

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»**

Темы работы
“Управлении серверами сообщений”
“Применение отложенных вычислений”
“Интеграция программных систем друг с другом”

Студент: Фаттахетдинов Сильвестр
Динарович

Группа: М8О-208Б-20
Вариант: 29

Преподаватель: Миронов Евгений Сергеевич

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2021

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/silverfatt>

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. Список основных поддерживаемых команд:

Создание нового узла

Удаление существующего узла

Выполнение функции

Проверка доступности узлов

Общие сведения о программе

Код работы содержится в двух файлах – `main_prog.cpp` (код для управляющего узла) и `child_node` (код для вычислительного узла). Также для удобства был создан `Makefile`. После компиляции появляются два исполняемых файла – `main_prog` и `child_node`. Для начала работы программы требуется запустить `./main_prog`.

Общий метод и алгоритм решения

1) create id

Вставка нового узла осуществляется по правилам бинарного дерева. Если это первый вычислительный узел – то узел `id` станет корнем этого дерева. Иначе – `id` будет сравниваться со всеми узлами дерева, в зависимости от результатов сравнения будет помещаться в левый или правый сокет (если тот свободен, иначе он опять сравнится, но с потомком). Если встретится узел с таким же `id`, то узел не создастся, а пользователю выведется ошибка.

2) `exec id k n1...nk`

Аналогично предыдущей команде сигнал отправляется вниз по дереву. Если встречается узел с искомым `id` – он считает сумму чисел и возвращает её наверх, иначе пользователю выведется ошибка.

3) `kill id`

Искомый процесс завершает работу. Все его потомки отсекаются от системы, но родительские процессы сохраняют работоспособность. Если искомого процесса нет, пользователю выведется ошибка.

4) `heartbeat time`

В течение $10 * \text{Time}$ миллисекунд все вычислительные узлы каждые `time` миллисекунд сообщают о своей работоспособности.

Исходный код

`main_prog.cpp`

```
#include <zmq.hpp>
#include <unistd.h>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    zmq::context_t context(1);
```

```

zmq::socket_t main_socket(context, ZMQ_REP);
string adr = "tcp://127.0.0.1:3000";
string command;
int child_id = 0;
cout << "Commands:\ncreate id\nexec id k n1...nk\nheartbeat time\nkill
id\nexit\n";
while (1)
{
    cout << "Please, enter command\n";
    cin >> command;
    if (command == "create")
    {
        if (child_id == 0)
        {
            int id;
            cin >> id; //1
            int id_tmp = id - 1;
            main_socket.bind(adr + to_string(++id_tmp));
//tcp://127.0.0.1:3001
            string new_adr = adr + 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[to_string(id).size() + 1];
            memcpy(id_, to_string(id).c_str(), to_string(id).size() + 1);
            char *args[] = {"./child_node", adr_, id_, NULL};
            int id2 = fork();
            if (id2 == -1)
            {
                cout << "ERROR: CALCULATING NODE WAS NOT CREATED\n";
                id = 0;
                exit(1);
            }
            else if (id2 == 0)
            {
                execv("./child_node", args);
            }
            else
            {
                child_id = id;
            }
            zmq::message_t message;
            main_socket.recv(&message);
            string recieved_message(static_cast<char *>(message.data()),
message.size());
            cout << recieved_message << "\n";
            delete[] adr_;
            delete[] id_;
        }
        else
        {

```

```

        int id;
        cin >> id;
        string message_string = command + " " + to_string(id);
        zmq::message_t message(message_string.size());
        memcpy(message.data(), message_string.c_str(),
message_string.size());
        main_socket.send(message);
        main_socket.recv(&message);
        string recieved_message(static_cast<char *>(message.data()),
message.size());
        cout << recieved_message << "\n";
    }
}
else if (command == "exec")
{
    int id, N;
    cin >> id >> N;
    string s;
    vector<int> v(N);
    for (int i = 0; i < N; ++i)
    {
        cin >> v[i];
        s = s + to_string(v[i]) + '$'; //exec 1 3 1 22 3      //1$22$3$
    }
    string message_string = command + " " + to_string(id) + " " + s;
    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(),
message_string.size());
    main_socket.send(message);
    main_socket.recv(&message);
    string recieved_message(static_cast<char *>(message.data()),
message.size());
    cout << recieved_message << "\n";
}
else if (command == "heartbeat")
{
    int TIME;
    const int beat_amount = 1;
    cin >> TIME;
    // "heartbeat 2000"
    for (int j = 0; j < 10; j++)
    {
        string message_string = command + " " + to_string(TIME);
        zmq::message_t message(message_string.size());
        memcpy(message.data(), message_string.c_str(),
message_string.size());
        // receive answer from child
        //for (int j = 0; j < beat_amount; j++)
        //{
            main_socket.send(message);

