

Московский авиационный институт
Национальный исследовательский университет

Операционные системы
Кафедра 806

Лабораторные работы №6-7-8

Студент:	Минибаев Айдар
Группа:	М8О-301Б-18
Преподаватель:	Миронов Е.С.
Дата:	.
Подпись:	.

Москва 2020

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 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. Топология - первая. Тип вычислительной команды узлов - локальный таймер. Тип проверки узлов на доступность - heartbeat.

2. Метод решения

В топологии присутствуют две различные сущности: сервер и воркер.

Устройство сервера:

Сокеты:

1. push_worker - мапа сокетов для отправки сообщений определённым воркерам
2. pull_worker - сокет на получение сообщений от воркеров
3. heartbeat_worker - сокет на получение периодических пинг-сообщений от воркеров
4. main_socket - сокет на получение сообщений от других серверов
5. neighbours - мапа сокетов для отправки сообщений определённым серверам

Потоки:

1. sender - поток, осуществляющий взаимодействие с пользователем, работает с сокетами push_worker и neighbours
2. reciever - поток, получающий сообщения от воркеров, работает с сокетами pull_worker и neighbours
3. heartbeat - поток, мониторящий доступность воркеров, работает с сокетом heartbeat_worker
4. main_socket - поток, обеспечивающий общение сервера с другими серверами, работает с сокетами main_socket, neighbours

Устройство воркера:

Сокеты:

1. push_socket - сокет на отправку сообщений на сервер
2. pull_socket - сокет на получение сообщений от сервера

3. heartbeat_socket - сокет на отправку периодических пинг-сообщений на сервер

Потоки:

1. work - поток, осуществляющий взаимодействие с сервером, работает с сокетами push_socket и pull_socket

2. heartbeat - поток, отправляющий серверу отчёт о том, что он жив, работает с сокетом heartbeat_socket

3. Программная реализация

functions.hpp

```
#pragma once
#include <string>
#include <zconf.h>
#include "zmq.hpp"

bool send_message(zmq::socket_t& socket, const std::string& message_string);

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

std::string get_port_name(int port);

int bind_socket(zmq::socket_t& socket);

void create_worker(int id, int pull_port, int push_port, int heartbeat_port, int heartbeat);

std::string msg_to_string(zmq::message_t& message);

std::string get_prefix(std::istream& is);
```

functions.cpp

```
#include "functions.hpp"
#include <cstring>

bool send_message(zmq::socket_t& socket, const std::string& message_string) {
    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(), message_string.size());
    bool ok = false;
    try {
        ok = socket.send(message);
    }catch(...) {ok = false;}
    return ok;
}

std::string msg_to_string(zmq::message_t& message) {
    return std::string(static_cast<char*>(message.data()), message.size());
}

std::string recieve_message(zmq::socket_t& socket) {
    zmq::message_t message;
    bool ok;
```

```

    try {
        ok = socket.recv(&message);
    } catch (...) {
        ok = false;
    }
    std::string recieved_message(static_cast<char*>(message.data()), message.size());
    if (recieved_message.empty() || !ok) {
        return "Error: Worker is not available";
    }
    return recieved_message;
}

std::string get_port_name(int port) {
    return "tcp://127.0.0.1:" + std::to_string(port);
}

int bind_socket(zmq::socket_t& socket) {
    int port = 30000;
    while (true) {
        try {
            socket.bind(get_port_name(port));
            break;
        } catch (...) {
            port++;
        }
    }
    return port;
}

void create_worker(int id, int pull_port, int push_port, int heartbeat_port, int heartbeat) {
    char* arg1 = strdup((std::to_string(id)).c_str());
    char* arg2 = strdup((std::to_string(pull_port)).c_str());
    char* arg3 = strdup((std::to_string(push_port)).c_str());
    char* arg4 = strdup((std::to_string(heartbeat_port)).c_str());
    char* arg5 = strdup((std::to_string(heartbeat)).c_str());
    char* args[] = { "./worker", arg1, arg2, arg3, arg4, arg5, NULL };
    execv("./worker", args);
}

std::string get_prefix(std::istream& is) {
    std::string res;
    while (true) {
        char c = ' ';
        while (c == ' ')
            c = is.get();
        is.unget();
        if (c != 'p') {
            break;
        }
        std::string prt;
        std::string portNumber;
        is >> prt >> portNumber;
        res = res + "port " + portNumber + " ";
    }
    return res;
}

