Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

**Управление серверами сообщений**

Студент: Чурсина Наталья Алексеевна

Группа: М8О –308Б-18

Вариант: 2

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Тестирование
6. Демонстрация работы программы
7. Вывод

**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.   
Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

**Задание согласно варианту 45:**

Тип топологии: 1.

Тип вычислительных команд: 1.

тип команд доступности узлов 2.

**Общие сведения о программе**

Программа состоит из двух основных файлов и библиотеки, реализующей взаимодействия с узлами. Помимо этого используется библиотека zmq, которая реализует очередь сообщений.

1) client.cpp - программа, которая является управляющим узлом

2) server.сpp – программа, которая производит вычисления

3) zmq\_requests.h/cpp – небольшая библиотека для принятия/отправки сообщений.

Очередь сообщений - [компонент](https://ru.wikipedia.org/wiki/%25D0%259A%25D0%25BE%25D0%25BC%25D0%25BF%25D0%25BE%25D0%25BD%25D0%25B5%25D0%25BD%25D1%2582%25D0%25BD%25D0%25BE-%25D0%25BE%25D1%2580%25D0%25B8%25D0%25B5%25D0%25BD%25D1%2582%25D0%25B8%25D1%2580%25D0%25BE%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25BD%25D0%25BE%25D0%25B5_%25D0%25BF%25D1%2580%25D0%25BE%25D0%25B3%25D1%2580%25D0%25B0%25D0%25BC%25D0%25BC%25D0%25B8%25D1%2580%25D0%25BE%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25B8%25D0%25B5), используемый для [межпроцессного](https://ru.wikipedia.org/wiki/%25D0%259C%25D0%25B5%25D0%25B6%25D0%25BF%25D1%2580%25D0%25BE%25D1%2586%25D0%25B5%25D1%2581%25D1%2581%25D0%25BD%25D0%25BE%25D0%25B5_%25D0%25B2%25D0%25B7%25D0%25B0%25D0%25B8%25D0%25BC%25D0%25BE%25D0%25B4%25D0%25B5%25D0%25B9%25D1%2581%25D1%2582%25D0%25B2%25D0%25B8%25D0%25B5) или [межпотокового](https://ru.wikipedia.org/wiki/%25D0%259F%25D0%25BE%25D1%2582%25D0%25BE%25D0%25BA_%25D0%25B2%25D1%258B%25D0%25BF%25D0%25BE%25D0%25BB%25D0%25BD%25D0%25B5%25D0%25BD%25D0%25B8%25D1%258F) взаимодействия внутри одного [процесса](https://ru.wikipedia.org/wiki/%25D0%259F%25D1%2580%25D0%25BE%25D1%2586%25D0%25B5%25D1%2581%25D1%2581_(%25D0%25B8%25D0%25BD%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%25D1%2582%25D0%25B8%25D0%25BA%25D0%25B0)). Для [обмена сообщениями](https://ru.wikipedia.org/wiki/%25D0%259E%25D0%25B1%25D0%25BC%25D0%25B5%25D0%25BD_%25D1%2581%25D0%25BE%25D0%25BE%25D0%25B1%25D1%2589%25D0%25B5%25D0%25BD%25D0%25B8%25D1%258F%25D0%25BC%25D0%25B8) используется очередь. Очереди сообщений предоставляют асинхронный протокол передачи данных, означая, что отправитель и получатель сообщения не обязаны взаимодействовать с очередью сообщений одновременно. Размещённые в очереди сообщения хранятся до тех пор, пока получатель не получит их.

Очереди сообщений имеют неявные или явные ограничения на размер данных, которые могут передаваться в одном сообщении, и количество сообщений, которые могут оставаться в очереди.

Многие реализации очередей сообщений функционируют внутренне: внутри операционной системы или внутри приложения. Такие очереди существуют только для целей этой системы.

