n';

command\_stream >> command;

if (command == "ping") {

*int* id\_search;

command\_stream >> id\_search;

if (id\_search == id) {

msg\_str = "Ok: 1";

send\_message(parent\_socket, msg\_str);

continue;

}

*bool* flag\_succes = false;

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

send\_message(childs[i], request\_string);

result = recieve\_message(childs[i]);

if (result == "Ok: 1") {

send\_message(parent\_socket, result);

flag\_succes = true;

break;

}

}

if (!flag\_succes) {

send\_message(parent\_socket, "Ok: 0");

}

continue;

}

if (command == "pid") {

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

send\_message(parent\_socket, msg\_str);

continue;

}

if (command == "create") {

*int* child\_id, parent\_id;

command\_stream >> child\_id >> parent\_id;

if (parent\_id == id) {

zmq::socket\_t tmp(context, zmq::socket\_type::req);

tmp.bind(get\_port\_name(BASE\_PORT + child\_id + 1));

pid\_t child\_pid = fork();

if (child\_pid == -1) {

msg\_str = "Error: Fork error";

send\_message(parent\_socket, msg\_str);

continue;

} else if (child\_pid == 0) {

create\_node(child\_id, BASE\_PORT + child\_id + 1);

} else {

send\_message(tmp, "pid");

result = recieve\_message(tmp);

send\_message(parent\_socket, result);

childs.push\_back(std::move(tmp));

continue;

}

}

*bool* flag\_succes = false;

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

send\_message(childs[i], request\_string);

result = recieve\_message(childs[i]);

if (result.substr(0, 2) == "Ok") {

send\_message(parent\_socket, result);

flag\_succes = true;

break;

}

}

if (!flag\_succes) {

send\_message(parent\_socket, "Error: Not found");

}

continue;

}

if (command == "remove") {

*int* remove\_id;

command\_stream >> remove\_id;

*bool* flag\_get\_remove\_id = false;

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

send\_message(childs[i], "get\_id");

result = recieve\_message(childs[i]);

if (result == std::to\_string(remove\_id) + " ") {

send\_message(childs[i], "kill");

sleep(1);

childs[i].unbind(get\_port\_name(BASE\_PORT + remove\_id + 1));

childs.erase(std::next(childs.begin() + i));

flag\_get\_remove\_id = true;

break;

}

}

if (flag\_get\_remove\_id) {

send\_message(parent\_socket, "Ok");

continue;

}

*bool* flag\_succes = false;

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

send\_message(childs[i], request\_string);

result = recieve\_message(childs[i]);

if (result.substr(0, 2) == "Ok") {

send\_message(parent\_socket, result);

flag\_succes = true;

break;

}

}

if (!flag\_succes) {

send\_message(parent\_socket, "Error: Not found");

}

continue;

}

if (command == "kill") {

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

send\_message(childs[i], "get\_id");

result = recieve\_message(childs[i]);

send\_message(childs[i], "kill");

childs[i].unbind(get\_port\_name(BASE\_PORT + std::stoi(result) + 1));

}

parent\_socket.disconnect(get\_port\_name(parent\_port));

flag = false;

}

if (command == "get\_value") {

*int* id\_map;

std::string var\_name;

command\_stream >> id\_map >> var\_name;

if (id\_map == id) {

if (node\_map.count(var\_name) < 1) {

msg\_str = "Ok:" + std::to\_string(id) + ": '" + var\_name + "' not found";

send\_message(parent\_socket, msg\_str);

continue;

}

msg\_str = "Ok:" + std::to\_string(id) + ": " + std::to\_string(node\_map[var\_name]);

send\_message(parent\_socket, msg\_str);

continue;

}

*bool* flag\_succes = false;

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

send\_message(childs[i], request\_string);

result = recieve\_message(childs[i]);

if (result.substr(0, 2) == "Ok") {

send\_message(parent\_socket, result);

flag\_succes = true;

break;

}

}

if (!flag\_succes) {

send\_message(parent\_socket, "Error: Not found");

}

continue;

}

if (command == "set\_value") {

*int* id\_map, value;

std::string var\_name;

command\_stream >> id\_map >> var\_name >> value;

if (id\_map == id) {

node\_map[var\_name] = value;

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

send\_message(parent\_socket, msg\_str);

continue;

}

*bool* flag\_succes = false;

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

send\_message(childs[i], request\_string);

result = recieve\_message(childs[i]);

if (result.substr(0, 2) == "Ok") {

send\_message(parent\_socket, result);

flag\_succes = true;

break;

}

}

if (!flag\_succes) {

send\_message(parent\_socket, "Error: Not found");

}

continue;

}

if (command == "get\_id") {

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

send\_message(parent\_socket, msg\_str);

continue;

}

}

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

childs[i].close();

}

parent\_socket.close();

return EXIT\_SUCCESS;

}

tree.cpp

#include "../include/tree.h"

#include <iostream>

*bool* Tree::create\_node(*int* *id*, *int* *parent*)

{

std::map <*int*, std::set <*int*>> :: iterator it;

it = this->tree.find(*id*);

if (it != this->tree.end()) {

return false;

