Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

**Сервер сообщений ZeroMQ**

Студент: Камеш Михаил Алексеевич

Группа: М8О–207Б–18

Вариант: 34

Преподаватель: Соколов Андрей Алексеевич

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. Примеры работы
6. Вывод

Постановка задания

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.

Вариант задания.

Топология 2 – список списков вычислительных узлов.

Тип задания вычислительных узлов 4 – поиск подстроки в строке.

Команда проверки 1 – ping all

Общие сведения о программе

Основной файл main содержит в себе списки текущих процессов, и пути доступа к ним, при создании он отправляет работникам сообщения о создании и наше мини дерево

перестраивается в соответствии с конфигурацией. Внутри каждого работника есть два нода, которые хранят данные о пути выше или ниже от себя, для пересылки им сообщений.

Все данное действо написано используя 0mq – библиотеку для С++, в которой возможна реализация передачи данных по TCP протоколу.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Класс сообщений, где есть методы отправки и принятия различных сообщений.

2. Класс работников содержит в себе параметры связи с сокетами, привязки к определенным портам, а также методы по перестройке или связи и отправке сообщений между собой.

3. Коллектор, он отвечает за асинхронность выполнения задач, собирая данные после их

завершения по команде update.

Файлы

Package.hpp

#ifndef MESSAGE\_HPP

#define MESSAGE\_HPP

#include "zmq.hpp"

#include <string>

#include <ctime>

#include <vector>

#include <unistd.h>

#include <iostream>

#include "Worker.hpp"

class Package {

public:

enum {

ERR, CREATE, REMOVE,

PING, REPLY, TAKEPORT,

REPAR, EXEC, TERM,

KILLNPASS,

COLL\_UPDATE, COLL\_NEW\_TASK, COLL\_RESPOND,

COLL\_EXIT, COLL\_RES, COLL\_GET\_RESULT,

COLL\_REMOVE

};

int type;

int id;

int data;

Package() {}

Package(Worker& from) { recv(from); }

Package(int type, int id, int data) : type(type), id(id), data(data) {}

int send(Worker& to) {

zmq::message\_t typeMes(&type, sizeof(int));

zmq::message\_t idMes(&id, sizeof(int));

zmq::message\_t dataMes(&data, sizeof(int));

auto start = clock();

while (true) {

if (to.Sock().send(typeMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(idMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(dataMes, zmq::send\_flags::dontwait) )

{

return 1;

}

if (clock() - start > (CLOCKS\_PER\_SEC)) return 0;

}

}

void recv(Worker& from) {

zmq::message\_t typeMes;

zmq::message\_t idMes;

zmq::message\_t dataMes;

from.Sock().recv(typeMes, zmq::recv\_flags::none);

from.Sock().recv(idMes, zmq::recv\_flags::none);

from.Sock().recv(dataMes, zmq::recv\_flags::none);

type = \*((int\*)(typeMes.data()));

id = \*((int\*)(idMes.data()));

data = \*((int\*)(dataMes.data()));

}

std::string recv\_string(Worker& from) {

zmq::message\_t pack;

from.Sock().recv(pack, zmq::recv\_flags::dontwait);

char \* text = (char \*)pack.data();

std::string res(text);

return res;

}

std::vector <int> recv\_vector(Worker& from) {

zmq::message\_t pack;

from.Sock().recv(pack, zmq::recv\_flags::dontwait);

int \* data = (int \*)pack.data();

std::vector <int> res;

for (size\_t i = 1; i <= data[0]; i++)

res.push\_back(data[i]);

return res;

}

int recvCheck(Worker& from) {

zmq::message\_t typeMes;

zmq::message\_t idMes;

zmq::message\_t dataMes;

auto start = clock();

while (true) {

if (from.Sock().recv(typeMes, zmq::recv\_flags::dontwait) &&

from.Sock().recv(idMes, zmq::recv\_flags::dontwait) &&

from.Sock().recv(dataMes, zmq::recv\_flags::dontwait) )

{

type = \*((int\*)(typeMes.data()));

id = \*((int\*)(idMes.data()));

data = \*((int\*)(dataMes.data()));

return 1;

}

if (clock() - start > (CLOCKS\_PER\_SEC)) return 0;

