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»

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

Студент: Лукманова Аэлита
Группа: М8О–201Б–19
Вариант: 39
Преподаватель: Миронов Е. С.
Оценка: _____
Дата: _____
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.

Вариант 39:

Топология: узлы находятся в идеально сбалансированном бинарном дереве. Каждый следующий узел должен добавляться в самое наименьшее левое поддерево.

Тип команды для вычислительных узлов: поиск подстроки в строке.

Тип проверки доступности узлов: Heartbit. 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.

Общие сведения о программе

Программа представлена в виде двух директорий: avl и client_and_server. В первой находится реализация avl дерева на C++ для построения и перестроения дерева процессов. Во второй папке находятся файлы, отвечающие за клиент-серверное сообщение. Программу собирать при помощи общего Makefile’a, запускать программу при помощи команд ./server и ./client. В программе используется очередь сообщений zmq. Так же

используются заголовочные файлы: stdio.h, unistd.h, stdlib.h, stdbool.h, signal.h. В программе используются следующие системные вызовы:

1. **kill** — используется для отправки любого сигнала на любой процесс. В случае успеха возвращает 0. Иначе -1.
2. **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.
3. **execl –** загружает и запускает другую программу. Таким образом, новая программа полностью замещает текущий процесс. Новая программа начинает свое выполнение с функции main. Аргументы командной строки передаются в форме списка arg0, arg1.... argn, NULL.

В программе используются следующие функции из zeromq:

1. **zmq_msg_init** — создает сообщение для очереди сообщений.
2. **zmq_msg_close –** закрывает сообщение для очереди сообщений.
3. **zmq_msg_send –** отправить сообщение на сокет.
4. **zmq_msg_recv –** принять сообщение с сокета.
5. **zmq_ctx_nex –** создать новый контекст для ввода/вывода.
6. **zmq_socket –** создать новый сокет.
7. **zmq_bind –** связать сокет с именованным файлом.
8. **zmq_connect –** присоединиться к сокету.
9. **zmq_setsockopt –** установить опции сокета.
10. **zmq_close –** закрыть сокет.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить zmq.
2. Продумать архитектуру для программы и межпроцессорного взаимодействия.
3. Написать avl дерево.
4. Написать общие функции для межпроцессного взаимодействия.
5. Реализовать клиент и сервер.
6. Интегрировать avl дерево в проект.

Основные файлы программы

avl.hpp:

```
#pragma once
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <memory>
#include <cstdint>
#include <zmq.h>
#include <unistd.h>
extern "C" {
#include "../client_and_server/zmq_handle.h"
}
/**
 * структура для процесса-вершины дерева с укаанным id
 */
struct tree_node {
    int32_t pid;
    int32_t balance;
    std::weak_ptr<tree_node> parent;
    std::shared_ptr<tree_node> left;
    std::shared_ptr<tree_node> right;

    tree_node(): pid(-1), balance(0), left(nullptr), right(nullptr) {
        parent.lock() = nullptr;
    }
    tree_node(int32_t pid): pid(pid), balance(0), left(nullptr), right(nullptr) {
        parent.lock() = nullptr;
    }
    ~tree_node() = default;
};

class avl_tree {
    private:
        // корень дерева
        std::shared_ptr<tree_node> _root;
        // рекурсивный поиск по дереву, возвращает указатель на искомый узел или
nullptr
        std::shared_ptr<tree_node> _find(int32_t pid, std::shared_ptr<tree_node>
node);
        // рекурсивынй поиск с обновлением пути
        bool _search(int32_t pid, int32_t* path, std::shared_ptr<tree_node> node,
int32_t* height);
        // рекурсивная функция для вставки
        bool _insert(int32_t pid, std::shared_ptr<tree_node> node);
        // левый поворот
        void _left_rotate(std::shared_ptr<tree_node> node);
        // правый поворот
        void _right_rotate(std::shared_ptr<tree_node> node);
        // перебалансировка вершины с балансом равным 2 по модулю
        std::shared_ptr<tree_node> _rebalance(std::shared_ptr<tree_node> node);
        // удаление всего поддерева с корнем в вершине node
        void _delete_sub_tree(std::shared_ptr<tree_node> node);
        // рекурсивно перестраиваем дерево после удаления целого поддерева
        void _rec_reconstruct(std::shared_ptr<tree_node>& new_root,
std::shared_ptr<tree_node> node);
        // функция, вызывающая _rec_reconstruct, она обновляет корень
        void _reconstruct();
        // рекурсивное обновление баланса для вставки
        void _go_up_insert(std::shared_ptr<tree_node> node, std::shared_ptr<tree_node>
prev);
        // вывод дерева на экран рекурсивно
        void _print(const std::string& prefix, std::shared_ptr<tree_node> node, bool
is_left, int32_t height);
        // поиск и удаленеи вершины
        void _remove(std::shared_ptr<tree_node> node);
        // рекурсивное обновление баланса для удаления
        void _go_up_remove(std::shared_ptr<tree_node> node, std::shared_ptr<tree_node>
prev);
    public:
```

```
        /**
         * конструктор по умолчанию
         */
        avl_tree(): _root(nullptr) {};
        /**
         * поиск в дереве по id процесса
         * возвращает вектор-путь от корневого процесса до искомого
         * если искомый процесс не найден, то возвращаемый вектор имеет длину 0
         */
        bool search(int32_t pid, int32_t* path, int32_t* path_len);
        /**
         * вставка в дерево по id процесса
         * в случае успеха вернет true
         * в случае неуспеха вернет false
         */
        bool insert(int32_t pid);
        /**
         * печать дерева на экран
         */
        void print();
        /**
         * удаление всего поддерва, корнем которого является процесс с указанным id
         * в случае успеха дерево перебалансируется
         * в случае, если такого процесса нет, то метод вернет false
         */
        bool delete_sub_tree(int32_t* pids, int32_t len);
        /**
         * удаление процесса с номером pid
         * в случае неуспеха вернет false
         */
        bool remove(int32_t pid);
        /**
         * получить pid родителя
         * в случае неуспеха вернется -1
         */
        int32_t get_parent_pid(int32_t pid);
        /**
         * получить pid корня
         * в случае неудачи вернет -1
         */
        int32_t get_root_pid();
        /**
         * деструктор по умолчанию
         */
        ~avl_tree() = default;
};
```

