

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

**Лабораторная работа №5-7 по курсу
«Операционные системы»**

ОЧЕРЕДИ СООБЩЕНИЙ

Студент: Филиппов Владимир Михайлович

Группа: М8О–210Б–22

Вариант: 29

Преподаватель: Соколов Андрей Алексеевич

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2023.

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (№5)
- Применение отложенных вычислений (№6)
- Интеграция программных систем друг с другом (№7)

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.

Вариант: бинарное дерево, heartbeat, сумма чисел.

Общие сведения о программе

Программа устроена в виде сервера и вычислительных узлов. Сервер отправляет какие-то сообщения узлам, они их обрабатывают и отвечают серверу. Для выполнения Л.Р. я использовал библиотеку ZMQ в ее Си варианте. Так как используемых функций очень много, приведу лишь некоторые из них:

- 1. zmq_ctx_new** – создает новый контекст.
- 2. zmq_socket** – инициализирует новый сокет.
- 3. zmq_setsockopt** – устанавливает опции сокета (вариант очереди, время блокировки и т. д.).
- 4. zmq_msg_recv** – принимает сообщение из сокета.
- 5. zmq_msg_send** – отправляет сообщение в сокет.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы библиотеки ZMQ.
2. Написать программы, реализующие сервер и клиент, а также заголовочный файл, который содержит структуру сообщения и функции управления очередью.
3. Написать программу, которая симулировала бы заданную топологию.
4. Протестировать.

Основные файлы программы

calc_node.cpp:

```
#include <unistd.h>
#include <iostream>
#include <ctime>
#include "my_zmq.h"
#include "binary_tree.h"
using node_id_type = int;
void tokenize(std::vector<int> &args, const std::string &input) {
    std::stringstream ss(input);
    std::string s;
    while (std::getline(ss, s, ' ')) {
        args.push_back(std::stoi(s));
    }
}
int main() {
    BinaryTree<node_id_type> *control_node = new BinaryTree<node_id_type>(-1);
    std::string s;
    node_id_type id;
    std::pair<void *, void *> child;
    long long child_id = -1;
    while (std::cin >> s >> id) {
        if (s == "create") {
            TNode<node_id_type> *node = control_node->find(id);
            if (node != nullptr) {
                std::cout << "Node with this id is already exist" << std::endl;
                continue;
            }
            if (control_node->get_root()->_data == -1) { // если это первый
                // вычислительный узел.
                my_zmq::init_pair_socket(child.first, child.second);
                child_id = id;
                if (zmq_bind(child.second, ("tcp://*:" + std::to_string(PORT_BASE +
                    id)).c_str()) != 0) {
                    perror("ZMQ_Bind");
                }
            }
        }
    }
}
```

```

        exit(EXIT_FAILURE);
    }
    int pid = fork();
    if (pid == 0) { //son
        execl(NODE_EXECUTABLE_NAME, NODE_EXECUTABLE_NAME,
std::to_string(id).c_str(), nullptr);
        perror("Execl");
        exit(EXIT_FAILURE);
    } else if (pid > 0) { // parent
        control_node->get_root()->set_data(id);
    } else { // error
        perror("Fork");
        exit(EXIT_FAILURE);
    }
} else { // если это не первый вычислительный узел
    auto *msg = new msg_t({create, 0, id});
    msg_t reply = *msg;
    my_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;
    }
}
}
control_node->print();
} else if (s == "heartbeat") {
    auto *msg = new msg_t({ping, 0, id});
    msg_t reply = *msg;
    const int wait = 1000 * 4 * id;
    int counter = 0;
    zmq_setsockopt(child.second, ZMQ_RCVTIMEO, &wait, sizeof(int));
    my_zmq::send_msg_wait(msg, child.second);
    while (true) {
        if (counter == 3) {
            break;
        }
        bool flag = my_zmq::recv_wait_for_time(reply, child.second);
        if (reply.action == success && flag) {
            std::cout << "ok: " << reply.id << std::endl;
            counter++;
        } else {
            std::cout << "unbelievable but root node is unavailable: " << child_id
<<std::endl;
            counter++;
        }
    }
}
zmq_setsockopt(child.second, ZMQ_RCVTIMEO, &WAIT_TIME, sizeof(int));
} else if (s == "exec") {
    auto *terminate_msg = new msg_t({exec_add, -1, id});
    std::vector<int> buf;
    std::string num;
    while(true) {