```

```

        main_socket.recv(&message);
        string recieved_message(static_cast<char *>(message.data()),
message.size());
        if (recieved_message != "OK")
        {
            cout << "Unavailable nodes: ";
        }
        cout << recieved_message << "\n";
        sleep((unsigned)(TIME / 1000));
    }
}
else if (command == "kill")
{
    int id;
    cin >> id;
    if (child_id == 0)
    {
        cout << "Error: there aren't any nodes\n";
    }
    else if (child_id == id)
    {
        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);
        cout << "Tree was deleted\n";
        child_id = 0;
    }
    else
    {
        string kill_message = command + " " + 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);
        string received_message(static_cast<char *>(message.data()),
message.size());
        cout << received_message << "\n";
    }
}
else if (command == "exit")
{
    if (child_id)
    {
        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);
        cout << "Tree was deleted\n";
        child_id = 0;
    }
    main_socket.close();
    context.close();
    return 0;
}
else
{
    cout << "Error: incorrect command\n";
}
}
}

```

child_node.cpp

```

#include <zmq.hpp>
#include <iostream>
#include <unistd.h>
using namespace std;

void send_message(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()
        << "\n";
    }
}

int main(int argc, char *argv[])
{
    string adr = argv[1];
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    main_socket.connect(argv[1]);
    send_message("OK: " + to_string(getpid()), main_socket);
    int id = stoi(argv[2]); //
    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);
    string adr_left = "tcp://127.0.0.1:300";
    zmq::socket_t right_socket(context_r, ZMQ_REP);
}

```



```

string adr_right = "tcp://127.0.0.1:300";
while (1)
{
    zmq::message_t message_main;
    main_socket.recv(&message_main);
    //sleep(1);
    string recieved_message(static_cast<char *>(message_main.data()),
message_main.size());
    string command;
    for (int i = 0; i < recieved_message.size(); ++i)
    {
        if (recieved_message[i] != ' ')
        {
            command += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    if (command == "exec")
    { //-----EXEC-----
        int id_proc;
        string id_proc_, value_;
        string key;
        int sum = 0;
        int pos;
        for (int i = 5; i < recieved_message.size(); ++i)
        { //exec 2 55$2$3$121$
            if (recieved_message[i] != ' ')
            {
                id_proc_ += recieved_message[i];
            }
            else
            {
                pos = i;
                break;
            }
        }
        //0123456
        //exec 2 55$2$3$121$
        id_proc = stoi(id_proc_);
        int sub = 1;
        int number = 0;
        if (id_proc == id)
        { // id == proc_id, this node must execute
            for (int i = recieved_message.size() - 1; i >= pos; --i)
            {
                if (recieved_message[i] == '$' || recieved_message[i] == ' ')
                {

```

```

        sub = 1;

        sum = sum + number;
        number = 0;
    }
    else
    {
        number = number + (recieved_message[i] - '0') * sub;
        sub = sub * 10;
    }
}
string res = "OK:" + id_proc_ + ":" + to_string(sum);
zmq::message_t message(res.size());
memcpy(message.data(), res.c_str(), res.size());
if (!main_socket.send(message))
{
    cout << "Error: can't send message to main node from node
with pid: " << getpid() << "\n";
}
}
else
{ // id != prod_id
    if (id > id_proc)
    { //go to left
        if (left_id == 0)
        { // if node not exists
            string message_string = "Error:id: Not found";
            zmq::message_t message(message_string.size());
            memcpy(message.data(), message_string.c_str(),
message_string.size());
            if (!main_socket.send(message))
            {
                cout << "Error: can't send message to main node from
node with pid: " << getpid() << "\n";
            }
        }
    }
    else
    {
        zmq::message_t message(recieved_message.size());
        memcpy(message.data(), recieved_message.c_str(),
recieved_message.size());
        if (!left_socket.send(message))
        {
            cout << "Error: can't send message to left node from
node with pid: " << getpid() << "\n";
        }
        // catch and send to parent
        if (!left_socket.recv(&message))
        {