avl.cpp:

```
#include "avl.hpp"

void check_and_send_rebind(std::shared_ptr<tree_node> node, std::shared_ptr<tree_node>
new_parent) {
    if (!node.use_count()) {
        return;
    }
    if (!new_parent.use_count()) {
        mm_send_rebind(node->pid, -1);
    }
    else {
        mm_send_rebind(node->pid, new_parent->pid);
    }
}

void check_and_send_rebind(std::shared_ptr<tree_node> node, std::weak_ptr<tree_node>
new_parent) {
    if (!node.use_count()) {
        return;
    }
    if (!new_parent.use_count()) {
        mm_send_rebind(node->pid, -1);
    }
```

```
    else {
        mm_send_rebind(node->pid, new_parent.lock()->pid);
    }
}

// получить pid корня - начало
int32_t avl_tree::get_root_pid() {
    if (_root == nullptr) {
        return -1;
    }
    return _root->pid;
}
// получить pid корня - конец

// получить pid родителя - начало
int32_t avl_tree::get_parent_pid(int32_t pid) {
    std::shared_ptr<tree_node> node = _find(pid, _root);
    if (node == nullptr || node == _root) {
        return -1;
    }
    return node->parent.lock()->pid;
}
// получить pid родителя - конец

// поиск по дереву - начало
bool avl_tree::search(int32_t pid, int32_t* path, int32_t* path_len) {
    *path_len = 0;
    if (!_search(pid, path, _root, path_len)) {
        return false;
    }
    (*path_len)++;
    return true;
}

bool avl_tree::_search(int32_t pid, int32_t* path, std::shared_ptr<tree_node> node,
int32_t* height) {
    if (node == nullptr) {
        *height = 0;
        return false;
    }
    else if (pid == node->pid) {
        path[*height] = node->pid;
        return true;
    }
    else {
        path[*height] = node->pid;
        std::shared_ptr<tree_node> to = pid < node->pid ? node->left : node->right;
        (*height)++;
        return _search(pid, path, to, height);
    }
}
// поиск по дереву - конец

// повороты - начало
void avl_tree::_left_rotate(std::shared_ptr<tree_node> node) {
    int32_t node_b = node->balance;
    std::shared_ptr<tree_node> right_son = node->right; //запоминаем b
    int32_t rs_b = right_son->balance;
    if (right_son == nullptr) {
        return;
    }
    node->right = right_son->left; // перевешиваем a->b на a->q
    if (right_son->left != nullptr) {
        check_and_send_rebind(right_son->left, node);
        right_son->left->parent = node; // связываем q->a
    }
    check_and_send_rebind(right_son, node->parent);
    right_son->parent = node->parent; // связываем b->p
    if (node == _root) {
        _root = right_son;
    }
    else if (node == node->parent.lock()->left) {
```

```
            node->parent.lock()->left = right_son; // перевешиваем p->a на p->b
        }
        else {
            node->parent.lock()->right = right_son;
        }
        right_son->left = node; // перевешиваем b->q на b->a
        check_and_send_rebind(node, right_son);
        node->parent = right_son; // связываем a->b

        node->balance++;
        right_son->balance++;

        if (node_b == -2 && rs_b == -1) {
            node->balance = 0;
            right_son->balance = 0;
        }
        else if (node_b == -2 && rs_b == 0) {
            node->balance = -1;
            right_son->balance = 1;
        }
    }

    void avl_tree::_right_rotate(std::shared_ptr<tree_node> node) {
        int32_t node_b = node->balance;
        std::shared_ptr<tree_node> left_son = node->left; // запоминаем a
        int32_t ls_b = left_son->balance;
        if (left_son == nullptr) {
            return;
        }
        node->left = left_son->right; // перевешиваем b->a на b->q
        if (left_son->right != nullptr) {
            check_and_send_rebind(left_son->right, node);
            left_son->right->parent = node; // связываем q->b
        }
        check_and_send_rebind(left_son, node->parent);
        left_son->parent = node->parent; // связываем a->p
        if (node == _root) {
            _root = left_son;
        }
        else if (node == node->parent.lock()->right) {
            node->parent.lock()->right = left_son; // перевешиваем p->b на p->a
        }
        else {
            node->parent.lock()->left = left_son;
        }
        left_son->right = node; // перевешиваем a->q на a->b
        check_and_send_rebind(node, left_son);
        node->parent = left_son; // связываем b->a

        node->balance--;
        left_son->balance--;

        if (node_b == 2 && ls_b == 1) {
            node->balance = 0;
            left_son->balance = 0;
        }
        else if (node_b == 2 && ls_b == 0) {
            node->balance = 1;
            left_son->balance = -1;
        }
    }
    // повороты - конец

    // перебалансировка - начало
    std::shared_ptr<tree_node> avl_tree::_rebalance(std::shared_ptr<tree_node> node) {
        if (node->balance == -2) {
            if (node->right->balance > 0) {
                _right_rotate(node->right);
            }
            _left_rotate(node);
        }
        else {
```

```
        if (node->left->balance < 0)
            _left_rotate(node->left);
        _right_rotate(node);
    }
    return node->parent.lock();
}
// перебалансировка - конец

// вставка в дерево - начало
bool avl_tree::insert(int32_t pid) {
    if (_root == nullptr) {
        // отдельный случай свзязывания с мастером
        _root = std::shared_ptr<tree_node>(new tree_node(pid));
        return true;
    }
    else {
        int ret = _insert(pid, _root);
        return ret;
    }
}

bool avl_tree::_insert(int32_t pid, std::shared_ptr<tree_node> node) {
    if (pid == node->pid) {
        return false;
    }
    else if (pid < node->pid) {
        if (node->left == nullptr) {
            node->left = std::shared_ptr<tree_node>(new tree_node(pid));
            node->left->parent = node; // связывание НОВОЙ ноды с родителем
            _go_up_insert(node, node->left);
            return true;
        }
        else {
            return _insert(pid, node->left);
        }
    }
    else {
        if (node->right == nullptr) {
            node->right = std::shared_ptr<tree_node>(new tree_node(pid));
            node->right->parent = node; // связывание НОВОЙ ноды с родителем
            _go_up_insert(node, node->right);
            return true;
        }
        else {
            return _insert(pid, node->right);
        }
    }
}

void avl_tree::_go_up_insert(std::shared_ptr<tree_node> node,
std::shared_ptr<tree_node> prev) {
    if (prev == node->left) {
        node->balance++;
    }
    else {
        node->balance--;
    }
    if (node->balance == 0) {
        return;
    }
    else if (std::abs(node->balance) == 1) {
        if (node == _root) {
            return;
        }
        _go_up_insert(node->parent.lock(), node);
    }
    else {
        node = _rebalance(node);
        if (node->balance == 0 || node == _root) {
            return;
        }
        _go_up_insert(node->parent.lock(), node);
```

```
    }
}
// вставка в дерево - конец

// вывод дерева на экран - начало
void avl_tree::print() {
    _print("", _root, false, 1);
}

void avl_tree::_print(const std::string& prefix, std::shared_ptr<tree_node> node, bool
is_left, int32_t height) {
    if (node != nullptr) {
        std::string new_prefix = "";
        for (int32_t i = 0; i < height; ++i) {
