

**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Операционные Системы»

## **Лабораторная работа № 6**

Тема: Управлении серверами сообщений,  
применение отложенных вычислений, интеграция  
программных систем друг с другом

Студент: Семин Александр  
Витальевич

Группа: М8О-206Б-20

Преподаватель: Соколов Андрей  
Алексеевич

Дата:

Оценка:

Москва, 2021

## 1. 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. 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 поддереву.

**Набор команд** – поиск подстроки в строке

**Проверка доступности узла:** ping id.

## 2. Набор тестов

*Первый тестовый набор:*

progger@asus:~/Desktop/OS\_labs/l6\$ ./server

Menu:

1. create <ID> -- create a node
2. exec <ID> <STR1> <STR2> -- search STR2 in STR1
3. ping <ID> -- check node existence
3. kill <ID> -- delete a calculation node

Please enter your command:> create 5

Ok: 7746

> create 2

Ok: 7771

> create 7

Ok: 7774

> ping 2

Ok: 1

> kill 2

Ok

> ping 2

Ok: 0

> exec 7 3 1 2 3

Ok: 7: 6

### 3. Листинг программы

#### ZMQFunctions.h

```
#pragma once
#include <bits/stdc++.h>
#include <zmq.hpp>
const int MAIN_PORT = 4040;

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

std::string receive_message(zmq::socket_t &socket) {
    zmq::message_t message;
    int chars_read;
    try {
        chars_read = (int)socket.recv(&message);
    }
    catch (...) {
        chars_read = 0;
    }
    if (chars_read == 0) {
        throw -1;
    }
    std::string received_msg(static_cast<char*>(message.data()),
message.size());
    return received_msg;
}

void connect(zmq::socket_t &socket, int port) {
    std::string address = "tcp://127.0.0.1:" + std::to_string(port);
    socket.connect(address);
}

void disconnect(zmq::socket_t &socket, int port) {
    std::string address = "tcp://127.0.0.1:" + std::to_string(port);
    socket.disconnect(address);
}

int bind(zmq::socket_t &socket, int id) {
    int port = MAIN_PORT + id;
    std::string address = "tcp://127.0.0.1:" + std::to_string(port);
    while(1){
        try{
            socket.bind(address);
            break;
        }
        catch(...){
            port++;
        }
    }
}
```

```

        return port;
    }

    void unbind(zmq::socket_t &socket, int port) {
        std::string address = "tcp://127.0.0.1:" + std::to_string(port);
        socket.unbind(address);
    }

```

## BalancedTree.h

```

#ifndef BALANCED_TREE_H
#define BALANCED_TREE_H
#include <bits/stdc++.h>
class BalancedTree {
    class BalancedTreeNode {
    public:
        int id;
        BalancedTreeNode* left;
        BalancedTreeNode* right;
        int height;
        bool available;
        BalancedTreeNode (int id) {
            this->id = id;
            available = true;
            left = NULL;
            right = NULL;
        }
        void CheckAvailability (int id) {
            if (this->id == id){
                available = false;
            }
            else {
                if (left != NULL) {
                    left->CheckAvailability(id);
                }
                if (right != NULL) {
                    right->CheckAvailability(id);
                }
            }
        }
        void Remove (int id, std::set<int> &ids) {
            if (left != NULL && left->id == id) {
                left->RecursionRemove(ids);
                ids.erase(left->id);
                delete left;
                left = NULL;
            }
            else if (right != NULL && right->id == id) {
                right->RecursionRemove(ids);
                ids.erase(right->id);
                delete right;
                right = NULL;
            }
            else {

```

```

        if (left != NULL) {
            left->Remove(id, ids);
        }
        if (right != NULL) {
            right->Remove(id, ids);
        }
    }
}

void RecursionRemove (std::set<int> &ids) {
    if (left != NULL) {
        left->RecursionRemove(ids);
        ids.erase(left->id);
        delete left;
        left = NULL;
    }
    if (right != NULL) {
        right->RecursionRemove(ids);
        ids.erase(right->id);
        delete right;
        right = NULL;
    }
}

void AddInNode (int id, int parent_id, std::set<int> &ids)
{
    if (this->id == parent_id) {
        if (left == NULL){
            left = new BalancedTreeNode(id);
        }
        else {
            right = new BalancedTreeNode(id);
        }
        ids.insert(id);
    }
    else {
        if (left != NULL) {
            left->AddInNode(id, parent_id, ids);
        }
        if (right != nullptr) {
            right->AddInNode(id, parent_id, ids);
        }
    }
}

int MinimalHeight() {
    if (left == NULL || right == NULL) {
        return 0;
    }
    int left_height = -1;
    int right_height = -1;
    if (left != NULL && left->available == true) {
        left_height = left->MinimalHeight();
    }
    if (right != NULL && right->available == true) {
        right_height = right->MinimalHeight();
    }
}

