

Московский Авиационный Институт  
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики  
Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу  
«Операционные системы»**

**Сервера сообщений**

Студент	Гришин Алексей Юрьевич
Группа	М8О-208Б-21
Вариант	43
Преподаватель	Соколов Андрей Алексеевич
Оценка	5
Дата	29.12.2022
Подпись	

Москва, 2022.

# Постановка задачи

## Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений
- Применение отложенных вычислений
- Интеграция программных систем друг с другом

## 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 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: `create <id> [<parent>]`
- Исполнение команды на вычислительном узле: `exec <id> [<param>]`
- Проверка доступности узла (формат указан в варианте)
- Удаление узла: `remove <id>`

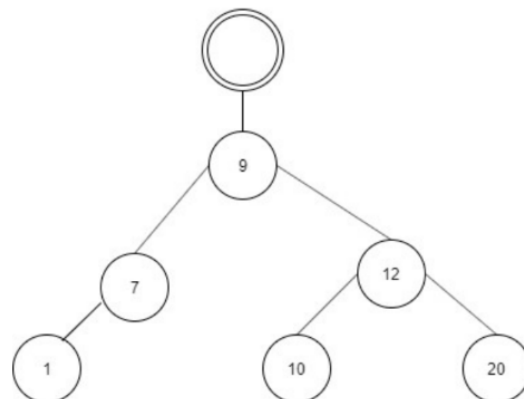
## Вариант 43

### Топология

Все вычислительные узлы находятся в идеально сбалансированном бинарном дереве.

### Набор команд

Необходимо реализовать таймер. Формат команды сохранения значения:



`exec id subcommand`

где, `subcommand` – одна из трех команд:

1. `start` – запустить таймер
2. `stop` – остановить таймер
3. `time` – показать время локального таймера в миллисекундах

## Команда проверки

Формат команды: `pingall`

Вывод всех недоступных узлов вывести разделенные через точку запятой.

## Общие сведения о программе

Основная команда компилируется из файла `main.c`. Использует следующие вспомогательные модули: `worker_tree`, `worker`, `queue`, `user_commands`, `mq`. Также, используются заголовочные файлы `string.h`, `sys/wait.h`, `stdio.h`.

Программа, представляющая исполнительный узел, компилируется из файла `node.c` и использует вспомогательные модули `mq`, `worker`, `time`. Также, используются заголовочные файлы `stdlib`, `string`, `unistd.h`, `sys/wait.h`, `stdio.h`.

## Системные 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 сокет 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.

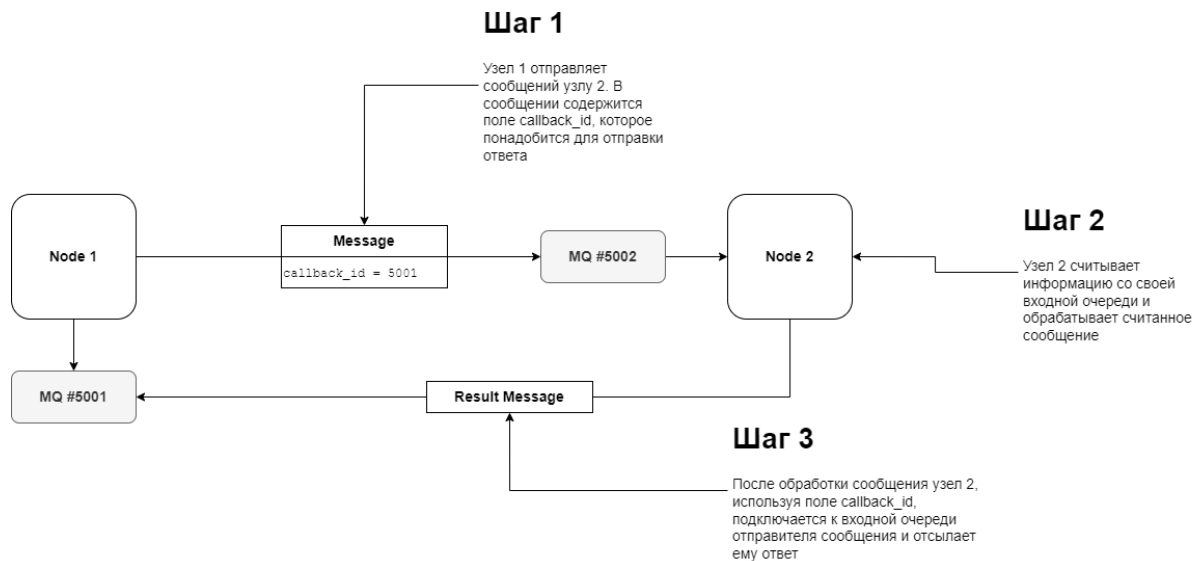
8. `zmq_msg_data(msg)` – возвращает указатель на начало области памяти, где хранятся данные сообщения
9. `zmq_msg_send(msg, socket, flag)` – отправляет сообщение `msg` по сокету `socket`. Дополнительно можно указать опции в параметре `flag`. Например, флаг `ZMQ_DONTWAIT` устанавливает операции неблокирующий 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. `zmq_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.

## Описание

Библиотека ZeroMQ основана на сокетах. Сокеты – один из способов межпроцессорного взаимодействия, который позволяет общаться процессам как на одной ЭВМ, так и на разных. Главное – чтобы они были соединены сетью. Но, из названия следует, что библиотека должна быть связана с очередью сообщений, а мы взаимодействуем лишь с сокетами. Дело в том, что при установлении соединения между двумя сокетами создается очередь сообщений, в которую и записываются или считываются сообщения. Иными словами, сокеты являются лишь оберткой для работы с очередями сообщений.