```

```

#pragma once

#include <chrono>

struct Timer {
    using milli = std::chrono::milliseconds;

    Timer() {
        start_ = std::chrono::system_clock::now();
        end_ = start_;
        timer_on_ = false;
    }

    void start() {
        timer_on_ = true;
        start_ = std::chrono::system_clock::now();
    }

    void stop() {
        end_ = std::chrono::system_clock::now();
        timer_on_ = false;
    }

    int time() {
        if (timer_on_)
            end_ = std::chrono::system_clock::now();
        return std::chrono::duration_cast<milli>(end_ - start_).count();
    }

    bool status() {
        return timer_on_;
    }
private:
    std::chrono::time_point<std::chrono::system_clock> start_;
    std::chrono::time_point<std::chrono::system_clock> end_;
    bool timer_on_;
};

```

worker.cpp

```

#include <iomanip>
#include <iostream>
#include <thread>
#include <memory>
#include <functional>

#include "functions.hpp"
#include "timer.hpp"

struct Worker {
    Worker(int id, int pull_port, int push_port, int heartbeat_port, int heartbeat_time) :
        context(1),
        pull_socket(context, ZMQ_PULL),
        push_socket(context, ZMQ_PUSH),
        heartbeat_socket(context, ZMQ_PUSH),

```

```

my_id(id),
heartbeat_time(heartbeat_time)
{
    pull_socket.connect(get_port_name(pull_port));
    push_socket.connect(get_port_name(push_port));
    heartbeat_socket.connect(get_port_name(heartbeat_port));
}

void Work() {
    std::string request_string;
    request_string = recieve_message(pull_socket);
    std::string message = "Ok:" + std::to_string(getpid());
    send_message(push_socket, message);

    Timer timer;
    while (true) {
        request_string = recieve_message(pull_socket);

        std::istringstream cmd_stream(request_string);
        std::string prefix;
        prefix = get_prefix(cmd_stream);
        std::string cmd;
        cmd_stream >> cmd;

        bool sent = false;
        if (cmd == "exec") {
            int id;
            std::string type;
            cmd_stream >> id >> type;
            std::string message = prefix + " Ok:" + std::to_string(id);

            if (type == "start") {
                timer.start();
                sent = send_message(push_socket, message);
            }
            else if (type == "stop") {
                timer.stop();
                sent = send_message(push_socket, message);
            }
            else if (type == "time") {
                int time = timer.time();
                message += ": " + std::to_string(time);
                sent = send_message(push_socket, message);
            }
        }
        else if (cmd == "heartbeat") {
            int time;
            cmd_stream >> time;
            heartbeat_time = time;
        }
        if (!sent && cmd != "heartbeat") {
            std::cout << "Something going wrong" << std::endl;
        }
    }
}

```

```

void Heartbeat() {
    Timer timer;
    int timer_zero = 0;
    timer.start();
    while (true) {
        while (!heartbeat_time)
            ;
        if (timer.time() - timer_zero < heartbeat_time) {
            continue;
        }
        timer.start();
        std::string message = std::to_string(my_id) + " is ok";
        send_message(heartbeat_socket, message);
    }
}

private:
    zmq::context_t context;
    zmq::socket_t pull_socket;
    zmq::socket_t push_socket;
    zmq::socket_t heartbeat_socket;
    int my_id;
    int heartbeat_time;
};

int main(int argc, char* argv[]) { // id and parent port number
    if (argc != 6) {
        std::cout << "Usage: " << argv[0] << " id pull_port push_port heartbeat_port heartbeat_time\n";
        return 1;
    }
    int id = std::stoi(argv[1]);
    int pull_port = std::stoi(argv[2]);
    int push_port = std::stoi(argv[3]);
    int heartbeat_port = std::stoi(argv[4]);
    int heartbeat_time = std::stoi(argv[5]);

    Worker worker{id, pull_port, push_port, heartbeat_port, heartbeat_time};
    std::thread work(std::bind(&Worker::Work, &worker));
    std::thread heartbeat(std::bind(&Worker::Heartbeat, &worker));

    work.join();
    heartbeat.join();
    return 0;
}

