**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Операционные системы»

**Лабораторная работа № 6**

Тема: Управлении серверами сообщений,  
Применение отложенных вычислений,  
Интеграция программных систем друг с другом.

Студент: Тимофеев А.В.

Группа: 80-207

Преподаватель: Миронов Е. С.

Дата:

Оценка:

**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. 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

Формат команды: create id [parent]

id – целочисленный идентификатор нового вычислительного узла

parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено

введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода:

«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла

«Error: Already exists» - вычислительный узел с таким идентификатором уже существует

«Error: Parent not found» - нет такого родительского узла с таким идентификатором

«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не

удается связаться

«Error: [Custom error]» - любая другая обрабатываемая ошибка

Пример:

> create 10 5

Ok: 3128

Вариант:

24:

2 Аналогично топологии 2, но узлы находятся в дереве общего вида. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

4 поиск подстроки в строке

3 Формат команды: heartbit time

Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от

узла нет сигнала в течении 4\*time миллисекунд, то должна выводится пользователю строка:

«Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

**Алгоритм решения**

Связь между вычислительными узлами будем поддерживать с помощью ZMQ\_PAIR. Это наиболее подходящая связь для жесткого контроля небольшого количества узлов, на мой взгляд. При инициализации установим время ожидания ZMQ\_SNDTIMEO и ZMQ\_RECVTIMEO, чтобы предусмотреть случай, когда дочерний процесс был убит. Для обмена информацией будем использовать специальную структуру node\_token\_t, в которой есть перечислимое поле actions. Вычислительные узлы обрабатывают каждое сообщение: если идентификатор сообщения не совпадает с идентификатором узла, то он отправляет сообщение дальше и ждёт ответа снизу. Каждый вычислительный узел имеет отдельный поток для вычислений и свою очередь вычислений. Чтобы получить результат вычислений обратно, нужно запросить их от вычислительного узла. Такой подход необходим, потому что неизвестно, сколько нужно ждать результат от узла. Для поиска подстроки в строке я использовал алгоритм Кнута-Морриса-Пратта с препроцессингом через Z-функцию строки.

**Листинг программы**

**calculation\_node.cpp**

#include <list>

#include <pthread.h>

#include <queue>

#include <tuple>

#include <unistd.h>

#include "search.hpp"

#include "zmq\_std.hpp"

const std::string SENTINEL\_STR = "$";

long long node\_id;

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

std::queue< std::pair<std::string, std::string> > calc\_queue;

std::queue< std::list<unsigned int> > done\_queue;

void\* thread\_func(void\*) {

while (1) {

pthread\_mutex\_lock(&mutex);

while (calc\_queue.empty()) {

pthread\_cond\_wait(&cond, &mutex);

}

std::pair<std::string, std::string> cur = calc\_queue.front();

calc\_queue.pop();

pthread\_mutex\_unlock(&mutex);

if (cur.first == SENTINEL\_STR and cur.second == SENTINEL\_STR) {

break;

} else {

std::vector<unsigned int> res = KMPStrong(cur.first, cur.second);

std::list<unsigned int> res\_list;

for (const unsigned int & elem : res) {

res\_list.push\_back(elem);

}

pthread\_mutex\_lock(&mutex);

done\_queue.push(res\_list);

pthread\_mutex\_unlock(&mutex);

}

}

return NULL;

}

int main(int argc, char\*\* argv) {

int rc;

assert(argc == 2);

node\_id = std::stoll(std::string(argv[1]));

void\* node\_parent\_context = zmq\_ctx\_new();

void\* node\_parent\_socket = zmq\_socket(node\_parent\_context, ZMQ\_PAIR);

rc = zmq\_connect(node\_parent\_socket, ("tcp://localhost:" + std::to\_string(PORT\_BASE + node\_id)).c\_str());

assert(rc == 0);

long long child\_id = -1;

void\* node\_context = NULL;

void\* node\_socket = NULL;

pthread\_t calculation\_thread;

rc = pthread\_mutex\_init(&mutex, NULL);

assert(rc == 0);

