Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа №6
по курсу «Операционные системы»

Студент:	Марков А.Н.
Группа:	М80-208Б-18
Преподаватель:	Миронов Е.С.
Оценка:	
Дата:	

Москва
2020

Содержание

1. Постановка задачи.
2. Общие сведения о программе.
3. Общий метод и алгоритм решения.
4. Основные файлы программы.
5. Демонстрация работы программы.
6. Листинг программы.
7. Вывод.

Постановка задачи.

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.

Вариант 54:

- Тип топологии: 3. Дерево общего вида.
- Тип вычислительных команд: 2. Локальный целочисленный словарь.
- Тип команд доступности узлов: 1. pingall.

Общие сведения о программе.

Программа состоит из шести файлов:

- manager.cpp — исполняемый файл для управляющего узла.
- worker.cpp — исполняемый файл для вычислительного узла.
- lab_api.hpp — файл, содержащий сигнатуры функций, которые служат для взаимодействия узлов.
- lab_api.cpp — файл, содержащий реализации функций из файла lab_api.hpp.
- tree.hpp — файл, содержащий класс дерева общего вида.
- tree.cpp — файл, содержащий реализации методов класса дерева общего вида из файла tree.hpp.

Связь между узлами осуществляется с помощью сокетов и очередей сообщений ZeroMQ.

Общий метод и алгоритм решения.

В моем варианте топология подразумевает дерево общего вида, т. е. каждый узел может иметь сколько угодно дочерних узлов. Для получения доступа к нужному дочернему узлу за O(logN), где N — количество дочерних узлов, в каждом узле содержатся три std::map: для сокетов, для потоков, для портов. Для общения между узлами я использовал библиотеку ZeroMQ.

1. Управляющий узел принимает команды, обрабатывает их и пересылает их дочерним вычислительным узлам, а они передают сообщение дальше по дереву, если это необходимо. После работы вычислительных узлов, управляющий узел выводит сообщение о результате вычислений. В случае ошибок, управляющий узел выводит сообщение об этом.

2. Вычислительный узел определяет ему ли передан запрос, если не ему, то он передает его дальше одному из своих дочерних узлов. Если ему, то он выполняет некоторую работу и возвращает сообщение родительскому узлу.

3. Если узел недоступен, то по истечении попытки подключиться к узлу будет выдано сообщение о недоступности узла.

4. При удалении узла, все его дочерние узлы также удаляются.

Демонстрация работы программы.

```
oem@Alex-PC:~/Documents/OS/OS/lab06/src/build$ ./manager
c 1 -1
Ok:12263
c 2 -1
Ok:12268
c 3 -1
Ok:12274
c 4 -1
Ok:12279
c 5 3
Ok:12285
exec 5 h 10
Ok:10
pingall
Ok: -1
e
oem@Alex-PC:~/Documents/OS/OS/lab06/src/build$ ./manager
c 1 -1
Ok:12967
c 2 -1
Ok:12974
c 3 1
Ok:12979
c 4 1
Ok:12984
c 5 1
Ok:12990
c 6 3
Ok:12996
c 7 3
Ok:13005
c 8 6
Ok:13010
c 9 5
Ok:13016
print
--------------------------------
1
  3
    6
      8
```

```
   7
  4
  5
   9
2
--------------------------------
exec 8 myvar 10
Ok:10
exec 8 myvar
Ok:8: 10
exec 8 myvar 15
Ok:15
exec 8 myvar
Ok:8: 15
exec 9 myvar
Ok:9: 'myvar' not found
exec 11
Ok:9: '' not found
exec 11 myvar
Error:11: Not found
print
--------------------------------
1
  3
    6
      8
    7
  4
  5
    9
2
--------------------------------
remove 3
Ok
print
--------------------------------
1
  4
  5
    9
2
--------------------------------
pingall
```

Ok: -1
exit

Листинг программы.