Для общения по сети сокеты используют стек протоколов TCP / IP, следуя которому каждому узлу сети соответствует пара из адреса и порта. **Адрес** – битовая структура, размер которой зависит от версии протокола. Так, для протокола IPv4 адрес имеет размер в 32 бита, а для IPv6 – 128 битов. **Порт** – целое число в диапазоне от  $[0; 2^{16} - 1]$ . Порты созданы с целью того, чтобы при сгруппировать сообщения, посылаемые ЭВМ, на принадлежность каждому процессу, чтобы не возникло ситуации, что процесс А считал сообщение, предназначенное процессу В.

Для организации общения между узлами вычислительной системы я использовал паттерн обмена сообщениями Pipeline. При таком паттерне мы ограничиваем все сокеты, участвующие в обмене сообщения на PUSH и PULL сокеты. Задача PUSH сокетов заключается в отправке сообщений, а задача PULL сокетов – в чтении отправленных сообщений. По моему мнению, такой паттерн ближе всего относится к идее очереди сообщений. Процесс обмена сообщениями между двумя узлами приведен ниже.



При создании очереди сообщений ее идентификатор генерируется самостоятельно. Если посмотреть на реализацию создания очередей сообщений, то мы увидим, что на самом деле создается PULL сокет, порт которого подбирается автоматически. Идентификатор очереди на самом деле является портом, который прослушивает соответствующий сокет.

Чтобы вычислительная система следовала топологии, указанной в варианте задания, я выстроил соответствующую иерархии из вычислительных узлов. То есть, если в топологии указано, что узел 2 является дочерним по отношению к узлу 1, то это же отношение иерархии сохраняется и на их процессы (процесс, соответствующий узлу 1, является родительским по отношению к процессу, соответствующему узлу 2). Отправка сообщения осуществляется через broadcast. То есть, изначально корневой узел отправляет сообщение, полученное от пользователя, своим дочерним узлам. Далее, дочерние узлы, получив сообщение от родительского либо обрабатывают его и возвращают результат обратно родительскому узлу, либо передают сообщение своим дочерним узлам и ждут ответа от них. Если при получении ответов от дочерних узлов оба оказались отрицательными, то только в этом случае обратно возвращается отрицательный ответ.

Проверка на работоспособность узла осуществляется через timeout. Устанавливается определенное количество миллисекунд, после истечения которых узел считается неактивным.

Также, было создано “мнимое” представление топологии узлов в виде структуры данных бинарного дерева. Такое дерево будет играть роль виртуальной копии иерархии узлов, что позволит быстро определить, какому узлу необходимо послать запрос на создание дочернего узла, при этом поддерживая сбалансированность двоичного дерева.

## Исходный код

### mq/mq.c

```
#include "mq.h"
#include <zmq.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <error.h>

int listeners_counter = 0;
static char * __mq_server_address = "0.0.0.0";
static int __max_endpoint_size = 256;

#define HANDLE_ERROR \
    exit(errno)

int __auto_bind(void * socket) {
    char endpoint[__max_endpoint_size];
    char addr[__max_endpoint_size];
    size_t endpoint_size = __max_endpoint_size;
    int port;

    sprintf(endpoint, "tcp://%s:*", __mq_server_address);

    if (zmq_bind(socket, endpoint) != 0)
        return -1;

    zmq_getsockopt(socket, ZMQ_LAST_ENDPOINT, & endpoint, & endpoint_size);
    sscanf(endpoint, "tcp://0.0.0.0:%d", & port);
    # // TODO: добавить зависимость от __mq_server_address:

    return port;
}

void mq_create(MessageQueue * mq) {
    mq -> context = zmq_ctx_new();
    mq -> socket = zmq_socket(mq -> context, ZMQ_PULL);
    zmq_setsockopt(mq -> socket, ZMQ_IDENTITY, & listeners_counter,
sizeof(listeners_counter));
    listeners_counter++;

    mq -> id = __auto_bind(mq -> socket);

    if (mq -> id == -1) {
        HANDLE_ERROR;
    }
}
```

```

void mq_connect(MessageQueue * mq, int mq_id) {
    char endpoint[__max_endpoint_size];

    mq -> context = zmq_ctx_new();
    mq -> socket = zmq_socket(mq -> context, ZMQ_PUSH);
    mq -> id = mq_id;

    sprintf(endpoint, "tcp://%s:%d", __mq_server_address, mq_id);
    int res = zmq_connect(mq -> socket, endpoint);
    assert(res == 0);
}

int mq_id(MessageQueue * mq) {
    return mq -> id;
}

void mq_send(MessageQueue * mq, void * data, size_t size) {
    zmq_msg_t message;
    int res;

    if (zmq_msg_init( & message) != 0) {
        HANDLE_ERROR;
    }
    if (zmq_msg_init_size( & message, size) != 0) {
        HANDLE_ERROR;
    }

    memcpy(zmq_msg_data( & message), data, size);

    if (zmq_msg_send( & message, mq -> socket, 0) == -1) {
        HANDLE_ERROR;
    }
    if (zmq_msg_close( & message) != 0) {
        HANDLE_ERROR;
    }
}

void mq_recv(MessageQueue * mq, void ** buff, size_t * size) {
    zmq_msg_t message;
    int res;

    if (zmq_msg_init( & message) != 0) {
        HANDLE_ERROR;
    }
    if (zmq_msg_recv( & message, mq -> socket, 0) == -1) {
        if (errno == EAGAIN) {
            * buff = NULL;
            * size = -1;
            return;
        } else {
            HANDLE_ERROR;
        }
    }
}