}

}

int send\_exec(Worker& to, std::string text, std::string pattern) {

zmq::message\_t typeMes(&type, sizeof(int));

zmq::message\_t idMes(&id, sizeof(int));

zmq::message\_t dataMes(&data, sizeof(int));

zmq::message\_t textMes(text.data(), sizeof(text.size() + 1));

zmq::message\_t patternMes(pattern.data(), sizeof(pattern.size() + 1));

auto start = clock();

while (true) {

if (to.Sock().send(typeMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(idMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(dataMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(textMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(patternMes, zmq::send\_flags::dontwait))

{

return 1;

}

if (clock() - start > (CLOCKS\_PER\_SEC)) return 0;

}

}

int send\_res(Worker& to, std::vector <int> res) {

zmq::message\_t typeMes(&type, sizeof(int));

zmq::message\_t idMes(&id, sizeof(int));

zmq::message\_t dataMes(&data, sizeof(int));

zmq::message\_t textMes(res.data(), sizeof(int) \* res.size());

auto start = clock();

while (true) {

if (to.Sock().send(typeMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(idMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(dataMes, zmq::send\_flags::sndmore | zmq::send\_flags::dontwait) &&

to.Sock().send(textMes, zmq::send\_flags::dontwait))

{

return 1;

}

if (clock() - start > (CLOCKS\_PER\_SEC)) return 0;

}

}

};

#endif

Worker.hpp

#ifndef WORKER\_HPP

#define WORKER\_HPP

#include "zmq.hpp"

#include <string>

#include <vector>

#include <unistd.h>

#include <iostream>

#include <thread>

int BEGIN\_PORT = 10000;

const char \* BIND\_URL = "tcp://\*:";

const char \* CON\_URL = "tcp://localhost:";

std::string BindURLPort(int port) { return BIND\_URL + std::to\_string(port); }

std::string ConURLPort(int port) { return CON\_URL + std::to\_string(port); }

int TakePort(zmq::socket\_t& sock) {

int port = BEGIN\_PORT;

while (true) {

try { sock.bind(BindURLPort(port));

break;

} catch (const zmq::error\_t& err) { ++port; }

}

return port;

}

class Worker {

public:

enum {

ERR, CHILD,

PARENT, NEW, ADD, COLLECTOR

};

Worker() : id(-2), ctx(1), sock(ctx, ZMQ\_PAIR) {}

Worker(int type, int id, int inPort = -1) : ctx(1), sock(ctx, ZMQ\_PAIR), id(id) {

switch(type) {

case CHILD:

port = inPort;

sock.bind(BindURLPort(port));

break;

case PARENT:

port = inPort;

sock.connect(ConURLPort(port));

break;

case NEW:

port = TakePort(sock);

pid = fork();

if (pid == 0) {

using std::to\_string;

execl("node", to\_string(id).c\_str(), to\_string(port).c\_str(), (char\*)NULL);

}

break;

case COLLECTOR:

port = TakePort(sock);

break;

}

}

void New(int inId) {

id = inId;

port = TakePort(sock);

pid = fork();

if (pid == 0) {

using std::to\_string;

execl("node", to\_string(id).c\_str(), to\_string(port).c\_str(), (char\*)NULL);

}

}

void TakePortSetId(int Id) {

id = Id;

port = TakePort(sock);

}

void disConnect() { sock.disconnect(ConURLPort(port)); }

pid\_t Pid() { return pid; }

int& Id() { return id; }

int Port() { return port;}

zmq::socket\_t& Sock() { return sock;}

private:

int id;

int port;

pid\_t pid;

zmq::context\_t ctx;

zmq::socket\_t sock;

};

#endif

Work.c

#include "zmq.hpp"

#include <string>

#include <vector>

#include <chrono>

#include <mutex>

#include <thread>

#include "Worker.hpp"

#include "Package.hpp"

enum {

WORKER\_EMPTY,

WORKER\_FINISHED,

WORKER\_WORKING

};

std::mutex W\_lock;

std::vector <int> result\_of\_work;

int worker\_state = WORKER\_EMPTY;

std::thread \* worker = NULL;

std::vector <int> findPatt(std::string text, std::string patern) {

std::this\_thread::sleep\_for(std::chrono::seconds(1));

std::cout << text << " " << patern << "\n";

std::vector <int> pos; bool par;

for (size\_t i = 0; i < static\_cast<size\_t>(text.size()); i++) {

par = true;

if (text.size() - i < patern.size())

break;

for (size\_t j = 0; j < patern.size(); j++)

if (patern[j] != text[i + j]) {

par = false;

break;

