**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Операционные системы»

**Лабораторная работа № 6-8**

Тема: Управление серверами сообщений, применение отложенный вычислений, интеграция программных систем друг с другом

Студент: Волков

Группа: 80-207

Преподаватель: Миронов Е. С.

Дата:

Оценка:

1. **Постановка задачи**

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать два 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. При убийстве («kill -9») 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. Список основных поддерживаемых команд:

* Создание нового вычислительного узла;
* Удаление существующего вычислительного узла;
* Исполнение команды на вычислительном узле;
* Проверка доступности вычислительного узла.

*Вариант:*

Все вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел.

Исполнение команды – запуск таймера.

Команда проверки – проверка доступности конкретного узла.

1. **Описание программы**

Для начала был написан заголовочный файл под названием network, который помогал находить не занятые порты для подключения. Используются порты из диапазона динамических. Вероятность того, что они свободны – очень большая.

Связь между сокетами обеспечивается путем написания, как я называю, моста и самого передающего сокета. Мост нужен для того, чтобы можно было проверить живой ли сокет или нет. По второму сокету передается остальная нужная для дальнейшей работы информация. Также мост служит в некоторых местах в функциях перебиндивания сокета. Также стоит сказать, что мост сделан по системе req rep, а передатчик данных при помощи push pull.

Таймер я решил запускать при помощи execl, подключая к вычислимому сокету еще один сокет. Минус данного метода в том, что, если с таймером что-то случится, мы не сможем проверить живой он или нет. Когда таймер запускается, ему на вход приходит параметр в миллисекундах – на сколько нужно запустить таймер. Затем stop полностью останавливает таймер. time показывает время, оставшееся до срабатывания таймера. Когда таймер завершает работу, он говорит о том, что он завершил работу, затем посылает сигнал сокету, чтобы тот мог закрыть таймер. Такая сложная система сделана потому, что ожидание таймера происходит в отдельном потоке, привязанный к сокету таймера.

1. **Набор и результат тестов**

trying 49152 ok

trying 49153 ok

> *create 1 -1*

trying 49152

trying 49153

trying 49154

trying 49155

trying 49156

trying 49157

trying 49158

trying 49159

trying 49160

trying 49161

trying 49162

trying 49163 ok

trying 49164 ok

> trying 49165 ok

trying 49166 ok

**OK: 7324**

*create 2 1*

> trying 49167 ok

trying 49152

trying 49153

trying 49154

trying 49155

trying 49156

trying 49157

trying 49158

trying 49159

trying 49160

trying 49161

trying 49162

trying 49163

trying 49164

trying 49165

trying 49166

trying 49167

trying 49168 ok

trying 49169 ok

im in fork

trying 49170 ok

trying 49171 ok

*exec 1 start 10000*

im in client's start

> im timer

message sent. flag = 1

i get message

action = 6 true action = 6

**OK: timer 1 started**

*exec 2 start 2000*

im in client's start

> im timer

message sent. flag = 1

i get message

action = 6 true action = 6

**OK: timer 2 started**

**OK: timer out**

message was sent

im in client's done

client's done msg was sent

i get message

**OK: timer 2 closed**

**OK: timer out**

message was sent

im in client's done

client's done msg was sent

i get message

**OK: timer 1 closed**

*ping 1*

**OK: 1**

> *ping 2*

**OK: 2**

> *exit*

1. **Листинг программы**

Для удобства программа была разделена на несколько файлов.

server\_fin.cpp

#include <unistd.h>  
#include <pthread.h>  
#include <zmqpp/zmqpp.hpp>  
#include <iostream>  
#include <list>  
#include <string>  
#include <signal.h>  
#include <sstream>  
  
