

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»**

Студент: Кудрявов  
Группа: М8О-208Б-22  
Вариант: 37  
Преподаватель: Миронов Евгений Сергеевич  
Оценка: \_\_\_\_\_  
Дата: \_\_\_\_\_  
Подпись: \_\_\_\_\_

Москва, 2023

## **Содержание**

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

## Репозиторий

<https://github.com/Marsha2022/OS.git>

### Постановка задачи

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.

### Общие сведения о программе

Программа написана для операционной системы Linux. Для реализации очереди сообщений была использована библиотека ZeroMQ. Были написаны две программы – для управляющего узла и для вычислительного узла.

### Общий метод и алгоритм решения

Топология 1:

Все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: `create id -1`.

Набор команд 2 (локальный целочисленный словарь):

Формат команды сохранения значения: `exec id name value`.

`id` – целочисленный идентификатор вычислительного узла, на который отправляется команда.

name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+).

value – целочисленное значение

Формат команды загрузки значения: exes id name

Команда проверки 1:

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

В ходе выполнения лабораторной работы я использовал библиотеку ZeroMQ и следующие команды:

- bind() - устанавливает "сокеты" на адрес, а затем принимает входящие соединения на этом адресе.
- unbind() - отвязывает сокет от адреса.
- connect() - создание соединения между сокетом и адресом.
- disconnect() - разрывает соединение между сокетом и адресом.
- send() - отправка сообщений.
- recv() - получение сообщений.

### Исходный код

===== socket.hpp =====

```
#pragma once
```

```
#include <iostream>
```

```
#include <zmq.hpp>
```

```
#include <string>
```

```
#include <unistd.h>
```

```
#include <sstream>
```

```
#include <set>
```

```
#include <unordered_map>
```

```
#include <optional>
```

```
constexpr int MAIN_PORT = 4040;
```

```
void sendMessage(zmq::socket_t& socket, const std::string& msg);
```

```
std::string receiveMessage(zmq::socket_t& socket);
```

```
void connect(zmq::socket_t& socket, int id);
```

```
void disconnect(zmq::socket_t& socket, int id);
```

```
void bind(zmq::socket_t& socket, int id);
```

```
void unbind(zmq::socket_t& socket, int id);
```

```
=====topology.hpp =====
```

```
#pragma once
```

```
#include <iostream>
```

```
#include <list>
```

```
#include <set>
```

```
class Topology {
```

```
private:
```

```
    using listType = std::list<std::list<int>>;
```

```
    listType list;
```

```
public:
```

```
    Topology() : list() {}
```

```
    void insert(int id, int parentId);
```

```
    int find(int id);
```

```
    void erase(int id);
```

```
    int getFirstId(int listId);
```

```
    std::set<int> getSetOfChilds(int id);
```

```
};
```

===== calculation\_node.cpp =====

```
#include "socket.hpp"
```

```
#include <string>
```

```
int main(int argc, char* argv[]) {
```

```
    if (argc != 2 && argc != 3) {
```

```
        throw std::runtime_error("Wrong args for counting node");
```

```
    }
```

```
    int curId = atoi(argv[1]);
```

```
    int childId = -1;
```

```
    if (argc == 3) {
```

```
        childId = atoi(argv[2]);
```

```
    }
```

```
    std::string adr;
```

```
    std::string path = getenv("PATH_TO_CLIENT");
```

```
    std::unordered_map<std::string, int> dictionary;
```

```
    zmq::context_t context;
```

```
    zmq::socket_t parentSocket(context, ZMQ_REP);
```

```
    connect(parentSocket, curId);
```

```
    zmq::socket_t childSocket(context, ZMQ_REQ);
```

```
    if (childId != -1) {
```

```
        bind(childSocket, childId);
```

```
    }
```

```
    childSocket.set(zmq::sockopt::sndtimeo, 5000);
```

```
    std::string message;
```

```
    while (true) {
```

```
        message = receiveMessage(parentSocket);
```

```

std::istringstream request(message);
int destId;
request >> destId;

std::string command;
request >> command;

if (destId == curId) {

    if (command == "pid") {
        sendMessage(parentSocket, "OK: " +
std::to_string(getpid()));
    } else if (command == "create") {

        int new_childId;
        request >> new_childId;
        if (childId != -1) {
            unbind(childSocket, childId);
        }
        bind(childSocket, new_childId);
        pid_t pid = fork();
        if (pid < 0) {
            std::cout << "Can't create new process" <<
std::endl;

            return -1;
        }
        if (pid == 0) {
            if (execl(path.c_str(), path.c_str(),
std::to_string(new_childId).c_str(), std::to_string(childId).c_str(),
NULL)==-1) {

                std::cout << "Error with execl" << std::endl;
                perror("Error with execl");
                exit(EXIT_FAILURE);
            }

