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»**

**ОЧЕРЕДИ СООБЩЕНИЙ**

Студент: Армишев Кирилл Константинович

Группа: М8О–208Б–21

Вариант: 9

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2022.

**Постановка задачи**

## 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.

**Общие сведения о программе**

Основная программа компилируется из файла manager.c. Также подключаются файлы zmq\_struct.c — функции для работы с очередью. worker.c представляет собой вычислительную ноду к которой также подключается zmq\_struct.c.

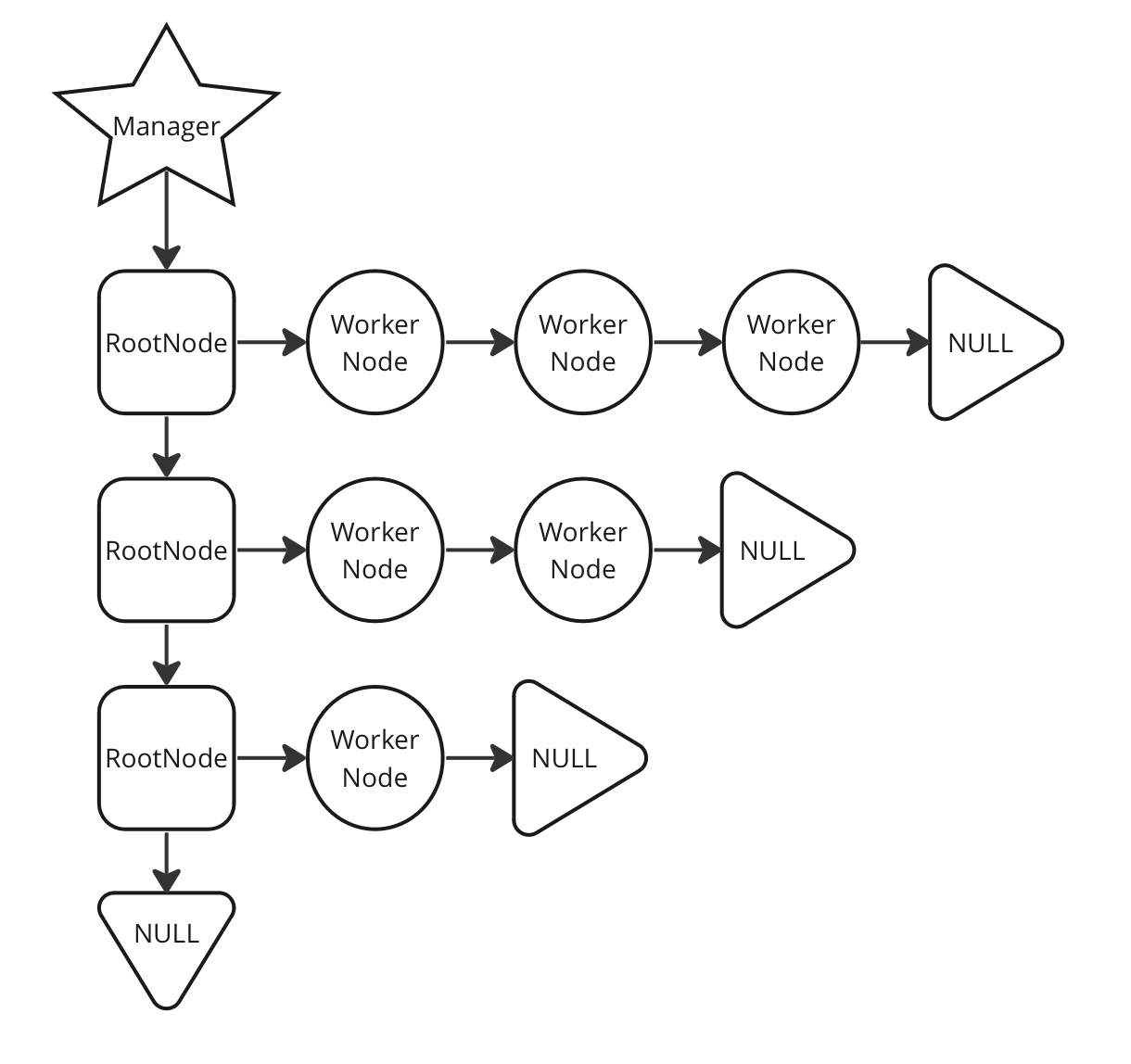
Вызовы

1. zmq\_ctx\_new() – создает новый ZeroMQ контекст. В нем мы можем запустить сокеты. Контекст является потокобезопасным.
2. zmq\_socket(ctx, type) – создает новый сокет. Первым параметром передается контекст, относительно которого создается сокет. Вторым параметром указывается тип паттерна соединения сокета: PUSH / PULL, SEND / RECV, PAIR.
3. zmq\_setsockopt(socket, name, value, length) – позволяет определить значение некоторого параметра сокета. Название сокета указывается в параметре name. Значение и размер значения указываются в параметрах value и length соответственно.
4. zmq\_bind(socket, endpoint) – подключает socket к endpoint, который представляет собой строку-адрес, состоящую из IP и PORT. Далее сокет является серверным. То есть, принимает входные соединения.
5. zmq\_connect(socket, endpoint) – подключает сокет к endpoint. Далее сокет играет роль клиентского.
6. zmq\_msg\_init(msg) – инициализирует объект сообщения, выделяя память под него.
7. zmq\_msg\_init\_size(msg, size) – выделяет необходимое количество ресурсов для сообщения.
8. zmq\_msg\_data(msg) – возвращает указатель на начало области памяти, где хранятся данные сообщения
9. zmq\_msg\_send(msg, socket, flag) – отправляет сообщение msg по сокету socket. Дополнительно можно указать опции в параметре flag. Например, флаг ZMQ\_DONTWAIT устанавливает операции неблокирующий режим.
10. zmq\_msg\_close(msg) – очищает память, занятую под сообщение
11. zmq\_msg\_recv(msg, socket, flags) – считывает входящее сообщение с сокета socket. Результат записывается в сообщение msg. Также, можно указать специальные опции при получении сообщения. Указываются они в параметре flags. Среди них есть флаг ZMQ\_DONTWAIT, который делает операцию чтения неблокирующей
12. z mq\_msg\_size(msg) – возвращает количество байтов, занимаемых содержанием сообщения
13. zmq\_ctx\_destroy(ctx) – удаляет ранее созданный контекст
14. zmq\_close(socket) – удаляет ранее созданный сокет.

