**Московский Авиационный Институт**

**(Национальный Исследовательский Университет)**

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №5-7 по курсу**

**«Операционные системы»**

Группа: М80-203Б-22

Студент: Касумова Н.Р.

Преподаватель: Миронов Е.С.

Оценка:

Подпись:

Дата: 29.12.23

**Цель работы:**

Целью является приобретение практических навыков в:

* Управление серверами сообщений №5
* Применение отложенных вычислений №6
* Интеграция программных систем друг с другом №7

**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.

Вариант 17

Топология: все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Команда для узлов: нахождение индексов вхождения подстроки в строку

Команда проверки: ping id

**Код программы**

**my\_zmq.hpp**

#pragma once

#include <assert.h>

#include <errno.h>

#include <string.h>

#include <string>

#include <zmq.h>

const char\* NODE\_EXECUTABLE\_NAME = "calculation"; const char SENTINEL = '$'; const int PORT\_BASE = 8000; const int WAIT\_TIME = 1000;

enum actions\_t { fail = 0, success = 1, create = 2, destroy = 3, bind = 4, ping = 5, exec = 6, info = 7, back = 8

};

/\*объявляем константы и структуры\*/ struct node\_token\_t { actions\_t action; long long parent\_id, id;

};

namespace my\_zmq { void init\_pair\_socket(void\* & context, void\* & socket) { /\*инициализируем контекст и сокет для передачи сообщений в пару\*/ int rc;

context = zmq\_ctx\_new(); socket = zmq\_socket(context, ZMQ\_PAIR); rc = zmq\_setsockopt(socket, ZMQ\_RCVTIMEO, &WAIT\_TIME, sizeof(int));

assert(rc == 0); rc = zmq\_setsockopt(socket, ZMQ\_SNDTIMEO, &WAIT\_TIME, sizeof(int));

assert(rc == 0);

}

template<class T> void recieve\_msg(T & reply\_data, void\* socket) { /\*шаблон для получения сообщения из сокета\*/

int rc = 0; zmq\_msg\_t reply; zmq\_msg\_init(&reply); rc = zmq\_msg\_recv(&reply, socket, 0); assert(rc == sizeof(T)); reply\_data = \*(T\*)zmq\_msg\_data(&reply); rc = zmq\_msg\_close(&reply); assert(rc == 0);

}

template<class T> void send\_msg(T\* token, void\* socket) {/\*функция отправляет сообщение

(передаваемое как указатель на объект типа T) через указанный сокет\*/ int rc = 0; zmq\_msg\_t message; zmq\_msg\_init(&message); rc = zmq\_msg\_init\_size(&message, sizeof(T)); assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL); assert(rc == 0); rc = zmq\_msg\_send(&message, socket, 0); assert(rc == sizeof(T));

}

template<class T> bool send\_msg\_dontwait(T\* token, void\* socket) {

/\*функция отправляет сообщение через ZeroMQ сокет, но не ждет ответа на него\*/

int rc; zmq\_msg\_t message; zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T)); assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL); assert(rc == 0); rc = zmq\_msg\_send(&message, socket, ZMQ\_DONTWAIT); if (rc == -1) { zmq\_msg\_close(&message); return false;

}

assert(rc == sizeof(T)); return true;

}

/\* Returns true if T was successfully queued on the socket \*/

template<class T> bool recieve\_msg\_wait(T & reply\_data, void\* socket) { /\*принимает сообщение

типа T через сокет ZMQ\_PAIR и ждет, пока сообщение не будет получено\*/ int rc = 0; zmq\_msg\_t reply; zmq\_msg\_init(&reply); rc = zmq\_msg\_recv(&reply, socket, 0); if (rc == -1) { zmq\_msg\_close(&reply); return false;

}

assert(rc == sizeof(T)); reply\_data = \*(T\*)zmq\_msg\_data(&reply); rc = zmq\_msg\_close(&reply); assert(rc == 0); return true;

