Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

Студент: Шубин Г. С.

Группа: М8О–308Б–19

Вариант: 37

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.

Топология – 4, команда для вычислительных узлов - exec id n k0..kn, проверка доступности узлов – heartbit ms.

**Общие сведения о программе**

Для работы с очередями используется ZMQ, программа собирается при помощи Makefile. Управляющий узел – server, вычислительные узлы – client. В программе используются следующие системные вызовы:

1. **kill** – убивает процесс с pid – первый аргумент и посылает сигнал – второй аргумент.
2. **socket.setsockopt** – устанавливает флаги для сокета.
3. **zmq::context\_t –** создает ZMQ контекст.
4. **zmq::socket\_t –** создает ZMQ сокет.
5. **zmq::message\_t** – создает ZMQ сообщение.
6. **socket.send** – отправляет ZMQ сообщение на socket.
7. **socket.bind** – принимает соединие к сокету.
8. **execv –** выполняет указанный файл.
9. **fork** – создает копию процесса.

**Общий метод и алгоритм решения**.

Создаем по сервер – исполняющий узел, дальше делаем fork, в дочернем процессе при помощи execv запускаем client, а с родителя с сервера отсылаем сообщение, внутри клиента также создаются сокеты – левый и правый и на них отправляются сообщения с родителя, а родитель получает сообщения от детей и так по всему дереву. Исполняющий узел получает сообщение выполняет команду и отправляет ответ вверх по дереву до управляющего узла.

**Листинг программы**

**Makefile:**

GPP = g++

GPPFLAGS = -std=c++11

SERVNANE = server

CLIENTNAME = client

all: server.o client.o server\_func.o tree.o

    $(GPP) $(GPPFLAGS) server.o server\_func.o tree.o -o $(SERVNANE) -lzmq

    $(GPP) $(GPPFLAGS) client.o server\_func.o -o $(CLIENTNAME) -lzmq

server.o: server\_func.h my\_tree.h

    $(GPP) $(GPPFLAGS) -c server.cpp -o server.o

client.o: server\_func.h

    $(GPP) $(GPPFLAGS) -c client.cpp -o client.o

server\_func.o: server\_func.h

    $(GPP) $(GPPFLAGS) -c server\_func.cpp -o server\_func.o

tree.o: my\_tree.h

    $(GPP) $(GPPFLAGS) -c my\_tree.cpp -o tree.o

clean:

    rm \*.o $(CLIENTNAME) $(SERVNANE)

**server.cpp:**

#include <iostream>

#include "zmq.hpp"

#include <string>

#include <vector>

#include <signal.h>

#include <sstream>

#include <algorithm>

#include "server\_func.h"

#include "my\_tree.h"

bool is\_number(std::string val)

{

    try

    {

        int tmp = std::stoi(val);

        return true;

    }

    catch(std::exception& e)

    {

        std::cout << "Error: " << e.what() << "\n";

        return false;

    }

}

void help()

{

    std::cout << "\nCOMMANDS\n";

    std::cout << "create <id>\n";

    std::cout << "exec <id> <n> <k0...kn>\n";

    std::cout << "remove <id>\n";

    std::cout << "heartbit\n";

    std::cout << "help\n";

    std::cout << "exit\n\n";

}

bool vct\_contains(std::vector<int>& vct, int val)

{

    for(size\_t i = 0; i < vct.size(); ++i)

    {

        if(vct[i] == val) { return true; }

    }

    return false;

}

void create(IdTree& ids, zmq::socket\_t& main\_socket, int& port,

            size\_t& child\_pid, int& child\_id)

{

    size\_t node\_id = 0;

    std::string str = "";

    std::string result = "";

    std::cin >> str;

    if(!is\_number(str)) { return; }

    node\_id = std::stoi(str);

    if(child\_pid == 0)

    {

        child\_pid = fork();

        if(child\_pid == -1)