            new_prefix += "    ";
        }
        _print(new_prefix, node->left, true, height + 1);
        std::cout << prefix;
        if (height == 1) {
            std::cout << "———";
        }
        else {
            std::cout << (is_left ? "┌——" : "└——");
        }
        std::cout << node->pid << "\n";
        _print(new_prefix, node->right, false, height + 1);
    }
}
// вывод дерева на экран - конец

// удаление поддерева - начало
bool avl_tree::delete_sub_tree(int32_t* pids, int32_t len) {
    for (int32_t i = 0; i < len; ++i) {
        int32_t pid = pids[i];
        std::shared_ptr<tree_node> to_delete = _find(pid, _root);
        if (to_delete == nullptr) {
            continue;
        }
        if (to_delete == _root) {
            _root = nullptr;
        }
        _delete_sub_tree(to_delete);
    }
    if (_root != nullptr) {
        _reconstruct();
    }
    return true;
}

std::shared_ptr<tree_node> avl_tree::_find(int32_t pid, std::shared_ptr<tree_node>
node) {
    if (node == nullptr) {
        return nullptr;
    }
    else if (pid == node->pid) {
        return node;
    }
    else {
        std::shared_ptr<tree_node> to = pid < node->pid ? node->left : node->right;
        return _find(pid, to);
    }
}

void avl_tree::_delete_sub_tree(std::shared_ptr<tree_node> node) {
    if (node == nullptr) {
        return;
    }
    else {
        if (node->parent.lock() != nullptr) {
            if (node->parent.lock()->left == node) {
                node->parent.lock()->left = nullptr;
            }
```

```
            else {
                node->parent.lock()->right = nullptr;
            }
            node->parent.lock() = nullptr;
        }
    }
}

void avl_tree::_reconstruct() {
    std::shared_ptr<tree_node> old_root = _root;
    _root = std::shared_ptr<tree_node>(new tree_node(old_root->pid));
    _rec_reconstruct(_root, old_root->left);
    _rec_reconstruct(_root, old_root->right);
}

void avl_tree::_rec_reconstruct(std::shared_ptr<tree_node>& new_root,
std::shared_ptr<tree_node> node) {
    if (node == nullptr) {
        return;
    }
    _insert(node->pid, new_root);
    _rec_reconstruct(new_root, node->left);
    _rec_reconstruct(new_root, node->right);
}
// удаление поддерева - конец

// удаление вершины дерева - начало
bool avl_tree::remove(int32_t pid) {
    std::shared_ptr<tree_node> node = _root;
    while ((node != nullptr) && (pid != node->pid)) {
        std::shared_ptr<tree_node> to = (pid < node->pid) ? node->left : node->right;
        node = to;
    }
    if (node == nullptr) {
        return false;
    }
    _remove(node);
    return true;
}

void avl_tree::_remove(std::shared_ptr<tree_node> node) {
    std::shared_ptr<tree_node> to_delete = node;
    int32_t to_delete_balance = to_delete->balance;
    std::shared_ptr<tree_node> to_replace;
    std::shared_ptr<tree_node> to_replace_parent;
    if (node->left == nullptr) {
        to_replace = node->right;
        if (to_replace != nullptr) {
            check_and_send_rebind(to_replace, node->parent);
            to_replace->parent = node->parent; // (!!!!!!!) связываение to_replace-
>node.parent
            to_replace_parent = node->parent.lock();
        }
        else {
            to_replace_parent = node->parent.lock();
            if (node == _root) {
                to_replace_parent = nullptr;
                _root = nullptr;
            }
        }
        if (_root != nullptr) {
            if (node != _root) {
                if (node->parent.lock()->left == node) {
                    node->parent.lock()->left = to_replace;
                }
                else {
                    node->parent.lock()->right = to_replace;
                }
            }
            else {
                _root = to_replace;
            }
```

```
        }
    }
    else if (node->right == nullptr) {
        to_replace = node->left;
        check_and_send_rebind(to_replace, node->parent);
        to_replace->parent = node->parent; // (!!!!!!) связывание to_replace-
>node.parent
        to_replace_parent = node->parent.lock();
        if (node != _root) {
            if (node->parent.lock()->left == node) {
                node->parent.lock()->left = to_replace;
            }
            else {
                node->parent.lock()->right = to_replace;
            }
        }
        else {
            _root = to_replace;
        }
    }
    else { // если есть дети и слева и справа
        std::shared_ptr<tree_node> min_in_right = node->right;
        while (min_in_right->left != nullptr) {
            min_in_right = min_in_right->left;
        }
        to_delete = min_in_right;
        to_delete_balance = to_delete->balance;
        to_replace = to_delete->right;
        if (to_delete->parent.lock() == node) {
            if (to_replace != nullptr) {
                check_and_send_rebind(to_replace, to_delete);
                to_replace->parent = to_delete; // (!!!!!!!!!) связывание to_replace-
>to_delete
            }
            to_replace_parent = to_delete;
        }
        else {
            to_delete->parent.lock()->left = to_replace;
            if (to_replace != nullptr) {
                check_and_send_rebind(to_replace, to_delete->parent);
                to_replace->parent = to_delete->parent;  // (!!!!!!!!!) связывание
to_replace->to_delete.parent
            }
            to_replace_parent = to_delete->parent.lock();
            to_delete->right = node->right;
            check_and_send_rebind(to_delete->right, to_delete);
            to_delete->right->parent = to_delete; // (!!!!!!!!!) связывание
to_delete.right->to_delete
        }
        if (node != _root) {
            if (node->parent.lock()->left == node) {
                node->parent.lock()->left = to_delete;
            }
            else {
                node->parent.lock()->right = to_delete;
            }
        }
        else {
            _root = to_delete;
        }
        check_and_send_rebind(to_delete, node->parent);
        to_delete->parent = node->parent; // (!!!!!!!!!) связывание to_delete ->
node.parent
        to_delete->left = node->left;

        check_and_send_rebind(to_delete->left, to_delete);
        to_delete->left->parent = to_delete; // (!!!!!!!!!) связывание to_delete.left
-> to_delete
        to_delete->balance = node->balance;
    }
    if (to_replace_parent != nullptr) {
        _go_up_remove(to_replace_parent, to_replace);
```

```
    }
}

void avl_tree::_go_up_remove(std::shared_ptr<tree_node> node,
std::shared_ptr<tree_node> prev) {
    if (node->left == nullptr && node->right == nullptr) {
        node->balance = 0;
    }
    else {
        if (prev == node->left) {
            node->balance--;
        }
        else {
            node->balance++;
        }
    }
    if (std::abs(node->balance) == 1) {
        return;
    }
    else if (node->balance == 0) {
        if (node == _root) {
            return;
        }
        _go_up_remove(node->parent.lock(), node);
    }
    else {
        node = _rebalance(node);
        if (node->balance == 0 || node == _root) {
            return;
        }
        _go_up_remove(node->parent.lock(), node);
    }
}
// удаление вершины дерева - конец
```