  
#include "network.h"  
  
struct nodes{  
 int id;  
 std::list<nodes> bros;  
};  
  
using line\_t = std::pair <nodes, std::pair<//changed  
 std::pair < zmqpp::socket, std::string>, std::pair<zmqpp::socket, std::string>>>;  
std::list <line\_t> network;  
  
class node\_coord {  
private:  
 std::pair <std::list<line\_t>::iterator, nodes> data;  
  
public:  
 node\_coord(std::list<line\_t>::iterator &i1,nodes &i2) : data(i1, i2) {}  
  
 node\_coord(std::list<line\_t>::iterator &&i1,nodes &&i2) : data(i1, i2) {}  
  
 node\_coord(node\_coord &other) : data(other.data) {}  
  
 node\_coord(node\_coord &&other) : data(other.data) {}  
  
 std::list<line\_t>::iterator &line() {  
 return data.first;  
 }  
  
 nodes &node() {  
 return data.second;  
 }  
  
 int &id() {  
 return data.second.id;  
 }  
  
 int &parent() {  
 // main parent  
 return data.first->first.id;  
 }  
  
 zmqpp::socket &out\_sock() {  
 return data.first->second.first.first;  
 }  
  
 zmqpp::socket &ping\_sock() {  
 return data.first->second.second.first;  
 }  
  
 std::string &out\_port() {  
 return data.first->second.first.second;  
 }  
  
 std::string &ping\_port() {  
 return data.first->second.second.second;  
 }  
};  
  
zmqpp::context context;  
  
std::list<nodes>::iterator find\_node(nodes& b\_it, int id){  
 auto result = b\_it.bros.end();  
 for(auto iter = b\_it.bros.begin(); iter != b\_it.bros.end() && b\_it.bros.size() != 0; ++iter){  
 if(iter->id == id){  
 return iter;  
 }  
  
 result = find\_node(\*iter,id);  
 if(result != iter->bros.end() && result->id == id){  
 return result;  
 }  
  
 result = b\_it.bros.end();  
 }  
 return result;  
}  
  
node\_coord find\_node(int id){  
 for(auto b\_it = network.begin(); b\_it != network.end(); ++b\_it){  
 if(b\_it->first.id == id){  
 return {b\_it, b\_it->first};  
 }  
  
 auto result = find\_node(b\_it->first,id);  
 if(result != b\_it->first.bros.end() && result->id == id){  
 return {b\_it,\*result};  
 }  
 }  
 return {network.end(), nodes()};  
}  
  
void\* result\_waiter(void\* arg){  
 zmqpp::socket\* in\_sock = reinterpret\_cast<zmqpp::socket\*> (arg);  
 zmqpp::message msg;  
 int act, rid;  
  
 while(\*in\_sock){  
 in\_sock->receive(msg);  
 msg >> rid >> act;  
  
 switch(static\_cast<action>(act)){  
 case action::test: {  
 std::cout << "Hello " << rid << std::endl;  
 break;  
 }  
  
 case action::fork: {  
 int pid;  
 msg >> pid;  
 std::cout << "OK: " << pid << std::endl;  
 break;  
 }  
  
 case action::exit: {  
 node\_coord node = find\_node(rid);  
 }  
  
 default: {}  
 }  
 }  
 return NULL;  
}  
  
bool ping(zmqpp::socket &ping\_sock, std::string &ping\_port, int id) {  
 int packet[2] = {id, 1};  
 size\_t length = 2 \* sizeof(int);  
 if (!ping\_sock.send\_raw(reinterpret\_cast<char \*>(packet), length, zmqpp::socket::dont\_wait)) {  
 return false;  
 }  
 if (!ping\_sock.receive\_raw(reinterpret\_cast<char \*>(packet), length)) {  
 ping\_sock.close();  
 ping\_sock = zmqpp::socket(context, zmqpp::socket\_type::req);  
 ping\_sock.set(zmqpp::socket\_option::receive\_timeout, 1000);  
 ping\_sock.connect(host + ping\_port);  
 return false;  
 }  
 return (packet[1] == 1) ? true : false;  
}  
  
int main(){  
 zmqpp::socket in\_sock(context, zmqpp::socket\_type::pull);  
 std::string in\_port = std::to\_string(try\_bind(in\_sock));  
  
 zmqpp::socket bridge(context,zmqpp::socket\_type::rep);  
 std::string bridge\_port = std::to\_string(try\_bind(bridge));  
  