}

/\* Returns true if T was successfully queued on the socket \*/ template<class T>

bool send\_msg\_wait(T\* token, void\* socket) { /\*отправляет сообщение через zmq-сокет\*/

int rc; zmq\_msg\_t message; zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T)); assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL); assert(rc == 0); rc = zmq\_msg\_send(&message, socket, 0);

if (rc == -1) {

zmq\_msg\_close(&message);

return false;

}

assert(rc == sizeof(T)); return true;

}

/\*

* Returns true if socket successfully queued
* message and recieved reply

\*/

/\* send\_msg && receive\_msg \*/ template<class T> bool send\_recieve\_wait(T\* token\_send, T & token\_reply, void\* socket) { if (send\_msg\_wait(token\_send, socket)) { if (recieve\_msg\_wait(token\_reply, socket)) {

return true; } else { return false;

}

} else { return false;

}

}

}

**topology.hpp** #pragma once #include <iostream>

#include <list>

template<class T>

class topology\_t { private:

using list\_type = std::list< std::list<T> >; using iterator = typename std::list<T>::iterator; using list\_iterator = typename list\_type::iterator;

list\_type container; size\_t container\_size; public:

topology\_t() noexcept : container(), container\_size(0) {}

~topology\_t() {}

bool erase(const T & elem) { /\*функция удаляет элемент типа T из контейнера container\*/

for (list\_iterator it1 = container.begin(); it1 != container.end();

++it1) {

for (iterator it2 = it1->begin(); it2 != it1->end(); ++it2) {

if (\*it2 == elem) { if (it1->size() > 1) { it1->erase(it2);

} else {

container.erase(it1);

}

--container\_size;

return true;

}

}

}

return false;

}

long long find(const T & elem) {// в каком списке существует (или нет) элемент с идентификатором $id

long long ind = 0;

for (list\_iterator it1 = container.begin(); it1 != container.end();

++it1) {

for (iterator it2 = it1->begin(); it2 != it1->end(); ++it2) { if (\*it2 == elem) { return ind;

}

}

++ind;

}

return -1;

}

bool insert(const T & parent, const T & elem) {

/\*Функция insert принимает на вход два аргумента: parent и elem. Она ищет элемент parent в контейнере и добавляет элемент elem после него в ту же подсписок,

если parent был найден. Если parent не найден, функция возвращает false, иначе - true\*/

for (list\_iterator it1 = container.begin(); it1 != container.end();

++it1) {

for (iterator it2 = it1->begin(); it2 != it1->end(); ++it2) {

if (\*it2 == parent) { it1->insert(++it2, elem); ++container\_size;

return true;

}

}

}

return false;

}

void insert(const T & elem) { /\*добавляет элемент в структуру topology\_t\*/

std::list<T> new\_list; new\_list.push\_back(elem); ++container\_size; container.push\_back(new\_list);

}

size\_t size() { return container\_size;

}

template<class U> /\*оператор вывода в поток\*/

friend std::ostream & operator << (std::ostream & of, const

topology\_t<U> & top) {

for (auto it1 = top.container.begin(); it1 != top.container.end();

++it1) {

of << "{";

for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

of << \*it2 << " ";

}

of << "}" << std::endl;

}

return of;

}

};

**search.hpp** #pragma once

#include <string>

#include <vector>

const int SIZE = 1024;

std::vector<int> search(const std::string& str, const std::string& ptrn) { std::vector<int> ans; int n = str.size();

int m = ptrn.size();

for (int i = 0; i <= n - m; ++i) { bool found = true;

for (int j = 0; j < m; ++j) { if (str[i + j] != ptrn[j]) { found = false;

break;

}

}

if (found) {

ans.push\_back(i);

}

}

return ans;

}

# control.cpp

#include <unistd.h>

#include <vector>

#include "topology.hpp"