zmq_handle.h:

```
#pragma once
#include <zmq.h>
#include <zmq_utils.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "defines.h"

typedef enum {
    exec_cmd,
    hrbt_cmd,
    rebind_cmd,
    relax_cmd
} cmd_type;

typedef struct {
    char text[MAX_STRLEN];
    char pattern[MAX_STRLEN];
    int32_t text_size;
    int32_t pattern_size;
    int32_t to;
    cmd_type cmd;
    int32_t sleep_time;
    int32_t change_to;
} event;

typedef enum {
    left,
    right,
    hrbt
} socket_type;

void        create_message(zmq_msg_t* msg, event* e);
```

```
void        init_cmp_name(int pid, char* name, socket_type node_type);
void        send_to(void* socket, event* e);
void        mm_send_relax();
void        mm_send_rebind(int id, int target_id);
void        print_err_cmp(int32_t pid);
void        print_err_mas();
int32_t     init_master_socket(void** context, void** pub, void** sub);
int32_t     init_computing_socket(  void** context, void** left_pub, void** right_pub,
            void** parent_sub, void** hrbt_pub, char* socket_name_l, char*
socket_name_r, char* socket_name_h, char* parent_name, int32_t pid);

int32_t     deinit_master_socket(void* context, void* pub, void* sub);
int32_t     deinit_computing_socket(void* context, void* left_pub, void* right_pub,
                                    void* parent_sub, void* hrbt_pub, int32_t pid);
```