  
 //проверка создание потока  
 pthread\_t result\_waiter\_id;  
 check(pthread\_create(&result\_waiter\_id, NULL,result\_waiter, (void\*) &in\_sock),  
 -1,  
 "pthread\_create failed");  
  
 if(pthread\_detach(result\_waiter\_id) != 0){  
 perror("detach error");  
 }  
  
 std::string command;  
 while(std::cout << "> " && std::cin >> command && command != "exit"){  
 if(command == "create"){  
 int id, parent;  
 std::cin >> id >> parent;  
  
 if (id == -1 || find\_node(id).line() != network.end()) {  
 std::cerr << "Error: Already exists" << std::endl;  
 continue;  
 }  
  
 if (parent == -1) {  
 //создание узла родителя  
 network.push\_back({nodes(),  
 std::make\_pair(std::make\_pair(zmqpp::socket(context, zmqpp::socket\_type::push), ""),  
 std::make\_pair(zmqpp::socket(context, zmqpp::socket\_type::req),  
 ""))});  
 network.back().first.id = id;  
 node\_coord node(--network.end(), network.back().first);  
  
 int pid = fork();  
 check(pid, -1, "fork error");  
 if (pid == 0) {  
 check(execl("client", "client", std::to\_string(id).c\_str(), std::to\_string(-1).c\_str(),  
 bridge\_port.c\_str(), NULL), -1, "execl error")  
 }  
  
 zmqpp::message ports;  
 bridge.receive(ports);  
  
 ports >> node.out\_port();  
 node.out\_sock().connect(host + node.out\_port());  
  
 ports >> node.ping\_port();  
 node.ping\_sock().connect(host + node.ping\_port());  
 node.ping\_sock().set(zmqpp::socket\_option::receive\_timeout, 1000);  
  
 ports.pop\_back();  
 ports.pop\_back();  
 ports << in\_port;  
 bridge.send(ports);  
 }  
  
 else {  
 node\_coord parent\_node = find\_node(parent);  
 if (parent\_node.line() == network.end()) {  
 std::cerr << "Error: Parent not found" << std::endl;  
 continue;  
 }  
  
 if (!ping(parent\_node.ping\_sock(), parent\_node.ping\_port(), parent)) {  
 std::cerr << "Error: Parent is unavailable" << std::endl;  
 continue;  
 }  
  
 parent\_node.line()->first.bros.push\_back({id,std::list<nodes>()});  
  
 zmqpp::message msg;  
 msg << parent << static\_cast<int>(action::fork) << id;  
 parent\_node.out\_sock().send(msg);  
 }  
 }  
  
 if(command == "exec"){  
 int id;  
 std::cin >> id;  
  
 node\_coord node = find\_node(id);  
  
 if (node.line() == network.end()) {  
 std::cerr << "Error: Not found" << std::endl;  
 continue;  
 }  
  
 if (!ping(node.ping\_sock(), node.ping\_port(), id)) {  
 std::cerr << "Error: Node is unavailable" << std::endl;  
 continue;  
 }  
  
 std::string act;  
 std::cin >> act;  
 zmqpp::message msg;  
 msg << id;  
 if(act == "start"){  
 std::cout << "im in client's start" << std::endl;  
 size\_t wait;  
 std::cin >> wait;  
 msg << static\_cast<int>(action::start) << wait;  
 node.out\_sock().send(msg);  
 }  
 else if(act == "stop"){  
 msg << static\_cast<int>(action::stop);  
 node.out\_sock().send(msg);  
 }  
 else if(act == "time"){  
 msg << static\_cast<int>(action::time);  
 node.out\_sock().send(msg);  
 }  
 }  
  
 if(command == "ping"){  
 int id;  
 std::cin >> id;  
  
 auto node = find\_node(id);  
 if(ping(node.ping\_sock(), node.ping\_port(), id)){  
 std::cout << "OK: " << id << std::endl;  
 }  
 else{  
 std::cerr << "Error: node is unavailable" << std::endl;  
 }  
 }  
  
 if(command == "test"){  
 int pid = fork();  
 if(pid == 0){  
 execl("client", "client", bridge\_port.c\_str(), NULL);  
 }  
 zmqpp::message msg;  
 bridge.receive(msg);  
 int id;  
 msg >> id;  
 std::cout << id << std::endl;  
  
 msg.pop\_back();  
 msg.pop\_back();  
  
 msg << "im done";  
  
