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: Боев Савелий Сергеевич

Группа: М8О-207Б-21

Вариант: 48

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

[**Репозиторий** 3](#_Toc130656169)

[**Постановка задачи** 3](#_Toc130656170)

[**Цель работы** 3](#_Toc130656171)

[**Задание** 3](#_Toc130656172)

[**Исходный код** 4](#_Toc130656173)

[**Демонстрация работы программы** 24](#_Toc130656174)

[**Выводы** 25](#_Toc130656175)

# **Репозиторий**

<https://github.com/IamNoobLEL/Labs-OSi>

# **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.

Вариант 48

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: локальный целочисленный словарь

Команда проверки: heartbeat time

# **Исходный код**

**Код программы calculation\_node.cpp**

#include "my\_zmq.h"

#include <iostream>

#include <map>

#include <unistd.h>

long long node\_id;

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

std::string key;

int val;

std::map<std::string, int> dict;

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 = nullptr;

void\* node\_socket = nullptr;

std::cout << "OK: " << getpid() << std::endl;

bool has\_child = false, awake = true, add = false;

while (awake) {

node\_token\_t token({ fail, 0, 0 });

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

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

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

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;

auto\* 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\_receive\_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(), nullptr);

assert(rc != -1);

return 0;

}

else {

bool ok = true;

if (has\_child) {

auto\* 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\_receive\_wait(token\_bind, reply\_bind, node\_socket);

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

}

if (ok) {

auto\* 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\_receive\_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) {

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

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_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) {

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

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_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;

auto\* 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\_receive\_wait(token\_down, reply\_down, node\_socket);

if (reply\_down.action == destroy) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

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

assert(rc == 0);

has\_child = false;

child\_id = -1;

}

else if (reply\_down.action == bind) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(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;

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

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

ok = my\_zmq::send\_receive\_wait(token\_ping, reply\_ping, node\_socket) and (reply\_ping.action == success);

}

if (ok) {

reply->action = success;

}

}

else if (token.id == node\_id) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

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

assert(rc == 0);

awake = false;

reply->action = bind;

reply->id = child\_id;

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

}

else {

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

node\_token\_t reply\_down = token;

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and (reply\_down.action == success)) {

\*reply = reply\_down;

}

}

}

else if (token.id == node\_id) {

reply->action = destroy;

awake = false;

}

}

else if (token.action == exec\_check) {

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

char c = token.parent\_id;

if (c == SENTINEL) {

if (dict.find(key) != dict.end()) {

std::cout << "OK:" << node\_id << ":" << dict[key] << std::endl;

}

else {

std::cout << "OK:" << node\_id << ":'" << key << "' not found" << std::endl;

}

reply->action = success;

key = "";

}

else {

key += c;

reply->action = success;

}

}

else if (has\_child) {

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

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

}

else if (token.action == exec\_add) {

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

char c = token.parent\_id;

if (c == SENTINEL) {

add = true;

reply->action = success;

}

else if (add) {

val = token.parent\_id;

dict[key] = val;

std::cout << "OK:" << node\_id << std::endl;

add = false;

key = "";

reply->action = success;

}

else {

key += c;

reply->action = success;

}

}

else if (has\_child) {

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

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

}

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

}

rc = zmq\_close(node\_parent\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(node\_parent\_context);

assert(rc == 0);

}

**Код программы control\_node.cpp**

#include <unistd.h>

#include <vector>

#include <thread>

#include <chrono>

#include <algorithm>

#include <zmq.hpp>

#include "my\_zmq.h"

#include "topology.h"

std::vector<long long> ping\_storage(0);

topology\_t<long long> control\_node;

std::vector<std::pair<void\*, void\*>> children;// [context, socket]

void pinger(int wait) {

while (true) {

for (size\_t i = 0; i < ping\_storage.size();i++) {

int value = ping\_storage[i];

int ind = control\_node.find(value);

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

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

if (ind != -1 and my\_zmq::send\_receive\_wait(token, reply, children[ind].second) and reply.action == success) {

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

continue;

}

else {

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

auto iterator = std::find(ping\_storage.begin(), ping\_storage.end(), value);

if (iterator != ping\_storage.end()) {

ping\_storage.erase(iterator);

}

break;

}

}

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

}

}

void delete\_control\_node(long long id) {

int ind = control\_node.find(id);

int rc;

bool ok;

if (ind != -1) {

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

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

ok = my\_zmq::send\_receive\_wait(token, reply, children[ind].second);

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

rc = zmq\_close(children[ind].second);

assert(rc == 0);

rc = zmq\_ctx\_destroy(children[ind].first);

assert(rc == 0);

auto it = children.begin();

while (ind--) {

++it;

}

children.erase(it);

}

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

rc = zmq\_close(children[ind].second);

assert(rc == 0);