        {

            std::cout << "Unable to create first worker node\n";

            child\_pid = 0;

            exit(1);

        }

        else if(child\_pid == 0)

        {

            create\_node(node\_id, port);

        }

        else

        {

            child\_id = node\_id;

            send\_message(main\_socket,"pid");

            result = recieve\_message(main\_socket);

        }

    }

    else

    {

        std::string msg\_s = "create " + std::to\_string(node\_id);

        send\_message(main\_socket, msg\_s);

        result = recieve\_message(main\_socket);

    }

    if (result.substr(0,2) == "Ok") { ids.insert(node\_id); }

    std::cout << result << "\n";

}

void remove(IdTree& ids, zmq::socket\_t& main\_socket, int& port,

            size\_t& child\_pid, int& child\_id)

{

    size\_t node\_id = 0;

    std::string str = "";

    std::cin >> str;

    if(!is\_number(str)) { return; }

    node\_id = std::stoi(str);

    if(ids.find(node\_id) == nullptr)

    {

        std::cout << "Error: id doesn't exist\n";

        return;

    }

    if(child\_pid == 0)

    {

        std::cout << "Error:Not found\n";

        return;

    }

    if(node\_id == child\_id)

    {

        kill(child\_pid, SIGTERM);

        kill(child\_pid, SIGKILL);

        child\_id = 0;

        child\_pid = 0;

        ids.erase(node\_id);

        std::cout << "Ok\n";

        return;

    }

    std::string message\_string = "remove " + std::to\_string(node\_id);

    send\_message(main\_socket, message\_string);

    std::string recieved\_message = recieve\_message(main\_socket);

    if(recieved\_message.substr(0, std::min<int>(recieved\_message.size(), 2)) == "Ok")

    {

        ids.erase(node\_id);

    }

    std::cout << recieved\_message << "\n";

}

void exec(IdTree& ids, zmq::socket\_t& main\_socket, int& port,

          size\_t& child\_pid, int& child\_id)

{

    std::string id\_str = "";

    std::string n\_str = "";

    int id = 0;

    int n = 0;

    std:: cin >> id\_str >> n\_str;

    if(!is\_number(id\_str) || !is\_number(n\_str)) { return; }

    id = std::stoi(id\_str);

    n = std::stoi(n\_str);

    if(ids.find(id) == nullptr)

    {

        std::cout << "Error: id doesn't exist\n";

        return;

    }

    if(n <= 0)

    {

        std::cout << "Count mus't be > 0\n";

        return;

    }

    std::string numb\_str = "";

    std::vector<int> numbers(n);

    for (int i = 0; i < n; ++i)

    {

        std::cin >> numb\_str;

        if(!is\_number(numb\_str)) { return; }

        numbers[i] = std::stoi(numb\_str);

    }

    std::string message\_string = "exec " + std::to\_string(id) + " ";

    message\_string += std::to\_string(n);

    for (int i = 0; i < n; ++i)

    {

        message\_string += " " + std::to\_string(numbers[i]);

    }

    send\_message(main\_socket, message\_string);

    std::string recieved\_message = recieve\_message(main\_socket);

    std::cout << recieved\_message << "\n";

}

void heartbit(IdTree& ids, zmq::socket\_t& main\_socket, int& port,

          size\_t& child\_pid, int& child\_id)

{

    int time = 0;

    std::string str\_time = "";

    bool all\_available = true;

    int rec\_id = 0;

    std::cin >>  str\_time;

    if(!is\_number(str\_time)) { return; }

    time = std::stoi(str\_time);

    if(time <= 0)

    {

        std::cout << "Error: time must be > 0\n";

        return;

    }

    std::vector<int> from\_tree = ids.get\_nodes();

    std::vector<int> recieved\_ids;

    for(size\_t i = 0; i < from\_tree.size(); ++i)