 bridge.send(msg);  
 }  
 }  
  
 return 0;  
}  
  
  
// topology - 2  
// commands - 3  
// ping - 2 - ping id

network.h

#pragma once  
  
#include <zmqpp/zmqpp.hpp>  
#include <string>  
  
#define check(VAL, BADVAL, MSG) if (VAL == BADVAL) { perror(MSG); exit(1); }  
  
const std::string host = "tcp://127.0.0.1:";  
  
int get\_port() {  
 static unsigned int port(49152);  
 static zmqpp::context context;  
 static zmqpp::socket socket(context, zmqpp::socket\_type::pull);  
 while (true) {  
 std::cout << "trying " << port;  
 try {  
 socket.bind(host + std::to\_string(port));  
 }  
 catch (zmqpp::zmq\_internal\_exception &ex) {  
 ++port;  
 std::cout << std::endl;  
 continue;  
 }  
 socket.unbind(host + std::to\_string(port));  
 std::cout << " ok" << std::endl;  
 return port++;  
 }  
}  
  
int try\_bind(zmqpp::socket &socket) {  
 static unsigned int port(49152);  
 while (true) {  
 try {  
 std::cout << "trying " << port;  
 socket.bind(host + std::to\_string(port));  
 }  
 catch (zmqpp::zmq\_internal\_exception &ex) {  
 ++port;  
 std::cout << std::endl;  
 continue;  
 }  
 std::cout << " ok" << std::endl;  
 return port++;  
 }  
}  
  
enum class action : int {  
 fork, exit, unbind\_front, unbind\_back, rebind\_front, rebind\_back, start,stop,time,test,done  
};

client.cpp

#include <unistd.h>  
#include <pthread.h>  
#include <zmqpp/zmqpp.hpp>  
#include <iostream>  
#include <string>  
#include <memory>  
#include <algorithm>  
#include <signal.h>  
#include <map>  
  
#include "network.h"  
//#define DEBUG  
  
int id = -1, next\_id = -1, prev\_id = -1;  
std::unique\_ptr<zmqpp::socket> front\_in(nullptr), front\_out(nullptr), front\_ping(nullptr),  
 back\_in(nullptr), back\_out(nullptr), back\_ping(nullptr);  
std::string back\_in\_port, back\_out\_port, back\_ping\_port, front\_ping\_port;  
zmqpp::context context;  
  
  
void\* ping(void\*) {  
 int packet[2];  
 size\_t length = 2 \* sizeof(int);  
 while (\*front\_ping) {  
 front\_ping->receive\_raw(reinterpret\_cast<char \*>(packet), length);  
 // std::cout << "received ping " << packet[0] << " " << packet[1] << std::endl;  
 if (packet[0] != id) {  
 // std::cout << "sending forward" << std::endl;  
 if (back\_ping.get() != nullptr) {  
 if (!back\_ping->send\_raw(reinterpret\_cast<char \*>(packet), length, zmqpp::socket::dont\_wait)) {  
 packet[1] = 0;  
 } else if (!back\_ping->receive\_raw(reinterpret\_cast<char \*>(packet), length)) {  
 packet[1] = 0;  
 back\_ping->close();  
 back\_ping.reset(new zmqpp::socket(context, zmqpp::socket\_type::req));  
 back\_ping->set(zmqpp::socket\_option::receive\_timeout, 1000);  
 back\_ping->connect(host + back\_ping\_port);  
 }  
 } else {  
 front\_ping->close();  
 front\_ping.reset(new zmqpp::socket(context, zmqpp::socket\_type::rep));  
 front\_ping->bind(host + front\_ping\_port);  
 continue;  
 }  
 }  
 // std::cout << "sending back" << std::endl;  
 front\_ping->send\_raw(reinterpret\_cast<char \*>(packet), length);  
 }  
 return NULL;  
}  
  
