Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики Кафедра вычислительной математики и программирования

Лабораторная работа по предмету "операционные системы" №6

Студент: Мокеева С.А.
Преподаватель: Соколов А.А.
Группа: М8О-206Б-20
Дата: 12.04.2022
Оценка:
Подпись:

Москва 2022г.

Вариант 33.

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:

Создание нового вычислительного узла

Формат команды: create id [parent]

id – целочисленный идентификатор нового вычислительного узла

parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода:

«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла

«Error: Already exists» - вычислительный узел с таким идентификатором уже существует

«Error: Parent not found» - нет такого родительского узла с таким идентификатором

«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться

«Error: [Custom error]» - любая другая обрабатываемая ошибка

Пример:

> create 10 5

Ok: 3128

Примечания: создание нового управляющего узла осуществляется пользователем программы

при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.

Исполнение команды на вычислительном узле

Формат команды: exec id [params]

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Формат вывода:

«Ok:id: [result]», где result – результат выполненной команды

«Error:id: Not found» - вычислительный узел с таким идентификатором не найден

«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error:id: [Custom error]» - любая другая обрабатываемая ошибка

Пример: Можно найти в описании конкретной команды, определенной вариантом задания.

Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Топология

Топология 2

Все вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду:

create id -1.

Типы команд для вычислительных узлов

Набор команд 1 (подсчет суммы n чисел)

Формат команды: exec id n k1 … kn

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

n – количество складываемых чисел (от 1 до 108)

k1 … kn – складываемые числа

Пример:

> exec 10 3 1 2 3

Ok:10: 6

Тип проверки доступности узлов

Команда проверки 1

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

Пример:

> pingall

Ok: -1 // Все узлы доступны

> pingall

Ok: 7;10;15 // узлы 7, 10, 15 — недоступны

Не выбран ни один файл

Иное

Реализация

Топология

В контроллере для каждого узла храним путь к нему. Например, пусть сеть такая:
узел 1
узел 11
узел 111
узел 112
узел 12
узел 2
Тогда хранится такое:
1: 1
11: 1, 11
111: 1, 11, 111
112: 1, 11, 112
12: 1, 12
2: 2
И команды будут выполняться следующим образом.
Создание узла
Ребёнок (непосредственный)
В контроллере создаём сокет для общения с ребёнком, запускаем процесс.
Потомок
Например, уже есть узлы 1, 11, 111, создаём узел 1111. В сокет ребёнка (т.е. узла 1) отправляем команду "1 11 111 create 1111". Узел 1 находит себя в списке, видит в списке следующий узел (11) и отправляет команду ему. Узел 11 делает то же. Узел 111 видит, что следующего нет, и исполняет команду сам.
Суммирование
Аналогично: контроллер в сокет ребёнка отправляет команду с полным путём.
pingall
Через пингование каждого по отдельности, и это пингование аналогично суммированию.

Файл computer.cpp

```
#include <cassert>
#include <map>
#include <sstream>
#include <unistd.h>

#include "requester.hpp"

using namespace std;

class Computer : public Requester
{
    zmq::socket_t socket_parent; //create a socket

public:
    Computer(int id_self) :
        Requester(id_self),
```

```
        socket_parent(context, ZMQ_REP) // A socket of type ZMQ_REQ is used
by a client to send requests to and receive replies from a service.
    {
        // Например, если наш идентификатор 11, то мы знаем, что создавшим
        // нас узлом приготовлен сокет на порту 10011, и мы подключаемся к
нему.
        connect(socket_parent, id_self); //инициирует соединение на сокете
    }

    void loop()
    {
        while (true) {
            string request_string; //переменная для запросов
            try {
                request_string = receive_message(socket_parent); //если
получаем сообщение от сокета
            } catch (...) {
                continue; //продолжаем
            }
            debug("request_string: " + request_string);

            istringstream request_stream(request_string); //istringstream
copies the string that you give it

            size_t len_path; //объявляем длину пути
            request_stream >> len_path; //считываем длину пути
            vector<int> path(len_path); //объявляем вектор из элементов
пути
            int i_self = -1; //задаём i_self по умолчанию
            for (size_t i = 0; i < len_path; i++) {
                request_stream >> path[i]; //считываем путь
                if (path[i] == id_self) { //если путь - id
                    i_self = i; //длина пути i
                }
            }
            if (i_self == -1) { //если путь неверный
                send_message(socket_parent, "Error: Incorrect path");
//отправляем сообщение об ошибке
                continue;
            }
            debug("i_self: " + to_string(i_self));

            if (i_self < (int) len_path - 1) { // не последний
                // Передаём запрос дальше.
                auto& socket_child = get_socket(path[i_self + 1]);
//получаем сокет с путём большим на единицу
                send_message(socket_child, request_string);  //отправляем
сообщение
                auto response = receive_message(socket_child); //получаем
ответ

                // Пересылаем ответ родителю.
                send_message(socket_parent, response);

                continue;
            }

            // Обрабатываем сами.
            string operation;
            request_stream >> operation; //считываем операцию
```

```
            if (operation == "create") { //если операция "создать"
                int id;
                request_stream >> id; //считываем id
                int pid = create_node(id); //создаём узел
                send_message(socket_parent, "Ok: " + to_string(pid));
//отправляем сообщение родителю
            } else if (operation == "exec") { //если операция "exec"
                int n;
                request_stream >> n; //считываем количество элементов
                int sum = 0;
                int x;
                for (int i = 0; i < n; ++i) {
                    request_stream >> x; //считываем элементы
                    sum += x; //суммируем
                }
                send_message( //отправляем сообщение
                    socket_parent,
                    "Ok:" + to_string(id_self) + ": " + to_string(sum)
                );
            }
        }
    }
};

int main(int argc, char* argv[])
{
    if (argc != 2) {
        return 1;
    }
    int id_self = atoi(argv[1]);

    Computer(id_self).loop();
}
```