```

```

    }

    * size = zmq_msg_size( & message);
    * buff = malloc( * size);
    memcpy( * buff, zmq_msg_data( & message), * size);

    if (zmq_msg_close( & message) != 0) {
        HANDLE_ERROR;
    }
}

void mq_set_timeout(MessageQueue * mq, int timeout) {
    int res;
    if (zmq_setsockopt(mq -> socket, ZMQ_RCVTIMEO, & timeout,
sizeof(timeout))) {
        HANDLE_ERROR;
    }
    if (zmq_setsockopt(mq -> socket, ZMQ_SNDTIMEO, & timeout,
sizeof(timeout))) {
        HANDLE_ERROR;
    }
}

void mq_close(MessageQueue * mq) {
    zmq_close(mq -> socket);
    zmq_ctx_destroy(mq -> context);
}

```

## mq/mq.h

```

#ifndef __MESSAGE_QUERY_H__
#define __MESSAGE_QUERY_H__

#include <stddef.h>

#define WORKER_OK 0
#define WORKER_ERROR_TIMEOUT 1
#define WORKER_ERROR_INTERNAL 2

typedef struct {
    int id;
    void * socket;
    void * context;
} MessageQueue;

void mq_create(MessageQueue * mq);
void mq_connect(MessageQueue * mq, int mq_id);
int mq_id(MessageQueue * mq);
void mq_send(MessageQueue * mq, void * data, size_t size);
void mq_recv(MessageQueue * mq, void ** buff, size_t * size);

```



```
void mq_set_timeout(MessageQueue * mq, int timeout);
void mq_close(MessageQueue * mq);

#endif
```

## queue/queue.c

```
#include "queue.h"

void queue_init(Queue * q) {
    q -> head = NULL;
}

void queue_push(Queue * q, void * value, size_t value_size) {
    QElement * element = (QElement * ) malloc(sizeof(QElement));
    element -> value = value;
    element -> value_size = value_size;
    element -> next = NULL;

    QElement * head = q -> head;
    if (head == NULL) {
        q -> head = element;
        return;
    }

    while (head -> next != NULL) head = head -> next;
    head -> next = element;
}

bool queue_empty(Queue * q) {
    return q -> head == NULL;
}

void queue_pop(Queue * q, void ** value, size_t * value_size) {
    void * res_value = NULL;
    size_t res_size = -1;

    if (q -> head != NULL) {
        QElement * head = q -> head;
        q -> head = q -> head -> next;
        res_value = head -> value;
        res_size = head -> value_size;
        free(head);
    }

    if (value != NULL) * value = res_value;
    if (value_size != NULL) * value_size = res_size;
}

void queue_delete(Queue * q) {
```

```
    while (!queue_empty(q)) queue_pop(q, NULL, NULL);
}
```

## queue/queue.h

```
#ifndef __QUEUE_H__
#define __QUEUE_H__

#include <stdlib.h>

#include <stdbool.h>

typedef struct __QElement {
    void * value;
    size_t value_size;
    struct __QElement * next;
} QElement;

typedef struct {
    QElement * head;
} Queue;

void queue_init(Queue * q);
void queue_push(Queue * q, void * value, size_t value_size);
void queue_pop(Queue * q, void ** value, size_t * value_size);
bool queue_empty(Queue * q);
void queue_delete(Queue * q);

#endif
```

## timer/timer.c

```
#include "timer.h"
#include <sys/time.h>
#include <stddef.h>

Timemark __get_time() {
    struct timeval timemark;
    gettimeofday( & timemark, NULL);
    return timemark.tv_sec * 1000 LL + timemark.tv_usec / 1000;
}

void __timer_update(Timer * timer) {
    Timemark curr_timemark = __get_time();
    timer -> passed_time += (curr_timemark - timer -> last_timemark);
    timer -> last_timemark = curr_timemark;
}
```

```

void timer_init(Timer * timer) {
    timer -> passed_time = 0;
    timer -> last_timemark = __get_time();
    timer -> is_active = 0;
}

void timer_start(Timer * timer) {
    timer -> last_timemark = __get_time();
    timer -> is_active = 1;
}

void timer_stop(Timer * timer) {
    __timer_update(timer);
    timer -> is_active = 0;
}

Milliseconds timer_time(Timer * timer) {
    if (timer -> is_active) __timer_update(timer);
    return timer -> passed_time;
}

```

## timer/timer.h

```

#ifndef __TIMER_H__
#define __TIMER_H__

typedef unsigned long Milliseconds;
typedef unsigned long Timemark;

typedef struct {
    Milliseconds passed_time;
    Timemark last_timemark;
    int is_active;
} Timer;

void timer_init(Timer * timer);
void timer_start(Timer * timer);
void timer_stop(Timer * timer);
Milliseconds timer_time(Timer * timer);

#endif

