Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу
«Операционные системы»

Управление серверами сообщений

Студент: Черненко Илья Денисович
Группа: М80 – 206Б-18
Вариант: 46
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2019

Постановка задачи

Реализовать распределенную систему по обработке запросов. В данной 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 сервера сообщений zmq. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом.

Вариант задания: 46. Топология — дерево общего вида. Тип вычислительной команды — локальный целочисленный словарь. Тип проверки узлов на доступность — ping.

Общие сведения о программе

Программа состоит из двух файлов, которые компилируются в исполнительные файлы(которые представляют управляющий и вычислительные узлы), а так же из статической библиотеки, которая подключается к вышеуказанным файлам. Общение между процессами происходит с помощью библиотеки zmq.

Общий метод и алгоритм решения

- Управляющий узел принимает команды, обрабатывает их и пересылает дочерним узлам или выводит сообщение об ошибке.
- Дочерние узлы проверяют, может ли быть команда выполнена в данном узле, если нет, то она пересылается в один из дочерних узлов, из которого возвращается некоторое сообщение об успехе или об ошибке, которое потом пересылается обратно по дереву.
- Если узел недоступен, то будет выведено сообщение о недоступности узла, которое будет передано управляющему узлу.
- При удалении узла, все его потомки рекурсивно уничтожаются.

Код программы

sf.h

```
#pragma once
#include <iostream>
#include <string>
#include "zmq.hpp"
#include "unistd.h"
bool send_msg(zmq::socket_t& socket, const std::string& message);
std::string get_msg(zmq::socket_t& socket);
int bind_socket(zmq::socket_t& socket);
void crt_node(int id, int portNumber);
```

sf.cpp

```
#include "sf.h"
bool send_msg(zmq::socket_t& socket, const std::string& message) {
    zmq::message_t m(message.size());
    memcpy(m.data(), message.c_str(), message.size());
    try {
        socket.send(m);
        return true;
    } catch(...) {
        return false;
    }
}
std::string get_msg(zmq::socket_t& socket) {
    zmq::message_t message;
    bool msg_got;
    try {
        msg_got = socket.recv(&message);
```

```
    } catch(...) {
        msg_got = false;
    }
    std::string received(static_cast<char*>(message.data()), message.size());
    if(!msg_got || received.empty()) {
        return "Error: Node is unavailable";
    } else {
        return received;
    }
}
int bind_socket(zmq::socket_t& socket) {
    int port = 30000;
    std::string port_tmp = "tcp://127.0.0.1:";
    while(true) {
        try {
            socket.bind(port_tmp + std::to_string(port));
            break;
        } catch(...) {
            port++;
        }
    }
    return port;
}
void crt_node(int id, int portNumber) {
    char* arg0 = strdup("./child_node");
    char* arg1 = strdup((std::to_string(id)).c_str());
    char* arg2 = strdup((std::to_string(portNumber)).c_str());
    char* args[] = {arg0, arg1, arg2, nullptr};
    execv("./child_node", args);
```

```
}
```

