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

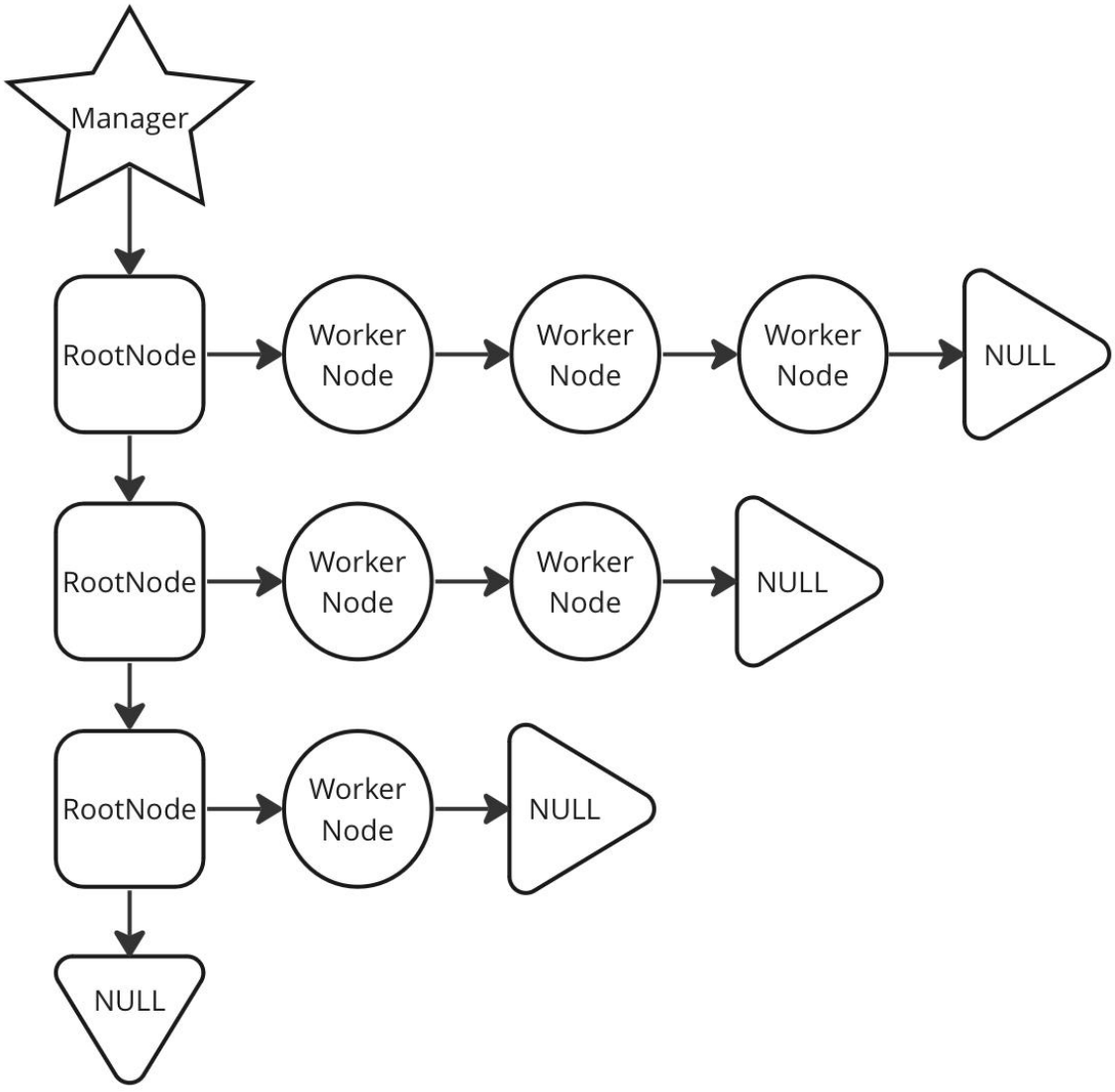
Основная программа компилируется из файла manager.c. Также подключаются файлы zmq_struct.c — функции для работы с очередью. worker.c представляет собой вычислительную ноду к которой также подключается zmq_struct.c.

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>
```

```c
#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);
}
```

```c
/* 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;
}
```

```c
/* 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.