Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

Студент: Речинская Ангелина Юрьевна

Группа: М80-206Б-20

Вариант: 37

Преподаватель: Соколов Андрей Алексеевич

Дата: 29.12.2021

Оценка: 5

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Цель работы**

Целью является приобретение практических навыков в:

· Управлении серверами сообщений (№ 6)

· Применение отложенных вычислений (№ 7)

· Интеграция программных систем друг с другом (№ 8)

**Задание**

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

Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.

Удаление существующего вычислительного узла

Формат команды: remove id id – целочисленный идентификатор удаляемого вычислительного узла

Формат вывода:

«Ok» - успешное удаление

«Error: Not found» - вычислительный узел с таким идентификатором не найден

«Error: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

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

Пример:

> remove 10

Ok

Примечание: при удалении узла из топологии его процесс должен быть завершен и работоспособность вычислительной сети не должна быть нарушена.

Исполнение команды на вычислительном узле

Формат команды: exec id [params]

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Формат вывода:

«Ok:id: [result]», где result – результат выполненной команды

«Error:id: Not found» - вычислительный узел с таким идентификатором не найден

«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error:id: [Custom error]» - любая другая обрабатываемая ошибка Пример: Можно найти в описании конкретной команды, определенной вариантом задания.

Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Топология 1

Все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Набора команд 2 (локальный целочисленный словарь)

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name

Пример:

> exec 10 MyVar

Ok:10: 'MyVar' not found

> exec 10 MyVar 5

Ok:10

> exec 12 MyVar

Ok:12: 'MyVar' not found

> exec 10 MyVar

Ok:10: 5

> exec 10 MyVar 7

Ok:10

> exec 10 MyVar

Ok:10: 7

Примечания: Можно использовать std:map.

Команда проверки 1

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

Пример:

> pingall

Ok: -1 // Все узлы доступны

> pingall

Ok: 7;10;15 // узлы 7, 10, 15 — недоступны

**Листинг программы**

**#include <string.h>**

**#include <iostream>**

**#include <unistd.h>**

**#include <vector>**

**#include <algorithm>**

**#include <csignal>**

**#include <map>**

**#include <string>**

**#include "wrap\_zmq.h"**

**#include "socket.h"**

**#include "client.h"**

**#include <signal.h>**

**#include <sys/types.h>**

**using namespace std;**

**Client::Client(int new\_id, string parent\_endpoint, int new\_parent\_id, map<string,int> &new\_map){**

**id = new\_id;**

**parent\_id = new\_parent\_id;**

**context = create\_zmq\_ctx();**

**string endpoint = create\_endpoint(EndpointType::CHILD\_PUB\_NEXT, getpid());**

**child\_publisher\_next = new Socket(context, SocketType::PUBLISHER, endpoint);**

**endpoint = create\_endpoint(EndpointType::PARENT\_PUB, getpid());**

**parent\_publisher = new Socket(context, SocketType::PUBLISHER, endpoint);**

**parent\_subscriber = new Socket(context, SocketType::SUBSCRIBER, parent\_endpoint);**

**map1=new\_map;**

**next\_subscriber = nullptr;**

**terminated = false;**

**}**

**Client::~Client(){**

**if(terminated) return;**

**terminated = true;**

**try{**

**delete child\_publisher\_next;**

**delete parent\_publisher;**

**delete parent\_subscriber;**

**map1.clear();**

**if(next\_subscriber)**

**delete next\_subscriber;**

**destroy\_zmq\_ctx(context);**

**} catch(runtime\_error& err){**

**cout << "Server wasn't stopped " << err.what() << endl;**

**}**

**}**

**bool& Client::get\_status(){**

**return terminated;**

**}**

**void Client::send\_up(Message msg){**

**msg.to\_up = true;**

**parent\_publisher->send(msg);**

**}**

**void Client::send\_down(Message msg){**

**msg.to\_up = false;**

**child\_publisher\_next->send(msg);**

**}**

**int Client::get\_id(){**

**return id;**

**}**

**map<string,int> &Client::get\_map(){**

**return map1;**

**}**

**int Client::add\_child(int new\_id){**

**pid\_t pid = fork();**

**if(pid == -1) throw runtime\_error("Can not fork.");**

**if(!pid){**

**string endpoint;**

**endpoint = child\_publisher\_next->get\_endpoint();**

**execl("client", "client", to\_string(new\_id).data(),**

**endpoint.data(), to\_string(id).data(), nullptr);**

**throw runtime\_error("Can not execl.");**

**}**

**string endpoint = create\_endpoint(EndpointType::PARENT\_PUB, pid);**

**size\_t timeout = 10000;**

**next\_subscriber = new Socket(context, SocketType::SUBSCRIBER, endpoint);**

**zmq\_setsockopt(next\_subscriber->get\_socket(), ZMQ\_RCVTIMEO, &timeout, sizeof(timeout));**

**return pid;**

**}**

**void process\_msg(Client& client, Message msg){**

**switch(msg.command){**

**case CommandType::ERROR:**

**throw runtime\_error("Error message received.");**

**case CommandType::RETURN:{**

**msg.get\_to\_id() = SERVER\_ID;**

**client.send\_up(msg);**

**break;**

**}**

**case CommandType::CREATE\_CHILD:{**

**msg.get\_create\_id() = client.add\_child(msg.get\_create\_id());**

**msg.get\_to\_id() = SERVER\_ID;**

**client.send\_up(msg);**

**break;**

**}**

**case CommandType::REMOVE\_CHILD:{**

**if(msg.to\_up){**

**client.send\_up(msg);**

**break;**

**}**

**if(msg.to\_id != client.get\_id() && msg.to\_id != UNIVERSAL\_MSG){**

**client.send\_down(msg);**

**break;**

**}**

**msg.get\_to\_id() = PARENT\_SIGNAL;**

**msg.get\_create\_id() = client.get\_id();**

**client.send\_up(msg);**

**msg.get\_to\_id() = UNIVERSAL\_MSG;**

**client.send\_down(msg);**

**client.~Client();**

**throw invalid\_argument("Exiting child...");**

**break;**

**}**

**case CommandType::EXEC\_CHILD: {**

**string t;**

**for(int i=0;i<msg.k;i++){**

**t=t+msg.name[i];**

**}**

**bool error=false;**

**if(msg.change==false){**

**if((client.get\_map()).find(t)!