

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: Прохоров Данила
Михайлович

Группа: М8О-208Б-20
Вариант: 1

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/dmporkhorov>

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 – `control_node.cpp` (код для управляющего узла) и `calculation_node.cpp` (код для вычислительного узла). Также для удобства был создан `Makefile`. После компиляции появляются два исполняемых файла – `server` и `child_node`. Для начала работы программы требуется запустить `./server`.

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. При создании узла с id 0 ошибок не будет, но он будет недоступен из-за особенностей работы программы. В целом можно создавать id с отрицательными узлами, однако иногда при удалении и последующем обращении программа падает с ошибкой.

2) I) `exec id start` – запускает таймер в вычислительном узле с данным id.

II) `exec id stop` – останавливает таймер в вычислительном узле с данным id.

III) `exec id time` – показывает замеренное время в вычислительном узле с данным id.

Аналогично предыдущей команде сигнал отправляется вниз по дереву. Если встречается узел с искомым id – он выполняет одну из 3 команд, иначе пользователю выведется ошибка.

3) `kill id`

Искомый процесс завершает работу. Все его потомки отсекаются от системы и процессы убиваются, но родительские процессы сохраняют работоспособность. Если искомого процесса нет, пользователю выведется ошибка. На месте удалённого узла можно создать новые.

4) `heartbeat time amount`

По сути это не heartbeat, а многократный pingall. В течение `amount*time` миллисекунд все вычислительные узлы каждые `time`

миллисекунд сообщают о своей работоспособности.

Исходный код

control_node.cpp

```
#include
<iostream>

#include <zmq.hpp>
#include <unistd.h>

int main()
{
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REP);
    mainn_socket.setsockopt(ZMQ_RCVTIMEO, 3000);
    std::string adr = "tcp://127.0.0.1:3000";
    std::string command;
    int child_id = 0;
    while(true)
    {
        std::cout << "Enter command\n";
        std::cin >> command;
        if (command == "create")
        {
            if(!child_id)
            {
                int id;
                std::cin >> id;
                int temp = id - 1;
                while(true)
                {
                    try
                    {
                        {
                            main_socket.bind(adr + std::to_string(++temp));
                            break;
                        }
                    }
                    catch(...)
                    {
                        {
                        }
                    }
                }
                std::string new_adr = adr + std::to_string(temp);
                char* address_of_child = new char[new_adr.size() + 1];
                memcpy(address_of_child, new_adr.c_str(), new_adr.size() + 1);
            }
        }
    }
}
```

```

        char* id_of_child = new char[std::to_string(id).size() + 1];
        memcpy(id_of_child, std::to_string(id).c_str(),
std::to_string(id).size() + 1);
        char* arguments[] = {"/child_node", address_of_child,
id_of_child, NULL};
        int process = fork();
        if (process == -1)
        {
            std::cout << "Unable to create first worker node\n";
            id = 0;
            return 1;
        }
        else if (!process)
        {
            execv("/child_node", arguments);
        }
        else
        {
            child_id = id;
        }
        zmq::message_t message;
        main_socket.recv(&message);
        std::string
recieved_message(static_cast<char*>(message.data()), message.size());
        std::cout << recieved_message << "\n";
        delete [] address_of_child;
        delete [] id_of_child;
    }
    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);
        main_socket.recv(&message);
        std::string
recieved_message(static_cast<char*>(message.data()), message.size());
        std::cout << recieved_message << "\n";
    }
}
else if(command == "exec")
{
    int id, value;
    std::string parameter;

```

```

        std::cin >> id >> parameter;
        std::string message_string = command + " " + std::to_string(id) + "
" + parameter;
        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);
        std::string recieved_message(static_cast<char*>(message.data()),
message.size());
        std::cout << recieved_message << "\n";
    }
    else if (command == "heartbeat")
    {
        int time, amount;
        std::cin >> time >> amount;
        for (int j = 0; j < amount; j++)
        {
            std::string message_string = command + " " +
std::to_string(time);
            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);
            std::string
recieved_message(static_cast<char*>(message.data()), message.size());
            if (recieved_message != "OK")
            {
                std::cout << "Unavailable nodes: ";
            }
            std::cout << recieved_message << "\n";
            usleep((unsigned)((unsigned long long)(1000) * time));
        }
    }
    else if (command == "kill")
    {
        int id;
        std::cin >> id;
        if(!child_id)
        {
            std::cout << "Error: there isn't nodes\n";
        }
        else if (child_id == id)
        {
            std::string kill_message = "terminate";
            zmq::message_t message(kill_message.size());