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

assert(rc == 0);

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

rc = zmq\_bind(children[ind].second, ("tcp://\*:" + std::to\_string(PORT\_BASE + id)).c\_str());

assert(rc == 0);

}

if (ok) {

control\_node.erase(id);

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

}

else {

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

}

}

else {

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

}

}

int main() {

int rc;

bool ok;

std::string s;

std::thread new\_thread;

long long id;

std::cout << "Create id parent: create calculation node (use parent = -1 if parent is control node)" << std::endl;

std::cout << "Heartbeat milliseconds: ping calculation node with id $id" << std::endl;

std::cout << "Remove id: delete calculation node with id " << std::endl;

std::cout << "Exec id key val: add [key, val] add local dictionary" << std::endl;

std::cout << "Exec id key: check local dictionary" << std::endl;

while (std::cin >> s >> id) {

if (s == "create") {

long long parent\_id;

std::cin >> parent\_id;

int ind;

if (parent\_id == -1) {

void\* new\_context = nullptr;

void\* new\_socket = nullptr;

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

rc = zmq\_bind(new\_socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + 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(id).c\_str(), nullptr);

assert(rc != -1);

return 0;

}

else {

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

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

if (my\_zmq::send\_receive\_wait(token, reply, new\_socket) and reply.action == success) {//проверка создания нового сокета(выч ноды)

children.emplace\_back(std::make\_pair(new\_context, new\_socket));//добавляем в вектор новый сокет ребёнка тип н дентей у контрол ноды

control\_node.insert(id);//вставляем ид в топологию

}

else {

rc = zmq\_close(new\_socket);

assert(rc == 0);

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

assert(rc == 0);

}

}

ping\_storage.push\_back(id);

}

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

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

continue;

}

else {

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

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

continue;

}

auto\* token = new node\_token\_t({ create, parent\_id, id });

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

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

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

ping\_storage.push\_back(id);

}

else {

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

continue;

}

}

}

else if (s == "remove") {

delete\_control\_node(id);

}

else if (s == "heartbeat") {

if (ping\_storage.empty()) {

std::cout << "Error: there are no calculation nodes at all" << std::endl;

continue;

}

new\_thread = std::thread(pinger, id);

}

else if (s == "exec") {

ok = true;

std::string key;

char c;

int val = -1;

bool add = false;

std::cin >> key;

if ((c = getchar()) == ' ') {

add = true;

std::cin >> val;

}

int ind = control\_node.find(id);

if (ind == -1) {

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

continue;

}

key += SENTINEL;

if (add) {

for (auto i : key) {

auto\* token = new node\_token\_t({ exec\_add, i, id });

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

if (!my\_zmq::send\_receive\_wait(token, reply, children[ind].second) or reply.action != success) {

std::cout << "Fail: " << i << std::endl;

ok = false;

break;

}

}

auto\* token = new node\_token\_t({ exec\_add, val, id });

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

if (!my\_zmq::send\_receive\_wait(token, reply, children[ind].second) or reply.action != success) {

std::cout << "Fail: " << val << std::endl;

ok = false;

}

}

else {

for (auto i : key) {

auto\* token = new node\_token\_t({ exec\_check, i, id });

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

if (!my\_zmq::send\_receive\_wait(token, reply, children[ind].second) or reply.action != success) {

ok = false;

std::cout << "Fail: " << i << std::endl;

break;

}

}

}

if (!ok) {

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

}

}

}

new\_thread.detach();

return 0;

}

**Заголовочный файл topology.h**

#ifndef INC\_6\_8\_LAB\_\_TOPOLOGY\_H\_

#define INC\_6\_8\_LAB\_\_TOPOLOGY\_H\_

#include <iostream>

#include <list>

#include <map>

template<typename T>

class topology\_t {

public:

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;

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

~topology\_t() = default;

void insert(const T &elem) {

std::list<T> new\_list;

new\_list.emplace\_back(elem);

++container\_size;

container.emplace\_back(new\_list);

}

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

for (list\_iterator external\_it = container.begin(); external\_it != container.end(); ++external\_it) {

for (iterator internal\_it = external\_it->begin(); internal\_it != external\_it->end(); ++internal\_it) {

if (\*internal\_it == parent) {

external\_it->insert(++internal\_it, elem);

++container\_size;

return true;

}

}

}

return false;

}

bool erase(const T &elem) {

for (list\_iterator external\_it = container.begin(); external\_it != container.end(); ++external\_it) {

for (iterator internal\_it = external\_it->begin(); internal\_it != external\_it->end(); ++internal\_it) {

if (\*internal\_it == elem) {

if (external\_it->size() > 1) {

external\_it->erase(internal\_it);

} else {

container.erase(external\_it);

}

--container\_size;

return true;