    {

        std::string msg\_tree = "heartbit " + std::to\_string(time);

        msg\_tree += " " + std::to\_string(from\_tree[i]);

        send\_message(main\_socket, msg\_tree);

        std::string recieved = recieve\_message(main\_socket);

        if(recieved != "Error: Node is not available")

        {

            try { recieved\_ids.push\_back(std::stoi(recieved)); }

            catch(...) { std::cout << "Error: stoi\n"; }

        }

    }

    for(size\_t i = 0; i < from\_tree.size(); ++i)

    {

        if(!vct\_contains(recieved\_ids, from\_tree[i]))

        {

            all\_available = false;

            std::cout << "Heartbit: node " << from\_tree[i] << " is unavailable\n";

            ids.erase(from\_tree[i]);

        }

    }

    if(all\_available) { std::cout << "Ok\n"; }

}

int main()

{

    IdTree ids;

    std::string command = "";

    size\_t child\_pid = 0;

    int child\_id = 0;

    int linger = 0;

    zmq::context\_t context(1);

    zmq::socket\_t main\_socket(context, ZMQ\_REQ);

    main\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

    main\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

    int port = bind\_socket(main\_socket);

    help();

    while(true)

    {

        try

        {

            std::cin >> command;

            if(command == "create")

            {

                create(ids, main\_socket, port, child\_pid, child\_id);

            }

            else if(command == "remove")

            {

                remove(ids, main\_socket, port, child\_pid, child\_id);

            }

            else if(command == "exec")

            {

                exec(ids, main\_socket, port, child\_pid, child\_id);

            }

            else if(command == "heartbit")

            {

                heartbit(ids, main\_socket, port, child\_pid, child\_id);

            }

            else if(command == "help")

            {

                help();

            }

            else if(command == "exit")

            {

                system("killall client");

                break;

            }

        }

        catch(...)

        {

            std::cout << "something gone wrong\n";

        }

    }

    return 0;

}

**client.cpp:**

#include <iostream>

#include "zmq.hpp"

#include <string>

#include <sstream>

#include <exception>

#include <signal.h>

#include "server\_func.h"

void rl\_create(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket,

            int& port, int& create\_id, int& id, int& pid)

{

    if(pid == -1)

    {

        send\_message(parent\_socket, "Error: Cannot fork");

        pid = 0;

    }

    else if(pid == 0)

    {

        create\_node(create\_id,port);

    }

    else

    {

        id = create\_id;

        send\_message(socket, "pid");

        send\_message(parent\_socket, recieve\_message(socket));

    }

}

void create(std::istringstream& command\_stream, zmq::socket\_t& parent\_socket,

            zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,

            int& left\_pid, int& left\_id, int& right\_pid, int& right\_id, int& id,

            int& left\_port, int& right\_port, std::string& request\_string)

{

    int create\_id;

    command\_stream >> create\_id;

    if(create\_id == id)

    {

        std::string message\_string = "Error: Already exists";

        send\_message(parent\_socket, message\_string);

    }

    else if(create\_id < id)

    {

        if(left\_pid == 0)

        {

            left\_pid = fork();

            rl\_create(parent\_socket, left\_socket, left\_port,

                   create\_id, left\_id, left\_pid);

        }

        else

        {

            send\_message(left\_socket, request\_string);

            send\_message(parent\_socket, recieve\_message(left\_socket));

        }

    }

    else

    {

        if(right\_pid == 0)

        {

            right\_pid = fork();

            rl\_create(parent\_socket, right\_socket, right\_port,

                   create\_id, right\_id, right\_pid);

        }

        else

        {

            send\_message(right\_socket, request\_string);

            send\_message(parent\_socket, recieve\_message(right\_socket));

        }

    }

}

void rl\_remove(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket,

               int& port, int& delete\_id, int& id, int& pid,

               std::string& request\_string)

{

    if(id == 0)

    {

        send\_message(parent\_socket, "Error: Not found");

    }

    else if(id == delete\_id)

