Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5-6-7 по курсу**

**«Операционные системы»**

**УПРАВЛЕНИЕ СЕРВЕРАМИ СООБЩЕНИЙ**

**ПРИМЕНЕНИЕ ОТЛОЖЕННЫХ ВЫЧИСЛЕНИЙ**

**ИНТЕГРАЦИЯ ПРОГРАММНЫХ СИСТЕМ ДРУГ С ДРУГОМ**

Студент: Железнов Илья Васильевич

Группа: М8О–210Б–22

Вариант: 46

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**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. Список основных поддерживаемых команд:

Создание нового вычислительного узла Формат команды:

create id [parent] id – целочисленный идентификатор нового вычислительного узла

parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода:

«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла

«Error: Already exists» - вычислительный узел с таким идентификатором уже существует

«Error: Parent not found» - нет такого родительского узла с таким идентификатором

«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться

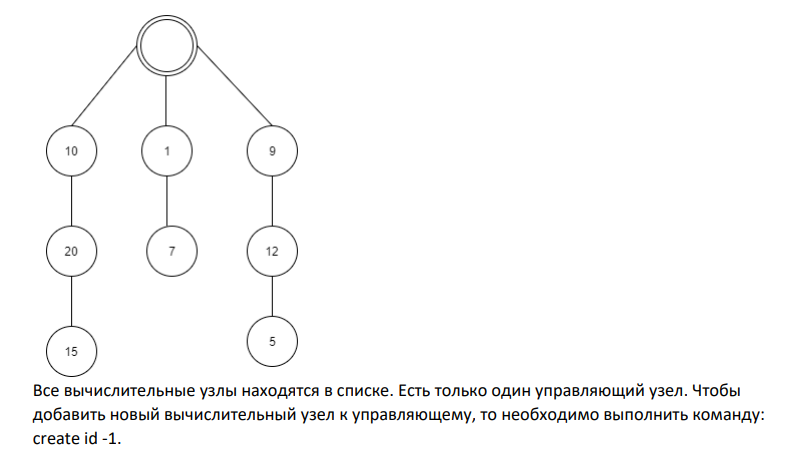
«Error: [Custom error]» - любая другая обрабатываемая ошибка

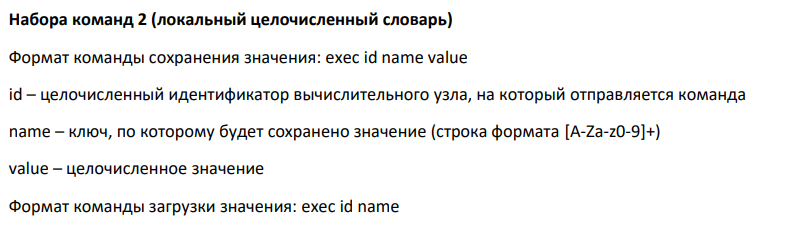
Пример:

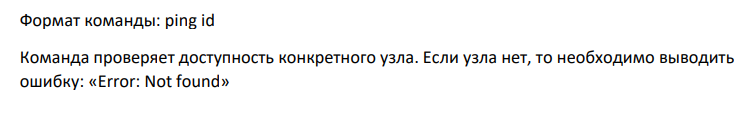
> create 10 5

Ok: 3128

Топология:

Команды для вычисления:

Тип проверки:



**Общие сведения о программе**

Программа компилируется из файла controlNode.cpp. Также используется заголовочные файлы: iostream, map, zmq.hpp, unitstd.h, vector, cassert . В программе используются следующие системные вызовы:

1. **void\* zmq\_ctx\_new()** – создает новый ZMQ контекст. Возвращает в случае успеха непрозрачный дескриптор вновь созданного контекста. В противном случае он должен вернуть NULL и установить для errno одно из значений.
2. **void\* zmq\_socket(void\* context, int type)** – создает ZMQ сокет. Void\* context – в каком контексте должен быть создан новый сокет; int type - указывает тип сокета, который определяет семантику связи через сокет. Сокет типа ZMQ\_PAIR может быть одновременно подключен только к одному узлу. Никакая маршрутизация или фильтрация сообщений не выполняется для сообщений, отправленных через сокет ZMQ\_PAIR. В случае успеха должен вернуть непрозрачный дескриптор вновь созданного сокета. В противном случае он должен вернуть NULL и установить для errno одно из значений.
3. **int zmq\_setsockopt(void\* socket, int option\_name, const void\* option\_value, size\_t option\_len)** – устанавливает опции ZMQ сокета. **int option\_name** – имя опции, которое нужно установить; **const void\* option\_value –** значение для опции, которую используем; **size\_t option\_len –** байтовый размер значения опции. **ZMQ\_SNDTIMEO**: Максимальное время до возврата операции отправки с EAGAIN. Устанавливает таймаут для операции отправки на сокете. Функция в случае успеха должна вернуть ноль. В противном случае он должен вернуть -1 и установить для errno одно из значений, определенных ниже.
4. **int zmq\_msg\_init(zmq\_msg\_t\* msg) –** инициализирует пустое ZMQ сообщение. Функция обычно возвращает 0.
5. **int zmq\_msg\_recv(zmq\_msg\_t\* msg, void\* socket, int flags)** – получает часть сообщения из сокета. В случае успеха функция должна вернуть количество байтов в сообщении. В противном случае он должен вернуть -1 и установить для errno одно из значений.
6. **int zmq\_msg\_close(zmq\_msg\_t\* msg)** — освобождает ZMQ сообщение. Функция должна вернуть ноль в случае успеха. В противном случае он должен вернуть -1 и установить для errno одно из значений.
7. **int zmq\_msg\_init\_size(zmq\_msg\_t\* msg, size\_t size)** - инициализирует ZMQ сообщение указанного размера. в случае успеха вернет ноль. В противном случае он должен вернуть -1 и установить для errno одно из значений.
8. **int zmq\_msg\_init\_data(zmq\_msg\_t\* msg, void\* data, size\_t size, zmq\_free\_fn \*ffn, void\* hint) —** инициал изирует ZMQ сообщение из указанного буффера. Возвращает обычно 0 в положительном результате работы.
9. **int zmq\_msg\_send(zmq\_msg\_t\* msg, void\* socket, int flag) —** отправляет часть сообщения на сокет. В случае успеха вернет количество байтов в сообщении (если количество байтов превышает MAX\_INT, функция вернет MAX\_INT). В противном случае он должен вернуть -1 и установить для errno одно из значений.
10. **int zmq\_bind(void\* socket, const char\* endpoint) –** принимает входящее соединение на сокет. **Endpoint –** адрес хоста. Вернет 0 при успехе.
11. **int zmq\_close(void\* socket) –** закрывает ZMQ сокет.
12. **int zmq\_ctx\_term(void\* context) –** уничтожает контекст ZMQ.
13. **assert(expr) –** прекращает работу программы при ложно утверждении.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Установить и изучить библиотеку zmq.hpp.
2. Написать 2 основных интерфейс файла topology.h и myZMQ.h для удобства реализации задачи.
3. Реализовать два файла для работы с сообщениями — это calulateNode.cpp и controlNode.cpp.
4. Написать правильный cmake файл, для правильно работы программы.
5. В controlNode реализовать простой интерфейс для упрощенный работы и тестировки программы.