}

it = this->tree.find(*parent*);

if (it == this->tree.end()) {

return false;

}

std::set <*int*> NullSet;

this->tree[*id*] = NullSet;

this->tree[*parent*].insert(*id*);

return true;

}

*bool* Tree::delete\_node(*int* *id*)

{

std::map <*int*, std::set <*int*>> :: iterator it;

std::set <*int*> :: iterator it\_set;

it = this->tree.find(*id*);

if (it == this->tree.end()) {

return false;

}

std::set <*int*> node\_childs = this->tree[*id*];

for (const *auto* &iter : this->tree) {

it\_set = iter.second.find(*id*);

if (it\_set != iter.second.end()) {

this->tree[iter.first].erase(it\_set);

break;

}

}

for (const *auto* &iter\_child : node\_childs) {

this->delete\_node(iter\_child);

}

this->tree.erase(it);

return true;

}

std::set <*int*> Tree::get\_node\_list()

{

std::set <*int*> node\_set;

for (*auto* iter : this->tree) {

node\_set.insert(iter.first);

}

return node\_set;

}

*void* Tree::print\_tree()

{

for (*auto* iter : this->tree) {

std::cout << iter.first << ": ";

for (*auto* it : iter.second) {

std::cout << it << " ";

}

std::cout << "\n";

}

}

Tree::Tree()

{

std::set <*int*> NullSet;

tree[-1] = NullSet;

}

zmq\_manage.cpp

#include "../include/zmq\_manage.h"

std::string get\_port\_name(const *int* *port*)

{

return "tcp://127.0.0.1:" + std::to\_string(port + 1);

}

*bool* send\_message(zmq::*socket\_t* &*socket*, const std::string &*message\_string*)

{

zmq::*message\_t* message(message\_string.size());

memcpy(message.data(), message\_string.c\_str(), message\_string.size());

return socket.send(message);

}

std::string recieve\_message(zmq::*socket\_t* &*socket*)

{

zmq::*message\_t* message;

*bool* ok = false;

try

{

ok = socket.recv(&message);

}

catch (...)

{

ok = false;

}

std::string recieved\_message(static\_cast <*char*\*> (message.data()), message.size());

if (recieved\_message.empty() || !ok)

{

return "Error: Node is not available";

}

return recieved\_message;

}

*void* create\_node(*int* *id*, *int* *port*)

{

*char*\* arg0 = strdup("./server");

*char*\* arg1 = strdup((std::to\_string(id)).c\_str());

*char*\* arg2 = strdup((std::to\_string(port)).c\_str());

*char*\* args[] = {arg0, arg1, arg2, NULL};

execv("./server", args);

}

CMakeLists.txt

cmake\_minimum\_required(VERSION 3.8 FATAL\_ERROR)

project(lab6 LANGUAGES CXX)

add\_library(

lib\_tree SHARED

./include/tree.h

./src/tree.cpp

)

add\_library(

lib\_zmq\_manage SHARED

./include/zmq\_manage.h

./src/zmq\_manage.cpp

)

add\_executable(api\_daemon ./src/api\_daemon.cpp)

target\_link\_libraries(api\_daemon zmq lib\_tree lib\_zmq\_manage)

target\_compile\_options(api\_daemon PRIVATE -fsanitize=address -g)

target\_link\_options(api\_daemon PRIVATE -fsanitize=address -g)

add\_executable(client ./src/client.cpp)

target\_link\_libraries(client zmq lib\_tree lib\_zmq\_manage)

target\_compile\_options(client PRIVATE -fsanitize=address -g)

target\_link\_options(client PRIVATE -fsanitize=address -g)

add\_executable(server ./src/server.cpp)

target\_link\_libraries(server zmq lib\_zmq\_manage)

target\_compile\_options(server PRIVATE -fsanitize=address -g)

target\_link\_options(server PRIVATE -fsanitize=address -g)

Демонстрация работы программы

lyutoev@lyutoev  ~/workshop/os/OS/lab6/build   main ±  ./client

commands:

       create id parent

       remove id

       exec id var\_name value

       exec id var\_name

       ping id

       heartbeat time

create 1 -1

Ok: 26793

create 2 -2

Error: Not found -2

Ok: 0

create 3 -1

Ok: 26813

create 2 3

Ok: 26822

create 4 1

Ok: 26830

create 6 -1

Ok: 26843

create 8 1

Ok: 26861

print

-1: 1 3 6

1: 4 8

2:

3: 2

4:

6:

8:

ping 5

Ok: 0

ping 1

Ok: 1

exec 1 a -1

Ok: 1

exec 2 a

Ok:2: 'a' not found

exec 1 a

Ok:1: -1

exec 2 a 1

Ok: 2

exec 2 a

Ok:2: 1

heartbeat 4000

Ok

exit

Ok

lyutoev@lyutoev  ~/workshop/os/OS/lab6/build   main ± 

Выводы

В ходе выполнения данной лабораторной работы я написал достаточно сложный проект, научился создавать приложения использующие асинхронность и работать с очерядями сообщеий на примере ZeroMQ.