rc = pthread\_cond\_init(&cond, NULL);

assert(rc == 0);

rc = pthread\_create(&calculation\_thread, NULL, thread\_func, NULL);

assert(rc == 0);

std::string pattern, text;

bool flag\_sentinel = true;

node\_token\_t\* info\_token = new node\_token\_t({info, getpid(), getpid()});

zmq\_std::send\_msg\_dontwait(info\_token, node\_parent\_socket);

std::list<unsigned int> cur\_calculated;

bool has\_child = false;

bool awake = true;

bool calc = true;

while (awake) {

node\_token\_t token;

zmq\_std::recieve\_msg(token, node\_parent\_socket);

node\_token\_t\* reply = new node\_token\_t({fail, node\_id, node\_id});

if (token.action == back) {

if (token.id == node\_id) {

if (calc) {

pthread\_mutex\_lock(&mutex);

if (done\_queue.empty()) {

reply->action = exec;

} else {

cur\_calculated = done\_queue.front();

done\_queue.pop();

reply->action = success;

reply->id = getpid();

}

pthread\_mutex\_unlock(&mutex);

calc = false;

} else {

if (cur\_calculated.size() > 0) {

reply->action = success;

reply->id = cur\_calculated.front();

cur\_calculated.pop\_front();

} else {

reply->action = exec;

calc = true;

}

}

} else {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (zmq\_std::send\_recieve\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == bind and token.parent\_id == node\_id) {

/\*

\* Bind could be recieved when parent created node

\* and this node should bind to parent's child

\*/

zmq\_std::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str());

assert(rc == 0);

has\_child = true;

child\_id = token.id;

node\_token\_t\* token\_ping = new node\_token\_t({ping, child\_id, child\_id});

node\_token\_t reply\_ping({fail, child\_id, child\_id});

if (zmq\_std::send\_recieve\_wait(token\_ping, reply\_ping, node\_socket) and reply\_ping.action == success) {

reply->action = success;

}

} else if (token.action == create) {

if (token.parent\_id == node\_id) {

if (has\_child) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

}

zmq\_std::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str());

assert(rc == 0);

int fork\_id = fork();

if (fork\_id == 0) {

rc = execl(NODE\_EXECUTABLE\_NAME, NODE\_EXECUTABLE\_NAME, std::to\_string(token.id).c\_str(), NULL);

assert(rc != -1);

return 0;

} else {

bool ok = true;

node\_token\_t reply\_info({fail, token.id, token.id});

ok = zmq\_std::recieve\_msg\_wait(reply\_info, node\_socket);

if (reply\_info.action != fail) {

reply->id = reply\_info.id;

reply->parent\_id = reply\_info.parent\_id;

}

if (has\_child) {

node\_token\_t\* token\_bind = new node\_token\_t({bind, token.id, child\_id});

node\_token\_t reply\_bind({fail, token.id, token.id});

ok = zmq\_std::send\_recieve\_wait(token\_bind, reply\_bind, node\_socket);

ok = ok and (reply\_bind.action == success);

}

if (ok) {

/\* We should check if child has connected to this node \*/

node\_token\_t\* token\_ping = new node\_token\_t({ping, token.id, token.id});

node\_token\_t reply\_ping({fail, token.id, token.id});

ok = zmq\_std::send\_recieve\_wait(token\_ping, reply\_ping, node\_socket);

ok = ok and (reply\_ping.action == success);

if (ok) {

reply->action = success;

child\_id = token.id;

has\_child = true;

} else {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

}

}

}

} else if (has\_child) {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (zmq\_std::send\_recieve\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == ping) {

if (token.id == node\_id) {

reply->action = success;

} else if (has\_child) {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (zmq\_std::send\_recieve\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == destroy) {

if (has\_child) {

if (token.id == child\_id) {

bool ok = true;

node\_token\_t\* token\_down = new node\_token\_t({destroy, node\_id, child\_id});

node\_token\_t reply\_down({fail, child\_id, child\_id});

ok = zmq\_std::send\_recieve\_wait(token\_down, reply\_down, node\_socket);