=(client.get\_map()).end()){**

**msg.val=client.get\_map()[t];**

**} else {**

**error=true;**

**}**

**} else {**

**client.get\_map()[t]=msg.val;**

**}**

**msg.error=error;**

**msg.get\_to\_id() = SERVER\_ID;**

**msg.get\_create\_id() = client.get\_id();**

**client.send\_up(msg);**

**break;**

**}**

**default:**

**throw runtime\_error("Undefined command.");**

**}**

**}**

**Client\* client\_ptr = nullptr;**

**void TerminateByUser(int) {**

**if (client\_ptr != nullptr) {**

**client\_ptr->~Client();**

**}**

**cout << to\_string(getpid()) + " Terminated by user" << endl;**

**exit(0);**

**}**

**int main (int argc, char const \*argv[])**

**{**

**if(argc != 4){**

**cout << "-1" << endl;**

**return -1;**

**}**

**try{**

**if (signal(SIGINT, TerminateByUser) == SIG\_ERR) {**

**throw runtime\_error("Can not set SIGINT signal");**

**}**

**if (signal(SIGSEGV, TerminateByUser) == SIG\_ERR) {**

**throw runtime\_error("Can not set SIGSEGV signal");**

**}**

**if (signal(SIGTERM, TerminateByUser) == SIG\_ERR) {**

**throw runtime\_error("Can not set SIGTERM signal");**

**}**

**map<string,int> m;**

**Client client(stoi(argv[1]), string(argv[2]), stoi(argv[3]),m);**

**client\_ptr = &client;**

**cout << getpid() << ": " "Client started. " << "Id:" << client.get\_id() << endl;**

**for(;;){**

**Message msg = client.parent\_subscriber->receive();**

**if(msg.to\_id != client.get\_id() && msg.to\_id != UNIVERSAL\_MSG){**

**if(msg.to\_up){**

**client.send\_up(msg);**

**} else{**

**try{**

**msg.to\_up = false;**

**client.child\_publisher\_next->send(msg);**

**msg = client.next\_subscriber->receive();**

**if(msg.command == CommandType::REMOVE\_CHILD && msg.to\_id == PARENT\_SIGNAL){**

**msg.to\_id = SERVER\_ID;**

**delete client.next\_subscriber;**

**client.next\_subscriber = nullptr;**

**}**

**client.send\_up(msg);**

**}catch(...){**

**client.send\_up(Message());**

**}**

**}**

**} else{**

**process\_msg(client, msg);**

**}**

**}**

**} catch(runtime\_error& err){**

**cout << getpid() << ": " << err.what() << '\n';**

**} catch(invalid\_argument& inv){**

**cout << getpid() << ": " << inv.what() << '\n';**

**}**

**return 0;**

**}salutik@salutik:~/un/OS/lab6$ cat list.cpp**

**#include <iostream>**

**#include <vector>**

**#include "list.h"**

**using namespace std;**

**list\_el::list\_el(int new\_val) : value(new\_val), next(nullptr){}**

**int& list\_el::get\_value(){**

**return value;**

**}**

**list\_el\*& list\_el::get\_next(){**

**return next;**

**}**

**void list::delete\_list(list\_el\*& cur){**

**if(!cur) return;**

**delete\_list(cur->get\_next());**

**delete cur;**

**}**

**void list::print\_list(list\_el\*& cur){**

**if(!cur) {**

**cout <<endl;**

**return;**

**}**

**cout << cur->get\_value() << " ";**

**print\_list(cur->get\_next());**

**}**

**bool list::find\_el(list\_el\* cur, int val){**

**if(cur == nullptr){**

**return false;**

**}**

**if(val != cur->get\_value()){**

**return find\_el(cur->get\_next(), val);**

**} else {**

**return true;**

**}**

**}**

**bool list::find\_el\_in\_end(list\_el\* cur, int val){**

**if(cur == nullptr){**

**return false;**

**}**

**if((val == cur->get\_value()) && (cur->get\_next() == nullptr)){**

**return true;**

**} else {**

**return find\_el(cur->get\_next(), val);**

**}**

**}**

**void list::insert\_el(list\_el\*& cur, int new\_val){**

**if(cur == nullptr){**

**cur = new list\_el(new\_val);**

**} else {**

**return insert\_el(cur->get\_next(), new\_val);**

**}**

**}**

**void list::delete\_el\_list(list\_el\*& cur, int val){**

**if(!cur) return;**

**if(val != cur->get\_value()){**

**delete\_el\_list(cur->get\_next(), val);**

**} else{**

**delete\_list(cur);**

**cur = nullptr;**

**}**

**}**

**int list::get\_parent(list\_el\*& cur){**

**if(cur == nullptr){**

**return -1;**

**}**

**if(cur->get\_next() == nullptr){**

**return cur->get\_value();**

**} else {**

**return get\_parent(cur->get\_next());**

**}**

**}**

**list::list():root(nullptr){}**

**list::list(int new\_val){**

**root = new list\_el(new\_val);**

**}**

**void list::insert(int new\_val){**

**insert\_el(root, new\_val);**

**}**

**bool list::find(int val){**