Manager.cpp:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <cctype>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <zmq.hpp>
#include "tree.hpp"
#include "lab_api.hpp"

int main() {
   std::string command; //строка для команд
   TTree topology; // дерево
   int child_pid; // id дочернего потока
    zmq::context_t context(1); // 1 означает размер пула потоков для передачи сообщений
   std::map<int, zmq::socket_t> sockets; // словарь для сокетов, где
   // первый аргумент - id сокета
   // второй аргумент - сокет
   std::map<int, int> pids; // словарь для id потоков, где
   // первый аргумент - id сокета
   // второй аргумент - id потока
   std::map<int, int> ports; // словарь для портов, где
   // первый аргумент - id сокета
   // второй аргумент - порт

   while (std::cin >> command) {
      std::string res; // строка для результата
      int linger = 0; // задержка после закрытия сокета
      if (command == "create" || command == "c") {
            int new_id, parent_id; // переменные для id вычислительного и родительского узлов
         std::cin >> new_id >> parent_id;
            if (parent_id == -1 && pids.count(new_id) == 0) { // добавление вычислительного узла к управляющему, т.е. к корню
```

```
                sockets.emplace(new_id, zmq::socket_t(context, ZMQ_REQ)); // в словарь вставляется запрашивающий сокет с id new_id
            sockets.at(new_id).setsockopt(ZMQ_LINGER, &linger, sizeof(linger)); // для нового сокета устанавливается опция zmq_linger
                // для задержки после закрытия сокета, чтобы сообщения не отбрасывались в течение linger
              sockets.at(new_id).setsockopt(ZMQ_SNDTIMEO, 20); // для нового сокета устанавливается опция zmq_sndtimeo
            // для определения интервала ожидания отправки сообщения, если сообщение за этот период не будет отправлено, то вернется ошибка eagain
                int port = accept_connection(sockets.at(new_id)); // происходит привязка сокета с id new_id к некоторому порту
            child_pid = fork();
            if (child_pid == -1) {
               std::cout << "Unable to create worker node.\n";
               exit(1);
            } else if (child_pid == 0) { // дочерний процесс
               create_server(new_id, parent_id, port);
            } else {
               // заполнение словарей
               ports[new_id] = port;
               pids[new_id] = child_pid;
               //
               send_message(sockets.at(new_id), "pid");
               res = receive_message(sockets.at(new_id));
            }
         } else {
               if (topology.Search(new_id) != nullptr) { // если уже есть узел с id new_id
               std::cout << "Error: Already exists\n";
               continue;
            }
               if (topology.Search(parent_id) == nullptr) { // если нет родителя с нужным id
               std::cout << "Error: Parent not found\n";
               continue;
            }
               auto path = topology.GetPath(parent_id); // получаем вектор пар, путь до нужной вершины. Первый элемент пары - id узла, второй - направление движения по пути (либо к сыну, либо к брату)
               std::vector<int> path_sons; // вектор id узлов, из которых затем происходит движение по сыну
            for (auto it : path) {
```

```
            if (it.second == TO_SON) {
                path_sons.push_back(it.first);
            }
        }
        int id_on_path = path_sons.front();
        path_sons.erase(path_sons.begin());
        std::ostringstream msg_stream;
        msg_stream << "create " << path_sons.size();
        for (auto it : path_sons) {
            msg_stream << " " << it;
        }
        msg_stream << " " << new_id;
        send_message(sockets.at(id_on_path), msg_stream.str());
        res = receive_message(sockets.at(id_on_path));
    }
    if (res.substr(0, 2) == "Ok") {
        topology.Insert(new_id, parent_id);
    }
    std::cout << res << "\n";
} else if (command == "remove" || command == "r") {
    int remove_id;
    std::cin >> remove_id;
    if (pids.size() == 0) { // если нет узлов
        std::cout << "Error: Not found\n";
        continue;
    }
      if (pids.count(remove_id) != 0) { // если удаляемый вычислительный
узел является ребенком управляющего
        send_message(sockets.at(remove_id), "kill");
        res = receive_message(sockets.at(remove_id));
                kill(pids.at(remove_id), SIGTERM); // SIGTERM - сигнал
завершения процесса. Происходит запрос остановки работы процесса.
        pids.erase(remove_id);
        sockets.erase(remove_id);
        ports.erase(remove_id);
        if (res.substr(0, 2) == "Ok") {
            topology.Remove(remove_id);
        }
        std::cout << res << "\n";
        continue;
    }
    // иначе
```

