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: Ткаченко Егор Юрьевич
Группа: М8О-207Б-21
Вариант: 10
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/Tnirpps/OS_lab

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 два 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;

• Удаление существующего вычислительного узла;

• Исполнение команды на вычислительном узле;

• Проверка доступности вычислительного узла.

Задание варианта

Вариант 10.

Все вычислительные узлы находятся в списке. Есть только один управляющий узел.

Исполнение команды — поиск подстроки в строке.

Команда проверки — проверка доступности всех узлов.

Общие сведения о программе

Программа распределительного узла компилируется из файла main.c, программа вычислительного узла компилируется из файла node.c. В программе используется библиотека для работы с сервером сообщений ZeroMQ. В программе используются следующие системные вызовы:

fork — создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.
execl — используется для выполнения другой программы. Эта другая программа, называемая процессом-потомком (child process), загружается поверх программы, содержащей вызов exec. Имя файла, содержащего процесс-потомок, задано с помощью первого аргумента. Какие-либо аргументы, передаваемые процессу-потомку, задаются либо с помощью параметров от arg0 до argN, либо с помощью массива arg[].
Также были использованы следующие вызовы из библиотеки ZMQ:
zmq_ctx_new — создает новый контекст ZMQ.
zmq_connect — создает входящее соединение на сокет.
zmq_disconnect — отсоединяет сокет от заданного endpoint’a.
zmq_socket — создает ZMQ сокет.
zmq_close — закрывает ZMQ сокет.
zmq_ctx_destroy — уничтожает контекст ZMQ.

Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZMQ.

2. Проработать принцип общения между клиентскими узлами и между первым клиентом и сервером и алгоритм выполнения команд клиентами.

3. Реализовать необходимые функции-обертки над вызовами функций библиотеки ZMQ.