zmq_handle.c:

```
#include "zmq_handle.h"
static const int32_t    TIME_MS = 100;
void*                   EXEC_PUB;
void*                   HEARTBIT_SUB;

void
create_message(zmq_msg_t* msg, event* e) {
    zmq_msg_init_size(msg, sizeof(event));
    memcpy(zmq_msg_data(msg), e, sizeof(event));
}

void
init_cmp_name(int pid, char* name, socket_type node_type) {
    char pid_str[MAX_STRLEN];
    sprintf(pid_str, "%d", pid);
    if (node_type == left) {
        strcpy(name, CMP_SOCKET_PATTERN_L);
    } else if (node_type == right) {
        strcpy(name, CMP_SOCKET_PATTERN_R);
    }
    strcat(name, pid_str);
}

void
send_to(void* socket, event* e) {
    zmq_msg_t message;
    zmq_msg_init(&message);
    create_message(&message, e);
    zmq_msg_send(&message, socket, 0);
    zmq_msg_close(&message);
}

void
mm_send_rebind(int id, int target_id) {
    event sent_cmd;
    sent_cmd.to = id;
    sent_cmd.cmd = rebind_cmd;
    sent_cmd.change_to = target_id;
    zmq_msg_t zmqmsg;
    zmq_msg_init_size(&zmqmsg, sizeof(event));
    memcpy(zmq_msg_data(&zmqmsg), &sent_cmd, sizeof(event));
    int send = zmq_msg_send(&zmqmsg, EXEC_PUB, 0);
    zmq_msg_close(&zmqmsg);
}

void
mm_send_relax() {
    event sent_cmd;
    sent_cmd.cmd = relax_cmd;
    zmq_msg_t zmqmsg;
    zmq_msg_init_size(&zmqmsg, sizeof(event));
    memcpy(zmq_msg_data(&zmqmsg), &sent_cmd, sizeof(event));
    int send = zmq_msg_send(&zmqmsg, EXEC_PUB, 0);
    zmq_msg_close(&zmqmsg);
}
```

```
void
print_err_cmp(int32_t pid) {
    char msg[MAX_STRLEN];
    sprintf(msg, "ERROR, PID %d", pid);
    perror(msg);
}

void
print_err_mas() {
    perror("ERROR, MASTER: ");
}

int32_t
init_master_socket(void** context, void** pub, void** sub) {
    *context = zmq_ctx_new();
    if (*context == NULL) {
        fprintf(stderr, "MASTER: Unable to create context.\n");
        return MASTER_CTX_CREATE_ERR;
    }

    void* publisher = zmq_socket(*context, ZMQ_PUB);
    if (publisher == NULL) {
        print_err_mas();
        return MASTER_PUB_CREATE_ERR;
    }
    int32_t pc = zmq_bind(publisher, MASTER_SOCKET_PUB);
    if (pc != 0) {
        print_err_mas();
        return MASTER_PUB_BIND_ERR;
    }
    *pub = publisher;

    void* subscriber = zmq_socket(*context, ZMQ_SUB);
    if (subscriber == NULL) {
        print_err_mas();
        return MASTER_SUB_CREATE_ERR;
    }
    zmq_setsockopt(subscriber, ZMQ_SUBSCRIBE, NULL, 0);
    zmq_setsockopt(subscriber, ZMQ_CONNECT_TIMEOUT, &TIME_MS, sizeof(TIME_MS));
    pc = zmq_bind(subscriber, MASTER_SOCKET_SUB);
    if (pc != 0) {
        print_err_mas();
        return MASTER_SUB_BIND_ERR;
    }
    *sub = subscriber;
    return OK;
}

int32_t
init_computing_socket(  void** context, void** left_pub, void** right_pub, void**
parent_sub,
                        void** hrbt_pub, char* socket_name_l, char* socket_name_r,
                        char* socket_name_h, char* parent_name, int32_t pid) {
    *context = zmq_ctx_new();
    if (*context == NULL) {
        fprintf(stderr, "PID %d: Unable to create context.\n", pid);
        return CLIENT_CTX_CREATE_ERR;
    }

    void* left_publisher = zmq_socket(*context, ZMQ_PUB);
    if (left_publisher == NULL) {
        print_err_cmp(pid);
        return CLIENT_PUB_L_CREATE_ERR;
    }
    int32_t pc = zmq_bind(left_publisher, socket_name_l);
    if (pc != 0) {
        print_err_cmp(pid);
        return CLIENT_PUB_L_BIND_ERR;
    }
    *left_pub = left_publisher;
```

```
    void* right_publisher = zmq_socket(*context, ZMQ_PUB);
    if (right_publisher == NULL) {
        print_err_cmp(pid);
        return CLIENT_PUB_R_CREATE_ERR;
    }
    pc = zmq_bind(right_publisher, socket_name_r);
    if (pc != 0) {
        print_err_cmp(pid);
        return CLIENT_PUB_R_BIND_ERR;
    }
    *right_pub = right_publisher;

    void* hrbt_publisher = zmq_socket(*context, ZMQ_PUB);
    if (hrbt_publisher == NULL) {
        print_err_cmp(pid);
        return CLIENT_PUB_H_CREATE_ERR;
    }
    pc = zmq_connect(hrbt_publisher, socket_name_h);
    if (pc != 0) {
        print_err_cmp(pid);
        return CLIENT_PUB_H_CON_ERR;
    }
    *hrbt_pub = hrbt_publisher;

    void* parent_subscriber = zmq_socket(*context, ZMQ_SUB);
    if (parent_subscriber == NULL) {
        print_err_cmp(pid);
        return CLIENT_SUB_P_CREATE_ERR;
    }
    zmq_setsockopt(parent_subscriber, ZMQ_SUBSCRIBE, NULL, 0);
    zmq_setsockopt(parent_subscriber, ZMQ_CONNECT_TIMEOUT, &TIME_MS, sizeof(TIME_MS));
    pc = zmq_connect(parent_subscriber, parent_name);
    if (pc != 0) {
        print_err_cmp(pid);
        return CLIENT_SUB_P_CON_ERR;
    }
    *parent_sub = parent_subscriber;
    return OK;
}

int32_t
deinit_master_socket(void* context, void* pub, void* sub) {
    if (zmq_close(pub) != 0) {
        print_err_mas();
        return MASTER_CLOSE_PUB_ERR;
    }
    if (zmq_close(sub) != 0) {
        print_err_mas();
        return MASTER_CLOSE_SUB_ERR;
    }
    if (zmq_ctx_term(context) != 0) {
        print_err_mas();
        return MASTER_CLOSE_CTX_ERR;
    }
    return OK;
}

int32_t
deinit_computing_socket(void* context, void* left_pub, void* right_pub,
                        void* parent_sub, void* hrbt_pub, int32_t pid) {
    if (zmq_close(left_pub) != 0) {
        print_err_cmp(pid);
        return CLIENT_CLOSE_PUB_L_ERR;
    }
    if (zmq_close(right_pub) != 0) {
        print_err_cmp(pid);
        return CLIENT_CLOSE_PUB_R_ERR;
    }
    if (zmq_close(hrbt_pub) != 0) {
        print_err_cmp(pid);
        return CLIENT_CLOSE_PUB_H_ERR;
    }
```

```
    if (zmq_close(parent_sub) != 0) {
        print_err_cmp(pid);
        return CLIENT_CLOSE_SUB_P_ERR;
    }
    if (zmq_ctx_term(context) != 0) {
        print_err_cmp(pid);
        return CLIENT_CLOSE_CTX_ERR;
    }
    return OK;
}
```