```

```

        std::cin >> num;
        if (num == "!") {
            break;
        }
        buf.push_back(std::stoi(num));
    }
    for (int num: buf){
        auto *msg_to_child = new msg_t({exec_add, num, id});
        my_zmq::send_msg_wait(msg_to_child, child.second);
    }
    my_zmq::send_msg_wait(terminate_msg, child.second);
} else if (s == "remove") {
    std::pair<TNode<int> *, bool> res = control_node->find_insert(id);
    if (!res.second) {
        std::cout << "Error: Node with id " << id << " doesn't exists" <<
std::endl;
        continue;
    }
    auto *msg = new msg_t({destroy, -1, id});
    if (res.first != nullptr) {
        auto *msg = new msg_t({destroy, res.first->_data, id});
    }
    msg_t reply = *msg;
    my_zmq::send_receive_wait(msg, reply, child.second);
    if (reply.action == success) {
        std::cout << "Node " << id << " was deleted" << std::endl;
        control_node->deleteSubtree(control_node->get_root(), id);
        control_node->print();
    } else {
        std::cout << "Parent is unavailable" << std::endl;
    }
} else {
    std::cout << "what";
    continue;
}
}
std::cout << "Out tree:" << std::endl;
control_node->print();
auto *msg = new msg_t({destroy, -1, control_node->get_root()->_data});
msg_t reply = *msg;
my_zmq::send_receive_wait(msg, reply, child.second);
zmq_close(child.second);
zmq_ctx_destroy(child.first);
return 0;
}

```

test2.c

```

#include "my_zmq.h"
#include <iostream>
#include <map>
#include <unistd.h>

```

```

long long node_id;
void make_node(std::pair<void *, void *> &context_socket, bool &flag, long long
&id, msg_t &token, msg_t *reply) {
    my_zmq::init_pair_socket(context_socket.first, context_socket.second);
    if (zmq_bind(context_socket.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) {
        flag = true;
        id = token.id;
        reply->action = success;
    } else {
        perror("Fork");
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char **argv) {
    int sum = 0;
    if (argc != 2) {
        exit(EXIT_FAILURE);
    }
    node_id = std::stoll(std::string(argv[1]));
    long long left_id = -1, right_id = -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);
    }
    std::pair<void *, void *> left, right; // <context, socket>
    std::cout << "OK: " << getpid() << std::endl;
    bool has_left = false, has_right = false, awake = true;

```

```

while (awake) {
    msg_t token({fail, 0, 0});
    my_zmq::receive_msg(token, node_parent_socket);
    // std::cout << "INFO ABOUT NODE: left_id = " << left_id << " right_id = "
    << right_id << " node_id = " << node_id << std::endl;
    auto *reply = new msg_t({fail, node_id, node_id});
    if (token.action == create) {
        if (node_id > token.id && has_left) {
            auto *token_left = new msg_t(token);
            msg_t reply_left = *reply;
            my_zmq::send_receive_wait(token_left, reply_left, left.second);
            if (reply_left.action == success) {
                *reply = reply_left;
            }
        } else if (node_id < token.id && has_right) {
            auto *token_right = new msg_t(token);
            msg_t reply_right = *reply;
            my_zmq::send_receive_wait(token_right, reply_right, right.second);
            if (reply_right.action == success) {
                *reply = reply_right;
            }
        }
        if (has_left == false && node_id > token.id) {
            make_node(left, has_left, token.id, token, reply);
            left_id = token.id;
        }
        if (has_right == false && node_id < token.id) {
            make_node(right, has_right, token.id, token, reply);
            right_id = token.id;
        }
        my_zmq::send_msg_no_wait(reply, node_parent_socket);
    } else if (token.action == ping) {
        msg_t ping_left = msg_t({fail, node_id, node_id}), ping_right =
msg_t({fail, node_id, node_id});
        auto *msg_to_parent = new msg_t({success, 0, node_id});
        const int wait = 1000 * token.id * 4;
        for (int i = 0; i < 3; i++) {
            my_zmq::send_msg_wait(msg_to_parent, node_parent_socket);
            sleep(token.id);
        }
        if (has_left) {
            int counter = 0;