#include "myzmq.hpp"

using node\_id\_type = long long;

int main() { int rc;

topology\_t<node\_id\_type> control\_node; std::vector< std::pair<void\*, void\*> > childs; /\*для хранения контекстов и сокетов для дочерних процессов\*/

std::string s;

node\_id\_type id;

while (std::cin >> s >> id) { if (s == "create") { node\_id\_type parent\_id; std::cin >> parent\_id; if (parent\_id == -1) { void\* new\_context = NULL; void\* new\_socket = NULL;

my\_zmq::init\_pair\_socket(new\_context, new\_socket);

rc = zmq\_bind(new\_socket, ("tcp://\*:" +

std::to\_string(PORT\_BASE + id)).c\_str());

/\*привязка сокета к TCP-адресу с использованием порта,

который вычисляется как сумма PORT\_BASE и id.\*/

assert(rc == 0);

int fork\_id = fork(); if (fork\_id == 0) { rc = execl(NODE\_EXECUTABLE\_NAME,

NODE\_EXECUTABLE\_NAME, std::to\_string(id).c\_str(), NULL); assert(rc != -1); return 0;

} else {

bool ok = true;

node\_token\_t reply\_info({fail, id, id}); ok = my\_zmq::recieve\_msg\_wait(reply\_info, new\_socket);

node\_token\_t\* token = new node\_token\_t({ping, id, id});

node\_token\_t reply({fail, id, id});

ok = my\_zmq::send\_recieve\_wait(token, reply, new\_socket);

if (ok and reply.action == success) {

childs.push\_back(std::make\_pair(new\_context, new\_socket));

control\_node.insert(id); std::cout << "OK: " << reply\_info.id

<< std::endl;

} else {

rc = zmq\_close(new\_socket); assert(rc == 0);

rc = zmq\_ctx\_term(new\_context);

assert(rc == 0);

}

}

} else if (control\_node.find(parent\_id) == -1) {

std::cout << "Error: Not found" << std::endl;

} else {

if (control\_node.find(id) != -1) {

std::cout << "Error: Already exists" << std::endl;

} else {

int ind = control\_node.find(parent\_id);

node\_token\_t\* token = new

node\_token\_t({create, parent\_id, id});

node\_token\_t reply({fail, id, id});

if (my\_zmq::send\_recieve\_wait(token, reply, childs[ind].second) and reply.action == success) {

std::cout << "OK: " << reply.id << std::endl;

control\_node.insert(parent\_id, id);

} else {

std::cout << "Error: Parent is

unavailable" << std::endl;

}

}

}

} else if (s == "remove") { int ind = control\_node.find(id); if (ind != -1) { node\_token\_t\* token = new

node\_token\_t({destroy, id, id});

node\_token\_t reply({fail, id, id}); bool ok = my\_zmq::send\_recieve\_wait(token, reply, childs[ind].second);

if (reply.action == destroy and reply.parent\_id == id) {

rc = zmq\_close(childs[ind].second); assert(rc == 0);

rc = zmq\_ctx\_term(childs[ind].first);

assert(rc == 0);

std::vector< std::pair<void\*, void\*>

>::iterator it = childs.begin();

while (ind--) {

++it;

}

childs.erase(it);

} else if (reply.action == bind and reply.parent\_id

== id) {

rc = zmq\_close(childs[ind].second); assert(rc == 0);

rc = zmq\_ctx\_term(childs[ind].first);

assert(rc == 0);

my\_zmq::init\_pair\_socket(childs[ind].first, childs[ind].second);

rc = zmq\_bind(childs[ind].second, ("tcp://\*:"

+ std::to\_string(PORT\_BASE + reply.id)).c\_str());

assert(rc == 0);

}

if (ok) {

control\_node.erase(id); std::cout << "OK" << std::endl;