```

```

        std::cout << "Can't execute new process" <<
std::endl;

        return -2;
    }
    sendMessage(childSocket, std::to_string(new_childId) +
" pid");

    childId = new_childId;
    sendMessage(parentSocket,
receiveMessage(childSocket));

    } else if (command == "check") {
        std::string key;
        request >> key;
        if (dictionary.find(key) != dictionary.end()) {
            sendMessage(parentSocket, "OK: " +
std::to_string(curId) + ": " + std::to_string(dictionary[key]));
        } else {
            sendMessage(parentSocket, "OK: " +
std::to_string(curId) + ": '" + key + "' not found");
        }
    } else if (command == "add") {
        std::string key;
        int value;
        request >> key >> value;
        dictionary[key] = value;
        sendMessage(parentSocket, "OK: " +
std::to_string(curId));
    } else if (command == "ping") {
        std::string reply;
        if (childId != -1) {
            sendMessage(childSocket, std::to_string(childId) +
" ping");

            std::string msg = receiveMessage(childSocket);
            reply += " " + msg;

```



```

        }
        sendMessage(parentSocket, std::to_string(curId) +
reply);
    } else if (command == "kill") {
        if (childId != -1) {
            sendMessage(childSocket, std::to_string(childId) +
" kill");

            std::string msg = receiveMessage(childSocket);
            if (msg == "OK") {
                sendMessage(parentSocket, "OK");
            }
            unbind(childSocket, childId);
            disconnect(parentSocket, curId);
            break;
        }
        sendMessage(parentSocket, "OK");
        disconnect(parentSocket, curId);
        break;
    }
}
else if (childId != -1) {
    sendMessage(childSocket, message);
    sendMessage(parentSocket, receiveMessage(childSocket));
    if (childId == destId && command == "kill") {
        childId = -1;
    }
} else {
    sendMessage(parentSocket, "Error: Node is unavailable");
}
}
}

```

===== control\_node.cpp =====

```
#include "topology.hpp"
```

```

#include "socket.hpp"

int main() {

    //export PATH_TO_CLIENT="/home/hacker/prog/my_os_labs/build/lab5-
7/server"

    std::string path = getenv("PATH_TO_CLIENT");
    Topology list;
    std::vector<zmq::socket_t> branches;
    std::set<int> not_available_nodes;
    zmq::context_t context;

    std::string command;

    while (true) {
        std::cin >> command;
        if (command == "create") {
            int nodeId, parentId;
            std::cin >> nodeId >> parentId;
            if (list.find(nodeId) != -1) {
                std::cout << "Error: Already exists" << std::endl;
            } else if (parentId == -1) {
                pid_t pid = fork();
                if (pid < 0) {
                    std::cout << "Can't create new process" <<
std::endl;

                    return -1;
                } else if (pid == 0) {
                    //execl(path.c_str(), path.c_str(),
std::to_string(nodeId).c_str(), NULL);
                    if (execl(path.c_str(), path.c_str(),
std::to_string(nodeId).c_str(), NULL)==-1) {
                        std::cout << "Error with execl" << std::endl;

```

```

        perror("Error with execl");
        exit(EXIT_FAILURE);
    }
    std::cout << "Can't execute new process" <<
std::endl;

    return -2;
}
branches.emplace_back(context, ZMQ_REQ);
branches[branches.size() -
1].set(zmq::sockopt::sndtimeo, 5000);
bind(branches[branches.size()-1], nodeId);

sendMessage(branches[branches.size() - 1],
std::to_string(nodeId) + " pid");

std::string reply =
receiveMessage(branches[branches.size() - 1]);
std::cout << reply << std::endl;
list.insert(nodeId, parentId);

} else if (list.find(parentId) == -1) {

    std::cout << "Error: Parent not found" << std::endl;

} else {
    int branch = list.find(parentId);
    sendMessage(branches[branch], std::to_string(parentId)
+ "create " + std::to_string(nodeId));

    std::string reply = receiveMessage(branches[branch]);
    std::cout << reply << std::endl;
    list.insert(nodeId, parentId);
}
} else if (command == "exec") {

