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»**

Студент:Шандрюк Пётр Николаевич

Группа: М8О-208Б-20

Вариант: 7

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Сборка программы
7. Демонстрация работы программы
8. Выводы

**Репозиторий**

https://github.com/Peter1811/OS/tree/main/lab6-8

**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» 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])
* **Исполнение команды на вычислительном узле** (Формат команды: exec id [params])
* **Проверка доступности узла** (Формат команды: ping id)
* **Удаление узла** (Формат команды remove id)

**Вариант №9:** топология — список, команда — работа с локальным словарем, проверка доступности — ping id.

**Общие сведения о программе**

Связь между вычислительными узлами будем поддерживать с помощью ZMQ\_PAIR. При инициализации установить время ожидания ZMQ\_SNDTIMEO и ZMQ\_RECVTIMEO, чтобы предусмотреть случай, когда дочерний процесс был убит. Для обмена информацией будем использовать специальную структуру node\_token\_t, в которой есть перечислимое поле actions. Вычислительные узлы обрабатывают каждое сообщение: если идентификатор сообщения не совпадает с идентификатором узла, то он отправляет сообщение дальше и ждёт ответа снизу.

**Общий метод и алгоритм решения**

Используемые методы системные вызовы:

| zmq\_ctx\_new() | Создает новый ØMQ контекст |
| --- | --- |
| void \*zmq\_socket (void \*context, int type); | Создает ØMQ сокет |
| int zmq\_setsockopt (void \*socket, int option\_name, const void \*option\_value, size\_t option\_len); | Устанавливает опции ØMQ сокета |
| int zmq\_msg\_init (zmq\_msg\_t \*msg); | Инициализирует пустое ØMQ сообщение |
| int zmq\_msg\_recv (zmq\_msg\_t \*msg, void \*socket, int flags); | Получает часть сообщения из сокета |
| int zmq\_msg\_close (zmq\_msg\_t \*msg); | Освобождает сообщение ØMQ |
| int zmq\_msg\_init\_size (zmq\_msg\_t \*msg, size\_t size); | Инициализирует ØMQ сообщение определенного размера |
| int zmq\_msg\_init\_data (zmq\_msg\_t \*msg, void \*data, size\_t size, zmq\_free\_fn \*ffn, void \*hint); | Инициализирует сообщение ØMQ из предоставленного буфера. |
| int zmq\_msg\_send (zmq\_msg\_t \*msg, void \*socket, int flags); | Отправляет часть сообщения на сокет |
| int zmq\_bind (void \*socket, const char \*endpoint); | Принимает входящие соединения на сокет |
| int zmq\_close (void \* socket ); | Закрывает сокет ØMQ |
| int zmq\_ctx\_term (void \* context ); | Уничтожает контекст ØMQ |
| assert(expr) | Прекращает работу программы при ложном утверждении |

**Исходный код**

**main\_prog.cpp**

#include <zmq.hpp>

#include <unistd.h>

#include <iostream>

#include <vector>

using namespace std;

int main() {

zmq::context\_t context(1);

zmq::socket\_t main\_socket(context, ZMQ\_REP);

string adr = "tcp://127.0.0.1:300";

string command;

bool is\_empty = false;

int child\_id = 0;

main\_socket.setsockopt(ZMQ\_RCVTIMEO, 3000);

// main\_socket.setsockopt(ZMQ\_SNDTIMEO, 3000);

cout << "Commands:\ncreate id\nexec id text pattern\nping id\nkill id\nexit\n";

while (1) {

cout << "Please, enter command\n";

cin >> command;

if (command == "create") {

if (child\_id == 0) {

int id;

cin >> id;

int id\_tmp = id - 1;

main\_socket.bind(adr + to\_string(++id\_tmp));

string new\_adr = adr + to\_string(id\_tmp);

char \*adr\_ = new char[new\_adr.size() + 1];

memcpy(adr\_, new\_adr.c\_str(), new\_adr.size() + 1);

char \*id\_ = new char[to\_string(id).size() + 1];

memcpy(id\_, to\_string(id).c\_str(), to\_string(id).size() + 1);

char \*args[] = {"./child\_node", adr\_, id\_, NULL};

int id2 = fork();

if (id2 == -1) {

std::cout << "ERROR: CALCULATING NODE WAS NOT CREATED\n";

id = 0;

exit(1);

}

else if (id2 == 0) {

execv("./child\_node", args);

}

else {

child\_id = id;

}

zmq::message\_t message;

main\_socket.recv(&message);

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

cout << recieved\_message << "\n";

delete[] adr\_;

delete[] id\_;

}

else {

int id;

cin >> id;

string message\_string = command + " " + to\_string(id);

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

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

main\_socket.send(message);

main\_socket.recv(&message);