```

## worker/worker.c

```

#include "worker.h"
#include <zmq.h>
#include <unistd.h>
#include <stdio.h>

```

```

#include <stdlib.h>
#include <assert.h>
#include <string.h>

void __send_unavailable_error(WorkerMessage * message, MessageQueue *
callback) {
    WorkerResult result = {
        .message_id = message -> message_id,
        .worker_id = message -> receiver_id,
        .message_type = message -> type,
        .result_type = RESULT_UNAVAILABLE
    };

    mq_send(callback, & result, sizeof(result));
}

pid_t worker_create(Worker * worker, char * execute_path, int worker_id)
{
    MessageQueue callback;
    pid_t pid;

    mq_create( & callback);

    pid = fork();

    if (pid == 0) {
        char callback_id_str[256];
        char worker_id_str[256];

        sprintf(callback_id_str, "%d", mq_id( & callback));
        sprintf(worker_id_str, "%d", worker_id);

        char * args[] = {
            execute_path,
            worker_id_str,
            callback_id_str,
            NULL
        };
        execv(execute_path, args);

        exit(2);
    }

    if (pid > 0) {
        CreateResult * recv_data;
        size_t recv_data_size;
        pid_t created_worker_pid;

        mq_recv( & callback, (void **) & recv_data, & recv_data_size);
        assert(recv_data != NULL);

        mq_connect( & worker -> mq, recv_data -> worker_mq_id);
    }
}

```

```

    worker -> id = worker_id;
    worker -> pid = recv_data -> worker_pid;

    free(recv_data);
    return worker -> pid;
}

if (pid < 0) {
    exit(2);
}
}

void worker_send_message(Worker * worker, WorkerMessage * message, size_t
message_size) {

    MessageQueue local_callback, callback;
    WorkerMessage * message_copy;
    void * recv_data;
    size_t recv_data_size;

    mq_connect( & callback, message -> callback_id);

    mq_create( & local_callback);
    mq_set_timeout( & local_callback, 1000);

    message_copy = (WorkerMessage * ) malloc(message_size);
    memcpy(message_copy, message, message_size);
    message_copy -> callback_id = mq_id( & local_callback);

    mq_send( & worker -> mq, message_copy, message_size);
    mq_recv( & local_callback, & recv_data, & recv_data_size);

    if (recv_data == NULL)
        __send_unavailable_error(message, & callback);
    else {
        mq_send( & callback, recv_data, recv_data_size);
        free(recv_data);
    }

    free(message_copy);
    mq_close( & local_callback);
    mq_close( & callback);
}

void worker_delete(Worker * worker) {
    mq_close( & worker -> mq);
}

```

## worker/worker.h

```
#ifndef __WORKER_H__
#define __WORKER_H__

#include "../mq/mq.h"
#include "../queue/queue.h"
#include <stdbool.h>
#include <unistd.h>

typedef struct {
    int id;
    MessageQueue mq;
    pid_t pid;
} Worker;

typedef enum {
    CREATE,
    PING,
    DELETE,
    TIMER_START,
    TIMER_STOP,
    TIMER_TIME
} MessageType;

typedef enum {
    RESULT_OK,
    RESULT_UNAVAILABLE,
    RESULT_PING,
    NOT_FOUND,
    ALREADY_EXISTS
} ResultType;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
    MessageType type;
} WorkerMessage;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
    MessageType type;
    int created_worker_id;
} CreateMessage;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
```

```
    MessageType type;
} PingMessage;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
    MessageType type;
} DeleteMessage;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
    MessageType type;
} StartTimerMessage;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
    MessageType type;
} StopTimerMessage;

typedef struct {
    int message_id;
    int receiver_id;
    int callback_id;
    MessageType type;
} GetTimeMessage;

typedef struct {
    int message_id;
    int worker_id;
    MessageType message_type;
    ResultType result_type;
} WorkerResult;

typedef struct {
    int message_id;
    int worker_id;
    MessageType message_type;
    ResultType result_type;
    pid_t worker_pid;
    int worker_mq_id;
} CreateResult;

typedef struct {
    int message_id;
    int worker_id;
    MessageType message_type;
    ResultType result_type;
```

```

    unsigned long time;
} TimeResult;

typedef struct {
    int message_id;
    int worker_id;
    MessageType message_type;
    ResultType result_type;
    Queue unavailable_nodes;
} PingResult;

pid_t worker_create(Worker * worker, char * execute_path, int worker_id);
void worker_send_message(Worker * worker, WorkerMessage * message, size_t
message_size);
void worker_delete(Worker * worker);

#endif

```

### worker\_tree/worker\_tree.c

```

#include "worker_tree.h"
#include <stdlib.h>
#include <stdio.h>

void node_init(Node * node, WorkerId id) {
    node -> id = id;
    node -> left = NULL;
    node -> right = NULL;
}

bool node_full(Node * node) {
    return node -> left != NULL && node -> right != NULL;
}

void node_add(Node * parent, Node * child) {
    if (parent -> left == NULL) parent -> left = child;
    else if (parent -> right == NULL) parent -> right = child;
}

void tree_init(WorkerTree * tree) {
    tree -> root = NULL;
}

Result tree_create_node(WorkerTree * tree, WorkerId worker_id) {
    Result res = {
        .parent = -1,
        .child = worker_id
    };

    if (tree -> root == NULL) {

