Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Операционные системы" № 5-7

Студент: Пирязев М.А.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 15.12.2022

Оценка:

Подпись:

Москва 2023 г.

Оглавление

Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (№5)
- Применение отложенных вычислений (№6)
- Интеграция программных систем друг с другом (№7)

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.

Вариант 7

Топология 3: Все вычислительные узлы хранятся в бинарном дереве поиска.
[parent] — является необязательным параметром.

Тип набора команд 4 (поиск подстроки в строке)

Формат команды:

> exec id

> text_string

> pattern_string

[result] – номера позиций, где найден образец, разделенный точкой с запятой

text_string — текст, в котором искать образец. Алфавит: [A-Za-z0-9]. Максимальная длина строки

108 символов

pattern_string — образец

Пример:

> exec 10

> abracadabra

> abra

Ok:10:0;7

> exec 10

> abracadabra

> mmm

Ok:10: -1

Команда проверки 2:

Формат команды: ping id

Команда проверяет доступность конкретного узла. Если узла нет, то необходимо выводить

ошибку: «Error: Not found»

Пример:

> ping 10

Ok: 1 // узел 10 доступен

> ping 17

Ok: 0 // узел 17 недоступен

Общие сведения о программе

Код программы состоит из основной программы (main.cpp) и дочерней программы (child.cpp), которые взаимодействуют с помощью сокетов ZeroMQ. Основная программа настраивает сокет ZMQ_REP и может отправлять команды, такие как "create", "exec", "ping", "kill" и "exit" рабочим узлам. После получения этих команд основная программа взаимодействует с рабочими узлами, отправляя и получая сообщения через сокеты ZeroMQ.

В свою очередь, дочерняя программа представляет рабочие узлы и использует сокет ZMQ_REQ для общения с основной программой. Она обрабатывает полученные от основной программы команды, и также взаимодействует с другими рабочими узлами.

В целом, основная и дочерняя программы формируют распределенную систему, где основная программа управляет рабочими узлами, отправляя команды и получая ответы, в то время как дочерняя программа представляет рабочие узлы и обрабатывает полученные команды. Общение между основной программой и рабочими узлами осуществляется через сокеты ZeroMQ.

Общий алгоритм решения

1. Основная программа (main.cpp) инициализирует ZeroMQ контекст и создает сокет типа ZMQ_REP для прослушивания команд от пользователя.

2. При получении команды "create" основная программа проверяет, есть ли свободные рабочие узлы. Если есть, то создается новый рабочий узел, иначе команда перенаправляется на уже существующий рабочий узел.

3. При получении команд "exec", "ping", "kill" основная программа отправляет соответствующие команды рабочим узлам и получает ответы.

4. При получении команды "exit" основная программа завершает работу, отправляя команду "DIE" всем рабочим узлам и закрывая сокеты и контекст ZeroMQ.

Дочерняя программа (child.cpp) представляет рабочие узлы и работает следующим образом:

1. Дочерняя программа инициализирует ZeroMQ контекст и создает сокет типа ZMQ_REQ для общения с основной программой.

2. При получении команды от основной программы, дочерняя программа выполняет соответствующие действия, например, создает новый рабочий узел, выполняет задачу, отвечает на запрос "ping" или завершает свою работу.

3. Дочерняя программа также может взаимодействовать с другими рабочими узлами, отправляя им команды и получая ответы.