Другие реализации позволяют передавать сообщения между различными компьютерными системами, потенциально подключая несколько приложений и несколько операционных систем. Эти системы очередей сообщений обычно обеспечивают расширенную функциональность для обеспечения устойчивости, чтобы гарантировать, что сообщения не будут «потеряны» в случае сбоя системы.

ZMQ - это высокопроизводительная библиотека асинхронных сообщений, предназначенная для использования в распределенных или параллельных приложениях. Он обеспечивает очередь сообщений, но в отличие от промежуточного программного обеспечения, ориентированного на сообщения, система ZMQ может работать без выделенного посредника сообщений.

ZMQ поддерживает общие шаблоны обмена сообщениями (публикация / подпрограмма, запрос / ответ, клиент / сервер и т.д.) По различным транспортам (TCP, внутрипроцессный, межпроцессный, многоадресный, WebSocket и т.д.), что делает межпроцессный обмен сообщениями простым как обмен сообщениями между потоками.

**Сокеты**— название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Основные функции и вызовы:

1) **void \* zmq\_ctx\_new()** - создать новый контекст ØMQ. Функция zmq\_ctx\_new () должна вернуть дескриптор вновь созданного контекста в случае успеха. В противном случае он должен вернуть NULL и установить для errno одно из значений.

2) **void \* zmq\_socket (void \* context , int type )** - Функция должна создать сокет ZMQ в указанном контексте и вернуть непрозрачный дескриптор вновь созданному сокету. Аргумент type указывает тип сокета, который определяет семантику связи через сокет.

Вновь созданный сокет изначально не связан и не связан ни с какими конечными точками. Чтобы установить поток сообщений, сокет должен быть сначала подключен по крайней мере к одной конечной точке с помощью zmq\_connect (3), или по крайней мере одна конечная точка должна быть создана для приема входящих соединений с помощью zmq\_bind (3).

Сокет типа ZMQ\_REQ используется клиентом для отправки запросов и получения ответов от службы.

ZMQ\_REP используется службой для получения запросов и отправки ответов клиенту.

3) **int execv(const char \**path*, char \*const *argv*[])** - функция **execv**() заменяет текущий образ процесса новым.**execv()**предоставляет новой программе список аргументов в виде массива указателей на строки, заканчивающиеся null. Первый аргумент, по соглашению, должен указать на имя, ассоциированное с файлом, который необходимо запустить. Массив указателей *должен* заканчиваться указателем null.

4) **int zmq\_msg\_init\_size (zmq\_msg\_t \* msg , size\_t size) -** функция  должна выделить любые ресурсы, необходимые для хранения сообщения размером в байтах, и инициализировать объект сообщения, на который ссылается msg, для представления вновь выделенного сообщения. Реализация должна выбирать, хранить ли содержимое сообщения в стеке (маленькие сообщения) или в куче (большие сообщения). По причинам производительности zmq\_msg\_init\_size () не должен очищать данные сообщения.

6) **int zmq\_msg\_recv (zmq\_msg\_t \* msg , void \* socket , int flags) -** Функция должна получить часть сообщения из сокета, на который ссылается аргумент сокета, и сохранить ее в сообщении, на которое ссылается аргумент msg . Любой контент, ранее сохраненный в msg, должен быть должным образом освобожден. Если есть нет частей сообщений доступны на указанном сокете zmq\_msg\_recv () функция должна блокироваться , пока запрос не может быть удовлетворен. Флаги аргумент представляет собой комбинацию флагов, определенных ниже: **ZMQ\_DONTWAIT -** Указывает, что операция должна выполняться в неблокирующем режиме. Если в указанном *сокете* нет доступных сообщений , *функция zmq\_msg\_recv ()* завершится ошибкой с *errno,* установленным в EAGAIN.

7) **int zmq\_send (void *\* socket* , void *\* buf* , size\_t *len* , int *flags* ) -** Функция  должна *поставить в* очередь сообщение, созданное из буфера, на который ссылаются аргументы *buf* и *len* . *flags -* аргумент представляет собой комбинацию флагов , определенных ниже:

**ZMQ\_DONTWAIT -** Для типов сокетов (DEALER, PUSH), которые блокируются, когда нет доступных одноранговых узлов (или все одноранговые узлы имеют максимальную отметку), указывает, что операция должна выполняться в неблокирующем режиме. Если сообщение не может быть поставлено в очередь на *сокете* , *функция zmq\_send ()* завершится ошибкой с *errno,* установленным в EAGAIN.