```

```

    }
    if (right_height == -1 && left_height == -1) {
        available = false;
        return -1;
    }
    else if (right_height == -1) {
        return left_height + 1;
    }
    else if (left_height == -1) {
        return right_height + 1;
    }
    else {
        return std::min(left_height, right_height) + 1;
    }
}

int IDMinimalHeight(int height, int current_height) {
    if (height < current_height) {
        return -2;
    }
    else if (height > current_height) {
        int current_id = -2;
        if (left != NULL && left->available == true) {
            current_id = left->IDMinimalHeight(height,
(current_height + 1));
        }
        if (right != NULL && right->available == true &&
current_id == -2){
            current_id = right->IDMinimalHeight(height,
(current_height + 1));
        }
        return current_id;
    }
    else {
        if (left == NULL || right == NULL){
            return id;
        }
        return -2;
    }
}

~BalancedTreeNode() {}

};

private:
    BalancedTreeNode* root;

public:
    std::set<int> ids;
    BalancedTree() {
        root = new BalancedTreeNode(-1);
    }
    bool Exist(int id) {
        if (ids.find(id) != ids.end()) {
            return true;
        }
        return false;
    }

```

```

    }
    void AvailabilityCheck(int id) {
        root->CheckAvailability(id);
    }
    int FindID() {
        int h = root->MinimalHeight();
        return root->IDMinimalHeight(h, 0);
    }
    void AddInTree(int id, int parent) {
        root->AddInNode(id, parent, ids);
    }
    void RemoveFromRoot(int idElem) {
        root->Remove(idElem, ids);
    }
    ~BalancedTree() {
        root->RecursionRemove(ids);
        delete root;
    }
};
#endif

```

## CalculationNode.h

```

#include <bits/stdc++.h>
#include "ZMQFunctions.h"
#include "unistd.h"

class CalculationNode {
private:
    zmq::context_t context;
public:
    zmq::socket_t left, right, parent;
    int id, left_id = -2, right_id = -2, parent_id;
    int left_port, right_port, parent_port;
    CalculationNode(int id, int parent_port, int parent_id):
        id(id),
        parent_port(parent_port),
        parent_id(parent_id),
        left(context, ZMQ_REQ),
        right(context, ZMQ_REQ),
        parent(context, ZMQ_REP)
    {
        if (id != -1) {
            connect(parent, parent_port);
        }
    }
    std::string create (int child_id) {
        int port;
        bool isleft = false;
        if (left_id == -2) {
            left_port = bind(left, child_id);
            left_id = child_id;
            port = left_port;
            isleft = true;
        }
    }
};

```

```

    }
    else if (right_id == -2) {
        right_port = bind(right, child_id);
        right_id = child_id;
        port = right_port;
    }
    else {
        std:: string fail = "Error: can not create the calculation
node";

        return fail;
    }
    int fork_id = fork();
    if (fork_id == 0) {
        if (execl("./client", "client", std::
to_string(child_id).c_str(), std:: to_string(port).c_str(), std::
to_string(id).c_str(), (char*)NULL) == -1) {
            std:: cout << "Error: can not run the execl-command" <<
std:: endl;

            exit(EXIT_FAILURE);
        }
    }
    else {
        std:: string child_pid;
        try {
            if (isleft) {
                left.setsockopt(ZMQ_SNDTIMEO, 3000);
                send_message(left, "pid");
                child_pid = receive_message(left);
            }
            else {
                right.setsockopt(ZMQ_SNDTIMEO, 3000);
                send_message(right, "pid");
                child_pid = receive_message(right);
            }
            return "Ok: " + child_pid;
        }
        catch (int) {
            std:: string fail = "Error: can not connect to the
child";

            return fail;
        }
    }
}

std:: string ping (int id) {
    std:: string answer = "Ok: 0";
    if (this->id == id) {
        answer = "Ok: 1";
        return answer;
    }
    else if (left_id == id) {
        std:: string message = "ping " + std:: to_string(id);
        send_message(left, message);
        try {