client.c:

```
#include <stdio.h>
#include <unistd.h>
#include <zmq.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <limits.h>
#include "zmq_handle.h"
#include "defines.h"

int32_t         CLIENT_PID;
int32_t         PARENT_PID;
int32_t         TMP_PARENT_PID;
static char     SOCKET_NAME_L[MAX_STRLEN];
static char     SOCKET_NAME_R[MAX_STRLEN];
static char     SOCKET_NAME_H[MAX_STRLEN];
static char     PARENT_NAME[MAX_STRLEN];
static void*    CONTEXT;
static void*    LEFT_SOCKET;
static void*    RIGHT_SOCKET;
static void*    PARENT_SOCKET;
static void*    HRBT_SOCKET;

void
process_sigterm(int32_t sig) {
    printf("Computing node %d finishing it's work...\n", CLIENT_PID);
    deinit_computing_socket(CONTEXT, LEFT_SOCKET, RIGHT_SOCKET,
                            PARENT_SOCKET, HRBT_SOCKET, CLIENT_PID);
    exit(EXIT_SUCCESS);
}

void
mm_pass_rebind(int id, event* cmd) {
    if (CLIENT_PID == id) {
        TMP_PARENT_PID = cmd->change_to;
    } else {
        event sent_cmd;
        sent_cmd.cmd = rebind_cmd;
        sent_cmd.to = id;
        sent_cmd.change_to = cmd->change_to;
        zmq_msg_t zmqmsg;
        zmq_msg_init_size(&zmqmsg, sizeof(event));
        memcpy(zmq_msg_data(&zmqmsg), &sent_cmd, sizeof(event));
        if (id < CLIENT_PID) {
            zmq_msg_send(&zmqmsg, LEFT_SOCKET, 0);
        } else {
            zmq_msg_send(&zmqmsg, RIGHT_SOCKET, 0);
        }
        zmq_msg_close(&zmqmsg);
    }
}

void
mm_pass_relax() {
    {
        event sent_cmd;
```

```
        sent_cmd.cmd = relax_cmd;
        zmq_msg_t zmqmsg;
        zmq_msg_init_size(&zmqmsg, sizeof(event));
        memcpy(zmq_msg_data(&zmqmsg), &sent_cmd, sizeof(event));
        zmq_msg_send(&zmqmsg, LEFT_SOCKET, 0);
        zmq_msg_close(&zmqmsg);
    }
    {
        event sent_cmd;
        sent_cmd.cmd = relax_cmd;
        zmq_msg_t zmqmsg;
        zmq_msg_init_size(&zmqmsg, sizeof(event));
        memcpy(zmq_msg_data(&zmqmsg), &sent_cmd, sizeof(event));
        zmq_msg_send(&zmqmsg, RIGHT_SOCKET, 0);
        zmq_msg_close(&zmqmsg);
    }
    {
        if (PARENT_PID != TMP_PARENT_PID) {
            zmq_disconnect(PARENT_SOCKET, PARENT_NAME);
            if (CLIENT_PID < TMP_PARENT_PID) {
                if (TMP_PARENT_PID == -1) {
                    sprintf(PARENT_NAME, MASTER_SOCKET_PUB);
                } else {
                    sprintf(PARENT_NAME, CMP_SOCKET_PATTERN_L"%d", TMP_PARENT_PID);
                }
            } else {
                if (TMP_PARENT_PID == -1) {
                    sprintf(PARENT_NAME, MASTER_SOCKET_PUB);
                } else {
                    sprintf(PARENT_NAME, CMP_SOCKET_PATTERN_R"%d", TMP_PARENT_PID);
                }
            }
            zmq_connect(PARENT_SOCKET, PARENT_NAME);
            PARENT_PID = TMP_PARENT_PID;
        }
    }
}

int32_t
naive(char* text, char* pattern, int32_t text_size, int32_t pattern_size, int32_t*
res) {
    int32_t i = 0;
    int32_t j = 0;
    int32_t matching_c = 0;
    while (i < text_size) {
        int32_t h = i;
        while (h < text_size && j < pattern_size && text[h] == pattern[j]) {
            h++;
            j++;
        }
        if (j == pattern_size) {
            res[matching_c] = i;
            matching_c++;
        }
        j = 0;
        i++;
    }
    return matching_c;
}

void
compute(event* e) {
    int32_t res[MAX_STRLEN];
    int32_t len = naive(e->text, e->pattern, e->text_size, e->pattern_size, res);
    printf("Result is:\n");
    if (len == 0) {
        printf("There are no occurrences.\n");
    } else {
        for (int32_t i = 0; i < len; ++i) {
            printf("%d;", res[i]);
        }
        printf("\n");
```

```
    }
}

void
computing_loop() {
    while (true) {
        zmq_msg_t message;
        zmq_msg_init(&message);
        zmq_msg_init_size(&message, sizeof(event));
        zmq_msg_recv(&message, PARENT_SOCKET, 0);
        event e;
        memcpy(&e, zmq_msg_data(&message), sizeof(event));
        zmq_msg_close(&message);
        if (e.cmd == exec_cmd) {
            if (e.to != CLIENT_PID) {
                if (e.to > CLIENT_PID) {
                    send_to(RIGHT_SOCKET, &e);
                } else {
                    send_to(LEFT_SOCKET, &e);
                }
            } else {
                compute(&e);
            }
        } else if (e.cmd == hrbt_cmd) {
            send_to(LEFT_SOCKET, &e);
            send_to(RIGHT_SOCKET, &e);
            for (int32_t i = 0; i < 5; ++i) {
                event e_copy = e;
                e_copy.to = CLIENT_PID;
                send_to(HRBT_SOCKET, &e_copy);
                usleep(1e3 * e_copy.sleep_time);
            }
        } else if (e.cmd == rebind_cmd) {
            mm_pass_rebind(e.to, &e);
        } else {
            mm_pass_relax();
        }
    }
}

int
main(int argc, char* argv[]) {
    if (signal(SIGTERM, process_sigterm) == SIG_ERR) {
        perror("ERROR: ");
        return CLIENT_SIG_ERR;
    }
    CLIENT_PID = atoi(argv[1]);
    strcpy(PARENT_NAME,argv[2]);
    PARENT_PID = atoi(argv[3]);
    TMP_PARENT_PID = atoi(argv[3]);
    init_cmp_name(CLIENT_PID, SOCKET_NAME_L, left);
    init_cmp_name(CLIENT_PID, SOCKET_NAME_R, right);
    strcpy(SOCKET_NAME_H, MASTER_SOCKET_SUB);
    if (init_computing_socket(  &CONTEXT, &LEFT_SOCKET, &RIGHT_SOCKET, &PARENT_SOCKET,
                                &HRBT_SOCKET, SOCKET_NAME_L, SOCKET_NAME_R,
SOCKET_NAME_H,
                                PARENT_NAME, CLIENT_PID) != 0) {
        return CLIENT_INIT_ERR;
    }
    computing_loop();
    if (deinit_computing_socket(CONTEXT, LEFT_SOCKET, RIGHT_SOCKET,
                                PARENT_SOCKET, HRBT_SOCKET, CLIENT_PID) != 0) {
        return CLIENT_DEINIT_ERR;
    }
    return 0;
}
```