Файл controller.cpp

```
#include <cassert>
#include <list>
#include <map>
#include <set>
#include <sstream>
#include <stdexcept>
#include <unistd.h>
#include <vector>

#include "requester.hpp"

using namespace std;

class Controller : public Requester
{
    map<int, vector<int>> id2path; // для всех узлов

    bool is_id_known(int id)
    {
        return id2path.find(id) != id2path.end();
    }

    // Обозначения:
    // - request = path + command
    // - path = size + ids
    // - command = operation + arguments
    string send_command(int id, string command)
```

```
    {
        // Находим путь.
        auto it_path = id2path.find(id);
        assert(it_path != id2path.end());
        auto& path = it_path->second;
        assert(path.size() != 0);
        int id_child = path[0];

        // Формируем сообщение.
        string request = to_string(path.size());
        for (int id : path) {
            request += " " + to_string(id);
        }
        request += " " + command;

        // Находим сокет.
        auto& socket = get_socket(id_child);

        // Отправляем сообщение и возвращаем ответ.
        debug("send_command to " + to_string(id) + ": " + command);
        send_message(socket, request);
        return receive_message(socket);
    }

    void operation_create()
    {
        int id, id_parent;
        cin >> id >> id_parent;

        if (id2path.find(id) != id2path.end()) {
            cerr << "Error: Already exists" << endl;
            return;
        }

        // Копируем путь из родителя.
        auto& path = id2path[id] = vector<int>();
        if (id_parent != -1) {
            assert(id2path.find(id_parent) != id2path.end());
            auto& path_parent = id2path[id_parent];
            path.assign(path_parent.begin(), path_parent.end());
        }
        // Дополняем путь.
        path.push_back(id);

        if (id_parent == -1) {
            // Создаём узел сами.
            int pid = create_node(id);
            cout << "Ok: " << pid << '\n';
            return;
        }

        if (! is_id_known(id_parent)) {
            cerr << "Error: Parent not found" << endl;
            return;
        }

        // Делегируем команду родителю создаваемого узла.
        try {
            cout << send_command(id_parent, "create " + to_string(id)) <<
endl;
        } catch (...) {
```

```
        cerr << "Error: Parent is unavailable" << endl;
    }
}

void operation_exec()
{
    // Вначале читаем команду полностью.
    int id;
    int n;
    cin >> id >> n;

    string args_string = to_string(n);
    for (int i = 0; i < n; i++) {
        int arg;
        cin >> arg;
        args_string += ' ' + to_string(arg);
    }

    // Затем обрабатываем.
    if (! is_id_known(id)) {
        cerr << "Error:" << id << ": Not found" << endl;
        return;
    }

    try {
        cout << send_command(id, "exec " + args_string) << endl;
    } catch (...) {
        cerr << "Error:" << id << ": Node is unavailable" << endl;
    }
}

bool is_node_available(int id)
{
    string response;
    try {
        response = send_command(id, "exec 0");
    } catch (...) {
        return false;
    }

    return response.rfind("Ok:", 0) == 0; // начинается с этой строки
}

void operation_pingall()
{
    vector<int> ids_unavailable;
    for (const auto& pair_id_path : id2path) {
        int id = pair_id_path.first;
        if (! is_node_available(id)) {
            ids_unavailable.push_back(id);
        }
    }

    cout << "Ok: ";
    if (ids_unavailable.empty()) {
        cout << -1;
    } else {
        for (size_t i = 0; i < ids_unavailable.size(); i++) {
            if (i > 0) {
                cout << ';';
            }
```

```
                cout << ids_unavailable[i];
            }
        }
        cout << endl;
    }

public:
    Controller() : Requester(-1) {}

    void loop()
    {
        while (true) { //считывается команда
            debug("reading operation");
            string operation;
            if (! (cin >> operation)) {
                break;
            }

            if (operation == "create") {
                operation_create();
            } else if (operation == "exec") {
                operation_exec();
            } else if (operation == "pingall") {
                operation_pingall();
            } else {
                cerr << "Error: Incorrect operation" << endl;
            }
        }
        debug("exiting");
    }
};

int main()
{
    Controller().loop();
}
```

