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: Гуликов К.А.

Группа: М8О-206Б-20

Вариант: 23

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

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

**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: heartbit time)
* **Удаление узла** (Формат команды remove id)

**Вариант №23:** топология — идеальное сбалансированное дерево, команда — поиск подстроки в строке, проверка доступности — heartbit.

**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**

**Balance\_Tree.h**

#ifndef BALANCE\_TREE\_H

#define BALANCE\_TREE\_H

#include <iostream>

template<typename T>

class Balance\_Tree {

public:

Balance\_Tree(T id, Balance\_Tree<T> \*parent) : data(id), left(nullptr), right(nullptr), parent(parent) {};

void insert(T id) {

if (nodes\_left <= nodes\_right) {

if (left != nullptr) {

left->insert(id);

} else {

left = new Balance\_Tree<T>(id, this);

}

++nodes\_left;

return;

}

if (right != nullptr) {

right->insert(id);

} else {

right = new Balance\_Tree<T>(id, this);

}

++nodes\_right;

}

std::pair<Balance\_Tree<T> \*, bool>

find\_insert() { // возвращает узел, куда произойдет вставка и true - если вставится в левый узел, false - в правый.

if (nodes\_left <= nodes\_right) {

if (left != nullptr) {

return left->find\_insert();

} else {

return std::make\_pair(this, true);

}

}

if (right != nullptr) {

return right->find\_insert();

}

return std::make\_pair(this, false);

}

bool empty() {

return (this->nodes\_left + this->nodes\_right) == 0;

}

template<typename S>

friend Balance\_Tree<S> \*find(S search, Balance\_Tree<S> \*t);

template<typename S>

friend std::ostream &operator<<(std::ostream &os, Balance\_Tree<S> \*t);

template<typename S>

friend void print\_tree(std::ostream &os, Balance\_Tree<S> \*t, size\_t x);

T get\_data() {

return data;

}

void set\_data(T id) {

data = id;

}

Balance\_Tree<T> \*get\_parent(){

return this->parent;

}

template<typename S>

friend void remove(S id, Balance\_Tree<S> \*t);

private:

T data;

Balance\_Tree<T> \*left, \*right, \*parent;

size\_t nodes\_left = 0, nodes\_right = 0;

};

template<typename T>

void print\_tree(std::ostream &os, Balance\_Tree<T> \*t, size\_t x) {

if (t == nullptr) {

return;

}

print\_tree(os, t->right, x + 1);

for (size\_t i = 0; i < x; ++i) {

os << '\t';

}

os << t->data << std::endl;

print\_tree(os, t->left, x + 1);

}

template<typename T>

std::ostream &operator<<(std::ostream &os, Balance\_Tree<T> \*t) {

print\_tree(os, t, 0);

return os;

}

template<typename T>

Balance\_Tree<T> \*find(T id, Balance\_Tree<T> \*t) { // поиск узла дерева. Производится обход в порядке КЛП

if (t->data == id) {

return t;

}

Balance\_Tree<T> \*tree = nullptr;

if (t->left != nullptr) {

tree = find(id, t->left);

}

if (t->right != nullptr and tree == nullptr) {

tree = find(id, t->right);

}

return tree;

}

template<typename T>

void remove(T id,

Balance\_Tree<T> \*t) { // удаляется узел и его потомки. Затем уменьшаем количество левых (правых) узлов у родителей

if (t == nullptr) {

return;

}

if (t->left != nullptr and t->left->data == id) {

int n = t->nodes\_left;

delete t->left;

t->left = nullptr;

auto \*tree = t;

t->nodes\_left = 0;

while (tree->parent != nullptr) {

if (tree->parent->right ==

tree) { // определяем, является текущий узел левым и правым потомком, затем уменьшаем количество левых (правых) узлов у родителя.

tree->parent->nodes\_right -= n;

} else {

tree->parent->nodes\_left -= n;

}

tree = tree->parent;

}

return;