```



```

    tree -> root = (Node * ) malloc(sizeof(Node));
    node_init(tree -> root, worker_id);
    return res;
}

Queue q;
queue_init( & q);
queue_push( & q, (void * ) tree -> root, sizeof(tree -> root));

while (!queue_empty( & q)) {
    Node * node;
    queue_pop( & q, (void ** ) & node, NULL);

    if (!node_full(node)) {
        Node * c = (Node * ) malloc(sizeof(Node));
        node_init(c, worker_id);
        node_add(node, c);
        queue_delete( & q);

        res.parent = node -> id;
        return res;
    }

    queue_push( & q, node -> left, sizeof(node -> left));
    queue_push( & q, node -> right, sizeof(node -> right));
}

queue_delete( & q);
}

bool __find_node(Node * node, WorkerId worker_id) {
    if (node == NULL) return false;
    if (node -> id == worker_id) return true;

    return __find_node(node -> left, worker_id) || __find_node(node ->
right, worker_id);
}

bool tree_exists(WorkerTree * tree, WorkerId worker_id) {
    return __find_node(tree -> root, worker_id);
}

void __print_node(Node * node, int layer) {
    if (node == NULL) return;

    __print_node(node -> right, layer + 1);
    for (int i = 0; i < layer; i++) printf("\t");
    printf("[Worker #%d]\n", node -> id);
    __print_node(node -> left, layer + 1);
}

void print_tree(WorkerTree * tree) {

```

```

    __print_node(tree -> root, 0);
}

void __delete_node(Node * node) {
    if (node == NULL) return;

    __delete_node(node -> left);
    __delete_node(node -> right);
    free(node);
}

void tree_delete(WorkerTree * tree) {
    __delete_node(tree -> root);
}

Node * __remove_node(Node * node, WorkerId worker_id) {
    if (node == NULL) return NULL;
    if (node -> id == worker_id) {
        __delete_node(node);
        return NULL;
    }
    node -> left = __remove_node(node -> left, worker_id);
    node -> right = __remove_node(node -> right, worker_id);
    return node;
}

void tree_remove_node(WorkerTree * tree, WorkerId worker_id) {
    tree -> root = __remove_node(tree -> root, worker_id);
}

void tree_get_nodes(WorkerTree * tree, Queue * nodes) {
    Queue q;
    queue_init( & q);
    queue_push( & q, (void * ) tree -> root, sizeof(tree -> root));

    while (!queue_empty( & q)) {
        Node * node;
        queue_pop( & q, (void ** ) & node, NULL);

        if (node == NULL) continue;

        queue_push(nodes, (void * ) & node -> id, sizeof(node -> id));
        queue_push( & q, (void * ) node -> left, sizeof(node -> left));
        queue_push( & q, (void * ) node -> right, sizeof(node -> right));
    }

    queue_delete( & q);
}

```

## worker\_tree/worker\_tree.h

```
#ifndef __WORKER_TREE_H__
#define __WORKER_TREE_H__

#include "../queue/queue.h"
#include <stdbool.h>
#include <stdlib.h>

typedef int WorkerId;

typedef struct __Node {
    WorkerId id;
    struct __Node * left;
    struct __Node * right;
} Node;

typedef struct {
    Node * root;
} WorkerTree;

typedef struct {
    WorkerId parent;
    WorkerId child;
} Result;

void tree_init(WorkerTree * tree);
Result tree_create_node(WorkerTree * tree, WorkerId worker_id);
void tree_remove_node(WorkerTree * tree, WorkerId worker_id);
bool tree_exists(WorkerTree * tree, WorkerId worker_id);
void tree_get_nodes(WorkerTree * tree, Queue * nodes);
void print_tree(WorkerTree * tree);
void tree_delete(WorkerTree * tree);

#endif
```

## main.c

```
#include "../worker_tree/worker_tree.h"
#include "../worker/worker.h"
#include "../queue/queue.h"
#include "user_commands.h"
#include "../mq/mq.h"
#include <string.h>
#include <sys/wait.h>
#include <stdio.h>

MessageQueue callback;
Worker root_worker;
WorkerTree tree;
```

```

int command_counter = 1;

WorkerResult base_result(UserCommand cmd, MessageType message_type,
ResultType result_type) {
    WorkerResult result = {
        .message_id = cmd.command_id,
        .worker_id = cmd.worker_id,
        .message_type = message_type,
        .result_type = result_type
    };

    return result;
}

void send_to_output(WorkerResult * result, size_t result_size) {
    MessageQueue output;
    mq_connect( & output, mq_id( & callback));
    mq_send( & output, result, result_size);
    mq_close( & output);
}

void send_not_found_result(UserCommand cmd, MessageType type) {
    WorkerResult result = base_result(cmd, type, NOT_FOUND);
    send_to_output( & result, sizeof(result));
}

void send_already_exists_result(UserCommand cmd, MessageType type) {
    WorkerResult result = base_result(cmd, type, ALREADY_EXISTS);
    send_to_output( & result, sizeof(result));
}

int get_command_id() {
    return command_counter++;
}

UserCommand scan_user_command() {
    char command_type[256];
    int command_id = get_command_id();
    UserCommand command;

    printf("(%d) > ", command_id);
    scanf("%s", command_type);

    if (!strcmp(command_type, "create")) {
        command.type = USER_CREATE;
        scanf("%d", & command.worker_id);
    }

    if (!strcmp(command_type, "remove")) {
        command.type = USER_REMOVE;
        scanf("%d", & command.worker_id);
    }
}