    {

        send\_message(socket, "kill\_children");

        recieve\_message(socket);

        kill(pid,SIGTERM);

        kill(pid,SIGKILL);

        id = 0;

        pid = 0;

        send\_message(parent\_socket, "Ok");

    }

    else

    {

        send\_message(socket, request\_string);

        send\_message(parent\_socket, recieve\_message(socket));

    }

}

void remove(std::istringstream& command\_stream, zmq::socket\_t& parent\_socket,

            zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,

            int& left\_pid, int& left\_id, int& right\_pid, int& right\_id, int& id,

            int& left\_port, int& right\_port, std::string& request\_string)

{

    int delete\_id;

    command\_stream >> delete\_id;

    if (delete\_id < id)

    {

        rl\_remove(parent\_socket, left\_socket, left\_port,

                  delete\_id, left\_id, left\_pid, request\_string);

    }

    else

    {

        rl\_remove(parent\_socket, right\_socket, right\_port,

                  delete\_id, right\_id, right\_pid, request\_string);

    }

}

void rl\_exec(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket,

             int& id, int& pid, std::string& request\_string)

{

    if(pid == 0)

    {

        std::string recieve\_message = "Error:" + std::to\_string(id);

        recieve\_message += ": Not found";

        send\_message(parent\_socket, recieve\_message);

    }

    else

    {

        send\_message(socket, request\_string);

        send\_message(parent\_socket, recieve\_message(socket));

    }

}

void exec(std::istringstream& command\_stream, zmq::socket\_t& parent\_socket,

          zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,

          int& left\_pid, int& left\_id, int& right\_pid, int& right\_id, int& id,

          int& left\_port, int& right\_port, std::string& request\_string)

{

    int exec\_id;

    command\_stream >> exec\_id;

    if(exec\_id == id)

    {

        int n;

        command\_stream >> n;

        int sum = 0;

        for(int i = 0; i < n; ++i)

        {

            int cur\_num;

            command\_stream >> cur\_num;

            sum += cur\_num;

        }

        std::string recieve\_message = "Ok:" + std::to\_string(id) + ":";

        recieve\_message += std::to\_string(sum);

        send\_message(parent\_socket, recieve\_message);

    }

    else if(exec\_id < id)

    {

        rl\_exec(parent\_socket, left\_socket, exec\_id,

                left\_pid, request\_string);

    }

    else

    {

        rl\_exec(parent\_socket, right\_socket, exec\_id,

                right\_pid, request\_string);

    }

}

bool repl(std::string& str, std::string& from, std::string& to)

{

    size\_t start\_pos = str.find(from);

    if(start\_pos == std::string::npos)

    {

        return false;

    }

    str.replace(start\_pos, from.length(), to);

    return true;

}

void heartbit(std::istringstream& command\_stream, zmq::socket\_t& parent\_socket,

              zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket, int& left\_pid,

              int& right\_pid, int& left\_id, int& right\_id,

              int& id, std::string& request\_string)

{

    bool flag = true;

    int time = 0;

    int my\_id = 0;

    command\_stream >> time >> my\_id;

    left\_socket.setsockopt(ZMQ\_RCVTIMEO, 4\*time);

    right\_socket.setsockopt(ZMQ\_RCVTIMEO, 4\*time);

    if(my\_id == id)

    {

        send\_message(parent\_socket, std::to\_string(id));

    }

    else if(my\_id < id)

    {

        send\_message(left\_socket, request\_string);

        send\_message(parent\_socket, recieve\_message(left\_socket));

    }

    else if(my\_id > id)

    {

        send\_message(right\_socket, request\_string);

        send\_message(parent\_socket, recieve\_message(right\_socket));

    }

}

void kill\_children(zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket,

                   zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid)

{

    if(left\_pid == 0 && right\_pid == 0)

    {

        send\_message(parent\_socket, "Ok");

    }

    else

    {

        if(left\_pid != 0)