```

```

    auto *token_left = new msg_t({ping, node_id, token.id});
    zmq_setsockopt(left.second, ZMQ_RCVTIMEO, &wait, sizeof(int));
    my_zmq::send_msg_wait(token_left, left.second);
    while (true) {
        if (counter == 3) {
            break;
        }
        bool flag_l = my_zmq::recv_wait_for_time(ping_left, left.second);
        if (ping_left.action == success && flag_l) {
            std::cout << "ok: " << ping_left.id << std::endl;
            counter++;
        } else {
            std::cout << "unbelievable but left node is unavailable: " << left_id
<<std::endl;
        }
    }
    zmq_setsockopt(left.second, ZMQ_RCVTIMEO, &WAIT_TIME,
sizeof(int));
}
if (has_right) {
    int counter = 0;
    auto *token_right = new msg_t({ping, node_id, token.id});
    zmq_setsockopt(right.second, ZMQ_RCVTIMEO, &wait, sizeof(int));
    my_zmq::send_msg_wait(token_right, right.second);
    while (true) {
        if (counter == 3) {
            break;
        }
        bool flag_r = my_zmq::recv_wait_for_time(ping_right, right.second);
        if (ping_right.action == success && flag_r) {
            std::cout << "ok: " << ping_right.id << std::endl;
            counter++;
        } else {
            std::cout << "unbelievable but right node is unavailable: " <<
right_id <<std::endl;
        }
    }
    zmq_setsockopt(right.second, ZMQ_RCVTIMEO, &WAIT_TIME,
sizeof(int));
}
} else if (token.action == exec_add) {
    if (node_id == token.id) {

```



```

    if (token.parent_id == -1) {
        std::cout << "Summary equal to " << sum << std::endl;
        sum = 0;
        continue;
    }
    sum += token.parent_id;
}
if (node_id > token.id && has_left) {
    auto *token_left = new msg_t({exec_add, token.parent_id, token.id});
    msg_t reply_left = *reply;
    my_zmq::send_msg_no_wait(token_left, left.second);
} else if (node_id < token.id && has_right) {
    auto *token_right = new msg_t({exec_add, token.parent_id, token.id});
    msg_t reply_right = *reply;
    my_zmq::send_msg_no_wait(token_right, right.second);
}
} else if (token.action == destroy) {
    if (node_id == token.parent_id) {
        msg_t reply_right = *reply;
        msg_t reply_left = *reply;
        if (token.id == left_id) {
            auto *token_left = new msg_t(token);
            my_zmq::send_receive_wait(token_left, reply_left, left.second);
            if (reply_left.action == success) {
                zmq_close(left.second);
                zmq_ctx_destroy(left.first);
                left.first = left.second = nullptr;
                left_id = -1;
                has_left = false;
                reply_left.action == success;
                my_zmq::send_msg_no_wait(&reply_left, node_parent_socket);
            }
        } else {
            auto *token_right = new msg_t(token);
            my_zmq::send_receive_wait(token_right, reply_right, right.second);
            if (reply_right.action == success) {
                zmq_close(right.second);
                zmq_ctx_destroy(right.first);
                right.first = right.second = nullptr;
                right_id = -1;
                has_right = false;
                reply_right.action == success;
            }
        }
    }
}

```

```

        my_zmq::send_msg_no_wait(&reply_right, node_parent_socket);
    }
}
wait(NULL);
continue;
}
if (token.id == node_id) {
    auto *msg_to_parent = new msg_t({success, 0, node_id});
    if (has_left) {
        auto *token_left = new msg_t({destroy_child, token.parent_id,
token.id});
        msg_t reply_left = msg_t({fail, 0, 0});
        my_zmq::send_receive_wait(token_left, reply_left, left.second);
        if (reply_left.action == success) {
            zmq_close(left.second);
            zmq_ctx_destroy(left.first);
            left.first = left.second = nullptr;
            left_id = -1;
            has_left = false;
        }
    }
    if (has_right) {
        auto *token_right = new msg_t({destroy_child, token.parent_id,
token.id});
        msg_t reply_right = *reply;
        my_zmq::send_receive_wait(token_right, reply_right, right.second);
        if (reply_right.action == success) {
            zmq_close(right.second);
            zmq_ctx_destroy(right.first);
            right.first = right.second = nullptr;
            right_id = -1;
            has_right = false;
        }
    }
    my_zmq::send_msg_no_wait(msg_to_parent, node_parent_socket);
    exit(3);
}
msg_t reply_right = *reply;
msg_t reply_left = *reply;
if (node_id > token.id && has_left) {
    auto *token_left = new msg_t(token);
    my_zmq::send_receive_wait(token_left, reply_left, left.second);