**Основные файлы программы**

**myZMQ.h:**

#pragma once

#include <cassert>

#include <cerrno>

#include <cstring>

#include <string>

#include <zmq.hpp>

#include <random>

enum actions\_t {

    FAIL = 0,

    SUCCESS = 1,

    CREATE,

    DESTROY,

    BIND,

    PING,

    EXEC\_CHECK,

    EXEC\_ADD

};

const char\* NODE\_EXECUTABLE\_NAME = "calculationNode";

const int PORT\_BASE = 8000;

const int WAIT\_TIME = 1000;

const char SENTINEL = '$';

struct node\_token\_t {

    actions\_t action;

    long long parentId, id;

};

namespace myZMQ{

void initPairSocket(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 receiveMsg(T & replyData, void\* socket)

{

    int rc = 0;

    zmq\_msg\_t reply;

    zmq\_msg\_init(&reply);

    rc = zmq\_msg\_recv(&reply, socket, 0);

    assert(rc == sizeof(T));

    replyData = \*(T\* )zmq\_msg\_data(&reply);

    rc = zmq\_msg\_close(&reply);

    assert(rc == 0);

}

template <typename T>

bool receiveMsgWait(T & replyData, 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));

    replyData = \*(T\* )zmq\_msg\_data(&reply);

    rc = zmq\_msg\_close(&reply);

    assert(rc == 0);

    return true;

}

template <typename T>

void sendMsg(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 sendMsgNoWait(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 sendMsgWait(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;

}

template <typename T>

bool sendReceiveWait(T\* tokenSend, T & tokenReply, void\* socket)

{

    if (sendMsgWait(tokenSend, socket)) {

        if (receiveMsgWait(tokenReply, socket)) {

            return true;

        }

    }

    return false;

}

} // namespace myZMQ

**topology.h:**

#pragma once

#include <iostream>

#include <list>

#include <map>

template <typename T>

class Topology\_t

{

public:

    using list\_type = std::list<std::list<T>>;

    using iterator = typename std::list<T>::iterator;

    using list\_iterator = typename list\_type::iterator;

    Topology\_t() : \_container(), \_size(0) {};

    ~Topology\_t() = default;

    void insert(const T & elem)

    {

        std::list<T> newList;

        newList.emplace\_back(elem);

        ++\_size;

        \_container.emplace\_back(newList);

    }

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

    {

        for (list\_iterator externalIt = \_container.begin(); externalIt != \_container.end(); ++externalIt) {

            for (iterator internalIt = externalIt->begin(); internalIt != externalIt->end(); ++internalIt) {

                if (\*internalIt == parent) {

                    externalIt->insert(++internalIt, elem);

                    ++\_size;

                    return true;

                }

            }

    }

        return false;

    }

    bool erase(const T & elem)

    {

        for (list\_iterator externalIt = \_container.begin(); externalIt != \_container.end(); ++externalIt) {

            for (iterator internalIt = externalIt->begin(); internalIt != externalIt->end(); ++internalIt) {

                if (\*internalIt == elem) {

                    if (externalIt->size() > 1) {

                        externalIt->erase(internalIt);

                    } else {

                        \_container.erase(externalIt);

                    }

                    --\_size;

                    return true;

                }

            }

    }

        return false;

    }

    size\_t size()

    {

        return \_size;

    }

    int find(const T & elem)

    {

        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;

    }

// fields struct

    list\_type \_container;

    size\_t \_size;

};

**calculateNode.cpp:**

#include <iostream>

#include <map>

#include <unistd.h>

#include "myZMQ.h"

long long NodeID;

int main(int argc, char\* argv[])

{

    std::string key;

    int val;

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

    int rc;

    assert(argc == 2);

    NodeID = std::stoll(std::string(argv[1]));

    void\* nodeParentContext = zmq\_ctx\_new();

    void\* nodeParentSocket = zmq\_socket(nodeParentContext, ZMQ\_PAIR);

    rc = zmq\_connect(nodeParentSocket, ("tcp://localhost:" + std::to\_string(PORT\_BASE + NodeID)).c\_str());

    assert(rc == 0);

    long long childID = -1;

    void\* nodeContext = nullptr;

    void\* nodeSocket = nullptr;

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

    bool hasChild = false, awake = true, add = false;

    while (awake) {

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

        myZMQ::receiveMsg(token, nodeParentSocket);

        auto\* reply = new node\_token\_t({FAIL, NodeID, NodeID});

        /\*

            Привязка может быть получена, когда родительский узел создал узел

            и этот узел должен быть привязан к дочернему элементу родителя.