}

if (t->right != nullptr and t->right->data == id) {

int n = t->nodes\_right;

delete t->right;

t->right = nullptr;

t->nodes\_right = 0;

auto \*tree = t;

while (tree->parent != nullptr) {

if (tree->parent->right ==

tree) { // определяем, является текущий узел левым и правым потомком, затем уменьшаем количество левых (правых) узлов у родителя.

tree->parent->nodes\_right -= n;

} else {

tree->parent->nodes\_left -= n;

}

tree = tree->parent;

}

return;

}

if (t->left != nullptr) {

remove(id, t->left);

}

if (t->right != nullptr) {

remove(id, t->right);

}

}

#endif //BALANCE\_TREE\_H

**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>

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\_

**control\_node.cpp**

#include <unistd.h>

#include <iostream>

#include <ctime>

#include "my\_zmq.h"

#include "Balance\_Tree.h"

using node\_id\_type = long long;

int main() {

bool ok;

Balance\_Tree<node\_id\_type> \*control\_node = new Balance\_Tree<node\_id\_type>(-1, nullptr);

std::string s;

node\_id\_type id;

std::cout << "\t\tUsage" << std::endl;

std::cout << "Create id: create calculation node" << std::endl;

std::cout << "heartbit time: set time to heartbit" << std::endl;

std::cout << "Remove id: delete calculation node with id $id and all id's children" << 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;

std::cout << "Print 0: print topology" << std::endl;

std::pair<void \*, void \*> child; // context, socket

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

if (s == "create") {

auto \*tree = find(id, control\_node);

if (tree != nullptr) {

std::cout << "Node with id " << id << " already exists" << std::endl;

continue;

}

if (control\_node->get\_data() == -1) {

zmq::init\_pair\_socket(child.first, child.second);

if (zmq\_bind(child.second, ("tcp://\*:" + std::to\_string(PORT\_BASE + id)).c\_str()) != 0) {

perror("ZMQ\_Bind");

exit(EXIT\_FAILURE);

}

int fork\_id = fork();

if (fork\_id == 0) {

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

perror("Execl");

exit(EXIT\_FAILURE);

} else if (fork\_id > 0) {

control\_node->set\_data(id);

} else {

perror("Fork");

exit(EXIT\_FAILURE);

}

} else {

auto inst = control\_node->find\_insert();

auto \*msg = new msg\_t({fail, inst.first->get\_data(), id});

(inst.second) ? msg->action = create\_left : msg->action = create\_right;

msg\_t reply = \*msg;

zmq::send\_receive\_wait(msg, reply, child.second);

if (reply.action == success) {

control\_node->insert(id);

} else {

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

}

}

} else if (s == "remove") {

auto \*tree = find(id, control\_node);

if (tree == nullptr) {

std::cout << "Error: Node with id " << id << " doesn't exists" << std::endl;

continue;

}

auto \*msg = new msg\_t({destroy, tree->get\_parent()->get\_data(), id});

msg\_t reply = \*msg;

zmq::send\_receive\_wait(msg, reply, child.second);

if (reply.action == success) {

std::cout << "Node " << id << " was deleted" << std::endl;

remove(id, control\_node);

} else {

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

}

} else if (s == "heartbit") {

clock\_t time = id;

for (int i = 0; i < 10; ++i) {

clock\_t start = clock();

auto \*msg = new msg\_t({ping, 1, control\_node->get\_data()});

msg\_t reply = \*msg;

zmq::send\_msg\_no\_wait(msg, child.second);

while (clock() - start < 4 \* time) {

zmq::receive\_msg(reply, child.second);

std::cout << "OK: " << reply.id << ": " << reply.parent\_id << std::endl;

}

std::cout << std::endl;

}

} else if (s == "exec") {

auto \*tree = find(id, control\_node);

if (tree == nullptr) {

std::cout << "Error: Node doesn't exists" << std::endl;

continue;

}

ok = true;

std::string key;

char c;

int val = -1;

bool add = false;

std::cin >> key;

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

add = true;

std::cin >> val;

