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: Лисин Р.С.

Группа: М8О–206Б–20

Вариант: 30

Преподаватель: Соколов Андрей Алексеевич

Оценка: 5

Дата: 11.12.21

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

*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.*

Вариант 30.

Топология 3: все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.

Набора команд 4 (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  
108 символов  
pattern\_string — образец  
Пример:  
> exec 10  
> abracadabra  
> abra  
Ok:10:0;7  
> exec 10  
> abracadabra  
> mmm  
Ok:10: -1  
Примечания: Выбор алгоритма поиска не важен  
Тип проверки доступности узлов  
Команда проверки 1  
Формат команды: pingall  
Вывод всех недоступных узлов вывести разделенные через точку запятую.  
Пример:  
> pingall  
Ok: -1 // Все узлы доступны  
> pingall  
Ok: 7;10;15 // узлы 7, 10, 15 — недоступны

**Общие сведения о программе**

Система состоит из двух программ: сервер и клиент. Используется сервер сообщений ZeroMQ.

Программа сервер компилируется из файла server.c. Программа клиент из файла client.c. Обе программы используют заголовочный файл zmq\_comm.h, который содержит обертки над вызовами

Также используется заголовочные файлы: stdio.h, string.h, signal.h, stdbool.h, stdio.h, stdbool.h, sys/time.h, zmq.h, stdlib.h, zmq\_comm.h. В программе используются следующие системные вызовы:

1. **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.
2. **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[].
3. **raise —** посылает исполняющейся в текущий момент программе сигнал, который зада­ется параметром signal.

Также были использованы следующие вызовы из библиотеки ZMQ:

1. **zmq\_ctx\_new ––** создает новый контекст ZMQ.
2. **zmq\_connect** — создает входящее соединение на сокет.
3. **zmq\_disconnect** — отсоединяет сокет от заданного endpoint’a.
4. **zmq\_socket** — создает ZMQ сокет.
5. **zmq\_setsockopt** — задает параметры ZMQ сокета.
6. **zmq\_close —** закрывает ZMQ сокет.
7. **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

**Основные файлы программы**

**zmq\_comm.h :**

#include <stdbool.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <errno.h>

#include <zmq.h>

#define MAX\_LEN 64

#define ERR\_ZMQ\_CTX 100

#define ERR\_ZMQ\_SOCKET 101

#define ERR\_ZMQ\_BIND 102

#define ERR\_ZMQ\_CLOSE 102

#define ERR\_ZMQ\_CTX\_DESTROY 103

#define ERR\_ZMQ\_CONNECT 104

#define ERR\_ZMQ\_DISCONNECT 105

#define ERR\_ZMQ\_SEND 106

#define SERVER\_SOCKET\_PUB "ipc://tmp/server\_pub"

#define CLIENT\_PARENT\_PUB\_PATTERN "ipc://tmp/client\_parent\_pub\_"

#define CLIENT\_LEFT\_PUB\_PATTERN "ipc://tmp/client\_left\_pub\_"

#define CLIENT\_RIGHT\_PUB\_PATTERN "ipc://tmp/client\_right\_pub\_"

#define CLIENT\_EXE "./client"

#define CLIENT\_SIG\_ERR 3

#define SERVER\_SIG\_ERR 4

typedef enum {

create, // to\_id = id to create

delete,

kill\_all, // no to\_id

ping,

change\_sub,

test,

substring

} cmd\_type;

typedef struct {

cmd\_type cmd;

int to\_id;

int value;

char str[MAX\_LEN];

char sub[MAX\_LEN];

