

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу
«Операционные системы»**

Студент: Люгге Т.В.
Группа: М8О-201Б-21
Вариант: 32
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Содержание

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

Репозиторий

https://github.com/MonkeDLyugge/LabOS/tree/main/lab6_8

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (№6)
- Применение отложенных вычислений (№7)
- Интеграция программных систем друг с другом (№8)

Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность. Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд: create, exec, remove, pingall.

Общие сведения о программе

Программа распределительного узла компилируется из файла `distribution.cpp`, программа вычислительного узла компилируется из файла `computing.cpp`. В программе используется библиотека для работы с потоками и мьютексами, а также сторонняя библиотека для работы с сервером сообщений ZeroMQ.

Общий метод и алгоритм решения

Распределительный узел хранит в себе двоичное дерево, элементами которого являются `async_node` – структуры, хранящие в себе `id` вычислительного узла, порт для связи с ним, поток отправляющий/принимающий запросы/ответы к узлу, очередь отправленных на узел запросов и мьютекс, обеспечивающий возможность работать с этой очередью в несколько потоков.

В функции `main` запускается бесконечный цикл обработки пользовательских запросов, при получении запроса он отправляется в очередь обработчика соответствующего узла, откуда позже будет извлечён потоком обработки и переслан требуемому узлу (обеспечение асинхронности).

Ping выполняет проверку доступности узла: если монитор сокета получает сигнал `ZMQ_EVENT_CONNECTED`, то узел считается доступным, если же вместо этого приходит `ZMQ_EVENT_CONNECT_RETRIED`, то узел считается недоступным.

Исходный код

distribution.cpp

```
#include "node.h"

int ConvertStrToNum(const std::string& commString){
    if (commString == "start"){
        return 0;
    }
    if (commString == "time"){
        return 1;
    }
    return 2;
}

async_node* find_node_exec(async_node* ptr, int id)
{
    if (ptr == nullptr)
        return nullptr;
    if (ptr->id > id)
        return find_node_exec(ptr->L, id);
    if (ptr->id < id)
        return find_node_exec(ptr->R, id);
    return ptr;
}

async_node* find_node_create(async_node* ptr, int id)
{
    if (ptr == nullptr)
        return nullptr;
    if (ptr->L == nullptr && ptr->id > id)
        return ptr;
    if (ptr->R == nullptr && ptr->id < id)
        return ptr;
    if (ptr->id > id)
        return find_node_create(ptr->L, id);
    if (ptr->id < id)
        return find_node_create(ptr->R, id);
    return nullptr;
}

bool destroy_node(async_node*& ptr, int id)
{
    if (ptr == nullptr)
        return false;
    if (ptr->id > id)
        return destroy_node(ptr->L, id);
    if (ptr->id < id)
        return destroy_node(ptr->R, id);
    ptr->active = false;
    ptr->make_query({REMOVE});
    if (ptr->L != nullptr)
        destroy_node(ptr->L, ptr->L->id);
    if (ptr->R != nullptr)
        destroy_node(ptr->R, ptr->R->id);
    ptr = nullptr;
    return true;
}

bool ping(int id)
{
    std::string port = protocol + std::to_string(id);
    std::string ping = "inproc://ping" + std::to_string(id);
    void* context = zmq_ctx_new();
    void *req = zmq_socket(context, ZMQ_REQ);

    zmq_socket_monitor(req, ping.c_str(), ZMQ_EVENT_CONNECTED | ZMQ_EVENT_CONNECT_RETRIED);
}
```

```

void *soc = zmq_socket(context, ZMQ_PAIR);
zmq_connect(soc, ping.c_str());
zmq_connect(req, port.c_str());

zmq_msg_t msg;
zmq_msg_init(&msg);
zmq_msg_recv(&msg, soc, 0);
uint8_t* data = (uint8_t*)zmq_msg_data(&msg);
uint16_t event = *(uint16_t*)(data);

zmq_close(req);
zmq_close(soc);
zmq_msg_close(&msg);
zmq_ctx_destroy(context);
return event % 2;
}

async_node* tree = nullptr;

int main()
{
    while (true)
    {
        std::string command;
        std::cin >> command;
        if (command == "create")
        {
            int id;
            std::cin >> id;
            id += MIN_PORT;
            if (tree == nullptr)
            {
                std::string id_str = std::to_string(id);
                int pid = fork();
                if (pid == 0)
                    execl("server", "server", id_str.c_str(), NULL);
                std::cout << "Ok: " << pid << '\n';
                tree = new async_node(id);
            }
            else
            {
                async_node* node = find_node_create(tree, id);

                if (node != nullptr){
                    if (!ping(node->id))
                    {
                        std::cerr << "Error:" << id - MIN_PORT << ": Parent is unavailable\n";
                        continue;
                    }
                    node->make_query({CREATE, id});
                }
                else
                    std::cerr << "Error: Already exists\n";
            }
        }

        if (command == "exec")
        {
            int id, commandNumber;
            std::string commandString;
            std::cin >> id >> commandString;
            id += MIN_PORT;
            commandNumber = ConvertStrToNum(commandString);
            std::vector<int> vectData(2);
            vectData[0] = EXEC;
            vectData[1] = commandNumber;
            async_node* node = find_node_exec(tree, id);

