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

Лабораторная работа №6 по курсу «Операционные системы»

Работа с динамическими библиотеками

Студент: Симонов Сергей Яковлевич
Группа: М80 – 206Б-18
Вариант: 26
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2019

Содержание

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, 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.
Тип топологии: Дерево общего вида. Тип команды: Локальный таймер. Тип проверки доступности узлов: Пинг узла с указанным id.

Общие сведения о программе, метод и алгоритм решения

Программа разбита на файлы soketRoutine.hpp, soketRoutine.hpp (отвечают за работу с сокетами – получение и отправку сообщений, создание сокета), calcNode.cpp (содержит описание логики вычислительного узла), handlerNode.cpp (содержит описание логики управляющего узла).
Каждый вычислительный узел при создании получает номер порта родителя, к которому он должен подключиться, а также свой id. Внутри себя он содержит 3 хэш-таблицы, содержащие сокеты детей узла, их идентификаторы процессов и номера портов.
При получении нового сообщения, адресованного конкретному узлу, строится путь до этого узла – вектор, содержащий id узлов на пути от управляющего к указанному и сообщение посылается в основной сокет, откуда согласно указанному пути пересылается через другие сокеты к необходимому узлу.

Основные файлы программы

calcNode.cpp

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

```
#include <unordered_map>

#include "socketRoutine.hpp"

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 parentPort = std::stoi(argv[2]);
    zmq::context_t ctx;
    zmq::socket_t parentSocket(ctx, ZMQ_REP);
    std::string portTemplate = "tcp://127.0.0.1:";
    parentSocket.connect(portTemplate + std::to_string(parentPort));
    std::unordered_map<int, zmq::socket_t> sockets;
    std::unordered_map<int, int> pids;
    std::unordered_map<int, int> ports;
    auto start = std::chrono::high_resolution_clock::now();
    auto stop = std::chrono::high_resolution_clock::now();
    auto time = 0;
    bool clockStarted = false;
    while(true) {
        std::string action = ReceiveMessage(parentSocket);
        std::stringstream s(action);
        std::string command;
        s >> command;
        if(command == "pid") {
            std::string reply = "Ok: " + std::to_string(getpid());
            SendMessage(parentSocket, reply);
        } else if(command == "create") {
            int size, nodeId;
            s >> size;
            std::vector<int> path(size);
            for(int i = 0; i < size; ++i) {
                s >> path[i];
            }
            s >> nodeId;
            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(nodeId, std::move(socket));
            int port = BindSocket(sockets.at(nodeId));
            std::cout << port << std::endl;
            int pid = fork();
            if(pid == -1) {
               SendMessage(parentSocket, "Unable to fork");
            } else if(pid == 0) {
               CreateNode(nodeId, port);
            } else {
               ports[nodeId] = port;
               pids[nodeId] = pid;
               SendMessage(sockets.at(nodeId), "pid");
               SendMessage(parentSocket, ReceiveMessage(sockets.at(nodeId)));
            }
         } else {
            int nextId = path.front();
            path.erase(path.begin());
            std::stringstream msg;
            msg << "create " << path.size();
            for(int i : path) {
               msg << " " << i;
            }
            msg << " " << nodeId;
            SendMessage(sockets.at(nextId), msg.str());
            SendMessage(parentSocket, ReceiveMessage(sockets.at(nextId)));
         }
      } else if(command == "remove") {
         int size, nodeId;
         s >> size;
         std::vector<int> path(size);
         for(int i = 0; i < size; ++i) {
            s >> path[i];
         }
         s >> nodeId;
         if(path.empty()) {
            SendMessage(sockets.at(nodeId), "kill");
            ReceiveMessage(sockets.at(nodeId));
            kill(pids[nodeId], SIGTERM);
            kill(pids[nodeId], SIGKILL);
            pids.erase(nodeId);
            sockets.at(nodeId).disconnect(portTemplate +
std::to_string(ports[nodeId]));
            ports.erase(nodeId);
            sockets.erase(nodeId);
            SendMessage(parentSocket, "Ok");
```

```
    } else {
        int nextId = path.front();
        path.erase(path.begin());
        std::stringstream msg;
        msg << "remove " << path.size();
        for(int i : path) {
            msg << " " << i;
        }
        msg << " " << nodeId;
        SendMessage(sockets.at(nextId), msg.str());
        SendMessage(parentSocket, ReceiveMessage(sockets.at(nextId)));
    }
} else if(command == "exec") {
    int size;
    std::string subcommand;
    s >> subcommand >> size;
    std::vector<int> path(size);
    for(int i = 0; i < size; ++i) {
        s >> path[i];
    }
    if(path.empty()) {
        if(subcommand == "start") {
            start = std::chrono::high_resolution_clock::now();
            clockStarted = true;
            SendMessage(parentSocket, "Ok:" + std::to_string(id));
        } else if(subcommand == "stop") {
            if(clockStarted) {
                stop = std::chrono::high_resolution_clock::now();
                time += std::chrono::duration_cast<std::chrono::milliseconds>
                    (stop - start).count();
                clockStarted = false;
            }
            SendMessage(parentSocket, "Ok:" + std::to_string(id));
        } else if(subcommand == "time") {
            SendMessage(parentSocket, "Ok: " + std::to_string(id) + ": "
                    + std::to_string(time));
        }
    } else {
        int nextId = path.front();
        path.erase(path.begin());
        std::stringstream msg;
        msg << "exec " << subcommand << " " << path.size();
        for(int i : path) {
            msg << " " << i;
        }
```

```
            SendMessage(sockets.at(nextId), msg.str());
            SendMessage(parentSocket, ReceiveMessage(sockets.at(nextId)));
        }
    } 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()) {
            SendMessage(parentSocket, "Ok: 1");
        } else {
            int nextId = path.front();
            path.erase(path.begin());
            std::stringstream msg;
            msg << "ping " << path.size();
            for(int i : path) {
                msg << " " << i;
            }
            std::string received;
            if(!SendMessage(sockets.at(nextId), msg.str())) {
                received = "Node is unavailable";
            } else {
                received = ReceiveMessage(sockets.at(nextId));
            }
            SendMessage(parentSocket, received);
        }
    } else if(command == "kill") {
        for(auto& item : sockets) {
            SendMessage(item.second, "kill");
            ReceiveMessage(item.second);
            kill(pids[item.first], SIGTERM);
            kill(pids[item.first], SIGKILL);
        }
        SendMessage(parentSocket, "Ok");
    }
    if(parentPort == 0) {
        break;
    }
  }
}
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
project(lab6)

set(CMAKE_CXX_STANDARD 17)
add_executable(main handlerNode.cpp)
add_executable(calcNode calcNode.cpp)
add_library(sockets socketRoutine.cpp socketRoutine.hpp)

target_link_libraries(sockets zmq)
target_link_libraries(main zmq sockets)
target_link_libraries(calcNode zmq sockets)
```