**ZMQ\_SNDMORE -** Указывает, что отправляемое сообщение является сообщением, состоящим из нескольких частей, и что последующие части сообщения должны следовать. Подробное описание см. В разделе, посвященном сообщениям, состоящим из нескольких частей, ниже.

ZMQ\_NOBLOCK - Указывает, что операция должна выполняться в неблокирующем режиме. Если сообщение не может быть поставлено в очередь в сокете, функция socket\_t::send () должна завершиться с ошибкой, когда errno установлено в EAGAIN.

ZMQ\_SNDMORE - Указывает, что отправляемое сообщение является сообщением, состоящим из нескольких частей, и что последующие части сообщения должны следовать.

8) **void \*memcpy(void \*dest, const void \*source, size\_t count)** - Функция memcpy() копирует count символов из массива, на который указывает source, в массив, на который указывает dest. Если массивы перекрываются, поведение memcpy() не определено.

9) **int zmq\_msg\_close (zmq\_msg\_t \* msg ) -** Функция  должна информировать инфраструктуру ØMQ о том, что любые ресурсы, связанные с объектом сообщения, на который ссылается msg , больше не требуются и могут быть освобождены. Фактическое высвобождение ресурсов, связанных с объектом сообщения, должно быть отложено ØMQ до тех пор, пока все пользователи сообщения или базового буфера данных не укажут, что он больше не требуется. Приложения должны гарантировать, что zmq\_msg\_close () вызывается, когда сообщение больше не требуется, в противном случае могут возникнуть утечки памяти. Обратите внимание, что это НЕ требуется после успешного выполнения zmq\_msg\_send () .

10) **int zmq\_msg\_recv (zmq\_msg\_t \* msg , void \* socket , int flags) -** Функция zmq\_msg\_recv () должна получить часть сообщения из сокета, на который ссылается аргумент сокета, и сохранить ее в сообщении, на которое ссылается аргумент msg . Любой контент, ранее сохраненный в msg, должен быть должным образом освобожден. Если есть нет частей сообщений доступны на указанном сокете zmq\_msg\_recv () функция должна блокироваться , пока запрос не может быть удовлетворен. flags - аргумент представляет собой комбинацию флагов , определенных ниже:

**ZMQ\_DONTWAIT -** Указывает, что операция должна выполняться в неблокирующем режиме. Если в указанном сокете нет доступных сообщений , функция zmq\_msg\_recv () завершится ошибкой с errno, установленным в EAGAIN.

11) **void\* zmq\_msg\_data (zmq\_msg\_t \* msg ) -** Функция zmq\_msg\_data () должна возвращать указатель на содержимое сообщения объекта сообщения, на который ссылается msg.

12) **size\_t zmq\_msg\_size (zmq\_msg\_t \* msg ) -** Функция zmq\_msg\_size () должна возвращать размер в байтах содержимого объекта сообщения, на который ссылается msg .

**Общий метод и алгоритм решения**.

Имеется управляющий узел и выполняющие узлы

1. Управляющий узел принимает команды, обрабатывает их и пересылает их дочерним узлам. В случае ошибок, управляющий узел выводит сообщение об этом.
2. Из дочерних узлов возвращается некоторое сообщение (об успехе либо ошибки выполнения переданной команды).
3. Если узел недоступен, то по истечении попытки подключиться к узлу будет выдано сообщение о недоступности узла.