child_node.cpp

```
#include <string>
#include <sstream>
#include <zmq.hpp>
#include <csignal>
#include <iostream>
#include <unordered_map>
#include "sf.h"
int main(int argc, char* argv[]) {
    if(argc != 3) {
        std::cerr << "Not enough parameters" << std::endl;
        exit(-1);
    }
    int id = std::stoi(argv[1]);
    int parent_port = std::stoi(argv[2]);
    zmq::context_t ctx;
    zmq::socket_t parent_socket(ctx, ZMQ_REP);
    std::string port_tmp = "tcp://127.0.0.1:";
    parent_socket.connect(port_tmp + std::to_string(parent_port));
    std::unordered_map<int, int> pids;
    std::unordered_map<int, int> ports;
    std::unordered_map<int, zmq::socket_t> sockets;
    while(true) {
        std::string action = get_msg(parent_socket);
        std::stringstream s(action);
        std::string command;
        s >> command;
        if(command == "pid") {
```

```
    std::string reply = "Ok: " + std::to_string(getpid());
    send_msg(parent_socket, reply);
} else if(command == "create") {
    int size, node_id;
    s >> size;
    std::vector<int> path(size);
    for(int i = 0; i < size; ++i) {
        s >> path[i];
    }
    s >> node_id;
    if(size == 0) {
        auto socket = zmq::socket_t(ctx, ZMQ_REQ);
        socket.setsockopt(ZMQ_SNDTIMEO, 5000);
        socket.setsockopt(ZMQ_LINGER, 5000);
        socket.setsockopt(ZMQ_RCVTIMEO, 5000);
        socket.setsockopt(ZMQ_REQ_CORRELATE, 1);
        socket.setsockopt(ZMQ_REQ_RELAXED, 1);
        sockets.emplace(node_id, std::move(socket));
        int port = bind_socket(sockets.at(node_id));
        int pid = fork();
        if(pid == -1) {
            send_msg(parent_socket, "Unable to fork");
        } else if(pid == 0) {
            crt_node(node_id, port);
        } else {
            ports[node_id] = port;
            pids[node_id] = pid;
            send_msg(sockets.at(node_id), "pid");
            send_msg(parent_socket, get_msg(sockets.at(node_id)));
```

```
            }
        } else {
            int next_smb = path.front();
            path.erase(path.begin());
            std::stringstream msg;
            msg << "create " << path.size();
            for(int i : path) {
                msg << " " << i;
            }
            msg << " " << node_id;
            send_msg(sockets.at(next_smb), msg.str());
            send_msg(parent_socket, get_msg(sockets.at(next_smb)));
        }
    } else if(command == "remove") {
        int size, node_id;
        s >> size;
        std::vector<int> path(size);
        for(int i = 0; i < size; ++i) {
            s >> path[i];
        }
        s >> node_id;
        if(path.empty()) {
            send_msg(sockets.at(node_id), "kill");
            get_msg(sockets.at(node_id));
            kill(pids[node_id], SIGTERM);
            kill(pids[node_id], SIGKILL);
            pids.erase(node_id);
            sockets.at(node_id).disconnect(port_tmp +
std::to_string(ports[node_id]));
```

```
            ports.erase(node_id);
            sockets.erase(node_id);
            send_msg(parent_socket, "Ok");
        } else {
            int next_smb = path.front();
            path.erase(path.begin());
            std::stringstream msg;
            msg << "remove " << path.size();
            for(int i : path) {
                msg << " " << i;
            }
            msg << " " << node_id;
            send_msg(sockets.at(next_smb), msg.str());
            send_msg(parent_socket, get_msg(sockets.at(next_smb)));
        }
    } else if(command == "exec") {
        int size;
        s >> size;
        std::vector<int> path(size);
        for(int i = 0; i < size; ++i) {
            s >> path[i];
        }
        if(path.empty()) {
            send_msg(parent_socket, "Node is available");
        } else {
            int next_smb = path.front();
            path.erase(path.begin());
            std::stringstream msg;
            msg << "exec " << path.size();
```

```
            for(int i : path) {
                msg << " " << i;
            }
            std::string received;
            if(!send_msg(sockets.at(next_smb), msg.str())) {
                received = "Node is unavailable";
            } else {
                received = get_msg(sockets.at(next_smb));
            }
            send_msg(parent_socket, received);
        }
    } else if(command == "ping") {
        int size;
        s >> size;
        std::vector<int> path(size);
        for(int i = 0; i < size; ++i) {
            s >> path[i];
        }
        if(path.empty()) {
            send_msg(parent_socket, "Ok: 1");
        } else {
            int next_smb = path.front();
            path.erase(path.begin());
            std::stringstream msg;
            msg << "ping " << path.size();
            for(int i : path) {
                msg << " " << i;
            }
            std::string received;
```

```
            if(!send_msg(sockets.at(next_smb), msg.str())) {
                received = "Node is unavailable";
            } else {
                received = get_msg(sockets.at(next_smb));
            }
            send_msg(parent_socket, received);
        }
    } else if(command == "kill") {
        for(auto& item : sockets) {
            send_msg(item.second, "kill");
            get_msg(item.second);
            kill(pids[item.first], SIGTERM);
            kill(pids[item.first], SIGKILL);
        }
        send_msg(parent_socket, "Ok");
    }
    if(parent_port == 0) {
        break;
    }
  }
}
```