/\* We should get special reply from child \*/

if (reply\_down.action == destroy and reply\_down.parent\_id == child\_id) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

has\_child = false;

child\_id = -1;

} else if (reply\_down.action == bind and reply\_down.parent\_id == node\_id) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

zmq\_std::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + reply\_down.id)).c\_str());

assert(rc == 0);

child\_id = reply\_down.id;

node\_token\_t\* token\_ping = new node\_token\_t({ping, child\_id, child\_id});

node\_token\_t reply\_ping({fail, child\_id, child\_id});

if (zmq\_std::send\_recieve\_wait(token\_ping, reply\_ping, node\_socket) and reply\_ping.action == success) {

ok = true;

}

}

if (ok) {

reply->action = success;

}

} else if (token.id == node\_id) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

has\_child = false;

reply->action = bind;

reply->id = child\_id;

reply->parent\_id = token.parent\_id;

awake = false;

} else {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (zmq\_std::send\_recieve\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.id == node\_id) {

/\* Special message to parent \*/

reply->action = destroy;

reply->parent\_id = node\_id;

reply->id = node\_id;

awake = false;

}

} else if (token.action == exec) {

if (token.id == node\_id) {

char c = token.parent\_id;

if (c == SENTINEL) {

if (flag\_sentinel) {

std::swap(text, pattern);

} else {

pthread\_mutex\_lock(&mutex);

if (calc\_queue.empty()) {

pthread\_cond\_signal(&cond);

}

calc\_queue.push({pattern, text});

pthread\_mutex\_unlock(&mutex);

text.clear();

pattern.clear();

}

flag\_sentinel = flag\_sentinel ^ 1;

} else {

text = text + c;

}

reply->action = success;

} else if (has\_child) {

node\_token\_t\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (zmq\_std::send\_recieve\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

}

zmq\_std::send\_msg\_dontwait(reply, node\_parent\_socket);

}

if (has\_child) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

}

rc = zmq\_close(node\_parent\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_parent\_context);

assert(rc == 0);

pthread\_mutex\_lock(&mutex);

if (calc\_queue.empty()) {

pthread\_cond\_signal(&cond);

}

calc\_queue.push({SENTINEL\_STR, SENTINEL\_STR});

pthread\_mutex\_unlock(&mutex);

rc = pthread\_join(calculation\_thread, NULL);

assert(rc == 0);

rc = pthread\_cond\_destroy(&cond);

assert(rc == 0);

rc = pthread\_mutex\_destroy(&mutex);

assert(rc == 0);

}

**control.cpp**

#include <unistd.h>

#include <vector>

#include <iostream>

#include <pthread.h>

#include <time.h>

#include "topology.hpp"

#include "zmq\_std.hpp"

using node\_id\_type = long long;

int main() {

int rc;

topology\_t<node\_id\_type, node\_t> control\_node;

std::vector< std::pair<void\*, void\*> > childs;

std::vector<node\_id\_type> ids;

std::string s;

node\_id\_type id;

while (std::cin >> s >> id) {

if (s == "create") {

node\_id\_type parent\_id;

std::cin >> parent\_id;

if (parent\_id == -1) {

void\* new\_context = NULL;

void\* new\_socket = NULL;

zmq\_std::init\_pair\_socket(new\_context, new\_socket);

rc = zmq\_bind(new\_socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + id)).c\_str());

assert(rc == 0);

int fork\_id = fork();

if (fork\_id == 0) {

rc = execl(NODE\_EXECUTABLE\_NAME, NODE\_EXECUTABLE\_NAME, std::to\_string(id).c\_str(), NULL);

assert(rc != -1);

return 0;

} else {

bool ok = true;

node\_token\_t reply\_info({fail, id, id});

ok = zmq\_std::recieve\_msg\_wait(reply\_info, new\_socket);

node\_token\_t\* token = new node\_token\_t({ping, id, id});

node\_token\_t reply({fail, id, id});

ok = zmq\_std::send\_recieve\_wait(token, reply, new\_socket);

if (ok and reply.action == success) {

childs.push\_back(std::make\_pair(new\_context, new\_socket));

control\_node.insert(id);

std::cout << "OK: " << reply\_info.id << std::endl;

ids.push\_back(id);