}

key += SENTINEL;

for (auto i: key) {

auto \*token = new msg\_t({fail, i, id});

(add) ? token->action = exec\_add : token->action = exec\_check;

msg\_t reply({fail, id, id});

if (!zmq::send\_receive\_wait(token, reply, child.second) or reply.action != success) {

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

ok = false;

break;

}

}

if (add) {

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

msg\_t reply = \*msg;

if (!(zmq::send\_receive\_wait(msg, reply, child.second) and reply.action == success)) {

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

ok = false;

break;

}

}

if (!ok) {

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

}

} else if (s == "print") {

std::cout << control\_node;

}

}

std::cout << "Out tree:" << std::endl;

std::cout << control\_node;

auto \*msg = new msg\_t({destroy, -1, control\_node->get\_data()});

msg\_t reply = \*msg;

zmq::send\_receive\_wait(msg, reply, child.second);

zmq\_close(child.second);

zmq\_ctx\_destroy(child.first);

return 0;

}

**calculation\_node.cpp**

#include "my\_zmq.h"

#include <iostream>

#include <map>

#include <unistd.h>

long long node\_id;

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

if (argc != 2) {

exit(EXIT\_FAILURE);

}

std::string key;

int val;

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

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);

if (zmq\_connect(node\_parent\_socket, ("tcp://localhost:" + std::to\_string(PORT\_BASE + node\_id)).c\_str()) != 0) {

perror("ZMQ\_Connect");

exit(EXIT\_FAILURE);

}

long long left\_id = -1, right\_id = -1;

std::pair<void \*, void \*> left, right; // <context, socket>

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

bool has\_left = false, has\_right = false, awake = true, add = false;

while (awake) {

msg\_t token({fail, 0, 0});

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

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

if (token.action == create\_left) {

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

zmq::init\_pair\_socket(left.first, left.second);

if (zmq\_bind(left.second, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str()) != 0) {

perror("Bind");

exit(EXIT\_FAILURE);

}

int fork\_id = fork();

if (fork\_id == 0) {

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

perror("Execl");

exit(EXIT\_FAILURE);

} else if (fork\_id > 0) {

has\_left = true;

left\_id = token.id;

reply->action = success;

} else {

perror("Fork");

exit(EXIT\_FAILURE);

}

} else {

if (has\_left) {

auto \*token\_left = new msg\_t(token);

msg\_t reply\_left = \*reply;

zmq::send\_receive\_wait(token\_left, reply\_left, left.second);

if (reply\_left.action == success) {

\*reply = reply\_left;

}

}

if (has\_right and reply->action != success) {

auto \*token\_right = new msg\_t(token);

msg\_t reply\_right = \*reply;

zmq::send\_receive\_wait(token\_right, reply\_right, right.second);

\*reply = reply\_right;

}

}

} else if (token.action == create\_right) {

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

zmq::init\_pair\_socket(right.first, right.second);

if (zmq\_bind(right.second, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str()) != 0) {

perror("Bind");

exit(EXIT\_FAILURE);

}

int fork\_id = fork();

if (fork\_id == 0) {

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

perror("Execl");

exit(EXIT\_FAILURE);

} else if (fork\_id > 0) {

has\_right = true;

right\_id = token.id;

reply->action = success;

} else {

perror("Fork");

exit(EXIT\_FAILURE);

}

} else {

if (has\_left) {

auto \*token\_left = new msg\_t(token);

msg\_t reply\_left = \*reply;

zmq::send\_receive\_wait(token\_left, reply\_left, left.second);

if (reply\_left.action == success) {

\*reply = reply\_left;

}

}

if (has\_right and reply->action != success) {

auto \*token\_right = new msg\_t(token);

msg\_t reply\_right = \*reply;

zmq::send\_receive\_wait(token\_right, reply\_right, right.second);

\*reply = reply\_right;