```

server.cpp

```

#include <iostream>
#include <string>
#include <map>
#include <set>
#include <thread>
#include <memory>

```

```

#include <functional>
#include <signal.h>

#include "zmq.hpp"
#include "functions.hpp"
#include "timer.hpp"

#define HEARTBEAT_LIVES 4
#define TIMEOUT 200

struct Server {
public:
    Server(int port) :
        context{1},
        main_socket{context, ZMQ_PULL},
        pull_worker{context, ZMQ_PULL},
        heartbeat_worker{context, ZMQ_PULL},
        my_port(port),
        heartbeat_time(0)
    {
        my_port = port;
        try {
            main_socket.bind(get_port_name(port));
        } catch(...) {
            std::cout << "Port is unavailable" << std::endl;
            my_port = bind_socket(main_socket);
            std::cout << "Your port is " << my_port << std::endl;
        }
        int linger = 0;
        pull_worker_port = bind_socket(pull_worker);
        heartbeat_worker.setsockopt(ZMQ_RCVTIMEO, TIMEOUT);
        heartbeat_worker.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
        heartbeat_worker_port = bind_socket(heartbeat_worker);
    }

    void Reciever () {
        while (true) {
            zmq::message_t message;
            bool ok;
            ok = pull_worker.recv(&message);

            std::string result = msg_to_string(message);
            std::istringstream is(result);
            std::string what_is;
            is >> what_is;
            if (what_is == "port") {
                int port;
                is >> port;
                std::string res_mes;
                while (is) {
                    std::string s;
                    is >> s;
                    res_mes += s + " ";
                }
                res_mes = "recv " + res_mes;
                send_message(neighbours[port], res_mes);
            }
        }
    }
};

```



```

    }
    else {
        std::cout << result << std::endl;
    }
}
}

```

```

void Heartbeater() {
    Timer timer;
    int timer_zero = 0;
    timer.start();
    while (true) {
        while (!heartbeat_time)
            ;
        if (timer.time() - timer_zero < heartbeat_time) {
            continue;
        }
        timer.start();
        for (auto& worker : push_worker) {
            if (push_lives[worker.first].first > -5)
                --push_lives[worker.first].first;
            bool ok;
            zmq::message_t message;
            try {
                ok = heartbeat_worker.recv(&message);
            } catch (...) {
                ok = false;
            }
            if (!ok) {
                continue;
            }
            std::string heartbeat_string = msg_to_string(message);
            std::istringstream heartbeat_stream(heartbeat_string);
            std::string str_id;
            heartbeat_stream >> str_id;
            push_lives[std::stoi(str_id)].first = HEARTBEAT_LIVES;
        }
        delete_zero_map(); // mark dead workers from both maps
    }
}

```

private:

```

void delete_zero_map() {
    std::map<int, std::pair<int, int>>::iterator live_cur, live_end;
    live_cur = push_lives.begin();
    live_end = push_lives.end();
    while (live_cur != live_end) {
        if (live_cur->second.first == 0){
            live_cur++;
        }
        else
            ++live_cur;
    }
}