```

```

        if (reply_left.action == success) {
            my_zmq::send_msg_no_wait(&reply_left, node_parent_socket);
        }
    } else if (node_id < token.id && has_right) {
        auto *token_right = new msg_t(token);
        my_zmq::send_receive_wait(token_right, reply_right, right.second);
        if (reply_right.action == success) {
            my_zmq::send_msg_no_wait(&reply_right, node_parent_socket);
        }
    }
} else if (token.action == destroy_child) {
    msg_t reply_left = *reply;
    msg_t reply_right = *reply;
    if (has_left) {
        auto *token_left = new msg_t({destroy_child, node_id, token.id});
        my_zmq::send_receive_wait(token_left, reply_left, left.second);
    } else {
        reply_left.action == success;
    }
    if (has_right) {
        auto *token_right = new msg_t({destroy_child, node_id, token.id});
        my_zmq::send_receive_wait(token_right, reply_right, right.second);
    } else {
        reply_left.action == success;
    }
    if (!has_right && !has_left) {
        auto *reply_to_parent = new msg_t({success, token.parent_id,
node_id});
        my_zmq::send_msg_no_wait(reply_to_parent, node_parent_socket);
        zmq_close(node_parent_socket);
        zmq_ctx_destroy(node_parent_context);
        exit(3);
    }
    if (reply_right.action == success && reply_left.action == success) {
        auto *reply_to_parent = new msg_t({success, token.parent_id,
node_id});
        my_zmq::send_msg_no_wait(reply_to_parent, node_parent_socket);
        zmq_close(left.second);
        zmq_ctx_destroy(left.first);
        left.first = left.second = nullptr;
        left_id = -1;
        has_left = false;
    }
}

```

```

        zmq_close(right.second);
        zmq_ctx_destroy(right.first);
        right.first = right.second = nullptr;
        right_id = -1;
        has_right = false;
        zmq_close(node_parent_socket);
        zmq_ctx_destroy(node_parent_context);
        exit(3);
    }
}
}
}

```

binary_tree.h

```

#pragma once
#include <iostream>
template <typename T>
class TNode {
public:
    TNode(T data) : _data(data), _left(nullptr), _right(nullptr) {}
    T _data;
    TNode<T> *_left;
    TNode<T> *_right;
    void set_data(const T &data) {
        _data = data;
    }
};

template <typename T>
class BinaryTree {
private:
    static void DestroyNode(TNode<T>* node) {
        if (node) {
            DestroyNode(node->_left);
            DestroyNode(node->_right);
            delete node;
        }
    }
private:
    TNode<T> *_root;
public:
    BinaryTree(T key) {
        _root = new TNode(key);
    }

```

```

}
~BinaryTree() { DestroyNode(_root); }
void insert(T x) {
    TNode<T>** cur = &_amp;_root;
    while (*cur) {
        TNode<T>& node = **cur;
        if (x < node._data) {
            cur = &node._left;
        } else if (x > node._data) {
            cur = &node._right;
        } else {
            return;
        }
    }
    *cur = new TNode(x);
}

std::pair<TNode<T>*, bool> find_insert(T key) {
    TNode<T> *curr = _root;
    TNode<T> *parent = nullptr;
    std::pair<TNode<T>*, bool> res{nullptr, true};
    while (curr && curr->_data != key) {
        parent = curr;
        if (curr->_data > key)
            curr = curr->_left;
        else
            curr = curr->_right;
    }
    if (curr == nullptr) {
        res.first = nullptr;
        res.second = false;
        return res;
    }
    res.first = parent;
    return res;
}

void print() {
    print_tree(_root);
    std::cout << std::endl;
}

void print_tree(TNode<T> *curr) {
    if (curr) {