**return find\_el(root, val);**

**}**

**bool list::find\_in\_end(int val){**

**return find\_el\_in\_end(root, val);**

**}**

**void list::print(){**

**print\_list(root);**

**}**

**void list::delete\_el(int val){**

**delete\_el\_list(root, val);**

**}**

**int list::get\_place(){**

**return get\_parent(root);**

**}**

**void list::get\_all(list\_el\* cur, vector<int>& tmp){**

**if(!cur) return;**

**tmp.push\_back(cur->get\_value());**

**get\_all(cur->get\_next(), tmp);**

**}**

**vector<int> list::get\_all\_elems(){**

**vector<int> tmp;**

**get\_all(root, tmp);**

**return tmp;**

**}**

**list::~list() {**

**delete\_list(root);**

**}**

**salutik@salutik:~/un/OS/lab6$ cat client.h**

**#ifndef \_CLIENT\_H**

**#define \_CLIENT\_H**

**#include <string.h>**

**#include <iostream>**

**#include <unistd.h>**

**#include "wrap\_zmq.h"**

**#include "socket.h"**

**#include <map>**

**using namespace std;**

**class Client{**

**private:**

**int id;**

**void\* context;**

**bool terminated;**

**map<string,int> map1;**

**public:**

**Socket\* child\_publisher\_next;**

**Socket\* parent\_publisher;**

**Socket\* parent\_subscriber;**

**Socket\* next\_subscriber;**

**Client(int new\_id, string parent\_endpoint, int new\_parent\_id, map<string,int> &new\_map);**

**~Client();**

**bool& get\_status();**

**void send\_up(Message msg);**

**void send\_down(Message msg);**

**Message receive();**

**map<string,int> &get\_map();**

**int get\_id();**

**int add\_child(int id);**

**int parent\_id;**

**};**

**#endifsalutik@salutik:~/un/OS/lab6$ cat list.h**

**#ifndef \_LIST\_H**

**#define \_LIST\_H**

**#include <vector>**

**using namespace std;**

**class list\_el{**

**list\_el\* next;**

**int value;**

**public:**

**list\_el(int new\_val);**

**int& get\_value();**

**list\_el\*& get\_next();**

**};**

**class list {**

**private:**

**list\_el\* root;**

**void delete\_list(list\_el\*& cur);**

**static void print\_list(list\_el\*& cur);**

**bool find\_el(list\_el\* cur, int val);**

**bool find\_el\_in\_end(list\_el\* cur, int val);**

**void insert\_el(list\_el\*& cur, int new\_val);**

**void delete\_el\_list(list\_el\*& cur, int val);**

**int get\_parent(list\_el\*& cur);**

**public:**

**list();**

**list(int new\_val);**

**void insert(int new\_val);**

**bool find(int val);**

**bool find\_in\_end(int val);**

**void print();**

**void delete\_el(int val);**

**int get\_place();**

**void get\_all(list\_el\* cur, vector<int>& tmp);**

**vector<int> get\_all\_elems();**

**~list();**

**};**

**#endifsalutik@salutik:~/un/OS/lab6$ cat server.cpp**

**#include <string.h>**

**#include <iostream>**

**#include <vector>**

**#include <unistd.h>**

**#include <csignal>**

**#include <map>**

**#include "server.h"**

**#include "socket.h"**

**#include "wrap\_zmq.h"**

**void\* subscriber\_thread(void\* server){**

**Server\* server\_ptr = (Server\*) server;**

**pid\_t serv\_pid = server\_ptr->get\_pid();**

**try{**

**pid\_t child\_pid = fork();**

**if(child\_pid == -1) throw runtime\_error("Can not fork.");**

**if(child\_pid == 0){**

**execl("client", "client", "0", server\_ptr->get\_publisher()->get\_endpoint().data(), "-1", nullptr);**

**throw runtime\_error("Can not execl");**

**server\_ptr->~Server();**

**return (void\*)-1;**

**}**

**string endpoint = create\_endpoint(EndpointType::PARENT\_PUB, child\_pid);**

**server\_ptr->get\_subscriber() = new Socket(server\_ptr->get\_context(), SocketType::SUBSCRIBER, endpoint);**

**for(;;){**

**Message msg = server\_ptr->get\_subscriber()->receive();**

**server\_ptr->last\_msg = msg;**

**switch(msg.command){**

**case CommandType::CREATE\_CHILD:**

**cout << "OK:" << msg.get\_create\_id() << endl;**

**break;**

**case CommandType::REMOVE\_CHILD:**

**cout << "OK" << endl;**

**server\_ptr->get\_list().delete\_el(msg.get\_create\_id());**

**break;**

**case CommandType::RETURN:**

**break;**

**case CommandType::EXEC\_CHILD:**

**if(msg.error){**

**cout << "OK:" << msg.get\_create\_id() << ": '" << msg.name << "' not found\n";**

**} else if(msg.change){**

**cout << "OK:" << msg.get\_create\_id() << endl;**

**} else{**

**cout << "OK:" << msg.get\_create\_id() << ": " << msg.val << "\n";**

**}**

**break;**

**default:**

**break;**

**}**

**}**

**} catch(runtime\_error& err){**

**cout << "Server wasn't started " << err.what() << endl;**

**}**

**return nullptr;**

**}**

**Server::Server(){**

**context = create\_zmq\_ctx();**

**pid = getpid();**

**string endpoint = create\_endpoint(EndpointType::CHILD\_PUB\_NEXT, getpid());**

**publisher = new Socket(context, SocketType::PUBLISHER, endpoint);**

**if(pthread\_create(&receive\_msg, 0, subscriber\_thread, this) != 0){**

**throw runtime\_error("Can not run second thread.");**

**}**

**working = true;**

**}**

**Server::~Server(){**

**if(!working) return;**

**working = false;**

**send(Message(CommandType::REMOVE\_CHILD, 0, 0));**

**try{**

**delete publisher;**

**delete subscriber;**

**publisher = nullptr;**

**subscriber = nullptr;**

**destroy\_zmq\_ctx(context);**

**sleep(2);**

**} catch (runtime\_error &err){**

**cout << "Server wasn't stopped " << err.what() << endl;**

**}**

**}**

**pid\_t Server::get\_pid(){**

**return pid;**

**}**

**void Server::print\_list(){**

**t.print();**

**}**

**bool Server::check(int id){**

**Message msg(CommandType::RETURN, id, 0);**

**send(msg);**

**sleep(msg\_wait\_time);**

**msg.get\_to\_id() = SERVER\_ID;**

**return last\_msg == msg;**

**}**

**void Server::send(Message msg){**

**msg.to\_up = false;**

**publisher->send(msg);**

**}**

**Message Server::receive(){**

**return subscriber->receive();**

**}**

**Socket\*& Server::get\_publisher(){**

**return publisher;**

**}**

**Socket\*& Server::get\_subscriber(){**

**return subscriber;**

**}**

**void\* Server::get\_context(){**

**return context;**

**}**

**list& Server::get\_list(){**

**return t;**

**}**

**void Server::create\_child(int id,int parent){**

**if(t.find(id)){**

**throw runtime\_error("Error:" + to\_string(id) + ":Node with that number already exists.");**

**}**

**if(parent!=0 && !t.find\_in\_end(parent)){**

**throw runtime\_error("Error:" + to\_string(id) + ":I can not do it parent or it does not exist.");**

**}**

**if(t.get\_place() && !check(t.get\_place())){**

**throw runtime\_error("Error:" + to\_string(id) + ":Parent node is unavailable.");**

**}**

**send(Message(CommandType::CREATE\_CHILD, t.get\_place(), id));**

**t.insert(id);**

**}**

**void Server::remove\_child(int id){**

**if(!t.find(id)){**

**throw runtime\_error("Error:" + to\_string(id) + ":Node with that number doesn't exist.");**

**}**

**if(!check(id)){**

**throw runtime\_error("Error:" + to\_string(id) + ":Node is unavailable.");**

**}**

**send(Message(CommandType::REMOVE\_CHILD, id, 0));**

**}**

**#define MAX\_CAP 1000**

**void Server::exec\_child(int id){**

**char name[MAX\_CAP];**

**char p;**

**cin >> p;**

**int k=0;**

**while(p!=' ' && p!='\n'){**

**name[k]=p;**

**p=getchar();**

**k++;**

**}**

**name[k]='\0';**

**k++;**

**int val=0;**

**bool change=false;**

**if(p==' '){**

**cin >> val;**

**change=true;**

**}**

**if(!t.find(id)){**

**throw runtime\_error("Error:" + to\_string(id) + ":Node with that number doesn't exist.");**

**}**

**if(!check(id)){**

**throw runtime\_error("Error:" + to\_string(id) + ":Node is unavailable.");**

**}**

**send(Message(CommandType::EXEC\_CHILD, id, k, name, change, val, false, 0));**

**}**

**void process\_cmd(Server& server, string cmd){**

**if(cmd == "create"){**

**int id;**

**cin >> id;**

**int parent;**

**cin >> parent;**

**server.create\_child(id,parent);**

**} else if (cmd == "remove"){**

**int id;**

**cin >> id;**

**server.remove\_child(id);**

**} else if (cmd == "exec"){**

**int id;**

**cin >> id;**

**server.exec\_child(id);**

**} else if(cmd == "exit"){**

**throw invalid\_argument("Exiting...");**

**} else if(cmd == "pingall"){**

**vector<int> tmp = server.get\_list().get\_all\_elems();**

**cout << "OK:";**

**bool fnd = false;**

**for(int& i : tmp){**

**if(!server.check(i)){**

**if(fnd) cout << ';';**

**cout << i;**

**fnd = true;**

**}**

**}**

**if(!fnd) cout << "-1";**

**cout << endl;**

**} else if(cmd == "status"){**

**int id;**

**cin >> id;**

**if(!server.get\_list().find(id)){**

**throw runtime\_error("Error:" + to\_string(id) + ":Node with that number doesn't exist.");**

**}**

**if(server.check(id)){**

**cout << "OK" << endl;**

**} else{**

**cout << "Node is unavailable" << endl;**

**}**

**} else if(cmd == "print"){**

**server.print\_list();**

**} else {**

**cout << "It is not a command!\n";**

**}**

**}**

**Server\* server\_ptr = nullptr;**

**void TerminateByUser(int) {**

**if (server\_ptr != nullptr) {**

**server\_ptr->~Server();**

**}**

**cout << to\_string(getpid()) + " Terminated by user" << endl;**

**exit(0);**

**}**

**int main (int argc, char const \*argv[])**

**{**

**try{**

**if (signal(SIGINT, TerminateByUser) == SIG\_ERR) {**

**throw runtime\_error("Can not set SIGINT signal");**

**}**

**if (signal(SIGSEGV, TerminateByUser) == SIG\_ERR) {**

**throw runtime\_error("Can not set SIGSEGV signal");**

**}**

**if (signal(SIGTERM, TerminateByUser) == SIG\_ERR) {**

**throw runtime\_error("Can not set SIGTERM signal");**

**}**

**Server server;**

**server\_ptr = &server;**

**cout << getpid() << " server started correctly!\n";**

**for(;;){**

**try{**

**string cmd;**

**while(cin >> cmd){**

**process\_cmd(server, cmd);**

**}**

**} catch(const runtime\_error& arg){**

**cout << arg.what() << endl;**

**}**

**}**

**} catch(const runtime\_error& arg){**

**cout << arg.what() << endl;**

**} catch(...){}**

**sleep(5);**

**return 0;**

**}salutik@salutik:~/un/OS/lab6$ cat server.h**

**#ifndef \_SERVER\_H**

**#define \_SERVER\_H**

**#include "list.h"**

**#include "socket.h"**

**#include "wrap\_zmq.h"**

**#define msg\_wait\_time 1**

**class Server{**

**public:**

**Server();**

**~Server();**

**void print\_list();**

**void send(Message msg);**

**Message receive();**

**void create\_child(int id, int parent);**

**void remove\_child(int id);**

**void exec\_child(int id);**

**pid\_t get\_pid();**

**bool check(int id);**

**Socket\*& get\_publisher();**

**Socket\*& get\_subscriber();**

**void\* get\_context();**

**list& get\_list();**

**Message last\_msg;**

**private:**

**pid\_t pid;**

**list t;**

**void \*context = nullptr;**

**Socket\* publisher;**

**Socket\* subscriber;**

**bool working;**

**pthread\_t receive\_msg;**

**};**

**#endifsalutik@salutik:~/un/OS/lab6$ cat Socket.cpp**

**#include <string>**

**#include <iostream>**

**#include <stdexcept>**

**#include "wrap\_zmq.h"**

**#include "socket.h"**

**using namespace std;**

**Socket::Socket(void\* context, SocketType new\_socket\_type, string new\_endpoint)**

**: socket\_type(new\_socket\_type), endpoint(new\_endpoint) {**

**socket = create\_zmq\_socket(context, new\_socket\_type);**

**switch (socket\_type) {**

**case SocketType::PUBLISHER:**

**bind\_zmq\_socket(socket, new\_endpoint);**

**break;**

**case SocketType::SUBSCRIBER:**

**connect\_zmq\_socket(socket, new\_endpoint);**

**break;**

**default:**

**throw logic\_error("Undefined connection type");**

**}**

**}**

**Socket::~Socket() {**

**try {**

**switch(socket\_type){**

**case SocketType::PUBLISHER:**

**unbind\_zmq\_socket(socket, endpoint);**

**break;**

**case SocketType::SUBSCRIBER:**

**disconnect\_zmq\_socket(socket, endpoint);**

**break;**

**}**

**close\_zmq\_socket(socket);**

**} catch (exception& ex) {**

**cout << "Socket wasn't closed: " << ex.what() << endl;**

**}**

**}**

**void Socket::send(Message message) {**

**if (socket\_type == SocketType::PUBLISHER) {**

**send\_zmq\_msg(socket, message);**

**} else {**

**throw logic\_error("SUB socket can't send messages");**

**}**

**}**

**Message Socket::receive() {**

**if (socket\_type == SocketType::SUBSCRIBER) {**

**return