```

public:

```

void Sender () {
    while (true) {
        std::string cmd;
        std::cin >> cmd;
        if (cmd == "create" || cmd == "remove" || cmd == "exec") {
            int id;
            try {
                std::cin >> id;
            } catch(...) {
                std::cout << "Wrong command" << std::endl;
                continue;
            }
            bool worker_here = (push_worker.find(id) != push_worker.end());
            std::pair<bool, int> checked = CheckNeibs(id);

            // запросы к своим воркерам
            if (worker_here && cmd == "create") {
                std::cout << "Error: Already exists" << std::endl;
            }
            else if (worker_here && cmd == "remove") {
                RemoveNode(id, "", std::cout);
            }
            else if (worker_here && cmd == "exec") {
                Exec(std::cin, id, std::cout, "");
            }

            // запросы к соседу
            else if (checked.first && cmd == "create") {
                std::cout << "Error: Already exists" << std::endl;
            }
            else if (checked.first && cmd == "remove") {
                std::string send_mes = "port " + std::to_string(my_port) +
                    " send remove " + std::to_string(id);
                if (!send_message(neighbours[checked.second], send_mes)) {
                    std::cout << "Server is unavailable" << std::endl;
                }
            }
            else if (checked.first && cmd == "exec") {
                std::string type;
                std::cin >> type;
                std::string send_mes = "port " + std::to_string(my_port) +
                    " send exec " + std::to_string(id) + " " + type;
                if (!send_message(neighbours[checked.second], send_mes)) {
                    std::cout << "Server is unavailable" << std::endl;
                }
            }
        }

        // некуда запрашивать
        else if (cmd == "create") {
            CreateNode(id, std::cout);
        }
        else if (cmd == "remove") {
            std::cout << "Error:" << id << ": Not found" << std::endl;
        }
        else if (cmd == "exec") {
            std::cout << "Error:" << id << ": Not found" << std::endl;
        }
    }
}

```

```

    }
}

else if (cmd == "union") {
    Union(std::cin, std::cout);
}
else if (cmd == "heartbeat") {
    Heartbeat(std::cin, std::cout);
}
else {
    std::cout << "Wrong command" << std::endl;
}
}
}

```

private:

// Проверить, нет ли такой ноды у соседей

```

std::pair<bool, int> CheckNeibs(int id) {
    int where_is = -1;
    int sent = false;
    for (auto& neib : neibWorkers) {
        if (neib.second.find(id) == neib.second.end())
            continue;
        where_is = neib.first;
        sent = true;
        break;
    }
    return std::pair{sent, where_is};
}

```

```

void CreateNode(int id, std::ostream& os) {
    int linger = 0;
    push_worker[id] = zmq::socket_t(context, ZMQ_PUSH);
    push_worker[id].setsockopt(ZMQ_RCVTIMEO, TIMEOUT);
    push_worker[id].setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
    int port = bind_socket(push_worker[id]);

    int pid = fork();
    if (pid == -1) {
        os << "Error: Unable to create worker" << std::endl;
        pid = 0;
        push_worker.erase(id);
        return ;
    }
    else if (pid == 0) {
        create_worker(id, port, pull_worker_port, heartbeat_worker_port, heartbeat_time);
    }
    else {
        push_lives[id].first = HEARTBEAT_LIVES;
        push_lives[id].second = pid;
        send_message(push_worker.at(id), "pid");
    }

    // рассказать о новорожденном
    std::string message = "send added " + std::to_string(my_port) + " " + std::to_string(id);
    for (auto& neib : neighbours) {

```

```

        send_message(neib.second, message);
    }
}

bool RemoveNode(int id, std::string prefix, std::ostream& os) {
    if (push_lives[id].first <= 0) {
        os << "Error: Node is unavailable" << std::endl;
        return false;
    }
    if (push_worker.find(id) == push_worker.end()) {
        os << "Error:" << id << ": Not found" << std::endl;
        return false;
    }
    kill(push_lives[id].second, SIGTERM);
    kill(push_lives[id].second, SIGKILL);
    push_lives.erase(id);
    push_worker.erase(id);

    // р а с с к а з а т ь  о б  у м е р ш е м
    std::string message = "send deleted " + std::to_string(my_port) + " " +
std::to_string(id);
    for (auto& neib : neighbours) {
        send_message(neib.second, message);
    }

    if (prefix.size() != 0) {
        std::istringstream is(prefix);
        std::string str;
        int port;
        is >> str >> port;
        send_message(neighbours[port], "Ok.");
    }
    else {
        os << "Ok." << std::endl;
    }
}

bool Exec(std::istream& is, int id, std::ostream& os, const std::string& prefix) {
    if (push_worker.find(id) == push_worker.end()) {
        os << "Error:" << id << ": Not found" << std::endl;
        return false;
    }
    if (push_lives[id].first <= 0) {
        os << "Error: Node is unavailable" << std::endl;
        return false;
    }
    std::string type;
    is >> type;
    std::string msg_string = prefix + "exec " + std::to_string(id) + " " + type;

    bool sent = send_message(push_worker.at(id), msg_string);
    return sent;
}

void Union(std::istream& is, std::ostream& os) {
    int port;