        \*/

        if (token.action == BIND && token.parentId == NodeID) {

            myZMQ::initPairSocket(nodeContext, nodeSocket);

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

            assert(rc == 0);

            hasChild = true;

            childID = token.id;

            auto\* tokenPing = new node\_token\_t({PING, childID, childID});

            node\_token\_t replyPing({FAIL, childID, childID});

            if (myZMQ::sendReceiveWait(tokenPing, replyPing, nodeSocket) && replyPing.action == SUCCESS) {

                reply->action = SUCCESS;

            }

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

            if (token.parentId == NodeID) {

                if (hasChild) {

                    rc  = zmq\_close(nodeSocket);

                    assert(rc == 0);

                    rc = zmq\_ctx\_term(nodeContext);

                    assert(rc == 0);

                }

                myZMQ::initPairSocket(nodeContext, nodeSocket);

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

                assert(rc == 0);

                int forkID = fork();

                if (forkID == 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 (hasChild) {

                        auto\* tokenBind = new node\_token\_t({BIND, token.id, childID});

                        node\_token\_t replyBind({FAIL, token.id, token.id});

                        OK = myZMQ::sendReceiveWait(tokenBind, replyBind, nodeSocket);

                        OK = OK && (replyBind.action == SUCCESS);

                    }

                    if (OK) {

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

                        node\_token\_t replyPing({FAIL, token.id, token.id});

                        OK = myZMQ::sendReceiveWait(tokenPing, replyPing, nodeSocket);

                        OK = OK && (replyPing.action == SUCCESS);

                        if (OK) {

                            reply->action = SUCCESS;

                            childID = token.id;

                            hasChild = true;

                        } else {

                            rc = zmq\_close(nodeSocket);

                            assert(rc == 0);

                            rc = zmq\_ctx\_term(nodeContext);

                            assert(rc == 0);

                        }

                    }

                }

            } else if (hasChild) {

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

                node\_token\_t replyDown(token);

                replyDown.action = FAIL;

                if (myZMQ::sendReceiveWait(tokenDown, replyDown, nodeSocket) && replyDown.action == SUCCESS) {

                    \*reply = replyDown;

                }

            }

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

            if (token.id == NodeID) {

                reply->action = SUCCESS;

            } else if (hasChild) {

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

                node\_token\_t replyDown(token);

                replyDown.action = FAIL;

                if (myZMQ::sendReceiveWait(tokenDown, replyDown, nodeSocket) && replyDown.action == SUCCESS) {

                    \*reply = replyDown;

                }

            }

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

            if (hasChild) {

                if (token.id == childID) {

                    bool OK;

                    auto\* tokenDown = new node\_token\_t({DESTROY, NodeID, childID});

                    node\_token\_t replyDown = {FAIL, childID, childID};

                    OK = myZMQ::sendReceiveWait(tokenDown, replyDown, nodeSocket);

                    if (replyDown.action == DESTROY) {

                        rc = zmq\_close(nodeSocket);

                        assert(rc == 0);

                        rc = zmq\_ctx\_destroy(nodeContext);

                        assert(rc == 0);

                        hasChild = false;

                        childID = -1;

                    } else if (replyDown.action == BIND) {

                        rc = zmq\_close(nodeSocket);

                        assert(rc == 0);

                        rc = zmq\_ctx\_destroy(nodeContext);

                        assert(rc == 0);

                        myZMQ::initPairSocket(nodeContext, nodeSocket);

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

                        assert(rc == 0);

                        childID = replyDown.id;

                        auto\* tokenPing = new node\_token\_t({PING, childID, childID});

                        node\_token\_t replyPing({FAIL, childID, childID});

                        OK = myZMQ::sendReceiveWait(tokenPing, replyPing, nodeSocket) && (replyPing.action == SUCCESS);