4. Написать программу сервера и клиента

Исходный код

================================ main.c ================================

```
#include <stdio.h>
#include "../headers/msgQ.h"

int* NODES;
```

```
int split_copy(const char* text, char* dest, int index) {
    int i = 0;
    for (int j = 0; j < index; ++j) {
        // skip word in text
        while (text[i] != ' ' && text[i] != '\0' && text[i] != '\n') {
            ++i;
        }
        if (text[i] == ' ') {
            ++i;
        }
    }
    int k = i;
    while (text[i] != ' ' && text[i] != '\0' && text[i] != '\n') {
        dest[i - k] = text[i];
        ++i;
    }
    dest[i - k] = '\0';
    return i - k;
}

int __str_to_int(const char* str) {
    int res = 0;
    int protect = 0;
    for (int i = 0; str[i] > 0; ++i) {
        if (str[i] >= '0' && str[i] <= '9') {
            if (protect++ > 4) return -100000;
            res = res * 10 + str[i] - '0';
        }
    }
    return res;
}

int node_exist(int value) {
    if (NODES == NULL) return 0;
    int i = 0;
    while (NODES[i] != TERMINATOR) {
        if (NODES[i] == value) return 1;
```

```
        ++i;
    }
    return 0;
}

void stop_all_children(void* requester, char* addr, message token) {
    for (int i = 0; NODES[i] != TERMINATOR; ++i) {
        token.cmd = delete;
        if (ping_process(NODES[i])) {
            reconnect_zmq_socket(requester, NODES[i], addr);
            send_msg_wait(requester, &token);
            receive_msg_wait(requester, &token);
        }
    }
}

int main (int argc, char const *argv[])  {
    void *context = zmq_ctx_new();
    void *requester = create_zmq_socket(context, ZMQ_REQ);
    NODES = create_vector();
    if (NODES == NULL) return 1;
    char query_line [MN];
    char query_word[MN];
    char query_int[MN];
    char addr[MN] = SERVER_SOCKET_PATTERN;
    int last_created = -1;
    printf("ME: %d\n", getpid());
    message token = {create, 0, "", ""};

    while (fgets(query_line, MN, stdin) != NULL) {
        split_copy(query_line, query_word, 0);

        if (strcmp(query_word, "create") == 0) {
            if (split_copy(query_line, query_int, 1) == 0) {
                printf("\tbad id, try another one\n");
                continue;
            }
            int id_process = __str_to_int(query_int) + MIN_ADDR;
```

```
if (node_exist(id_process)) {
    printf("\tthis node was created earlier\n");
    continue;
}
char flag_creation_str[MN];
int flag_creation = 0;
int l = split_copy(query_line, flag_creation_str, 2);
if (l == 2) {
    if (flag_creation_str[0] == '-' && flag_creation_str[1] == '1') {
        flag_creation = 1;
    }
}
if (length(NODES) == 0 || flag_creation) {
    if (id_process > MIN_ADDR) {
        printf("\tI'm creating %d\n", id_process);
        NODES = push_back(NODES, id_process);
        if (NODES == NULL) {
            printf("\tSm-th wrong with dynamic array\n");
            return 1;
        }
        reconnect_zmq_socket(requester, id_process, addr);
    } else {
        printf("\tSm-th wrong with number\n");
        continue;
    }
    memset(query_int, 0, MN);
    sprintf(query_int, "%d", id_process);
    char *Child_argv[] = {"node", query_int, NULL};
    int pid = fork();
    if (pid == -1) {
        return 1;
    }
    if (pid == 0) {
        execv("node", Child_argv);
        return 0;
    }
    last_created = id_process;
} else {
```

```
            if (id_process < MIN_ADDR) {
                printf("\tbad id\n");
                continue;
            }
            if (ping_process(last_created) == false) {