} else {

rc = zmq\_close(new\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(new\_context);

assert(rc == 0);

}

}

} else if (control\_node.find(parent\_id) == -1) {

std::cout << "Error: Not found" << std::endl;

} else {

if (control\_node.find(id) != -1) {

std::cout << "Error: Already exists" << std::endl;

} else {

int ind = control\_node.find(parent\_id);

node\_token\_t\* token = new node\_token\_t({create, parent\_id, id});

node\_token\_t reply({fail, id, id});

if (zmq\_std::send\_recieve\_wait(token, reply, childs[ind].second) and reply.action == success) {

std::cout << "OK: " << reply.id << std::endl;

ids.push\_back(id);

control\_node.insert(parent\_id, id);

} else {

std::cout << "Error: Parent is unavailable" << std::endl;

}

}

}

} else if (s == "remove") {

int ind = control\_node.find(id);

//std::cout << control\_node << std::endl;

if (ind != -1) {

node\_token\_t\* token = new node\_token\_t({destroy, id, id});

node\_token\_t reply({fail, id, id});

bool ok = zmq\_std::send\_recieve\_wait(token, reply, childs[ind].second);

if (reply.action == destroy and reply.parent\_id == id) {

rc = zmq\_close(childs[ind].second);

assert(rc == 0);

rc = zmq\_ctx\_term(childs[ind].first);

assert(rc == 0);

std::vector< std::pair<void\*, void\*> >::iterator it = childs.begin();

while (ind--) {

++it;

}

childs.erase(it);

} else if (reply.action == bind and reply.parent\_id == id) {

rc = zmq\_close(childs[ind].second);

assert(rc == 0);

rc = zmq\_ctx\_term(childs[ind].first);

assert(rc == 0);

zmq\_std::init\_pair\_socket(childs[ind].first, childs[ind].second);

rc = zmq\_bind(childs[ind].second, ("tcp://\*:" + std::to\_string(PORT\_BASE + reply.id)).c\_str());

assert(rc == 0);

}

if (ok) {

control\_node.erase(id);

for (auto iter = ids.begin(); iter != ids.end(); ++iter){

if (\*iter == id){

ids.erase(iter);

break;

}

}

std::cout << "OK" << std::endl;

} else {

std::cout << "Error: Node is unavailable" << std::endl;

}

} else {

std::cout << "Error: Not found" << std::endl;

}

} else if (s == "heartbit") {

int ind = 0;

int tmp = 0;

double start = 0, end = 0;

//std::cout << "start = " << start <<std::endl;

start = clock();

//std::cout << "start = " << start <<std::endl;

for(;;){

//std::cout << "start = " << start <<std::endl;

for (auto iter = ids.begin(); iter != ids.end(); ++iter){

ind = control\_node.find(\*iter);

if (ind == -1){

std::cout << "Error: Not found" << \*iter << std::endl;

}else{

node\_token\_t\* token = new node\_token\_t({ping, \*iter, \*iter});

node\_token\_t reply({fail, \*iter, \*iter});

if (zmq\_std::send\_recieve\_wait(token, reply, childs[ind].second) and reply.action == success) {

std::cout << "OK:" << \*iter << std::endl;

} else {

std::cout << "Heartbit: node" << \*iter << "is unavailable now" << std::endl;

}

}

++tmp;

}

//std::cout << "end = " << end <<std::endl;

end = clock();

//std::cout << "end = " << end <<std::endl;

if(start > 4\*id or end > 4\*id){

break;

}

}

} else if (s == "back") {

int ind = control\_node.find(id);

if (ind != -1) {

node\_token\_t\* token = new node\_token\_t({back, id, id});

node\_token\_t reply({fail, id, id});

if (zmq\_std::send\_recieve\_wait(token, reply, childs[ind].second)) {

if (reply.action == success) {

node\_token\_t\* token\_back = new node\_token\_t({back, id, id});

node\_token\_t reply\_back({fail, id, id});

std::vector<unsigned int> calculated;

while (zmq\_std::send\_recieve\_wait(token\_back, reply\_back, childs[ind].second) and reply\_back.action == success) {

calculated.push\_back(reply\_back.id);