Файл requester.h

```
#include <cassert>
#include <iostream>
#include <map>
#include <unistd.h>

#include "zmq_functions.hpp"

const bool IS_DEBUG = false;

using namespace std;

class Requester //класс управляющий|вычислительный узел
{
protected: // доступ только подклассам, но не пользователям класса
    int id_self;
    zmq::context_t context; // передаётся один и тот же во все сокеты
    map<int, zmq::socket_t> id2socket; // только для детей

    Requester(int id_self) :
        id_self(id_self),
        context(1) // один IO-thread (стандартно для внешних соединений)
    {}

    void make_socket(int id) //создание сокета
```

```
    {
        debug("make_socket(" + to_string(id) + ")");

        // Создаём сокет.
        assert(id2socket.find(id) == id2socket.end());
        id2socket.insert(
            make_pair(
                id,
                move(zmq::socket_t(context, ZMQ_REQ)) // for REQuests
            )
        );

        // Настраиваем сокет.
        auto& socket = id2socket.find(id)->second;
        socket.setsockopt(ZMQ_SNDTIMEO, 1000); // timeout = 1000 ms
        bind(socket, id); // слушаем порт 10000 + id
    }

    zmq::socket_t& get_socket(int id)  //получение сокета
    {
        debug("get_socket(" + to_string(id) + ")");

        auto it_socket = id2socket.find(id);
        assert(it_socket != id2socket.end());
        return it_socket->second;
    }

    int create_node(int id) //создаём узел
    {
        debug("create_node(" + to_string(id) + ")");

        pid_t pid = fork(); //создаём дочерний процесс
        if (pid < 0) {
            perror("Can't create new process");
            return -1;
        }
        if (pid == 0) {
            execl(
                "./computer",
                "./computer", to_string(id).c_str(), NULL // argv
            );
            perror("Can't execute new process");
            return -1;
        }

        make_socket(id);
        debug("create_node(" + to_string(id) + "): done");
        return pid;
    }

    void debug(string line) {
        if (IS_DEBUG) {
            cerr << "DEBUG (node " << id_self << "): " << line << endl;
        }
    }
};
```

Файл zmq_functions.h

```
#include <iostream>
#include <zmq.hpp>

using namespace std;
```

```
const int PORT_BASE = 10000;

void send_message(zmq::socket_t& socket, const string& msg) //отправить
сообщение
{
    zmq::message_t message(msg.size());
    memcpy(message.data(), msg.c_str(), msg.size()); //Функция memcpy
копирует msg.size() байтов первого блока памяти, на который ссылается
указатель msg.c_str(),
      //во второй блок памяти, на который ссылается указатель
message.data().
    socket.send(message); //Возвращает значение true, если сообщение
успешно отправлено, значение false, если это не так
}

string receive_message(zmq::socket_t& socket) //получить сообщение
{
    zmq::message_t message;
    if (! socket.recv(&message)) { //если сокет не получил сообщение
        throw runtime_error("socket.recv returned false"); //выкинем ошибку
    }
    string received_msg(static_cast<char*>(message.data()),
message.size()); //данные в сообщении, размер сообщения
    return received_msg;
}

string id2address(int id) //новый адрес (127.0.0.1 - адрес интернет-
протокола loop-back)
                          //Использование адреса 127.0.0.1 позволяет
устанавливать соединение и передавать информацию
                                    //для программ-серверов, работающих
на том же компьютере, что и программа-клиент, независимо от конфигурации
аппаратных сетевых средств компьютер
{
    return "tcp://127.0.0.1:" + to_string(PORT_BASE + id);
}

void connect(zmq::socket_t& socket, int id) //инициирует соединение на
сокете
{
    socket.connect(id2address(id));
}

void disconnect(zmq::socket_t& socket, int id) //обрывает соединение на
сокете
{
    socket.disconnect(id2address(id));
}

void bind(zmq::socket_t& socket, int id) //create an endpoint for accepting
connections and bind it to the socket referenced by the socket argument.
{
    socket.bind(id2address(id));
}

void unbind(zmq::socket_t& socket, int id)
{
    socket.unbind(id2address(id));
}
```