```

```

        print_tree(curr->_left);
        std::cout << curr->_data << " ";
        print_tree(curr->_right);
    }
}
TNode<T> *get_root() {
    return _root;
}
TNode<T> *find(T key) {
    TNode<T> *curr = _root;
    while (curr && curr->_data != key) {
        if (curr->_data > key)
            curr = curr->_left;
        else
            curr = curr->_right;
    }
    return curr;
}
TNode<T>* deleteSubtree(TNode<T> *root, T id) {
    if (root == nullptr) {
        return nullptr;
    }

    if (root->_data == id) {
        // Удаляем все поддерево, начиная с корневого узла root
        deleteEntireSubtree(root);
        return nullptr;
    }

    root->_left = deleteSubtree(root->_left, id);
    if (root->_left == nullptr) {
        root->_right = deleteSubtree(root->_right, id);
    }

    return root;
}
void deleteEntireSubtree(TNode<T>* root) {
    if (root == nullptr) {
        return;
    }
    deleteEntireSubtree(root->_left);
    deleteEntireSubtree(root->_right);
}

```

```

    std::cout << root << std::endl;
    delete root;
    root = nullptr;
}

void erase(T key) {
    TNode<T> * curr = _root;
    TNode<T> * parent = NULL;
    while (curr && curr->_data != key) {
        parent = curr;
        if (curr->_data > key) {
            curr = curr->_left;
        } else {
            curr = curr->_right;
        }
    }
    if (!curr) return;
    if (curr->_left == NULL) {
        if (parent && parent->_left == curr)
            parent->_left = curr->_right;
        if (parent && parent->_right == curr)
            parent->_right = curr->_right;
        delete curr;
        return;
    }
    if (curr->_right == NULL) {
        if (parent && parent->_left == curr)
            parent->_left = curr->_left;
        if (parent && parent->_right == curr)
            parent->_right = curr->_left;
        delete curr;
        return;
    }
    TNode<T> *replace = curr->_right;
    while (replace->_left)
        replace = replace->_left;
    int replace_value = replace->_data;
    this->erase(replace_value);
    curr->_data = replace_value;
}

};

```

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.h>
#include <random>
#include <iostream>
#include <sys/wait.h>
#include <sstream>
#include <algorithm>
enum actions_t {
    fail = 0,
    success = 1,
    create = 2,
    destroy = 4,
    ping = 5,
    exec_check = 6,
    exec_add = 7,
    destroy_child = 8
};
const char *NODE_EXECUTABLE_NAME = "calc.exe";
const int PORT_BASE = 2000;
const int WAIT_TIME = 5000;
const char SENTINEL = '$';

struct msg_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 recv_wait_for_time(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) {
        return false;
    }
    reply_data = *(T *)zmq_msg_data(&reply);
    return true;
}

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>
bool receive_msg_no_wait(T &reply_data, void *socket) {

```

```

    int rc = 0;
    zmq_msg_t reply;
    zmq_msg_init(&reply);
    rc = zmq_msg_recv(&reply, socket, ZMQ_DONTWAIT);
    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;
}

```

```

}
template<typename T>
bool test_rcv(T &reply_data, void *socket) {
    while (true) {
        int rc = 0;
        zmq_msg_t reply;
        zmq_msg_init(&reply);
        int time = clock();
        rc = zmq_msg_rcv(&reply, socket, 0);
        std::cout << rc << std::endl;
        if (rc == -1) {
            return false;
        }
        reply_data = *(T *)zmq_msg_data(&reply);
        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 zmq
#endif // INC_6_8_LAB__ZMQ_H_

```

Пример работы

```

[main][build]$ ./control.exe
create 1
create 1
1
OK: 22047
create 2
create 2
1 2
OK: 22053
create 4
create 4
1 2 4
OK: 22058
create -3
create -3
-3 1 2 4
OK: 22088
heartbit 1
heartbit 1
ok: 1
ok: 1
ok: 1
ok: -3
ok: -3
ok: -3
ok: 2
ok: 2
ok: 2
ok: 4
ok: 4
ok: 4
exec 1 2 3 4 5
exec 1
iamexec
!
Summary equal to 14

```

Вывод

Эта лабораторная, на мой субъективный взгляд, была самая сложная из всех. Кроме того, что нужно было читать кучу документацию на английском, так еще и само взаимодействие процессов не самое простое из-за топологии. Однако, в то же время эта л.р. была и самой интересной из всех. Писать свой “сервер” для обработки запросов было очень интересно.

Уверен, что знания работы с сокетами и очередями сообщений будут нужны во время моей будущей профессиональной карьеры, потому что IPC зачастую осуществляется как раз за счет них.