                printf("\tCannot create Node from [%d]\n", last_created - MIN_ADDR);
                continue;
            }
            token.cmd = create;
            token.value = id_process;
            create_addr(addr, last_created);
            printf("\tsend to %s\n", addr);
            send_msg_wait(requester, &token);
            receive_msg_wait(requester, &token);
            if (token.cmd == success) {
                NODES = push_back(NODES, id_process);
                last_created = id_process;
                if (NODES == NULL) {
                    printf("\tSm-th wrong with dynamic array\n");
                    return 1;
                }
            } else {
                printf("\t%d was successfully created\n", id_process - MIN_ADDR);
            }
        }

    } else if (strcmp(query_word, "exec") == 0) {
        split_copy(query_line, query_int, 1);
        split_copy(query_line, query_int, 1);
        int id_process = __str_to_int(query_int) + MIN_ADDR;
        if (node_exist(id_process) && ping_process(id_process)) {
            fgets(token.str, MAX_LEN, stdin);
            fgets(token.sub, MAX_LEN, stdin);
            for (int i = MAX_LEN - 1; i >= 0; --i) {
                if (token.str[i] == '\n') {
                    token.str[i] = '\0';
                    break;
                }
```

```
            }
            token.cmd = exec;
            reconnect_zmq_socket(requester, id_process, addr);
            send_msg_wait(requester, &token);
            receive_msg_wait(requester, &token);
            if (token.cmd == success) {
                printf("\tall matches: ");
                for (int i = 0; i < MAX_LEN; ++i) {
                    if (token.res[i] == -1) {
                        if (i == 0) printf("NO");
                        break;
                    }
                    printf("%d ", token.res[i]);
                }
                printf("\n");
            } else {
                printf("\tcannot exec %d\n", id_process - MIN_ADDR);
            }
        } else {
            printf("\t[%d] Node hasn't connection\n", id_process - MIN_ADDR);
        }

    } else if (strcmp(query_word, "remove") == 0) {
        split_copy(query_line, query_int, 1);
        int id_process = __str_to_int(query_int) + MIN_ADDR;
        if (node_exist(id_process) && ping_process(id_process)) {
            reconnect_zmq_socket(requester, id_process, addr);
            token.cmd = delete;
            send_msg_wait(requester, &token);
            receive_msg_wait(requester, &token);
            if (token.cmd != success) {
                printf("\tcannot remove %d\n", id_process - MIN_ADDR);
            }
        } else {
            printf("\t[%d] Node hasn't connection\n", id_process - MIN_ADDR);
        }

    } else if (strcmp(query_word, "pingall") == 0) {
```

```
            printf("\t");
            for (int i = 0; NODES[i] != TERMINATOR; ++i) {
                if (ping_process(NODES[i]) == true) {
                    printf("[%d is OKE] ", NODES[i] - MIN_ADDR);
                } else {
                    printf("[%d is BAD] ", NODES[i] - MIN_ADDR);
                }
            }
            printf("\n");
        } else {
            printf("\tBad command. Please, try again\n");
        }
        memset(query_line, 0, MN);
        memset(query_word, 0, MN);
        memset(query_int, 0, MN);
    }
    stop_all_children(requester, addr, token);
    close_zmq_socket(requester);
    destroy_zmq_context(context);
    destroy(NODES);
    return 0;
}
```