**Общий метод и алгоритм решения**.

ZeroMQ представляет собой брокер сообщений, каждый раз, когда мы связываем два сокета между ними возникает очередь сообщений. Реализуем главный узел, который будет отвечать за создание новых узлов и удаление старых. Реализуем создание новых узлов, с помощью системного вызова fork(), с помощью execlp() будем запускать экземпляры программ worker.

Схема реализации распределенной системы:



**Основные файлы программы**

**Manager.c:**

#include "log/log.h"  
  
#include <stdlib.h> *// malloc, size\_t*#include <stdio.h>  
#include <unistd.h>  
#include <czmq.h>  
#include <assert.h>  
#include <string.h>  
#include <time.h> *// timespec, nanosleep*#include <zmq.h>  
#include <sys/wait.h>  
#include <pthread.h>  
  
  
  
static const int MAX\_LIFE\_POINTS = 4; *// максимальное количество очков жизни*static char\* WORKER\_EXEC\_PATH = "worker"; *// путь к исполняемому файлу вычислительного узла*static const long START\_HEARTBEAT\_INTERVAL = 1000;  
static const long MAX\_TIMEOUT = 1000; *// таймаут (в мс) для принятия / отправки сообщений*static const int OK = 0; *// операция выполнено успешно*static const int ALREADY\_EXISTS = 1; *// узел уже существует*static const int NOT\_EXISTS = 2; *// узла не существуте*static const int UNACTIVE = 3; *// узел неактивен  
  
  
/\* Topology functional \*/  
  
// структура элемента вычислительного узла*typedef struct \_\_WorkerNode {  
 struct \_\_WorkerNode\* next; *// указатель на следующий элемент (NULL если элемент последний)* int id; *// идентификатор вычислительного узла* zsock\_t\* socket; *// сокет для отправки запросов на узел* int life\_points; *// очки жизни узла*} WorkerNode;  
  
  
*// структура корневого элемента  
// @note такие элементы нужны для связки нескольких список из рабочих узлов*typedef struct \_\_RootNode {  
 struct \_\_RootNode\* next; *// ссылка на следующий корневой элемент (NULL если элемент последний)* WorkerNode\* root\_worker; *// ссылка на первый элемент в списке вычислительных узлов*} RootNode;  
  
  
*// структура топологии*typedef struct {  
 RootNode\* root; *// первый элемент списка корневых элементов*} Topology;  
  
  
*// структура итератора по топологии  
// @note нужен, чтобы упростить обход всех вычислительных узлов в топологии*typedef struct {  
 RootNode\* root\_node; *// текущий корневой элемент* WorkerNode\* worker\_node; *// текущий вычислительный узел*} TopologyIterator;  
  
  
*// Конструктор элемента вычислительного узла  
// @argument node - ссылка на структуру элемента  
// @argument id - id вычислительного узла  
// @argument socket - сокет для общения с вычислительным узлом*void worker\_node\_init(WorkerNode\* node, int id, zsock\_t\* socket)  
{  
 node->id = id;  
 node->socket = socket;  
 node->next = **NULL**;  
 node->life\_points = MAX\_LIFE\_POINTS;  
}  
  