server.c:

```
#include <stdio.h>
#include <unistd.h>
#include <zmq.h>
#include <stdbool.h>
```

```
#include <stdint.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <limits.h>

#include "../avl/avl_c_externs.h"
#include "ui.h"
#include "zmq_handle.h"
#include "defines.h"

static int32_t      REAL_PID_TABLE[MAX_PID_NUM];
static avl_tree*    AVL_TREE_PTR;
static void*        CONTEXT;
extern void*        EXEC_PUB;
extern void*        HEARTBIT_SUB;

void
create_event(event* e, char* text, char* pattern, int32_t pid, cmd_type cmd, int32_t
sleep_time) {
    strcpy(e->text, text);
    strcpy(e->pattern, pattern);
    e->text_size = 0;
    for (int32_t i = 0; text[i] != '\0'; i++) {
        e->text_size++;
    }
    e->pattern_size = 0;
    for (int32_t i = 0; pattern[i] != '\0'; i++) {
        e->pattern_size++;
    }
    e->to = pid;
    e->cmd = cmd;
    e->sleep_time = sleep_time;
}

void
init_table() {
    for (int32_t i = 0; i < MAX_PID_NUM; ++i) {
        REAL_PID_TABLE[i] = VOID_PID;
    }
}

bool
remove_node(int32_t pid) {
    if (pid < 0 || pid > MAX_PID_NUM - 1) {
        printf("Invalid pid.\n");
        return false;
    }
    int32_t rpid = REAL_PID_TABLE[pid];
    if (rpid == VOID_PID) {
        printf("There is no node with this pid.\n");
        return false;
    }
    REAL_PID_TABLE[pid] = VOID_PID;
    remove_from_tree(AVL_TREE_PTR, pid);
    sleep(1);
    kill(rpid, SIGTERM);
    return true;
}

void
terminate_all_nodes() {
    for (int32_t i = 0; i < MAX_PID_NUM; ++i) {
        if (REAL_PID_TABLE[i] != VOID_PID) {
            remove_node(i);
        }
    }
}

void
```

```
process_signal(int32_t sig) {
    terminate_all_nodes();
    deinit_avl(AVL_TREE_PTR);
    deinit_master_socket(CONTEXT, EXEC_PUB, HEARTBIT_SUB);
    exit(EXIT_SUCCESS);
}

bool
create_node(int32_t pid) {
    if (pid < 0 || pid > MAX_PID_NUM - 1) {
        printf("Invalid pid.\n");
        return false;
    }
    if (REAL_PID_TABLE[pid] != VOID_PID) {
        printf("Node with this pid has already created.\n");
        return false;
    }
    char parent_name[MAX_STRLEN];
    int32_t path_len = MAX_STRLEN;
    int32_t path[MAX_STRLEN];
    add_to_tree(AVL_TREE_PTR, pid);
    get_path(AVL_TREE_PTR, pid, &path_len, path);
    int32_t parent_pid = -1;
    if (path_len == 1) {
        strcpy(parent_name, MASTER_SOCKET_PUB);
    } else {
        parent_pid = get_parent_id(AVL_TREE_PTR, pid);
        if (pid < parent_pid) {
            init_cmp_name(parent_pid, parent_name, left);
        } else {
            init_cmp_name(parent_pid, parent_name, right);
        }
    }
    int32_t fv = fork();
    if (fv < 0) {
        remove_from_tree(AVL_TREE_PTR, pid);
        printf("Unable to create node, fork err.\n");
        return false;
    } else if (fv == 0) {
        char client_id[MAX_STRLEN];
        sprintf(client_id, "%d", pid);
        char client_name[MAX_STRLEN];
        sprintf(client_name, "client_%d", pid);
        char parent_id[MAX_STRLEN];
        sprintf(parent_id, "%d", parent_pid);
        execl(CLIENT_PROG_NAME, client_name, client_id, parent_name, parent_id, NULL);
    } else {
        REAL_PID_TABLE[pid] = fv;
    }
    return true;
}

bool
send_exec(char* text, char* pattern, int32_t pid) {
    if (pid < 0 || pid > MAX_PID_NUM - 1) {
        printf("Invalid pid.\n");
        return false;
    }
    if (REAL_PID_TABLE[pid] == VOID_PID) {
        printf("There is no node with this pid.\n");
        return false;
    }
    event e;
    create_event(&e, text, pattern, pid, exec_cmd, 0);
    send_to(EXEC_PUB, &e);
    return true;
}

void
send_heartbit(int32_t input_time) {
    event e;
    create_event(&e, "", "", -1, hrbt_cmd, input_time);
```

```
    send_to(EXEC_PUB, &e);
}

void
kill_child(int32_t pid) {
    int32_t rpid = REAL_PID_TABLE[pid];
    if (rpid == VOID_PID) {
        printf("There is no node with this pid.\n");
        return;
    }
    REAL_PID_TABLE[pid] = VOID_PID;
    kill(rpid, SIGTERM);
}

void
heartbit(int32_t input_time) {
    int32_t time_to_wait = 4 * input_time;
    zmq_setsockopt(HEARTBIT_SUB, ZMQ_RCVTIMEO, &time_to_wait, sizeof(time_to_wait));
    int32_t heartbit_table[MAX_PID_NUM];
    for (int32_t i = 0; i < MAX_PID_NUM; ++i) {
        if (REAL_PID_TABLE[i] == VOID_PID) {
            heartbit_table[i] = -1;
        } else {
            heartbit_table[i] = 0;
        }
    }
    send_heartbit(input_time);
    while (true) {
        zmq_msg_t message;
        zmq_msg_init(&message);
        zmq_msg_init_size(&message, sizeof(event));
        int32_t rv = zmq_msg_recv(&message, HEARTBIT_SUB, 0);
        if (rv == -1 && errno == EAGAIN) {
            break;
        }
        event* e = (event*)zmq_msg_data(&message);
        heartbit_table[e->to]++;
        zmq_msg_close(&message);
    }
    int32_t count_of_dead = 0;
    int32_t count_of_void = 0;
    int32_t dead_table[MAX_PID_NUM];
    for (int32_t i = 0; i < MAX_PID_NUM; ++i) {
        if (heartbit_table[i] == 0) {
            printf("OK: %d\n", i);
            dead_table[count_of_dead] = i;
            count_of_dead++;
        } else if (heartbit_table[i] == -1) {
            count_of_void++;
        }
    }
    if (count_of_void == MAX_PID_NUM) {
        printf("There are no alive nodes now.\n");
    } else if (count_of_dead == 0) {
        printf("All is alive.\n");
    }
    if (count_of_dead != 0) {
        delete_subtree(AVL_TREE_PTR, dead_table, count_of_dead);
        sleep(1);
        for (int i = 0; i < count_of_dead; ++i) {
            kill_child(dead_table[i]);
        }
    }
}

void
print_table() {
    for (int32_t i = 0; i < MAX_PID_NUM; ++i) {
        if (REAL_PID_TABLE[i] != VOID_PID) {
            printf("Client id: %d, pid: %d\n", i, REAL_PID_TABLE[i]);
        }
    }
```

```
}

void
server_loop() {
    char cmd[MAX_STRLEN];
    int32_t pid = VOID_PID;
    while (true) {
        input_val iv = input_str(cmd);
        if (iv == eof) {
            printf("Server finishing it's work...\n");
            break;
        } else if (iv == bad) {
            printf("Unknown command, try help.\n");
            continue;
        }
        if (strcmp(cmd, "create") == 0) {
            iv = input_int(&pid);
            if (iv == eof) {
                printf("Server finishing it's work...\n");
                break;
            } else if (iv == bad) {
                printf("Unknown arg, try help.\n");
                continue;
            }
            if (!create_node(pid)) {
                printf("Unable to create node with pid %d\n", pid);
            } else {
                printf("OK: %d\n", REAL_PID_TABLE[pid]);
            }
        } else if (strcmp(cmd, "remove") == 0) {
            iv = input_int(&pid);
            if (iv == eof) {
                printf("Server finishing it's work...\n");
                break;
            } else if (iv == bad) {
                printf("Unknown arg, try help.\n");
                continue;
            }
            if (!remove_node(pid)) {
                printf("Unable to remove node with pid %d\n", pid);
            } else {
                printf("OK\n");
            }
        } else if (strcmp(cmd, "exec") == 0) {
            char text[MAX_STRLEN];
            char pattern[MAX_STRLEN];
            iv = input_int(&pid);
            if (iv == eof) {
                printf("Server finishing it's work...\n");
                break;
            } else if (iv == bad) {
                printf("Unknown arg, try help.\n");
                continue;
            }
            iv = input_str(text);
            if (iv == eof) {
                printf("Server finishing it's work...\n");
                break;
            } else if (iv == bad) {
                printf("Unknown arg, try help.\n");
                continue;
            }
            iv = input_str(pattern);
            if (iv == eof) {
                printf("Server finishing it's work...\n");
                break;
            } else if (iv == bad) {
                printf("Unknown arg, try help.\n");
                continue;
            }
            send_exec(text, pattern, pid);
        } else if (strcmp(cmd, "help") == 0) {
```

```
            help();
        } else if (strcmp(cmd, "heartbit") == 0) {
            int32_t input_time;
            iv = input_int(&input_time);
            if (iv == eof) {
                printf("Server finishing it's work...\n");
                break;
            } else if (iv == bad) {
                printf("Unknown arg, try help.\n");
                continue;
            }
            heartbit(input_time);
        } else if (strcmp(cmd, "print") == 0) {
            print_tree(AVL_TREE_PTR);
        } else if (strcmp(cmd, "table") == 0) {
            print_table();
        } else {
            printf("Unknown command, try help.\n");
        }
    }
    terminate_all_nodes();
}

int
main() {
    if (signal(SIGSEGV, process_signal) == SIG_ERR) {
        perror("ERROR: ");
        return SERV_SIG_ERR;
    }
    if (signal(SIGINT, process_signal) == SIG_ERR) {
        perror("ERROR: ");
        return SERV_SIG_ERR;
    }
    init_avl(&AVL_TREE_PTR);
    init_table();
    if (init_master_socket(&CONTEXT, &EXEC_PUB, &HEARTBIT_SUB) != 0) {
        return SERV_INIT_ERR;
    }
    help();
    server_loop();
    deinit_avl(AVL_TREE_PTR);
    if (deinit_master_socket(CONTEXT, EXEC_PUB, HEARTBIT_SUB) != 0) {
        return SERV_DEINIT_ERR;
    }
    return 0;
}
```