  
#ifdef DEBUG  
[[noreturn]] void\* awake(void\* arg){  
 bool\* flag = static\_cast<bool\*>(arg);  
 while(true) {  
 sleep(5);  
 std::cout << "im making flag true" << std::endl;  
 \*flag = true;  
 }  
}  
#endif  
  
  
void\* back\_to\_front(void\*) {  
 zmqpp::message msg;  
 //std::cout << "im in back\_to\_front" << std::endl; //debug  
#ifdef DEBUG  
 bool flag = true;  
 pthread\_t flag\_awake;//debug  
 check(pthread\_create(&flag\_awake, NULL, awake, (void\*) &flag),  
 -1,  
 "pthread\_create error");  
 if(pthread\_detach(flag\_awake) != 0){  
 perror("detach error");  
 }  
#endif  
  
 while (back\_in.get() == nullptr || \*back\_in) {  
#ifdef DEBUG  
 if(flag){  
 std::cout << "im in back\_to\_front" << std::endl; //debug  
 std::cout << back\_in\_port << std::endl;  
 }  
#endif  
  
 if (back\_in.get() == nullptr) {  
#ifdef DEBUG  
 flag = false;  
#endif  
 continue;  
 }  
  
 back\_in->receive(msg);  
 // std::cout << "sending back" << std::endl;  
 int rid;  
 msg >> rid;  
 if (rid != id) {  
 front\_out->send(msg);  
 continue;  
 }  
  
 int act;  
 msg >> act;  
 switch (static\_cast<action>(act)) {  
 case action::rebind\_back: {  
 msg >> next\_id;  
 back\_out->disconnect(host + back\_out\_port);  
 msg >> back\_out\_port;  
 back\_out->connect(host + back\_out\_port);  
  
 back\_in->unbind(host + back\_in\_port);  
 msg >> back\_in\_port;  
 back\_in->bind(host + back\_in\_port);  
  
 back\_ping->disconnect(host + back\_ping\_port);  
 msg >> back\_ping\_port;  
 back\_ping->connect(host + back\_ping\_port);  
 break;  
 }  
  
 case action::unbind\_back: {  
 back\_in.reset(nullptr);  
 back\_out.reset(nullptr);  
 back\_ping.reset(nullptr);  
 next\_id = -1;  
 break;  
 }  
  
 default: {}  
 }  
 }  
 return NULL;  
}  
  
  
int main(int argc, char\* argv[]){  
 assert(argc >= 4);  
 id = strtoul(argv[1], nullptr, 10);  
 prev\_id = strtoul(argv[2], nullptr, 10);  
 std::string bridge\_port(argv[3]);  
 // std::cout << getpid() << " " << id << std::endl;  
  
 zmqpp::message msg;  
  
 zmqpp::socket bridge(context, zmqpp::socket\_type::req);  
 bridge.connect(host + bridge\_port);  
  
 front\_in.reset(new zmqpp::socket(context, zmqpp::socket\_type::pull));  
 std::string front\_in\_port = std::to\_string(try\_bind(\*front\_in));  
  
 front\_ping.reset(new zmqpp::socket(context, zmqpp::socket\_type::rep));  
 front\_ping\_port = std::to\_string(try\_bind(\*front\_ping));  
  
 msg << front\_in\_port << front\_ping\_port;  
 bridge.send(msg);  
  
 bridge.receive(msg);  
  
 std::string front\_out\_port;  
 msg >> front\_out\_port;  
 front\_out.reset(new zmqpp::socket(context, zmqpp::socket\_type::push));  
 front\_out->connect(host + front\_out\_port);  
  
 bridge.disconnect(host + bridge\_port);  
 bridge.close();  
 bridge = zmqpp::socket(context, zmqpp::socket\_type::rep);  
 bridge\_port = std::to\_string(try\_bind(bridge));  
  
 {  
 zmqpp::message ans;  
 ans << id << static\_cast<int>(action::fork) << getpid();  
 front\_out->send(ans);  
#ifdef DEBUG  
 std::cout << "sending the msg"<< std::endl;  
#endif  
 }  
  
 pthread\_t ping\_id;  
 check(pthread\_create(&ping\_id, NULL, ping, NULL), -1, "pthread\_create error");  
 if (pthread\_detach(ping\_id) != 0) {  
 perror("detach error");  
 }  
  
 pthread\_t back\_to\_front\_id;  
 check(pthread\_create(&back\_to\_front\_id, NULL, back\_to\_front, NULL), -1, "pthread\_create error");  
 if (pthread\_detach(back\_to\_front\_id) != 0) {  
 perror("detach error");  
 }  
  
 int act;  
 int tid;  
  