```

```

is >> port;
if (port == my_port) {
    std::cout << "Error: Other server port and the current port "
                << my_port << " is equal" << std::endl;
    return;
}
if (neighbours.find(port) != neighbours.end()) {
    std::cout << "We are already connected" << std::endl;
    return;
}

std::string send_union = "send union " + std::to_string(port);
for (auto& neib : neighbours) {
    send_message(neib.second, send_union);
}

// с к а з а т ь о с в о и х в о р к е р а х
std::string join_message = "join " + std::to_string(my_port);
for (auto& worker : push_worker) {
    join_message += " " + std::to_string(worker.first);
}

int linger = 0;
neighbours[port] = zmq::socket_t(context, ZMQ_PUSH);
neighbours[port].connect(get_port_name(port));
neighbours[port].setsockopt(ZMQ_SNDTIMEO, TIMEOUT);
neighbours[port].setsockopt(ZMQ_LINGER, &linger, sizeof(linger));

bool ok = send_message(neighbours[port], join_message);
if (!ok) {
    os << "Error: Message didn't send" << std::endl;
    return;
}
}

void Heartbeat(std::istream& is, std::ostream& os) {
    int time;
    is >> time;
    heartbeat_time = time;
    std::string message = "heartbeat " + std::to_string(time);
    for (auto& worker : push_worker) {
        send_message(worker.second, message);
    }
    os << "Ok." << std::endl;
}

public:
void Unionist() {
    while (true) {
        zmq::message_t message;
        while (true) {
            try {
                main_socket.recv(&message);
            } catch (...) { continue; }
            break;
        }
    }
}

```

```

std::stringstream is(msg_to_string(message));
std::string prefix = get_prefix(is);
std::string cmd;
is >> cmd;

if (cmd.substr(0, 2) == "Ok") {
    std::cout << cmd << std::endl;
}
else if (cmd == "join") {
    int port;
    is >> port;

    int linger = 0;
    neighbours[port] = zmq::socket_t(context, ZMQ_PUSH);
    neighbours[port].connect(get_port_name(port));
    neighbours[port].setsockopt(ZMQ_SNDTIMEO, TIMEOUT);
    neighbours[port].setsockopt(ZMQ_LINGER, &linger, sizeof(linger));

    neibWorkers[port].clear();
    while (is) {
        int id;
        is >> id;
        neibWorkers[port].insert(id);
        if (push_worker.find(id) != push_worker.end())
            RemoveNode(id, prefix, std::cout);
    }

    std::string send_union = "send union ";
    for (auto& neib : neighbours) {
        send_union += std::to_string(neib.first);
        send_message(neighbours[port], send_union);
    }

    std::string ans_mes = "send added " + std::to_string(my_port);
    for (auto& worker : push_worker) {
        ans_mes += " " + std::to_string(worker.first);
    }

    send_message(neighbours[port], ans_mes);
    if (!send_message(neighbours[port], "Ok."))
        std::cout << "He is dead" << std::endl;
}

else if (cmd == "send") {
    UnionistSend(is, prefix);
}
else if (cmd == "recv") {
    UnionistRecv(is, prefix);
}
}
}

```

private:

```

// получатель - рабочий
void UnionistSend(std::istream& is, const std::string& prefix) {