```

```

        if (node != nullptr){
            node->make_query(vectData);
            if (!ping(id)) {
                std::cerr << "Error:" << id - MIN_PORT << ": Node is unavailable\n";
                break;
            }
        } else {
            std::cerr << "Error:" << id - MIN_PORT << ": Not found\n";
        }
    }

    if (command == "remove")
    {
        int id;
        std::cin >> id;
        id += MIN_PORT;
        if (!ping(id))
        {
            std::cerr << "Error:" << id - MIN_PORT << ": Node is unavailable\n";
            continue;
        }
        bool state = destroy_node(tree, id);
        if (state)
            std::cout << "Ok\n";
        else
            std::cerr << "Error: Not found\n";
    }

    if (command == "ping")
    {
        int id;
        std::cin >> id;
        id += MIN_PORT;
        async_node* node = find_node_exec(tree, id);
        if (node != nullptr){
            if (ping(id)) {
                std::cout << "Ok: 1"<< '\n';
            } else {
                std::cout << "Ok: 0"<< '\n';
            }
        } else {
            std::cerr << "Error:" << id - MIN_PORT << ": Not found\n";
        }
    }
}
}

```

computing.cpp

```

#include <zmq.h>
#include <unistd.h>
#include <string.h>
#include <stack>
#include <ctime>
#include <iostream>
#include <vector>

const int CREATE = 1;
const int EXEC = 0;
const int REMOVE = -1;

int main(int argc, char* argv[])
{
    if(argc != 2) {

```

```

    std::cerr << "Not enough parameters" << std::endl;
    exit(-1);
}
int act;
int id = atoi(argv[1]);
std::string port = "tcp://*:" + std::to_string(id);
void *context = zmq_ctx_new();
void *responder = zmq_socket(context, ZMQ_REP);
act = zmq_bind(responder, port.c_str());
if (act == -1) {
    return -1;
}
std::stack<long long> stack;
while (true)
{
    zmq_msg_t msg;
    act = zmq_msg_init(&msg);
    if (act == -1) {
        return -1;
    }

    act = zmq_msg_recv(&msg, responder, 0);
    if (act == -1) {
        return -1;
    }

    int* data = (int*)zmq_msg_data(&msg);
    int t = *data;

    switch (t)
    {
        case CREATE:
        {
            int n = *(++data);
            std::string id_str = std::to_string(n);
            int pid = fork();
            if (pid == -1){
                return -1;
            }
            if (pid == 0){
                execl("server", "server", id_str.c_str(), NULL);
            } else {
                act = zmq_send(responder, &pid, sizeof(int), 0);
                if (act == -1) {
                    return -1;
                }
            }
            break;
        }

        case EXEC:
        {
            int commandNumber = *(++data);
            long long timer = 0;
            if (commandNumber == 0){
                long long start = std::time(nullptr);
                stack.push(start);
                timer = -1;
            }
            if ((commandNumber == 1) && !stack.empty()){
                long long end = std::time(nullptr);
                timer = (end - stack.top()) * 1000;
            }
            if (commandNumber == 2){
                stack.pop();
                timer = -1;
            }
        }
    }
}

```

```

        act = zmq_send(responder, &timer, sizeof(long long), 0);
        if (act == -1) {
            return -1;
        }
        break;
    }

    case REMOVE:
    {
        zmq_send(responder, &id, sizeof(int), 0);
        zmq_close(responder);
        zmq_ctx_destroy(context);
        return 0;
    }
}
act = zmq_msg_close(&msg);
if (act == -1) {
    return -1;
}
}
}
}

```

Node.h

```

#ifndef NODE_H
#define NODE_H

#include "zmq.h"
#include "string.h"
#include "unistd.h"
#include "stdlib.h"
#include "pthread.h"

#include <iostream>
#include <queue>
#include <vector>

std::string protocol = "tcp://localhost:";
const int MIN_PORT = 1024;
void* async_node_thd(void*);

const int CREATE = 1;
const int EXEC = 0;
const int REMOVE = -1;

struct async_node
{
    int id, act;
    std::string port;
    bool active;
    async_node* L;
    async_node* R;
    pthread_mutex_t mutex;
    pthread_t thd;
    std::queue <std::vector <int>> q;

    async_node(int i)
    {
        id = i;
        port = protocol + std::to_string(i);
        active = true;
        L = nullptr;
        R = nullptr;
        act = pthread_mutex_init(&mutex, NULL);
        if (act != 0){
            std::cout << "Error:" << id - MIN_PORT << ": Gateway mutex error\n";
            return;
        }
    }
}