Реализация

child.c

```
#include "zmq.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include <zconf.h>
#include <vector>
#include <signal.h>
#include <fstream>
#include <algorithm>
#include <map>

void send(std::string message_string, zmq::socket_t& socket) {
    zmq::message_t message_back(message_string.size());
    memcpy(message_back.data(), message_string.c_str(), message_string.size());
    if (!socket.send(message_back)) {
        std::cout << "Error: can't send message from node with pid " << getpid() <<
std::endl;
    }
}


int main(int argc, char* argv[]) {
    std::string adr = argv[1];
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    main_socket.connect(argv[1]);
    send("OK: " + std::to_string(getpid()), main_socket);
    int id = std::stoi(argv[2]); // id of this node
    std::map<std::string, int> m;
    int left_id = 0;
    int right_id = 0;
    zmq::context_t context_l(1);
    zmq::context_t context_r(1);
    zmq::socket_t left_socket(context_l, ZMQ_REP);
    std::string adr_left = "tcp://127.0.0.1:300";
    zmq::socket_t right_socket(context_r, ZMQ_REP);
    std::string adr_right = "tcp://127.0.0.1:300";
    while (1) {
        zmq::message_t message_main;
        main_socket.recv(&message_main);
        std::string recieved_message(static_cast<char*>(message_main.data()), mes-
sage_main.size());
        std::string command;
        for (int i = 0; i < recieved_message.size(); ++i) {
            if (recieved_message[i] != ' ') {
                command += recieved_message[i];
```

```
            } else {
                break;
            }
        }
        if (command == "exec") {
            int id_proc; // id of node for adding
            std::string id_proc_;
            std::string big, small, for_return;
            int flag = 0;
            std::vector<int> answers;
            for (int i = 5; i < recieved_message.size(); ++i) {
                if (recieved_message[i] != ' ') {
                    id_proc_ += recieved_message[i];
                } else {
                    break;
                }
            }
            id_proc = std::stoi(id_proc_);
            if (id_proc == id) { // id == proc_id
                for (int i = 6 + id_proc_.size(); i < recieved_message.size(); ++i) {
                    if (recieved_message[i] == ' ')
                        ++flag;
                    if ((recieved_message[i] != ' ') && (flag == 0)) {
                        big += recieved_message[i];
                    } else if ((recieved_message[i] != ' ') && (flag == 1)) {
                        small += recieved_message[i];
                    }
                }
                if (big.size() >= small.size()) {
                    int start = 0;
                    while (big.find(small, start) != -1) {
                        start = big.find(small, start);
                        answers.push_back(start);
                        ++start;
                    }
                }
                if (answers.size() == 0) {
                    for_return = "-1";
                } else {
                    for_return = std::to_string(answers[0]);
                    for (int i = 1; i < answers.size(); ++i) {
                        for_return = for_return + ";" + std::to_string(answers[i]);
                    }
                }
                for_return = "OK:" + id_proc_ + ":" + for_return;
                send(for_return, main_socket);
            } else {
                if (id > id_proc) {
                    if (left_id == 0) { // if node not exists
                        std::string message_string = "Error:id: Not found";
```

```
                    send("Error:id: Not found", main_socket);
                } else {
                    zmq::message_t message(recieved_message.size());
                    memcpy(message.data(), recieved_message.c_str(), recieved_mes-
sage.size());
                    if (!left_socket.send(message)) {
                        std::cout << "Error: can't send message to left node from
node with pid: " << getpid()
                                  << std::endl;
                    }
                    // catch and send to parent
                    if (!left_socket.recv(&message)) {
                        std::cout << "Error: can't receive message from left node in
node with pid: " << getpid()
                                  << std::endl;
                    }
                    if (!main_socket.send(message)) {
                        std::cout << "Error: can't send message to main node from
node with pid: " << getpid()
                                  << std::endl;
                    }
                }
            } else {
                if (right_id == 0) { // if node not exists
                    std::string message_string = "Error:id: Not found";
                    zmq::message_t message(message_string.size());
                    memcpy(message.data(), message_string.c_str(), mes-
sage_string.size());
                    if (!main_socket.send(message)) {
                        std::cout << "Error: can't send message to main node from
node with pid: " << getpid()
                                  << std::endl;
                    }
                } else {