} message;

typedef enum {

server\_pub,

client\_left\_pub,

client\_right\_pub,

client\_parent\_pub

} endpoint\_type;

void create\_msg(message\* msg, const cmd\_type cmd, const int to\_id, const int value);

void print\_message(message\* msg);

void create\_zmq\_msg(zmq\_msg\_t\* zmq\_msg, const message\* msg);

void send\_msg(void\* socket, const message\* msg);

int get\_msg(void\* socket, message\* msg);

void create\_endpoint(char\* endpoint, int id, endpoint\_type type);

void\* create\_zmq\_context();

void connect\_zmq\_socket(void\* socket, char\* endpoint);

void disconnect\_zmq\_socket(void\* socket, char\* endpoint);

void bind\_zmq\_socket(void\* socket, char\* endpoint);

void unbind\_zmq\_socket(void\* socket, char\* endpoint);

void reconnect\_zmq\_socket(void\* socket, char\* from, char\* to);

void\* create\_zmq\_socket(void\* context, const int type);

void close\_zmq\_socket(void\* socket);

void destroy\_zmq\_context(void\* context);

**zmq\_comm.c :**

#include "zmq\_comm.h"

void create\_msg(message\* msg, const cmd\_type cmd, const int to\_id, const int value) {

msg->cmd = cmd;

msg->to\_id = to\_id;

msg->value = value;

}

void print\_message(message\* msg) {

printf("\tmsg content:\n"

"\t cmd = \"%d\"\n"

"\tto\_id = \"%d\"\n"

"\tvalue = \"%d\"\n",

msg->cmd, msg->to\_id, msg->value);

fflush(stdout);

}

void create\_zmq\_msg(zmq\_msg\_t\* zmq\_msg, const message\* msg) {

zmq\_msg\_init\_size(zmq\_msg, sizeof(\*msg));

memcpy(zmq\_msg\_data(zmq\_msg), msg, sizeof(\*msg));

}

void send\_msg(void\* socket, const message\* msg) {

zmq\_msg\_t zmq\_msg;

create\_zmq\_msg(&zmq\_msg, msg);

if (!zmq\_msg\_send(&zmq\_msg, socket, 0)) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR send\_msg \n");

exit(ERR\_ZMQ\_SEND);

}

zmq\_msg\_close(&zmq\_msg);

}

// 1 on success

int get\_msg(void\* socket, message\* msg) {

zmq\_msg\_t zmq\_msg;

zmq\_msg\_init(&zmq\_msg);

zmq\_msg\_init\_size(&zmq\_msg, sizeof(message));

int rc = zmq\_msg\_recv(&zmq\_msg, socket, 0);

if (rc == -1 && errno == EAGAIN) {

return 0;

}

memcpy(msg, zmq\_msg\_data(&zmq\_msg), sizeof(message));

zmq\_msg\_close(&zmq\_msg);

return 1;

}

void create\_endpoint(char\* endpoint, int id, endpoint\_type type) {

char id\_str[MAX\_LEN];

sprintf(id\_str, "%d", id);

char\* pattern;

if (type == server\_pub) {

strcpy(endpoint, SERVER\_SOCKET\_PUB);

return;

} else if (type == client\_left\_pub) {

pattern = CLIENT\_LEFT\_PUB\_PATTERN;

} else if (type == client\_right\_pub) {

pattern = CLIENT\_RIGHT\_PUB\_PATTERN;

} else if (type == client\_parent\_pub) {

pattern = CLIENT\_PARENT\_PUB\_PATTERN;

}

strcpy(endpoint, pattern);

strcat(endpoint, id\_str);

}

///

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 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 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 unbind\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_unbind(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_unbind ");

exit(ERR\_ZMQ\_BIND);

}

}

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);

}

if (type == ZMQ\_SUB) {

zmq\_setsockopt(socket, ZMQ\_SUBSCRIBE, 0, 0);

}

return socket;

}

void reconnect\_zmq\_socket(void\* socket, char\* from, char\* to) {

disconnect\_zmq\_socket(socket, from);

connect\_zmq\_socket(socket, to);

}

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);

}

}

**server.c**

#include <stdio.h>

#include <string.h>

#include <signal.h>

#include <stdbool.h>

#include "zmq\_comm.h"

static void\* CONTEXT;

static void\* PUBLISHER;

static void\* SUBSCRIBER;

char PUB\_ENDPOINT[MAX\_LEN];

char SUB\_ENDPOINT[MAX\_LEN];

bool FIRST\_CLIENT\_CONNECTED;

int FIRST\_CLIENT\_ID;

pid\_t SERVER\_PID;

int FEEDBACK\_TIME;

void init\_server() {

FEEDBACK\_TIME = 1000;