                    }

                    if (OK) {

                        reply->action = SUCCESS;

                    }

                } else if (token.id == NodeID) {

                    rc = zmq\_close(nodeSocket);

                    assert(rc == 0);

                    rc = zmq\_ctx\_destroy(nodeContext);

                    assert(rc == 0);

                    awake = false;

                    reply->action = BIND;

                    reply->id = childID;

                    reply->parentId = token.parentId;

                } else {

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

                    node\_token\_t replyDown = token;

                    replyDown.action = FAIL;

                    if (myZMQ::sendReceiveWait(tokenDown, replyDown, nodeSocket) && replyDown.action == SUCCESS) {

                        \*reply = replyDown;

                    }

                }

            } else if (token.id == NodeID) {

                reply->action = DESTROY;

                awake = false;

            }

        } else if (token.action == EXEC\_CHECK) {

            if (token.id == NodeID) {

                char c = token.parentId;

                if (c == SENTINEL) {

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

                        std::cout << "OK:" << NodeID << ":" << dictionary[key] << std::endl;

                    } else {

                        std::cout << "OK:" << NodeID << ":'" << key << "' not found" << std::endl;

                    }

                    reply->action = SUCCESS;

                    key = "";

                } else {

                    key += c;

                    reply->action = SUCCESS;

                }

            } else if (hasChild) {

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

                node\_token\_t replyDown(token);

                replyDown.action = FAIL;

                if (myZMQ::sendReceiveWait(tokenDown, replyDown, nodeSocket) && replyDown.action == SUCCESS) {

                    \*reply = replyDown;

                }

            }

        } else if (token.action == EXEC\_ADD) {

            if (token.id == NodeID) {

                char c = token.parentId;

                if (c == SENTINEL) {

                    add = true;

                    reply->action = SUCCESS;

                } else if (add) {

                    val = token.parentId;

                    dictionary[key] = val;

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

                    add = false;

                    key = "";

                    reply->action = SUCCESS;

                } else {

                    key += c;

                    reply->action = SUCCESS;

                }

            } else if (hasChild) {

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

                node\_token\_t replyDown(token);

                replyDown.action = FAIL;

                if (myZMQ::sendReceiveWait(tokenDown, replyDown, nodeSocket) && replyDown.action == SUCCESS) {

                    \*reply = replyDown;

                }

            }

        }

        myZMQ::sendMsgNoWait(reply, nodeParentSocket);

    }

    rc = zmq\_close(nodeParentSocket);

    assert(rc == 0);

    rc = zmq\_ctx\_destroy(nodeParentContext);

    assert(rc == 0);

    return 0;

}

**controlNode.cpp:**

#include <unistd.h>

#include <vector>

#include <zmq.hpp>

#include "myZMQ.h"

#include "topology.h"

using node\_id\_type = long long;

void deleteControlNode(node\_id\_type id, Topology\_t<node\_id\_type> & controlNode, std::vector<std::pair<void\* , void\* >> children)

{

    int ind = controlNode.find(id);

    int rc;

    bool OK;

    if (ind != -1) {

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

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

        OK = myZMQ::sendReceiveWait(token, reply, children[ind].second);

        if (reply.action == DESTROY && reply.parentId == id) {

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

            assert(rc == 0);

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

            assert(rc == 0);

            auto it = children.begin();

            while (ind--) {

                ++it;

            }

            children.erase(it);

        } else if (reply.action == BIND && reply.parentId == id) {

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

            assert(rc == 0);

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

            assert(rc == 0);

            myZMQ::initPairSocket(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) {

            controlNode.erase(id);

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

        } else {

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

        }

    } else {

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

    }

}

void help()

{

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

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

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

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

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

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

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

}

int main()

{

    int rc;

    bool OK;

    Topology\_t<node\_id\_type> controlNode;