} else {

std::cout << "Error: Node is unavailable" <<

|  |  |  |
| --- | --- | --- |
| std::endl; |  |  |
|  |  | } |
|  |  | } else { |
|  |  | std::cout << "Error: Not found" << std::endl; |
|  |  | } |
|  |  | } else if (s == "ping") { |
|  |  | int ind = control\_node.find(id); |
|  |  | if (ind != -1) { |
| id, id}); |  | node\_token\_t\* token = new node\_token\_t({ping, |
|  |  | node\_token\_t reply({fail, id, id}); |
|  |  | if (my\_zmq::send\_recieve\_wait(token, reply, |

childs[ind].second) and reply.action == success) {

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

} else {

std::cout << "OK: 0" << std::endl;

}

} else {

std::cout << "Error: Not found" << std::endl;

}

} else if (s == "back") { int ind = control\_node.find(id); if (ind != -1) {

node\_token\_t\* token = new node\_token\_t({back, id, id});

node\_token\_t reply({fail, id, id}); if (my\_zmq::send\_recieve\_wait(token, reply, childs[ind].second)) {

if (reply.action == success) {

node\_token\_t\* token\_back = new node\_token\_t({back, id, id});

node\_token\_t reply\_back({fail, id, id});

std::vector<int> calculated; while

(my\_zmq::send\_recieve\_wait(token\_back, reply\_back, childs[ind].second) and reply\_back.action == success) {

calculated.push\_back(reply\_back.id);

}

if (calculated.empty()) {

std::cout << "OK: " << reply.id

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

} else {

std::cout << "OK: " << reply.id

<< " : ";

for (size\_t i = 0; i < calculated.size() - 1; ++i) {

std::cout << calculated[i]

<< ", ";

}

std::cout << calculated.back()

<< std::endl;

}

} else {

std::cout << "Error: No calculations to back" << std::endl;

}

} else {

std::cout << "Error: Node is unavailable" << std::endl;

}

} else {

std::cout << "Error: Not found" << std::endl;

}

} else if (s == "exec") { std::string pattern, text; std::cin >> pattern >> text; int ind = control\_node.find(id);

if (ind != -1) {

bool ok = true;

std::string text\_pattern = pattern + SENTINEL + text + SENTINEL;

for (size\_t i = 0; i < text\_pattern.size(); ++i) {

node\_token\_t\* token = new

node\_token\_t({exec, text\_pattern[i], id});

node\_token\_t reply({fail, id, id});

if (!my\_zmq::send\_recieve\_wait(token,

reply, childs[ind].second) or reply.action != success) { ok = false; break;

}

}

if (ok) {

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

} else {

std::cout << "Error: Node is unavailable" <<

std::endl;

}

} else {

std::cout << "Error: Not found" << std::endl;

}

}

}

for (size\_t i = 0; i < childs.size(); ++i) { rc = zmq\_close(childs[i].second); assert(rc == 0);

rc = zmq\_ctx\_term(childs[i].first);

assert(rc == 0);

}

}

# calculation.cpp

#include <list>

#include <pthread.h>

#include <queue>

#include <tuple>

#include <unistd.h>

#include "search.hpp"

#include "myzmq.hpp"

const std::string SENTINEL\_STR = "$";

long long node\_id; pthread\_mutex\_t mutex; pthread\_cond\_t cond;

std::queue< std::pair<std::string, std::string> > calc\_queue; std::queue< std::list<int> > done\_queue;

void\* thread\_func(void\*) {

while (1) {

pthread\_mutex\_lock(&mutex);

while (calc\_queue.empty()) {

pthread\_cond\_wait(&cond, &mutex);

}

std::pair<std::string, std::string> cur = calc\_queue.front(); calc\_queue.pop(); pthread\_mutex\_unlock(&mutex);

if (cur.first == SENTINEL\_STR and cur.second == SENTINEL\_STR) {

break;

} else {

std::vector<int> res = search(cur.first, cur.second);

std::list<int> res\_list; for (const int& elem : res) {

res\_list.push\_back(elem);