  
*// Деструктор элемента вычислительного узла*void worker\_node\_remove(WorkerNode\* node)  
{  
 zstr\_send(node->socket, "exit");  
 **zsock\_destroy**(&node->socket);  
 node->next = **NULL**;  
}  
  
  
*// Конструктор итератора по топологии*void topology\_iter\_init(TopologyIterator\* iter, Topology\* tp)  
{  
 iter->root\_node = tp->root;  
 iter->worker\_node = (tp->root != **NULL**) ? tp->root->root\_worker : **NULL**;  
}  
  
  
*// Получение текущего вычислительного узла, на который указывает итератор  
// @argument iter - ссылка на итератор  
// @note возвращает NULL, если все элементы были пройдены*WorkerNode\* topology\_iter\_curr(TopologyIterator\* iter)  
{  
 return iter->worker\_node;  
}  
  
  
*// Переход к следующему вычислительному узлу  
// @argument iter - ссылка на итератор*WorkerNode\* topology\_iter\_next(TopologyIterator\* iter)  
{  
 iter->worker\_node = iter->worker\_node->next;  
  
 *// если мы дошли до конца в списке и есть следующий список...* if(iter->worker\_node == **NULL** && iter->root\_node->next != **NULL**)  
 {  
 iter->root\_node = iter->root\_node->next;  
 iter->worker\_node = iter->root\_node->root\_worker;  
 }  
  
 return topology\_iter\_curr(iter);  
}  
  
  
*// Инициализация структуры топологии  
// @argument tp - ссылка на структуру топологии*void topology\_init(Topology\* tp)  
{  
 tp->root = **NULL**;  
}  
  
  
*// Удаление вычислительного узла из списка вместе со всеми дочерними элементами  
// @note под дочерними понимаются все те элементы, которые идут в списке после текущего  
// @argument root - ссылка на текущий элемент списка  
// @argument id - id вычислительного удаляемого узла (при -1 удаляются все элементы)*WorkerNode\* \_\_remove\_worker(WorkerNode\* root, int id)  
{  
 if(root == **NULL**)  
 return **NULL**;  
  
 *// если текущий элемент - тот, который надо удалить...* if(root->id == id || id == -1)  
 {  
 free(zstr\_recv(root->socket));  
 \_\_remove\_worker(root->next, -1);  
 worker\_node\_remove(root);  
 free(root);  
 return **NULL**;  
 }  
  
 root->next = \_\_remove\_worker(root->next, id);  
 return root;  
}  
  
  
*// Удаление вычислительного узла из списка, находящихся среди тех, на которые  
// указывают корневые элементы  
// @argument root - ссылка на текущий корневой элемент  
// @argument id - id вычислительного удаляемого узла (при -1 удаляются все элементы)*RootNode\* \_\_remove\_worker\_from\_root(RootNode\* root, int id)  
{  
 if(root == **NULL**)  
 return **NULL**;  
  
 root->root\_worker = \_\_remove\_worker(root->root\_worker, id);  
  
 *// если корневой элемент ссылается на пустой список, то он больше не нужен* if(root->root\_worker == **NULL**)  
 {  
 RootNode\* ret = root->next;  
 free(root);  
 return ret;  
 }  
  
 root->next = \_\_remove\_worker\_from\_root(root->next, id);  
 return root;  
}  
  
  
*// Добавление корневого элемента, у которого в списке 1 элемент - node  
// @argument tp - ссылка на топологии, куда добавляем  
// @argument node - узел, который будет в списке создаваемого корневого элемента*void \_\_add\_to\_root\_node(Topology\* tp, WorkerNode\* node)  
{  
 RootNode\* new\_root\_node = (RootNode\*) malloc(sizeof(RootNode));  
 new\_root\_node->root\_worker = node;  
  
 new\_root\_node->next = tp->root;  
 tp->root = new\_root\_node;  
}  
  
  
*// Поиск вычислительного узла по id в топологии  
// @argument tp - ссылка на топологию  
// @argument id - id вычислительного узла, который ищем  
// @return ссылку на найденный узел или NULL если узла нет в топологии*WorkerNode\* topology\_search(Topology\* tp, int id)  
{  
 TopologyIterator iter;  
 topology\_iter\_init(&iter, tp);  
  
 for(WorkerNode\* node = topology\_iter\_curr(&iter); node != **NULL**; node = topology\_iter\_next(&iter))  
 if(node->id == id)  
 return node;  
  
 return **NULL**;  
}  
  
  
*// Добавление вычислительного узла в топологию  
// @argument tp - ссылка на топологию, в которую добавляем узел  
// @argument id - id создаваемого вычислительного узла  
// @argument parent\_id - id родительского вычислительного узла (0, если родительский узел - управляющий)  
// @return OK - элемент успешно добавлен  
// @return ALREADY\_EXISTS - узел с таким id уже существует  
// @return NOT\_EXISTS - узла с id parent\_id не существует*int topology\_add(Topology\* tp, int id, int parent\_id)  
{  
 if(topology\_search(tp, id) != **NULL**)  
 return ALREADY\_EXISTS;  
  