                    zmq::message_t message(recieved_message.size());
                    memcpy(message.data(), recieved_message.c_str(), recieved_mes-
sage.size());
                    if (!right_socket.send(message)) {
                        std::cout << "Error: can't send message to right node from
node with pid: " << getpid()
                                  << std::endl;
                    }
                    // catch and send to parent
                    if (!right_socket.recv(&message)) {
                        std::cout << "Error: can't receive message from left node in
node with pid: " << getpid()
                                  << std::endl;
                    }
                    if (!main_socket.send(message)) {
```

```
                            std::cout << "Error: can't send message to main node from
node with pid: " << getpid()
                                      << std::endl;
                        }
                    }
                }
            }
        } else if (command == "create") {
            int id_proc; // id of node for creating
            std::string id_proc_;
            for (int i = 7; i < recieved_message.size(); ++i) {
                if (recieved_message[i] != ' ') {
                    id_proc_ += recieved_message[i];
                } else {
                    break;
                }
            }
            id_proc = std::stoi(id_proc_);
            if (id_proc == id) {
                send("Error: Already exists", main_socket);
            } else if (id_proc > id) {
                if (right_id == 0) { // there is not right node
                    right_id = id_proc;
                    int right_id_tmp = right_id - 1;
                    right_socket.bind(adr_right + std::to_string(++right_id_tmp));
                    adr_right += std::to_string(right_id_tmp);
                    char* adr_right_ = new char[adr_right.size() + 1];
                    memcpy(adr_right_, adr_right.c_str(), adr_right.size() + 1);
                    char* right_id_ = new char[std::to_string(right_id).size() + 1];
                    memcpy(right_id_, std::to_string(right_id).c_str(),
std::to_string(right_id).size() + 1);
                    char* args[] = {"./child", adr_right_, right_id_, NULL};
                    int f = fork();
                    if (f == 0) {
                        execv("./child", args);
                    } else if (f == -1) {
                        std::cout << "Error in forking in node with pid: " << getpid()
<< std::endl;
                    } else {
                        // catch message from new node
                        zmq::message_t message_from_node;
                        if (!right_socket.recv(&message_from_node)) {
                            std::cout << "Error: can't receive message from right node
in node with pid:" << getpid()
                                      << std::endl;
                        }
                        std::string recieved_message_from_node(static_cast<char*>(mes-
sage_from_node.data()),
                                                               mes-
sage_from_node.size());
```

```
                        // send message to main node
                        if (!main_socket.send(message_from_node)) {
                            std::cout << "Error: can't send message to main node from
node with pid:" << getpid()
                                      << std::endl;
                        }
                    }
                    delete[] adr_right_;
                    delete[] right_id_;
                } else { // send task to right node
                    send(recieved_message, right_socket);
                    // catch and send to parent
                    zmq::message_t message;
                    if (!right_socket.recv(&message)) {
                        std::cout << "Error: can't receive message from left node in
node with pid: " << getpid()
                                  << std::endl;
                    }
                    if (!main_socket.send(message)) {
                        std::cout << "Error: can't send message to main node from node
with pid: " << getpid()
                                  << std::endl;
                    }
                }
            } else {
                if (left_id == 0) { // there is not left node
                    left_id = id_proc;
                    int left_id_tmp = left_id - 1;
                    left_socket.bind(adr_left + std::to_string(++left_id_tmp));
                    adr_left += std::to_string(left_id_tmp);
                    char* adr_left_ = new char[adr_left.size() + 1];
                    memcpy(adr_left_, adr_left.c_str(), adr_left.size() + 1);
                    char* left_id_ = new char[std::to_string(left_id).size() + 1];
                    memcpy(left_id_, std::to_string(left_id).c_str(),
std::to_string(left_id).size() + 1);
                    char* args[] = {"./child", adr_left_, left_id_, NULL};
                    int f = fork();
                    if (f == 0) {
                        execv("./child", args);
                    } else if (f == -1) {
                        std::cout << "Error in forking in node with pid: " << getpid()
<< std::endl;
                    } else {