 zmqpp::socket timer\_sock(context, zmqpp::socket\_type::push);  
 std::string timer\_port = std::to\_string(try\_bind(timer\_sock));  
 bool timer\_started = false;  
 while (true) {  
 if (!front\_in->receive(msg)) {  
 perror("");  
 }  
  
 msg >> tid;  
  
 if (tid != id) {  
 // std::cout << "sending forward" << std::endl;  
 back\_out->send(msg);  
 continue;  
 }  
  
 msg >> act;  
  
 switch (static\_cast<action>(act)) {  
 case action::fork: {  
 int cid;  
 msg >> cid;  
  
  
 if (next\_id == -1) {  
 next\_id = cid;  
  
 int pid = fork();  
 check(pid, -1, "fork error");  
 if (pid == 0) {  
 check(execl("client", "client", std::to\_string(cid).c\_str(), std::to\_string(id).c\_str(),  
 bridge\_port.c\_str(), NULL), -1, "execl error");  
 }  
  
  
 back\_in.reset(new zmqpp::socket(context, zmqpp::socket\_type::pull));  
 back\_in\_port = std::to\_string(try\_bind(\*back\_in));  
  
 zmqpp::message ports;  
 bridge.receive(ports);  
  
 ports >> back\_out\_port >> back\_ping\_port;  
 ports.pop\_back();  
 ports.pop\_back();  
  
 back\_out.reset(new zmqpp::socket(context, zmqpp::socket\_type::push));  
 back\_ping.reset(new zmqpp::socket(context, zmqpp::socket\_type::req));  
 back\_ping->set(zmqpp::socket\_option::receive\_timeout, 1000);  
  
 back\_out->connect(host + back\_out\_port);  
 back\_ping->connect(host + back\_ping\_port);  
  
 ports << back\_in\_port;  
 bridge.send(ports);  
 }  
 std::cout << "im in fork" << std::endl;  
 break;  
 }  
  
 case action::done: {  
 std::cout << "im in client's done" << std::endl;  
 zmqpp::message exit\_msg;  
 exit\_msg << tid << static\_cast<int>(action::exit);  
 timer\_sock.send(exit\_msg);  
 std::cout << "client's done msg was sent" << std::endl;  
 timer\_started = false;  
 break;  
 }  
  
 case action::start: {  
 if(timer\_started){  
 std::cout << "timer already started" << std::endl;  
 break;  
 }  
  
 int pid;  
 pid = fork();  
 check(pid,-1,"fork error");  
 if(pid == 0){  
 check(execl("timer", "timer",timer\_port.c\_str(), front\_in\_port.c\_str(), NULL),  
 -1,  
 "execl error");  
 }  
 size\_t wait;  
 msg >> wait;  
  
 zmqpp::message timer\_msg;  
 timer\_msg << tid << static\_cast<int>(action::start) << wait;  
 timer\_sock.send(timer\_msg);  
  
 timer\_started = true;  
 std::cout << "message sent. flag = " << timer\_started << std::endl;  
  
 break;  
 }  
  
 case action::time: {  
 std::cout << "im in client's time" << std::endl;  
 if(!timer\_started){  
 std::cout << "You had not started timer" << std::endl;  
 break;  
 }  
 std::cout << "trying to send msg" << std::endl;  
  
 zmqpp::message timer\_msg;  
 timer\_msg << tid << static\_cast<int>(action::time) ;  
 std::cout << "msg = " << timer\_sock.send(timer\_msg) << std::endl;  
 break;  
 }  
  
 case action::stop: {  
 if(!timer\_started){  
 std::cout << "Please start timer" << std::endl;  
 break;  
 }  
  
 zmqpp::message timer\_msg;  
 timer\_msg << tid << static\_cast<int>(action::stop) ;  
 timer\_sock.send(timer\_msg);  
 timer\_started = false;  
 break;  
 }  
  
 default:{}  
 }  
 }  
  
  
 return 0;  
}

timer.cpp

#include <unistd.h>  
#include <pthread.h>  
#include <zmqpp/zmqpp.hpp>  
#include <iostream>  
#include <string>  
#include <memory>  
#include <algorithm>  
#include <signal.h>  
#include <map>  
  