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

cout << recieved\_message << "\n";

}

}

else if (command == "exec") {

int id;

string text, pattern, s;

cin >> id >> text >> pattern;

s = text + "$" + pattern;

string message\_string = command + " " + to\_string(id) + " " + s;

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

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

main\_socket.send(message);

main\_socket.recv(&message);

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

cout << recieved\_message << "\n";

}

else if (command == "ping") {

int id;

cin >> id;

string message\_string = command + " " + to\_string(id);

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

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

main\_socket.send(message);

// receive answer from child

main\_socket.recv(&message);

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

cout << recieved\_message << "\n";

}

else if (command == "kill") {

int id;

cin >> id;

if (child\_id == 0) {

cout << "Error: there aren't any nodes\n";

}

else if (child\_id == id) {

string kill\_message = "DIE";

zmq::message\_t message(kill\_message.size());

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

main\_socket.send(message);

cout << "Tree was deleted\n";

child\_id = 0;

}

else {

string kill\_message = command + " " + to\_string(id);

zmq::message\_t message(kill\_message.size());

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

main\_socket.send(message);

main\_socket.recv(&message);

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

cout << received\_message << "\n";

}

}

else if (command == "exit") {

if (child\_id) {

string kill\_message = "DIE";

zmq::message\_t message(kill\_message.size());

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

main\_socket.send(message);

cout << "Tree was deleted\n";

child\_id = 0;

}

main\_socket.close();

context.close();

return 0;

}

else {

cout << "Error: incorrect command\n";

}

}

}

**Child\_node.cpp**

#include <zmq.hpp>

#include <iostream>

#include <unistd.h>

#include <fstream>

using namespace std;

void send\_message(std::string message\_string, zmq::socket\_t &socket) {

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

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

if (!socket.send(message\_back))

{

std::cout << "Error: can't send message from node with pid " << getpid() << "\n";

}

}

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

string adr = argv[1];

zmq::context\_t context(1);

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

main\_socket.setsockopt(ZMQ\_RCVTIMEO, 3000);

main\_socket.setsockopt(ZMQ\_SNDTIMEO, 3000);

main\_socket.connect(argv[1]);

send\_message("OK: " + to\_string(getpid()), main\_socket);

int id = stoi(argv[2]); //

int left\_id = 0;

int right\_id = 0;

zmq::context\_t context\_l(1);

zmq::context\_t context\_r(1);

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

string adr\_left = "tcp://127.0.0.1:300";

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

left\_socket.setsockopt(ZMQ\_RCVTIMEO, 3000);

left\_socket.setsockopt(ZMQ\_SNDTIMEO, 3000);

right\_socket.setsockopt(ZMQ\_RCVTIMEO, 3000);

right\_socket.setsockopt(ZMQ\_SNDTIMEO, 3000);

string adr\_right = "tcp://127.0.0.1:300";

while (1) {

zmq::message\_t message\_main;

main\_socket.recv(&message\_main);

//sleep(1);

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

string command;

for (int i = 0; i < recieved\_message.size(); ++i) {

if (recieved\_message[i] != ' ') {

command += recieved\_message[i];

}

else {

break;

}

}

if (command == "exec") { //----------EXEC----------

int id\_proc;

string id\_proc\_, value\_;

string key;

int pos;

for (int i = 5; i < recieved\_message.size(); ++i) {

if (recieved\_message[i] != ' ') {

id\_proc\_ += recieved\_message[i];

}

else {

pos = i;

break;

}

}

//0123456

//exec 2 55$2$3$121$

id\_proc = stoi(id\_proc\_);

if (id\_proc == id) { // id == proc\_id, this node must execute

int j = pos;

string text, pattern;

string res = "OK:" + to\_string(id) + ":";

while (recieved\_message[j] != '$') {

text.push\_back(recieved\_message[j]);

++j;

}

++j;

while (j != recieved\_message.size()) {

pattern.push\_back(recieved\_message[j]); //tttt t

++j;

}

int flag = 0;

if (pattern.size() <= text.size()) {

//cout << text << " " << pattern << "\n";

for (int i = 0; i <= text.size() - pattern.size(); ++i) {

//cout << "TEST";

string pat;

for (int z = 0; z < pattern.size(); ++z)

{

if (text[i + z] == pattern[z])

pat.push\_back(text[i + z]);

}

if (pat == pattern)

{

res = res + to\_string(i) + ";";

flag = 1;

}

pat.clear();

}

}

if (flag == 0)

{

res = res + "-1";

}

zmq::message\_t message(res.size());

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

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

else

{ // id != prod\_id

if (id > id\_proc)

{ //go to left

if (left\_id == 0)

{ // if node not exists

string message\_string = "Error:id: Not found";