CONTEXT = create\_zmq\_context();

PUBLISHER = create\_zmq\_socket(CONTEXT, ZMQ\_PUB);

create\_endpoint(PUB\_ENDPOINT, 0, server\_pub);

bind\_zmq\_socket(PUBLISHER, SERVER\_SOCKET\_PUB);

SUBSCRIBER = create\_zmq\_socket(CONTEXT, ZMQ\_SUB);

zmq\_setsockopt(SUBSCRIBER, ZMQ\_RCVTIMEO, &FEEDBACK\_TIME, sizeof(FEEDBACK\_TIME));

FIRST\_CLIENT\_CONNECTED = false;

}

void deinit\_server() {

unbind\_zmq\_socket(PUBLISHER, PUB\_ENDPOINT);

close\_zmq\_socket(PUBLISHER);

if (FIRST\_CLIENT\_CONNECTED) {

disconnect\_zmq\_socket(SUBSCRIBER, SUB\_ENDPOINT);

}

close\_zmq\_socket(SUBSCRIBER);

destroy\_zmq\_context(CONTEXT);

}

void term\_clients() {

if (FIRST\_CLIENT\_CONNECTED) {

message msg;

msg.cmd = kill\_all;

send\_msg(PUBLISHER, &msg);

}

}

void hdl\_signal(int signal) {

printf("[%d] Terminating server...\n", SERVER\_PID); fflush(stdout);

term\_clients();

deinit\_server();

exit(EXIT\_SUCCESS);

}

void send\_exec\_msg(const int to\_id, const cmd\_type type, char\* str, char\* sub) {

message msg;

strcpy(msg.str, str);

strcpy(msg.sub, sub);

create\_msg(&msg, type, to\_id, 0);

send\_msg(PUBLISHER, &msg);

}

void send\_create\_msg(const int to\_create\_id) {

message msg;

create\_msg(&msg, create, to\_create\_id, 0);

send\_msg(PUBLISHER, &msg);

}

void send\_ping\_msg(const int to\_id) {

message msg;

create\_msg(&msg, ping, to\_id, 0);

send\_msg(PUBLISHER, &msg);

}

void send\_remove\_msg(const int to\_id) {

message msg;

create\_msg(&msg, delete, to\_id, 0);

send\_msg(PUBLISHER, &msg);

}

// returns PID on success, either 0

int ping\_client(const int client\_id) {

send\_ping\_msg(client\_id);

message msg;

if (!get\_msg(SUBSCRIBER, &msg)) {

return 0;

}

if (msg.to\_id == -1 && msg.cmd == ping) {

return msg.value;

}

fprintf(stderr, "[%d] SERVER ERROR: received another msg in ping\_client\n", SERVER\_PID); fflush(stdout);

print\_message(&msg);

return 0;

}

int create\_child\_process(const int client\_id) {

if (client\_id < 1) {

fprintf(stderr, "[%d] SERVER ERROR: invalid id\n", SERVER\_PID);

return -1;

}

if (!FIRST\_CLIENT\_CONNECTED) {

FIRST\_CLIENT\_ID = client\_id;

char parent\_sub\_endpoint[MAX\_LEN]; // Подписка клиента на родителя

char parent\_pub\_endpoint[MAX\_LEN]; // Паблиш клиента родителю

create\_endpoint(parent\_sub\_endpoint, 0, server\_pub);

create\_endpoint(parent\_pub\_endpoint, client\_id, client\_parent\_pub);

int fork\_pid = fork();

if (fork\_pid < 0) {

fprintf(stderr, "[%d] SERVER ERROR: Unable to fork a child\n", SERVER\_PID);

exit(10);

} else if (fork\_pid == 0) {

char client\_id\_str[MAX\_LEN];

sprintf(client\_id\_str, "%d", client\_id);

// argv = [id, parent\_id, parent\_sub\_end, parent\_pub\_end]

execl(CLIENT\_EXE, CLIENT\_EXE, client\_id\_str, "-1", parent\_sub\_endpoint, parent\_pub\_endpoint, NULL);

