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.

2. Общие сведения о программе

Программа написана на языке C++ 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.

3. 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.

4. 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>
```

```cpp
#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>
```

```cpp
#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");
        }
    }
}
```

5. Демонстрация работы программы

```
kirill@LAPTOP-F153AKTP:~/OS/os_lab6-8/src$ ./control
create 1 -1
OK: 383
create 10 -1
OK: 388
create 40 10
OK: 391
create 20 40
OK: 394
create 40 2
Error: Already exists
create 12 2
Error: Parent not found
exec 10 myvar1
OK: 10: 'myvar1' not found
exec 10 myvar1 5
OK: 10
exec 10 myvar1
OK: 10: 5
pingall
OK: 1 10 20 40
kill 10
OK
kill 20
 Error: incorrect node id
pingall
OK: 1
exit
```

6. 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.