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

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

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

else

{

zmq::message\_t message(recieved\_message.size());

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

if (!left\_socket.send(message))

{

cout << "Error: can't send message to left node from node with pid: " << getpid() << "\n";

}

// catch and send to parent

if (!left\_socket.recv(&message))

{

cout << "Error: can't receive message from left node in node with pid: " << getpid() << "\n";

}

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

}

else

{ // go to right

if (right\_id == 0)

{ // if node not exists

string message\_string = "Error:id: Not found";

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

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

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

else

{

zmq::message\_t message(recieved\_message.size());

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

if (!right\_socket.send(message))

{

cout << "Error: can't send message to right node from node with pid: " << getpid() << "\n";

}

// catch and send to parent

if (!right\_socket.recv(&message))

{

cout << "Error: can't receive message from left node in node with pid: " << getpid() << "\n";

}

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

}

}

}

else if (command == "create")

{

int id\_proc; // id of node for creating

string id\_proc\_;

for (int i = 7; i < recieved\_message.size(); ++i)

{

if (recieved\_message[i] != ' ')

{

id\_proc\_ += recieved\_message[i];

}

else

{

break;

}

}

id\_proc = stoi(id\_proc\_);

if (id\_proc == id)

{

send\_message("Error: Already exists", main\_socket);

}

else if (id\_proc > id)

{

if (right\_id == 0)

{ // there is not right node

right\_id = id\_proc;

int right\_id\_tmp = right\_id - 1;

while (1)

{

try

{

right\_socket.bind(adr\_right + to\_string(++right\_id\_tmp));

break;

}

catch (...)

{

}

}

adr\_right += to\_string(right\_id\_tmp);

char \*adr\_right\_ = new char[adr\_right.size() + 1];

memcpy(adr\_right\_, adr\_right.c\_str(), adr\_right.size() + 1);

char \*right\_id\_ = new char[to\_string(right\_id).size() + 1];

memcpy(right\_id\_, to\_string(right\_id).c\_str(), to\_string(right\_id).size() + 1);

char \*args[] = {"./child\_node", adr\_right\_, right\_id\_, NULL};

int f = fork();

if (f == 0)

{

execv("./child\_node", args);

}

else if (f == -1)

{

cout << "Error in forking in node with pid: " << getpid() << "\n";

}

else

{

zmq::message\_t message\_from\_node;

if (!right\_socket.recv(&message\_from\_node))

{

cout << "Error: can't receive message from right node in node with pid:" << getpid() << "\n";

}

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

if (!main\_socket.send(message\_from\_node))

{

cout << "Error: can't send message to main node from node with pid:" << getpid() << "\n";

}

}

delete[] adr\_right\_;

delete[] right\_id\_;

}

else

{ // send task to right node

send\_message(recieved\_message, right\_socket);

// catch and send to parent

zmq::message\_t message;

if (!right\_socket.recv(&message))

{

cout << "Error: can't receive message from left node in node with pid: " << getpid() << "\n";

}

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

}

else

{

if (left\_id == 0)

{ // there is not left node

left\_id = id\_proc;

int left\_id\_tmp = left\_id - 1;

while (1)

{

try

{

left\_socket.bind(adr\_left + to\_string(++left\_id\_tmp));

break;

}

catch (...)

{

}

}

adr\_left += to\_string(left\_id\_tmp);

char \*adr\_left\_ = new char[adr\_left.size() + 1];

memcpy(adr\_left\_, adr\_left.c\_str(), adr\_left.size() + 1);

char \*left\_id\_ = new char[to\_string(left\_id).size() + 1];

memcpy(left\_id\_, to\_string(left\_id).c\_str(), to\_string(left\_id).size() + 1);

char \*args[] = {"./child\_node", adr\_left\_, left\_id\_, NULL};

int f = fork();

if (f == 0)

{

execv("./child\_node", args);

}

else if (f == -1)

{

cout << "Error in forking in node with pid: " << getpid() << "\n";

}

else

{

// catch message from new node

zmq::message\_t message\_from\_node;

if (!left\_socket.recv(&message\_from\_node))

{

cout << "Error: can't receive message from left node in node with pid:" << getpid() << "\n";

}

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

// send message to main node

if (!main\_socket.send(message\_from\_node))

{

cout << "Error: can't send message to main node from node with pid:" << getpid() << "\n";

}

}

delete[] adr\_left\_;

delete[] left\_id\_;

}

else

{ // send task to left node

send\_message(recieved\_message, left\_socket);

// catch and send to parent

zmq::message\_t message;

if (!left\_socket.recv(&message))

{

cout << "Error: can't receive message from left node in node with pid: " << getpid() << "\n";

}

if (!main\_socket.send(message))

{

cout << "Error: can't send message to main node from node with pid: " << getpid() << "\n";

