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: Примаченко Александр Александрович
Группа: М8О-208Б-20
Вариант: 23
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/SashaPaladin/OS/tree/master/6-8_lab

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.