Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

Студент: Гречников А.В.

Группа: М8О–206Б–20

Вариант: 23

Преподаватель: Соколов Андрей Алексеевич

Дата: 12.01.2022

Оценка: 3

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2021

**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

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

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.

Вариант 23.

Топология: Все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.

Набор команд: поиск подстроки в строке

Формат команды:

> exec id

> text\_string

> pattern\_string

[result] – номера позиций, где найден образец, разделенный точкой с запятой

text\_string — текст, в котором искать образец. Алфавит: [A-Za-z0-9]. Максимальная длина строки 10^8 символов

pattern\_string — образец

Тип проверки доступности узлов:

Формат команды: heartbit time

Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от узла нет сигнала в течении 4\*time миллисекунд, то должна выводится пользователю строка:

«Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

Сервер сообщений: ZeroMQ

**Алгоритм**

client.c подключается к корневому узлу дерева и отправляет команды узлам. worker.c выполняет команды. Использован паттерн rep/req.

На первом шаге client.c с помощью fork создает корневой узел и передает ему порт, к которому необходимо подключиться и порты, к которым могут подключаться его правые и левые поддеревья. Корневой узел удалять нельзя. Список всех узлов будет храниться в бинарном дереве. Для создания нового узла необходимо ввести ключ. Если в бинарном дереве нет такого ключа, то тогда этот ключ и порты для его правых и левых поддеревьев передаются корневому узлу. Каждый узел хранит свой ключ при вводе, порты правых и левых поддеревьев и два числа, показывающих, есть ли у узла левый или правый узел. Если у узла есть оба дочерних узла — вспомогательные значения равны 1 —, то передаваемый ключ сравнивается с ключом узла — если меньше, то передается в левое поддерево, иначе — в правое. Если у узла нет

левого дочернего узла и передаваемый ключ меньше ключа узла, то с помощью fork создается новый узел, ему передаются ключ и порты, вспомогательное значение на родительском узле увеличивается, аналогично с правым дочерним узлом. Если узел имеет только одно поддерево и ему нужно передать этот ключ, то туда отправляется ключ.

При удалении узла сначала проверяется, есть ли такой узел в бинарном дереве — если нет или ключ равен корневому узлу, то узел не удаляется. Иначе находим все левые поддеревья узла, и их удаляем, затем — все правые поддеревья и их удаляем, затем удаляем сам узел. Удаление узла: двигаемся по дереву и находим удаляемый узел. Как только его нашли, то отправляем сообщение его родителю, что мы его собираемся удалить. Родитель узла обнуляет вспомогательное значение и дожидается с помощью wait, когда узел завершит работу. Узел выходит из цикла и завершает работу.

Для проверки доступности узла сначала вводится hearbit, в течение которого узел должен послать ему ответ. Далее вводится ping и ключ узла, после после запускается таймер, и посылается сигнал узлу.

Для выполнения действия на узле необходимо ввести exec, строку, в которой необходимо найти образец, образец и ключ узла. Далее строки отправляются на узел, который ищет подстроку в строке: проходим строку и одновременно сравниваем ее с образцом, и если в итоге достигнут конец образца, то вхождение найдено, и выводится позиция, с которой начиналось сравнение.

Если образец не найден, то выводится -1.

При завершении работы вводится exit, после чего удаляются все узлы.

**Основные файлы программы**

client.c:

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#include "zmq.h"

#include <stdlib.h>

#include "ball\_tree.h"

#include <sys/wait.h>

#include <time.h>

typedef struct MD

{

int port2, port3;

char message[128];

char text\_string[128];

char pattern\_string[128];

int key;

} MessageMD;

void function1(void \*respond, int key, int port2, int port3, char \*command)

{

MessageMD md;

md.key = key;

md.port2 = port2;

md.port3 = port3;

strcpy(md.message, command);

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md, sizeof(MessageMD));

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

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

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

zmq\_msg\_close(&reply);

}

void function2(void \*respond, int key, char \*command)

{

int n = 0;

MessageMD md;

md.key = key;

strcpy(md.message, command);

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md, sizeof(MessageMD));