} else {

strcpy(SUB\_ENDPOINT, parent\_pub\_endpoint);

connect\_zmq\_socket(SUBSCRIBER, SUB\_ENDPOINT);

FIRST\_CLIENT\_CONNECTED = true;

printf("[%d] OK: %d\n", SERVER\_PID, fork\_pid);

}

} else {

int client\_pid = ping\_client(client\_id);

if (client\_pid) {

fprintf(stderr, "[%d] Error: already exist with pid = \"%d\"\n", SERVER\_PID, client\_pid);

return -1;

} else {

send\_create\_msg(client\_id);

ping\_client(client\_id);

int client\_pid = ping\_client(client\_id);

if (!client\_pid) {

fprintf(stderr, "[%d] Error: client wasn't created\n", SERVER\_PID);

return -1;

} else {

printf("[%d] OK: %d\n", SERVER\_PID, client\_pid);

}

}

}

return 0;

}

void exec\_command(const int client\_id, const cmd\_type cmd, char\* str, char\* sub) {

if (!ping\_client(client\_id)) {

fprintf(stderr, "[%d] Error: not found\n", SERVER\_PID);

return;

}

send\_exec\_msg(client\_id, cmd, str, sub);

message msg;

if (!get\_msg(SUBSCRIBER, &msg)) {

fprintf(stderr, "[%d] Error: client haven't responed\n", SERVER\_PID);

return;

}

if (msg.to\_id == -1 && msg.cmd == cmd) {

if (cmd == substring) {

printf("[%d] OK:%d:%s\n", SERVER\_PID, client\_id, msg.str);

}

return;

}

fprintf(stderr, "[%d] SERVER ERROR: received another msg in exec\_command\n", SERVER\_PID); fflush(stdout);

print\_message(&msg);

return;

}

void remove\_client(const int client\_id) {

int ping\_res = ping\_client(client\_id);

if (!ping\_res) {

fprintf(stderr, "[%d] Error: not found\n", SERVER\_PID);

return;

}

send\_remove\_msg(client\_id);

if (FIRST\_CLIENT\_ID == client\_id) {

disconnect\_zmq\_socket(SUBSCRIBER, SUB\_ENDPOINT);

memset(SUB\_ENDPOINT, 0, MAX\_LEN);

FIRST\_CLIENT\_CONNECTED = false;

}

if (!ping\_client(client\_id)) {

printf("[%d] OK\n", SERVER\_PID);

} else {

fprintf(stderr, "[%d] Error: client wasn't removed\n", SERVER\_PID);

}

}

void skip\_line() {

char cc = getchar();

while (cc != '\n' && cc != EOF) { cc = getchar(); }

}

/\*

FUNCTIONS :

+ create id

+ remove id

+ exec id

str

substr

+ pingall

\*/

void input\_loop() {

char cmd[MAX\_LEN];

int ids[MAX\_LEN];

int id;

int control = 0;

while (3) {

int scanf\_res = scanf("%s", cmd);

if (scanf\_res == EOF) {

printf("[%d] Shutting down server...\n", SERVER\_PID);

break;

}

else if (scanf\_res == 0) {

printf("[%d] Invalid command\n", SERVER\_PID);

skip\_line();

continue;

}

if (strcmp(cmd, "pingall")) {

scanf\_res = scanf("%d", &id);

if (scanf\_res == EOF) {

printf("[%d] Shutting down server...\n", SERVER\_PID);

break;

}

else if (scanf\_res == 0) {

printf("[%d] Invalid argument\n", SERVER\_PID);

skip\_line();

continue;

}

}

if (!strcmp(cmd, "create")) {

if (create\_child\_process(id) != -1) {

ids[control++] = id;

}

}

else if (!strcmp(cmd, "remove")) {

remove\_client(id);

}

else if (!strcmp(cmd, "exec")) {

char str[MAX\_LEN];

char sub[MAX\_LEN];

int scanf\_res = scanf("%s", str);

int another\_res = scanf("%s", sub);

if (scanf\_res == EOF || another\_res == EOF) {

printf("[%d] Shutting down server.\n", SERVER\_PID);

break;

