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 буде 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.

1. 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.

1. 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.

1. 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.

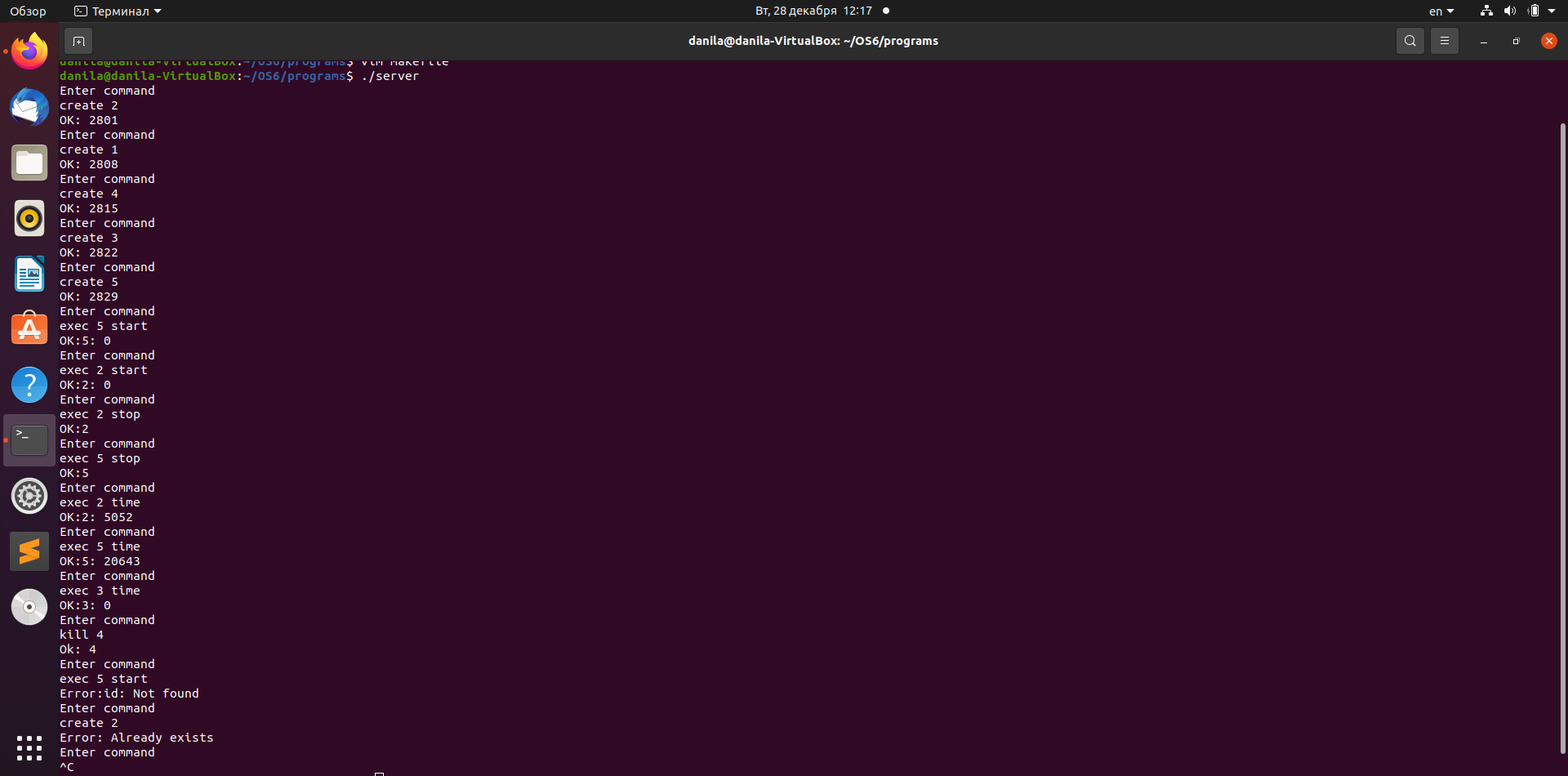
**Исходный код**

**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://127.0.0.1:300"; |
|  | 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://127.0.0.1:300"; |
|  | 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://127.0.0.1:300"; |
|  | 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://127.0.0.1:300"; |
|  | left\_id = 0; |
|  | } |
|  | if (right\_id) |
|  | { |
|  | send\_message("terminate", right\_socket); |
|  | right\_socket.unbind(address\_of\_right); |
|  | address\_of\_right = "tcp://127.0.0.1:300"; |
|  | right\_id = 0; |
|  | } |
|  |  |
|  | parent\_socket.disconnect(adr); |
|  |  |
|  | return 0; |
|  | } |
|  | } |
|  | } |

**Демонстрация работы программы**

**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.