Пример работы

```
MacBook-Air-K:os_lab_0678 AK$ ./server
+--------------+-----------------------------------------+
|   command    |               description               |
|--------------+-----------------------------------------|
|--------------+-----------------------------------------|
| create <pid> |        creates computing node           |
|              |       (pids range from 0 to 63)         |
|--------------+-----------------------------------------|
| remove <pid> |         removes computing node          |
|              |        (pids range from 0 to 63)        |
|--------------+-----------------------------------------|
|  exec <pid>  |          exec computing node            |
|    <text>    |         finds pattern in text           |
|   <pattern>  |        (pids range from 0 to 63)        |
|--------------+-----------------------------------------|
```

```
|     help     |              print usage                  |
|              |       (pids range from 0 to 63)           |
|--------------+-------------------------------------------|
|     print    |         print process avl tree            |
|--------------+-------------------------------------------|
|   heartbit   |      heartbit all computing nodes         |
|    <time>    |   master will wait signal for 4*time ms   |
|--------------+-------------------------------------------|
|     table    |             print pid table               |
+--------------+-------------------------------------------+

create 1
OK: 1760
create 2
OK: 1764
create 3
OK: 1768
create 4
OK: 1772
create 5
OK: 1777
create 6
OK: 1781
create 7
OK: 1785
print
        ┌──1
    ┌──2
        └──3
──4
        ┌──5
    └──6
        └──7
heartbit 100
All is alive.
exec 1 aaba a
Result is:
0;1;3;
exec 7 aaba a
Result is:
0;1;3;
remove 4
OK
Computing node 4 finishing it's work...
print
        ┌──1
    ┌──2
        └──3
──5
    └──6
        └──7
heartbit 100
All is alive.
exec 1 aaba a
Result is:
0;1;3;
exec 7 aaba a
Result is:
0;1;3;
Server finishing it's work...
Computing node 1 finishing it's work...
Computing node 2 finishing it's work...
Computing node 3 finishing it's work...
Computing node 5 finishing it's work...
Computing node 6 finishing it's work...
Computing node 7 finishing it's work...
```

Вывод

Очереди сообщенийполезны для межпроцессного и межмашинного сообщения, ведь они являются связующим звеном между различными компонентами в ваших приложениях и обеспечивают надежный и масштабируемый интерфейс взаимодействия с другими подключенными системами и устройствами. Навык использования данной технологии очень пригодится в будущем, так как сейчас много внимания уделяется масштабируемым сервисам с большим числом вычислительных блоков или процессов, которые позволяют параллельно и независимо друг от друга выполнять рутинную работу.

Также данная лабораторная работа показала мне важность проектирования архитектуры приложения с точки зрения масштабируемости и максимальной модульности ПО для более удобной интеграции одних блоков кода в другие.