Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторные работы №6-8 по курсу**

**«Операционные системы»**

**Управление серверами сообщений, применение отложенных вычислений, интеграция программных систем друг с другом.**

Студент: Каширин Кирилл Дмитриевич

Группа: М80 – 208Б-20

Вариант: 15

Преподаватель: Миронов Евгений Сергеевич

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

1. **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 топологией «список списков». Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. В данной системе необходимо предусмотреть проверку доступности узлов. При убийстве 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.

1. **Общие сведения о программе**

Программа написана на языке С++ на операционной системе Ubuntu. В программе используется очередь сообщений ZeroMQ.

Программа поддерживает следующие команды:

* create [id] [parent\_id] – создать новый узел [id], родителем которого является узел [parent\_id]. Если [parent\_id] = -1, то родительский узел – управляющий.
* kill [id] – удалить узел [id]. Все дочерние узлы будут также удалены.
* exec [id] add [key] [value] – добавить переменную [key] со значением [value] в словарь узла [id]
* exec [id] check [key] – запросить значение переменной [key] в словаре узла [id].
* pingall – проверить доступность узлов. Будет выведен список всех доступных на данный момент узлов.
* exit – выйти из программы.

1. **Общий метод и алгоритм решения**

В программе используется тип соединения Request-Response. Узлы передают информацию друг другу при помощи очереди сообщений. Все сообщения имеют следующий вид:

[id узла, которому предназначено сообщение] [команда] [аргументы]

Управляющий узел хранит структуру «список списков», в которую записывает id существующих узлов. При помощи этой структуры он определяет, в какой список нужно направить сообщение.

Вычислительный узел, получив сообщение, сравнивает свой id и id из сообщения. Если они совпадают, то узел начинает обрабатывать запрос, в противном случае узел направляет это же сообщение своему ребенку и ждет от него ответа.

Для удобства функции отправки и получения сообщений, а также функции для подключения к сокетам вынесены в отдельный заголовочный файл, который подключается к программам узлов.

Для хранения локального словаря используется контейнер std::unordered\_map. Для проверки доступности узлов используется контейнер std::set. Управляющий узел отправляет запрос всем спискам узлов и получает в ответ строку с id всех доступных узлов списка. Все id добавляются в set, а потом выводятся на экран.

1. **Основные файлы программы**

**List\_of\_list.h**

#include <list>

#include <stdexcept>

class list\_of\_list {

private:

    std::list<std::list<int>> container;

public:

    void insert(int id, int parent\_id) {

        if (parent\_id == -1) {

            std::list<int> new\_list;

            new\_list.push\_back(id);

            container.push\_back(new\_list);

        }

        else {

            int list\_id = find(parent\_id);

            if (list\_id == -1) {

                throw std::runtime\_error("Wrong parent id");

            }

            auto it1 = container.begin();

            std::advance(it1, list\_id);

            for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

                if (\*it2 == parent\_id) {

                    it1->insert(++it2, id);

                    return;

                }

            }

        }

    }

    int find(int id) {

        int cur\_list\_id = 0;

        for (auto it1 = container.begin(); it1 != container.end(); ++it1) {

            for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

                if (\*it2 == id) {

                    return cur\_list\_id;

                }

            }

            ++cur\_list\_id;

        }

        return -1;

    }

    void erase(int id) {

        int list\_id = find(id);

        if (list\_id == -1) {

            throw std::runtime\_error("Wrong id");

        }

        auto it1 = container.begin();

        std::advance(it1, list\_id);

        for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

            if (\*it2 == id) {

                it1->erase(it2, it1->end());

                if (it1->empty()) {

                    container.erase(it1);

                }

                return;

            }

        }

    }

    int get\_first\_id(int list\_id) {

        auto it1 = container.begin();

        std::advance(it1, list\_id);

        if (it1->begin() == it1->end()) {

            return -1;

        }

        return \*(it1->begin());

    }

};

**control.cpp**

#include <unistd.h>

#include <sstream>

#include <set>

#include <zmq.hpp>

#include <iostream>

#include "list\_of\_list.h"

using namespace std;

const int MAIN\_PORT = 4040;

void send\_message(zmq::socket\_t& socket, const string& msg) {

    zmq::message\_t message(msg.size());

    memcpy(message.data(), msg.c\_str(), msg.size());

    socket.send(message);

}

string receive\_message(zmq::socket\_t& socket) {

    zmq::message\_t message;

    int chars\_read;

    try {

        chars\_read = (int)socket.recv(&message);

    }

    catch (...) {

        chars\_read = 0;

    }

    if (chars\_read == 0) {

        return "Error: node is unavailable [zmq\_func]";

    }

    string received\_msg(static\_cast<char\*>(message.data()), message.size());

    return received\_msg;

}

int main() {

    list\_of\_list network;

    std::vector<zmq::socket\_t> branches;

    zmq::context\_t context;

    string command;

    while (true) {

        cin >> command;

        if (command == "create") {

            int node\_id, parent\_id;

            cin >> node\_id >> parent\_id;

            if (network.find(node\_id) != -1) {

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

            } else if (parent\_id == -1) {

                pid\_t pid = fork();

                if (pid < 0) {

                    cout << "Can't create new process" << endl;

                    return -1;