        {

            send\_message(left\_socket, "kill\_children");

            recieve\_message(left\_socket);

            kill(left\_pid,SIGTERM);

            kill(left\_pid,SIGKILL);

        }

        if(right\_pid != 0)

        {

            send\_message(right\_socket, "kill\_children");

            recieve\_message(right\_socket);

            kill(right\_pid,SIGTERM);

            kill(right\_pid,SIGKILL);

        }

        send\_message(parent\_socket, "Ok");

    }

}

int main(int argc, char\*\* argv)

{

    int id = std::stoi(argv[1]);

    int parent\_port = std::stoi(argv[2]);

    zmq::context\_t context(3);

    zmq::socket\_t parent\_socket(context, ZMQ\_REP);

    parent\_socket.connect(get\_port\_name(parent\_port));

    int left\_pid = 0;

    int right\_pid = 0;

    int left\_id = 0;

    int right\_id = 0;

    int linger = 0;

    zmq::socket\_t left\_socket(context, ZMQ\_REQ);

    zmq::socket\_t right\_socket(context, ZMQ\_REQ);

    left\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

    left\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

    right\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

    right\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

    int left\_port = bind\_socket(left\_socket);

    int right\_port = bind\_socket(right\_socket);

    while(true)

    {

        std::string request\_string = recieve\_message(parent\_socket);

        std::istringstream command\_stream(request\_string);

        std::string command;

        command\_stream >> command;

        if(command == "id")

        {

            std::string parent\_string = "Ok:" + std::to\_string(id);

            send\_message(parent\_socket, parent\_string);

        }

        else if(command == "pid")

        {

            std::string parent\_string = "Ok:" + std::to\_string(getpid());

            send\_message(parent\_socket, parent\_string);

        }

        else if(command == "create")

        {

            create(command\_stream, parent\_socket, left\_socket, right\_socket,

                   left\_pid, left\_id, right\_pid, right\_id, id, left\_port,

                   right\_port, request\_string);

        }

        else if(command == "remove")

        {

            remove(command\_stream, parent\_socket, left\_socket, right\_socket,

                   left\_pid, left\_id, right\_pid, right\_id, id, left\_port,

                   right\_port, request\_string);

        }

        else if(command == "exec")

        {

            exec(command\_stream, parent\_socket, left\_socket, right\_socket,

                 left\_pid, left\_id, right\_pid, right\_id, id, left\_port,

                 right\_port, request\_string);

        }

        else if(command == "heartbit")

        {

            heartbit(command\_stream, parent\_socket, left\_socket, right\_socket,

                     left\_pid, right\_pid, left\_id, right\_id, id, request\_string);

        }

        else if(command == "kill\_children")

        {

            kill\_children(parent\_socket, left\_socket,

                          right\_socket,  left\_pid, right\_pid);

        }

        if(parent\_port == 0) { break; }

    }

}

**my\_tree.cpp:**

#include<iostream>

#include <vector>

#include <algorithm>

#include "my\_tree.h"

int pow2(int val)

{

    return (1 << val);

}

IdTree::~IdTree()

{

    delete\_node(root);

}

void IdTree::insert(int id)

{

    root = insert(root, id);

}

void IdTree::erase(int id)

{

    root = remove(root, id);

}

TreeNode\* IdTree::find(int id)

{

    return find(root, id);

}

void IdTree::delete\_node(TreeNode\* node)

{

    if(node == nullptr) { return; }

    delete\_node(node->right);

    delete\_node(node->left);

    delete node;

}

std::vector<int> IdTree::get\_nodes() const

{

    std::vector<int> result;

    get\_nodes(root, result);

    return result;

}

void IdTree::get\_nodes(TreeNode\* node, std::vector<int>& v) const

{

    if(node == nullptr) { return; }

    get\_nodes(node->left,v);

    v.push\_back(node->data);

    get\_nodes(node->right, v);

}

TreeNode\* IdTree::find(TreeNode\* r, int id)