```
            auto path = topology.GetPath(remove_id); // получаем путь до удаляемой вершины
        if (path.empty()) {
            std::cout << "Error: Not found\n";
            continue;
        }
        std::vector<int> path_sons;
        for (auto it : path) {
            if (it.second == TO_SON) {
                path_sons.push_back(it.first);
            }
        }
        int id_on_path = path_sons.front();
        path_sons.erase(path_sons.begin());
        path_sons.pop_back(); // выбрасываем из пути удаляемый элемент
        std::ostringstream msg_stream;
        msg_stream << "remove " << path_sons.size();
        for (auto it : path_sons) {
            msg_stream << " " << it;
        }
        msg_stream << " " << remove_id;
        send_message(sockets.at(id_on_path), msg_stream.str());
        res = receive_message(sockets.at(id_on_path));
        if (res.substr(0, 2) == "Ok") {
            topology.Remove(remove_id);
        }
        std::cout << res << "\n";
    } else if (command == "exec") {
        int id, value;
        std::string name, help_str, id_str, value_str;
        bool find = true;
        std::cin.ignore(256, ' ');
        std::getline(std::cin, help_str);
        int i;
        for (i = 0; i < help_str.size(); i++) { // считывание id
            if (std::isdigit(help_str[i])) {
                id_str += help_str[i];
            } else if (help_str[i] == ' ') {
                if (id_str.empty()) {
                    continue;
                } else {
                    id = std::stoi(id_str);
                    break;
```

```
            }
        } else { // no digit
            break;
        }
    }
    for (; i < help_str.size(); i++) { // считывание name
        if (std::isdigit(help_str[i]) || std::isalpha(help_str[i]) || help_str[i] == '+')
{
            name += help_str[i];
        } else if (help_str[i] == ' ') {
            if (name.empty()) {
                continue;
            }
            break;
        } else { // неподходящий символ
            break;
        }
    }
    for (; i < help_str.size(); i++) { // считывание value
        if (std::isdigit(help_str[i])) {
            value_str += help_str[i];
        } else if (help_str[i] == ' ') {
            if (value_str.empty()) {
                continue;
            }
            break;
        } else { // no digit
            break;
        }
    }
    if (!value_str.empty()) {
        value = std::stoi(value_str);
        find = false;
    }
    auto path = topology.GetPath(id);
    if (path.empty()) {
        std::cout << "Error:" << id << ": Not found\n";
        continue;
    }
    std::vector<int> path_sons;
    for (auto it : path) {
        if (it.second == TO_SON) {
            path_sons.push_back(it.first);
```

```
            }
        }
        auto next_id = path_sons.front();
        path_sons.erase(path_sons.begin());
        std::ostringstream msg_stream;
        msg_stream << "exec " << path_sons.size();
        for (auto it : path_sons) {
            msg_stream << " " << it;
        }
        if (find) { // если поиск в словаре
            msg_stream << " find " << name;
        } else { // если вствка в словарь
            msg_stream << " save " << name << " " << value;
        }
        send_message(sockets.at(next_id), msg_stream.str());
        res = receive_message(sockets.at(next_id));
        std::cout << res << "\n";
    } else if (command == "pingall") {
        if (pids.empty()) {
            std::cout << "Error: no nodes\n";
            continue;
        }
        std::string pre_result;
        for (auto it = sockets.begin(); it != sockets.end(); it++) {
            send_message(it->second, "ping");
            pre_result += receive_message(it->second);
        }
        std::istringstream result(pre_result);
        std::set<int> responded_id;
        int temp;
        while (result >> temp) {
            responded_id.insert(temp);
        }
        std::set<int> all_id = topology.GetNodes();
        std::set<int> result_set;
        auto it_resp = responded_id.begin();
        auto it_all = all_id.begin();
        for (int i = 0; i < all_id.size(); i++) {
            if (*it_resp == *it_all) {
                it_resp++;
                it_all++;
            } else {
                result_set.insert(*it_all);
```

```
                it_all++;
            }
        }
        std::cout << "Ok: ";
        if (result_set.empty()) {
            std::cout << "-1\n";
            continue;
        }
        for (auto it : result_set) {
            std::cout << it << ";";
        }
        std::cout << "\n";
    } else if (command == "print") {
        std::cout << "--------------------------------\n";
        topology.Print();
        std::cout << "--------------------------------\n";
    } else if (command == "exit" || command == "quit" || command == "e" ||
command == "q") {
        for (auto it = sockets.begin(); it != sockets.end(); it++) {
            send_message(it->second, "kill");
            receive_message(it->second);
            kill(pids.at(it->first), SIGTERM);
        }
        break;
    } else {
        std::cout << "Error: incorrect input\n";
    }
  }

  return 0;
}
```

worker.cpp:

```
#include <iostream>
#include <zmq.hpp>
#include <string>
#include <map>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <sstream>
#include <stdexcept>
```

```
#include "lab_api.hpp"

int main(int argc, char **argv) {
   /* argv[1] - id, argv[2] - parent_id, argv[3] - parent_port*/
   int id = std::stoi(argv[1]);
   int parent_id = std::stoi(argv[2]);
   int parent_port = std::stoi(argv[3]);

    zmq::context_t context(3); // у Димы написано "сыну, брату и отцу", т.е., скорее всего, для каждого узла будет свой поток обработки сообщений
   zmq::socket_t parent_socket(context, ZMQ_REP); // создаем сокет, который отправляет ответы на запросы
         parent_socket.connect(get_port_name(parent_port)); // подключаем созданный сокет к тому же порту, что и переданный по аргументу

   std::map<int, zmq::socket_t> sockets; // словарь для сокетов, где
   // первый аргумент - id сокета
   // второй аргумент - сокет
   std::map<int, int> pids; // словарь для id потоков, где
   // первый аргумент - id сокета
   // второй аргумент - id потока
   std::map<int, int> ports; // словарь для портов, где
   // первый аргумент - id сокета
   // второй аргумент - порт
   std::map<std::string, int> local_dict; // локальный целочисленный словарь

   while (true) {
      int linger = 0; // задержка для закрытия сокета
         std::string request = receive_message(parent_socket); // получение сообщение
         std::istringstream command_stream(request); // создается строчный поток для удобной обработки сообщения
      std::string command;
      command_stream >> command; // считывается команда

      if (command == "pid") {
         std::string answer = "Ok:" + std::to_string(getpid());
         send_message(parent_socket, answer);
      } else if (command == "create") {
         int size, new_id, port;
         command_stream >> size;
         std::vector<int> path(size);
         for (int i = 0; i < size; i++) {
```

```
            command_stream >> path[i];
        }
        command_stream >> new_id;
        if (path.empty()) {
            sockets.emplace(new_id, zmq::socket_t(context, ZMQ_REQ));
            sockets.at(new_id).setsockopt(ZMQ_SNDTIMEO, 20);
            sockets.at(new_id).setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
            port = accept_connection(sockets.at(new_id));
            int pid = fork();
            if (pid == -1) {
                send_message(parent_socket, "Can't fork");
                continue;
            } else if (pid == 0) {
                create_server(new_id, id, port);
            } else {
                ports[new_id] = port;
                pids[new_id] = pid;
                send_message(sockets.at(new_id), "pid");
                send_message(parent_socket, receive_message(sockets.at(new_id)));
            }
        } else {
            int next_id = path.front();
            path.erase(path.begin());
            std::ostringstream msg_stream;
            msg_stream << "create " << path.size();
            for (auto it : path) {
                msg_stream << " " << it;
            }
            msg_stream << " " << new_id;
            send_message(sockets.at(next_id), msg_stream.str());
            send_message(parent_socket, receive_message(sockets.at(next_id)));
        }
    } else if (command == "kill") {
        for (auto it = sockets.begin(); it != sockets.end(); it++) {
            send_message(it->second, "kill");
            receive_message(it->second);
            kill(pids.at(it->first), SIGTERM);
        }
        send_message(parent_socket, "Ok");
    } else if (command == "remove") {
        int size, remove_id;
        command_stream >> size;
        std::vector<int> path(size);
```

```
        for (int i = 0; i < size; i++) {
            command_stream >> path[i];
        }
        command_stream >> remove_id;
            if (path.empty()) { // если мы находимся в родительском узле для
удаляемого
            if (sockets.count(remove_id) == 0) {
                send_message(parent_socket, "Error: Not found\n");
                continue;
            }
            send_message(sockets.at(remove_id), "kill");
            std::string res = receive_message(sockets.at(remove_id));
            kill(pids.at(remove_id), SIGTERM);
            pids.erase(remove_id);
            sockets.erase(remove_id);
            ports.erase(remove_id);
            send_message(parent_socket, res);
        } else { // если мы находимся на пути к удаляемому
            auto next_id = path.front();
            path.erase(path.begin());
            std::ostringstream msg_stream;
            msg_stream << "remove " << path.size();
            for (auto it : path) {
                msg_stream << " " << it;
            }
            msg_stream << " " << remove_id;
            send_message(sockets.at(next_id), msg_stream.str());
            send_message(parent_socket, receive_message(sockets.at(next_id)));
        }
    } else if (command == "exec") {
        int size;
        command_stream >> size;
        std::vector<int> path(size);
        for (int i = 0; i < size; i++) {
            command_stream >> path[i];
        }
        if (size == 0) { // находимся в нужном узле
            std::string find_or_save, name;
            command_stream >> find_or_save >> name;
            if (find_or_save == "find") {
                auto search = local_dict.find(name);
                std::ostringstream msg_stream;
                if (search != local_dict.end()) { // если элемент найден
```