handlerNode.cpp

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

#include "socketRoutine.hpp"

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

class NTree {
public:
    bool Insert(int nodeId, int parentId) {
        if(root == nullptr) {
            root = std::make_shared<TreeNode>(nodeId, std::weak_ptr<TreeNode>());
            return true;
        }
        std::vector<int> pathToNode = PathTo(parentId);
        if(pathToNode.empty()) {
            return false;
        }
```

```
        pathToNode.erase(pathToNode.begin());
        std::shared_ptr<TreeNode> temp = root;
        for(const auto& node : pathToNode) {
            temp = temp->children[node];
        }
        temp->children[nodeId] = std::make_shared<TreeNode>(nodeId, temp);
        return true;
    }

    bool Remove(int nodeId) {
        std::vector<int> pathToNode = PathTo(nodeId);
        if(pathToNode.empty()) {
            return false;
        }
        pathToNode.erase(pathToNode.begin());
        std::shared_ptr<TreeNode> temp = root;
        for(const auto& node : pathToNode) {
            temp = temp->children[node];
        }
        if(temp->parent.lock()) {
            temp = temp->parent.lock();
            temp->children.erase(nodeId);
        } else {
            root = nullptr;
        }
        return true;
    }
    std::vector<int> PathTo(int id) const {
        std::vector<int> path;
        if(!findNode(root, id, path)) {
            return {};
        } else {
            return path;
        }
    }
private:
    bool findNode(const std::shared_ptr<TreeNode>& 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(findNode(node.second, id, path)) {
                return true;
            }
        }
        path.pop_back();
        return false;
    }
    std::shared_ptr<TreeNode> root = nullptr;
};

int main() {
    NTree calcs;
    std::string action;
    int childPid = 0;
    int childId = 0;
    zmq::context_t ctx(1);
    zmq::socket_t handlerSocket(ctx, ZMQ_REQ);
    handlerSocket.setsockopt(ZMQ_SNDTIMEO, 5000);
    handlerSocket.setsockopt(ZMQ_LINGER, 5000);
    handlerSocket.setsockopt(ZMQ_RCVTIMEO, 5000);
    handlerSocket.setsockopt(ZMQ_REQ_CORRELATE, 1);
    handlerSocket.setsockopt(ZMQ_REQ_RELAXED, 1);
    int portNumber = BindSocket(handlerSocket);
    std::cout << portNumber << std::endl;
    while(true) {
        std::cin >> action;
        if(action == "create") {
            int nodeId, parentId;
            std::string result;
            std::cin >> nodeId >> parentId;
            if(!childPid) {
                childPid = fork();
                if(childPid == -1) {
                    std::cout << "Unable to create process" << std::endl;
                    exit(-1);
                } else if(childPid == 0) {
                    CreateNode(nodeId, portNumber);
                } else {
                    parentId = 0;
                    childId = nodeId;
                    SendMessage(handlerSocket, "pid");
                    result = ReceiveMessage(handlerSocket);
                }
```

