Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу

«Операционные системы»

Студент: Ядров Артем Леонидович
Группа: М8О-208Б-20
Вариант: 9
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/Yadroff/OS/tree/master/2_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: ping id)
- **Удаление узла** (Формат команды remove id)

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

Общие сведения о программе

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

Общий метод и алгоритм решения

Используемые методы системные вызовы:

zmq_ctx_new()	Создает новый ØMQ контекст
void *zmq_socket (void *context, int type);	Создает ØMQ сокет
int zmq_setsockopt (void *socket, int option_name, const void *option_value, size_t option_len);	Устанавливает опции ØMQ сокета
int zmq_msg_init (zmq_msg_t *msg);	Инициализирует пустое ØMQ сообщение
int zmq_msg_recv (zmq_msg_t *msg, void *socket, int flags);	Получает часть сообщения из сокета
int zmq_msg_close (zmq_msg_t *msg);	Освобождает сообщение ØMQ
int zmq_msg_init_size (zmq_msg_t *msg, size_t size);	Инициализирует ØMQ сообщение определенного размера
int zmq_msg_init_data (zmq_msg_t *msg, void *data, size_t size, zmq_free_fn *ffn, void *hint);	Инициализирует сообщение ØMQ из предоставленного буфера.
int zmq_msg_send (zmq_msg_t *msg, void *socket, int flags);	Отправляет часть сообщения на сокет
int zmq_bind (void *socket, const char *endpoint);	Принимает входящие соединения на сокет
int zmq_close (void * socket);	Закрывает сокет ØMQ
int zmq_ctx_term (void * context);	Уничтожает контекст ØMQ
assert(expr)	Прекращает работу программы при ложном утверждении

Исходный код

topology.h

```
#ifndef INC_6_8_LAB__TOPOLOGY_H_
```

```
#define INC_6_8_LAB__TOPOLOGY_H_

#include <iostream>
#include <list>
#include <map>

using node_id_type = long long;

template<typename 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() : 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) {// in which list exists (or not) element with id $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_
```

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 <vector>
#include <zmq.hpp>

#include "my_zmq.h"
#include "topology.h"

using node_id_type = long long;
```

```
int main() {
  int rc;
  bool ok;
  topology_t<node_id_type> control_node;
  std::vector<std::pair<void *, void *>> children;// [context, socket]
  std::string s;
  node_id_type id;
  while (std::cin >> s >> id) {
        if (s == "create") {
          node_id_type 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) {
                    children.emplace_back(std::make_pair(new_context,
new_socket));
                    control_node.insert(id);
                } else {
                    rc = zmq_close(new_socket);
                    assert(rc == 0);
                    rc = zmq_ctx_term(new_context);
                    assert(rc == 0);
                }
            }
        } else if ((ind = 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;
                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);
            } else {
                std::cout << "Error: Parent is unavailable" << std::endl;
            }
        }
    } else if (s == "remove") {
        int ind = control_node.find(id);
```

```
            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_term(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" << 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) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        auto *token = new node_token_t({ping, id, id});
        node_token_t reply({fail, id, id});
        if (my_zmq::send_receive_wait(token, reply, children[ind].second) and
reply.action == success) {
            std::cout << "OK: 1" << std::endl;
        } else {
            std::cout << "OK: 0" << std::endl;
        }
    } 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;
        }
    }
}

for (auto [context, socket]: children) {
    rc = zmq_close(socket);
    assert(rc == 0);
    rc = zmq_ctx_term(context);
    assert(rc == 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) {
```

```
  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) {
          /*
                    * 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;
            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) {
                    /* We should check if child has connected to this node */
                    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 = true;
              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);
                /* 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;
                        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) {
                          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 {
                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) {
            /* Special message to parent */
            reply->action = destroy;
            reply->parent_id = node_id;
            reply->id = node_id;
            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);
    }
    if (has_child) {
          rc = zmq_close(node_socket);
          assert(rc == 0);
          rc = zmq_ctx_term(node_context);
          assert(rc == 0);
    }
```

```
  rc = zmq_close(node_parent_socket);
  assert(rc == 0);
  rc = zmq_ctx_term(node_parent_context);
  assert(rc == 0);
}
```

Сборка программы

Сборка осуществляется при помощи утилиты cmake.

[Temi4**@**localhost 6-8_lab**]**$ **cat** CMakeLists.txt

cmake_minimum_required**(**VERSION 3.20**)**

project**(**6-8_lab/src**)**

add_definitions**(** **-Wall** -Werror -Wextra**)**

add_executable**(**control src/control_node.**cpp)**

add_executable**(**calculation_node src/calculation_node.**cpp)**

target_link_libraries**(**control zmq**)**

target_link_libraries**(**calculation_node zmq**)[**Temi4**@**localhost 6-8_lab**]**$ cmake CMakeLists.txt

-- The **C** compiler identification is GNU 11.2.1

-- The CXX compiler identification is GNU 11.2.1

-- Detecting **C** compiler ABI info

-- Detecting **C** compiler ABI info - **done**

-- Check **for** working **C** compiler: /usr/bin/**cc** - skipped

-- Detecting **C** compile features

-- Detecting **C** compile features - **done**

-- Detecting CXX compiler ABI info

-- Detecting CXX compiler ABI info - **done**

-- Check **for** working CXX compiler: /usr/bin/**c**++ - skipped

-- Detecting CXX compile features

-- Detecting CXX compile features - **done**

-- Configuring **done**

-- Generating **done**

-- Build files have been written to: /home/Temi4/CLionProjects/OS/6-8_lab

[Temi4**@**localhost 6-8_lab**]**$ **make**

[25**%]** Building CXX object CMakeFiles/control.**dir**/src/control_node.**cpp**.o

[50**%]** Linking CXX executable control

[50**%]** Built target control

[75**%]** Building CXX object CMakeFiles/calculation_node.**dir**/src/calculation_node.**cpp**.o

[100**%]** Linking CXX executable calculation_node

[100**%]** Built target calculation_node

Демонстрация работы программы

[Temi4**@**localhost src**]**$ ./control
create 10 -1
OK: 26141
create 1 -1
OK: 26147
ping 10
OK: 1
ping 1
OK: 1
create 2 10
OK: 26155
ping 2
OK: 1
exec 10 abc
OK:10:'abc' not found
exec 10 abc 10
OK:10
exec 10 abc
OK:10:10
exec 2 abc
OK:2:'abc' not found
remove 10

OK
ping 2
OK: 0
ping 1
OK: 1

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.