get\_zmq\_msg(socket);**

**} else {**

**throw logic\_error("PUB socket can't receive messages");**

**}**

**}**

**void Socket::subscribe(string new\_endpoint) {**

**if (socket\_type == SocketType::SUBSCRIBER) {**

**connect\_zmq\_socket(socket, new\_endpoint);**

**endpoint = new\_endpoint;**

**} else {**

**throw logic\_error("Subscribe is only for SUB sockets");**

**}**

**}**

**string Socket::get\_endpoint() const{**

**return endpoint;**

**}**

**void\*& Socket::get\_socket(){**

**return socket;**

**}salutik@salutik:~/un/OS/lab6$ cat socket.h**

**#ifndef \_SOCKET\_H**

**#define \_SOCKET\_H**

**#include <string>**

**#include "wrap\_zmq.h"**

**using namespace std;**

**class Socket {**

**public:**

**Socket(void\* context, SocketType new\_socket\_type, string new\_endpoint);**

**~Socket();**

**void send(Message message);**

**Message receive();**

**void subscribe(string new\_endpoint);**

**string get\_endpoint() const;**

**void\*& get\_socket();**

**private:**

**void\* socket;**

**SocketType socket\_type;**

**string endpoint;**

**};**

**#endifsalutik@salutik:~/un/OS/lab6$ cat wrap\_zmq.cpp**

**#include <tuple>**

**#include <cstring>**

**#include <iostream>**

**#include "socket.h"**

**#include "wrap\_zmq.h"**

**#include <unistd.h>**

**using namespace std;**

**void\* create\_zmq\_ctx(){**

**void\* context = zmq\_ctx\_new();**

**if(context == nullptr){**

**throw runtime\_error("Can not create new context.");**

**}**

**return context;**

**}**

**void destroy\_zmq\_ctx(void\* context){**

**sleep(1);**

**if(zmq\_ctx\_destroy(context) != 0){**

**throw runtime\_error("Can not destroy context.");**

**}**

**}**

**int get\_zmq\_socket\_type(SocketType type){**

**switch(type){**

**case SocketType::PUBLISHER:**

**return ZMQ\_PUB;**

**case SocketType::SUBSCRIBER:**

**return ZMQ\_SUB;**

**default:**

**throw runtime\_error("Undefined socket type.");**

**}**

**}**

**void\* create\_zmq\_socket(void\* context, SocketType type){**

**int zmq\_type = get\_zmq\_socket\_type(type);**

**void\* socket = zmq\_socket(context, zmq\_type);**

**if(socket == nullptr){**

**throw runtime\_error("Can not create socket.");**

**}**

**return socket;**

**}**

**void close\_zmq\_socket(void\* socket){**

**sleep(1);**

**if(zmq\_close(socket) != 0){**

**throw runtime\_error("Can not close socket.");**

**}**

**}**

**string create\_endpoint(EndpointType type, pid\_t id){**

**switch(type){**

**case EndpointType::PARENT\_PUB:**

**return "ipc:///tmp/parent\_pub\_" + to\_string(id);**

**case EndpointType::CHILD\_PUB\_NEXT:**

**return "ipc:///tmp/child\_pub\_next\_" + to\_string(id);**

**default:**

**throw runtime\_error("Wrong Endpoint type.");**

**}**

**}**

**void bind\_zmq\_socket(void\* socket, string endpoint){**

**if(zmq\_bind(socket, endpoint.data()) != 0){**

**throw runtime\_error("Can not bind socket.");**

**}**

**}**

**void unbind\_zmq\_socket(void\* socket, string endpoint){**

**sleep(1);**

**if(zmq\_unbind(socket, endpoint.data()) != 0){**

**throw runtime\_error("Can not unbind socket.");**

**}**

**}**

**void connect\_zmq\_socket(void\* socket, string endpoint){**

**if(zmq\_connect(socket, endpoint.data()) != 0){**

**throw runtime\_error("Can not connect socket.");**

**}**

**zmq\_setsockopt(socket, ZMQ\_SUBSCRIBE, 0, 0);**

**}**

**void disconnect\_zmq\_socket(void\* socket, string endpoint){**

**if(zmq\_disconnect(socket, endpoint.data()) != 0){**

**throw runtime\_error("Can not disconnect socket.");**

**}**

**}**

**atomic<int> Message::counter;**

**Message::Message(){**

**command = CommandType::ERROR;**

**uniq\_num = counter++;**

**to\_up = false;**

**}**

**Message::Message(CommandType new\_command, int new\_to\_id, int new\_k, char\* new\_name, bool new\_change, int new\_val, bool new\_error, int new\_id)**

**: command(new\_command), to\_id(new\_to\_id), k(new\_k), change(new\_change), val(new\_val), error(new\_error), uniq\_num(counter++), to\_up(false), create\_id(new\_id){**