                } else if (pid == 0) {

                    execl("./counting", "./counting", to\_string(node\_id).c\_str(), NULL);

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

                    return -2;

                }

                branches.emplace\_back(context, ZMQ\_REQ);

                branches[branches.size() - 1].setsockopt(ZMQ\_SNDTIMEO, 5000);

                string adr = "tcp://127.0.0.1:" + to\_string(MAIN\_PORT + node\_id);

                branches[branches.size()-1].bind(adr);

                send\_message(branches[branches.size() - 1], to\_string(node\_id) + " pid");

                string reply = receive\_message(branches[branches.size() - 1]);

                cout << reply << endl;

                network.insert(node\_id, parent\_id);

            } else if (network.find(parent\_id) == -1) {

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

            } else {

                int branch = network.find(parent\_id);

                send\_message(branches[branch], to\_string(parent\_id) + "create " + to\_string(node\_id));

                string reply = receive\_message(branches[branch]);

                cout << reply << endl;

                network.insert(node\_id, parent\_id);

            }

        } else if (command == "exec") {

            string s;

            getline(cin, s);

            string exec\_command;

            vector<string> tmp;

            string tmp1 = "";

            for (int 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) {

                exec\_command = "check";

            } else {

                exec\_command = "add";

            }

            int dest\_id = stoi(tmp[0]);

            int branch = network.find(dest\_id);

            if (branch == -1) {

                cout << "There is no such node id" << endl;

            } else {

                if (exec\_command == "check") {

                    send\_message(branches[branch], tmp[0]+" check "+tmp[1]);

                } else if (exec\_command == "add") {

                    string value;

                    send\_message(branches[branch], tmp[0]+" add "+tmp[1]+" "+tmp[2]);

                }

                string reply = receive\_message(branches[branch]);

                cout << reply << endl;

            }

        } else if (command == "kill") {

            int id;

            cin >> id;

            int branch = network.find(id);

            if (branch == -1) {

                cout << " Error: incorrect node id" << endl;

            } else {

                bool is\_first = (network.get\_first\_id(branch) == id);

                send\_message(branches[branch], to\_string(id)+"kill");

                 std::string reply = receive\_message(branches[branch]);

                std::cout << reply << std::endl;

                network.erase(id);

                if (is\_first) {

                    string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);

                    branches[branch].unbind(address);

                    branches.erase(branches.begin() + branch);

                }

            }

        } else if (command == "pingall") {

            set<int> available\_nodes;

            for (size\_t i = 0; i < branches.size(); ++i) {

                int first\_node\_id = network.get\_first\_id(i);

                send\_message(branches[i], std::to\_string(first\_node\_id) + " pingall");

                string received\_message = receive\_message(branches[i]);

                istringstream reply(received\_message);

                int node;

                while(reply >> node) {

                    available\_nodes.insert(node);

                }

            }

            cout << "OK: ";

            if (available\_nodes.empty()) {

                cout << "no available nodes" << endl;

            }

            else {

                for (auto v : available\_nodes) {

                    cout << v << " ";

                }

                cout << endl;

            }

        } else if (command == "exit") {

            for (size\_t i = 0; i < branches.size(); ++i) {

                int first\_node\_id = network.get\_first\_id(i);

                send\_message(branches[i], to\_string(first\_node\_id) + " kill");

                string reply = receive\_message(branches[i]);

                if (reply != "OK") {

                    cout << reply << endl;

                } else {

                    string adr = "tcp://127.0.0.1:" + to\_string(MAIN\_PORT + first\_node\_id);

                    branches[i].unbind(adr);

                }

            }

            exit(0);

        } else {

            cout << "Not correct command" << endl;

        }

    }

}

**counting.cpp**

#include <unordered\_map>

#include <unistd.h>

#include <sstream>

#include <zmq.hpp>

#include <iostream>

using namespace std;

const int MAIN\_PORT = 4040;

void send\_message(zmq::socket\_t& socket, const string& msg) {

    zmq::message\_t message(msg.size());

    memcpy(message.data(), msg.c\_str(), msg.size());

    socket.send(message);

}

string receive\_message(zmq::socket\_t& socket) {

    zmq::message\_t message;

    int chars\_read;

    try {

        chars\_read = (int)socket.recv(&message);

    }

    catch (...) {

        chars\_read = 0;

    }

    if (chars\_read == 0) {

        return "Error: node is unavailable [zmq\_func]";

    }

    string received\_msg(static\_cast<char\*>(message.data()), message.size());

    return received\_msg;

}

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

    if (argc != 2 && argc != 3) {

        throw runtime\_error("Wrong args for counting node");

    }

    int cur\_id = atoi(argv[1]);

    int child\_id = -1;

    if (argc == 3) {

        child\_id = atoi(argv[2]);