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

    std::string str;

    node\_id\_type id;

    help();

    while (std::cin >> str >> id) {

        if ((str == "create") || (str == "Create")) {

            node\_id\_type parentID;

            std::cin >> parentID;

            int ind;

            if (parentID == -1) {

                void\* newContext = nullptr;

                void\* newSocket = nullptr;

                myZMQ::initPairSocket(newContext, newSocket);

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

                assert(rc == 0);

                int forkID = fork();

                if (forkID == 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 (myZMQ::sendReceiveWait(token, reply, newSocket) && (reply.action == SUCCESS)) {

                        children.emplace\_back(std::make\_pair(newContext, newSocket));

                        controlNode.insert(id);

                    } else {

                        rc = zmq\_close(newSocket);

                        assert(rc == 0);

                        rc = zmq\_ctx\_destroy(newContext);

                        assert(rc == 0);

                    }

                }

            } else if ((ind = controlNode.find(parentID)) == -1) {

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

                continue;

            } else {

                if (controlNode.find(id) != -1) {

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

                    continue;

                }

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

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

                if (myZMQ::sendReceiveWait(token, reply, children[ind].second) && reply.action == SUCCESS) {

                    controlNode.insert(parentID, id);

                } else {

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

                }

            }

        } else if (str == "remove" || str == "Remove") {

            deleteControlNode(id, controlNode, children);

        } else if (str == "ping" || str == "Ping") {

            int ind = controlNode.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 (myZMQ::sendReceiveWait(token, reply, children[ind].second) && (reply.action == SUCCESS)) {

                std::cout << "OK: 1" << std::endl;

            } else {

                std::cout << "OK: 0" << std::endl;

            }

        } else if (str == "exec" || str == "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 = controlNode.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 (!myZMQ::sendReceiveWait(token, reply, children[ind].second) || 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 (!myZMQ::sendReceiveWait(token, reply, children[ind].second) || 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, id, id});

                    if (!myZMQ::sendReceiveWait(token, reply, children[ind].second) || (reply.action != SUCCESS)) {

                        OK = false;

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

                        break;

                    }

                }

            }

            if (!OK) {

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

            }

        } else if (str == "print" || str == "Print") {

            std::cout << controlNode;

        }

    }

    std::cout << controlNode;

    for (auto i : controlNode.\_container) {

        for (size\_t s = i.size(); s > 1; --s) {

            node\_id\_type last = i.back();

            deleteControlNode(last, controlNode, children);

            i.pop\_back();

        }

    }

    std::vector<node\_id\_type> afterRoot;

    for (auto i : controlNode.\_container) {

        afterRoot.push\_back(i.back());

    }

    for (auto i : afterRoot) {

        deleteControlNode(i, controlNode, children);

    }

    return 0;

}

**Пример работы**

**Usage**

**Create id parent: create calculation node (use parent = -1 if parent is control node)**

**Ping id: ping calculation node with id $id**

**Remove id: delete calculation node with id $id**

**Exec id key val: add [key, val] add local dictionary**

**Exec id key: check local dictionary**

**Print 0: print topology**

**create 10 -1**

**OK: 943**

**create 1 -1**

**OK: 948**

**ping 10**

**OK: 1**

**ping 1**

**OK: 1**

**create 2 10**

**OK: 953**

**ping 2**

**OK: 1**

**exec 10 abc**

**OK:10:'abc' not found**

**exec 10 abc 10**

**OK:10**

**exec 10 abc**

**OK:10:10**

**remove 10**

**OK: 10**

**print 0**

**{2 }**

**{1 }**

**ping 2**

**OK: 0**

**ping 1**

**OK: 1**

**Вывод**

В завершении своей работы, хочу заметить, что изучение сокетов и очереди сообщений очень важные и интересные темы для программиста, который занимается бэкенд-разработкой. Знания работы сокетов, позволяет понимать базовые принципы работы многих технологий общения сервера и клиента. Изучив ZMQ, как технологию для очереди сообщений, я понял насколько она удобная и простая в понимании. Работа очередей сообщений позволяет двусвязно передавать сообщения в обе стороны и производить над ними операции.