                        // catch message from new node
                        zmq::message_t message_from_node;
                        if (!left_socket.recv(&message_from_node)) {
                            std::cout << "Error: can't receive message from left node in
node with pid:" << getpid()
                                      << std::endl;
                        }
```

```
                        std::string recieved_message_from_node(static_cast<char*>(mes-
sage_from_node.data()),
                                                               mes-
sage_from_node.size());
                        // send message to main node
                        if (!main_socket.send(message_from_node)) {
                            std::cout << "Error: can't send message to main node from
node with pid:" << getpid()
                                      << std::endl;
                        }
                    }
                    delete[] adr_left_;
                    delete[] left_id_;
                } else { // send task to left node
                    send(recieved_message, left_socket);
                    // catch and send to parent
                    zmq::message_t message;
                    if (!left_socket.recv(&message)) {
                        std::cout << "Error: can't receive message from left node in
node with pid: " << getpid()
                                  << std::endl;
                    }
                    if (!main_socket.send(message)) {
                        std::cout << "Error: can't send message to main node from node
with pid: " << getpid()
                                  << std::endl;
                    }
                }
            }
        } else if (command == "ping") {
            int id_proc; // id of node for creating
            std::string id_proc_;
            for (int i = 5; i < recieved_message.size(); ++i) {
                if (recieved_message[i] != ' ') {
                    id_proc_ += recieved_message[i];
                } else {
                    break;
                }
            }
            id_proc = std::stoi(id_proc_);
            if (id_proc == id) {
                send("OK: 1", main_socket);
            } else if (id_proc < id) {
                if (left_id == 0) {
                    send("OK: 0", main_socket);
                } else {
                    left_socket.send(message_main);
                    zmq::message_t answ;
                    left_socket.recv(&answ);
                    main_socket.send(answ);
```

```
            }
        } else if (id_proc > id) {
            if (right_id == 0) {
                send("OK: 0", main_socket);
            } else {
                right_socket.send(message_main);
                zmq::message_t answ;
                right_socket.recv(&answ);
                main_socket.send(answ);
            }
        }
    } else if (command == "kill") {
        int id_proc; // id of node for killing
        std::string id_proc_;
        for (int i = 5; i < recieved_message.size(); ++i) {
            if (recieved_message[i] != ' ') {
                id_proc_ += recieved_message[i];
            } else {
                break;
            }
        }
        id_proc = std::stoi(id_proc_);
        if (id_proc > id) {
            if (right_id == 0) {
                send("Error: there isn`t node with this id", main_socket);
            } else {
                if (right_id == id_proc) {
                    send("Ok: " + std::to_string(right_id), main_socket);
                    send("DIE", right_socket);
                    right_socket.unbind(adr_right);
                    adr_right = "tcp://127.0.0.1:300";
                    right_id = 0;
                } else {
                    right_socket.send(message_main);
                    zmq::message_t message;
                    right_socket.recv(&message);
                    main_socket.send(message);
                }
            }
        } else if (id_proc < id) {
            if (left_id == 0) {
                send("Error: there isn`t node with this id", main_socket);
            } else {
                if (left_id == id_proc) {
                    send("Ok: " + std::to_string(left_id), main_socket);
                    send("DIE", left_socket);
                    left_socket.unbind(adr_left);
                    adr_left = "tcp://127.0.0.1:300";
                    left_id = 0;
                } else {
```

```
                        left_socket.send(message_main);
                        zmq::message_t message;
                        left_socket.recv(&message);
                        main_socket.send(message);
                    }
                }
            }
        } else if (command == "DIE") {
            if (left_id) {
                send("DIE", left_socket);
                left_socket.unbind(adr_left);
                adr_left = "tcp://127.0.0.1:300";
                left_id = 0;
            }
            if (right_id) {
                send("DIE", right_socket);
                right_socket.unbind(adr_right);
                adr_right = "tcp://127.0.0.1:300";
                right_id = 0;
            }
            main_socket.unbind(adr);
            return 0;
        }
    }
}
```