 WorkerNode\* new\_node = (WorkerNode\*) malloc(sizeof(WorkerNode));  
 worker\_node\_init(new\_node, id, **NULL**);  
  
 if(parent\_id == 0)  
 {  
 \_\_add\_to\_root\_node(tp, new\_node);  
 return OK;  
 }  
  
 WorkerNode\* parent = topology\_search(tp, parent\_id);  
  
 if(parent == **NULL**)  
 {  
 free(new\_node);  
 return NOT\_EXISTS;  
 }  
  
 new\_node->next = parent->next;  
 parent->next = new\_node;  
  
 return OK;  
}  
  
  
*// Удаление вычилсительного узла из топологии  
// @argument tp - ссылка на топологию, в которой будем удалять  
// @argument id - id удаляемого узла  
// @return NOT\_EXISTS - такого узла не существует  
// @return OK - успешно удален*int topology\_remove(Topology\* tp, int id)  
{  
 if(topology\_search(tp, id) == **NULL**)  
 return NOT\_EXISTS;  
 tp->root = \_\_remove\_worker\_from\_root(tp->root, id);  
 return OK;  
}  
  
  
*// Деструктор топологии  
// @argument tp - ссылка на топологию, которую удаляем*void topology\_destroy(Topology\* tp)  
{  
 while(tp->root != **NULL**)  
 topology\_remove(tp, tp->root->root\_worker->id); ???  
}  
  
  
*/\* Manager state \*/*static Topology topology;  
  
static zsock\_t\* heartbeat\_socket; *// сокет для получения HEARTBEAT сообщений от вычислительных узлов*long heartbeat\_interval = START\_HEARTBEAT\_INTERVAL; *// интервал, с которым вычислительные узлы отправляют сообщения "я жив"*static zsock\_t\* init\_socket; *// сокет, на который вычислительные узлы отправляют их endpoint после инициализации*pthread\_t decrement\_thread; *// поток декремента очков жизни вычислительных узлов*pthread\_t heartbeat\_listen\_thread; *// поток прослушивания сообщений "я жив" от вычислительных узлов  
  
  
/\* Heartbeat functional \*/  
  
  
// Поток исполнения "засыпает" на msec миллисекунд  
// @note Вспомогательная функция. Аналог sleep только для миллисекунд*int msleep(long msec)  
{  
 struct timespec ts;  
 int res;  
  
 ts.tv\_sec = msec / 1000;  
 ts.tv\_nsec = (msec % 1000) \* 1000000;  
  
 do res = nanosleep(&ts, &ts);  
 while (res && **errno** == **EINTR**);  
  
 return res;  
}  
  
  
*// Код для потока прослушивания вычислительного узла*void\* heartbeat\_listen\_thread\_func()  
{  
 while(1)  
 {  
 char\* active\_worker\_id = zstr\_recv(heartbeat\_socket);  
 if(active\_worker\_id == **NULL**)  
 break;  
  
 zstr\_send(heartbeat\_socket, "OK");  
  
 WorkerNode\* worker\_node = topology\_search(&topology, atoi(active\_worker\_id));  
 if(worker\_node != **NULL**) worker\_node->life\_points = 4;  
  
 zstr\_free(&active\_worker\_id);  
 }  
  
 return **NULL**;  
}  
  
  
*// Код для потока уменьшения очков жизни у вычислительных узлов*void\* heartbeat\_decrement\_thread\_func()  
{  
 TopologyIterator iter;  
  
 while(1)  
 {  
 msleep(heartbeat\_interval);  
 topology\_iter\_init(&iter, &topology);  
  
 for(WorkerNode\* node = topology\_iter\_curr(&iter); node != **NULL**; node = topology\_iter\_next(&iter))  
 {  
 if(node->life\_points <= 0) continue;  
 if(--node->life\_points == 0)  
 printf("Heartbeat: node %d is unavailable now\n", node->id);  
 }  
 }  
  
 return **NULL**;  
}  
  
  
*/\* MAIN FUNCTIONAL \*/  
  
  
// Создание дочернего процесса - вычислительного узла  
// @argument worker\_id - id создаваемого вычислительного узла*pid\_t \_\_create\_worker\_process(int worker\_id)  
{  
 pid\_t id = fork();  
 if(id != 0) return id;  
  
 char endpoint\_arg[256], worker\_id\_arg[256];  
 const char\* manager\_endpoint = zsock\_endpoint(init\_socket);  
  
 **strcpy**(endpoint\_arg, manager\_endpoint);  
 **sprintf**(worker\_id\_arg, "%d", worker\_id);  
  
 char\* args[] = {WORKER\_EXEC\_PATH, endpoint\_arg, worker\_id\_arg, **NULL**};  
  
 execv(WORKER\_EXEC\_PATH, args);  
 exit(1); *// если запустить код узла не получилось - аварийно выходим*}  
  