```

```

        memcpy(message.data(), kill_message.c_str(),
kill_message.size());
        main_socket.send(message);
        std::cout << "Tree deleted successfully\n";
        child_id = 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()), message.size());
        std::cout << received_message << "\n";
    }
}
else if(command == "exit")
{
    if (child_id)
    {
        std::string kill_message = "terminate";
        zmq::message_t message(kill_message.size());
        memcpy(message.data(), kill_message.c_str(),
kill_message.size());
        main_socket.send(message);
        std::cout << "Tree deleted successfully\n";
    }
    main_socket.close();
    context.close();
    return 0;
}
else
{
    std::cout << "Error: incorrect command\n";
}
}
}

```

calculation_node.cpp

```

#include
<iostream>

#include <zmq.hpp>
#include <unistd.h>
#include <chrono>

```



```

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[])
{
    std::string adr = argv[1], address_of_left = "tcp://127.0.0.1:300",
address_of_right = "tcp://127.0.0.1:300";
    zmq::context_t parent_context(1), left_context(1), right_context(1);
    zmq::socket_t parent_socket(parent_context, ZMQ_REQ),
left_socket(left_context, ZMQ_REP), right_socket(right_context, ZMQ_REP);
    parent_socket.setsockopt(ZMQSNDTIMEO, 3000);
    parent_socket.setsockopt(ZMQSNDTIMEO, 3000);
    left_socket.setsockopt(ZMQSNDTIMEO, 3000);
    left_socket.setsockopt(ZMQSNDTIMEO, 3000);
    right_socket.setsockopt(ZMQSNDTIMEO, 3000);
    right_socket.setsockopt(ZMQSNDTIMEO, 3000);
    parent_socket.connect(adr);
    send_message("OK: " + std::to_string(getpid()), parent_socket);
    int id = std::stoi(argv[2]), left_id = 0, right_id = 0, time_clock = 0;
    bool measuring = false;
    std::chrono::high_resolution_clock::time_point t1, t2;
    while (true)
    {
        zmq::message_t message_main;
        parent_socket.recv(&message_main);
        std::string recieved_message(static_cast<char*>(message_main.data()),
message_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_of_process;
    std::string process_id, parameter;
    for(int i = 5; i < recieved_message.size(); ++i)
    {
        if (recieved_message[i] != ' ')
        {
            process_id += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    id_of_process = std::stoi(process_id);
    if(id_of_process == id)
    {
        for (int i = 6 + process_id.size(); i <
recieved_message.size(); ++i)
        {
            if(recieved_message[i] != ' ')
            {
                parameter += recieved_message[i];
            }
            else
            {
                break;
            }
        }
        std::string message = "OK:" + std::to_string(id);
        if (parameter == "start")
        {
            t1 = std::chrono::high_resolution_clock::now();
            measuring = true;
            message += ": " + std::to_string(time_clock);
        }
        else if (parameter == "stop")
        {
            if (measuring)
            {
                t2 = std::chrono::high_resolution_clock::now();
                time_clock +=
std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
            }

```

```

        measuring = false;
    }
    else if (parameter == "time")
    {
        message += ": " + std::to_string(time_clock);
    }
    send_message(message, parent_socket);
}
else
{
    if (id > id_of_process)
    {
        if (!left_id)
        {
            std::string message_string = "Error:id: Not found";
            send_message("Error:id: Not found", parent_socket);
        }
        else
        {
            zmq::message_t message(recieved_message.size());
            memcpy(message.data(), recieved_message.c_str(),
recieved_message.size());
            if(!left_socket.send(message))
            {
                std::cout << "Error: can't send message to left
node from node with pid: " << getpid() << "\n";
            }
            if(!left_socket.recv(&message))
            {
                std::cout << "Error: can't receive message from
left node in node with pid: " << getpid() << "\n";
            }
            if(!parent_socket.send(message))
            {
                std::cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
            }
        }
    }
    else
    {
        if (!right_id)
        {
            std::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(!parent_socket.send(message))