}

if (par)

pos.push\_back(i);

}

for (int a : pos) {

std::cout << a << " ";

}

std::cout << "\n";

if (pos.size() == 0) {

pos.push\_back(1);

pos.push\_back(-1);

} else {

pos.insert(pos.begin(), pos.size());

}

return pos;

}

void foo(std::string text, std::string patt) {

std::vector <int> res = findPatt(text, patt);

W\_lock.lock();

worker\_state = WORKER\_FINISHED;

result\_of\_work.clear();

result\_of\_work = res;

W\_lock.unlock();

return;

}

int main(int argc, char \*\* argv) {

if (argc != 2)

return 0;

int Id = atoi(argv[0]);

Worker \* Parent = new Worker(Worker::PARENT, 0, atoi(argv[1]));

Worker \* Child = new Worker;

Worker \* tmp;

Package to, from;

while(true) {

from.recv(\*Parent);

switch (from.type) {

case Package::CREATE: {

if (Id == from.data) {

if (Child->Id() == -2) {

Child->New(from.id);

from.type = Package::REPLY;

from.data = Child->Pid();

from.send(\*Parent);

} else {

from.type = Package::ERR;

from.send(\*Parent);

}

} else {

from.send(\*Child);

if (!to.recvCheck(\*Child)) break;

else to.send(\*Parent);

}

break;

} case Package::REMOVE: {

if (Child->Id() == from.id){

tmp = new Worker();

tmp->TakePortSetId(0);

to.type = Package::KILLNPASS;

to.data = tmp->Port();

to.send(\*Child);

if(!to.recvCheck(\*Child)) {

to.type = Package::TERM;

to.send(\*tmp);

delete tmp;

break;

}

if (to.type != Package::ERR) {

tmp->Id() = to.id;

std::swap(tmp, Child);

delete tmp;

} else {

std::cout << "Error: while removing\n";

}

from.send(\*Parent);

break;

} else {

from.send(\*Child);

if(!to.recvCheck(\*Child)) { break; }

else { to.send(\*Parent); }

break;

}

} case Package::KILLNPASS: {

if (Child->Id() != -2) {

to.type = Package::REPAR;

to.data = from.data;

to.send(\*Child);

}

to.type = Package::REPLY;

to.id = Child->Id();

to.send(\*Parent);

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

return 0;

break;

} case Package::TAKEPORT: {

Child->TakePortSetId(from.id);

from.type = Package::REPLY;

from.data = Child->Port();

from.send(\*Parent);

break;

} case Package::REPAR: {

tmp = new Worker(Worker::PARENT, 0, from.data);

std::swap(tmp, Parent);

delete tmp;

break;

} case Package::TERM: {

if (Child->Id() != -2)

from.send(\*Child);

// worker->join();

// delete worker;

// delete Parent;

// delete Child;

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

return 0;

break;

} case Package::EXEC: {

if (Id == from.id && worker\_state == WORKER\_EMPTY) {

worker\_state = WORKER\_WORKING;

std::string text = from.recv\_string(\*Parent);

std::string patt = from.recv\_string(\*Parent);

worker = new std::thread(foo, text, patt);

} else if (Child->Id() != -2) {

std::string text = from.recv\_string(\*Parent);

std::string patt = from.recv\_string(\*Parent);

from.send\_exec(\*Child, text, patt);

}

break;

} case Package::COLL\_GET\_RESULT: {

if (from.id == Id) {

W\_lock.lock();

if (worker\_state == WORKER\_FINISHED) {

to.type = Package::COLL\_RES;

to.id = -10;

to.data = Id;

to.send\_res(\*Parent, result\_of\_work);

worker\_state = WORKER\_EMPTY;

worker->join();

// delete worker;

}

W\_lock.unlock();

} else if (Child->Id() != -2) {

from.send(\*Child);

if (!to.recvCheck(\*Child))

break;

else

to.send\_res(\*Parent, to.recv\_vector(\*Child));

}

break;

} case Package::PING: {

if (Id == from.id) {

to.type = Package::REPLY;

to.send(\*Parent);