}

if (calculated.empty()) {

std::cout << "OK: " << reply.id << " : -1" << std::endl;

} else {

std::cout << "OK: " << reply.id << " : ";

for (size\_t i = 0; i < calculated.size() - 1; ++i) {

std::cout << calculated[i] << ", ";

}

std::cout << calculated.back() << std::endl;

}

} else {

std::cout << "Error: No calculations to back" << std::endl;

}

} else {

std::cout << "Error: Node is unavailable" << std::endl;

}

} else {

std::cout << "Error: Not found" << std::endl;

}

} else if (s == "exec") {

std::string pattern, text;

std::cin >> pattern >> text;

int ind = control\_node.find(id);

if (ind != -1) {

bool ok = true;

std::string text\_pattern = pattern + SENTINEL + text + SENTINEL;

for (size\_t i = 0; i < text\_pattern.size(); ++i) {

node\_token\_t\* token = new node\_token\_t({exec, text\_pattern[i], id});

node\_token\_t reply({fail, id, id});

if (!zmq\_std::send\_recieve\_wait(token, reply, childs[ind].second) or reply.action != success) {

ok = false;

break;

}

}

if (ok) {

std::cout << "OK" << std::endl;

} else {

std::cout << "Error: Node is unavailable" << std::endl;

}

} else {

std::cout << "Error: Not found" << std::endl;

}

}

}

for (size\_t i = 0; i < childs.size(); ++i) {

rc = zmq\_close(childs[i].second);

assert(rc == 0);

rc = zmq\_ctx\_term(childs[i].first);

assert(rc == 0);

}

}

**search.cpp**

#include "search.hpp"

std::vector<unsigned int> PrefixFunction(const std::string & s) {

unsigned int n = s.size();

std::vector<unsigned int> p(n);

for (unsigned int i = 1; i < n; ++i) {

p[i] = p[i - 1];

while (p[i] > 0 and s[i] != s[p[i]]) {

p[i] = p[p[i] - 1];

}

if (s[i] == s[p[i]]) {

++p[i];

}

}

return p;

}

std::vector<unsigned int> KMPWeak(const std::string & pattern, const std::string & text) {

std::vector<unsigned int> p = PrefixFunction(pattern);

unsigned int m = pattern.size();

unsigned int n = text.size();

unsigned int i = 0;

std::vector<unsigned int> ans;

if (m > n) {

return ans;

}

while (i < n - m + 1) {

unsigned int j = 0;

while (j < m and pattern[j] == text[i + j]) {

++j;

}

if (j == m) {

ans.push\_back(i);

} else {

if (j > 0 and j > p[j - 1]) {

i = i + j - p[j - 1] - 1;

}

}

++i;

}

return ans;

}

std::vector<unsigned int> ZFunction(const std::string & s) {

unsigned int n = s.size();

std::vector<unsigned int> z(n);

unsigned int l = 0, r = 0;

for (unsigned int i = 1; i < n; ++i) {

if (i <= r) {

z[i] = std::min(z[i - l], r - i);

}

while (i + z[i] < n and s[i + z[i]] == s[z[i]]) {

++z[i];

}

if (i + z[i] > r) {

l = i;

r = i + z[i];

}

}

return z;

}

std::vector<unsigned int> StrongPrefixFunction(const std::string & s) {

std::vector<unsigned int> z = ZFunction(s);

unsigned int n = s.size();

std::vector<unsigned int> sp(n);

for (unsigned int i = n - 1; i > 0; --i) {

sp[i + z[i] - 1] = z[i];

}

return sp;

}

std::vector<unsigned int> KMPStrong(const std::string & pattern, const std::string & text) {

std::vector<unsigned int> p = StrongPrefixFunction(pattern);

unsigned int m = pattern.size();

unsigned int n = text.size();

unsigned int i = 0;

std::vector<unsigned int> ans;

if (m > n) {

return ans;