main.cpp

```
#include <iostream>
#include <string>
#include <zmq.hpp>
#include <vector>
#include <csignal>
#include <sstream>
#include <memory>
```

```
#include <unordered_map>
#include "sf.h"

struct Node {
    Node(int id, std::weak_ptr<Node> parent) : id(id), parent(parent) {};
    int id;
    std::weak_ptr<Node> parent;
    std::unordered_map<int, std::shared_ptr<Node>> children;
    std::unordered_map<std::string, int> dictionary;
};

class General_tree {
public:
    bool insert(int node_id, int parent_id) {
        if(root == nullptr) {
            root = std::make_shared<Node>(node_id, std::weak_ptr<Node>());
            return true;
        }
        std::vector<int> path = get_path(parent_id);
        if(path.empty()) {
            return false;
        }
        path.erase(path.begin());
        std::shared_ptr<Node> tmp = root;
        for(const auto& node : path) {
            tmp = tmp->children[node];
        }
        tmp->children[node_id] = std::make_shared<Node>(node_id, tmp);
        return true;
```

```
}

bool rmv(int node_id) {
    std::vector<int> path = get_path(node_id);
    if(path.empty()) {
        return false;
    }
    path.erase(path.begin());
    std::shared_ptr<Node> tmp = root;
    for(const auto& node : path) {
        tmp = tmp->children[node];
    }
    if(tmp->parent.lock()) {
        tmp = tmp->parent.lock();
        tmp->children.erase(node_id);
    } else {
        root = nullptr;
    }
    return true;
}
[[nodiscard]] std::vector<int> get_path(int id) const {
    std::vector<int> path;
    if(!get_node(root, id, path)) {
        return {};
    } else {
        return path;
    }
}
void add_dictionary(int id, std::string name, int value) {
```

```
        std::vector<int> path = get_path(id);
        path.erase(path.begin());
        std::shared_ptr<Node> tmp = root;
        for(const auto& node : path) {
            tmp = tmp->children[node];
        }
        tmp->dictionary[name] = value;
    }

    void find_dictionary(int id, std::string name) {
        std::vector<int> path = get_path(id);
        path.erase(path.begin());
        std::shared_ptr<Node> tmp = root;
        for(const auto& node : path) {
            tmp = tmp->children[node];
        }
        if (tmp->dictionary.find(name) == tmp->dictionary.end()) {
            std::cout << "'" << name << "' not found" << std::endl;
        } else {
            std::cout << tmp->dictionary[name] << std::endl;
        }
    }
private:
    bool get_node(const std::shared_ptr<Node>& current, int id, std::vector<int>&
path) const {
        if(!current) {
            return false;
        }
        if(current->id == id) {
```

```
            path.push_back(current->id);
            return true;
        }
        path.push_back(current->id);
        for(const auto& node : current->children) {
            if(get_node(node.second, id, path)) {
                return true;
            }
        }
        path.pop_back();
        return false;
    }
    std::shared_ptr<Node> root = nullptr;
};

int main() {
    General_tree tree;
    std::string command;
    int child_pid = 0;
    int child_id = 0;
    zmq::context_t ctx(1);
    zmq::socket_t rule_socket(ctx, ZMQ_REQ);
    rule_socket.setsockopt(ZMQ_SNDTIMEO, 5000);
    rule_socket.setsockopt(ZMQ_LINGER, 5000);
    rule_socket.setsockopt(ZMQ_RCVTIMEO, 5000);
    rule_socket.setsockopt(ZMQ_REQ_CORRELATE, 1);
    rule_socket.setsockopt(ZMQ_REQ_RELAXED, 1);
    int port_n = bind_socket(rule_socket);
    while(std::cin >> command) {
```

```
if(command == "create") {
    int node_id, parent_id;
    std::string result;
    std::cin >> node_id >> parent_id;
    if(!child_pid) {
        child_pid = fork();
        if(child_pid == -1) {
            std::cout << "Unable to create process" << std::endl;
            exit(-1);
        } else if(child_pid == 0) {
            crt_node(node_id, port_n);
        } else {
            parent_id = 0;
            child_id = node_id;
            send_msg(rule_socket, "pid");
            result = get_msg(rule_socket);
        }
    } else {
        if(!tree.get_path(node_id).empty()) {
            std::cout << "Error: Already exists" << std::endl;
            continue;
        }
        std::vector<int> path = tree.get_path(parent_id);
        if(path.empty()) {
            std::cout << "Error: Parent not found" << std::endl;
            continue;
        }
        path.erase(path.begin());
        std::stringstream s;
```

```
            s << "create " << path.size();