============================== client.c ==================================

```
#include <stdio.h>
#include <string.h>

#include "../headers/msgQ.h"

int* z_function(const char* s) {
    int n = (int) strlen(s);
    int* z = malloc(n * sizeof(int));
    if (z == NULL) {
        return NULL;
    }
    int l = 0, r = 0;
    z[0] = 0;
    for (int i = 1; i < n; ++i) {
        if (i <= r) {
```

```
//          here is z[i] = min(z[i - l], r - i) :
            z[i] = (z[i - l] < r - i) ? z[i - l] : r - i;
        }
        while (i + z[i] < n && s[i + z[i]] == s[z[i]]) {
            ++z[i];
        }
        if (i + z[i] > r) {
            l = i;
            r = i + z[i];
        }
    }
    return z;
}



void find_substr(message* token) {
    /*
        z_function(<pattern> + '#' + <text>)
        here we use '\n' as '#'
     */
    int n = (int) strlen(token->str);
    int m = (int) strlen(token->sub);
    char string [n + m + 1];
    memcpy(string, token->sub, m);
    memcpy(string + m, token->str, n);
    string[n + m] = '\0';
    int* z = z_function(string);
    clear_token(token);
    if (z == NULL) {
        return;
    }
    int k = 0;
    for (int i = 0; i < n; i++) {
        if (z[m + i] == m - 1) {
            // one indexing
            token->res[k++] = i + 1;
        }
    }
```

```
    token->cmd = success;
    free(z);
}

int main(int argc, char const *argv[])  {
    if (argc < 2) {
        printf("argv error\n");
        return 1;
    }
    void *context = create_zmq_context();
    void *responder = create_zmq_socket(context, ZMQ_REP);
    char adr[30] = TCP_SOCKET_PATTERN;
    strcat(adr, argv[1]);
    printf("\t[%d] has been created\n", getpid());
    bind_zmq_socket(responder, adr);

    while (1) {
        message token;
        receive_msg_wait(responder, &token);
        int id_process;
        char query_int[30];
        switch (token.cmd) {
            case delete:
                token.cmd = success;
                printf("\t[%d] has been destroyed\n", getpid());
                send_msg_wait(responder, &token);
                close_zmq_socket(responder);
                destroy_zmq_context(context);
                return 0;
            case create:
                id_process = token.value;
                memset(query_int, 0, 30);
                sprintf(query_int, "%d", id_process);
                char *Child_argv[] = {"node", query_int, NULL};
                int pid = fork();
                if (pid == -1) {
                    return 1;
                }
```

```
            if (pid == 0) {
                execv("node", Child_argv);
                return 0;
            } else {
                token.cmd = success;
                send_msg_wait(responder, &token);
            }
            break;
        case exec:
            find_substr(&token);
            send_msg_wait(responder, &token);
            break;
        default:
            // equals "delete brunch"
            token.cmd = success;
            send_msg_wait(responder, &token);
            close_zmq_socket(responder);
            destroy_zmq_context(context);
            return 0;
      }
   }
}
============================== msgQ.c ==============================
#include "../headers/msgQ.h"

void create_addr(char* addr, int id) {
   char str[MN];
   memset(str, 0, MN);
   sprintf(str, "%d", id);
   memset(addr, 0, MN);
   memcpy(addr, SERVER_SOCKET_PATTERN, sizeof(SERVER_SOCKET_PATTERN));
   strcat(addr, str);
}

void clear_token(message* msg) {
   msg->cmd = delete;
   msg->value = 0;
   memset(msg->str, 0, MAX_LEN);
```

```
    memset(msg->sub, 0, MAX_LEN);
    for (int i = 0; i < MAX_LEN; ++i) {
        msg->res[i] = -1;
    }
}

void* create_zmq_context() {
    void* context = zmq_ctx_new();
    if (context == NULL) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_ctx_new ");
        exit(ERR_ZMQ_CTX);
    }
    return context;
}

void bind_zmq_socket(void* socket, char* endpoint) {
    if (zmq_bind(socket, endpoint) != 0) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_bind ");
        exit(ERR_ZMQ_BIND);
    }
}