{

    if(r == nullptr || r->data == id) { return r; }

    if(id < r->data) { return find(r->left, id); }

    if(id > r->data) { return find(r->right, id); }

}

int IdTree::height(TreeNode\* node)

{

    int h = 0;

    if(node != nullptr)

    {

        int l\_height = height(node->left);

        int r\_height = height(node->right);

        int max\_height = std::max(l\_height, r\_height);

        h = max\_height + 1;

    }

    return h;

}

int IdTree::diff(TreeNode\* node)

{

    int l\_height = height(node->left);

    int r\_height = height(node->right);

    int b\_factor = l\_height - r\_height;

    return b\_factor;

}

TreeNode\* IdTree::rr\_rotation(TreeNode\* parent)

{

    TreeNode\* node;

    node = parent->right;

    parent->right = node->left;

    node->left = parent;

    return node;

}

TreeNode\* IdTree::ll\_rotation(TreeNode\* parent)

{

    TreeNode\* node;

    node = parent->left;

    parent->left = node->right;

    node->right = parent;

    return node;

}

TreeNode\* IdTree::lr\_rotation(TreeNode\* parent)

{

    TreeNode\* node;

    node = parent->left;

    parent->left = rr\_rotation(node);

    return ll\_rotation(parent);

}

TreeNode\* IdTree::rl\_rotation(TreeNode\* parent)

{

    TreeNode\* node;

    node = parent->right;

    parent->right = ll\_rotation(node);

    return rr\_rotation(parent);

}

TreeNode\* IdTree::balance(TreeNode\* node)

{

    int bal\_factor = diff(node);

    if(bal\_factor > 1)

    {

        if(diff(node->left) > 0) { node = ll\_rotation(node); }

        else { node = lr\_rotation(node); }

    }

    else if(bal\_factor < -1)

    {

        if(diff(node->right) > 0) { node = rl\_rotation(node); }

        else { node = rr\_rotation(node); }

    }

    return node;

}

TreeNode\* IdTree::find\_min(TreeNode\* node)

{

    if(node == nullptr) { return nullptr; }

    else if(node->left == nullptr) { return node; }

    else { return find\_min(node->left); }

}

TreeNode\* IdTree::insert(TreeNode\* root, int val)

{

    if(root == nullptr)

    {

        root = new TreeNode;

        root->data = val;

        root->left = nullptr;

        root->right = nullptr;

        return root;

    }

    else if(val < root->data)

    {

        root->left = insert(root->left, val);

        root = balance(root);

    }

    else if(val >= root->data)

    {

        root->right = insert(root->right, val);

        root = balance(root);

    }

    return root;

}

TreeNode\* IdTree::remove(TreeNode\* root\_node, int val)

{

    TreeNode\* node;

    if(root\_node == nullptr) { return nullptr; }

    else if(val < root\_node->data) { root\_node->left = remove(root\_node->left, val); }

    else if(val >root\_node->data) { root\_node->right = remove(root\_node->right, val); }

    else if(root\_node->left && root\_node->right)

    {

        node = find\_min(root\_node->right);

        root\_node->data = node->data;

        root\_node->right = remove(root\_node->right, root\_node->data);

    }

    else

    {

        node = root\_node;

        if(root\_node->left == nullptr) { root\_node = root\_node->right; }

        else if(root\_node->right == nullptr) { root\_node = root\_node->left; }

        delete node;

    }

    if(root\_node == nullptr) { return root\_node; }

    root\_node = balance(root\_node);

}

**my\_tree.h:**

#pragma once

struct TreeNode

{

    int data;

    TreeNode\* left;

    TreeNode\* right;

};

class IdTree

{

public:

    void insert(int);

    void erase(int);

    TreeNode\* find(int);

    std::vector<int> get\_nodes() const;

    ~IdTree();

private:

    TreeNode\* root = nullptr;

    int height(TreeNode\*);

    int diff(TreeNode\*);