}

while (i < n - m + 1) {

unsigned int j = 0;

while (j < m and pattern[j] == text[i + j]) {

++j;

}

if (j == m) {

ans.push\_back(i);

} else {

if (j > 0 and j > p[j - 1]) {

i = i + j - p[j - 1] - 1;

}

}

++i;

}

return ans;

}

**search.hpp**

#ifndef SEARCH\_HPP

#define SEARCH\_HPP

#include <string>

#include <vector>

std::vector<unsigned int> PrefixFunction(const std::string & s);

std::vector<unsigned int> KMPWeak(const std::string & pattern, const std::string & text);

std::vector<unsigned int> ZFunction(const std::string & s);

std::vector<unsigned int> StrongPrefixFunction(const std::string & s);

std::vector<unsigned int> KMPStrong(const std::string & pattern, const std::string & text);

#endif /\* SEARCH\_HPP \*/

**topology.hpp**

#ifndef TOPOLOGY\_HPP

#define TOPOLOGY\_HPP

#include <iostream>

#include <list>

#include <memory>

#include <utility>

class node\_t{

private:

std::pair<long long,std::list<node\_t>> id\_node;

using iterator = typename std::list<node\_t>::iterator;

public:

explicit node\_t() noexcept : id\_node() {}

node\_t(const long long & elem){

id\_node.first = elem;

}

bool insert(const long long & parent, const long long & elem) {

for (iterator it1 = id\_node.second.begin(); it1 != id\_node.second.end(); ++it1) {// проходим по списку детей

if ((\*it1).get\_id() == parent) { // если в этом списке найден родитель, вставляем к нему в список ребенка

node\_t new\_node(elem);

(\*it1).id\_node.second.push\_back(new\_node);

return true;

}else if(!(\*it1).node\_empty()){// если это не родитель, проверяем есть ли у данного узла дети,

// если есть проверяем и вставляем в их списки

if((\*it1).insert(parent,elem)){

return true;

}

}

}

return false;

}

bool insert(const long long & elem) {

this->id\_node.second.push\_back(elem);

return true;

}

bool erase(const long long & elem) {

for (iterator it1 = this->id\_node.second.begin(); it1 != this->id\_node.second.end(); ++it1) {// проходим по списку детей

if ((\*it1).get\_id() == elem) {

if (!((\*it1).node\_empty())){// если список детей узла на который указывает итератор не пуст то передаем этот узел в last\_elem\_del

long long id1 = (\*it1).last\_elem\_del();

(\*it1).set\_id(id1);

return true;

}else{

this->id\_node.second.erase(it1);

return true;

}

}else if(not((\*it1).node\_empty())){

if ((\*it1).erase(elem)){

return true;

}

}

}

return false;

}

long long get\_id(){

return this->id\_node.first;

}

void set\_id(const long long & elem){

this->id\_node.first = elem;

}

long long last\_elem\_del(){

if (this->id\_node.second.front().node\_empty()){// если полученый узел имеет детей а его первый ребенок не имеет детей

// то запоминаем id первого ребенка и удаляем его

long long id1 = this->id\_node.second.front().get\_id();

this->id\_node.second.pop\_front();

return id1;

}else{

return this->id\_node.second.front().last\_elem\_del();

}

}

long long find(const long long & elem) {

long long ind = 0;

for (iterator it1 = this->id\_node.second.begin(); it1 != this->id\_node.second.end(); ++it1) {

if ((\*it1).get\_id() == elem){

return ind;

}else if(!(\*it1).node\_empty()){

if ((\*it1).find(elem) == 0){

return ind;

}

}

}

return -1;

}

bool node\_empty(){

return this->id\_node.second.empty();

}

~node\_t(){}

};

template<class L, class N> // L - long long N - node t

class topology\_t {

private:

std::shared\_ptr<N> root;

size\_t container\_size;

public:

explicit topology\_t() noexcept : root(nullptr), container\_size(0) {}

~topology\_t() {}

bool insert(const L & parent, const L & elem) {

if (this->root != nullptr){

if((\*root).get\_id() == parent){

return (\*root).insert(elem);