}

}

}

return false;

}

size\_t size() {

return container\_size;

}

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

int ind = 0;

for (auto &external : container) {

for (auto &internal : external) {

if (internal == elem) {

return ind;

}

}

++ind;

}

return -1;

}

template<typename S>

friend std::ostream &operator<<(std::ostream &os, const topology\_t<S> &topology) {

for (auto &external : topology.container) {

os << "{";

for (auto &internal : external) {

os << internal << " ";

}

os << "}" << std::endl;

}

return os;

}

};

#endif//INC\_6\_8\_LAB\_\_TOPOLOGY\_H\_

**Заголовочный файл my\_zmq.h**

#ifndef INC\_6\_8\_LAB\_\_ZMQ\_H\_

#define INC\_6\_8\_LAB\_\_ZMQ\_H\_

#include <cassert>

#include <cerrno>

#include <cstring>

#include <string>

#include <zmq.hpp>

#include <random>

enum actions\_t {

fail = 0,

success = 1,

create = 2,

destroy = 3,

bind = 4,

ping = 5,

exec\_check = 6,

exec\_add = 7

};

const char \*NODE\_EXECUTABLE\_NAME = "calculation\_node";

const int PORT\_BASE = 8000;

const int WAIT\_TIME = 1000;

const char SENTINEL = '$';

struct node\_token\_t {

actions\_t action;

long long parent\_id, id;

};

namespace my\_zmq {

void init\_pair\_socket(void \*&context, void \*&socket) {

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<typename T>

void receive\_msg(T &reply\_data, void \*socket) {

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<typename T>

bool receive\_msg\_wait(T &reply\_data, void \*socket) {

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;

}

template<typename T>

void send\_msg(T \*token, void \*socket) {

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<typename T>

bool send\_msg\_no\_wait(T \*token, void \*socket) {

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<typename T>

bool send\_msg\_wait(T \*token, void \*socket) {

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;

}

/\* send\_msg && receive\_msg \*/

template<typename T>

bool send\_receive\_wait(T \*token\_send, T &token\_reply, void \*socket) {

if (send\_msg\_wait(token\_send, socket)) {

if (receive\_msg\_wait(token\_reply, socket)) {

return true;

}

}

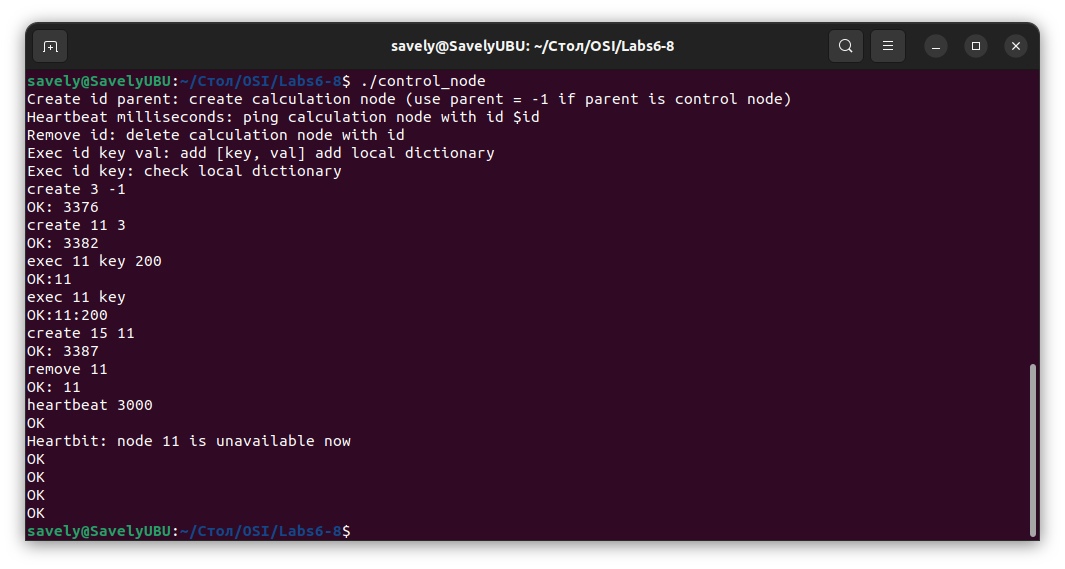
return false;

}

}// namespace my\_zmq

#endif//INC\_6\_8\_LAB\_\_ZMQ\_H\_

## **Демонстрация работы программы**



# **Выводы**

В ходе выполнения лабораторной работы я приобрел практические навыки в управлении серверами сообщений, применении отложенных вычислений и интеграции программных систем друг с другом, а также познакомился с технологией очереди сообщений, реализованной в библиотеке ZeroMQ. Эта технология позволяет удобным образом реализовывать межпроцессорное взаимодействие.