} else if (scanf\_res == 0 || another\_res == 0) {

printf("[%d] Invalid exec command\n", SERVER\_PID);

skip\_line();

continue;

}

cmd\_type type = substring;

exec\_command(id, type, str, sub);

}

else if (!strcmp(cmd, "pingall")) {

bool flag = true;

printf("[%d] OK: ", SERVER\_PID);

for (int i = 0; i < control; i++) {

if (!ping\_client(ids[i])) {

printf("%d;", ids[i]);

flag = false;

}

}

if (flag) {

printf("-1");

}

printf("\n");

}

else if (!strcmp(cmd, "test")) {

if (id == 0) {

printf("[%d] I'm server\n"

"\t publisher = \"%s\"\n"

"\t subscriber = \"%s\"\n",

SERVER\_PID, PUB\_ENDPOINT, SUB\_ENDPOINT);

fflush(stdout);

}

else {

message msg;

create\_msg(&msg, test, id, 0);

send\_msg(PUBLISHER, &msg);

}

}

else {

printf("[%d] Invalid command\n", SERVER\_PID);

skip\_line();

continue;

}

}

}

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

if (signal(SIGINT, hdl\_signal) == SIG\_ERR) {

fprintf(stderr, "[%d] SERVER", getpid());

perror("ERROR signal ");

return SERVER\_SIG\_ERR;

}

if (signal(SIGSEGV, hdl\_signal) == SIG\_ERR) {

fprintf(stderr, "[%d] SERVER", getpid());

perror("ERROR signal ");

return SERVER\_SIG\_ERR;

}

SERVER\_PID = getpid();

printf("[%d] Starting server...\n", SERVER\_PID);

init\_server();

input\_loop();

term\_clients();

deinit\_server();

return 0;

}

**client.c**

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <sys/time.h>

#include "zmq\_comm.h"

// context and sockets

static void \* CONTEXT;

static void \* SUBSCRIBER;

static void \* PARENT\_PUB;

static void \* RIGHT\_PUB;

static void \* LEFT\_PUB;

// SOCKETS ENDPOINT

// for subscriber socket

static char PARENT\_SUB\_ENDPOINT[MAX\_LEN]; // recieve by parent

static char LEFT\_SUB\_ENDPOINT[MAX\_LEN];

static char RIGHT\_SUB\_ENDPOINT[MAX\_LEN];

static char PARENT\_PUB\_ENDPOINT[MAX\_LEN]; // recieve by parent

static char LEFT\_PUB\_ENDPOINT[MAX\_LEN];

static char RIGHT\_PUB\_ENDPOINT[MAX\_LEN];

static int PARENT\_ID;

static int CLIENT\_PID;

static int CLIENT\_ID;

bool HAS\_LEFT;

bool HAS\_RIGHT;

void init\_client() {

create\_endpoint(LEFT\_PUB\_ENDPOINT, CLIENT\_ID, client\_left\_pub);

create\_endpoint(RIGHT\_PUB\_ENDPOINT, CLIENT\_ID, client\_right\_pub);

CONTEXT = create\_zmq\_context();

SUBSCRIBER = create\_zmq\_socket(CONTEXT, ZMQ\_SUB);

connect\_zmq\_socket(SUBSCRIBER, PARENT\_SUB\_ENDPOINT);

PARENT\_PUB = create\_zmq\_socket(CONTEXT, ZMQ\_PUB);

bind\_zmq\_socket(PARENT\_PUB, PARENT\_PUB\_ENDPOINT);

RIGHT\_PUB = create\_zmq\_socket(CONTEXT, ZMQ\_PUB);

bind\_zmq\_socket(RIGHT\_PUB, RIGHT\_PUB\_ENDPOINT);

LEFT\_PUB = create\_zmq\_socket(CONTEXT, ZMQ\_PUB);

bind\_zmq\_socket(LEFT\_PUB, LEFT\_PUB\_ENDPOINT);