---

main.c

```
#include "zmq.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include <zconf.h>
#include <vector>
#include <signal.h>
#include <sstream>
#include <set>
#include <algorithm>

int main() {
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REP);
    std::string adr = "tcp://127.0.0.1:300";
    std::string command;
    int child_id = 0;
    while (1) {
        std::cout << "command:";
```

```
        std::cin >> command;
        if (command == "create") {
            if (child_id == 0) {
                int id;
                std::cin >> id;
                int id_tmp = id - 1;
                main_socket.bind(adr + std::to_string(++id_tmp));
                std::string new_adr = adr + std::to_string(id_tmp);
                char* adr_ = new char[new_adr.size() + 1];
                memcpy(adr_, new_adr.c_str(), new_adr.size() + 1);
                char* id_ = new char[std::to_string(id).size() + 1];
                memcpy(id_, std::to_string(id).c_str(), std::to_string(id).size() + 1);
                char* args[] = {"./child", adr_, id_, NULL};
                int id2 = fork();
                if (id2 == -1) {
                    std::cout << "Unable to create first worker node" << std::endl;
                    id = 0;
                    exit(1);
                } else if (id2 == 0) {
                    execv("./child", args);
                } else {
                    child_id = id;
                }
                zmq::message_t message;
                main_socket.recv(&message);
                std::string recieved_message(static_cast<char*>(message.data()), mes-
sage.size());
                std::cout << recieved_message << std::endl;
                delete[] adr_;
                delete[] id_;
            } else {
                int id;
                std::cin >> id;
                std::string message_string = command + " " + std::to_string(id);
                zmq::message_t message(message_string.size());
                memcpy(message.data(), message_string.c_str(), message_string.size());
                main_socket.send(message);

                // catch message from new node
                main_socket.recv(&message);
                std::string recieved_message(static_cast<char*>(message.data()), mes-
sage.size());
                std::cout << recieved_message << std::endl;
            }
        } else if (command == "exec") {
            int id;
            std::string big, small;
            std::cin >> id;
            std::cin >> big >> small;
```

```
            std::string message_string = command + " " + std::to_string(id) + " " + big
+ " " + small;
            zmq::message_t message(message_string.size());
            memcpy(message.data(), message_string.c_str(), message_string.size());
            main_socket.send(message);
            // return value from map
            main_socket.recv(&message);
            std::string recieved_message(static_cast<char*>(message.data()), mes-
sage.size());
            std::cout << recieved_message << std::endl;
        } else if (command == "ping") {
            int id;
            std::cin >> id;
            std::string message_string = command + " " + std::to_string(id);
            zmq::message_t message(message_string.size());
            memcpy(message.data(), message_string.c_str(), message_string.size());
            main_socket.send(message);
            // receive answer from child
            main_socket.recv(&message);
            std::string recieved_message(static_cast<char*>(message.data()), mes-
sage.size());
            std::cout << recieved_message << std::endl;
        } else if (command == "kill") {
            int id;
            std::cin >> id;
            if (child_id == 0) {
                std::cout << "Error: there isn't nodes" << std::endl;
            } else if (child_id == id) {
                std::string kill_message = command + " " + std::to_string(id);
                zmq::message_t message(kill_message.size());
                memcpy(message.data(), kill_message.c_str(), kill_message.size());
                main_socket.send(message);
                main_socket.recv(&message);
                std::string received_message(static_cast<char*>(message.data()), mes-
sage.size());
                std::cout << received_message << std::endl;
                std::cout << "Tree deleted successfully" << std::endl;
                return 0;
            } else {
                std::string kill_message = command + " " + std::to_string(id);
                zmq::message_t message(kill_message.size());
                memcpy(message.data(), kill_message.c_str(), kill_message.size());
                main_socket.send(message);
                main_socket.recv(&message);
                std::string received_message(static_cast<char*>(message.data()), mes-
sage.size());
                std::cout << received_message << std::endl;
            }
        } else if (command == "exit") {
            if (child_id) {
```

```
                std::string kill_message = "DIE";
                zmq::message_t message(kill_message.size());
                memcpy(message.data(), kill_message.c_str(), kill_message.size());
                main_socket.send(message);
                std::cout << "Tree was deleted" << std::endl;
            }
            main_socket.close();
            context.close();
            break;
        } else {
            std::cout << "Error: incorrect command\n";
        }
    }
}
```

Пример работы

Test 1

Input	Output
./main	command:
create 1	OK: 3354
command:create 2	OK: 3367
command:create 3	OK: 3377
command:ping 1	OK: 1
command:ping 2	OK: 1
command:exec 2 abcabc a	OK:2:0;3
command:kill 2	Ok: 2
command:ping 2	OK: 0
command:ping 3	OK: 0
command:exec 1 abcabc c	OK:1:2;5
command:exit	Tree was deleted

Вывод

После выполнения данной лабораторной работы можно сделать следующие выводы:

1. ZeroMQ предоставляет эффективный механизм для организации взаимодействия между различными процессами в распределенной системе.

2. Управление созданием и управлением рабочими узлами через команды позволяет эффективно координировать выполнение задач в распределенной среде.

3. Использование разделяемых сокетов ZMQ_REP и ZMQ_REQ обеспечивает надежное и синхронное взаимодействие между основной программой и рабочими узлами.

4. Распределенные системы, построенные на основе ZeroMQ, могут быть масштабированы и обеспечивать высокую отказоустойчивость.

Эта лабораторная работа позволила понять принципы построения распределенных систем с использованием ZeroMQ и оценить их применимость для решения задач с координацией множества взаимодействующих процессов.