```

```

    if (!strcmp(command_type, "ping")) {
        command.type = USER_PING;
        command.worker_id = -1;
    }

    if (!strcmp(command_type, "exec")) {
        scanf("%d", & command.worker_id);
        char action[256];
        scanf("%s", action);

        if (!strcmp(action, "start")) command.type = USER_EXEC_START;
        if (!strcmp(action, "stop")) command.type = USER_EXEC_STOP;
        if (!strcmp(action, "time")) command.type = USER_EXEC_TIME;
    }

    if (!strcmp(command_type, "close")) {
        command.type = USER_CLOSE;
        command.worker_id = -1;
    }

    command.command_id = command_id;
    return command;
}

void result_type_repr(char * str, ResultType type) {
    switch (type) {
        case RESULT_OK:
            strcpy(str, " : ok");
            break;
        case RESULT_UNAVAILABLE:
            strcpy(str, " : error : node is unavailable");
            break;
        case NOT_FOUND:
            strcpy(str, " : error : not found");
            break;
        case ALREADY_EXISTS:
            strcpy(str, " : error : already exists");
            break;
        case RESULT_PING:
            strcpy(str, " : ok");
            break;
    }
}

void handle_result_messages() {
    WorkerResult * result;
    size_t result_size;
    char result_type_str[256];

    mq_rcv( & callback, (void ** ) & result, & result_size);
    result_type_repr(result_type_str, result -> result_type);
}

```

```

    printf("[Main] <<< (%d)", result -> message_id);

    if (result -> message_type == TIMER_TIME || result -> message_type ==
TIMER_START || result -> message_type == TIMER_STOP) {
        printf(" : %d", result -> worker_id);
    }

    printf("%s", result_type_str);

    if (result -> result_type == NOT_FOUND || result -> result_type ==
ALREADY_EXISTS || result -> result_type == RESULT_UNAVAILABLE) {
        printf("\n");
        return;
    }

    if (result -> result_type == RESULT_PING) {
        PingResult * ping_result = (PingResult * ) result;

        printf(": ");
        while (!queue_empty( & ping_result -> unavailable_nodes)) {
            int * unavailable_node_id;
            queue_pop( & ping_result -> unavailable_nodes, (void ** ) &
unavailable_node_id, NULL);
            printf("%d; ", * unavailable_node_id);
        }
    }

    if (result -> message_type == CREATE) {
        CreateResult * create_result = (CreateResult * ) result;
        printf(" : %d", create_result -> worker_pid);
    }

    if (result -> message_type == TIMER_TIME) {
        TimeResult * time_result = (TimeResult * ) result;
        printf(" : %lu", time_result -> time);
    }

    printf("\n");
}

void handle_create_command(UserCommand cmd) {
    Result add_node_result;
    CreateMessage message;

    if (tree_exists( & tree, cmd.worker_id)) {
        send_already_exists_result(cmd, CREATE);
        return;
    }

    add_node_result = tree_create_node( & tree, cmd.worker_id);

```

```

    message.message_id = cmd.command_id;
    message.receiver_id = add_node_result.parent;
    message.callback_id = mq_id( & callback);
    message.type = CREATE;
    message.created_worker_id = add_node_result.child;

    worker_send_message( & root_worker, (WorkerMessage * ) & message,
sizeof(message));
}

void handle_remove_command(UserCommand cmd) {
    Result remove_node_result;
    DeleteMessage message;

    if (!tree_exists( & tree, cmd.worker_id)) {
        send_not_found_result(cmd, DELETE);
        return;
    }

    tree_remove_node( & tree, cmd.worker_id);

    message.message_id = cmd.command_id;
    message.receiver_id = cmd.worker_id;
    message.callback_id = mq_id( & callback);
    message.type = DELETE,

    worker_send_message( & root_worker, (WorkerMessage * ) & message,
sizeof(message));
}

void handle_exec_start_command(UserCommand cmd) {
    StartTimerMessage message = {
        .message_id = cmd.command_id,
        .receiver_id = cmd.worker_id,
        .callback_id = mq_id( & callback),
        .type = TIMER_START
    };

    if (!tree_exists( & tree, cmd.worker_id)) {
        send_not_found_result(cmd, TIMER_START);
        return;
    }

    worker_send_message( & root_worker, (WorkerMessage * ) & message,
sizeof(message));
}

void handle_exec_stop_command(UserCommand cmd) {
    StopTimerMessage message = {
        .message_id = cmd.command_id,
        .receiver_id = cmd.worker_id,
        .callback_id = mq_id( & callback),

```

```

        .type = TIMER_STOP
    };

    if (!tree_exists( & tree, cmd.worker_id)) {
        send_not_found_result(cmd, TIMER_STOP);
        return;
    }

    worker_send_message( & root_worker, (WorkerMessage * ) & message,
sizeof(message));
}

void handle_exec_time_command(UserCommand cmd) {
    GetTimeMessage message = {
        .message_id = cmd.command_id,
        .receiver_id = cmd.worker_id,
        .callback_id = mq_id( & callback),
        .type = TIMER_TIME
    };

    if (!tree_exists( & tree, cmd.worker_id)) {
        send_not_found_result(cmd, TIMER_TIME);
        return;
    }

    worker_send_message( & root_worker, (WorkerMessage * ) & message,
sizeof(message));
}

void handle_ping_command(UserCommand cmd) {
    Queue nodes, unavailable_nodes;
    MessageQueue local_callback, output;
    PingResult result;

    queue_init( & nodes);
    queue_init( & unavailable_nodes);
    mq_create( & local_callback);
    mq_connect( & output, mq_id( & callback));

    tree_get_nodes( & tree, & nodes);

    while (!queue_empty( & nodes)) {
        int * worker_id;

        queue_pop( & nodes, (void ** ) & worker_id, NULL);

        PingMessage ping_message = {
            .message_id = -1,
            .receiver_id = * worker_id,
            .callback_id = mq_id( & local_callback),
            .type = PING
        };
    };
}