```



```

        message = receive_message(left);
        if (message == "Ok: 1") {
            answer = message;
        }
    }
    catch(int){}
}
else if (right_id == id) {
    std::string message = "ping " + std::to_string(id);
    send_message(right, message);
    try {
        message = receive_message(right);
        if (message == "Ok: 1") {
            answer = message;
        }
    }
    catch(int){}
}
return answer;
}

std::string sendstring (std::string string, int id) {
    std::string answer = "Error: Parent not found";
    if (left_id == -2 && right_id == -2) {
        return answer;
    }
    else if (left_id == id) {
        if (ping(left_id) == "Ok: 1") {
            send_message(left, string);
            try{
                answer = receive_message(left);
            }
            catch(int){}
        }
    }
    else if (right_id == id) {
        if (ping(right_id) == "Ok: 1") {
            send_message(right, string);
            try {
                answer = receive_message(right);
            }
            catch(int){}
        }
    }
    else {
        if (ping(left_id) == "Ok: 1") {
            std::string message = "send " + std::to_string(id) +
" " + string;

            send_message(left, message);
            try {
                message = receive_message(left);
            }
            catch(int) {
                message = "Error: Parent not found";
            }
        }
    }
}

```

```

    }
    if (message != "Error: Parent not found") {
        answer = message;
    }
}
if (ping(right_id) == "Ok: 1") {
    std::string message = "send " + std::to_string(id) +
" " + string;

    send_message(right, message);
    try {
        message = receive_message(right);
    }
    catch(int) {
        message = "Error: Parent not found";
    }
    if (message != "Error: Parent not found") {
        answer = message;
    }
}
}
return answer;
}
std::string exec (std::string string) {

    std::istringstream string_thread(string);
    std::string s1, s2;
    string_thread >> s1;
    string_thread >> s2;
    int res;
    res = s1.find(s2);

    std::string answer = "Ok: " + std::to_string(id) + ": " + std::
to_string(res);
    return answer;
}
std::string treeclear (int child) {
    if (left_id == child) {
        left_id = -2;
        unbind(left, left_port);
    }
    else {
        right_id = -2;
        unbind(right, right_port);
    }
    return "Ok";
}
std::string kill () {
    if (left_id != -2){
        if (ping(left_id) == "Ok: 1") {
            std::string message = "kill";
            send_message(left, message);
            try {
                message = receive_message(left);

```

```

        }
        catch(int){}
        unbind(left, left_port);
        left.close();
    }
}
if (right_id != -2) {
    if (ping(right_id) == "Ok: 1") {
        std::string message = "kill";
        send_message(right, message);
        try {
            message = receive_message(right);
        }
        catch (int){}
        unbind(right, right_port);
        right.close();
    }
}
return std::to_string(parent_id);
}
~CalculationNode() {}
};

```

## Server.cpp

```

#include <bits/stdc++.h>
#include "CalculationNode.h"
#include "ZMQFunctions.h"
#include "BalancedTree.h"

int main() {
    std::string command;
    CalculationNode node(-1, -1, -1);
    std::string answer;
    std::cout << "Menu:\n"
        "\t1. create <ID> -- create a node\n"
        "\t2. exec <ID> <STR1> <STR2> -- search STR2 in STR1\n"
        "\t3. ping <ID> -- check node existence\n"
        "\t3. kill <ID> -- delete a calculation node\n";
    BalancedTree tree;
    while ((std::cout << "Please enter your command:\n") && (std::cin >>
command)) {
        if (command == "create") {
            int child;
            std::cin >> child;
            if (tree.Exist(child)) {
                std::cout << "Error: Already exists" << std::endl;
            }
            else {
                while (true) {
                    int idParent = tree.FindID();
                    if (idParent == node.id) {
                        answer = node.create(child);
                        tree.AddInTree(child, idParent);
                        break;
                    }
                }
            }
        }
    }
}