}else{

return (\*root).insert(parent, elem);

}

}

return false;

}

void insert(const L & elem) {

if (root == nullptr){

N new\_node(elem);

root = std::make\_shared<N>(new\_node);

}else{

std::cout << "ERROR: Root exists" << std::endl;

}

}

long long find(const L & elem) {

long long ind = 0;

if (this->root != nullptr){

if((\*root).get\_id() == elem){

return ind;

}else{

return (\*root).find(elem);

}

}

return -1;

}

bool erase(const L & elem) {

if (this->root != nullptr){

if(((\*root).get\_id() == elem) and (not((\*root).node\_empty()))){// если удаляем корень

long long id1 = (\*root).last\_elem\_del(); // находим последний эл-т и удаляем, его id копируем в корень

(\*root).set\_id(id1);

return true;

}else if(((\*root).get\_id() == elem) and ((\*root).node\_empty())){

this->root = nullptr;

return true;

}else if(((\*root).get\_id() != elem) and (not((\*root).node\_empty()))){

return (\*root).erase(elem);

}

}

return false;

}

};

#endif

**zmq\_std.hpp**

#ifndef ZMQ\_STD\_HPP

#define ZMQ\_STD\_HPP

#include <assert.h>

#include <errno.h>

#include <string.h>

#include <string>

#include "zmq.h"

const char\* NODE\_EXECUTABLE\_NAME = "calculation";

const char SENTINEL = '$';

const int PORT\_BASE = 8000;

const int WAIT\_TIME = 1000;

enum actions\_t {

fail = 0,

success = 1,

create = 2,

destroy = 3,

bind = 4,

ping = 5,

exec = 6,

info = 7,

back = 8

};

struct node\_token\_t {

actions\_t action;

long long parent\_id, id;

};

namespace zmq\_std {

void init\_pair\_socket(void\* & context, void\* & socket) {

int rc;

context = zmq\_ctx\_new();

socket = zmq\_socket(context, ZMQ\_PAIR);

rc = zmq\_setsockopt(socket, ZMQ\_RCVTIMEO, &WAIT\_TIME, sizeof(int));

assert(rc == 0);

rc = zmq\_setsockopt(socket, ZMQ\_SNDTIMEO, &WAIT\_TIME, sizeof(int));

assert(rc == 0);

}

template<class T>

void recieve\_msg(T & reply\_data, void\* socket) {

int rc = 0;

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

rc = zmq\_msg\_recv(&reply, socket, 0);

assert(rc == sizeof(T));

reply\_data = \*(T\*)zmq\_msg\_data(&reply);

rc = zmq\_msg\_close(&reply);

assert(rc == 0);

}

template<class T>

void send\_msg(T\* token, void\* socket) {

int rc = 0;

zmq\_msg\_t message;

zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T));

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL);

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, 0);

assert(rc == sizeof(T));

}

template<class T>

bool send\_msg\_dontwait(T\* token, void\* socket) {

int rc;

zmq\_msg\_t message;

zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T));

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL);

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, ZMQ\_DONTWAIT);

if (rc == -1) {

zmq\_msg\_close(&message);

return false;

}

assert(rc == sizeof(T));

return true;

}

template<class T>

bool recieve\_msg\_wait(T & reply\_data, void\* socket) {

int rc = 0;

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

rc = zmq\_msg\_recv(&reply, socket, 0);

if (rc == -1) {

zmq\_msg\_close(&reply);

return false;

}

assert(rc == sizeof(T));

reply\_data = \*(T\*)zmq\_msg\_data(&reply);

rc = zmq\_msg\_close(&reply);

assert(rc == 0);

return true;

}

/\* Returns true if T was successfully queued on the socket \*/

template<class T>

bool send\_msg\_wait(T\* token, void\* socket) {

int rc;

zmq\_msg\_t message;

zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T));

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL);