```



```

    WorkerResult * ping_result;
    size_t ping_result_size;

    worker_send_message( & root_worker, (WorkerMessage * ) &
ping_message, sizeof(ping_message));
    mq_rcv( & local_callback, (void ** ) & ping_result, &
ping_result_size);

    if (ping_result -> result_type == RESULT_UNAVAILABLE)
        queue_push( & unavailable_nodes, (void * ) worker_id, sizeof( *
worker_id));
    }

    result.message_id = cmd.command_id;
    result.worker_id = -1;
    result.message_type = PING;
    result.result_type = RESULT_PING;
    memcpy( & result.unavailable_nodes, & unavailable_nodes,
sizeof(unavailable_nodes));

    mq_send( & output, & result, sizeof(result));
}

void handle_close_command(UserCommand cmd) {
    MessageQueue local_callback;

    mq_create( & local_callback);

    DeleteMessage message = {
        .message_id = cmd.command_id,
        .receiver_id = root_worker.id,
        .callback_id = mq_id( & local_callback),
        .type = DELETE
    };

    worker_send_message( & root_worker, (WorkerMessage * ) & message,
sizeof(message));
    waitpid(root_worker.pid, NULL, 0);

    exit(0);
}

int main(void) {
    mq_create( & callback);
    tree_init( & tree);

    worker_create( & root_worker, "worker", -1);
    tree_create_node( & tree, root_worker.id);

    while (1) {
        UserCommand cmd = scan_user_command();

```

```

switch (cmd.type) {
case USER_CREATE:
    handle_create_command(cmd);
    print_tree( & tree);
    break;
case USER_PING:
    handle_ping_command(cmd);
    break;
case USER_REMOVE:
    handle_remove_command(cmd);
    print_tree( & tree);
    break;
case USER_EXEC_START:
    handle_exec_start_command(cmd);
    break;
case USER_EXEC_STOP:
    handle_exec_stop_command(cmd);
    break;
case USER_EXEC_TIME:
    handle_exec_time_command(cmd);
    break;
case USER_CLOSE:
    handle_close_command(cmd);
    break;
}

    handle_result_messages();
}
}

```

## node.c

```

#include "../mq/mq.h"
#include "../worker/worker.h"
#include "../timer/timer.h"
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int id;
MessageQueue mq;
Timer timer;

#define WORKERS_AMOUNT 2
int created_workers = 0;
Worker child_workers[WORKERS_AMOUNT];

void delete_child_worker(int worker_id) {

```

```

int i;
int found = 0;

for (i = 0; i < created_workers; i++) {
    if (child_workers[i].id == worker_id) {

        found = 1;
        break;
    }
}

for (int j = i; j < created_workers - 1; j++) {
    memcpy( & child_workers[j], & child_workers[j + 1],
sizeof(child_workers[j]));
}

if (found) created_workers--;
if (created_workers < 0) created_workers = 0;
}

void init_worker(int worker_id, int callback_id) {
    MessageQueue callback;
    int worker_mq_id;

    mq_create( & mq);
    id = worker_id;

    mq_connect( & callback, callback_id);

    CreateResult result = {
        .worker_id = id,
        .message_type = CREATE,
        .result_type = RESULT_OK,
        .worker_pid = getpid(),
        .worker_mq_id = mq_id( & mq)
    };

    mq_send( & callback, & result, sizeof(result));

    timer_init( & timer);
}

void handle_ping_message(PingMessage * message) {
    MessageQueue callback;
    WorkerResult result = {
        .message_id = message -> message_id,
        .worker_id = id,
        .message_type = message -> type,
        .result_type = RESULT_OK
    };

    mq_connect( & callback, message -> callback_id);
}

```

```

mq_send( & callback, & result, sizeof(result));

mq_close( & callback);
}

void handle_create_message(CreateMessage * message) {
    MessageQueue callback;
    CreateResult result = {
        .message_id = message -> message_id,
        .worker_id = id,
        .message_type = message -> type,
        .result_type = RESULT_OK,
        .worker_pid = -1,
        .worker_mq_id = -1
    };

    mq_connect( & callback, message -> callback_id);

    if (created_workers >= WORKERS_AMOUNT) {
        mq_close( & callback);
        exit(2);
    }

    result.worker_pid = worker_create( & child_workers[created_workers],
"worker", message -> created_worker_id);
    result.worker_mq_id = mq_id( & child_workers[created_workers].mq);
    created_workers++;

    mq_send( & callback, & result, sizeof(result));
    mq_close( & callback);
}

void handle_delete_message>DeleteMessage * message) {
   >DeleteMessage local_message;
   >MessageQueue local_callback, callback;
   >WorkerResult result = {
       >.message_id = message -> message_id,
       >.worker_id = id,
       >.message_type = message -> type,
       >.result_type = RESULT_OK
    };

    mq_create( & local_callback);
    mq_connect( & callback, message -> callback_id);

    memcpy( & local_message, message, sizeof(local_message));
    local_message.callback_id = mq_id( & local_callback);

    for (int i = 0; i < created_workers; i++) {
        local_message.receiver_id = child_workers[i].id;
        mq_send( & child_workers[i].mq, & local_message,
sizeof(local_message));
    }
}