```
            msg_stream << "Ok:" << id << ": " << search->second;
        } else {
            msg_stream << "Ok:" << id << ": \'" << name << "\' not found";
        }
        send_message(parent_socket, msg_stream.str());
    } else if (find_or_save == "save") {
        int value;
        command_stream >> value;
        auto search = local_dict.find(name);
        if (search != local_dict.end()) { // если элемент найден
            search->second = value;
        } else {
            local_dict.insert(std::make_pair(name, value));
        }
        std::string result = "Ok:" + std::to_string(value);
        send_message(parent_socket, result);
    }
  } else {
    std::string find_or_save, name;
    command_stream >> find_or_save >> name;
    auto next_id = path.front();
    path.erase(path.begin());
    std::ostringstream msg_stream;
    msg_stream << "exec " << path.size();
    for (auto it : path) {
        msg_stream << " " << it;
    }
    if (find_or_save == "find") {
        msg_stream << " find " << name;
    } else if (find_or_save == "save") {
        std::string value;
        command_stream >> value;
        msg_stream << " save " << name << " " << value;
    }
    send_message(sockets.at(next_id), msg_stream.str());
    send_message(parent_socket, receive_message(sockets.at(next_id)));
  }
} else if (command == "ping") {
  std::ostringstream stream;
  for (auto it = sockets.begin(); it != sockets.end(); it++) {
    send_message(it->second, "ping");
    std::string str = receive_message(it->second);
    if (str == "Error: Node is not available") {
```

```
                continue;
            }
            stream << str;
        }
        stream << id << " ";
        send_message(parent_socket, stream.str());
    }
  }

  return 0;
}
```

tree.hpp:

```
#ifndef TREE_HPP
#define TREE_HPP 1

#include <memory> // shared_ptr
#include <utility> // pair
#include <vector>
#include <set>

enum Direction {
  TO_SON, TO_BRO
};

struct TNode {
  int id_{0};
  std::shared_ptr<TNode> son_{nullptr};
  std::shared_ptr<TNode> brother_{nullptr};

        TNode(int id, const std::shared_ptr<TNode> &son, const
std::shared_ptr<TNode> &brother) : id_{id}, son_{son}, brother_{brother} {}
    TNode(std::shared_ptr<TNode> node) : id_{node->id_}, son_{node->son_},
brother_{node->brother_} {}
};

class TTree {
public:
            using pair_of_ptr = std::pair<std::shared_ptr<TNode>,
std::shared_ptr<TNode>>;
private:
  std::shared_ptr<TNode> root_{nullptr};
```

```
    std::shared_ptr<TNode> create_node(int id);
    std::shared_ptr<TNode> depth_search(std::shared_ptr<TNode> node, int id);
    void help_get_nodes(std::set<int> &s, std::shared_ptr<TNode> node);
     std::shared_ptr<TNode> breadth_search(std::shared_ptr<TNode> node, int
id);
                pair_of_ptr     remove_help(std::shared_ptr<TNode>     iter_node,
std::shared_ptr<TNode> prev_node, int id);
    void print_help(std::shared_ptr<TNode> node, int right, int down);
     bool help_get_path(std::shared_ptr<TNode> node, std::vector<std::pair<int,
Direction>> &path, int id);
public:
    TTree() = default;
    TTree(const TTree &tree) : root_{tree.root_} {}
    TTree(TTree &&tree) : root_{std::move(tree.root_)} {}
    std::shared_ptr<TNode> Search(int id);
    bool Insert(int id, int parent_id);
    bool Remove(int id);
    std::vector<std::pair<int, Direction>> GetPath(int id);
    std::set<int> GetNodes();
    void Print();
};