    }

    unordered\_map<string, int> dictionary;

    zmq::context\_t context;

    zmq::socket\_t parent\_socket(context, ZMQ\_REP);

    string adr = "tcp://127.0.0.1:" + to\_string(MAIN\_PORT + cur\_id);

    parent\_socket.connect(adr);

    zmq::socket\_t child\_socket(context, ZMQ\_REQ);

    if (child\_id != -1) {

        adr = "tcp://127.0.0.1:" + to\_string(MAIN\_PORT + child\_id);

        child\_socket.bind(adr);

    }

    child\_socket.setsockopt(ZMQ\_SNDTIMEO, 5000);

    string message;

    while (true) {

        message = receive\_message(parent\_socket);

        istringstream request(message);

        int dest\_id;

        request >> dest\_id;

        string command;

        request >> command;

        if (dest\_id == cur\_id) {

            if (command == "pid") {

                send\_message(parent\_socket, "OK: " + to\_string(getpid()));

            } else if (command == "create") {

                int new\_child\_id;

                request >> new\_child\_id;

                if (child\_id != -1) {

                    adr = "tcp://127.0.0.1:" + to\_string(MAIN\_PORT + child\_id);

                    child\_socket.unbind(adr);

                }

                adr = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + new\_child\_id);

                child\_socket.bind(adr);

                pid\_t pid = fork();

                if (pid < 0) {

                    cout << "Can't create new process" << endl;

                    return -1;

                }

                if (pid == 0) {

                    execl("./counting", "./counting", to\_string(new\_child\_id).c\_str(), to\_string(child\_id).c\_str(), NULL);

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

                    return -2;

                }

                send\_message(child\_socket, to\_string(new\_child\_id) + " pid");

                child\_id = new\_child\_id;

                send\_message(parent\_socket, receive\_message(child\_socket));

            } else if (command == "check") {

                string key;

                request >> key;

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

                    send\_message(parent\_socket, "OK: " + std::to\_string(cur\_id) + ": " + std::to\_string(dictionary[key]));

                } else {

                    send\_message(parent\_socket, "OK: " + std::to\_string(cur\_id) + ": '" + key + "' not found");

                }

            } else if (command == "add") {

                string key;

                int value;

                request >> key >> value;

                dictionary[key] = value;

                send\_message(parent\_socket, "OK: " + to\_string(cur\_id));

            }   else if (command == "pingall") {

                string reply;

                if (child\_id != -1) {

                    send\_message(child\_socket, to\_string(child\_id) + " pingall");

                    string msg = receive\_message(child\_socket);

                    reply += " " + msg;

                }

                send\_message(parent\_socket, to\_string(cur\_id) + reply);

            } else if (command == "kill") {

                if (child\_id != -1) {

                    send\_message(child\_socket, std::to\_string(child\_id) + " kill");

                    std::string msg = receive\_message(child\_socket);

                    if (msg == "OK") {

                        send\_message(parent\_socket, "OK");

                    }

                    string adr = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + child\_id);

                    child\_socket.unbind(adr);

                    std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + cur\_id);

                    parent\_socket.disconnect(address);

*//disconnect(parent\_socket, cur\_id);*

                    break;

                }

                send\_message(parent\_socket, "OK");

*// disconnect(parent\_socket, cur\_id);*

                 std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + cur\_id);

                    parent\_socket.disconnect(address);

                break;

            }

        }

         else if (child\_id != -1) {

            send\_message(child\_socket, message);

            send\_message(parent\_socket, receive\_message(child\_socket));

            if (child\_id == dest\_id && command == "kill") {

                child\_id = -1;

            }

        } else {

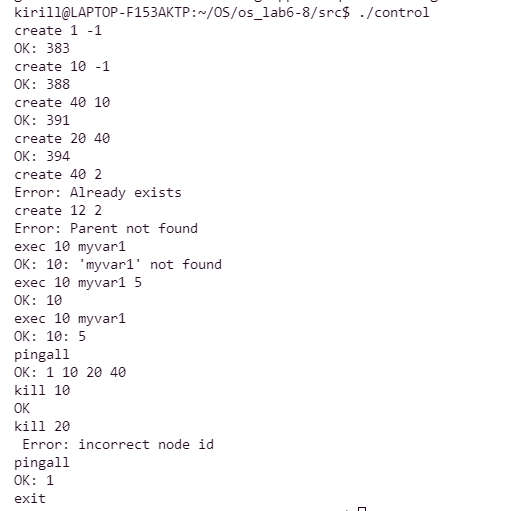
            send\_message(parent\_socket, "Error: node is unavailable");

        }

    }

}

1. **Демонстрация работы программы**



1. **Выводы**

Данная лабораторная работа была направлена на изучении технологии очереди сообщений, на основе которой необходимо было построить сеть с заданной топологией.

Наряду с каналами и отображаемыми файлами, очереди сообщений являются достаточно удобным способом для взаимодействия между процессами. ZeroMQ предоставляет достаточно простой интерфейс для передачи сообщений, а также поддерживает все возможные типы соединений.

Полученные мной навыки работы с очередями сообщений можно использовать при проектировании мессенджеров, многопользовательских игр, да и вообще для любых мультипроцессорных программ.