void disconnect_zmq_socket(void* socket, char* endpoint) {
    if (zmq_disconnect(socket, endpoint) != 0) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_disconnect ");
        exit(ERR_ZMQ_DISCONNECT);
    }
}

void connect_zmq_socket(void* socket, char* endpoint) {
    if (zmq_connect(socket, endpoint) != 0) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_connect ");
        exit(ERR_ZMQ_CONNECT);
    }
```

```
}


void* create_zmq_socket(void* context, const int type) {
    void* socket = zmq_socket(context, type);
    if (socket == NULL) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_socket ");
        exit(ERR_ZMQ_SOCKET);
    }
    return socket;
}

void reconnect_zmq_socket(void* socket, int to, char* addr) {
    if (addr[16] != '\0') {
        disconnect_zmq_socket(socket, addr);
    }
    create_addr(addr, to);
    connect_zmq_socket(socket, addr);
}

void close_zmq_socket(void* socket) {
    if (zmq_close(socket) != 0) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_close ");
        exit(ERR_ZMQ_CLOSE);
    }
}

void destroy_zmq_context(void* context) {
    if (zmq_ctx_destroy(context) != 0) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_ctx_destroy ");
        exit(ERR_ZMQ_CLOSE);
    }
}
```

```
int* create_vector() {
    int* ptr = malloc(sizeof(int));
    if (ptr == NULL) return NULL;
    ptr[0] = TERMINATOR;
    return ptr;
}

int length(const int* ptr) {
    if (ptr == NULL) -1;
    int i = 0;
    while (ptr[i] != TERMINATOR) {
        ++i;
    }
    return i;
}



// very bad, but essy solution: not to use capacity
int* push_back(int* ptr, int value) {
    if (ptr == NULL) return NULL;
    int size = 0;
    while (ptr[size] != TERMINATOR) {
        ++size;
    }
    int* tmp = realloc(ptr, (size + 2) * sizeof(int));
    if (tmp == NULL) {
        free(ptr);
        return NULL;
    }
    ptr = tmp;
    ptr[size] = value;
    ptr[size + 1] = TERMINATOR;
    return ptr;
}

void erase(int* ptr, int value) {
    if (ptr == NULL) return;
    int i = 0;
```

```
    while (ptr[i] != TERMINATOR) {
        if (ptr[i] == value) break;
        ++i;
    }
    // shift array after deleting the element
    while (ptr[i] != TERMINATOR) {
        ptr[i] = ptr[i + 1];
        i++;
    }
}

void destroy(int* ptr) {
    if (ptr == NULL) return;
    free(ptr);
}


bool receive_msg_wait(void* socket, message* token) {
    zmq_msg_t reply;
    zmq_msg_init(&reply);
    if (zmq_msg_recv(&reply, socket, 0) == -1) {
        zmq_msg_close(&reply);
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_msg_recv ");
        exit(ERR_ZMQ_MSG);
    }
    (*token) = * ((message*) zmq_msg_data(&reply));
    if (zmq_msg_close(&reply) == -1) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_msg_close ");
        exit(ERR_ZMQ_MSG);
    }
    return true;
}

bool send_msg_wait(void* socket, message* token) {
    zmq_msg_t msg;
    zmq_msg_init(&msg);
```

```
    if (zmq_msg_init_size(&msg, sizeof(message)) == -1) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_msg_init ");
        exit(ERR_ZMQ_MSG);
    }
    if (zmq_msg_init_data(&msg, token, sizeof(message), NULL, NULL) == -1) {
        fprintf(stderr, "[%d] ", getpid());
        perror("ERROR zmq_msg_init ");
        exit(ERR_ZMQ_MSG);
    }
    if (zmq_msg_send(&msg, socket, 0) == -1) {
        zmq_msg_close(&msg);
        return false;
    }
    return true;
}

bool ping_process(int id) {
    char addr_monitor[30];
    char addr_connection[30];
    char str[30];
    memset(str, 0, 30);
    sprintf(str, "%d", id);
    memset(addr_monitor, 0, 30);
    memcpy(addr_monitor, PING_SOCKET_PATTERN, sizeof(PING_SOCKET_PATTERN));
    strcat(addr_monitor, str);
    memset(addr_connection, 0, 30);
    memcpy(addr_connection, SERVER_SOCKET_PATTERN,
sizeof(SERVER_SOCKET_PATTERN));
    strcat(addr_connection, str);

    void* context = zmq_ctx_new();
    void *requester = zmq_socket(context, ZMQ_REQ);

    zmq_socket_monitor(requester, addr_monitor, ZMQ_EVENT_CONNECTED |
ZMQ_EVENT_CONNECT_RETRIED);
    void *socet = zmq_socket(context, ZMQ_PAIR);
    zmq_connect(socet, addr_monitor);
```