```
    } else {
        if(!calcs.PathTo(nodeId).empty()) {
            std::cout << "Error: Already exists" << std::endl;
            continue;
        }
        std::vector<int> path = calcs.PathTo(parentId);
        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 << " " << nodeId;
        SendMessage(handlerSocket, s.str());
        result = ReceiveMessage(handlerSocket);
    }

    if(result.substr(0, 2) == "Ok") {
        calcs.Insert(nodeId, parentId);
    }
    std::cout << result << std::endl;
} else if(action == "remove") {
    if(childPid == 0) {
        std::cout << "Error: Not found" << std::endl;
        continue;
    }
    int nodeId;
    std::cin >> nodeId;
    if(nodeId == childId) {
        SendMessage(handlerSocket, "kill");
        ReceiveMessage(handlerSocket);
        kill(childPid, SIGTERM);
        kill(childPid, SIGKILL);
        childId = 0;
        childPid = 0;
        std::cout << "Ok" << std::endl;
        calcs.Remove(nodeId);
        continue;
    }
    std::vector<int> path = calcs.PathTo(nodeId);
    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;
        }
        SendMessage(handlerSocket, s.str());
        std::string recieved = ReceiveMessage(handlerSocket);
        if(recieved.substr(0, 2) == "Ok") {
            calcs.Remove(nodeId);
        }
        std::cout << recieved << std::endl;
    } else if(action == "exec") {
        int nodeId;
        std::string subcommand;
        std::cin >> nodeId >> subcommand;
        std::vector<int> path = calcs.PathTo(nodeId);
        if(path.empty()) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        path.erase(path.begin());
        std::stringstream s;
        s << "exec " << subcommand << " " << path.size();
        for(int i : path) {
            s << " " << i;
        }
        SendMessage(handlerSocket, s.str());
        std::string received = ReceiveMessage(handlerSocket);
        std::cout << received << std::endl;
    } else if(action == "ping") {
        if(childPid == 0) {
            std::cout << "Error: Not found" << std::endl;
            continue;
        }
        int nodeId;
        std::cin >> nodeId;
        std::vector<int> path = calcs.PathTo(nodeId);
        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(!SendMessage(handlerSocket, s.str())) {
                received = "Node is unavailable";
            } else {
                received = ReceiveMessage(handlerSocket);
            }
            std::cout << received << std::endl;
        } else if(action == "exit") {
            SendMessage(handlerSocket, "kill");
            ReceiveMessage(handlerSocket);
            kill(childPid, SIGTERM);
            kill(childPid, SIGKILL);
            break;
        } else {
            std::cout << "Unknown command" << std::endl;
        }
        action.clear();
    }
    return 0;
}
```

socketRoutine.cpp

```
#include "socketRoutine.hpp"
#include <iostream>

bool SendMessage(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 ReceiveMessage(zmq::socket_t& socket) {
```

```
    zmq::message_t message;
    bool messageReceived;
    try {
        messageReceived = socket.recv(&message);
    } catch(...) {
        messageReceived = false;
    }

    std::string received(static_cast<char*>(message.data()), message.size());

    if(!messageReceived || received.empty()) {
        return "Error: Node is unavailable";
    } else {
        return received;
    }
}

int BindSocket(zmq::socket_t& socket) {
    int port = 30000;
    std::string portTemplate = "tcp://127.0.0.1:";
    while(true) {
        try {
            socket.bind(portTemplate + std::to_string(port));
            break;
        } catch(...) {
            port++;
        }
    }
    return port;
}

void CreateNode(int id, int portNumber) {
    char* arg0 = strdup("./calcNode");
    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("./calcNode", args);
}
```

socketRoutine.hpp

```
#pragma once

#include <zmq.hpp>
```

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

bool SendMessage(zmq::socket_t& socket, const std::string& message);

std::string ReceiveMessage(zmq::socket_t& socket);

int BindSocket(zmq::socket_t& socket);

void CreateNode(int id, int portNumber);
```

Демонстрация работы программы

```
sergey@sergey-RedmiBook-14:~/labs/OS/lab6/build$ ./main
30000
create 2 3
Ok: 31547
create 3 2
30001
Ok: 31551
create 4 2
30002
Ok: 31556
create 5 3
30003
Ok: 31559
ping 5
Ok: 1
ping 6
Error: Not found
ping 2
Ok: 1
exec 4 time
Ok: 4: 0
exec 4 start
Ok:4
exec 4 stop
Ok:4
exec 4 time
Ok: 4: 13051
exit
```

```
sergey@sergey-RedmiBook-14:~/labs/OS/lab6/build$ ps
  PID TTY          TIME CMD
 2295 pts/1    00:00:00 zsh
 2301 pts/1    00:00:01 bash
31567 pts/1    00:00:00 ps
```

Вывод

В ходе выполнения лабораторной работы я познакомился с очередью сообщений ZeroMQ, а конкретнее с сокетами, реализующими паттерн Request-Reply, а также получил опыт написания распределенных систем.