  
*/\* Usecases \*/*int create\_worker(int worker\_id, int parent\_id, pid\_t\* created\_worker\_pid)  
{  
 pid\_t pid;  
  
 int res = topology\_add(&topology, worker\_id, parent\_id);  
 if(res != OK) return res;  
  
 *// создаем новый процесс вычислительного узла* if(parent\_id == 0)  
 pid = \_\_create\_worker\_process(worker\_id);  
  
 else  
 {  
 char worker\_id\_arg[256];  
 WorkerNode\* parent\_node;  
 char\* reply;  
  
 parent\_node = topology\_search(&topology, parent\_id);  
 **sprintf**(worker\_id\_arg, "%d", worker\_id);  
  
 zstr\_sendx(parent\_node->socket, "create", zsock\_endpoint(init\_socket), worker\_id\_arg, **NULL**);  
 reply = zstr\_recv(parent\_node->socket);  
  
 if(reply == **NULL**)  
 {  
 topology\_remove(&topology, worker\_id);  
 return UNACTIVE;  
 }  
  
 sscanf(reply, "%d", &pid);  
 zstr\_free(&reply);  
 }  
  
 *// Инициализируем сокет для общения с вычислительным узлом* char\* worker\_endpoint = zstr\_recv(init\_socket);  
 zstr\_send(init\_socket, "OK");  
  
 WorkerNode\* worker\_node = topology\_search(&topology, worker\_id);  
 worker\_node->socket = **zsock\_new\_req**(worker\_endpoint);  
 zsock\_set\_rcvtimeo(worker\_node->socket, MAX\_TIMEOUT);  
 zsock\_set\_sndtimeo(worker\_node->socket, MAX\_TIMEOUT);  
  
 *// Кидаем ему запрос, чтобы он присылал сообщения "я жив"* char interval\_arg[256];  
  
 **sprintf**(interval\_arg, "%ld", heartbeat\_interval);  
 zstr\_sendx(worker\_node->socket, "heartbeat", interval\_arg , zsock\_endpoint(heartbeat\_socket), **NULL**);  
 free(zstr\_recv(worker\_node->socket));  
  
 \*created\_worker\_pid = pid;  
 return 0;  
}  
  
  
int set\_heartbeat\_interval(long interval)  
{  
 char interval\_arg[256];  
 TopologyIterator iter;  
  
 **sprintf**(interval\_arg, "%ld", interval); topology\_iter\_init(&iter, &topology);  
 heartbeat\_interval = interval;  
  
 for(WorkerNode\* node = topology\_iter\_curr(&iter); node != **NULL**; node = topology\_iter\_next(&iter))  
 {  
 zstr\_sendx(node->socket, "heartbeat", interval\_arg , zsock\_endpoint(heartbeat\_socket), **NULL**);  
 free(zstr\_recv(node->socket));  
 }  
  
 return OK;  
}  
  
  
void print\_node\_states()  
{  
 TopologyIterator iter;  
 topology\_iter\_init(&iter, &topology);  
  
 for(WorkerNode\* node = topology\_iter\_curr(&iter); node != **NULL**; node = topology\_iter\_next(&iter))  
 printf("%d\t|\t%s\n", node->id, (node->life\_points) ? "online" : "offline");  
}  
  
  
int main()  
{  
 *// Инициализация сокетов* init\_socket = **zsock\_new\_rep**("tcp://0.0.0.0:\*");  
 heartbeat\_socket = **zsock\_new\_rep**("tcp://0.0.0.0:\*");  
  
 zsock\_set\_rcvtimeo(init\_socket, MAX\_TIMEOUT);  
 zsock\_set\_sndtimeo(init\_socket, MAX\_TIMEOUT);  
  
 topology\_init(&topology);  
  
 pthread\_create(&heartbeat\_listen\_thread, **NULL**, heartbeat\_listen\_thread\_func, **NULL**);  
 pthread\_create(&decrement\_thread, **NULL**, heartbeat\_decrement\_thread\_func, **NULL**);  
  
 while(1)  
 {  
 char command\_type[256];  
 printf(">>> ");  
 scanf("%s", command\_type);  
  
 if(!strcmp(command\_type, "create"))  
 {  
 int worker\_id, parent\_id, res;  
 pid\_t created\_worker\_pid;  
  
 scanf("%d %d", &worker\_id, &parent\_id);  
 res = create\_worker(worker\_id, parent\_id, &created\_worker\_pid);  
  
 if(res == OK)  
 printf("Ok: %d\n", created\_worker\_pid);  
 if(res == ALREADY\_EXISTS)  
 printf("Error: Already exists\n");  
 if(res == NOT\_EXISTS)  
 printf("Error: Parent not exists\n");  
 if(res == UNACTIVE)  
 printf("Error: Parent is unavailable\n");  
 }  
  