```

```

    }

    mq_send( & callback, & result, sizeof(result));

    mq_close( & local_callback);
    mq_close( & callback);
    exit(0);
}

void handle_timer_start_message(StartTimerMessage * message) {
    MessageQueue callback;
    WorkerResult result = {
        .message_id = message -> message_id,
        .worker_id = id,
        .message_type = message -> type,
        .result_type = RESULT_OK
    };

    mq_connect( & callback, message -> callback_id);

    timer_start( & timer);

    mq_send( & callback, & result, sizeof(result));
    mq_close( & callback);
}

void handle_timer_stop_message(StopTimerMessage * message) {
    MessageQueue callback;
    WorkerResult result = {
        .message_id = message -> message_id,
        .worker_id = id,
        .message_type = message -> type,
        .result_type = RESULT_OK
    };

    mq_connect( & callback, message -> callback_id);

    timer_stop( & timer);

    mq_send( & callback, & result, sizeof(result));
    mq_close( & callback);
}

void handle_timer_time_message(GetTimeMessage * message) {
    MessageQueue callback;
    TimeResult result = {
        .message_id = message -> message_id,
        .worker_id = id,
        .message_type = message -> type,
        .result_type = RESULT_OK,
        .time = 0
    };
};

```

```

mq_connect( & callback, message -> callback_id);

result.time = timer_time( & timer);

mq_send( & callback, & result, sizeof(result));
mq_close( & callback);
}

void delegate_message(WorkerMessage * message, size_t message_size) {
    MessageQueue callback, local_callback;

    mq_connect( & callback, message -> callback_id);
    mq_create( & local_callback);
    mq_set_timeout( & local_callback, 100);

    message -> callback_id = mq_id( & local_callback);

    for (int i = 0; i < created_workers; i++)
        mq_send( & child_workers[i].mq, message, message_size);

    int res = 0;

    for (int i = 0; i < created_workers; i++) {
        WorkerResult * recv_data;
        size_t recv_data_size;

        mq_recv( & local_callback, (void ** ) & recv_data, & recv_data_size);

        if (recv_data == NULL)
            continue;

        if (recv_data -> result_type != RESULT_UNAVAILABLE) {
            mq_send( & callback, recv_data, recv_data_size);
            mq_close( & local_callback);
            free(recv_data);
            res = 1;
            break;
        }

        free(recv_data);
    }

    if (message -> type == DELETE) {
        delete_child_worker(message -> receiver_id);
    }

    if (res) return;

    WorkerResult result = {
        .message_id = message -> message_id,
        .worker_id = message -> receiver_id,
    }
}

```

```

        .message_type = message -> type,
        .result_type = RESULT_UNAVAILABLE
    };

    mq_send( & callback, & result, sizeof(result));
}

int main(int argc, char * argv[]) {
    WorkerMessage * message;
    size_t message_size;

    init_worker(atoi(argv[1]), atoi(argv[2]));

    while (1) {
        mq_rcv( & mq, (void ** ) & message, & message_size);

        if (message -> receiver_id != id) {
            delegate_message(message, message_size);
            free(message);
            continue;
        }

        switch (message -> type) {
        case PING:
            handle_ping_message((PingMessage * ) message);
            break;

        case CREATE:
            handle_create_message((CreateMessage * ) message);
            break;

        case DELETE:
            handle_delete_message((DeleteMessage * ) message);
            break;

        case TIMER_START:
            handle_timer_start_message((StartTimerMessage * ) message);
            break;

        case TIMER_STOP:
            handle_timer_stop_message((StopTimerMessage * ) message);
            break;

        case TIMER_TIME:
            handle_timer_time_message((GetTimeMessage * ) message);
            break;
        }

        free(message);
    }

    return 0;
}

```

```
}
```

## user\_commands.h


```
#ifndef __USER_COMMANDS_H__
#define __USER_COMMANDS_H__

typedef enum {
    USER_CREATE,
    USER_PING,
    USER_REMOVE,
    USER_EXEC_START,
    USER_EXEC_STOP,
    USER_EXEC_TIME,
    USER_CLOSE
} UserCommandType;

typedef struct {
    int command_id;
    UserCommandType type;
    int worker_id;
} UserCommand;

#endif
```

## Пример работы



```
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled$ ./main
(1) > create 1
[Worker #1]
[Worker #1]
[Main] <<< (1) : ok : 84
(2) > create 2
[Worker #2]
[Worker #1]
[Main] <<< (2) : ok : 99
(3) > create 3
[Worker #2]
[Worker #1]
[Worker #1]
[Main] <<< (3) : ok : 122
(4) > exec 3 start
[Main] <<< (4) : 3 : ok
(5) > exec 3 time
[Main] <<< (5) : 3 : ok : 3754
(6) > exec 3 stop
[Main] <<< (6) : 3 : ok
(7) > exec 3 time
[Main] <<< (7) : 3 : ok : 9394
(8) > exec 3 time
[Main] <<< (8) : 3 : ok : 9394
(9) > remove 3
[Worker #2]
[Worker #1]
[Worker #1]
[Main] <<< (9) : ok
(10) > close
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled$
```



```
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled$ ./main
(1) > create 1
[Worker #1]
[Main] <<< (1) : ok : 506
(2) > create 2
[Worker #1]
[Main] <<< (2) : ok : 521
(3) > create 3
[Worker #1]
[Worker #2]
[Main] <<< (3) : ok : 544
(4) > create 4
[Worker #1]
[Worker #3]
[Main] <<< (4) : ok : 565
(5) > ping
[Main] <<< (5) : ok: 1; 3; 4;
(6) > remove 1
[Worker #1]
[Main] <<< (6) : error : node is unavailable
(7) > ping
[Main] <<< (7) : ok:
(8) > close
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled$

alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled$ kill 506
alexg@DESKTOP-9V207HC: /mnt/d/Desktop/OS/os_lab_6/compiled$
```

## 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.