}

void deinit\_client() {

// printf("[%d] Deinit client %d.\n", CLIENT\_PID, CLIENT\_ID); fflush(stdout);

disconnect\_zmq\_socket(SUBSCRIBER, PARENT\_SUB\_ENDPOINT);

if (HAS\_LEFT) {

disconnect\_zmq\_socket(SUBSCRIBER, LEFT\_SUB\_ENDPOINT);

}

if (HAS\_RIGHT) {

disconnect\_zmq\_socket(SUBSCRIBER, RIGHT\_SUB\_ENDPOINT);

}

close\_zmq\_socket(SUBSCRIBER);

unbind\_zmq\_socket(PARENT\_PUB, PARENT\_PUB\_ENDPOINT);

unbind\_zmq\_socket(LEFT\_PUB, LEFT\_PUB\_ENDPOINT);

unbind\_zmq\_socket(RIGHT\_PUB, RIGHT\_PUB\_ENDPOINT);

close\_zmq\_socket(PARENT\_PUB);

close\_zmq\_socket(LEFT\_PUB);

close\_zmq\_socket(RIGHT\_PUB);

}

void hdl\_signal(int signal) {

// printf("[%d] Terminating client %d...\n", CLIENT\_PID, CLIENT\_ID); fflush(stdout);

deinit\_client();

exit(EXIT\_SUCCESS);

}

void print\_info() {

printf("[%d] I'm client %d\n"

"\t parent\_id = \"%d\"\n"

"\t parent\_sub = \"%s\"\n"

"\t left\_sub = \"%s\"\n"

"\t right\_sub = \"%s\"\n"

"\t parent\_pub = \"%s\"\n"

"\t left\_pub = \"%s\"\n"

"\t right\_pub = \"%s\"\n",

CLIENT\_PID, CLIENT\_ID, PARENT\_ID,

PARENT\_SUB\_ENDPOINT, LEFT\_SUB\_ENDPOINT, RIGHT\_SUB\_ENDPOINT,

PARENT\_PUB\_ENDPOINT, LEFT\_PUB\_ENDPOINT, RIGHT\_PUB\_ENDPOINT);

fflush(stdout);

}

void create\_child\_process(const int child\_id) {

// printf("[%d] Client %d: creating child...\n", CLIENT\_PID, CLIENT\_ID); fflush(stdout);

char parent\_sub\_endpoint[MAX\_LEN]; // parent --> child

bool is\_left = child\_id < CLIENT\_ID;

if (is\_left) {

strcpy(parent\_sub\_endpoint, LEFT\_PUB\_ENDPOINT);

} else {

strcpy(parent\_sub\_endpoint, RIGHT\_PUB\_ENDPOINT);

}

char parent\_pub\_endpoint[MAX\_LEN]; // parent <-- child

create\_endpoint(parent\_pub\_endpoint, child\_id, client\_parent\_pub);

int fork\_pid = fork();

if (fork\_pid < 0) {

fprintf(stderr, "[%d] Client error: Unable to fork a child\n", CLIENT\_PID);

exit(10);

} else if (fork\_pid == 0) {

char child\_id\_str[MAX\_LEN];

char parent\_id\_str[MAX\_LEN];

sprintf(child\_id\_str, "%d", child\_id);

sprintf(parent\_id\_str, "%d", CLIENT\_ID);

// argv = [id, parent\_id, parent\_sub\_end, parent\_pub\_end]

execl(CLIENT\_EXE, CLIENT\_EXE, child\_id\_str, parent\_id\_str, parent\_sub\_endpoint, parent\_pub\_endpoint, NULL);

} else {

if (is\_left) {

strcpy(LEFT\_SUB\_ENDPOINT, parent\_pub\_endpoint);

HAS\_LEFT = true;

} else {

strcpy(RIGHT\_SUB\_ENDPOINT, parent\_pub\_endpoint);

HAS\_RIGHT = true;

}

connect\_zmq\_socket(SUBSCRIBER, parent\_pub\_endpoint);

}

}

void change\_subscription(const int from\_id) {

if (from\_id < CLIENT\_ID) {

HAS\_LEFT = false;

disconnect\_zmq\_socket(SUBSCRIBER, LEFT\_SUB\_ENDPOINT);

memset(LEFT\_SUB\_ENDPOINT, 0, MAX\_LEN);