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

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

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

if (!strcmp((char \*)zmq\_msg\_data(&reply), "remove\0"))

n = 1;

zmq\_msg\_close(&reply);

zmq\_msg\_t req2;

zmq\_msg\_init\_size(&req2, strlen("..\0"));

memcpy(zmq\_msg\_data(&req2), "..\0", strlen("..\0"));

zmq\_msg\_send(&req2, respond, 0);

zmq\_msg\_close(&req2);

zmq\_msg\_t req3;

zmq\_msg\_init(&req3);

zmq\_msg\_recv(&req3, respond, 0);

zmq\_msg\_close(&req3);

if (n)

wait(0);

}

void function3(void \*respond, int key, char \*command, char \*text, char \*pattern)

{

MessageMD md;

md.key = key;

strcpy(md.message, command);

strcpy(md.text\_string, text);

strcpy(md.pattern\_string, pattern);

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md, sizeof(MessageMD));

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

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

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

zmq\_msg\_close(&reply);

}

void function4(void \*respond, int key, double time, char \*command)

{

MessageMD md;

md.key = key;

strcpy(md.message, "ping\0");

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md, sizeof(MessageMD));

clock\_t start = clock();

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

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

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

start = clock() - start;

zmq\_msg\_close(&reply);

if ((double)start / (CLOCKS\_PER\_SEC) < time \* 4)

printf("Time: %f %f\n", time, (double)start / (CLOCKS\_PER\_SEC));

else

printf("No %f %f\n", time, (double)start / (CLOCKS\_PER\_SEC));

}

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

{

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

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

int key, r\_key, port2 = 5051, port3 = 7071;

double time = 0;

char command[10], text\_string[128], pattern\_string[128];

printf("Input root key: ");

scanf("%d", &key);

zmq\_connect(respond, "tcp://localhost:4040");

if (!fork())

execl("./worker", "./worker", "4040", "5050", "7070", NULL);

MessageMD md1;

md1.key = key;

md1.port2 = port2;

md1.port3 = port3;

strcpy(md1.message, "start\0");

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md1, sizeof(MessageMD));

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

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

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

zmq\_msg\_close(&reply);

Node \*tree = create(key);

r\_key = key;

for (;;)

{

printf("input command ");

scanf("%s", command);

if (!strcmp(command, "exit"))

{

int k2;

while ((k2 = find\_key(tree->left)) != 0)

{

function2(respond, k2, "remove\0");

tree = delete\_node(tree, k2);

}

while ((k2 = find\_key(tree->right)) != 0)

{

function2(respond, k2, "remove\0");

tree = delete\_node(tree, k2);

}

function2(respond, r\_key, "remove\0");

break;

}

else if (!strcmp("remove", command))

{

printf("input key: ");

scanf("%d", &key);

Node \*k = search(tree, key);

if (!k || r\_key == key)

printf("can't remove\n");

else

{

int k2;

while ((k2 = find\_key(k->left)) != 0)

{

function2(respond, k2, "remove\0");

tree = delete\_node(tree, k2);

k = delete\_node(k, k2);

}

while ((k2 = find\_key(k->right)) != 0)

{

function2(respond, k2, "remove\0");

tree = delete\_node(tree, k2);

k = delete\_node(k, k2);

}

function2(respond, key, "remove\0");

tree = delete\_node(tree, key);

}

}

else if (!strcmp("ping", command))

{

printf("input key: ");

scanf("%d", &key);

if (!search(tree, key))

printf("Not exist\n");

else

function4(respond, key, time, "ping\0");

}

else if (!strcmp("create", command))

{

printf("input key: ");

scanf("%d", &key);

if (!search(tree, key))

{

port2++;

port3++;

function1(respond, key, port2, port3, "create\0");

tree = insert(tree, key);

}

else

printf("Node exists\n");

}

else if (!strcmp("heartbit", command))

{

printf("input time sec: ");

scanf("%lf", &time);

}

else if (!strcmp("exec", command))

{

printf("input key: ");

scanf("%d", &key);

if (search(tree, key))

{

printf("input text string: ");

scanf("%s", text\_string);

printf("input pattern string: ");

scanf("%s", pattern\_string);

function3(respond, key, "exec\0", text\_string, pattern\_string);

}

else

printf("Not exist\n");