**Листинг программы.**

**Файл client.cpp**

#include <unistd.h>

#include <sys/types.h>

#include <signal.h>

#include <iostream>

#include <exception>

#include <unordered\_map>

#include <vector>

#include <string>

#include <cstring>

#include "zmq.h"

#include "zmq\_requests.h"

using namespace std;

void PrintMenu(ostream& os = cout) {

os << "List of commands: " << endl

<< "1) create id [parent] - create new server node" << endl

<< "2) remove id - remove existing server node" << endl

<< "3) exec id n k1...kn - evaluate sum of n numbers" << endl

<< "4) ping id - check if server still available" << endl

<< "5) exit - exit program" << endl

<< "6) print - print menu" << endl;

}

string CreateNewServerNode(const unordered\_map<int, void\*>& socket, const int id, const int parent,

unordered\_map<int, pid\_t>& server\_nodes\_pids) {

string result = "";

if (parent == -1) {

if (server\_nodes\_pids.find(id) != server\_nodes\_pids.end()) {

result = string("Error: Already exists");

} else {

cout << "Enter the port to connect: ";

int port = 0;

cin >> port;

string address = "tcp://localhost:" + to\_string(port);

cout << "Enter the executable file (\"./smth\"): ";

string path = "";

cin >> path;

pid\_t pid = fork();

if (pid == -1) {

result = "Error: Unable to create new server node";

} else if (pid == 0) { // ребенок

CreateServerNode(path, id, port);

} else { // родитель

if (static\_cast<int>(zmq\_connect(socket.at(id), address.c\_str())) == 0) {

server\_nodes\_pids[id] = pid;

SendMessage(socket.at(id), "Ping");

result = ReceiveMessage(socket.at(id));

result += ": " + to\_string(pid);

} else {

result = "Can't connect to the server with id " + to\_string(id);

result += " and port " + to\_string(port) + ". Please try again";

}

}

}

} else {

result = string("Error: Parent not found");

}

return result;

}

string RemoveServerNode(const pid\_t pid) {

string result = "";

int is\_killed = kill(pid, SIGKILL); // второй аргумент для немедленного завершения процесса

is\_killed == 0 ? result = "Ok" : result = "Can't kill the process";

return result;

}

string Exec(const unordered\_map<int, void\*>& socket, const int id, vector<int> v) {

string result = "";

if (socket.find(id) != socket.end()) {

string to\_calc = "Exec ";

for (const auto& item : v) {

to\_calc += to\_string(item);

to\_calc += " ";

}

SendMessage(socket.at(id), to\_calc);

result = ReceiveMessage(socket.at(id));

} else {

result = "Error: Not found";

}

return result;

}

string PingServerNode(const unordered\_map<int, void\*>& socket, const int id,

unordered\_map<int, pid\_t>& server\_nodes\_pids) {

string result = "";

if (server\_nodes\_pids.find(id) != server\_nodes\_pids.end()) {

try {

SendMessage(socket.at(id), "Ping");

result = ReceiveMessage(socket.at(id));

result += ": 1";

} catch(...) {

result += ": 0";

}

} else {

result = "Error: Not found";

}

return result;

}

int main () {

unordered\_map<int, pid\_t> server\_nodes\_pids;

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

cout << "Client starting..." << endl;

PrintMenu();

unordered\_map<int, void\*> request;

while (true) {

string command;

cin >> command;

if (command == "create") {

int id = 0, parent = 0;

cin >> id >> parent;

if (server\_nodes\_pids.find(id) == server\_nodes\_pids.end()) {

request[id] = zmq\_socket(context, ZMQ\_REQ);

}

cout << CreateNewServerNode(request, id, parent, server\_nodes\_pids) << endl;

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

int id = 0;

cin >> id;

auto it = server\_nodes\_pids.find(id);

if (it != server\_nodes\_pids.end()) {

cout << RemoveServerNode(it->second) << endl;

server\_nodes\_pids.erase(it);

request.erase(id);

} else {

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

}

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

int id = 0, k = 0;

cin >> id >> k;

string ping = PingServerNode(request, id, server\_nodes\_pids);

if (ping == "Ok: 1") {

vector<int> numbers;

numbers.reserve(k);

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

int n = 0;

cin >> n;

numbers.push\_back(n);