    TreeNode\* rr\_rotation(TreeNode\*);

    TreeNode\* ll\_rotation(TreeNode\*);

    TreeNode\* lr\_rotation(TreeNode\*);

    TreeNode\* rl\_rotation(TreeNode\*);

    TreeNode\* balance(TreeNode \*);

    TreeNode\* insert(TreeNode \*, int);

    TreeNode\* remove(TreeNode\* t, int);

    TreeNode\* find(TreeNode\*, int);

    TreeNode\* find\_min(TreeNode\*);

    void get\_nodes(TreeNode\*,std::vector<int>&) const;

    void delete\_node(TreeNode\*);

};

**server\_func.cpp:**

#include "server\_func.h"

#include <iostream>

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 recieve\_message(zmq::socket\_t& socket)

{

    zmq::message\_t message;

    bool ok = false;

    try { ok = socket.recv(&message); }

    catch(...) { ok = false; }

    std::string recieved\_message(static\_cast<char\*>(message.data()), message.size());

    if(recieved\_message.empty() || !ok) { return "Error: Node is not available"; }

    return recieved\_message;

}

std::string get\_port\_name(int port) { return "tcp://127.0.0.1:" + std::to\_string(port); }

int bind\_socket(zmq::socket\_t& socket)

{

    int port = PORT;

    while (true)

    {

        try

        {

            socket.bind(get\_port\_name(port));

            break;

        } catch(...) { ++port; }

    }

    return port;

}

void create\_node(int id, int port)

{

    char\* arg0 = strdup(CLIENTNAME);

    char\* arg1 = strdup((std::to\_string(id)).c\_str());

    char\* arg2 = strdup((std::to\_string(port)).c\_str());

    char\* args[] = {arg0, arg1, arg2, NULL};

    execv(CLIENTNAME, args);

}

**server\_func.h:**

#pragma once

#include <string>

#include <zconf.h>

#include "zmq.hpp"

#ifndef PORT

#define PORT 8080

#endif

#ifndef CLIENTNAME

#define CLIENTNAME "./client"

#endif

bool send\_message(zmq::socket\_t&,const std::string&);

std::string recieve\_message(zmq::socket\_t&);

std::string get\_port\_name(int);

int bind\_socket(zmq::socket\_t&);

void create\_node(int,int);

**Пример работы**

gregory@DESKTOP-7L8SUG4:~/labsOS/lab6-8$ make

g++ -std=c++11 -c server.cpp -o server.o

g++ -std=c++11 -c client.cpp -o client.o

g++ -std=c++11 -c server\_func.cpp -o server\_func.o

g++ -std=c++11 -c my\_tree.cpp -o tree.o

g++ -std=c++11 server.o server\_func.o tree.o -o server -lzmq

g++ -std=c++11 client.o server\_func.o -o client -lzmq

gregory@DESKTOP-7L8SUG4:~/labsOS/lab6-8$ ./server

COMMANDS

create <id>

exec <id> <n> <k0...kn>

remove <id>

heartbit

help

exit

create 5

Ok:2511

create 6

Ok:2516

create 7

Ok:2523

exec 5 2 2 3

Ok:5:5

exec 6 1 2

Ok:6:2

exec 7 3 1 0 3

Ok:7:4

remove 7

Ok

exec 5 1 2

Ok:5:2

exec 6 2 2 3

Ok:6:5

heartbit 5

Ok

exit

**Вывод**

В Си, как и в большинстве ЯП есть такая структура, как сокеты, которые позволяют удобно организовывать построение и использование архитектуры клиент-сервер. Для общения в архитектуре клиент-сервер существуют очереди сообщений, при помощи них можно достаточно несложно организовать обмен информацией, однако ZMQ – имеет не самую лучшую документацию и в связке с fork и т.п. может вызывать некоторые трудности. Такие структуры, как деревья хорошо подходят для хранения информации о клиентах и сервере.