} else {

HAS\_RIGHT = false;

disconnect\_zmq\_socket(SUBSCRIBER, RIGHT\_SUB\_ENDPOINT);

memset(RIGHT\_SUB\_ENDPOINT, 0, MAX\_LEN);

}

}

void client\_loop() {

while(3) {

message msg;

get\_msg(SUBSCRIBER, &msg);

// back to server msg

if (msg.to\_id == -1) {

//printf("[%d] Client %d: resending msg to top through \"%s\"\n", CLIENT\_PID, CLIENT\_ID, PARENT\_PUB\_ENDPOINT);

fflush(stdout);

send\_msg(PARENT\_PUB, &msg);

continue;

}

// redirect msg, except no id one

if (msg.cmd != create && msg.cmd != kill\_all) {

if (msg.to\_id != CLIENT\_ID) {

if (msg.to\_id < CLIENT\_ID) {

send\_msg(LEFT\_PUB, &msg);

} else {

send\_msg(RIGHT\_PUB, &msg);

}

continue;

}

}

// parsing msg

if (msg.cmd == create) {

if (msg.to\_id == CLIENT\_ID) {

msg.to\_id = -1;

msg.value = CLIENT\_PID;

}

else if (msg.to\_id < CLIENT\_ID && HAS\_LEFT) {

send\_msg(LEFT\_PUB, &msg);

// printf("[%d] Client %d: send to left\n", CLIENT\_PID, CLIENT\_ID); fflush(stdout);

}

else if (msg.to\_id > CLIENT\_ID && HAS\_RIGHT) {

send\_msg(RIGHT\_PUB, &msg);

// printf("[%d] Client %d: send to right\n", CLIENT\_PID, CLIENT\_ID); fflush(stdout);

}

else {

create\_child\_process(msg.to\_id);

}

}

else if (msg.cmd == delete) {

if (PARENT\_ID != -1) {

msg.to\_id = PARENT\_ID;

msg.value = CLIENT\_ID;

msg.cmd = change\_sub;

// printf("[%d] Client %d: send unsub to %d\n", CLIENT\_PID, CLIENT\_ID, PARENT\_ID); fflush(stdout);

send\_msg(PARENT\_PUB, &msg);

}

msg.cmd = kill\_all;

send\_msg(LEFT\_PUB, &msg);

send\_msg(RIGHT\_PUB, &msg);

raise(SIGTERM);

}

else if (msg.cmd == kill\_all) {

send\_msg(LEFT\_PUB, &msg);

send\_msg(RIGHT\_PUB, &msg);

raise(SIGTERM);

}

else if (msg.cmd == ping) {

// printf("[%d] Client %d: recived ping msg\n", CLIENT\_PID, CLIENT\_ID); fflush(stdout);

msg.to\_id = -1;

msg.value = CLIENT\_PID;

send\_msg(PARENT\_PUB, &msg);

// printf("\t[%d] Client %d: send ping msg to \"%s\"\n", CLIENT\_PID, CLIENT\_ID, PARENT\_PUB\_ENDPOINT); fflush(stdout);

}

else if (msg.cmd == change\_sub) {

change\_subscription(msg.value);

}

else if (msg.cmd == substring) {

int control = 0;

char str[MAX\_LEN];

for (int i = 0; i < strlen(msg.str); i++) {

int save\_pos = i;

for (int j = 0; j < strlen(msg.sub); j++) {

if (msg.str[i] != msg.sub[j]) {

break;

}

if (j == strlen(msg.sub) - 1) {

str[control++] = save\_pos + '0';

str[control++] = ';';

break;

}

i++;

}

}

if (control) {

str[control - 1] = '\0';

}

else {

str[control++] = '-';

str[control++] = '1';

str[control] = '\0';

}

strcpy(msg.str, str);

msg.to\_id = -1;

send\_msg(PARENT\_PUB, &msg);

}

else if (msg.cmd == test) {

print\_info();

}

else {

printf("[%d] Client error: unknown command\n", CLIENT\_PID);

print\_message(&msg);

continue;