}

else if (!strcmp("print", command))

print(tree);

sleep(1);

}

zmq\_close(respond);

zmq\_ctx\_destroy(context);

return 0;

}

worker.c:

#define \_GNU\_SOURCE

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#include "zmq.h"

#include <stdlib.h>

#include <sys/wait.h>

typedef struct MD

{

int port2, port3;

char message[128];

char text\_string[128];

char pattern\_string[128];

int key;

} MessageMD;

typedef struct node

{

int key, port2, port3, n2, n3;

} Node;

void find\_pattern(char \*string, char \*pattern)

{

int count = 0;

int i, j, k;

printf("%d Found:", getpid());

for (i = 0; i < strlen(string); i++)

{

for (j = i, k = 0; k < strlen(pattern) && string[j] == pattern[k]; j++, k++);

if (k == strlen(pattern))

{

count++;

printf(" %d", i);

}

}

if (!count)

printf("-1\n");

else

printf("\n");

}

void function(void \*worker2, void \*worker3, char \*msg, int key, int global\_key, int port2, int port3)

{

MessageMD md1;

md1.key = key;

strcpy(md1.message, msg);

md1.port2 = port2;

md1.port3 = port3;

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md1, sizeof(MessageMD));

if (key < global\_key)

zmq\_msg\_send(&req, worker2, 0);

else

zmq\_msg\_send(&req, worker3, 0);

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

if (key < global\_key)

zmq\_msg\_recv(&reply, worker2, 0);

else

zmq\_msg\_recv(&reply, worker3, 0);

zmq\_msg\_close(&reply);

}

void function2(void \*worker, char \*msg, int key, int \*n)

{

MessageMD md1;

md1.key = key;

strcpy(md1.message, msg);

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md1, sizeof(MessageMD));

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

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

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

if (!strcmp((char \*)zmq\_msg\_data(&reply), "remove\0"))

(\*n) = 0;

zmq\_msg\_close(&reply);

zmq\_msg\_t req2;

zmq\_msg\_init\_size(&req2, strlen("..\0"));

memcpy(zmq\_msg\_data(&req2), "..\0", strlen("..\0"));

zmq\_msg\_send(&req2, worker, 0);

zmq\_msg\_close(&req2);

zmq\_msg\_t req3;

zmq\_msg\_init(&req3);

zmq\_msg\_recv(&req3, worker, 0);

zmq\_msg\_close(&req3);

if (!(\*n))

wait(0);

}

void sendd(void \*worker, char \*msg, int n)

{

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, n);

memcpy(zmq\_msg\_data(&req), msg, n);

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

zmq\_msg\_close(&req);

zmq\_msg\_t req2;

zmq\_msg\_init(&req2);

zmq\_msg\_recv(&req2, worker, 0);

zmq\_msg\_close(&req2);

zmq\_msg\_t reply;

zmq\_msg\_init\_size(&reply, strlen("world"));

memcpy(zmq\_msg\_data(&reply), "world", 5);

zmq\_msg\_send(&reply, worker, 0);

zmq\_msg\_close(&reply);

}

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

{

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

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

void \*worker2 = zmq\_socket(context, ZMQ\_REQ);

void \*worker3 = zmq\_socket(context, ZMQ\_REQ);

char \*adress, \*adress\_, \*adress\_\_;

int key, port2, port3, n2 = 0, n3 = 0;

port2 = atoi(argv[2]);

port3 = atoi(argv[3]);

asprintf(&adress, "tcp://\*:%s", argv[1]);

zmq\_bind(worker, adress);

zmq\_msg\_t request;

zmq\_msg\_init(&request);

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

MessageMD \*m = (MessageMD \*)zmq\_msg\_data(&request);

printf("\nNode Pid:%d K:%d created\n", getpid(), m->key);

fflush(stdout);

zmq\_msg\_close(&request);

key = m->key;

asprintf(&adress\_, "tcp://localhost:%s", argv[2]);

asprintf(&adress\_\_, "tcp://localhost:%s", argv[3]);

zmq\_connect(worker2, adress\_);

zmq\_connect(worker3, adress\_\_);

zmq\_msg\_t reply;

zmq\_msg\_init\_size(&reply, strlen("world"));

memcpy(zmq\_msg\_data(&reply), "world", 5);

zmq\_msg\_send(&reply, worker, 0);

zmq\_msg\_close(&reply);

for (;;)