 else if(!strcmp(command\_type, "remove"))  
 {  
 int worker\_id, result;  
  
 scanf("%d", &worker\_id);  
 result = topology\_remove(&topology, worker\_id);  
  
 if(result == NOT\_EXISTS)  
 printf("Error: Not found\n");  
 if(result == OK)  
 printf("Ok\n");  
 }  
  
 else if(!strcmp(command\_type, "heartbeat"))  
 {  
 long interval;  
 scanf("%ld", &interval);  
  
 set\_heartbeat\_interval(interval);  
 }  
  
 else if(!strcmp(command\_type, "exec"))  
 {  
 int worker\_id;  
 char method[256];  
 WorkerNode \*node;  
  
 scanf("%d %s", &worker\_id, method);  
 node = topology\_search(&topology, worker\_id);  
  
 if(node == **NULL**)  
 {  
 printf("Error:%d: Not found\n", worker\_id);  
 continue;  
 }  
  
 if(!strcmp(method, "start"))  
 {  
 zstr\_send(node->socket, "timer-start");  
 char\* reply = zstr\_recv(node->socket);  
  
 if(reply == **NULL**)  
 printf("Error:%d: Node is unavailable\n", worker\_id);  
 else  
 printf("Ok:%d\n", worker\_id);  
 }  
  
 else if(!strcmp(method, "stop"))  
 {  
 zstr\_send(node->socket, "timer-stop");  
 char\* reply = zstr\_recv(node->socket);  
  
 if(reply == **NULL**)  
 printf("Error:%d: Node is unavailable\n", worker\_id);  
 else  
 printf("Ok:%d\n", worker\_id);  
 }  
  
 else if(!strcmp(method, "time"))  
 {  
 zstr\_send(node->socket, "timer-time");  
 char\* reply = zstr\_recv(node->socket);  
  
 if(reply == **NULL**)  
 printf("Error:%d: Node is unavailable\n", worker\_id);  
 else  
 printf("Ok:%d: %s\n", worker\_id, reply);  
 }  
  
 else  
 printf("Error:%d: Unknown method", worker\_id);  
 }  
  
 else if(!strcmp(command\_type, "heartbeat-info"))  
 print\_node\_states();  
  
 else if(!strcmp(command\_type, "exit"))  
 break;  
  
 else  
 printf("Error: Unknown command\n");  
 }  
  
 pthread\_cancel(heartbeat\_listen\_thread);  
 pthread\_cancel(decrement\_thread);  
  
 topology\_destroy(&topology);  
  
 **zsock\_destroy**(&init\_socket);  
 **zsock\_destroy**(&heartbeat\_socket);  
  
 return 0;  
}

**Worker.c:**

#include "log/log.h"  
#include <stdlib.h> *// exit*#include <stdio.h> *// scanf*#include <unistd.h> *// execl, fork, pid\_t*#include <errno.h> *// errno*#include <string.h> *// strerror*#include <time.h> *// timespec, nanosleep*#include <pthread.h> *// pthread\_create, pthread\_t, pthread\_exit, pthread\_join*#include <czmq.h>  
  
  
static char \*WORKER\_EXEC\_PATH = "worker"; *// путь к исполняемому файлу вычислительного узла  
  
  
/\* Create worker node functional \*/  
  
  
// Создание дочернего процесса - вычислительного узла  
// @argument worker\_id - id создаваемого вычислительного узла  
// @argument manager\_endpoint - endpoint сокета, которому созданный узел отправит сгенерированный endpoint  
// @note аналог функции \_\_create\_worker\_process в менеджере*pid\_t create\_worker(int worker\_id, char \*manager\_endpoint) {  
 pid\_t id = fork();  
 if (id != 0) return id;  
  
 char worker\_id\_str[256];  
 **sprintf**(worker\_id\_str, "%d", worker\_id);  
  
 char \*args[] = {WORKER\_EXEC\_PATH, manager\_endpoint, worker\_id\_str, **NULL**};  
 execv(WORKER\_EXEC\_PATH, args);  
  
 exit(1);  
}  
  
  
*/\* Timer functional \*/  
  
  
// Структура таймера*typedef struct {  
 unsigned long elapsed\_time; *// сколько времени насчитал таймер* unsigned long last\_timemark; *// последняя временная точка старта таймера* **bool** is\_running; *// запущен ли сейчас таймер?*} Timer;  
  
  
*// Получение текущего времени в миллисекундах*unsigned long \_\_get\_current\_time() {  
 struct timeval timemark;  
 gettimeofday(&timemark, **NULL**);  
 return timemark.tv\_sec \* 1000LL + timemark.tv\_usec / 1000;  
}  
  
  
*// Конструктор таймера  
// @argument timer - ссылка на структуру таймера  
// @note при повторном вызове конструктора обновляет таймер*void timer\_init(Timer \*timer) {  
 timer->elapsed\_time = 0;  
 timer->last\_timemark = \_\_get\_current\_time();  
 timer->is\_running = **false**;  
}  
  