```

```

std::string s;
getline(std::cin, s);
std::string execCommand;
std::vector<std::string> tmp;
std::string tmp1 = " ";
for (size_t i = 1; i < s.size(); i++) {
    tmp1 += s[i];
    if (s[i] == ' ' || i == s.size() - 1) {
        tmp.push_back(tmp1);
        tmp1 = " ";
    }
}
if (tmp.size() == 2) {
    execCommand = "check";
} else {
    execCommand = "add";
}
int destId = stoi(tmp[0]);
int branch = list.find(destId);
if (branch == -1) {
    std::cout << "There is no such node id" << std::endl;
} else {
    if (execCommand == "check") {
        sendMessage(branches[branch], tmp[0] + "check" +
tmp[1]);

        } else if (execCommand == "add") {
            std::string value;
            sendMessage(branches[branch], tmp[0] + "add" +
tmp[1] + " " + tmp[2]);
        }
    std::string reply = receiveMessage(branches[branch]);
    std::cout << reply << std::endl;
}

```

```

    } else if (command == "kill") {
        int id;
        std::cin >> id;
        int branch = list.find(id);
        if (branch == -1) {
            std::cout << " Error: incorrect node id" << std::endl;
        } else {
            bool is_first = (list.getFirstId(branch) == id);
            sendMessage(branches[branch], std::to_string(id) +
"kill");

            std::string reply = receiveMessage(branches[branch]);
            std::cout << reply << std::endl;
            not_available_nodes.merge(list.getSetOfChilds(id));
            list.erase(id);
            if (is_first) {
                unbind(branches[branch], id);
                branches.erase(branches.begin() + branch);
            }
        }
    } else if (command == "pingall") {
        for (size_t i = 0; i < branches.size(); ++i) {
            int first_node_id = list.getFirstId(i);
            sendMessage(branches[i], std::to_string(first_node_id)
+ " ping");

            std::string received_message =
receiveMessage(branches[i]);
            std::istringstream reply(received_message);
            int node;
            while(reply >> node) {
                std::cout << "list " << node << std::endl;
                not_available_nodes.erase(node);
            }
        }
    }
}

```

```

        if (not_available_nodes.empty()) {
            std::cout << "OK: -1" << std::endl;
        } else {
            std::cout << "OK: ";
            for (size_t i : not_available_nodes) {
                std::cout << i << ' ';
            }
            std::cout << std::endl;
        }
    } else if (command == "exit") {
        for (size_t i = 0; i < branches.size(); ++i) {
            int firstNodeId = list.getFirstId(i);
            sendMessage(branches[i], std::to_string(firstNodeId) +
" kill");

            std::string reply = receiveMessage(branches[i]);
            if (reply != "OK") {
                std::cout << reply << std::endl;
            } else {
                unbind(branches[i], firstNodeId);
            }
        }
        exit(0);
    } else {
        std::cout << "Not correct command" << std::endl;
    }
}
}

```

===== socket.cpp =====

```
#include <socket.hpp>
```

```

void sendMessage(zmq::socket_t& socket, const std::string& msg) {
    zmq::message_t message(msg.size());

```

```

        memcpy(message.data(), msg.c_str(), msg.size());
        socket.send(message, zmq::send_flags::none);
    }

std::string receiveMessage(zmq::socket_t& socket) {
    zmq::message_t msg;
    int msgReceiv;
    try {
        std::optional<size_t> result = socket.recv(msg);
        if (result) {
            msgReceiv = static_cast<int>(*result);
        }
    }
    catch (...) {
        msgReceiv = 0;
    }
    if (msgReceiv == 0) {
        return "Error: Node is unavailable";
    }
    std::string receivedMsg(static_cast<char*>(msg.data()),
msg.size());
    return receivedMsg;
}

void connect(zmq::socket_t& socket, int id) {
    std::string address = "tcp://127.0.0.1:" +
std::to_string(MAIN_PORT + id);
    socket.connect(address);
}

void disconnect(zmq::socket_t& socket, int id) {

```

```

        std::string address = "tcp://127.0.0.1:" +
std::to_string(MAIN_PORT + id);
        socket.disconnect(address);
    }

```

```

void bind(zmq::socket_t& socket, int id) {
    std::string address = "tcp://127.0.0.1:" +
std::to_string(MAIN_PORT + id);
    socket.bind(address);
}

```

```

void unbind(zmq::socket_t& socket, int id) {
    std::string address = "tcp://127.0.0.1:" +
std::to_string(MAIN_PORT + id);
    socket.unbind(address);
}