Файл run.sh

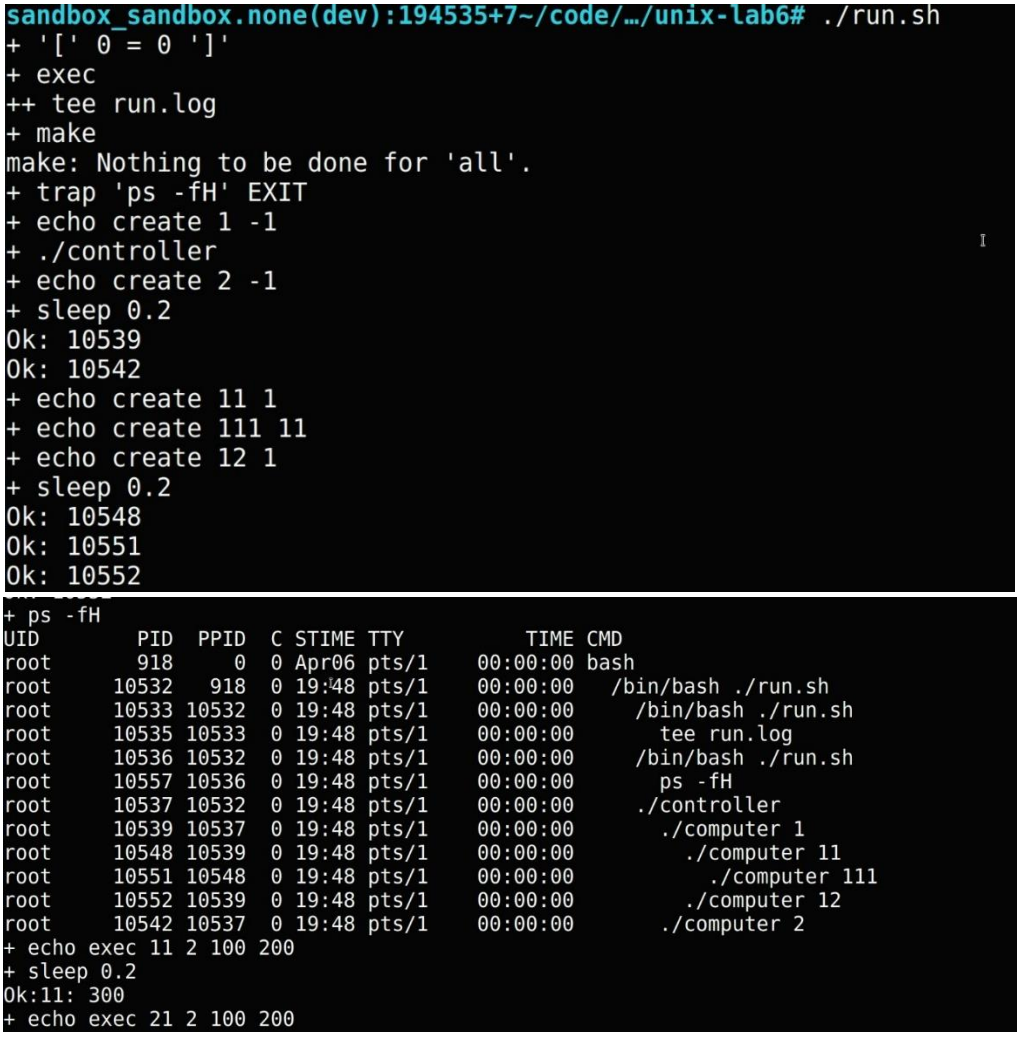
```
#!/bin/bash

set -ex -o pipefail

[ $# = 0 ]

exec &> >(tee run.log)

make

trap 'ps -fH' EXIT  # при любом выходе из скрипта

(
  echo create 1 -1
  echo create 2 -1
  sleep 0.2

  echo create 11 1
  echo create 111 11
  echo create 12 1
  sleep 0.2

  ps -fH >&2

  echo exec 11 2 100 200
  sleep 0.2

  echo exec 21 2 100 200  # должно напечататься сообщение об ошибке
  sleep 0.2

  echo pingall
  sleep 0.2

  pkill -9 computer
  ps -fH >&2

  echo pingall
  sleep 0.2

  echo exec 11 2 100 200
  sleep 0.2
) | ./controller

echo "Controller exited with code $?"
```

Файл make

```
all: \
      controller computer

deps:
      apt install libzmq3-dev

GCC = g++ -Wall -lzmq

controller: controller.cpp zmq_functions.hpp requester.hpp
      $(GCC) $< -o $@
computer: computer.cpp zmq_functions.hpp requester.hpp
      $(GCC) $< -o $@
```

```
clean:
	rm -f controller computer
```

Пример работы



Вывод

В данной лабораторной работе я научилась использовать библиотеку ZMQ, открыла для себя очень много нового. ZMQ предоставляет интерфейс для передачи сообщений, очереди сообщений – довольно удобный способ взаимодействия между процессами.