            for(int id : path) {
                s << " " << id;
            }
            s << " " << node_id;
            send_msg(rule_socket, s.str());
            result = get_msg(rule_socket);
        }

        if(result.substr(0, 2) == "Ok") {
            tree.insert(node_id, parent_id);
        }
        std::cout << result << std::endl;
    } else if(command == "remove") {
        if(child_pid == 0) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        int node_id;
        std::cin >> node_id;
        if(node_id == child_id) {
            send_msg(rule_socket, "kill");
            get_msg(rule_socket);
            kill(child_pid, SIGTERM);
            kill(child_pid, SIGKILL);
            child_id = 0;
            child_pid = 0;
            std::cout << "Ok" << std::endl;
            tree.rmv(node_id);
```

```
            continue;
        }
        std::vector<int> path = tree.get_path(node_id);
        if(path.empty()) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        path.erase(path.begin());
        std::stringstream s;
        s << "remove " << path.size() - 1;
        for(int i : path) {
            s << " " << i;
        }
        send_msg(rule_socket, s.str());
        std::string recieved = get_msg(rule_socket);
        if(recieved.substr(0, 2) == "Ok") {
            tree.rmv(node_id);
        }
        std::cout << recieved << std::endl;
    } else if(command == "exec") {
        if(child_pid == 0) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        int node_id;
        std::cin >> node_id;
        std::string name_value;
        std::getline(std::cin, name_value);
        std::vector<int> path = tree.get_path(node_id);
```

```
if(path.empty()) {
    std::cout << "Error: Not found" << std::endl;
    continue;
}
path.erase(path.begin());
std::stringstream s;
s << "exec " << path.size();
for(int i : path) {
    s << " " << i;
}
std::string received;
if(!send_msg(rule_socket, s.str())) {
    received = "Node is unavailable";
} else {
    received = get_msg(rule_socket);
    if (received == "Node is available") {
        std::string name;
        int value;
        int size_arguments = name_value.size();
        std::stringstream ss(name_value);
        bool searchNeeded = true;
        for (int i = 1; i < size_arguments; ++i) {
            if (name_value[i] == ' ') {
                ss >> name;
                ss >> value;
                tree.add_dictionary(node_id, name, value);
                std::cout << "Ok:" << node_id << std::endl;
                searchNeeded = false;
                break;
```

```
            }
        }
        if (searchNeeded) {
            ss >> name;
            std::cout << "Ok:" << node_id << ": ";
            tree.find_dictionary(node_id, name);
        }
    } else {
        std::cout << received << std::endl;
    }
  }
} else if(command == "ping") {
  if(child_pid == 0) {
    std::cout << "Error: Not found" << std::endl;
    continue;
  }
  int node_id;
  std::cin >> node_id;
  std::vector<int> path = tree.get_path(node_id);
  if(path.empty()) {
    std::cout << "Error: Not found" << std::endl;
    continue;
  }
  path.erase(path.begin());
  std::stringstream s;
  s << "ping " << path.size();
  for(int i : path) {
    s << " " << i;
  }
```

```
            std::string received;
            if(!send_msg(rule_socket, s.str())) {
                received = "Node is unavailable";
            } else {
                received = get_msg(rule_socket);
            }
            std::cout << received << std::endl;
        } else if(command == "exit") {
            send_msg(rule_socket, "kill");
            get_msg(rule_socket);
            kill(child_pid, SIGTERM);
            kill(child_pid, SIGKILL);
            break;
        } else {
            std::cout << "Unknown command" << std::endl;
        }
        command.clear();
    }
    return 0;
}
```

Демонстрация работы программы

```
create 1 -1
Ok: 3354
create 3 1
Ok: 3358
create 5 3
Ok: 3361
create 2 1
Ok: 3364
```

create 4 1

Ok: 3367

ping 1

Ok: 1

ping 2

Ok: 1

ping 3

Ok: 1

ping 4

Ok: 1

ping 5

Ok: 1

ping 10

Error: Not found

remove 3

Ok

ping 3

Error: Not found

ping 5

Error: Not found

exec 2 hello 777

Ok:2

exec 2 hello

Ok:2: 777

exec 2 hi

Ok:2: 'hi' not found

exit

Вывод

При написании данной лабораторной работы, я научился работать с очередями сообщений и применять их для реализации топологии процессов-узлов. Изучил принципы работы процессов в ОС Linux.