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, 0);

if (rc == -1) {

zmq\_msg\_close(&message);

return false;

}

assert(rc == sizeof(T));

return true;

}

/\*

\* Returns true if socket successfully queued

\* message and recieved reply

\*/

template<class T>

bool send\_recieve\_wait(T\* token\_send, T & token\_reply, void\* socket) {

if (send\_msg\_wait(token\_send, socket)) {

if (recieve\_msg\_wait(token\_reply, socket)) {

return true;

} else {

return false;

}

} else {

return false;

}

}

}

#endif /\* ZMQ\_STD\_HPP \*/

**Тесты и протокол исполнения**

dude@dude:~/Рабочий стол/OC/mylaba6/src$ ./control | cat >protocol

create 1 -1

create 10 1

create 5 1

create 7 5

create 3 1

create 6 5

create 8 7

create 2 1

create 4 3

create 9 8

create 11 3

heartbit 100

remove 5

remove 2

remove 3

remove 8

remove 7

remove 1

remove 4

remove 6

remove 11

remove 10

remove 9

dude@dude:~/Рабочий стол/OC/mylaba6/src$ cat protocol

OK: 2756

OK: 2762

OK: 2768

OK: 2776

OK: 2784

OK: 2792

OK: 2800

OK: 2808

OK: 2816

OK: 2824

OK: 2832

OK:1

OK:10

OK:5

OK:7

OK:3

OK:6

OK:8

OK:2

OK:4

OK:9

OK:11

OK

OK

OK

OK

OK

OK

OK

OK

OK

OK

OK

dude@dude:~/Рабочий стол/OC/mylaba6/src$ ./control | cat >protocol

create 1 -1

create 2 1

create 3 2

create 4 2

create 5 2

create 6 3

create 7 3

create 8 1

exec 4 qwer qwertyqwer

exec 2 qwer qwertyqwer

exec 5 qwer qwertyqwer

exec 1 qwer qwertyqwer

exec 3 qwer qwertyqwer

exec 7 qwe qwertyqwerqwe

exec 6 qwe qwertyqwerqwe

exec 8 qwe qwertyqwerqwe

back 1

back 2

back 4

back 5

back 3

back 8

back 7

back 6

heartbit 100

remove 4

remove 1

remove 3

remove 2

remove 5

remove 8

remove 7

remove 6

dude@dude:~/Рабочий стол/OC/mylaba6/src$ cat protocol

OK: 3262

OK: 3268

OK: 3274

OK: 3280

OK: 3288

OK: 3296

OK: 3302

OK: 3310

OK

OK

OK

OK

OK

OK

OK

OK

OK: 3262 : 0, 6

OK: 3268 : 0, 6

OK: 3280 : 0, 6

OK: 3288 : 0, 6

OK: 3274 : 0, 6

OK: 3310 : 0, 6, 10

OK: 3302 : 0, 6, 10

OK: 3296 : 0, 6, 10

OK:1

OK:2

OK:3

OK:4

OK:5

OK:6

OK:7

OK:8

OK

OK

OK

OK

OK

OK

OK

OK

**Вывод**:

В этой лабораторной работы я изучил основы работы с очередями сообщений ZeroMQ и реализовал программу с использованием этой библиотеки. Для достижения отказоустойчивости я взял ZMQ\_PAIR, потому что ZMQ\_PAIR лучше всего подходит для подобной работы. Самым сложным в работе оказались удаление узла из сети и вставка узла между другими узлами. При таких операциях нужно было переподключать сокеты на вычислительных узлах. Очереди сообщений используются для взаимодействия нескольких машин в одной большой сети. Опыт работы с ZeroMQ пригодится мне при настройке собственной системы распределённых вычислений.

**Список литературы**

1. Таненбаум Э., Бос Х. Современные операционные системы. – 4-е изд. – СПб.: Издательский дом «Питер», 2018.
2. ZeroMQ API – URL: http://api.zeromq.org/3-2:\_start
3. Программирование сокетов в Linux – URL: https://www.rsdn.org/article/unix/sockets.xml