  
*// Запускает таймер  
// @argument timer - ссылка на таймер*void timer\_start(Timer \*timer) {  
 if (timer->is\_running)  
 return;  
  
 timer->last\_timemark = \_\_get\_current\_time();  
 timer->is\_running = **true**;  
}  
  
  
*// Приостанавливает таймер  
// @argument timer - ссылка на таймер*void timer\_stop(Timer \*timer) {  
 if (!timer->is\_running)  
 return;  
  
 unsigned long current\_time = \_\_get\_current\_time();  
 timer->elapsed\_time += current\_time - timer->last\_timemark;  
 timer->is\_running = **false**;  
}  
  
  
*// Получение количества миллисекунд, которые насчитал таймер  
// @argument timer - ссылка на таймер*unsigned long timer\_time(Timer \*timer) {  
 if (timer->is\_running) {  
 unsigned long current\_time = \_\_get\_current\_time();  
 return timer->elapsed\_time + (current\_time - timer->last\_timemark);  
 }  
  
 return timer->elapsed\_time;  
}  
  
  
*/\* Heartbeat functional \*/  
  
  
// Тип данных функции, которая будет вызываться при отправке  
// сообщения "я жив"*typedef void (\*CallbackFunction)(void);   
  
*// Структура HEARTBEAT*typedef struct {  
 long polling\_interval; *// интервал, с которым вызывается callback функция* pthread\_t polling\_thread; *// поток отправки сообщений "я жив"* CallbackFunction callback; *// ссылка на callback функцию*} Heartbeat;  
  
  
*// Поток исполнения "засыпает" на msec миллисекунд  
// @note Вспомогательная функция. Аналог sleep только для миллисекунд*int msleep(long msec) {  
 struct timespec ts;  
 int res;  
  
 ts.tv\_sec = msec / 1000;  
 ts.tv\_nsec = (msec % 1000) \* 1000000;  
  
 do res = nanosleep(&ts, &ts);  
 while (res && **errno** == **EINTR**);  
  
 return res;  
}  
  
  
*// Функция потока отправки сообщений "я жив"*void \*\_\_polling\_thread\_func(void \*\_hb) { ???  
 Heartbeat \*hb = (Heartbeat \*) \_hb;  
  
 while (1) {  
 *// при -1 останавливаем поток* if (hb->polling\_interval == -1)  
 break;  
  
 msleep(hb->polling\_interval);  
 hb->callback();  
 }  
  
 return **NULL**;  
}  
  
  
*// Конструктор HEARTBEAT  
// @argument hb - ссылка на структуру HEARTBEAT  
// @argument callback - ссылка на callback функцию*void heartbeat\_init(Heartbeat \*hb, CallbackFunction callback) {  
 hb->polling\_interval = -1; *// маркер -1 означает, что поток отправки сообщений еще не создан* hb->callback = callback;  
}  
  
  
*// Начало отправки сообщений "я жив"  
// @argument hb - ссылка на структуру HEARTBEAT  
// @argument interval - интервал (мс), с которым отправляем сообщения*void heartbeat\_start\_polling(Heartbeat \*hb, long interval) {  
 *// если поток был создан - останавливаем его* if (hb->polling\_interval != -1) {  
 hb->polling\_interval = -1;  
 pthread\_join(hb->polling\_thread, **NULL**);  
 }  
  
 hb->polling\_interval = interval;  
 pthread\_create(&hb->polling\_thread, **NULL**, \_\_polling\_thread\_func, (void \*) hb);  
}  
  
  
*// Деструктор HEARTBEAT  
// @argument hb - ссылка на структуру HEARTBEAT*void heartbeat\_remove(Heartbeat \*hb) {  
 hb->polling\_interval = -1;  
 pthread\_join(hb->polling\_thread, **NULL**);  
}  
  
  
*/\* Main functional \*/*static zsock\_t \*input\_socket; *// сокет для прослушивания входных сообщений и отправки ответов*static zsock\_t \*heartbeat\_socket = **NULL**; *// сокет, на который узел отправляет сообщения "я жив"*int id; *// id вычислительного узла*static Heartbeat hb; *// объект HEARTBEAT*static Timer timer; *// таймер вычислительного узла  
  
  
// Отправляет сообщение "я жив" через ZMQ сокет менеджеру*void send\_alive\_message() {  
 zstr\_sendf(heartbeat\_socket, "%d", id);  
 free(zstr\_recv(heartbeat\_socket));  
}  
  