#endif // TREE_HPP
```

tree.cpp:

```
#include <iostream>
#include <memory> // shared_ptr
#include <utility> // pair
#include "tree.hpp"

std::shared_ptr<TNode> TTree::create_node(int id) {
    return std::make_shared<TNode>(id, nullptr, nullptr);
}

std::shared_ptr<TNode>   TTree::depth_search(std::shared_ptr<TNode>   node,
int id) {
    std::shared_ptr<TNode> result{nullptr};
    if (node != nullptr) {
        if (id == node->id_) {
            return node;
        }
        result = depth_search(node->son_, id);
```

```
        if (result == nullptr) {
            result = depth_search(node->brother_, id);
        }
    }

    return result;
}

void TTree::help_get_nodes(std::set<int> &s, std::shared_ptr<TNode> node) {
    if (node != nullptr) {
        s.insert(node->id_);
        help_get_nodes(s, node->son_);
        help_get_nodes(s, node->brother_);
    }
}

std::shared_ptr<TNode> TTree::breadth_search(std::shared_ptr<TNode> node,
int id) {
    std::shared_ptr<TNode> result{nullptr};
    if (node != nullptr) {
        if (id == node->id_) {
            return node;
        }
        result = breadth_search(node->brother_, id);
        if (result == nullptr) {
            result = breadth_search(node->son_, id);
        }
    }

    return result;
}

TTree::pair_of_ptr TTree::remove_help(std::shared_ptr<TNode> iter_node,
std::shared_ptr<TNode> prev_node, int id) {
        TTree::pair_of_ptr result = std::make_pair<std::shared_ptr<TNode>,
std::shared_ptr<TNode>>(nullptr, nullptr);
    if (iter_node != nullptr) {
        if (id == iter_node->id_) {
            result.first = iter_node;
            result.second = prev_node;
            return result;
        }
        result = remove_help(iter_node->son_, iter_node, id);
```

```
        if (result.first == nullptr && result.second == nullptr) {
            result = remove_help(iter_node->brother_, iter_node, id);
        }
    }

    return result;
}

void TTree::print_help(std::shared_ptr<TNode> node, int right, int down) {
    if (node != nullptr) {
        for (int i = 0; i < down; i++) {
            std::cout << "\n";
        }
        for (int i = 0; i < right; i++) {
            std::cout << "   ";
        }
        std::cout << node->id_;
        print_help(node->son_, right + 1, 1);
        print_help(node->brother_, right, 1);
    }
}

bool TTree::help_get_path(std::shared_ptr<TNode> iter,
std::vector<std::pair<int, Direction>> &path, int id) {
    if (iter == nullptr) {
        return false;
    }
    if (iter->id_ == id && iter->son_ == nullptr) {
        auto p = std::make_pair(iter->id_, TO_SON);
        path.push_back(p);
        return true;
    }
    if (iter->id_ == id && iter->son_ != nullptr) {
        auto p = std::make_pair(iter->id_, TO_SON);
        path.push_back(p);
        iter = iter->son_;
        p = std::make_pair(iter->id_, TO_BRO);
        path.push_back(p);
        while (iter->brother_ != nullptr) {
            iter = iter->brother_;
            auto p = std::make_pair(iter->id_, TO_BRO);
            path.push_back(p);
        }
```

```
        return true;
    }
    auto p = std::make_pair(iter->id_, TO_SON);
    path.push_back(p);
    std::shared_ptr<TNode> iter_son = iter;
    while (iter_son->son_ != nullptr) {
        iter_son = iter_son->son_;
        if (help_get_path(iter_son, path, id)) {
            return true;
        }
    }
    path.pop_back();
    p = std::make_pair(iter->id_, TO_BRO);
    path.push_back(p);
    std::shared_ptr<TNode> iter_bro = iter;
    while (iter_bro->brother_ != nullptr) {
        iter_bro = iter_bro->brother_;
        if (help_get_path(iter_bro, path, id)) {
            return true;
        }
    }
    path.pop_back();
    return false;
}

std::shared_ptr<TNode> TTree::Search(int id) {
    return depth_search(root_, id);
}

bool TTree::Insert(int id, int parent_id) {
    std::shared_ptr<TNode> new_node = create_node(id);
    std::shared_ptr<TNode> parent_node = depth_search(root_, parent_id);

    if (root_ == nullptr) {
        root_ = new_node;
        return true;
    }

    if (root_ != nullptr && parent_id == -1) {
        std::shared_ptr<TNode> it = root_;
        while (it->brother_ != nullptr) {
            it = it->brother_;
        }
```