```

```

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

```

```

{
    //sleep(2);
    int id_proc; // id of node for creating
    string id_proc_;
    for (int i = 7; i < recieved_message.size(); ++i)
    { //create 1
        if (recieved_message[i] != ' ')
        {
            id_proc_ += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    id_proc = stoi(id_proc_);
    if (id_proc == id)
    {
        send_message("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;
            while (1)
            {
                try
                {
                    right_socket.bind(adr_right +
to_string(++right_id_tmp));
                    break;
                }
                catch (...)
                {
                }
            }
            adr_right += 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[to_string(right_id).size() + 1];
            memcpy(right_id_, to_string(right_id).c_str(),
to_string(right_id).size() + 1);
            char *args[] = {"./child_node", adr_right_, right_id_, NULL};
            int f = fork();
            if (f == 0)
            {
                execv("./child_node", args);
            }
        }
    }
}

```

```

        else if (f == -1)
        {
            cout << "Error in forking in node with pid: " << getpid()
<< "\n";
        }
        else
        {
            zmq::message_t message_from_node;
            if (!right_socket.recv(&message_from_node))
            {
                cout << "Error: can't receive message from right node
in node with pid:" << getpid() << "\n";
            }
            string recieved_message_from_node(static_cast<char
*>(message_from_node.data()), message_from_node.size());
            if (!main_socket.send(message_from_node))
            {
                cout << "Error: can't send message to main node from
node with pid:" << getpid() << "\n";
            }
        }
        delete[] adr_right_;
        delete[] right_id_;
    }
    else
    { // send task to right node
        send_message(recieved_message, right_socket);
        // catch and send to parent
        zmq::message_t message;
        if (!right_socket.recv(&message))
        {
            cout << "Error: can't receive message from left node in
node with pid: " << getpid() << "\n";
        }
        if (!main_socket.send(message))
        {
            cout << "Error: can't send message to main node from node
with pid: " << getpid() << "\n";
        }
    }
}
else
{
    if (left_id == 0)
    { // there is not left node
        left_id = id_proc;
        int left_id_tmp = left_id - 1;
        while (1)
        {
            try

```

```

        {
            left_socket.bind(adr_left +
to_string(++left_id_tmp));
            break;
        }
        catch (...)
        {
        }
    }
    adr_left += 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[to_string(left_id).size() + 1];
    memcpy(left_id_, to_string(left_id).c_str(),
to_string(left_id).size() + 1);
    char *args[] = {"/child_node", adr_left_, left_id_, NULL};
    int f = fork();
    if (f == 0)
    {
        execv("/child_node", args);
    }
    else if (f == -1)
    {
        cout << "Error in forking in node with pid: " << getpid()
<< "\n";
    }
    else
    {
        // catch message from new node
        zmq::message_t message_from_node;
        if (!left_socket.recv(&message_from_node))
        {
            cout << "Error: can't receive message from left node
in node with pid:" << getpid() << "\n";
        }
        string recieved_message_from_node(static_cast<char
*>(message_from_node.data()), message_from_node.size());
        // send message to main node
        if (!main_socket.send(message_from_node))
        {
            cout << "Error: can't send message to main node from
node with pid:" << getpid() << "\n";
        }
    }
    delete[] adr_left_;
    delete[] left_id_;
}
else
{ // send task to left node
    send_message(recieved_message, left_socket);
}