}

pthread\_mutex\_lock(&mutex); done\_queue.push(res\_list);

pthread\_mutex\_unlock(&mutex);

}

}

return NULL;

}

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

int rc; assert(argc == 2);

node\_id = std::stoll(std::string(argv[1]));

void\* node\_parent\_context = zmq\_ctx\_new();

void\* node\_parent\_socket = zmq\_socket(node\_parent\_context, ZMQ\_PAIR);

rc = zmq\_connect(node\_parent\_socket, ("tcp://localhost:" + std::to\_string(PORT\_BASE + node\_id)).c\_str());

assert(rc == 0);

long long child\_id = -1; void\* node\_context = NULL; void\* node\_socket = NULL;

pthread\_t calculation\_thread; rc = pthread\_mutex\_init(&mutex, NULL); assert(rc == 0);

rc = pthread\_cond\_init(&cond, NULL);

assert(rc == 0);

rc = pthread\_create(&calculation\_thread, NULL, thread\_func, NULL); assert(rc == 0);

std::string pattern, text;

bool flag\_sentinel = true;

node\_token\_t\* info\_token = new node\_token\_t({info, getpid(), getpid()}); my\_zmq::send\_msg\_dontwait(info\_token, node\_parent\_socket);

std::list<int> cur\_calculated;

bool has\_child = false; bool awake = true; bool calc = true; while (awake) { node\_token\_t token;

my\_zmq::recieve\_msg(token, node\_parent\_socket);

node\_token\_t\* reply = new node\_token\_t({fail, node\_id, node\_id});

if (token.action == back) { if (token.id == node\_id) {

if (calc) { if (done\_queue.empty()) { reply->action = exec;

} else {

cur\_calculated = done\_queue.front();

done\_queue.pop(); reply->action = success;

reply->id = getpid();

}

calc = false;

} else {

if (cur\_calculated.size() > 0) { reply->action = success; reply->id = cur\_calculated.front(); cur\_calculated.pop\_front();

} else {

reply->action = exec;

calc = true;

}

} } else {

node\_token\_t\* token\_down = new node\_token\_t(token); node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_recieve\_wait(token\_down,

reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == bind and token.parent\_id == node\_id) {

/\*

* Bind could be recieved when parent created node
* and this node should bind to parent's child

\*/

my\_zmq::init\_pair\_socket(node\_context, node\_socket); rc = zmq\_bind(node\_socket, ("tcp://\*:" +

std::to\_string(PORT\_BASE + token.id)).c\_str());

assert(rc == 0); has\_child = true;

child\_id = token.id;

node\_token\_t\* token\_ping = new node\_token\_t({ping, child\_id, child\_id});

node\_token\_t reply\_ping({fail, child\_id, child\_id});

if (my\_zmq::send\_recieve\_wait(token\_ping, reply\_ping, node\_socket) and reply\_ping.action == success) { reply->action = success;

}

} else if (token.action == create) { if (token.parent\_id == node\_id) {

if (has\_child) { rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context); assert(rc == 0);

}

my\_zmq::init\_pair\_socket(node\_context, node\_socket); rc = zmq\_bind(node\_socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str());

assert(rc == 0);

int fork\_id = fork();

if (fork\_id == 0) {

rc = execl(NODE\_EXECUTABLE\_NAME,

NODE\_EXECUTABLE\_NAME, std::to\_string(token.id).c\_str(), NULL);

assert(rc != -1); return 0; } else {

bool ok = true;

node\_token\_t reply\_info({fail, token.id, token.id});

ok = my\_zmq::recieve\_msg\_wait(reply\_info, node\_socket);

if (reply\_info.action != fail) { reply->id = reply\_info.id; reply->parent\_id = reply\_info.parent\_id;