{

zmq\_msg\_t request;

zmq\_msg\_init(&request);

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

MessageMD \*md = (MessageMD \*)zmq\_msg\_data(&request);

char \*msg = " ";

zmq\_msg\_close(&request);

if (!strcmp(md->message, "create\0"))

{

if (n2 == 0 && md->key < key)

{

n2 = 1;

if (fork() == 0)

{

sprintf(adress, "%d", port2);

sprintf(adress\_, "%d", md->port2);

sprintf(adress\_\_, "%d", md->port3);

execl("./worker", "./worker", adress, adress\_, adress\_\_, NULL);

}

else

function(worker2, worker3, "create\0", md->key, key, md->port2, md->port3);

}

else if (n3 == 0 && md->key > key)

{

n3 = 1;

if (fork() == 0)

{

sprintf(adress, "%d", port3);

sprintf(adress\_, "%d", md->port2);

sprintf(adress\_\_, "%d", md->port3);

execl("./worker", "./worker", adress, adress\_, adress\_\_, NULL);

}

else

function(worker2, worker3, "create\0", md->key, key, md->port2, md->port3);

}

else

function(worker2, worker3, "create\0", md->key, key, md->port2, md->port3);

}

else if (!strcmp(md->message, "remove\0"))

{

if (md->key == key)

{

sendd(worker, "remove\0", strlen("remove\0") + 1);

break;

}

if (md->key < key)

{

sendd(worker, "hello\0", strlen("hello\0") + 1);

function2(worker2, "remove\0", md->key, &n2);

}

else if (md->key > key)

{

sendd(worker, "hello\0", strlen("hello\0") + 1);

function2(worker3, "remove\0", md->key, &n3);

}

}

else if (!strcmp(md->message, "ping\0"))

{

if (md->key == key)

{

zmq\_msg\_t reply;

zmq\_msg\_init\_size(&reply, strlen("ok\0"));

memcpy(zmq\_msg\_data(&reply), "ok\0", strlen("ok\0"));

zmq\_msg\_send(&reply, worker, 0);

zmq\_msg\_close(&reply);

continue;

}

else

{

MessageMD md1;

md1.key = md->key;

strcpy(md1.message, "ping\0");

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md1, sizeof(MessageMD));

if (md->key < key)

zmq\_msg\_send(&req, worker2, 0);

else

zmq\_msg\_send(&req, worker3, 0);

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

if (md->key < key)

zmq\_msg\_recv(&reply, worker2, 0);

else

zmq\_msg\_recv(&reply, worker3, 0);

msg = (char \*)zmq\_msg\_data(&reply);

zmq\_msg\_close(&reply);

}

}

else if (!strcmp(md->message, "exec\0"))

{

if (md->key == key)

find\_pattern(md->text\_string, md->pattern\_string);

else

{

MessageMD md1;

md1.key = md->key;

strcpy(md1.message, "exec\0");

strcpy(md1.text\_string, md->text\_string);

strcpy(md1.pattern\_string, md->pattern\_string);

zmq\_msg\_t req;

zmq\_msg\_init\_size(&req, sizeof(MessageMD));

memcpy(zmq\_msg\_data(&req), &md1, sizeof(MessageMD));

if (md->key < key)

zmq\_msg\_send(&req, worker2, 0);

else

zmq\_msg\_send(&req, worker3, 0);

zmq\_msg\_close(&req);

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

if (md->key < key)

zmq\_msg\_recv(&reply, worker2, 0);

else

zmq\_msg\_recv(&reply, worker3, 0);

msg = (char \*)zmq\_msg\_data(&reply);

zmq\_msg\_close(&reply);

}

}

zmq\_msg\_t reply;

zmq\_msg\_init\_size(&reply, strlen(msg));

memcpy(zmq\_msg\_data(&reply), msg, strlen(msg));

zmq\_msg\_send(&reply, worker, 0);

zmq\_msg\_close(&reply);