}

cout << Exec(request, id, move(numbers)) << endl;

} else {

cout << "Error: Server is unavailiable" << endl;

}

} else if (command == "ping") {

int id = 0;

cin >> id;

cout << PingServerNode(request, id, server\_nodes\_pids) << endl;

} else if (command == "exit") {

break;

} else if (command == "print") {

PrintMenu();

} else {

cout << "Wrong input" << endl;

}

}

for (const auto socket : request) {

zmq\_close(socket.second);

}

zmq\_ctx\_destroy(context);

return 0;

}

**Файл server.cpp**

#include <unistd.h>

#include <sys/types.h>

#include <iostream>

#include <string>

#include <cstring>

#include "zmq.h"

#include "zmq\_requests.h"

using namespace std;

int ParseRequest(const string& s) {

int result = 0;

string current\_number;

for (const auto item : s) {

if (item != ' ' && isdigit(item)) {

current\_number.push\_back(item);

} else if (item == ' ' && !current\_number.empty()) {

result += stoi(current\_number);

current\_number.clear();

}

}

return result;

}

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

int id = stoi(argv[1]);

int port = stoi(argv[2]);

string address = "tcp://\*:" + to\_string(port);

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

void\* respond = zmq\_socket(context, ZMQ\_REP);

zmq\_bind(respond, address.c\_str());

string request = "";

request = ReceiveMessage(respond);

while (true) {

sleep(1);

if (request == "Ping") {

SendMessage(respond, "Ok");

} else if (request.substr(0, 4) == "Exec") {

int sum = ParseRequest(request);

string result = "";

result = "Ok: " + to\_string(id) + ": " + to\_string(sum);

SendMessage(respond, result);

}

request = ReceiveMessage(respond);

}

zmq\_close(respond);

zmq\_ctx\_destroy(context);

return 0;

}

**Файл server.cpp**

#include <unistd.h>

#include <cstring>

#include <iostream>

#include <exception>

#include <string>

#include "zmq.h"

void CreateServerNode(std::string path, const int id, const int port) {

char\* id\_ = strdup(std::to\_string(id).c\_str());

char\* port\_ = strdup(std::to\_string(port).c\_str());

char\* args[] = {strdup(path.c\_str()), id\_, port\_, NULL};

execv(strdup(path.c\_str()), args);

}

void SendMessage(void\* request, const std::string message) {

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, message.size());

memcpy(zmq\_msg\_data(&req), message.c\_str(), message.size());

zmq\_msg\_send(&req, request, 0);

zmq\_msg\_close(&req);

}

std::string ReceiveMessage(void\* request) {

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

zmq\_msg\_recv(&reply, request, 0);

std::string result(static\_cast<char\*>(zmq\_msg\_data(&reply)), zmq\_msg\_size(&reply));

zmq\_msg\_close(&reply);

return result;

}

**Тестирование.**

См. следующий пункт.

**Демонстрация работы программы.**

user@DESKTOP-4RFVSK2:~/os/6lab$ ./client

Client starting...

List of commands:

1) create id [parent] - create new server node

2) remove id - remove existing server node

3) exec id n k1...kn - evaluate sum of n numbers

4) ping id - check if server still available

5) exit - exit program

6) print - print menu

create 1 -1

Enter the port to connect: 8080

Enter the executable file: ("./smth")./server

Ok: 1782

exec 1 5 1 2 3 4 5

Ok: 1: 15

ping 1

Ok: 1

create 2 -1

Enter the port to connect: 9090

Enter the executable file: ("./smth")./server

Ok: 1785

remove 0

Error: Not found

remove 1

Ok

exec 2 6 4 5 6 7 8 9

Ok: 2: 39

exit

**Вывод.**

Очередь сообщений очень полезная технология для различных ситуаций. Они являются важным компонентом в приложениях и архитектурах программного обеспечения за счет масштабируемости (хорошо распределяют процессы обработки информации), асинхронности и удобного интерфейса обмена данными между процессами.