```

```

        {
            std::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))
        {
            std::cout << "Error: can't send message to right
node from node with pid: " << getpid() << "\n";
        }
        if (!right_socket.recv(&message))
        {
            std::cout << "Error: can't receive message from
left node in node with pid: " << getpid() << "\n";
        }
        if(!parent_socket.send(message))
        {
            std::cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
        }
    }
}
}
}
else if (command == "create")
{
    int id_of_process;
    std::string process_id;
    for (int i = 7; i < recieved_message.size(); ++i)
    {
        if(recieved_message[i] != ' ')
        {
            process_id += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    id_of_process = std::stoi(process_id);
    if(id_of_process == id)
    {
        send_message("Error: Already exists", parent_socket);
    }
}

```

```

    }
    else if(id_of_process > id)
    {
        if (!right_id)
        {
            right_id = id_of_process;
            int temp = right_id - 1;
            while(true)
            {
                try
                {
                    right_socket.bind(address_of_right +
std::to_string(++temp));
                    break;
                }
                catch(...)
                {
                }
            }
            address_of_right += std::to_string(temp);
            char* right_address = new char[address_of_right.size() +
1];
            memcpy(right_address, address_of_right.c_str(),
address_of_right.size() + 1);
            char* id_of_right = new
char[std::to_string(right_id).size() + 1];
            memcpy(id_of_right, std::to_string(right_id).c_str(),
std::to_string(right_id).size() + 1);
            char* arguments[] = {"/child_node", right_address,
id_of_right, NULL};
            int process = fork();
            if (process == -1)
            {
                std::cout << "Error in forking in node with pid: " <<
getpid() << "\n";
            }
            else if (!process)
            {
                execv("/child_node", arguments);
            }
            else
            {
                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() << "\n";
                }
            }
        }
    }
}

```

```

        std::string
recieved_message_from_node(static_cast<char*>(message_from_node.data()),
message_from_node.size());
        if(!parent_socket.send(message_from_node))
        {
            std::cout << "Error: can't send message to main
node from node with pid:" << getpid() << "\n";
        }
    }
    delete [] right_address;
    delete [] id_of_right;
}
else
{
    send_message(recieved_message, right_socket);
    zmq::message_t message;
    if (!right_socket.recv(&message))
    {
        std::cout << "Error: can't receive message from left
node in node with pid: " << getpid() << "\n";
    }
    if (!parent_socket.send(message))
    {
        std::cout << "Error: can't send message to main node
from node with pid: " << getpid() << "\n";
    }
}
}
else
{
    if (!left_id)
    {
        left_id = id_of_process;
        int temp = left_id - 1;
        while(true)
        {
            try
            {
                left_socket.bind(address_of_left +
std::to_string(++temp));
                break;
            }
            catch(...)
            {
            }
        }
        address_of_left += std::to_string(temp);
        char* left_address = new char[address_of_left.size() + 1];

```

```

        memcpy(left_address, address_of_left.c_str(),
address_of_left.size() + 1);
        char* id_of_left = new char[std::to_string(left_id).size()
+ 1];

        memcpy(id_of_left, std::to_string(left_id).c_str(),
std::to_string(left_id).size() + 1);
        char* arguments[] = {"/child_node", left_address,
id_of_left, NULL};

        int process = fork();
        if (process == -1)
        {
            std::cout << "Error in forking in node with pid: " <<
getpid() << "\n";
        }
        if (!process)
        {
            execv("/child_node", arguments);
        }
        else
        {
            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() << "\n";
            }
            std::string
recieved_message_from_node(static_cast<char*>(message_from_node.data()),
message_from_node.size());
            if(!parent_socket.send(message_from_node))
            {
                std::cout << "Error: can't send message to main
node from node with pid:" << getpid() << "\n";
            }
        }
        delete [] left_address;
        delete [] id_of_left;
    }
    else
    {
        send_message(recieved_message, left_socket);
        zmq::message_t message;
        if (!left_socket.recv(&message))
        {
            std::cout << "Error: can't receive message from left
node in node with pid: " << getpid() << "\n";
        }
        if (!parent_socket.send(message))