  
*// Отправляет результат инициализации вычислительного узла менеджеру  
// @argument manager\_endpoint - endpoint менеджера  
// @argument worker\_endpoint - endpoint сокета, по которому узел будет получать сообщения*void send\_response\_to\_manager(char \*manager\_endpoint, const char \*worker\_endpoint) {  
 zsock\_t \*manager\_socket = zsock\_new\_req(manager\_endpoint);  
  
 zstr\_send(manager\_socket, worker\_endpoint);  
 free(zstr\_recv(manager\_socket));  
  
 zsock\_destroy(&manager\_socket);  
}  
  
  
int main(int argc, char \*argv[]) {  
 id = atoi(argv[2]);  
  
 input\_socket = zsock\_new\_rep("tcp://0.0.0.0:\*");  
 send\_response\_to\_manager(argv[1], zsock\_endpoint(input\_socket));  
  
 heartbeat\_init(&hb, send\_alive\_message);  
 timer\_init(&timer);  
  
 while (1) {  
 char \*request\_type = zstr\_recv(input\_socket);  
  
 if (!strcmp(request\_type, "create")) {  
 char \*manager\_endpoint = zstr\_recv(input\_socket);  
 char \*worker\_id\_arg = zstr\_recv(input\_socket);  
 pid\_t pid = create\_worker(atoi(worker\_id\_arg), manager\_endpoint);  
  
 zstr\_sendf(input\_socket, "%d", pid);  
  
 zstr\_free(&manager\_endpoint);  
 zstr\_free(&worker\_id\_arg);  
 } else if (!strcmp(request\_type, "heartbeat")) {  
 char \*interval\_arg, \*heartbeat\_endpoint;  
 long interval;  
  
 interval\_arg = zstr\_recv(input\_socket);  
 heartbeat\_endpoint = zstr\_recv(input\_socket);  
 zstr\_send(input\_socket, "OK");  
  
 sscanf(interval\_arg, "%ld", &interval);  
  
 *// Подключаемся к HEARTBEAT сокету менеджера  
 // Если ранее были подключены к другому сокету - разрываем соединение* if (heartbeat\_socket != **NULL**)  
 zsock\_destroy(&heartbeat\_socket);  
 heartbeat\_socket = zsock\_new\_req(heartbeat\_endpoint);  
  
 heartbeat\_start\_polling(&hb, interval);  
  
 zstr\_free(&interval\_arg);  
 zstr\_free(&heartbeat\_endpoint);  
 } else if (!strcmp(request\_type, "exit")) {  
 zstr\_send(input\_socket, "OK");  
 break;  
 } else if (!strcmp(request\_type, "timer-start")) {  
 timer\_start(&timer);  
 zstr\_send(input\_socket, "OK");  
 } else if (!strcmp(request\_type, "timer-stop")) {  
 timer\_stop(&timer);  
 zstr\_send(input\_socket, "OK");  
 } else if (!strcmp(request\_type, "timer-time")) {  
 unsigned long time = timer\_time(&timer);  
 zstr\_sendf(input\_socket, "%lu", time);  
 }  
  
 zstr\_free(&request\_type);  
 }  
  
 heartbeat\_remove(&hb);  
 zsock\_destroy(&input\_socket);  
 zsock\_destroy(&heartbeat\_socket);  
  
 return 0;  
}

**Пример работы**

kirillarmishev@2 bin % ./manager

>>> create 1 0

Ok: 70670

>>> create 2 1

Ok: 70676

>>> heartbeat 2000

>>> exec 1 time

Ok:1: 0

>>> exec 1 start

Ok:1

>>> exec 2 start

Ok:2

>>> exec 1 time

Ok:1: 11131

>>> exec 2 time

Ok:2: 6714

>>> exec 1 stop

Ok:1

>>> exec 1 time

Ok:1: 17688

>>> exec 1 start

Ok:1

>>> exec 1 stop

Ok:1

>>> exec 1 time

Ok:1: 21457

>>> heartbeat-info

1 | online

2 | online

>>> Heartbeat: node 2 is unavailable now

heartbeat-info

1 | online

2 | offline

>>> remove 2

Ok

**Вывод**

В ходе выполнения лабораторной работы я получил опыт работы с библиотекой ZeroMQ, ознакомился с основными концептами идеи сервера очереди сообщений. Также, я изучил основные классификации очередей сообщений: синхронное или асинхронное взаимодействие, сохранение на диск, пересылка по сети, использование брокера.

Использование очереди сообщений в вычислительных сетях заметно повышает ее эффективность, однако, добавляет не меньше трудностей, так как теперь возникает необходимость в мониторинге системы в реальном времени, чтобы отслеживать моменты, когда какие-либо узлы стали неработоспособными. В вычислительных сетях основная трудность заключается в организации обмена сообщениями, так как при отправке мы не можем быть уверенными, что ответ нам придет, или что сообщение вообще пришло получателю.

По своей идее архитектура вычислительных сетей очень похожа на микросервисную архитектуру. Отличие заключается лишь в назначении узлов. В микросервисной архитектуре каждый узел имеет более широкое применение, так как выполнять они могут разные задачи, в то время как в вычислительных сетях все узлы по своей сути являются простыми репликами.