```

```

        // catch and send to parent
        zmq::message_t message;
        if (!left_socket.recv(&message))
        {
            cout << "Error: can't receive message from left node in
node with pid: " << getpid() << "\n";
        }
        if (!main_socket.send(message))
        {
            cout << "Error: can't send message to main node from node
with pid: " << getpid() << "\n";
        }
    }
}
else if (command == "heartbeat")
{
    /*
    1)Получение команды хартбит от родителя
    2)Отправление команды хартбит потомкам
    ----10 раз----
    3)Принятие ответов от потомков
    4)Ответ родителю
    ----10 раз----
    */
    //cout << id << " TEST1\n";
    string timestr;
    for (int i = 10; i < recieved_message.size(); ++i)
    { //heartbeat 1000
        timestr.push_back(recieved_message[i]);
    }
    //cout << "STRING : " << timestr << "\n";
    if (left_id != 0)
        send_message(recieved_message, left_socket);
    //cout << id << " TEST2\n";
    if (right_id != 0)
        send_message(recieved_message, right_socket);
    //cout << id << " TEST3\n";
    int TIME = stoi(timestr);
    for (int i = 0; i < 1; i++)
    {
        //cout << id << " TEST4\n";
        string l = "OK";
        string r = "OK";
        if (left_id != 0)
        {
            zmq::message_t message_left;
            if (left_socket.recv(&message_left))
            {

```

```

        string left(static_cast<char *>(message_left.data()),
message_left.size());
        l = left;
    }
    else
    {
        l = to_string(left_id);
    }
}
//cout << id << " TEST5\n";
if (right_id != 0)
{
    zmq::message_t message_right;
    if (right_socket.recv(&message_right))
    {
        string right(static_cast<char *>(message_right.data()),
message_right.size());
        r = right;
    }
    else
    {
        r = to_string(right_id);
    }
}
//cout << id << " TEST6\n";
if (l == r && l == "OK")
{
    send_message("OK", main_socket);
}
else
{
    if (l != "OK" && r != "OK")
    {
        send_message(l + " " + r, main_socket);
    }
    else if (l != "OK")
    {
        send_message(l, main_socket);
    }
    else
    {
        send_message(r, main_socket);
    }
}
//cout << id << " TEST7\n";
sleep((unsigned)(TIME / 1000));
}
}
else if (command == "kill")
{

```



```

int id_proc; // id of node for killing
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 = stoi(id_proc_);
if (id_proc > id)
{
    if (right_id == 0)
    {
        send_message("Error: there isn't node with this id",
main_socket);
    }
    else
    {
        if (right_id == id_proc)
        {
            send_message("Ok: " + to_string(right_id), main_socket);
            send_message("DIE", right_socket);
            right_socket.unbind(adr_right);
            adr_right = "tcp://127.0.0.1:3000";
            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_message("Error: there isn't node with this id",
main_socket);
    }
    else
    {
        if (left_id == id_proc)

```

```

        {
            send_message("OK: " + to_string(left_id), main_socket);
            send_message("DIE", left_socket);
            left_socket.unbind(adr_left);
            adr_left = "tcp://127.0.0.1:3000";
            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_message("DIE", left_socket);
        left_socket.unbind(adr_left);
        adr_left = "tcp://127.0.0.1:3000";
        left_id = 0;
    }
    if (right_id)
    {
        send_message("DIE", right_socket);
        right_socket.unbind(adr_right);
        adr_right = "tcp://127.0.0.1:3000";
        right_id = 0;
    }
    main_socket.unbind(adr);
    return 0;
}
}
}

```

Демонстрация работы программы

```
silverfatt@DESKTOP-AGNE5GI:~/lab6$ ./main_prog
Commands:
create id
exec id k n1...nk
heartbeat time
kill id
exit
Please, enter command
create 3
OK: 573
Please, enter command
create 4
OK: 580
Please, enter command
create 2
OK: 587
Please, enter command
exec 2 3 1 4 5
OK:2:10
Please, enter command
heartbeat 1000
OK
OK
OK
OK
OK
OK
OK
OK
OK
OK
Please, enter command
create 5
OK: 594
Please, enter command
kill 4
Ok: 4
Please, enter command
kill 5
Error: there isn't node with this id
Please, enter command
exit
Tree was deleted
silverfatt@DESKTOP-AGNE5GI:~/lab6$
```

Выводы

Данная лабораторная работа научила меня пользоваться библиотекой ZMQ, познакомила с такой технологией как очереди сообщений. Благодаря ЛР 6-8 я вновь понял, насколько обширным и интересным является мир программирования. Данная лабораторная работа является достойным завершением курса Операционных Систем.