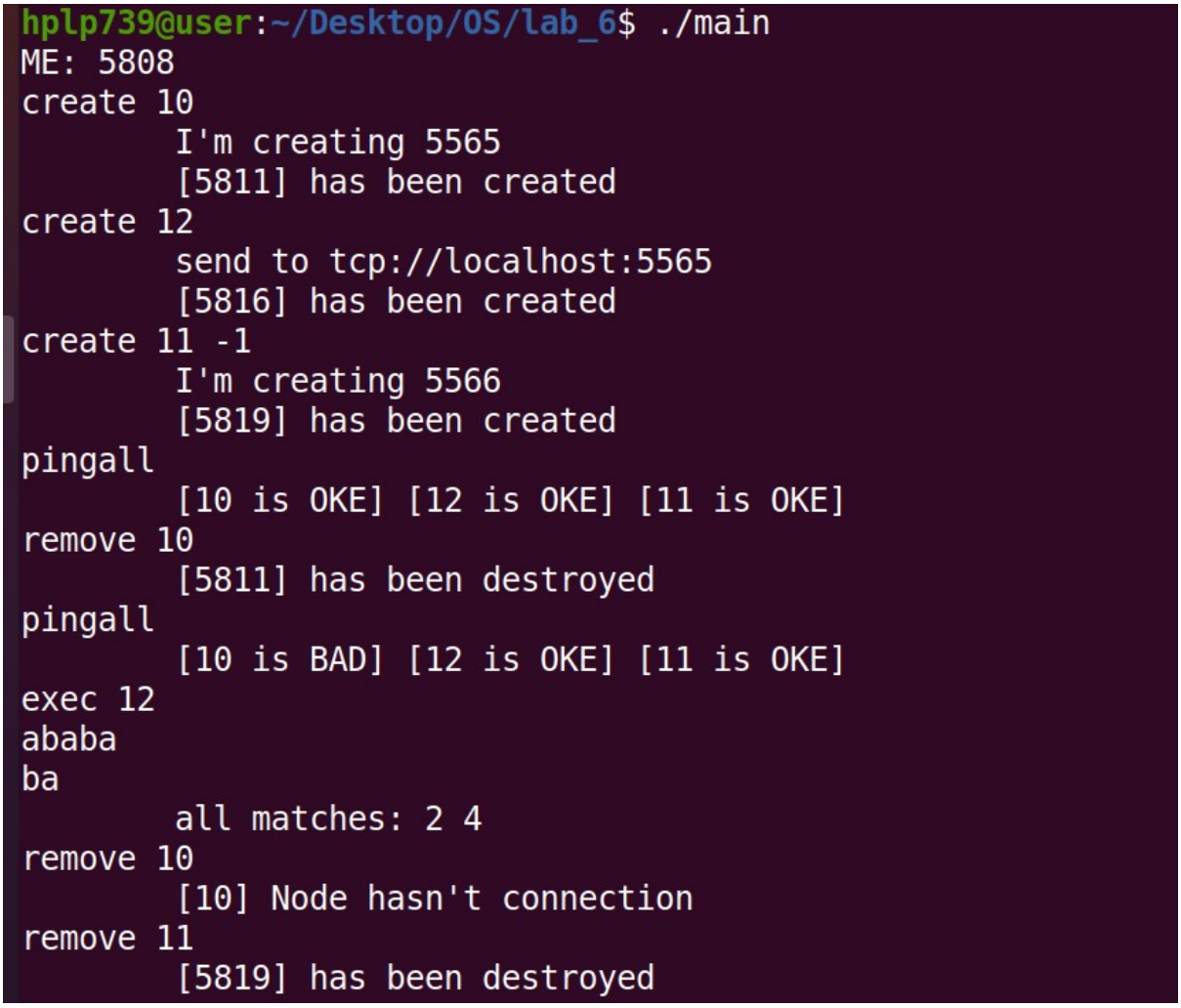
```
    zmq_connect(requester, addr_connection);

    zmq_msg_t msg;
    zmq_msg_init(&msg);
    zmq_msg_recv(&msg, socet, 0);
    uint8_t* data = (uint8_t*)zmq_msg_data(&msg);
    uint16_t event = *(uint16_t*)(data);

    zmq_close(requester);
    zmq_close(socet);
    zmq_msg_close(&msg);
    zmq_ctx_destroy(context);
    if (event == ZMQ_EVENT_CONNECT_RETRIED) {
        return false;
    } else {
        return true;
    }
}
```

```
================================  msgQ.h ==================================
#ifndef LAB_6_MSGQ_H
#define LAB_6_MSGQ_H

#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <zmq.h>
#include <assert.h>

#define MAX_LEN 64
#define MN 30

#define ERR_ZMQ_CTX           100
#define ERR_ZMQ_SOCKET        101
#define ERR_ZMQ_BIND          102
#define ERR_ZMQ_CLOSE         102
```

```
#define ERR_ZMQ_CONNECT        104
#define ERR_ZMQ_DISCONNECT     105
#define ERR_ZMQ_MSG            106

#define SERVER_SOCKET_PATTERN        "tcp://localhost:"
#define PING_SOCKET_PATTERN          "inproc://ping"
#define TCP_SOCKET_PATTERN           "tcp://*:"
#define MIN_ADDR 5555

#define TERMINATOR (-3000)

typedef enum {
   create,
   delete,
   exec,
   success
} cmd_type;

typedef struct {
   cmd_type  cmd;
   int       value;
   char      str[MAX_LEN];
   char      sub[MAX_LEN];
   int       res[MAX_LEN];
} message;

void  clear_token(message* msg);
void  create_addr(char* addr, int id);
void  bind_zmq_socket(void* socket, char* endpoint);

void* create_zmq_context();
void* create_zmq_socket(void* context, const int type);

void  connect_zmq_socket(void* socket, char* endpoint);
void  disconnect_zmq_socket(void* socket, char* endpoint);
void  reconnect_zmq_socket(void* socket, int to, char* addr);
void  close_zmq_socket(void* socket);
void  destroy_zmq_context(void* context);
```

```
bool receive_msg_wait(void* socket, message* token);
bool send_msg_wait(void* socket, message* token);
bool ping_process(int id);

// TODO move implementation "std:vector<int>" to a special .h and .c files
int* create_vector();
int length(const int* ptr);
int* push_back(int* ptr, int value);
void erase(int* ptr, int value);
void destroy(int* ptr);

#endif //LAB_6_MSGQ_H
```

Демонстрация работы программы



Выводы

В ходе выполнения лабораторной работы изучил основы работы с очередями сообщений ZMQ и реализовал программу с использованием этой библиотеки. В процессе открыл для себя много нового. Почти вся информация бралась из официальной документации.