}

printf("K:%d Pid:%d finished work\n", key, getpid());

zmq\_close(worker2);

zmq\_close(worker3);

zmq\_close(worker);

zmq\_ctx\_destroy(context);

return 0;

}

ball\_tree.h:

#ifndef \_BALL\_TREE\_H\_

#define \_BALL\_TREE\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct node

{

int key;

struct node \*left;

struct node \*right;

unsigned char height;

};

typedef struct node Node;

Node \*create(int key)

{

Node \*p = (Node \*)malloc(sizeof(Node));

p->key = key;

p->left = p->right = 0;

p->height = 1;

return p;

}

Node \*insert(Node \*tree, int val)

{

if (!tree)

return create(val);

if (val < tree->key)

tree->left = insert(tree->left, val);

else

tree->right = insert(tree->right, val);

return (tree);

}

Node \*min\_value(Node \*tree)

{

if (tree->left)

return min\_value(tree->left);

else

return (tree);

}

Node \*delete\_node(Node \*tree, int val)

{

if (!tree)

return tree;

else if (val < tree->key)

tree->left = delete\_node(tree->left, val);

else if (val > tree->key)

tree->right = delete\_node(tree->right, val);

else

{

Node \*tmp = tree;

if (tmp->left && tmp->right)

{

tmp = min\_value(tmp->right);

tree->key = tmp->key;

tree->right = delete\_node(tree->right, tree->key);

}

else if (!(tmp->right))

tree = tmp->left;

else if (!(tmp->left))

tree = tmp->right;

free(tmp);

}

return (tree);

}

void delete\_tree(Node \*tree)

{

if (tree)

{

delete\_tree(tree->left);

delete\_tree(tree->right);

free(tree);

}

}

int find\_key(Node \*p)

{

if (!p)

return 0;

if (!p->left && !p->right)

return p->key;

else if (p->right)

return find\_key(p->right);

else

return find\_key(p->left);

}

Node \*search(Node \*p, int key)

{

if (!p)

return NULL;

if (p->key > key)

return search(p->left, key);

else if (p->key < key)

return search(p->right, key);

return p;

}

void print(Node \*p)

{

if (p)

{

print(p->left);

printf("%d ", p->key);

print(p->right);

}

}

#endif

Makefile:

all\_done: worker client

worker: worker.c

gcc worker.c -lzmq -o worker

client: client.c ball\_tree.h

gcc client.c ball\_tree.h -lzmq -o client

**Пример работы**

artem@ideapad:~$ ./client

Input root key: 50

Node Pid:27781 K:50 created

input command create

input key: 11

Node Pid:27786 K:11 created

input command create

input key: 55

Node Pid:27789 K:55 created

input command create

input key: 12

Node Pid:27797 K:12 created

input command create

input key: 9

Node Pid:27800 K:9 created

input command create

input key: 51

Node Pid:27805 K:51 created

input command remove

input key: 11

K:9 Pid:27800 finished work

K:12 Pid:27797 finished work

K:11 Pid:27786 finished work

input command heartbit

input time sec: 2

input command ping

input key: 51

Time: 2.000000 0.000219

input command heartbit

input time sec: 0

input command ping

input key: 51

No 0.000000 0.000209

input command exec

input key: 51

input text string: wasdqwadfawawa

input pattern string: wa

27805 Found: 0 5 10 12

input command exit

K:51 Pid:27805 finished work

K:55 Pid:27789 finished work

K:50 Pid:27781 finished work

**Вывод**

В ходе лабораторной работы я реализовал распределенную систему по асинхронной обработке запросов. Для связи узлов использованы сокет rep/req: отправка и прием сообщений являются синхронными, поэтому необходимо отслеживать, существуют ли у узла дочерние узлы (с помощью специально введенных чисел), и последовательность отправки/получения сообщений: если последовательность не будет соблюдена, то будет получена ошибка: Req будет заблокирован при отправке, если он не получит ответ успешно, Rep будет заблокирован при получении, если он не получил запроса. То есть запрос/ответ является парным и должен быть успешным. Порт, к которому необходимо подключиться дочернему узлу, и порты, к которым будут подключаться его потомки, передаются при создании узла, номера самих портов генерируются в client.