#include "network.h"  
  
zmqpp::context context;  
  
class myTimer\_t{  
private:  
 size\_t time\_start;  
 size\_t timer\_wait{};  
  
public:  
 myTimer\_t(const size\_t new\_timer\_wait): timer\_wait(new\_timer\_wait), time\_start(clock()){}  
 myTimer\_t(): time\_start(0){}  
 ~myTimer\_t()=default;  
  
 void Start(const size\_t new\_timer\_wait){  
 timer\_wait = new\_timer\_wait;  
 time\_start = (size\_t)clock();  
 }  
  
 size\_t Time(){  
 return time\_start + timer\_wait - (size\_t)clock();  
 }  
};  
  
  
zmqpp::socket req\_sock(context, zmqpp::socket\_type::push);  
std::string req\_port;  
  
  
int tid;  
  
void\* awake (void\* arg){  
// std::cout << "im awake for "/\* << std::flush\*/;  
 size\_t\* wait = reinterpret\_cast<size\_t\*>(arg);  
// std::cout << \*wait << std::endl;  
 usleep((\*wait) \* 1000);  
 std::cout << "OK: timer out" << std::endl;  
 zmqpp::message done\_msg;  
 done\_msg << tid << static\_cast<int>(action::done);  
 req\_sock.send(done\_msg);  
 std::cout << "message was sent" << std::endl;  
 return NULL;  
}  
  
int main(int argc, char\* argv[]){  
 std::string timer\_port = argv[1];  
 req\_port = argv[2];  
 zmqpp::socket timer\_sock(context, zmqpp::socket\_type::pull);  
 timer\_sock.connect(host + timer\_port);  
 req\_sock.connect(host + req\_port);  
 myTimer\_t myTimer;  
 pthread\_t awake\_timer;  
 std::cout << "im timer" << std::endl;  
  
 while(true){  
 int act;  
 zmqpp::message msg;  
  
 timer\_sock.receive(msg);  
 msg >> tid >> act;  
 std::cout << "i get message" << std::endl << "action = " << act << " true action = "  
 << static\_cast<int>(action::start) << std::endl;  
  
 switch (static\_cast<action>(act)) {  
 case action::start: {  
 size\_t wait;  
 msg >> wait;  
// std::cout << "im in start with wait = " << wait << std::endl;  
 myTimer.Start(wait);  
 check(pthread\_create(&awake\_timer, NULL, awake, (void \*) &wait),  
 -1,  
 "pthread\_create error");  
  
 if(pthread\_detach(awake\_timer) != 0){  
 perror("pthread\_detach error");  
 }  
 std::cout << "OK: timer " << tid << " started" << std::endl;  
 break;  
 }  
  
 case action::stop: {  
 req\_sock.disconnect(host + req\_port);  
 req\_sock.close();  
 timer\_sock.disconnect(host + timer\_port);  
 timer\_sock.close();  
 pthread\_cancel(awake\_timer);  
 std::cout << "OK: Timer " << tid << " stopped" << std::endl;  
 exit(0);  
 }  
  
 case action::time: {  
 std::cout << "im in timer's time" << std::endl;  
 auto to\_wait = myTimer.Time();  
 std::cout << "OK: " << to\_wait << " left" << std::endl;  
 break;  
 }  
  
 case action::exit: {  
 req\_sock.disconnect(host + req\_port);  
 req\_sock.close();  
 timer\_sock.disconnect(host + timer\_port);  
 timer\_sock.close();  
 std::cout << "OK: timer " << tid << " closed" << std::endl;  
 exit(0);  
 return 0;  
 }  
  
 default: {}  
 }  
  
 }  
}

1. **Выводы**

Благодаря проделанной работе я научился создавать сервер при помощи библиотеки ZMQ. Работа с данной библиотекой научила меня в первую очередь усидчивости. Я теперь понимаю, как чувствуют себя программисты, которые целыми днями сидят и разбираются в тонкостях работы той или иной библиотеки.

После проделывания данной работы я также почувствовал себе силы в разработке backend направления. Также я понял, что подобные вещи, как сервер, удобнее всего делать на языке javascript + node js. В данном языке нет никаких усложнений работы с MQ подобными библиотеками таких, как создание fork и exec функций.