} else {

from.send(\*Child);

if (!to.recvCheck(\*Child))

break;

else

to.send(\*Parent);

}

break;

}

}

}

return 0;

}

Main.c

#include <iostream>

#include "zmq.hpp"

#include <string>

#include <vector>

#include <algorithm>

#include <map>

#include "Worker.hpp"

#include "Package.hpp"

#include <thread>

enum {

ERR, CREATE,

REMOVE, EXEC,

PING, EXIT, UPDATE

};

std::vector<Worker\*> lists;

std::map<int, int> NodesAdr;

std::string comId;

std::map <std::string, int> command = {

{"create", CREATE}, {"exec", EXEC},

{"remove", REMOVE}, {"ping", PING},

{"q", EXIT}, {"update", UPDATE}

};

std::string text, pattern;

Package to, from;

int id, parent, listID;

void run\_collector(int ParentPort) {

Worker main\_node (Worker::PARENT, -1, ParentPort);

Package mess\_to, mess\_from;

std::vector <int> tasks, pivet;

while (true) {

mess\_from.recv(main\_node);

switch (mess\_from.type) {

case Package::COLL\_UPDATE:

std::cout << "coll :: have " << tasks.size() << " tasks" << std::endl;

for (auto t : tasks) {

std::cout << t << " ";

}

std::cout << "\n";

begin:

for (int task : tasks) {

listID = NodesAdr[task];

mess\_to.type = Package::COLL\_GET\_RESULT;

mess\_to.id = task;

mess\_to.send(\*lists[listID]);

std::cout << "coll :: send " << task << "\n";

if(!mess\_from.recvCheck(\*lists[listID])) {

std::cout << "task " << task << " did not finished\n";

} else {

pivet.clear();

pivet = mess\_from.recv\_vector(\*lists[listID]);

std::cout << task << " ::";

for (size\_t j = 0; j < pivet.size(); j++)

std::cout << " " << pivet[j];

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

auto pos = std::find(tasks.begin(), tasks.end(), task);

tasks.erase(pos);

goto begin;

}

}

break;

case Package::COLL\_NEW\_TASK: {

auto res = std::find(tasks.begin(), tasks.end(), mess\_from.id);

if (res != tasks.end()) {

mess\_to.data = 1;

} else {

mess\_to.data = 0;

tasks.push\_back(mess\_from.id);

}

std::cout << "coll :: added new task " << mess\_from.id << std::endl;

break;

}

case Package::COLL\_EXIT:

std::cout << "coll :: exit" << std::endl;

return;

break;

case Package::COLL\_REMOVE:

if (tasks.size() != 0) {

mess\_to.data = 1;

} else {

mess\_to.data = 0;

}

break;

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

mess\_to.type = Package::COLL\_RESPOND;

mess\_to.send(main\_node);

}

}

int main(void) {

std::vector<int> allids;

std::vector<int> notrespondingsids;

int exit = 0;

Worker collector (Worker::COLLECTOR, -10);

std::thread collector\_th (run\_collector, collector.Port());

NodesAdr[-1] = -1;

while(!exit) {

std::cin >> comId;

switch (command[comId]) {

case CREATE: {

int id, parent;

std::cin >> id >> parent;

if (NodesAdr.find(parent) == NodesAdr.end()) {

std::cout << "Error: Parent not found\n";

break;

}

if (NodesAdr.find(id) != NodesAdr.end()) {

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

break;

}

if (parent == -1) {

lists.push\_back(new Worker(Worker::NEW, id));

allids.push\_back(id);

pid\_t added = lists[lists.size() - 1]->Pid();

NodesAdr[id] = lists.size() - 1;

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

} else {

listID = NodesAdr[parent];

allids.push\_back(id);

to.type = Package::CREATE;

to.id = id;

to.data = parent;

to.send(\*lists[listID]);

if(!from.recvCheck(\*lists[listID])) {

std::cout << "Error: Parent is unavailable\n";

break;

}

if (from.type != Package::ERR) {

std::cout << "Ok: " << from.data << '\n';

NodesAdr[id] = NodesAdr[parent];

} else {

std::cout << "Error: in creation\n";

}

}

break;

} case REMOVE: {

std::cin >> id;

if (NodesAdr.find(id) == NodesAdr.end() || id == -1) {

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

break;