```

```

        }
        else {
            std:: string message = "create " + std::
to_string(child);

            answer = node.sendstring(message, idParent);
            if (answer == "Error: Parent not found") {
                tree.AvailabilityCheck(idParent);
            }
            else {
                tree.AddInTree(child, idParent);
                break;
            }
        }
        }
        std:: cout << answer << std::endl;
    }
}
else if (command == "exec") {
    std:: string str;
    int child;
    std:: cin >> child;
    getline(std:: cin, str);
    if (!tree.Exist(child)) {
        std:: cout << "Node doesn't exist\n";
    }
    else {
        std:: string message = "exec " + str;
        answer = node.sendstring(message, child);
        std:: cout << answer << std:: endl;
    }
}
else if (command == "ping") {
    int child;
    std:: cin >> child;
    if (!tree.Exist(child)) {
        std::cout << "Ok: 0" << std:: endl;
    }
    else if (node.left_id == child || node.right_id == child) {
        answer = node.ping(child);
        std:: cout << answer << std:: endl;
    }
    else {
        std:: string message = "ping " + std:: to_string(child);
        answer = node.sendstring(message, child);
        if (answer == "Error: Parent not found") {
            answer = "Ok: 0";
        }
        std:: cout << answer << std:: endl;
    }
}
else if (command == "kill") {
    int child;
    std:: cin >> child;

```

```

        std:: string message = "kill";
        if (!tree.Exist(child)) {
            std:: cout << "Error: Parent is not existed" << std:: endl;
        }
        else {
            answer = node.sendstring(message, child);
            if (answer != "Error: Parent not found") {
                tree.RemoveFromRoot(child);
                if (child == node.left_id){
                    unbind(node.left, node.left_port);
                    node.left_id = -2;
                    answer = "Ok";
                }
                else if (child == node.right_id) {
                    node.right_id = -2;
                    unbind(node.right, node.right_port);
                    answer = "Ok";
                }
                else {
                    message = "clear " + std:: to_string(child);
                    answer = node.sendstring(message, std::
stoi(answer));
                }
                std:: cout << answer << std:: endl;
            }
        }
    }
    else {
        std:: cout << "Please enter correct command!" << std:: endl;
    }
}
node.kill();
return 0;
}

```

## Client.cpp:

```

#include <bits/stdc++.h>
#include "CalculationNode.h"
#include "ZMQFunctions.h"
#include "BalancedTree.h"

int main(int argc, char *argv[]) {
    if (argc != 4) {
        std:: cout << "Usage: 1) ./main, 2) child_id, 3) parent_port, 4)
parent_id" << std:: endl;
        exit(EXIT_FAILURE);
    }
    CalculationNode node(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));
    while(true) {
        std:: string message;
        std:: string command;
        message = receive_message(node.parent);

```

```

std:: istream request(message);
request >> command;
if (command == "pid") {
    std:: string answer = std:: to_string(getpid());
    send_message(node.parent, answer);
}
else if (command == "ping") {
    int child;
    request >> child;
    std:: string answer = node.ping(child);
    send_message(node.parent, answer);
}
else if (command == "create") {
    int child;
    request >> child;
    std:: string answer = node.create(child);
    send_message(node.parent, answer);
}
else if (command == "exec") {
    std::string str;
    getline(request, str);
    std::string answer = node.exec(str);
    send_message(node.parent, answer);
}
else if (command == "kill") {
    std:: string answer = node.kill();
    send_message(node.parent, answer);
    disconnect(node.parent, node.parent_port);
    node.parent.close();
    break;
}
}
return 0;
}

```

## Makefile:

```
files: server client
```

```
server: Server.cpp
```

```
    g++ -fsanitize=address Server.cpp -lzmq -o server -w
```

```
client: Client.cpp
```

```
    g++ -fsanitize=address Client.cpp -lzmq -o client -w
```

```
clean:
```

```
    rm -rf server client
```

#### 4. Выводы

В процессе выполнения работы я реализовал распределенную систему по асинхронной обработке запросов. В программе используется протокол передачи данных tcr, в котором общение между процессами происходит через определенные порты.

Обмен сообщений происходит посредством функций библиотеки ZMQ, а в частности, ее паттерном «Request – Reply». Это один из самых простых и прямолинейных паттернов, который своей реализацией очень напоминает pipe. Также я в процессе выполнения работы освежил знания по реализации идеально сбалансированного бинарного дерева на C++.