```

```

std::string cmd;
is >> cmd;
if (cmd == "exec") {
    int id;
    is >> id;
    Exec(is, id, std::cout, prefix);
}
else if (cmd == "remove") {
    int id;
    is >> id;
    RemoveNode(id, prefix, std::cout);
}
else if (cmd == "added") {
    int port;
    is >> port;
    neibWorkers[port].clear();
    while (is) {
        int id;
        is >> id;
        neibWorkers[port].insert(id);
    }
}
else if (cmd == "deleted") {
    int port;
    is >> port;
    while (is) {
        int id;
        is >> id;
        neibWorkers[port].erase(id);
    }
}
else if (cmd == "union") {
    int port;
    is >> port;
    if (port == my_port || neighbours.find(port) != neighbours.end()) {
        return;
    }
    // с к а з а т ь о с в о и х в о р к е р а х
    std::string join_message = "join " + std::to_string(my_port);
    for (auto& worker : push_worker) {
        join_message += " " + std::to_string(worker.first);
    }

    int linger = 0;
    neighbours[port] = zmq::socket_t(context, ZMQ_PUSH);
    neighbours[port].setsockopt(ZMQ_SNDTIMEO, TIMEOUT);
    neighbours[port].setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
    neighbours[port].connect(get_port_name(port));

    send_message(neighbours[port], join_message);
}
}

// п о л у ч а т е л ь з а к а з ч и к
void UnionistRecv(std::istream& is, const std::string& prefix) {
    if (prefix.size() == 0) {

```

```

        std::cout << is.rdbuf() << std::endl;
    }
    else {

        std::string what_is;
        is >> what_is;
        int port;
        is >> port;
        std::string sstr(std::istreambuf_iterator<char>(is), {});
        std::string res_mes = "recv " + sstr;
        send_message(neighbours[port], res_mes);
    }
}

private:
    zmq::context_t context;
    zmq::socket_t main_socket;
    zmq::socket_t pull_worker;
    zmq::socket_t heartbeat_worker;
    int my_port;
    int heartbeat_time;
    int pull_worker_port;
    int heartbeat_worker_port;
    std::map<int, zmq::socket_t> neighbours;
    std::map<int, std::set<int>> neibWorkers;
    std::map<int, zmq::socket_t> push_worker;
    std::map<int, std::pair<int, int>> push_lives; // колво хп и пид воркера

};

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cout << "Usage: " << argv[0] << " <port> " << std::endl;
        return 1;
    }
    int port = std::stoi(argv[1]);

    Server server{port};

    std::thread sender(std::bind(&Server::Sender, &server));
    std::thread reciever(std::bind(&Server::Reciever, &server));
    std::thread heartbeat(std::bind(&Server::Heartbeater, &server));
    std::thread main_socket(std::bind(&Server::Unionist, &server));

    sender.join();
    reciever.join();
    heartbeat.join();
    main_socket.join();
    return 0;
}

```

4. Тестирование

```

$ ./main 8081
create 1

```



```
Ok:25880
create 2
Ok:25886
create 3
Ok:25891
create 4
Ok:25896
```

Не прекращая работу первой программы откроем в новом терминале еще одну программу с портом 8082:

```
$ ./main 8082
create 5
Ok:25920
create 8
Ok:25926
create 9
Ok:25937
union 8081
Ok.
exec 1 start
Ok:1
exec 1 time
Ok:1: 6024
exec 1 stop
Ok:1
exec 1 time
Ok:1: 11784
Ok.
```

Не прекращая работу программ откроем в новом терминале еще одну программу с портом 8083:

```
$ ./main 8083
create 9
Ok:25976
create 10
Ok:25981
create 11
Ok:25986
union 8081
Ok.
Ok.
```

```
exec 6 start
Error:6: Not found
Wrong command
exec 8 start
Ok:8
exec 8 time
Ok:8: 6635
exec 8 stop
Ok:8
exec 8 time
Ok:8: 14578
exec 1 time
Ok:1: 11784
```

5. Выводы

В ходе выполнения лабораторной работы была изучена библиотека ZeroMQ, а также некоторые паттерны проектирования приложений, возникающие при её использовании. Приобрёл навыки в написании многопоточных приложений, связывании частей приложения в определённой топологии.