}

if (has\_child) { node\_token\_t\* token\_bind = new

node\_token\_t({bind, token.id, child\_id});

node\_token\_t reply\_bind({fail, token.id, token.id});

ok =

my\_zmq::send\_recieve\_wait(token\_bind, reply\_bind, node\_socket);

ok = ok and (reply\_bind.action == success);

}

if (ok) {

/\* We should check if child has connected to this node \*/

node\_token\_t\* token\_ping = new

node\_token\_t({ping, token.id, token.id});

node\_token\_t reply\_ping({fail, token.id, token.id});

ok =

my\_zmq::send\_recieve\_wait(token\_ping, reply\_ping, node\_socket);

ok = ok and (reply\_ping.action == success); if (ok) {

reply->action = success; child\_id = token.id;

has\_child = true;

} else {

rc = zmq\_close(node\_socket); assert(rc == 0);

rc = zmq\_ctx\_term(node\_context); assert(rc == 0);

}

}

}

} else if (has\_child) {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token); reply\_down.action = fail; if (my\_zmq::send\_recieve\_wait(token\_down,

reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == ping) { if (token.id == node\_id) { reply->action = success; } else if (has\_child) {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token); reply\_down.action = fail; if (my\_zmq::send\_recieve\_wait(token\_down,

reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == destroy) { if (has\_child) { if (token.id == child\_id) {

bool ok = true; node\_token\_t\* token\_down = new

node\_token\_t({destroy, node\_id, child\_id});

node\_token\_t reply\_down({fail, child\_id, child\_id});

ok = my\_zmq::send\_recieve\_wait(token\_down, reply\_down, node\_socket);

/\* We should get special reply from child \*/

if (reply\_down.action == destroy and

reply\_down.parent\_id == child\_id) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

has\_child = false;

child\_id = -1;

} else if (reply\_down.action == bind and

reply\_down.parent\_id == node\_id) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context); assert(rc == 0);

my\_zmq::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" +

std::to\_string(PORT\_BASE + reply\_down.id)).c\_str());

assert(rc == 0); child\_id = reply\_down.id; node\_token\_t\* token\_ping = new

node\_token\_t({ping, child\_id, child\_id});

node\_token\_t reply\_ping({fail, child\_id, child\_id});

if (my\_zmq::send\_recieve\_wait(token\_ping,

reply\_ping, node\_socket) and reply\_ping.action == success) {

ok = true;

}

}

if (ok) {

reply->action = success;

}

} else if (token.id == node\_id) { rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context); assert(rc == 0); has\_child = false; reply->action = bind; reply->id = child\_id;

reply->parent\_id = token.parent\_id;

awake = false;

} else {

node\_token\_t\* token\_down = new node\_token\_t(token); node\_token\_t reply\_down(token); reply\_down.action = fail;

if (my\_zmq::send\_recieve\_wait(token\_down,

reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.id == node\_id) { /\* Special message to parent \*/ reply->action = destroy; reply->parent\_id = node\_id; reply->id = node\_id;

awake = false;

}

} else if (token.action == exec) { if (token.id == node\_id) { char c = token.parent\_id; if (c == SENTINEL) { if (flag\_sentinel) { std::swap(text, pattern);

} else {

if (calc\_queue.empty()) {

pthread\_cond\_signal(&cond);

}

calc\_queue.push({pattern, text});

text.clear();

pattern.clear();

}

flag\_sentinel = flag\_sentinel ^ 1;

} else {

text = text + c;

}

reply->action = success; } else if (has\_child) {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token); reply\_down.action = fail; if (my\_zmq::send\_recieve\_wait(token\_down,

reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

}

my\_zmq::send\_msg\_dontwait(reply, node\_parent\_socket);

}

}

**Вывод**

В ходе выполнения данной лабораторной работы я попробовала себя в управлении серверами сообщений, применении отложенных вычислений и интеграции программных систем друг с другом, а также познакомился с технологией передачи сообщений с помощью библиотеки ZeroMQ.