```



```

    act = pthread_create(&thd, NULL, async_node_thd, this);
    if (act != 0){
        std::cout << "Error:" << id - MIN_PORT << ": Gateway thread error\n";
        return;
    }

    act = pthread_detach(thd);
    if (act != 0){
        std::cout << "Error:" << id << ": Gateway thread error\n";
        return;
    }
}

void make_query(std::vector <int> v)
{
    act = pthread_mutex_lock(&mutex);
    if (act != 0){
        std::cout << "Error:" << id - MIN_PORT << ": Gateway mutex lock error\n";
        active = false;
        return;
    }

    q.push(v);
    act = pthread_mutex_unlock(&mutex);
    if (act != 0){
        std::cout << "Error:" << id - MIN_PORT << ": Gateway mutex unlock error\n";
        active = false;
    }
}

~async_node()
{
    pthread_mutex_destroy(&mutex);
}
};

void* async_node_thd(void* ptr)
{
    int act;
    async_node* node = (async_node*)ptr;
    void* context = zmq_ctx_new();
    void *req = zmq_socket(context, ZMQ_REQ);
    act = zmq_connect(req, node->port.c_str());
    if (act == -1){
        std::cout << "Error: Connection with" << node->id - MIN_PORT << "\n";
    }
    while (node->active || !node->q.empty())
    {
        if (node->q.empty()) {
            continue;
        }

        act = pthread_mutex_lock(&node->mutex);
        if (act != 0) {
            std::cout << "Error:" << node->id - MIN_PORT << ": Gateway mutex lock error\n",
            node->active = false;
            break;
        }

        std::vector <int> vectData = node->q.front();
        node->q.pop();
        act = pthread_mutex_unlock(&node->mutex);
        if (act != 0){
            std::cout << "Error:" << node->id - MIN_PORT << ": Gateway mutex unlock error\n";
            node->active = false;

```

```

        break;
    }

    switch (vectData[0])
    {
        case CREATE:
        {
            zmq_msg_t msg;
            act = zmq_msg_init_size(&msg, 2 * sizeof(int));
            if (act == -1){
                std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
                break;
            }

            memcpy(zmq_msg_data(&msg), &vectData[0], 2 * sizeof(int));
            act = zmq_msg_send(&msg, req, 0);
            if (act == -1){
                std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
                break;
            }

            int pid;
            act = zmq_recv(req, &pid, sizeof(int), 0);
            if(act == -1){
                std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
                break;
            }

            if (vectData[1] < node->id){
                node->L = new async_node(vectData[1]);
            } else {
                node->R = new async_node(vectData[1]);
            }

            std::cout << "Ok: " << pid << '\n';
            zmq_msg_close(&msg);
            break;
        }

        case EXEC:
        {
            zmq_msg_t msg;
            int len = sizeof(int) * 2;
            act = zmq_msg_init_size(&msg, len);
            if (act == -1){
                std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
                break;
            }

            memcpy(zmq_msg_data(&msg), &vectData[0], len);
            act = zmq_msg_send(&msg, req, 0);
            if (act == -1){
                std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
                break;
            }

            long long ans;
            act = zmq_recv(req, &ans, sizeof(long long), 0);
            if(act == -1){
                std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
                break;
            }

            if (ans >= 0 ) {
                std::cout << "Ok:" << node->id - MIN_PORT << ':' << ans << '\n';
            } else {
                std::cout << "Ok:" << node->id - MIN_PORT << '\n';
            }
        }
    }

```

```

        zmq_msg_close(&msg);
        break;
    }

    case REMOVE:
    {
        act = zmq_send(req, &vectData[0], sizeof(int), 0);
        if (act == -1){
            std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
            break;
        }
        int ans;
        act = zmq_recv(req, &ans, sizeof(int), 0);
        if (act == -1){
            std::cout << "Error:" << node->id - MIN_PORT << ": Message error\n";
            break;
        }
        break;
    }
}

}
zmq_close(req);
zmq_ctx_destroy(context);
delete node;
return NULL;
}

#endif

```

Демонстрация работы программы

```
alex@alex-VirtualBox:~/Рабочий стол/OS-labs/build/lab6-8$ ./client
```

```
create 10
```

```
Ok: 7848
```

```
create 20
```

```
Ok: 7907
```

```
exec 10 time
```

```
Ok:10:0
```

```
exec 10 start
```

```
Ok:10
```

```
exec 30 start
```

```
Error:30: Not found
```

```
exec 10 start
```

```
Ok:10
```

```
exec 10 time
```

```
Ok:10:5000
```

```
exec 10 stop
```

```
Ok:10
```

```
ping 10
```

```
Ok: 1
```

```
exec 10
```

```
time
```

```
Ok:10:34000
```

```
11
```

exec 10 stop

Ok:10

Выводы

Составлена и отлажена программа на языке C++, осуществляющая отложенные вычисления на нескольких вычислительных узлах. Пользователь управляет программой через распределительный узел, который перенаправляет запросы в асинхронном режиме.