**for(int i = 0; i < k; ++i){**

**name[i] = new\_name[i];**

**}**

**}**

**Message::Message(CommandType new\_command, int new\_to\_id, int new\_id)**

**: command(new\_command), to\_id(new\_to\_id), uniq\_num(counter++), to\_up(false), create\_id(new\_id){}**

**bool operator==(const Message& lhs, const Message& rhs){**

**return tie(lhs.command, lhs.to\_id, lhs.create\_id, lhs.uniq\_num) ==**

**tie(rhs.command, rhs.to\_id, rhs.create\_id, rhs.uniq\_num);**

**}**

**int& Message::get\_create\_id(){**

**return create\_id;**

**}**

**int& Message::get\_to\_id(){**

**return to\_id;**

**}**

**void create\_zmq\_msg(zmq\_msg\_t\* zmq\_msg, Message& msg){**

**zmq\_msg\_init\_size(zmq\_msg, sizeof(msg));**

**memcpy(zmq\_msg\_data(zmq\_msg), &msg, sizeof(msg));**

**}**

**void send\_zmq\_msg(void\* socket, Message& msg){**

**zmq\_msg\_t zmq\_msg;**

**create\_zmq\_msg(&zmq\_msg, msg);**

**if(!zmq\_msg\_send(&zmq\_msg, socket, 0)){**

**throw runtime\_error("Can not send message.");**

**}**

**zmq\_msg\_close(&zmq\_msg);**

**}**

**Message get\_zmq\_msg(void\* socket){**

**zmq\_msg\_t zmq\_msg;**

**zmq\_msg\_init(&zmq\_msg);**

**if(zmq\_msg\_recv(&zmq\_msg, socket, 0) == -1){**

**return Message();**

**}**

**Message msg;**

**memcpy(&msg, zmq\_msg\_data(&zmq\_msg), sizeof(msg));**

**zmq\_msg\_close(&zmq\_msg);**

**return msg;**

**}salutik@salutik:~/un/OS/lab6$ cat wrap\_zmq.h**

**#ifndef \_WRAP\_ZMQ\_H**

**#define \_WRAP\_ZMQ\_H**

**#include <tuple>**

**#include <vector>**

**#include <atomic>**

**#include <zmq.h>**

**#include "wrap\_zmq.h"**

**using namespace std;**

**#define UNIVERSAL\_MSG -1**

**#define SERVER\_ID -2**

**#define PARENT\_SIGNAL -3**

**enum struct SocketType{**

**PUBLISHER,**

**SUBSCRIBER,**

**};**

**enum struct CommandType {**

**ERROR,**

**RETURN,**

**CREATE\_CHILD,**

**REMOVE\_CHILD,**

**EXEC\_CHILD,**

**};**

**enum struct EndpointType{**

**CHILD\_PUB\_NEXT,**

**PARENT\_PUB,**

**};**

**void\* create\_zmq\_ctx();**

**void destroy\_zmq\_ctx(void\* context);**

**int get\_zmq\_socket\_type(SocketType type);**

**void\* create\_zmq\_socket(void\* context, SocketType type);**

**void close\_zmq\_socket(void\* socket);**

**string create\_endpoint(EndpointType type, pid\_t id);**

**void bind\_zmq\_socket(void\* socket, string endpoint);**

**void unbind\_zmq\_socket(void\* socket, string endpoint);**

**void connect\_zmq\_socket(void\* socket, string endpoint);**

**void disconnect\_zmq\_socket(void\* socket, string endpoint);**

**#define MAX\_CAP 1000**

**class Message {**

**protected:**

**static std::atomic<int> counter;**

**public:**

**CommandType command = CommandType::ERROR;**

**int to\_id;**

**int create\_id;**

**int uniq\_num;**

**bool to\_up;**

**int k=0;**

**char name[MAX\_CAP] = {' '};**

**bool change=false;**

**int val=0;**

**bool error=false;**

**Message();**

**Message(CommandType new\_command, int new\_to\_id, int new\_k, char\* new\_name, bool new\_change, int new\_val, bool new\_error, int new\_id);**

**Message(CommandType new\_command, int new\_to\_id, int new\_id);**

**friend bool operator==(const Message& lhs, const Message& rhs);**

**int& get\_create\_id();**

**int& get\_to\_id();**

**};**

**void create\_zmq\_msg(zmq\_msg\_t\* zmq\_msg, Message& msg);**

**void send\_zmq\_msg(void\* socket, Message& msg);**

**Message get\_zmq\_msg(void\* socket);**

**#endif**

**Примеры работы**

salutik@salutik:~/un/OS/lab6$ ./server

2765 server started correctly!

2769: Client started. Id:0

create 5 0

OK:2772

2772: Client started. Id:5

create 3 5

OK:2777

2777: Client started. Id:3

exec 5 m 3

OK:5

exec 5 m

OK:5: 3

exec 5 m 10

OK:5

exec 5 m

OK:5: 10

exit

OK

2777: Exiting child...

2772: Exiting child...

2769: Exiting child...

**Вывод**

В процессе написания лабораторной я научилась основам работы с серверами сообщений на примере ZMQ, изучил возможные подходы к созданию приложений на основе серверов сообщений. Технология действительно удобна для решения ряда задач, а конкретно ZMQ обладает достаточно высокой скоростью передачи данных.

Конечно, у ZMQ есть ряд недостатков, но особых трудностей при написании лабораторной это не вызвало. Использование C++ оказалось оправданным в этой работе, так как это позволило быстро разработать ряд классов-абстракций и не держать в голове архитектуру целиком.