}

}

} else if (token.action == ping) {

msg\_t ping\_left = \*reply, ping\_right = \*reply;

ping\_left.action = ping\_right.action = fail;

reply->action = success;

reply->parent\_id = 1;

if (has\_left) {

auto \*msg\_left = new msg\_t({ping, -1, left\_id});

zmq::send\_receive\_wait(msg\_left, ping\_left, left.second);

if (ping\_left.action != success) {

ping\_left.action = success;

ping\_left.parent\_id = 0;

}

}

if (has\_right) {

auto \*msg\_right = new msg\_t({ping, -1, right\_id});

zmq::send\_receive\_wait(msg\_right, ping\_right, right.second);

if (ping\_right.action != success) {

ping\_right.action = success;

ping\_right.parent\_id = 0;

}

}

if (ping\_right.action == success) {

zmq::send\_msg\_no\_wait(&ping\_right, node\_parent\_socket);

}

if (ping\_left.action == success) {

zmq::send\_msg\_no\_wait(&ping\_left, node\_parent\_socket);

}

} else if (token.action == destroy) {

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

auto \*destroy\_child = new msg\_t({destroy, -1, token.id});

msg\_t reply\_child = \*destroy\_child;

if (has\_left and left\_id == token.id) {

zmq::send\_receive\_wait(destroy\_child, reply\_child, left.second);

has\_left = false;

left\_id = -1;

zmq\_close(left.second);

zmq\_ctx\_destroy(left.first);

left.first = left.second = nullptr;

} else {

zmq::send\_receive\_wait(destroy\_child, reply\_child, right.second);

has\_right = false;

right\_id = -1;

zmq\_close(right.second);

zmq\_ctx\_destroy(right.first);

}

\*reply = reply\_child;

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

if (has\_left) {

auto \*destroy\_left = new msg\_t({destroy, left\_id, left\_id});

msg\_t reply\_left = \*destroy\_left;

zmq::send\_receive\_wait(destroy\_left, reply\_left, left.second);

has\_left = false;

left\_id = -1;

zmq\_close(left.second);

zmq\_ctx\_destroy(left.first);

left.first = left.second = nullptr;

}

if (has\_right) {

auto \*destroy\_right = new msg\_t({destroy, left\_id, left\_id});

msg\_t reply\_right = \*destroy\_right;

zmq::send\_receive\_wait(destroy\_right, reply\_right, right.second);

has\_right = false;

right\_id = -1;

zmq\_close(right.second);

zmq\_ctx\_destroy(right.first);

right.first = right.second = nullptr;

}

awake = false;

reply->action = success;

} else {

auto \*token\_left = new msg\_t(token);

msg\_t reply\_left = \*reply;

zmq::send\_receive\_wait(token\_left, reply\_left, left.second);

if (reply\_left.action == success) {

\*reply = reply\_left;

}

}

if (has\_right and reply->action != success) {

auto \*token\_right = new msg\_t(token);

msg\_t reply\_right = \*reply;

zmq::send\_receive\_wait(token\_right, reply\_right, right.second);

\*reply = reply\_right;

}

} 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 {

auto \*token\_left = new msg\_t(token);

msg\_t reply\_left = \*reply;

zmq::send\_receive\_wait(token\_left, reply\_left, left.second);

if (reply\_left.action == success) {

\*reply = reply\_left;

}

}

if (has\_right and reply->action != success) {

auto \*token\_right = new msg\_t(token);

msg\_t reply\_right = \*reply;

zmq::send\_receive\_wait(token\_right, reply\_right, right.second);

\*reply = reply\_right;

}

} 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 {

auto \*token\_left = new msg\_t(token);

msg\_t reply\_left = \*reply;

zmq::send\_receive\_wait(token\_left, reply\_left, left.second);

if (reply\_left.action == success) {

\*reply = reply\_left;

}

}

if (has\_right and reply->action != success) {

auto \*token\_right = new msg\_t(token);

msg\_t reply\_right = \*reply;

zmq::send\_receive\_wait(token\_right, reply\_right, right.second);

\*reply = reply\_right;

}

}

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

}

zmq\_close(node\_parent\_socket);

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

}

**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.