}

}

}

// argv = [id, parent\_id, parent\_sub\_end, parent\_pub\_end]

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

if (signal(SIGTERM, hdl\_signal) == SIG\_ERR) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR signal ");

return CLIENT\_SIG\_ERR;

}

CLIENT\_PID = getpid();

CLIENT\_ID = atoi(argv[1]);

PARENT\_ID = atoi(argv[2]);

strcpy(PARENT\_SUB\_ENDPOINT, argv[3]);

strcpy(PARENT\_PUB\_ENDPOINT, argv[4]);

init\_client();

// printf("[%d] Starting client %d...\n", CLIENT\_PID, CLIENT\_ID);

// print\_info();

client\_loop();

deinit\_client();

return 0;

}

**Makefile:**

CC = gcc

CC\_FLAGS = -pedantic -Wall

LD\_FLAGS = -lzmq -L.

CC\_LIB\_FLAGS = -c -pedantic -Wall -fPIC

LD\_LIB\_FLAGS = -shared -lzmq

EXECUTABLE1 = server

EXECUTABLE2 = client

SRC\_EXE1 = server.c

SRC\_EXE2 = client.c

OBJ\_EXE1 = $(SRC\_EXE1:.c=.o)

OBJ\_EXE2 = $(SRC\_EXE2:.c=.o)

LIB = -ltools

SRC\_LIB = zmq\_comm.c

OBJ\_LIB = $(SRC\_LIB:.c=.o)

SO\_LIB = libtools.so

all: $(EXECUTABLE1) $(EXECUTABLE2)

$(EXECUTABLE1) : $(SRC\_EXE1) $(SO\_LIB)

$(CC) $(CC\_FLAGS) $(SRC\_EXE1) -o $(EXECUTABLE1) $(LD\_FLAGS) $(LIB)

$(EXECUTABLE2) : $(SRC\_EXE2) $(SO\_LIB)

$(CC) $(CC\_FLAGS) $(SRC\_EXE2) -o $(EXECUTABLE2) $(LD\_FLAGS) $(LIB)

$(SO\_LIB) : $(OBJ\_LIB)

$(CC) $^ -o $@ $(LD\_LIB\_FLAGS)

$(OBJ\_LIB) : $(SRC\_LIB)

$(CC) $(CC\_LIB\_FLAGS) $^

clean:

rm -f \*.o \*.so $(EXECUTABLE1) $(EXECUTABLE2)

**Пример работы**

roma@DESKTOP-JD58QU2:~/os\_lab\_6$ ./server

[41] Starting server...

create 10

[41] OK: 44

test 10

[44] I'm client 10

parent\_id = "-1"

parent\_sub = "ipc://tmp/server\_pub"

left\_sub = ""

right\_sub = ""

parent\_pub = "ipc://tmp/client\_parent\_pub\_10"

left\_pub = "ipc://tmp/client\_left\_pub\_10"

right\_pub = "ipc://tmp/client\_right\_pub\_10"

create 5

[41] OK: 47

test 5

[47] I'm client 5

parent\_id = "10"

parent\_sub = "ipc://tmp/client\_left\_pub\_10"

left\_sub = ""

right\_sub = ""

parent\_pub = "ipc://tmp/client\_parent\_pub\_5"

left\_pub = "ipc://tmp/client\_left\_pub\_5"

right\_pub = "ipc://tmp/client\_right\_pub\_5"

exec 5

ababab

ab

[41] OK:5:0;2;4

pingall

[41] OK: -1

remove 5

[41] OK

pingall

[41] OK: 5;

^C[41] Terminating server...

**Вывод**

В ходе написания клиентской программы я понял, что можно полностью отказаться от хранения бинарного дерева на серверной стороне, что позволило реализовать взаимодействие в формате черного ящика.

После первой тестовой версии программы от паттерна «Request – Reply» пришлось отказаться в пользу паттерна «Publisher – Subscriber» в виду больше гибкости.

Также я научился использовать библиотеку ZMQ, в процессе я открыл для себя много нового. Почти вся информация бралась из официальной документации.