}

to.type = Package::COLL\_REMOVE;

to.data = 0;

to.id = -1;

to.send(collector);

from.recv(collector);

if (from.data == 1) {

std::cout << "Error: No remove while job\n";

break;

}

listID = NodesAdr[id];

if (lists[listID]->Id() == id){

to.type = Package::KILLNPASS;

Worker\* tmp = new Worker();

tmp->TakePortSetId(0);

to.data = tmp->Port();

to.send(\*lists[listID]);

if(!from.recvCheck(\*lists[listID])) {

std::cout << "Error: Worker is unavailable\n";

to.type = Package::TERM;

to.send(\*tmp);

delete tmp;

break;

}

if (from.type != Package::ERR) {

tmp->Id() = from.id;

std::swap(tmp, lists[listID]);

delete tmp;

if (lists[listID]->Id() == -2) {

delete lists[listID];

lists[listID] = nullptr;

}

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

NodesAdr.erase(id);

} else {

std::cout << "Error: Worker is unavailable\n";

to.type = Package::TERM;

to.send(\*tmp);

delete tmp;

}

break;

} else {

to.type = Package::REMOVE;

to.id = id;

to.send(\*lists[listID]);

if(!from.recvCheck(\*lists[listID])) {

std::cout << "Error: Worker is unavailable\n";

break;

}

if (from.type != Package::ERR) {

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

NodesAdr.erase(id);

} else {

std::cout << "Error: Worker is unavailable\n";

}

}

break;

} case UPDATE: {

to.type = Package::COLL\_UPDATE;

to.data = 0;

to.id = -1;

to.send(collector);

from.recv(collector);

break;

} case EXEC: {

std::cin >> id;

if (NodesAdr.find(id) == NodesAdr.end()) {

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

break;

}

listID = NodesAdr[id];

to.type = Package::COLL\_NEW\_TASK;

to.id = id;

to.send(collector);

if (!from.recvCheck(collector)) {

std::cout << "Error: create task" << std::endl;

break;

};

if (from.data == 0) {

to.type = Package::EXEC;

to.id = id;

std::cin >> text >> pattern;

to.send\_exec(\*lists[listID], text, pattern);

} else {

std::cout << "node is busy" << std::endl;

}

break;

} case PING: {

for (auto i = allids.begin(); i!= allids.end(); ++i) {

listID = NodesAdr[\*i];

to.type = Package::PING;

to.id = \*i;

to.send(\*lists[listID]);

if(!from.recvCheck(\*lists[listID])) {

notrespondingsids.push\_back(\*i);

}

}

if (notrespondingsids.empty()) {

std::cout << "OK: -1" << std::endl;

}

else {

for (auto i = notrespondingsids.begin(); i!= notrespondingsids.end(); ++i) {

std::cout << \*i;

std::cout << " ";

}

std::cout << std::endl;

notrespondingsids.clear();

}

break;

} case EXIT: {

exit = 1;

to.type = Package::COLL\_EXIT;

to.send(collector);

collector\_th.join();

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

break;

} case ERR: {

std::cout << "Wrong Command\n";

break;

}

}

}

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

to.type = Package::TERM;

if (lists[i] != nullptr) {

to.send(\*lists[i]);

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

}

delete lists[i];

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

return 0;

}

Демонстрация работы программы.

create 20 -1

OK

create 21 20

OK

create 22 21

OK

create 23 22

OK

create 24 23

OK

create 25 24

OK

create 30 -1

OK

create 31 30

OK

create 32 31

OK

ping all

OK -1

delete 22

delete 31

ping all

OK 22 31

exec 21

adding task to 21

qweqwe

wewe

OK 1 4

Выводы

В ходе проведения работы была реализовна распределенную система по асинхронной обработке запросов. Была создана структуау для обработки и получения

сообщений, но в виду архитектуры передачи текста нефиксированной длинны, размер которой может быть неограничен, сообщения разбиты по классам,

в зависимости от их использования, что затрудняло отладку, так как в случае ошибки нельзя понять, что находится в сокете, пока процесс не прочитает

его и не обработает, часто в таком случае – неправильно, сломав всю программу. 0mq – как заверяют разработчики очень быстрый способ передачи сообщений,

были некоторые проблемы с данной библиотекой и разными версиями функций внутри неё.