```
        it->brother_ = new_node;
    }

    if (parent_node == nullptr) {
        return false;
    }

    if (parent_node->son_ == nullptr) {
        parent_node->son_ = new_node;
        return true;
    }

    std::shared_ptr<TNode> iter_node = parent_node->son_;
    while (iter_node->brother_ != nullptr) {
        iter_node = iter_node->brother_;
    }
    iter_node->brother_ = new_node;

    return true;
}

bool TTree::Remove(int id) {
    TTree::pair_of_ptr need_and_prev = remove_help(root_, nullptr, id);

     if (need_and_prev.first == nullptr && need_and_prev.second == nullptr) { // id not found
        return false;
    }

    if (need_and_prev.first == root_) {
        root_ = nullptr;
        return true;
    }

          if (need_and_prev.second->son_ == need_and_prev.first) { // если предыдущий является отцом
        need_and_prev.second->son_ = need_and_prev.first->brother_;
        need_and_prev.first = nullptr;
        return true;
    }

        if (need_and_prev.second->brother_ == need_and_prev.first) { // если предыдущий является братом
```

```
        need_and_prev.second->brother_ = need_and_prev.first->brother_;
        need_and_prev.first = nullptr;
        return true;
    }
}

std::vector<std::pair<int, Direction>> TTree::GetPath(int id) {
    std::vector<std::pair<int, Direction>> result;
    help_get_path(root_, result, id);
    return result;
}

std::set<int> TTree::GetNodes() {
    std::set<int> s;
    help_get_nodes(s, root_);
    return s;
}

void TTree::Print() {
    print_help(root_, 0, 0);
    std::cout << "\n";
}
```

lab_api.hpp:

```
#ifndef LAB_API_HPP
#define LAB_API_HPP 1

#include <string>
#include "zmq.hpp"

void create_server(int id, int parent_id, int port);
bool send_message(zmq::socket_t &socket, const std::string &message_string);
std::string receive_message(zmq::socket_t &socket);
std::string get_port_name(int port);
int accept_connection(zmq::socket_t &socket);

#endif // LAB_API_HPP
```

lab_api.cpp:

```
#include "lab_api.hpp"
#include <string.h> // for strdup
```

```
#include <unistd.h> // for execv

void create_server(int id, int parent_id, int port) {
    char *arg1 = strdup((std::to_string(id)).c_str()); // выделяется память под строку и эта строка копируется в выделенную память
   char *arg2 = strdup((std::to_string(parent_id)).c_str());
   char *arg3 = strdup((std::to_string(port)).c_str());
   char *args[] = {"./worker", arg1, arg2, arg3, NULL};
   execv("./worker", args);
   free(arg1);
   free(arg2);
   free(arg3);
}

bool send_message(zmq::socket_t &socket, const std::string &message_string) {
   zmq::message_t message(message_string.size());
   memcpy(message.data(), message_string.c_str(), message_string.size());
   return socket.send(message);
}

std::string receive_message(zmq::socket_t &socket) {
   zmq::message_t message;
   bool result_recv;
   try {
      result_recv = socket.recv(&message); // true, если сообщение получено
      // false, если сообщение не получено
   } catch (...) { // false, если ошибка
      result_recv = false;
   }

   std::string message_str(static_cast<char *>(message.data()), message.size());
   if (message_str.empty() || !result_recv) {
      return "Error: Node is not available";
   }
   return message_str;
}

std::string get_port_name(int port) {
   return "tcp://127.0.0.1:" + std::to_string(port);
}

int accept_connection(zmq::socket_t &socket) {
   int port = 4040;
```

```
    while (true) {
        try {
                socket.bind(get_port_name(port)); // bind, т.к. привязывается к
долгоживущему порту
            break;
        } catch (...) {
            port++;
        }
    }

    return port;
}
```

Вывод.

В данной лабораторной работе я получил первый опыт работы с серверами сообщений, познакомился с библиотекой zeromq, которая предоставляет возможность удобной работы с сокетами и очереди сообщений.