```

```

        {
            std::cout << "Error: can't send message to main node
from node with pid: " << getpid() << "\n";
        }
    }
}
}
else if(command == "heartbeat")
{
    std::string timestr;
    for(int i = 10; i<recieved_message.size(); ++i)
    {
        timestr += recieved_message[i];
    }
    if (left_id) send_message(recieved_message, left_socket);
    if (right_id) send_message(recieved_message, right_socket);
    int TIME = std::stoi(timestr);
    std::string l = "OK";
    std::string r = "OK";
    if(left_id)
    {
        zmq::message_t message_left;
        if (left_socket.recv(&message_left))
        {
            std::string left(static_cast<char*>(message_left.data()),
message_left.size());
            l = left;
        }
        else
        {
            l = std::to_string(left_id);
        }
    }
    if(right_id)
    {
        zmq::message_t message_right;
        if (right_socket.recv(&message_right))
        {
            std::string
right(static_cast<char*>(message_right.data()), message_right.size());
            r = right;
        }
        else
        {
            r = std::to_string(right_id);
        }
    }
    if (l == r && l == "OK")

```



```

{
    send_message("OK", parent_socket);
}
else
{
    if (l != "OK" && r != "OK")
    {
        send_message(l + " " + r, parent_socket);
    }
    else if (l != "OK")
    {
        send_message(l, parent_socket);
    }
    else
    {
        send_message(r, parent_socket);
    }
}
usleep((unsigned)((unsigned long long)(1000) * TIME));
}
else if (command == "kill")
{
    int id_of_process;
    std::string process_id;
    for(int i = 5; i < recieved_message.size(); ++i)
    {
        if(recieved_message[i] != ' ')
        {
            process_id += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    id_of_process = std::stoi(process_id);
    if (id_of_process > id)
    {
        if (!right_id)
        {
            send_message("Error: there isn't node with this id",
parent_socket);
        }
        else
        {
            if (right_id == id_of_process)
            {
                send_message("terminate", right_socket);
            }
        }
    }
}

```

```

        send_message("Ok: " + std::to_string(right_id),
parent_socket);

        right_socket.unbind(address_of_right);
        address_of_right = "tcp://127.0.0.1:3000";
        right_id = 0;
    }
    else
    {
        right_socket.send(message_main);
        zmq::message_t message;
        right_socket.recv(&message);
        parent_socket.send(message);
    }
}
}
else if (id_of_process < id)
{
    if(!left_id)
    {
        send_message("Error: there isn't node with this id",
parent_socket);
    }
    else
    {
        if (left_id == id_of_process)
        {
            send_message("terminate", left_socket);
            send_message("OK: " + std::to_string(left_id),
parent_socket);

            left_socket.unbind(address_of_left);
            address_of_left = "tcp://127.0.0.1:3000";
            left_id = 0;
        }
        else
        {
            left_socket.send(message_main);
            zmq::message_t message;
            left_socket.recv(&message);
            parent_socket.send(message);
        }
    }
}
}
else if (command == "terminate")
{
    if (left_id)
    {
        send_message("terminate", left_socket);

```

```

        left_socket.unbind(address_of_left);
        address_of_left = "tcp://127.0.0.1:3000";
        left_id = 0;
    }
    if (right_id)
    {
        send_message("terminate", right_socket);
        right_socket.unbind(address_of_right);
        address_of_right = "tcp://127.0.0.1:3000";
        right_id = 0;
    }

    parent_socket.disconnect(adr);

    return 0;
}
}
}

```

Демонстрация работы программы

```
danila@danila-VirtualBox:~/056/programs$ ./server
Enter command
create 2
OK: 2801
Enter command
create 1
OK: 2808
Enter command
create 4
OK: 2815
Enter command
create 3
OK: 2822
Enter command
create 5
OK: 2829
Enter command
exec 5 start
OK:5: 0
Enter command
exec 2 start
OK:2: 0
Enter command
exec 2 stop
OK:2
Enter command
exec 5 stop
OK:5
Enter command
exec 2 time
OK:2: 5052
Enter command
exec 5 time
OK:5: 20643
Enter command
exec 3 time
OK:3: 0
Enter command
kill 4
Ok: 4
Enter command
exec 5 start
Error:id: Not found
Enter command
create 2
Error: Already exists
Enter command
^C
```

Выводы

Данная лабораторная работа научила меня пользоваться библиотекой ZMQ, познакомила с такой технологией как очереди сообщений. На мой взгляд, самая интересная лабораторная работа, хоть и делать её было убийственно.