```

===== topology.cpp =====

```

#include "topology.hpp"

```

```

void Topology::insert(int id, int parentId) {

    if (parentId == -1) {
        std::list<int> newList;
        newList.push_back(id);
        list.push_back(newList);
        return;
    }

    int listId = find(parentId);
    if (listId == -1) {
        throw std::runtime_error("Wrong parent id");
    }

    auto it1 = list.begin();
    std::advance(it1, listId);

```



```

    for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
        if (*it2 == parentId) {
            it1->insert(++it2, id);
            return;
        }
    }
}

```

```

int Topology::find(int id) {
    int curListId = 0;
    for (auto it1 = list.begin(); it1 != list.end(); ++it1) {
        for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
            if (*it2 == id) {
                return curListId;
            }
        }
        ++curListId;
    }
    return -1;
}

```

```

std::set<int> Topology::getSetOfChilDs(int id) {
    int listId = find(id);
    if (listId == -1) {
        throw std::runtime_error("Wrong id");
    }
    std::set<int> getSetOfChilDs;
    getSetOfChilDs.insert(id);
    bool flag = false;
    for (auto it1 = list.begin(); it1 != list.end(); ++it1) {
        for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
            if (flag) {
                getSetOfChilDs.insert(*it2);
            } else if (*it2 == id) {

```

```

        flag = true;
    }
}
if (flag) {
    return getSetOfChilds;
}
return getSetOfChilds;
}

```

```

void Topology::erase(int id) {
    int listId = find(id);
    if (listId == -1) {
        throw std::runtime_error("Wrong id");
    }
    auto it1 = list.begin();
    std::advance(it1, listId);
    for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
        if (*it2 == id) {
            it1->erase(it2, it1->end());
            if (it1->empty()) {
                list.erase(it1);
            }
            return;
        }
    }
}

```

```

int Topology::getFirstId(int listId) {
    auto it1 = list.begin();
    std::advance(it1, listId);
    if (it1->begin() == it1->end()) {
        return -1;
    }
}

```

```

        return *(it1->begin());
    }

===== lab5-7_test.cpp =====

#include <gtest/gtest.h>

#include "topology.hpp"
#include "socket.hpp"
#include <thread>

TEST(FifthSeventhLabTest, SocketTest) {
    zmq::context_t context;
    zmq::socket_t repSocket(context, ZMQ_REP);
    bind(repSocket, 3);

    std::thread serverThread([&repSocket]() {
        std::string receivedMessage = receiveMessage(repSocket);
        EXPECT_EQ(receivedMessage, "TestMSG");

        sendMessage(repSocket, "ReplyMSG");
    });

    zmq::socket_t reqSocket(context, ZMQ_REQ);
    connect(reqSocket, 3);

    sendMessage(reqSocket, "TestMSG");

    std::string replyMessage = receiveMessage(reqSocket);
    EXPECT_EQ(replyMessage, "ReplyMSG");

    disconnect(reqSocket, 3);
    unbind(repSocket, 3);
    serverThread.join();
}

```

```

TEST(FifthSeventhLabTest, TopologyTest) {
    Topology topology;

    topology.insert(1, -1);
    topology.insert(2, 1);
    topology.insert(3, 2);

    EXPECT_EQ(topology.find(1), 0);
    EXPECT_EQ(topology.find(2), 0);
    EXPECT_EQ(topology.find(3), 0);

    EXPECT_EQ(topology.getFirstId(0), 1);

    topology.erase(2);

    EXPECT_EQ(topology.find(2), -1);
}

```

```

int main(int argc, char *argv[]) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

### Демонстрация работы программы

```

marshall@marshall:~/Desktop/OS_labs/build/lab5-7$ export
PATH_TO_CLIENT=
"/home/marshall/Desktop/OS_labs/build/lab5-7/server"
marshall@marshall i n e:~/Desktop/OS_labs/build/lab5-7$ ./client
create 1 -1 OK: 44114
create 2 -1
OK: 44149

```

```
exec 1 myvar 15
OK: 1
exec 2 myvar
OK: 2: 'myvar' not found
exec 1 myvar
OK: 1: 15
pingall
OK: -1
kill 2
OK
pingall
OK: 2
exit
marshall@marshal:~/Desktop/OS_labs/build/lab5-
7 $
```

### **Выводы**

В ходе выполнения лабораторной работы я получил знания и навыки использования серверов сообщений. Я получил понимание концепции асинхронного программирования, узнал о сокетах и сетевом протоколе TCP.