}

}

}

}

else if (command == "ping")

{

int id\_proc; // id of node for creating

string id\_proc\_;

for (int i = 5; i < recieved\_message.size(); ++i)

{

if (recieved\_message[i] != ' ')

{

id\_proc\_ += recieved\_message[i];

}

else

{

break;

}

}

id\_proc = stoi(id\_proc\_);

if (id\_proc == id)

{

string s = "OK: 1";

zmq::message\_t message(s.size());

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

main\_socket.send(message);

}

else if (id\_proc < id)

{

if (left\_id == 0)

{

string s = "OK: 0";

zmq::message\_t message(s.size());

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

main\_socket.send(message);

}

else

{

left\_socket.send(message\_main);

zmq::message\_t answ;

left\_socket.recv(&answ);

main\_socket.send(answ);

}

}

else if (id\_proc > id)

{

if (right\_id == 0)

{

string s = "OK: 0";

zmq::message\_t message(s.size());

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

main\_socket.send(message);

}

else

{

right\_socket.send(message\_main);

zmq::message\_t answ;

right\_socket.recv(&answ);

main\_socket.send(answ);

}

}

}

else if (command == "kill")

{

int id\_proc; // id of node for killing

string id\_proc\_;

for (int i = 5; i < recieved\_message.size(); ++i)

{

if (recieved\_message[i] != ' ')

{

id\_proc\_ += recieved\_message[i];

}

else

{

break;

}

}

id\_proc = stoi(id\_proc\_);

if (id\_proc > id)

{

if (right\_id == 0)

{

send\_message("Error: there isn`t node with this id", main\_socket);

}

else

{

if (right\_id == id\_proc)

{

send\_message("Ok: " + to\_string(right\_id), main\_socket);

send\_message("DIE", right\_socket);

right\_socket.unbind(adr\_right);

adr\_right = "tcp://127.0.0.1:300";

right\_id = 0;

}

else

{

right\_socket.send(message\_main);

zmq::message\_t message;

right\_socket.recv(&message);

main\_socket.send(message);

}

}

}

else if (id\_proc < id)

{

if (left\_id == 0)

{

send\_message("Error: there isn`t node with this id", main\_socket);

}

else

{

if (left\_id == id\_proc)

{

send\_message("OK: " + to\_string(left\_id), main\_socket);

send\_message("DIE", left\_socket);

left\_socket.unbind(adr\_left);

adr\_left = "tcp://127.0.0.1:300";

left\_id = 0;

}

else

{

left\_socket.send(message\_main);

zmq::message\_t message;

left\_socket.recv(&message);

main\_socket.send(message);

}

}

}

}

else if (command == "DIE")

{

if (left\_id)

{

send\_message("DIE", left\_socket);

left\_socket.unbind(adr\_left);

adr\_left = "tcp://127.0.0.1:300";

left\_id = 0;

}

if (right\_id)

{

send\_message("DIE", right\_socket);

right\_socket.unbind(adr\_right);

adr\_right = "tcp://127.0.0.1:300";

right\_id = 0;

}

main\_socket.unbind(adr);

return 0;

}

}

}

**Makefile**

**all:**

**g++ -fsanitize=address main\_prog.cpp -lzmq -o main\_prog -w**

**g++ -fsanitize=address child\_node.cpp -lzmq -o child\_node -w**

**Демонстрация работы программы**

**peter@DESKTOP-V53N291:$ make**

**peter@DESKTOP-V53N291:$ ./main\_prog**

Commands:

create id

exec id text pattern

ping id

kill id

exit

Please, enter command

create 5

OK: 138

Please, enter command

create 6

OK: 145

Please, enter command

create 7

OK: 152

Please, enter command

exec 7 texttext t

OK:7:1;4;5;8;

Please, enter command

ping 5

OK: 1

Please, enter command

ping 4

OK: 0

Please, enter command

kill 7

Ok: 7

Please, enter command

exit

Tree was deleted

**Выводы**

В ходе выполнения лабораторной работы я изучил основы работы с очередями сообщений ZeroMQ и реализовал программу с использованием этой библиотеки. Для достижения отказоустойчивости я пробовал разные способы связи, больше всего подошёл ZMQ\_PAIR. Самым сложным в работе оказались удаление узла из сети и вставка узла между другими узлами. При таких операциях нужно было переподключать сокеты на вычислительных узлах.

Когда параллельных вычислений становится мало, на помощь приходят распределённые вычисления (распределение вычислений осуществляется уже не между потоками процессора, а между отдельными ЭВМ). Очереди сообщений используются для взаимодействия нескольких машин в одной большой сети. Опыт работы с ZeroMQ пригодится мне при настройке собственной системы распределённых вычислений.